Àlex Sorribes Mulet

ASISTENTE DE VENTAS B2B VÍA WHATSAPP

TRABAJO DE FIN DE GRADO

dirigido por Marc Sànchez Artigas

Grado de Ingeniería Informática

UNIVERSITAT ROVIRA i VIRGILI
Escola Tècnica
Superior d'Enginyeria

Tarragona
2024

Resumen

Este proyecto consiste en el diseño, desarrollo e implementación de un nuevo canal de ventas B2B (Business to Business) en una plataforma de Sales Enablement en producción, propiedad de la empresa CatalogPlayer. Dicho canal permite enviar información de productos directamente al WhatsApp del cliente y recibir los pedidos que este haya realizado desde la aplicación. Durante el desarrollo, se proponen algunos cambios y mejoras a realizar respecto a la forma de desarrollar actual de la empresa CatalogPlayer. Se utiliza Symfony como *framework* de *backend* y se comunica con el resto de módulos y con la plataforma de Meta, propietaria de WhatsApp, vía API REST.

Abstract

This project consists of the design, development, and implementation of a new B2B (Business to Business) sales channel on a Sales Enablement platform in production, owned by the company CatalogPlayer. It allows sending product information directly to the client's WhatsApp and receiving the orders they have made from the application. During the development, some changes and improvements are proposed regarding the current development method of the company CatalogPlayer. Symfony is used as the backend framework and communicates with the other modules and the Meta platform, owner of WhatsApp, via REST API.

Resum

Aquest projecte consisteix en el disseny, desenvolupament i implementació d'un nou canal de vendes B2B (Business to business) a una plataforma de Sales Enablement en producció, propietat de l'empresa CatalogPlayer. Permet enviar informació de productes directament al WhatsApp del client i rebre les comandes que aquest hagi fet des de l'aplicació. Durant el desenvolupament, es proposen alguns canvis i millores a realitzar respecte a la forma de desenvolupar actual de l'empresa CatalogPlayer. S'utilitza Symfony com a *framework* de *backend* i es comunica amb la resta de mòduls i amb la plataforma de Meta, propietària de WhatsApp, via API REST.

Agraïments

Sempre agraït a Albert, qui va confiar en mi per desenvolupar la meva primera aplicació a la meva primera feina com a programador i sense experiència prèvia. Gràcies per creure que tothom mereix una oportunitat.

Agraït també a Jèssica per ensenyar-me les matemàtiques des del punt de vista del perquè en lloc d'ensenyar-me únicament la forma d'obtenir un resultat final.

Gràcies a Jaume i Rosa per ser els meus primers referents universitaris i ajudar-nos sempre en tot el que hem necessitat.

Gràcies al meu tutor de TFG, Marc Sànchez, per guiar-me durant la realització d'aquest projecte i a Arian Ribell per permetre'm dur-lo a terme.

I per acabar, però no menys importants, gràcies a tots els amics i amigues que m'han acompanyat i donat suport durant el camí.

Índice

Índice de figuras, tablas y código

1 Introducción

1.1 Contexto

La empresa CatalogPlayer nace en 2017 con el objetivo de ser una de las mejores plataformas Sales Enablement[1] del mercado. Para cumplir su objetivo, ha diseñado distintos productos que se complementan entre sí. El eje principal es su plataforma de ventas. Un *software* que actúa como CRM[2], Content Hub[3] y ERP[4].

Actúa como CRM al permitir guardar toda la información de los **clientes** y clientes potenciales ***-leads*** en adelante-. Permite gestionar y almacenar toda la información referente a facturación, segmentar a los clientes en grupos específicos y abrir canales de comunicación y venta con dichos clientes.

Actúa como Content Hub al permitir gestionar toda la información referente a categorías, atributos y precios sobre los productos que vende la **empresa** así como sus descripciones, variantes -como el color o talla de una camisa, por ejemplo- y sus fotos y/o documentos.

Actúa como ERP al ofrecer una solución a la empresa para administrar a sus comerciales, planificando sus visitas a clientes.

La plataforma, aparte, ofrece:

- Recomendaciones a *leads* basándose en sus características e historial de compras mediante una Inteligencia Artificial.
- Canal de comunicaciones con el cliente mediante correo electrónico o mensajes directos en la plataforma.
- Extranet donde los clientes tienen un espacio personalizado para hacer pedidos a la empresa.
- Aplicación de ventas compatible con iOS o Android con la que los comerciales de la empresa pueden realizar su trabajo de una forma mucho más cómoda, ágil y sencilla. La propia aplicación permite incluso trabajar *offline*. Ofrece también al comercial la posibilidad de acceder a su agenda y visitar a los clientes *in situ* siguiendo el itinerario de visitas calculado por la plataforma

Todas ellas son funcionalidades muy potentes, pero de poco sirven si no tienen información de la que nutrirse y que poder gestionar. Es por eso que CatalogPlayer ofrece con su plataforma la interoperabilidad con múltiples *softwares* externos, tales como Salesforce, Sage, SAP Business One, Odoo, Microsoft Dynamics, Libra, Prestashop, Woocommerce.

---

[1] El Sales Enablement, o habilitación de ventas, proporciona a los vendedores los argumentos de venta adecuados para mejorar su rendimiento y tener un impacto positivo en la eficiencia comercial [1].

[2] CRM: Customer Relationship Management o Gestión de Relaciones con el Cliente.

[3] Centro de contenido.

[4] ERP: Enterprise Resource Planning o Planificación de Recursos Empresariales.

Permite comunicarse usando cualquier formato y de forma bidireccional. SOAP[5], API[6] REST[7] y JSON[8] son formatos aceptados. Incluso puede conectarse a Dropbox o a un servidor FTP[9] y absorber los CSV[10] que allí se encuentre.

CatalogPlayer ofrece, aparte de la herramienta, una solución de interoperabilidad entre los distintos *softwares* del cliente. Actualmente, cuenta con 149 empresas, más de 10.000 usuarios, tiene registrados más de 1.800.000 productos principales (variantes[11] no incluidas) y gestiona la información de más de 1.400.000 clientes.

1.2 Descripción

La plataforma ya incluye distintos canales de venta propios. Aun así, la interoperabilidad que ofrece CatalogPlayer permite a la empresa vender sus productos y llegar a nuevos clientes usando herramientas externas, con la plataforma como punto de apoyo.

Casi todo el mundo tiene algún correo con el que se registra en distintas plataformas o sitios web, aunque La mayor parte de esa gente no lo consulta habitualmente. Entonces, ¿qué canal de comunicación podríamos usar para vender productos? Un e-commerce[12] soluciona todas las necesidades de una empresa a la hora de comerciar sus productos. El problema está en que es el cliente el que tiene que acceder a la página web de la tienda online.

La forma más rápida de llegar al cliente o *lead* es mediante mensajería instantánea. Y, en ese campo, hay un claro ganador en España. Aquí, Whatsapp es el principal canal de mensajería. En 2023, el 89.7% de los usuarios y usuarias de España utilizaban WhatsApp como canal principal de mensajería instantánea[2]. Este tiene la ventaja de ser una aplicación móvil, con lo que la información a transmitir llegaría directamente al teléfono móvil del cliente. En la mayoría de los casos, el cliente lo verá entre los primeros 10 y 15 minutos. No tardaría más de 30 minutos en consultar esa información, ya que es el tiempo máximo que tardaría en consultar el móvil[3].

WhatsApp ofrece distintas soluciones mediante WhatsApp Business, con el que la empresa puede interactuar con los clientes de forma directa. Además, le permite a la empresa añadir su información, como descripción, ubicación, horarios de atención al cliente, canales de contacto alternativos y algunas opciones de automatización de distintos tipos de mensajes[4].

---

[5] SOAP: Simple Object Access Protocol.
[6] API: Application Programming Interface.
[7] REST: Representational State Transfer.
[8] JSON: JavaScript Object Notation.
[9] FTP: File Transfer Protocol.
[10] CSV: Comma Separated Values.
[11] Un producto puede tener múltiples variantes. Por ejemplo, una misma camisa puede tener múltiples tallas y múltiples colores.
[12] e-commerce: Comercio electrónico.

Al ser una empresa del grupo Meta, permite interactuar con la API de nube[5] de WhatsApp Business mediante la API Graph[6] de Meta. Esta última es una herramienta muy potente para interactuar con todos los productos de Meta.

WhatsApp será, por lo tanto, el nuevo canal de ventas de CatalogPlayer. El reto es, aparte de diseñarlo y crearlo, hacer que funcione con el resto de componentes de la plataforma. También se tendrá en cuenta que, en un futuro, se tendrá que mantener y desarrollar en equipo, con lo que la escalabilidad del código y su facilidad a la hora de desplegar los cambios en producción serán aspectos clave de este proyecto.

1.3 Motivación

Ya en el Ciclo Superior de Desarrollo de Aplicaciones Multiplataforma, entendí que la programación era lo mío. Con 18 años, ya estudiaba y trabajaba en una empresa que me permitía aprender e implementar lo aprendido. Durante esa época, aprendí a distintos niveles el *frontend* (con Vue), el *backend* (con PHP y Laravel) y la gestión de servidores (usando Apache, Linux, Proxmox y MySQL).

A medida que pasaban los años, en la empresa -y en la universidad- empecé a centrarme más en el *backend*, algo de gestión de servidores y también, como añadido, en la gestión del equipo y de las tareas. Para entonces, ya éramos 4 desarrolladores.

Pasados 4 años, dejé el empleo para centrarme en terminar la universidad. Fue en unas prácticas durante el verano en una empresa local que vende viajes y paquetes de esquí cuando aprendí, aunque en un breve período de tiempo, lo complicado que puede llegar a ser el hecho de que muchos programadores manipulen el mismo código y actualicen la plataforma a tiempo real.

Durante todo este trayecto, me han llamado la atención conceptos como arquitectura hexagonal, comunicación por mensajes, sistemas distribuidos y patrones de diseño. La escalabilidad y mantenibilidad del código a lo largo del tiempo son aspectos muy importantes en los proyectos que habitualmente se dejan de lado. Son aspectos que me gusta plantearme, aunque a veces invierta, o pierda, más tiempo del necesario sin llegar a una solución óptima. ¿Existe tal cosa?

Este proyecto no solo será un tutorial de cómo aprovechar la API de Meta para integrar WhatsApp a una plataforma de ventas. Quiero intentar definir, aunque sea un poco, algunas directrices sobre cómo trabajar en un proyecto de *software* en equipo, minimizando todo lo posible las tareas de mantenimiento, despliegue del código y permitiendo al programador tener que hacer únicamente su trabajo: programar.

1.4 Objetivos

Los objetivos principales de este TFG son:

1. Diseñar un nuevo canal de ventas.

2. Diseñar el flujo de información entre los diferentes actores.
3. Diseñar el *onboarding*[13] para añadir esta funcionalidad a una nueva empresa.
4. Desarrollar los diferentes casos de uso dentro del nuevo proyecto y adaptar la infraestructura existente para posibilitar su funcionamiento sin afectar a los otros módulos.
5. Integrar el nuevo módulo a la plataforma.

Objetivos de valor añadido:

1. Diseñar un flujo de trabajo en equipo.
2. Preparar un futuro despliegue a producción mediante CI/CD[14].
3. Agrupar el código común entre módulos en una misma librería de código.

[13] Proceso de integrar un servicio a un ecosistema existente. P.ej. configurar la plataforma para una empresa.

[14] CI/CD: Continuous Integration / Continuous Delivery.

2 Herramientas y Tecnologías Usadas

2.1 Entorno de Desarrollo

2.1.1 Linux

Linux es un sistema operativo de código abierto [7]. Su diseño y desarrollo fue iniciado en 1991 por Linus Torvalds como un proyecto personal. Un sistema operativo es el *software* que gestiona directamente el *hardware* de un sistema y sus recursos, como el procesador, la memoria y el almacenamiento. Se encuentra entre las aplicaciones y el *hardware*, y establece las conexiones entre todos los sistemas de *software* y los recursos físicos que ejecutan las tareas [8].

El hecho de que sea de código abierto comporta que los desarrolladores puedan hacer sus modificaciones para adaptarlo a sus necesidades y hacer aportaciones al proyecto.

Todos los sistemas operativos basados en Linux incluyen el *kernel*, que gestiona los recursos de *hardware*, y un conjunto de paquetes de *software* que conforman el resto del sistema operativo.

Como sistema operativo para el desarrollo se ha usado Ubuntu 22.04.

2.1.2 Docker

Docker es una plataforma de *software* que permite crear, probar e implementar aplicaciones rápidamente. Empaqueta *software* en unidades estandarizadas llamadas **contenedores** que incluyen todo lo necesario para que el *software* se ejecute, incluidas las bibliotecas, herramientas de sistema, código y tiempo de ejecución [9].

Con Docker se puede implementar y ajustar la escala de aplicaciones rápidamente con cualquier entorno con la certeza de saber que el código se ejecutará.

Funciona de forma similar a una máquina virtual, pero con una capa añadida, el Docker Engine, la cual virtualiza el sistema operativo donde se ejecuta la aplicación, del mismo modo que una máquina virtualizaría el *hardware* donde se ejecuta el sistema operativo.

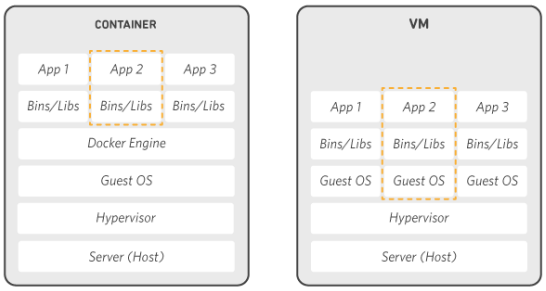

Figura 1. Segmentación en capas de un contenedor y una máquina virtual.

Docker se ha usado en este proyecto, durante el desarrollo, para encapsular el proyecto, el servidor web que procesa las peticiones, el intérprete de PHP, las librerías necesarias y la base de datos. También para su posterior despliegue a producción.

2.1.3 PHP

Es un lenguaje de código abierto muy popular, especialmente adecuado para el desarrollo web, ya que puede ser incrustado en HTML[15] [10]. El código se ejecuta en el servidor, generando el *output* que los navegadores web podrán procesar.

2.1.4 Composer

Es el gestor de dependencias de PHP. Permite declarar las librerías de las que tu proyecto depende, instalarlas y actualizarlas [11].

2.1.5 Nginx

Es un servidor web de código abierto que también puede ser usado como *proxy* inverso, caché de HTTP y como balanceador de carga. Fue creado por Igor Sysoev en 2004 como una respuesta al problema C10K[16]. Destaca por ofrecer un bajo uso de memoria y alta concurrencia.

2.1.6 Symfony

Symfony es un *framework* PHP de tipo *full-stack* construido con varios componentes independientes creados por el proyecto Symfony [13]. Su código, y el de todos los componentes y librerías que incluye, se publican bajo la licencia MIT[17] de *software* libre. En cuanto a términos de seguridad, en 2011 se encargó una auditoría de seguridad a una empresa independiente [14].

2.1.7 MySQL

Es un sistema de gestión de bases de datos relacionales de código abierto. Está respaldado por Oracle y basado en el lenguaje de consulta estructurado SQL[18] [15].

[15] HTML: Hyper Text Markup Language.
[16] Se refiere al problema de rendimiento de manejar 10000 conexiones concurrentes [12].
[17] MIT: Massachusetts Institute of Technology.
[18] SQL: Structured Query Language.

Se basa en un modelo cliente-servidor. El núcleo de MySQL es el servidor MySQL que maneja todas las instrucciones (SQL) de la base de datos. Esas instrucciones se pueden segmentar en los siguientes tipos de sublenguajes [16]:

- Lectura (**DQL** - Data Query Language): nos permite consultar y filtrar los datos.
- Manipulación (**DML** - Data Manipulation Language): nos permite insertar, actualizar o eliminar datos.
- Definición (**DDL** - Data Definition Language): no permite definir la estructura de los datos.
- Control (**DCL** - Data Control Language): nos permite administrar el sistema gestor de bases de datos.

2.1.8 HTTP

HTTP[19] es un protocolo de transmisión de información de la WWW[20]. Con el HTTP, se establecen criterios de sintaxis y semántica informática para el establecimiento de la comunicación entre los diferentes elementos que constituyen la arquitectura web: servidores, clientes y *proxies* [17].

Este lenguaje establece las pautas a seguir, los métodos de petición llamados “verbos” y cuenta con cierta flexibilidad para incorporar nuevas peticiones y funcionalidades. Los métodos implementados habitualmente en un sistema REST son [18]:

- GET: obtener uno o más registros.
- POST: crear uno o más registros. Permite enviar datos.
- PUT: actualizar uno o más registros. Permite enviar datos.
- PATCH: actualizar parcialmente uno o más registros. Permite enviar datos.
- DELETE: eliminar uno o más registros.

Las partes más importantes que lo forman son [19]:

- URL[21]: dirección del recurso con el que interactuar.
- Verbo.
- Cabeceras: estructura clave-valor que puede contener información adicional sobre la petición o respuesta.
- Datos a enviar.
- Código de respuesta.
    - 1XX: Información.
    - 2XX: Éxito.
    - 3XX: Redirección.
    - 4XX: Error en el cliente.
    - 5XX: Error en el servidor.

[19] HTTP: HyperText Transfer Protocol.
[20] WWW: World Wide Web.
[21] URL: Uniform Resource Locator.

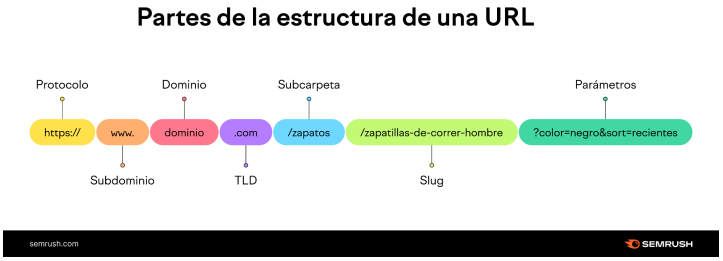

Figura 2. Ilustración de las partes de una URL [20].

En la anterior imagen se aprecian todas las partes de una URL. De ahora en adelante, me referiré a la suma de subcarpeta y *Slug* con el término ***endpoint***.

En el caso de CatalogPlayer y sus conectores, tan solo se usan los verbos GET y POST. Es este último el que también realiza la función de PUT y PATCH. Puede llegar a realizar también la tarea de DELETE, realizando una comprobación entre los registros que existan y los registros que no se envíen.

2.1.9 Webhook

Permiten la comunicación rápida y simple entre diferentes aplicaciones. Ofrece un punto de referencia donde interactuar con una aplicación sin la necesidad de hacer encuestas continuas.

2.2 Entorno de Producción

Para explicar el entorno de producción y el funcionamiento de la solución que ofrece CatalogPlayer de la forma más resumida posible, lo observaremos desde distintos puntos de vista.

Punto de vista del cliente

El cliente o *lead* es un interesado en adquirir los productos que vende la empresa. Para ello, puede establecer comunicación con un comercial de la empresa o interactuar con algún canal de venta de los que disponga la empresa.

Si se comunica con el comercial, es este el que le prepara y tramita el pedido. Si es mediante otro canal de venta de la empresa, es el cliente el que busca qué artículos comprar y se lo comunica a la empresa mediante un pedido. El canal de venta con el que el cliente interactúa puede ser o bien la extranet que ofrece la empresa mediante CatalogPlayer, o bien un servicio externo -como un *e-commerce*- que esté integrado a la empresa gracias a CatalogPlayer.

Punto de vista del comercial

El comercial es un trabajador de la empresa cuya función es gestionar las ventas. Su objetivo principal es vender más y mejor a cuantos clientes pueda y de la forma más rápida y ágil posible.

Para ello, atiende a los clientes en un punto de venta de la empresa -como una tienda- o yendo directamente al domicilio del cliente. Para ambos casos, se tramitará el pedido mediante una *tablet* que contiene una aplicación de CatalogPlayer. Dicha aplicación se comunica con la plataforma de CatalogPlayer y contiene una base de datos con todo el catálogo de la empresa, los clientes, tarifas e itinerario de visitas o tareas del comercial. En caso de que no hubiese conexión con internet, el comercial puede realizar su labor tramitando pedidos y atendiendo a clientes. Cuando se recupere la conexión, ya se sincronizará con la plataforma.

Punto de vista del administrador

El administrador es la persona de la empresa que está al cargo del proceso de ventas. Gracias a distintos servicios, CatalogPlayer entre ellos, gestiona toda la información relativa a productos, tarifas, descuentos, pedidos y facturas, clientes y comerciales. Tiene acceso directo a la plataforma con los permisos más altos que se pueden otorgar sin ser un gestor.

Mediante la plataforma, puede consultar toda la información relativa al proceso de ventas de su empresa gracias a estadísticas, informes y distintas vistas que le ayudan en sus tareas cotidianas. Puede también revisar el estado de los datos sincronizados entre los distintos servicios externos a CatalogPlayer que tiene integrados con la plataforma.

Punto de vista del gestor

El gestor es un representante o trabajador de CatalogPlayer. Es la persona de contacto entre la empresa y CatalogPlayer. Se encarga de entender las distintas necesidades de la empresa con respecto a su proceso de ventas y ofrecerle las soluciones disponibles en CatalogPlayer que necesite. Para ello, documenta todas las tareas a realizar y las comunica a los encargados de cada departamento de CatalogPlayer. Prepara también los servicios externos de la empresa para su integración con la plataforma y configura esta última según las demandas de la empresa.

2.2.1 Plataforma

CatalogPlayer es una plataforma de Sales Enablement que combina la comunicación comercial con la productividad en un entorno de movilidad empresarial con el objetivo de cubrir todo el ciclo comercial de venta antes, durante y después de forma ágil y eficiente. Con este objetivo, ofrece herramientas que permiten resolver los problemas asociados al proceso de venta y comunicación comercial tradicional, un proceso desfasado, ineficiente y costoso: documentación en papel, costosa y no actualizada; imagen corporativa no homogénea; falta de trazabilidad de procesos e históricos de clientes; *reporting* manual, y numerosas duplicidades, y procesos en tiempo no real [21].

Es el producto principal de CatalogPlayer, aunque no se puede definir como un producto individual. Es un conjunto de funcionalidades y servicios agrupados en una única aplicación.

En esta línea, centra su propuesta de valor en tres ejes:

- Integración con los principales ERP y CRM del mercado.
- Aplicación web -*backend* de gestión- para la gestión del ciclo comercial de venta.
- Aplicaciones móviles nativas con soporte *offline* para la presentación de contenidos y la gestión del ciclo comercial de venta.

Hablando en términos funcionales, CatalogPlayer está compuesta de módulos específicos para:

- Generación y presentación de catálogos interactivos.
- Gestión de contenidos a modo de PIM[22].
- Gestión de clientes y planificación de visitas -para clientes consolidados o potenciales- a modo de CRM (Customer Relationship Management).
- Reporte y análisis de la visita, seguimiento de la actividad y generación de indicadores claves de mejora (KPI[23]).
- Gestión de la venta, incluyendo pedidos, ofertas, seguimiento de actividad, etc.

La plataforma tiene como canal de presentación principal las aplicaciones móviles nativas (para dispositivos tipo *tablet*). Sin embargo, también dispone de otros canales de salida que permiten conectar con otras aplicaciones o servicios y presentar o recomendar contenidos.

En su origen, se diseñó para ser compatible con la *suite* de SAGE [22], por lo que todas las importaciones de datos se realizaban mediante absorción de archivos CSV. Cuando se necesitó ser compatible con otros formatos de datos y otros servicios externos, se diseñó la CPAPI y el CGC.

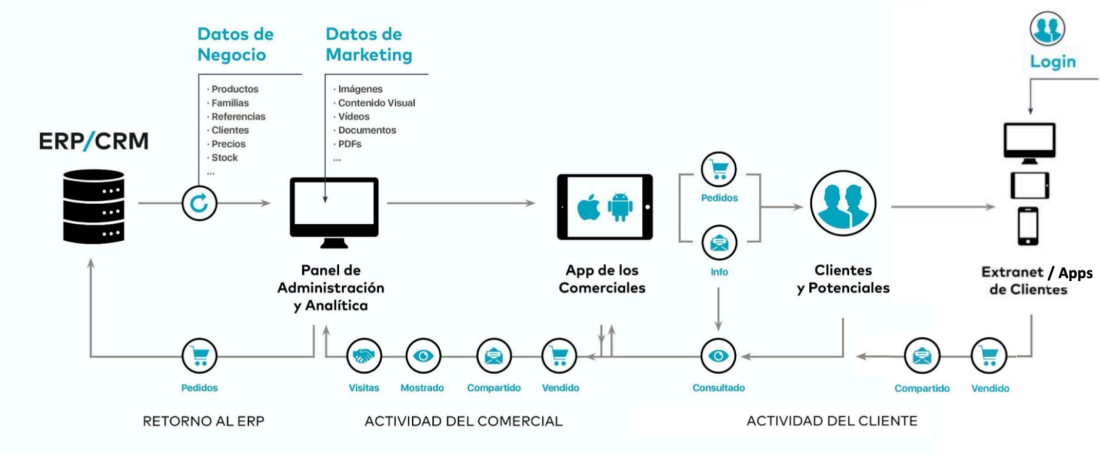

Figura 3. Secuencia del ciclo comercial de datos de módulos principales de CatalogPlayer.

2.2.2 CPAPI

Aplicación web desarrollada sobre Symfony. Su finalidad es hacer de intermediario entre la plataforma y el resto de **conectores**.

[22] PIM: Product Information Management.
[23] KPI: Key Performance Indicator.

Se representó cada entidad que la plataforma puede gestionar -producto, tarifa, cliente, usuario, precio, descuento, factura, pedido, etc.- en un formato al que, a partir de ahora, llamaremos **CPObject**.

Un CPObject es una representación en JSON de dichas entidades. Ese formato se define dentro del marco de JSON-Schema [23]. Dicho formato nos permite concretar los nombres de los campos del objeto, su tipo, su valor por defecto y sus reglas de validación.

La CPAPI transforma de CSV a CPObject cuando se consulta información de la plataforma y de CPObject a CSV cuando se envía la información a la plataforma.

2.2.3 Conector

En CatalogPlayer hay distintos departamentos de desarrollo:

- *Backend*: desarrollo de la plataforma.
- Aplicaciones Android.
- Aplicaciones IOS.
- Conectores.

El departamento de conectores, al cual pertenezco, se encarga de desarrollar y mantener los conectores y el CGC.

Un conector es una aplicación web que procesa peticiones API REST. Existe un conector para cada servicio externo con los que se pretende hacer posible la interoperabilidad con la plataforma -Salesforce, Sage, SAP Business One, Odoo, Microsoft Dynamics, Libra, Prestashop, Woocommerce, etc-. La CPAPI, por ejemplo, es un conector más de los más de 20 que existen actualmente.

Cada conector está preparado para recibir peticiones HTTP GET y POST. Recibe y devuelve los datos en formato CPObject. Internamente, los traduce al formato que necesite el servicio externo -CSV, Json, XML, etc.- y se comunica con él mediante el sistema de comunicación que necesite -API REST, API SOAP, archivo en un FTP, etc.-

En las cabeceras de la petición al conector podemos enviar:

- Fecha de la última sincronización: para filtrar los datos consultando únicamente aquellos cuya fecha de actualización sea superior a la fecha de la última sincronización.
- Configuración: JSON, que contiene los distintos parámetros de configuración que hayamos definido en el código. Nos permite configurar el comportamiento del conector y modificar la forma en la que gestionamos y transformamos los datos para cada petición.
- Credenciales del servicio externo: credenciales con las que comunicarse con el servicio externo. Por ejemplo, URL, usuario y contraseña para un FTP, o URL y llave de acceso para un Prestashop.

2.2.4 CGC

El CGC (CatalogPlayer Global Connector) actúa como **orquestador de conectores**. En él registramos y gestionamos los **conectores**, **aplicaciones** y **conexiones**.

Conector

La información que guardamos de un conector es:

- Identificador, nombre y descripción.
- URL: subdominio de un dominio de CatalogPlayer donde alojamos todos los conectores. Cuando el CGC tenga que comunicarse con el conector le hará una petición HTTP a esta dirección URL.
- Entidades: las entidades (CPObjects) de las que puede consultar o modificar la información. Cada entidad se define por el tipo: si es exportable, si es importable y la ruta específica (*endpoint*) del conector para realizar la petición.
- Acciones: ruta a la que se puede llamar para realizar una acción en concreto sin necesidad de transmitir la respuesta a otro conector.
- Webhooks: contiene un identificador para el *webhook* y la ruta que el CGC debe llamar cuando se quiera ejecutar.
- Configuraciones: conjunto de parámetros que se le pueden indicar para alterar el funcionamiento del conector.
- Credenciales: campos requeridos para interactuar con el servicio externo.

Aplicación

Se entiende como una instancia de un conector para una empresa en concreto.

Contiene:

- El conector que se ha de utilizar.
- Las credenciales con las que conectarse al servicio externo de la empresa.
- Las configuraciones concretas para la aplicación.

Conexión

Vínculo entre dos aplicaciones.

Se puede iniciar una **sincronización** de una entidad en concreto -siempre y cuando sea exportable en el origen e importable en el destino- o ejecutar una acción sobre una aplicación.

Se pueden programar, en formato cron [24], acciones y sincronizaciones de entidades para que se ejecuten de forma periódica.

Sincronización

Acción de consultar todos los datos de una entidad en concreto de una aplicación y enviarlos a otra aplicación para que los procese.

El CGC registra *logs* de cada sincronización que se realice y nos permite consultar el contenido -los datos en formato CPObject- de dichas sincronizaciones. También permite especificar validaciones para cada CPObject.

Es una herramienta que resulta muy útil para depurar fallos entre conectores, ya que actúa, o debería, como fuente de verdad.

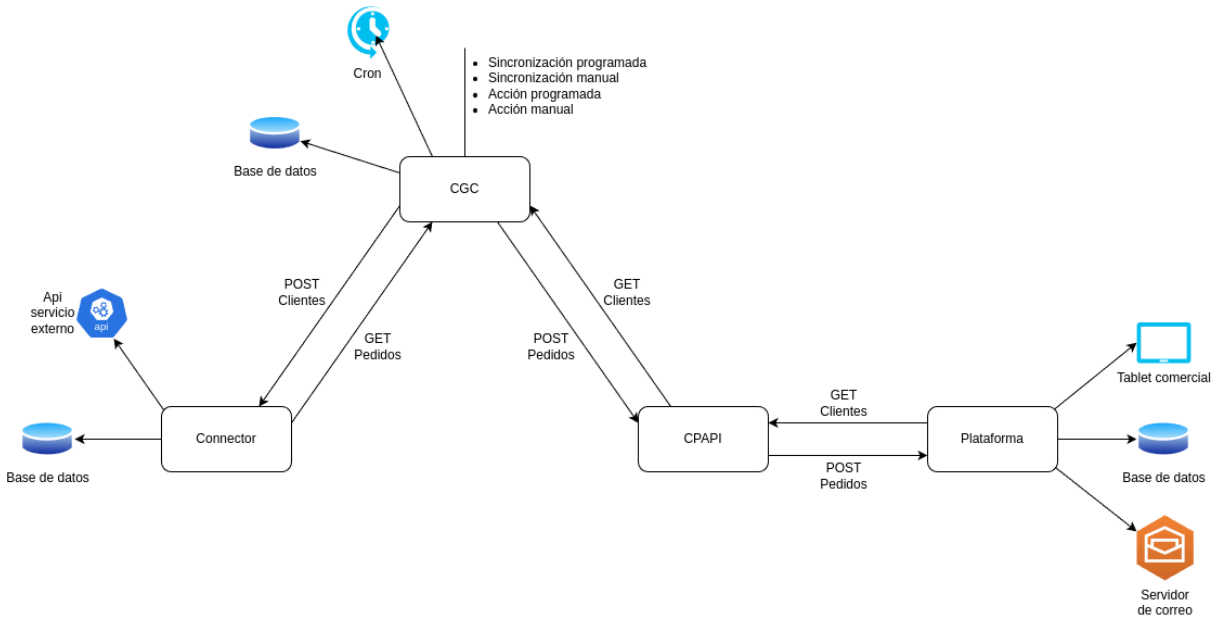

Figura 4. Diagrama que muestra cómo fluyen los datos entre conectores y plataforma.

2.2.5 WhatsApp Business

WhatsApp Business es una aplicación sin costo creada para facilitar la carga de trabajo de las pequeñas y medianas empresas. Tiene como objetivo simplificar la comunicación entre clientes y compañías [25].

Además de las conversaciones en tiempo real, el envío de audios, imágenes y vídeos, WhatsApp Business permite que las organizaciones creen un catálogo de productos y servicios, además de proporcionar información útil a los clientes, como dirección actualizada, formas de pago y horarios de atención.

La API de WhatsApp Business es una herramienta que ayuda a las medianas y grandes empresas a comunicarse con los clientes a través de la plataforma.

2.2.6 Linux

El sistema operativo que ejecutan los servidores es Debian 4. El resto ya se ha explicado en el apartado 2.1.1.

2.2.7 Docker

Ya se ha explicado en el apartado 2.1.2. La única diferencia será la paquetería que se instalará. Xdebug, por ejemplo, no se tiene que instalar en producción. Tampoco se deben instalar las dependencias de desarrollo que estén indicadas en el archivo de *composer*.

2.3 Herramientas de Desarrollo

2.3.1 Xdebug

Xdebug es una extensión que nos permite depurar el código PHP paso a paso [26].

Te permite analizar el rendimiento del código, registra cada llamada que se realiza a una función en el disco junto con sus argumentos, asignaciones de variables y valores de entorno.

2.3.2 PhpStorm

PhpStorm es un IDE[24]; es un editor de código de JetBrains específicamente diseñado para PHP y el desarrollo web.

El hecho de ser un IDE nos aporta [27]:

- Automatización de la edición de código: los lenguajes de programación tienen reglas sobre cómo estructurar instrucciones. Dado que un IDE conoce estas reglas, contiene muchas funciones inteligentes para escribir o editar automáticamente el código fuente.
- Resaltado de sintaxis: un IDE puede dar formato al texto escrito haciendo que algunas palabras aparecen en negrita o itálica, o utilizando diferentes colores de fuente. Estas pistas visuales hacen que el código fuente sea más legible y dan retroalimentación instantánea sobre errores sintácticos accidentales.
- Finalización de código inteligente: varios términos de búsqueda aparecen cuando comienza a escribir palabras en un motor de búsqueda. De manera similar, un IDE puede proponer sugerencias para completar una instrucción de código cuando el desarrollador comienza a escribir.
- Soporte para refactorización.
- Compilación.
- Pruebas.
- Depuración.

2.3.3 Postman

Postman es un *software* que nos permite realizar peticiones HTTP. Podemos crear cuantas peticiones queramos, agruparlas por carpetas, dentro de colecciones y distribuirlas por espacios de trabajo.

En cada petición podemos especificar las cabeceras, el sistema de autenticación, la URL de destino, el tipo de petición HTTP y el contenido de esta. Para cada petición, podemos añadir ejemplos de peticiones y respuestas posibles.

[24] IDE: Integrated Development Environment.

Podemos también crear entornos donde guardamos distintas variables que pueden ser llamadas desde las peticiones, pudiendo cambiar de credenciales o URL de destino, por ejemplo, en un solo clic.

Es una herramienta indispensable para un desarrollador de API. Podemos documentar en el propio *postman* el funcionamiento de estas, publicar colecciones -como la colección de WhatsApp Cloud API [28]- y, en definitiva, estructurar un entorno de trabajo con el que colaborar con los otros desarrolladores del equipo.

3 Requisitos

Los requisitos definen la funcionalidad y el propósito de una pieza particular de *software* [29]. Sirven como lenguaje común entre los miembros del equipo de desarrollo y las partes interesadas -los usuarios finales-. Si están bien definidos al desarrollarse, deberían satisfacer las necesidades del usuario. Nos permiten definir el alcance de un proyecto, identificar riesgos potenciales y proporcionan una base para las pruebas.

Existen de 2 tipos:

- **Requisitos funcionales**: describen las acciones específicas que el *software* debe poder realizar. A menudo se dividen en reglas de negocio o en **casos de uso**.
- **Requisitos no funcionales**: describen características específicas que el *software* debe poseer durante el desarrollo de la aplicación. Por lo general, se dividen en 3 categorías:
    - Rendimiento:
        - Tiempo de respuesta: definen los marcos de tiempo en los que la pieza de *software* debe responder a la petición.
        - Rendimiento: definen el número de peticiones que debe poder tolerar.
    - Seguridad: determinan las medidas que un sistema debe tomar para proteger la información que gestiona.
    - Calidad: agrupan medidas como conformidad, usabilidad, confiabilidad y mantenibilidad.

3.1 Requisitos Funcionales

- **RF1**. Una empresa puede registrar su **número de teléfono de WhatsApp** a nuestra **aplicación de Meta.**
- **RF2**. Una empresa puede suscribir los eventos de su **cuenta de WhatsApp** a la nuestra aplicación de Meta.
- **RF3**. Podemos crear un **catálogo** específico para una empresa en nuestra aplicación de meta.
- **RF4**. Una empresa puede actualizar el catálogo con los productos que tiene en la plataforma.
- **RF5**. El comercial de una empresa puede compartir algunos productos de su catálogo con un cliente o *lead.*
- **RF6**. La empresa puede saber el estado de esa compartición.
- **RF7**. Un cliente o *lead* puede hacer un pedido de algunos de los productos que el comercial le ha compartido.
- **RF8**. La empresa puede recibir los pedidos de los clientes o *leads.*

3.2 Requisitos No Funcionales

- **RNF1**. Debemos asegurarnos que quien nos informa del estado de los mensajes son los servidores de Meta.
- **RNF2**. Cuando compartimos un conjunto de productos con un cliente o l*ead* le debe llegar lo más rápido posible.

3.3 Diagrama de Casos de Uso

A partir de los requisitos especificados en el apartado anterior, se pueden extraer múltiples casos de uso que definen las funcionalidades que nuestro proyecto puede llevar a cabo. Un caso de uso es una abstracción del comportamiento esperado de una porción de código en concreto, pudiendo un caso de uso depender de otros para cumplir su objetivo.

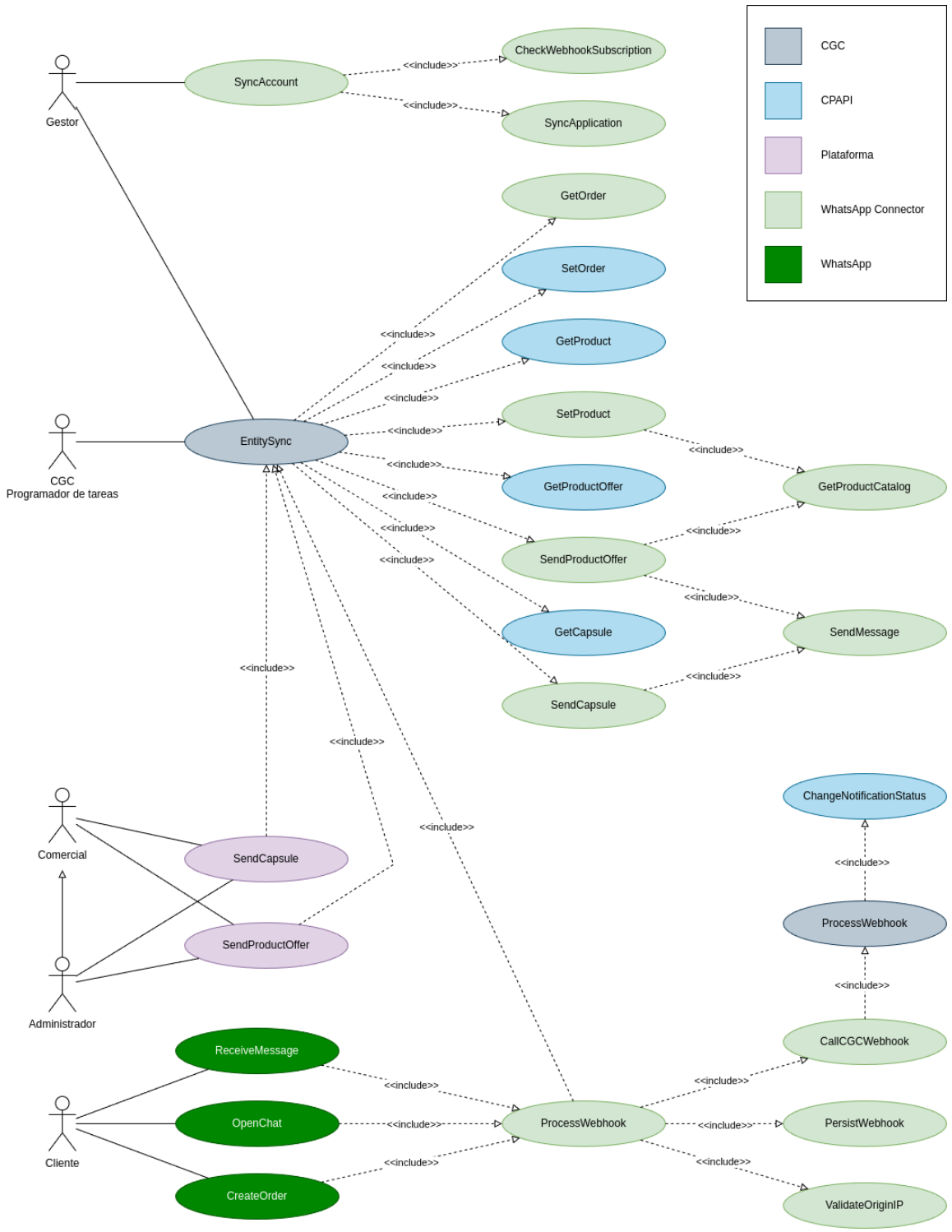

Figura 5. Diagrama de casos de uso.

En el diagrama de casos de uso anterior, diseñado con la herramienta draw.io [30], podemos ver los distintos actores involucrados en todo el proceso de principio a fin. Son los siguientes:

- **Gestor**: interlocutor entre la empresa y CatalogPlayer. Generalmente, trabaja directamente para CatalogPlayer, pero hay algunos casos en los que puede ser un socio externo. Es el encargado, entre otras muchas cosas, de preparar e integrar los datos, activos y servicios externos con la plataforma de CatalogPlayer.
- **CGC**: actúa como orquestador entre los distintos conectores. Permite crear enlaces entre distintos servicios. Pudiendo configurar, en esos enlaces, cada conector para cada cliente y propósito mediante configuraciones que modifican el comportamiento del conector, validar los datos transmitidos y programar dichas sincronizaciones.
  Aparte, sirve como nexo central para depurar fallos gracias a su sistema de *logs* y la opción de iniciar sincronizaciones o acciones de forma manual.
- **Comercial**: trabajador de la empresa a la que CatalogPlayer le provee su plataforma. Su objetivo es vender productos y atender las peticiones y dudas de los clientes. Este solamente interactúa con la plataforma, son la propia plataforma mediante eventos o el CGC, mediante sincronizaciones manuales o programadas, los que transmiten la información o acciones de un punto a otro de toda la infraestructura de la que la empresa dispone.
- **Cliente** o ***lead***: persona interesada en adquirir los productos de la empresa. Puede interactuar con la empresa a través de portales de venta electrónica (*e-commerce*), a través de una extranet de la plataforma o mediante contacto directo con el comercial.
  Cuando el proyecto esté operativo, WhatsApp también será una posible vía de interacción con la empresa.

También podemos ver los distintos casos de uso que, en conjunto, hacen viable WhatsApp como canal de ventas. Antes de enumerarlos, habría que comentar que este proyecto se ha desarrollado sobre un entorno ya existente. Por lo tanto, algunos casos de uso ya existían y funcionaban correctamente, algunos otros se han habido de modificar adaptándose al nuevo conector y la gran mayoría se han diseñado e implementado de 0.

Casos de uso que ya existían según el proyecto donde se ejecutan:

- CGC
  - CDU 04. EntitySync
  - CDU 19. ProcessWebhook
- CPAPI
  - CDU 06. SetOrder
  - CDU 07. GetProduct

Los siguientes también existían, pero no los documentaré, ya que escapan a mi gestión:

- Plataforma
  - CDU 21. SendCapsule
  - CDU 22. SendProductOffer
- WhatsApp
  - CDU 23. ReceiveMessage

- CDU 24. OpenChat
- CDU 25. CreateOrder

3.3.1 CDU 01. SyncAccount

Resumen de la funcionalidad: realiza algunas comprobaciones de la aplicación de WhatsApp de la empresa para asegurarse de que está bien configurada y preparada para ser usada. Aparte, crea un registro en la base de datos propia del conector para establecer una clara relación entre los identificadores de Meta y nuestros identificadores internos.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: estado de la aplicación de Meta y del conector.

Actores: gestor.

Precondición: ninguna.

Postcondición: la aplicación y el conector deben estar preparados para su uso o se debe informar del motivo por el cual no lo están.

Proceso normal principal:

1. Comprobar si las variables de entorno están establecidas.
2. Comprobar que en las cabeceras de la petición están los identificadores de Meta.
3. Ejecutar CDU 02. CheckWebhookSubscription.
4. Ejecutar CDU 03. SyncApplication.

Alternativas de proceso y excepciones:

1. Las variables de entorno no están establecidas o no se puede acceder a ellas.
    a. Informar del error y parar la ejecución.
2. Las cabeceras de la petición solicitadas no están presentes en la petición o están vacías.
    a. Informar del error y parar la ejecución.

3.3.2 CDU 02. CheckWebhookSubscription

Resumen de la funcionalidad: consulta las aplicaciones suscritas al *webhook* de la cuenta de WhatsApp de la empresa. En caso de que la aplicación de CatalogPlayer no se encuentre entre ellas, se suscribe.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: ninguno.

Actores: gestor.

Precondición: ninguna.

Postcondición: la aplicación de CatalogPlayer debe estar suscrita al *webhook* de la aplicación de la empresa.

Proceso normal principal:

1. Consultar las aplicaciones suscritas al *webhook* de la empresa.
2. Buscar la aplicación de CatalogPlayer en las aplicaciones suscritas al *webhook* de la empresa.
    a. Si está suscrita, se termina la ejecución del caso de uso.
    b. Si no está suscrita, se suscribe.

    Alternativas de proceso y excepciones:

2.b No se ha podido suscribir.

a. Informar del error y parar la ejecución.

3.3.3 CDU 03. SyncApplication

Resumen de la funcionalidad: busca en la base de datos, a partir de los identificadores de las cabeceras, el registro que asocia los identificadores de Meta con nuestros identificadores internos. Si no existe el registro, lo crea.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: registro correspondiente a los identificadores de las cabeceras de la petición.

Actores: gestor.

Precondición: ninguna.

Postcondición: debe existir un registro que asocie los identificadores de Meta con los de CatalogPlayer en la base de datos del conector.

Proceso normal principal:

1. Consultar los registros de Application filtrando por los identificadores de CatalogPlayer.
   a. Si existe el registro, actualizarlo con la información de las cabeceras referente a los identificadores de Meta y los de CatalogPlayer.
   b. Si no existe, crearlo con la información del anterior punto.

Alternativas de proceso y excepciones:

1. Existe más de un registro para la misma cabecera de identificador de CatalogPlayer.
   a. Informar del error y parar la ejecución.

3.3.4 CDU 04. EntitySync

Resumen de la funcionalidad: sincroniza una entidad en concreto de una aplicación. Se puede ejecutar manualmente o mediante una sincronización programada. Ya está implementada en CatalogPlayer.

Parámetros de entrada: conexión y entidad a sincronizar.

Parámetros de salida: ninguno.

Actores: CGC, gestor.

Precondición: debe existir, en el CGC, la conexión específica entre 2 aplicaciones.

Postcondición: todos los registros de una misma entidad deben estar sincronizados entre 2 aplicaciones vinculadas por una misma conexión.

Proceso normal principal:

1. El CGC consulta los datos de una entidad sobre una aplicación origen.
2. Registra el *log*.
3. El contenido de la respuesta lo envía a la aplicación destino.
4. Registra el *log*.

3.3.5 CDU 05. GetOrder

Resumen de la funcionalidad: consulta los *webhooks* de la base de datos filtrando por los que sean de tipo Order y estado "Pendiente de procesar". Antes de devolverlos, les cambia el estado a "Procesado".

Parámetros de entrada: identificadores del conector para esta empresa.
Parámetros de salida: pedidos pendientes de procesar para esa empresa.
Actores: CGC, gestor, cliente.
Precondición: ninguna.
Postcondición: todos los pedidos consultados deben tener como estado "Procesado".
Proceso normal principal:

1. Busca el registro en la base de datos que corresponda a los identificadores de la cabecera.
2. Obtiene todos los *webhooks* de tipo Order y con estado "Pendiente de procesar" correspondientes a la aplicación encontrada.
3. Transforma los registros de formato Meta a formato Order de Cpobject.
   Alternativas de proceso y excepciones:
   1. No encuentra ningún registro correspondiente.
      a. Informar del error y parar la ejecución.

3.3.6 CDU 06. SetOrder

Resumen de la funcionalidad: inserta los pedidos en la plataforma de la empresa.
Parámetros de entrada: pedidos.
Parámetros de salida: ninguno.
Actores: CGC, gestor, cliente.
Precondición: ninguna.
Postcondición: los pedidos que se hayan pasado deben estar insertados en la base de datos de la plataforma de la empresa.
Proceso normal principal:

1. Transforma los pedidos en formato CPObject a CSV.
2. Inserta los pedidos en formato CSV a la plataforma.

3.3.7 CDU 07. GetProduct

Resumen de la funcionalidad: consulta todos los productos de la plataforma de la empresa. Se le aplica como filtro los que se hayan actualizado desde la última sincronización de productos.

Parámetros de entrada: fecha de la última sincronización de productos.

Parámetros de salida: productos de la plataforma de la empresa actualizados desde la última sincronización.

Actores: CGC, gestor.

Precondición: ninguna.

Postcondición: ninguna.

Proceso normal principal:

1. Se consultan los productos de la plataforma actualizados desde la última sincronización.
2. Se transforman de formato CSV a formato CPObject.
3. Se devuelven como respuesta a la petición.

3.3.8 CDU 08. SetProduct

Resumen de la funcionalidad: crea o actualiza los productos en los servidores de Meta.

Parámetros de entrada: productos a crear o actualizar.

Parámetros de salida: ninguno.

Actores: CGC, gestor.

Precondición: se debe haber ejecutado con éxito el CDU 01. SyncAccount.

Postcondición: deben existir en Meta los productos a sincronizar.

Proceso normal principal:

1. Se ejecuta el CDU 08. GetProductCatalog indicándole que cree el catálogo en caso de que no exista.
2. Se transforman los productos de formato CPObject a formato Meta.
3. Se crea un BatchRequest de Meta con los productos.
4. Se consulta el estado del BatchRequest hasta que este haya terminado.

3.3.9 CDU 09. GetProductOffer

Resumen de la funcionalidad: consulta las ofertas de producto pendientes de enviar.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: ofertas de productos.

Actores: CGC, gestor, comercial, administrador.

Precondición: ninguna.

Postcondición: las ofertas de productos devueltas por el caso de uso estarán marcadas como enviadas.

Proceso normal principal:

1. Se consultan las ofertas de productos pendientes de enviar.

2. Se transforman de formato CSV a formato CPObject.
3. Se marcan como enviadas.
4. Se devuelven.

3.3.10 CDU 10. SendProductOffer

Resumen de la funcionalidad: envía un mensaje de WhatsApp de tipo "oferta de productos" a un número de teléfono.

Parámetros de entrada: ofertas de producto en formato CPObject.

Parámetros de salida: ninguno.

Actores: CGC, gestor, comercial, administrador.

Precondición: ninguna.

Postcondición: ninguna.

Proceso normal principal:

1. Ejecuta el CDU 13. GetProductCatalog.
2. Transforma las ofertas de productos en formato CPObject a formato Meta indicando el ID del catálogo devuelto por el paso anterior..
3. Ejecuta el CDU 14. SendMessage
4. Registra en los *logs* toda la respuesta de Meta.

3.3.11 CDU 11. GetCapsule

Resumen de la funcionalidad: consulta las cápsulas pendientes de enviar.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: cápsulas.

Actores: CGC, gestor, comercial, administrador.

Precondición: ninguna.

Postcondición: las cápsulas devueltas por el caso de uso estarán marcadas como enviadas.

Proceso normal principal:

1. Se consultan las cápsulas pendientes de enviar.
2. Se transforman de formato CSV a formato CPObject.
3. Se marcan como enviadas
4. Se devuelven.

3.3.12 CDU 12. SendCapsule

Resumen de la funcionalidad: envía un mensaje de WhatsApp de "tipo" con el contenido de la cápsula a un número de teléfono.

Parámetros de entrada: cápsulas en formato CPObject.

Parámetros de salida: ninguno.

Actores: CGC, gestor, comercial, administrador.

Precondición: ninguna.

Postcondición: ninguna.

Proceso normal principal:

1. Transforma las cápsulas en formato CPObject a formato mensaje Meta.

2. Ejecuta el CDU 14. SendMessage.
3. Registra en los logs toda la respuesta de Meta.

3.3.13 CDU 13. GetProductCatalog

Resumen de la funcionalidad: obtiene, de entre todos los catálogos de la aplicación de Meta de CatalogPlayer, el catálogo específico para la empresa. Si no existe, puede crearlo en caso de que así se le indique.

Parámetros de entrada: identificadores del conector para esta empresa.

Parámetros de salida: catálogo de Meta correspondiente a la empresa.

Actores: CGC, gestor, comercial, administrador.

Precondición: se debe haber ejecutado con éxito el CDU 01. SyncAccount.

Postcondición: ninguna.

Proceso normal principal:

1. Consulta, a partir de los identificadores de la cabecera y del archivo de variables de entorno, los catálogos de la aplicación de Meta de CatalogPlayer.
2. Busca un catálogo con nombre igual a “WhatsApp_{ID_EMPRESA}”.
    a. Lo encuentra.
        i. Lo devuelve.
    b. No lo encuentra.
        i. ¿Se ha indicado al caso de uso que lo cree en caso de que no exista?
            1. Sí.
                a. Lo crea.
                b. Lo devuelve.
            2. No.
                a. Informa del error y termina la ejecución.

3.3.14 CDU 14. SendMessage

Resumen de la funcionalidad: envía un mensaje a un número de teléfono.

Parámetros de entrada: identificadores del conector para esta empresa, mensaje y número de teléfono del destinatario.

Parámetros de salida: respuesta de Meta.

Actores: CGC, gestor, comercial, administrador.

Precondición: se debe haber ejecutado con éxito el CDU 01. SyncAccount.

Postcondición: ninguna.

Proceso normal principal:

1. Enviar mensaje.
2. Devolver respuesta.

3.3.15 CDU 15. ProcessWebhook (WhatsApp Connector)

Resumen de la funcionalidad: procesa un *webhook* de Meta.
Parámetros de entrada: *webhook* de Meta.
Parámetros de salida: ninguno.
Actores: cliente.
Precondición: se debe haber ejecutado con éxito el CDU 01. SyncAccount.
Postcondición: ninguna.
Proceso normal principal:

1. ¿Qué tipo de *webhook* es?
    a. Es de tipo order.
        i. Ejecutar CDU 16. PersistWebhook.
        ii. ¿El identificador de la aplicación de WhatsApp Business coincide con algún registro de la base de datos?
            1. Sí.
                a. Iniciar sincronización de la entidad order: ejecutar CDU 04. EntitySync.
    b. Es de tipo mensaje.
        i. ¿El identificador de la aplicación de WhatsApp Business coincide con algún registro de la base de datos?
            1. Sí.
                a. Ejecutar CDU 18. CallCgcWebhook.
    c. Es de otro tipo.
        i. No es una casuística contemplada. No se hace nada.

3.3.16 CDU 16. ValidateOriginIp

Resumen de la funcionalidad: indica si la IP de origen que ha realizado la llamada es una IP de los servidores de Meta.
Parámetros de entrada: IP que ha realizado la llamada.
Parámetros de salida: Booleano:

- Cierto → Es una IP de los servidores de Meta.
- Falso → No es una IP de los servidores de Meta.

Actores: cliente.
Precondición: ninguna.
Postcondición: ninguna.
Proceso normal principal:

1. Obtener un *driver* de caché.
2. Consultar el registro “webhook_valid_origin_ips” en la caché.
    a. ¿Ha caducado?
        i. Sí.
            1. Consultar las IP desde las que Meta nos puede hacer peticiones.
            2. Guardar el listado de IP en un registro de la caché con una caducidad de 1 hora.

3. Buscar la IP que ha realizado la llamada en el listado de IP devuelto por la caché.
    a. Si se encuentra entre ellas devolver “Cierto”.
    b. Si no, devolver “Falso”.

3.3.17 CDU 17. PersistWebhook

Resumen de la funcionalidad: registra en la base de datos el contenido del *webhook.*
Parámetros de entrada: *webhook*.
Parámetros de salida: registro *webhook* de la base de datos.
Actores: cliente.
Precondición: ninguna.
Postcondición: ninguna.
Proceso normal principal:

1. Construir objeto webhook.
2. Buscar aplicación coincidente con el identificador de la aplicación de WhatsApp que ha enviado el *webhook*.
3. Si se encuentra, vincularla al objeto *webhook* a crear.
4. Insertar en la base de datos.
5. Devolver registro *webhook*.

3.3.18 CDU 18. CallCgcWebhook

Resumen de la funcionalidad: realiza una llamada a CGC para que este realice alguna acción.
Parámetros de entrada: identificador del *webhook* a ejecutar, identificador de la aplicación con que ejecutarlo, contenido del *webhook.*
Parámetros de salida: resultado de la llamada.
Actores: cliente.
Precondición: se debe haber ejecutado con éxito el CDU 01. SyncAccount.
Postcondición: ninguna.
Proceso normal principal:

1. Llamar a CGC.
2. Devolver la respuesta.

3.3.19 CDU 19. ProcessWebhook (CGC)

Resumen de la funcionalidad: realiza una llamada *webhook* a una aplicación.
Parámetros de entrada: identificador del *webhook* a ejecutar, identificador de la aplicación con que ejecutarlo, contenido del *webhook*.
Parámetros de salida: resultado de la llamada.
Actores: cliente.
Precondición: ninguna.
Postcondición: ninguna.
Proceso normal principal:

1. Buscar aplicación a partir del identificador de aplicación.
2. Obtener aplicación a la que está vinculada a partir de la relación Conexión.

3. Consultar a partir de la aplicación destino y el identificador de *webhook* la ruta sobre la que realizar la llamada de *webhook*.
4. Realizar la llamada de *webhook* enviando su contenido.
5. Devolver respuesta.

   Alternativas de proceso y excepciones:

   1. No encuentra la aplicación buscada.
      a. Informar del error y parar la ejecución.
   3. No encuentra el *webhook* a partir del identificador de *webhook* y de la aplicación vinculada a la que se ha consultado en el apartado anterior.
      a. Informar del error y parar la ejecución.

3.3.20 CDU 20. ChangeNotificationStatus

Resumen de la funcionalidad: modifica el estado de una cápsula o de una oferta de productos, entre otros tipos de registro admitidos.

Parámetros de entrada: identificadores del conector para esta empresa y contenido del *webhook*.

Parámetros de salida: resultado de la operación.

Actores: cliente.

Precondición: ninguna.

Postcondición: las entidades indicadas en el contenido del *webhook* deberán tener el estado reflejado en la base de datos de la plataforma.

Proceso normal principal:

1. Transformar el contenido del *webhook* a CSV.
2. Realizar la petición correspondiente a la plataforma.
3. Devolver el resultado.

4 Diseño

4.1 Conector

A la hora de diseñar esta nueva funcionalidad, se tiene que tener en cuenta que se desarrollará e integrará en una infraestructura que ya está en producción. En estos casos, debemos observar cómo funcionan los módulos actuales, estudiar cómo cooperan entre sí y decidir si es una buena implementación o no. En caso de que sí lo sea, se puede replicar. En caso de que no, se tendría que cuantificar el coste de diseñar un nuevo paradigma e integrarlo con la infraestructura. Si el coste es aceptable, se puede continuar, previo aviso al equipo de desarrollo y con su autorización. En caso de que no lo sea, la opción más realista es replicar la implementación actual, aunque consideremos que no sea la más adecuada.

En el caso de CatalogPlayer, considero que la infraestructura de conectores está bien diseñada, es modular y permite escalar horizontalmente gracias al CGC. Por lo tanto, todo el código que se comunique directamente con WhatsApp Cloud API estará dentro de una aplicación web con la que el CGC y los servidores de Meta se comunicarán mediante peticiones HTTP. Estas serán de tipo GET si es un *action* o una consulta de información del servicio externo; en este caso, WhatsApp Cloud API. Serán de tipo POST si es un envío de datos a WhatsApp Cloud API para crear o actualizar información, o para que este ejecute una acción. WhastApp nos enviará las notificaciones en forma de *webhook*.

En las cabeceras de las peticiones es donde se indicarán las credenciales a usar, la transacción -metadatos de la sincronización-, las configuraciones específicas para esa llamada, la última fecha de sincronización y algunos más que no son objeto de estudio.

4.2 Librería de Código Común

Durante el desarrollo, se ha analizado cómo funcionan los otros conectores y se ha detectado código duplicado en ciertos servicios esenciales. Estos se agruparán dentro de una misma librería de código.

4.2.1 RequestContext

La mayoría de conectores accedían a las cabeceras de la petición en el controlador. Guardaban su contenido dentro de un *array* y lo inyectaban en cada función como parámetro extra a medida que se necesitaba. Esta dinámica no cumple la "O" de SOLID[25].

Considero que el correcto planteamiento de esta relación de dependencia es aglutinar toda la información que nos llega de la cabecera en una única clase o servicio. Este servicio se debe inyectar como dependencia en cada clase que se necesite.

[25] SOLID: 5 principios básicos en la programación orientada a objetos introducido por Robert C. Martin [31]. La "O" hace referencia al principio "Open/Closed", acuñado por Bertrand Meyer [32], que dice lo siguiente: "Un artefacto de *software* debe estar abierto para su extensión, pero cerrado para su modificación".

Este, aparte, deberá extender una interfaz para cumplir así con el principio de inversión de dependencias -la "D" de SOLID[26]-. Cada conector deberá extender el funcionamiento de esta clase para añadir funcionalidades concretas del conector como variables para guardar las credenciales, así como sus "accesores" y modificadores (*getters* y *setters*).

4.2.2 RequestContextFiller

Esta clase contendrá un único método, *fill*. Su funcionalidad es llamar en cada petición para rellenar la información de RequestContext con los datos de las cabeceras.

La ventaja de implementar esta clase es, aparte de reutilizar su función para cada conector, no haber de modificar cada conector individualmente cuando se añada o modifique algún dato o funcionalidad. Sería suficiente con actualizar la librería.

4.2.3 CurlService

La mayoría de *frameworks* web ya tienen implementados los servicios necesarios para realizar llamadas HTTP. Aun así, en todos los conectores se llama directamente a la librería de PHP para cURL [35] para realizar peticiones HTTP.

Es por ello que se ha creado un envoltorio (*wrapper*[27]) de las llamadas a la librería cURL en forma de servicio, el CurlService.

Este está diseñado para ser compatible entre distintos formatos de comunicación al delegar la tarea de convertir a texto -formato con que se enviará el contenido-, y viceversa, los datos a enviar o recibir. Esa tarea se delega a cualquier clase que implemente la interfaz CurlParser y sea inyectada en el CurlService vía constructor o vía función setParser.

4.2.4 LogService

Igual que en el punto anterior, el servicio de registrar también está presente en todos los *frameworks* web. En este caso, parece adecuada una implementación propia.

Todos los conectores necesitan:

- Crear una carpeta de *logs* específica para cada aplicación, identificada por el subdominio con el que cada empresa accede a su plataforma.
- Crear, dentro de la carpeta de *logs* específica de cada aplicación, otra subcarpeta para cada tipo de sincronización. Esta está identificada por el nombre de la entidad a sincronizar.

[26] La "D" de SOLID hace referencia al principio de inversión de dependencia, el cual nos indica que los sistemas más flexibles son aquellos en los que las dependencias del código fuente se refieren solo a abstracciones, no a concreciones [33].

[27] Un *wrapper* es una función que llama a una o varias funciones, unas veces únicamente por convenio y otras para adaptarlas con el objetivo de hacer una tarea ligeramente diferente [34].

- Los *logs* de tipo *debug* solamente se escriben si hay un "config" en las cabeceras que así lo indique.

Los métodos de registro principales, "info", "error" y "*debug*", siguen la nomenclatura del estándar PSR-3 [36]. Este servicio depende de la interfaz RequestContextualizer y no de la implementación de esta, RequestContext. Se ha diseñado así para poder modificar la implementación a utilizar gracias al Inyector de dependencias de Symfony [**Código 3**].

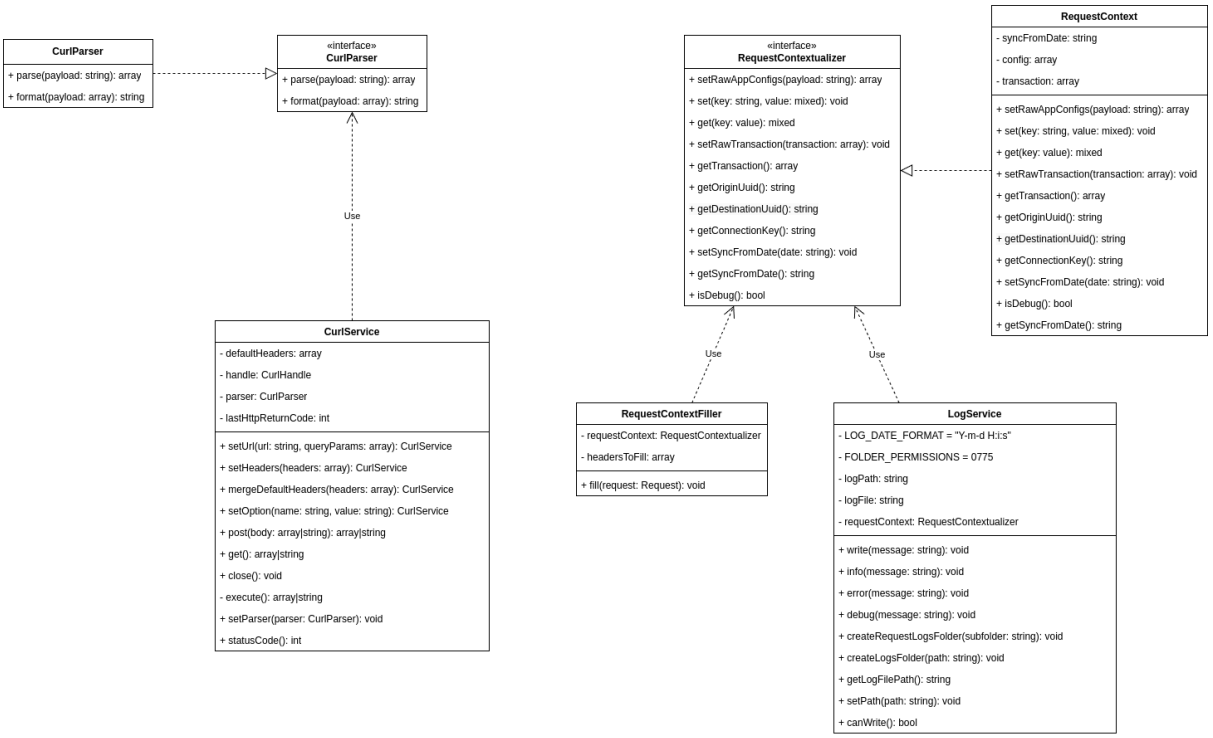

Figura 6. Diagrama UML[28] [37] de los servicios principales de la librería común.

4.3 Base de Datos

Normalmente, en un conector no necesitamos base de datos. Su finalidad es servir peticiones traduciéndolas en consultas a servicios externos. No suelen necesitar persistencia de datos. En este caso, aunque sea poco, se necesita un mínimo soporte de datos para poder guardar los *webhooks* que Meta nos envía y procesarlos más adelante. Pudiendo incluso procesarlo más de una vez en caso de que el destino final (la CPAPI) nos indique que ha habido algún error durante la transmisión o con el formato de datos. También es necesario saber a qué aplicación corresponde cada *webhook*. Necesitamos poder guardar y consultar la siguiente información:

4.3.1 Application

Representación de la aplicación definida en el punto 2.2.4 CGC. Se vincularán al registro de aplicación los *webhooks* que le correspondan.

[28] UML: Unified Modeling Language.

Campos

Aparte del identificador del registro, para poder crear una relación con *webhook*, necesitamos:

- **Identificador de la cuenta de WhatsApp Business**: para, cuando recibamos un *webhook* de Meta, saber a qué aplicación corresponde, junto con el identificador del número de teléfono.
- **Identificador del número de teléfono**: mismo motivo que el campo anterior.
- **Identificador de la aplicación**: para, cuando se realice una sincronización de pedidos pendientes de procesar, saber qué registros de *webhook* consultar.
- **Identificador de la conexión**: para cuando hagamos una llamada al CGC e iniciar la sincronización de pedidos pendientes a procesar.
- **Fecha de creación y actualización.**

4.3.2 Webhook

Guardará la información referente a los *webhooks* que Meta nos envíe al conector. De momento, tan solo se guardarán los referentes a un pedido de productos. En un futuro próximo, también los mensajes del cliente hacia la empresa y se podrán añadir más casuísticas.

Campos

- **Identificador de la aplicación**: para saber a qué aplicación corresponde. Puede darse el caso de que la aplicación no esté registrada previamente. Es por ello que la base de datos debe tolerar un registro de *webhook* con este identificador vacío.
- **Tipo de *webhook***: para categorizar la información que contiene.
- **Estado**: para saber si se está pendiente de procesar, se ha procesado o ha habido algún error de procesamiento.
- **Carga útil**: contenido del *webhook*.
- **Fecha de creación y actualización.**

Los anteriores requisitos se pueden representar con el siguiente diagrama:

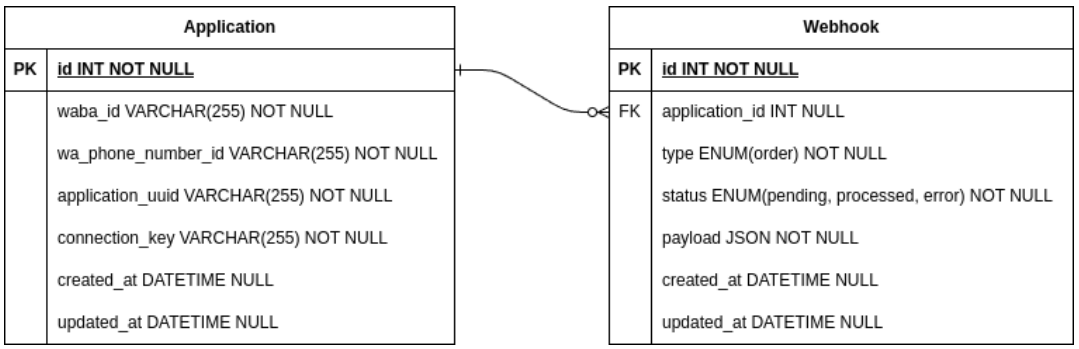

Figura 7. Diagrama entidad / relación de la base de datos del conector.

4.4 CPObject

La plataforma ya tenía las estructuras de datos de Capsule y ProductOffer definidas. Esas funcionalidades existen desde hace tiempo para el uso interno de la plataforma. Con el nuevo canal de ventas resultante de este proyecto, se han de definir esos datos en formato CPObject para su posible transmisión entre conectores.

Para ello, como ya se ha comentado anteriormente, se usará JSON Schema para definir y poder validar la estructura.

El CPObject resultante es el siguiente:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Capsule",
  "$id": "capsule.json",
  "description": "A catalog for specific user",
  "type": "object",
  "properties": {
    "Token": {
      "description": "The unique identifier for a capsule",
      "type": "string"
    },
    "Type": {
      "description": "Type of the capsule",
      "type": "enum",
      "enum": ["MICROSITE_CATALOG", "PRODUCT_OFFER"]
    },
    "Sellable": {
      "description": "If the capsule is sellable",
      "type": "boolean"
    },
    "Url": {
      "description": "The URL of the microsite",
      "type": "string"
    },
    "ImageThumb": {
      "description": "The URL of the thumbnail image",
      "type": "string"
    },
    "Contact": {
      "description": "The receptor of the capsule",
      "type": "object",
      "$ref": "contact.json"
    },
    "Customer": {
      "description": "The customer where the capsule is assigned to",
      "type": "object",
      "$ref": "customer.json"
    },
    "Language": {
      "description": "Language of the capsule",
      "type": "object",
```

```
            "$ref": "language.json"
          },
          "Header": {
            "description": "The title of the capsule",
            "type": "string"
          },
          "Body": {
            "description": "Text of the capsule",
            "type": "string"
          },
          "Footer": {
            "description": "The footer of the capsule",
            "type": "string"
          },
          "Products": {
                 "description": "The products of the capsule (only for
PRODUCT_OFFER type)",
            "type": "array",
            "items": {
              "type": "object",
              "$ref": "product.json"
            }
          },
          "CreatedAt": {
            "description": "The date and time when the capsule was created",
            "type": "string",
            "format": "date-time"
          },
          "UpdatedAt": {
              "description": "The date and time when the capsule was last
updated",
            "type": "string",
            "format": "date-time"
          }
        }
      }
```

Código 1. Representación en JSON Schema de la entidad Capsule.

4.5 Flujo de Trabajo en Equipo con Git

Debemos poder trabajar múltiples desarrolladores a la vez con el mismo proyecto y al mismo tiempo. Esto implica que, en ciertas ocasiones, nuestros cambios colisionen con los cambios de algún compañero. Para ello, entre otros motivos, usamos sistemas de control de versiones. Pero ¿cómo los usamos? ¿Cómo trabajamos con ellos?

Git es uno de los más usados. Los conceptos más básicos para entender su funcionamiento son:

- **Commit**: marca de tiempo que referencia un conjunto de cambios sobre un directorio de archivos. En esta marca, identificada por un *hash* único, tenemos la información del autor, fecha y hora, líneas eliminadas y líneas añadidas. Las modificaciones se representan con una eliminación de la versión anterior y una adición de la línea actual.

- **Rama**: conjunto de *commits* que tienen como origen un punto inicial -un commit-. Se puede crear una rama a partir de otra para realizar cambios que no se reflejarán en la original.
- **Fusión de ramas**: fusiona todos los *commits* de 2 ramas en una sola -la de destino-. En caso de haberse modificado la misma línea en las 2 ramas, se considera conflicto y se tendrá que resolver manualmente. La rama origen se quedará como estaba y la de destino contendrá todos los *commits* + 1 de fusión.
- **Repositorio**: es el conjunto de *commits* y ramas que forman el seguimiento de cambios de un proyecto. Físicamente, es una carpeta que guarda toda la información necesaria. El repositorio local se puede sincronizar con uno remoto -normalmente, alojado en un servidor- para que otros desarrolladores se sincronicen y lo actualicen con nuevos cambios [38].

El funcionamiento de git es simple y potente a la par. Pero hay muchas formas de trabajar con él. El flujo de trabajo (Gitflow) que usemos determinará las limitaciones que tendremos y las capacidades de que dispondremos como proyecto y equipo.

El mismo sistema gestor de control de versiones ya conlleva, por definición, un **historial de cambios de código** y la posibilidad de revisar cambios anteriores o hacer modificaciones en el actual permitiendo al resto de compañeros trabajar en una **línea de cambios** distinta a la tuya. Hay otras funcionalidades que podemos llevar a cabo usándolo de forma correcta.

- **Cooperación con otros desarrolladores**. Al actualizar nuestro repositorio local con los cambios del remoto, podemos deshacer nuestros cambios por los de otros desarrolladores. Es importante hacer un *commit* por cada tarea para evitar tener problemas en nuestro repositorio local y no subir cambios inacabados para prevenir que otros desarrolladores los tengan.
- **Entornos aislados**. Los cambios que se realizan para una tarea no tienen por qué dificultar el desarrollo de los cambios para otra tarea. Si usamos el sistema de ramificación de git, podemos tener una rama por cada tarea. También tenerla por cada entorno, pudiendo desplegar en un servidor un entorno de producción, de *staging* y/o de desarrollo; cada uno, con un código completamente diferente.
- **Trazabilidad de cambios**. Tenemos que poder asociar cada cambio con un *ticket* para obtener toda la información del mismo. Si no creamos esta asociación, dificultamos entender el porqué de esos cambios. Si tenemos múltiples *commits* referenciando un mismo *ticket,* nos creamos más trabajo para revisar los cambios efectuados para una misma tarea. Si creamos una rama por cada tarea, podremos localizarlo fácilmente con la plataforma que gestione el repositorio remoto.

Opción A

Realizamos los cambios en nuestro proyecto local.

Vamos programando en nuestro proyecto local y, al final de nuestra jornada, subimos los cambios al repositorio, aunque no estén acabados.

Desventajas:

- Cooperación con otros desarrolladores: nuestros cambios pueden sobreescribir los de nuestros compañeros y viceversa.
- Entorno aislado: no podemos cambiar de tarea hasta que no terminemos la actual.
- Trazabilidad de cambios: múltiples *commits* para hacer el seguimiento de una tarea.

Opción B

Realizamos un *commit* por cada tarea en la rama principal y lo subimos al terminar el desarrollo.

Ventajas:

- Cooperación con otros desarrolladores: nuestros cambios no sobreescribirán los de nuestros compañeros, a menos que así lo queramos.
- Trazabilidad de cambios: un único *commit* para hacer el seguimiento de una tarea facilita su seguimiento.

Desventajas:

- Entorno aislado: no podemos cambiar de tarea hasta que no terminemos la actual.

Opción C

Por cada tarea a realizar, creamos una rama. En esa rama, creamos un único *commit* y lo sobreescribimos con los cambios que vayamos realizando. Al final de nuestra jornada, podemos subir los cambios al repositorio remoto sin afectar a nadie. Si nos llega una tarea más urgente, podemos cambiar de rama sin preocuparnos de los cambios a medias.

Ventajas:

- Cooperación con otros desarrolladores: ni molestamos ni nos molestan. Todos los cambios de cada tarea están en ramas distintas hasta que son aceptados y pasan a la rama principal.
- Entorno aislado: podemos cambiar de tarea libremente sin preocuparnos. Si queremos probar los cambios, tan solo tenemos que seleccionar la rama de la tarea.
- Trazabilidad de cambios: cada tarea tiene, mientras se desarrolla, su propia rama. Al fusionar las ramas, la podremos localizar con un único *commit*.

Opción	Cooperación con otros desarrolladores	Entorno aislado	Trazabilidad de cambios
A	❌	❌	❌
B	✅	❌	✅
C	✅	✅	✅

Tabla 1. Comparación entre distintos flujos de trabajo con git.

Figura 8. Diagrama de ejemplo de un flujo de trabajo con git. Diseñado con mermaid [39].

4.6 Concesión de Permisos a Recursos de Meta

4.6.1 Autorización

Meta utiliza Oauth para conceder a terceros los permisos necesarios para gestionar los recursos de sus clientes. El *framework* de **autorización** Oauth 2.0 es un protocolo que permite a un usuario otorgar a un tercero acceso a sus recursos protegidos sin tener que revelar sus credenciales o, incluso, su identificación [40].

Para conseguir su cometido, Oauth actúa como intermediario entre los recursos del servidor -en este caso, la empresa- y el actor que quiere acceder a ellos -en este caso, el conector de WhatsApp de CatalogPlayer-. Se encarga de, una vez el cliente se ha **autenticado**, proveerle de un *token* de acceso. Oauth posteriormente podrá, con ese *token*, determinar el cliente al que corresponde, los recursos a los que se le han permitido acceder y con qué permisos.

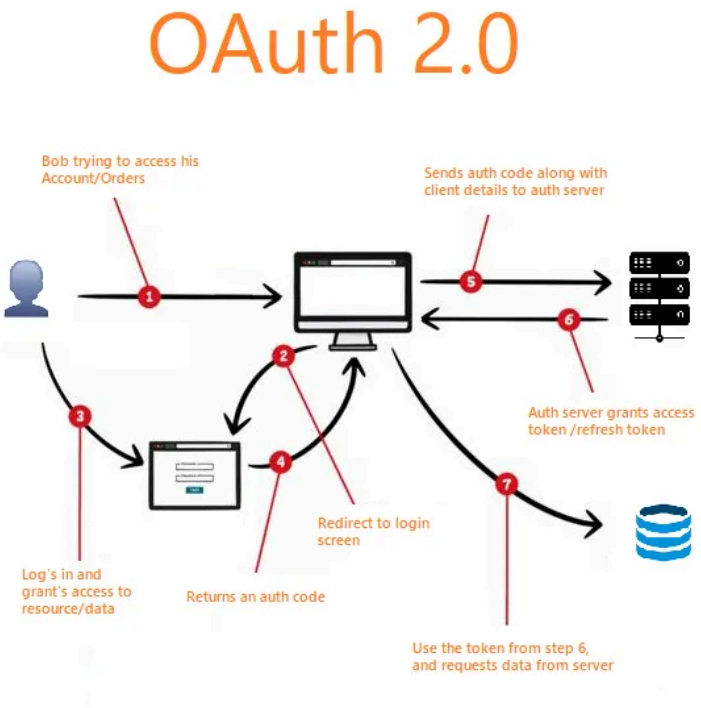

Figura 9. Diagrama de flujo de Oauth [41].

4.6.2 Concesión de permisos por parte de la empresa a CatalogPlayer

Para poder gestionar los recursos de Meta de la empresa, estos deben haber seleccionado CatalogPlayer como proveedor de tecnología y haber proporcionado a CatalogPlayer los permisos necesarios. El primer requisito, lo tiene que llevar a cabo la empresa desde su panel de control de Meta. Para el segundo requisito, se ha creado una pequeña página web en el conector donde la empresa puede hacer *login* con la cuenta de Facebook vinculada a WhatsApp Business. Una vez se ha autenticado, la empresa podrá conceder permisos sobre sus recursos a CatalogPlayer y devolver un Token de acceso que CatalogPlayer se guardará.

Esa página de login es un SDK proveído por Meta. Se le pasan 2 variables desde el conector.

- Identificador de la aplicación de Meta de CatalogPlayer. Necesaria para saber desde qué aplicación se gestionarán los recursos delegados.
- Identificador de la configuración del *login* de Facebook. Previamente, se ha creado, desde el panel de gestión de CatalogPlayer en Meta, una configuración para el *login* de Facebook donde se han especificado los permisos que solicitaremos en ese *login*.

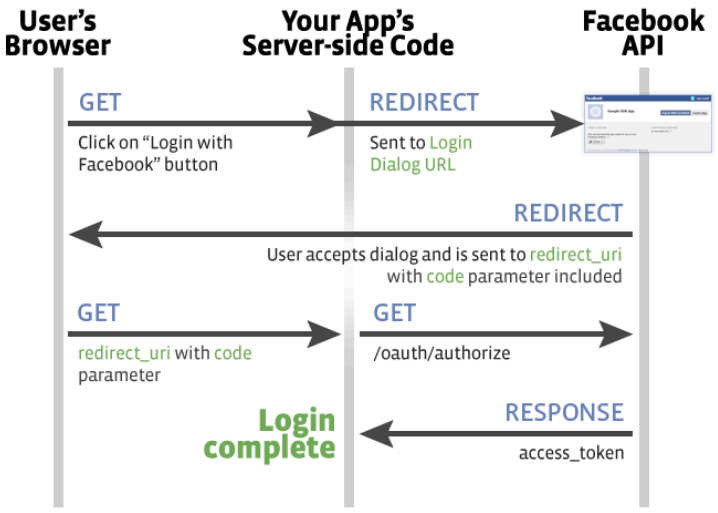

Figura 10. Ejemplo de autorización mediante Oauth y Facebook[42].

Cuando el cliente haga clic en el botón de *login,* se le abrirá una ventana nueva con los formularios necesarios para proveer a CatalogPlayer el acceso y los permisos indicados en la configuración del *login* de Facebook. El cliente será guiado por distintos pasos con formularios para:

1. Iniciar sesión con su cuenta de Facebook donde tiene los recursos de la empresa.
2. Seleccionar el negocio vinculado a su cuenta de Facebook y configurar parámetros, como el nombre visible de la empresa o dirección.

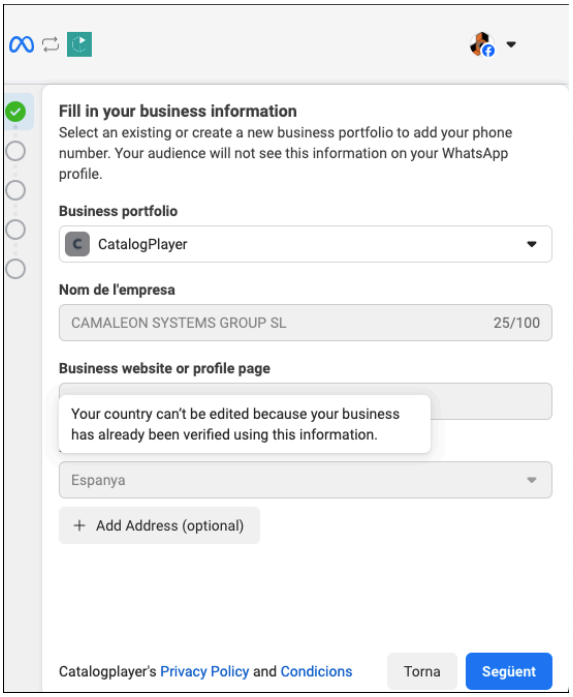

Figura 11. Selección del negocio por parte de la empresa.

3. Seleccionar WhatsApp Business Profile vinculado a su negocio.

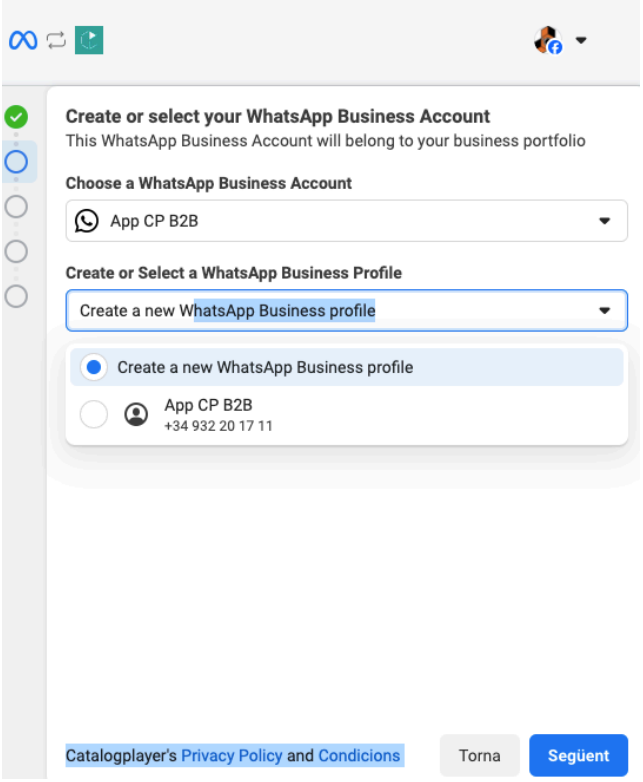

Figura 12. Selección de la cuenta de WhatsApp Business por parte de la empresa.

4. Seleccionar el número de teléfono vinculado a su cuenta de WhatsApp Business y validar el número de teléfono mediante un código que se le enviará vía SMS o llamada.

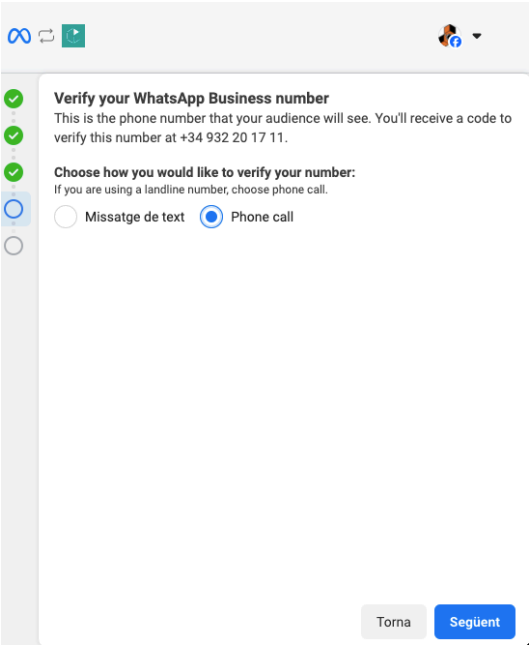

Figura 13. Validación del número de teléfono de WhatsApp por parte de la empresa.

5. Enviar toda la información solicitada a los servidores de Meta. Meta asignará los recursos solicitados a CatalogPlayer, creará un *token* de acceso y lo devolverá como respuesta.

Ese *token* de acceso es el que Meta devuelve como respuesta al *login* que ha realizado el cliente y proporciona los permisos solicitados. Llegados a este punto, Meta ya ha asignado, internamente, los recursos de la empresa a la aplicación de CatalogPlayer con los permisos de la configuración del *login* de Facebook.

5 Implementación

5.1 Conector

Como punto de partida sobre el que desarrollar todo el código del conector, se ha utilizado el *framework* Symfony en su versión 6.4 LTS[29] [43]. Es muy importante escoger las versiones LTS del *software* a implementar en producción para asegurarnos soporte durante años. Esta versión en concreto salió en noviembre de 2023 y, al ser una versión LTS, tendrá actualizaciones durante 3 años, hasta noviembre de 2026, y actualizaciones de seguridad durante un año más.

Para ejecutar el código, se utiliza PHP 8.2 [44]. En este caso hay, a fecha de redacción de este documento, menos margen de soporte de actualizaciones. PHP ofrece actualizaciones durante 2 años para cada versión. Y ofrece 2 años más de soporte de seguridad. Este último período dura hasta el año 2027.

5.1.1 Librería código común

Todos los conectores están preparados para ser llamados mediante peticiones HTTP [2.2.3]. Estas peticiones contienen, en sus cabeceras, la información del contexto de cada llamada. Esa información debe leerse, formatearse y disponerla para todo el código que la requiera durante la ejecución. Para ese fin, se ha desarrollado el siguiente código apoyándonos en el *framework* de Symfony y la librería de código común.

5.1.1.1 RequestContext

Se ha creado el servicio RequestContext que se extiende del RequestContext de la librería de código común. Esta implementación propia de este conector contiene, como variables de instancia, las credenciales necesarias para realizar llamadas al servicio de Meta:

- `private string $bearerToken`: *token* de acceso a Meta. Corresponde a un usuario de sistema de la aplicación de CatalogPlayer en Meta.
- `private string $businessId`: identificador del negocio de CatalogPlayer en Meta.
- `private string $applicationSecret`: clave de acceso secreta de la aplicación de CatalogPlayer en Meta.
- `private string $whatsappBusinessAccountId`: identificador de la aplicación de WhatsApp de la empresa.
- `private string $whatsappPhoneNumberId`: identificador del número de teléfono de WhatsApp de la empresa.

Para todas las variables anteriores se han implementado sus correspondientes *getters* y *setters*, funciones necesarias para leer y modificar variables a las que no se tienen -o no se debería tener- acceso directo desde fuera del servicio.

[29] LTS: Long-Term Support.

También se ha creado la variable `$connectionKey`. En el RequestContext de la librería de código común, existe un *getter* para este dato, pero no una variable concreta que la contenga. Ese RequestContext consulta el dato a partir de la información de la transacción. Aunque esto supone un problema para cuando sea Meta el que realice la petición en forma de *webhook*. Cuando lo haga, deberemos relacionar la petición de Meta a una aplicación de CatalogPlayer. Gracias a esa relación, sabremos qué connectorKey corresponde a la empresa y podremos iniciar una sincronización de pedidos.

En el RequestContext del conector se ha sobreescrito el método getConnectionKey del RequestContext de la librería para consultar primero la variable de instancia antes que llamar al getConnectionKey de la librería que consultaría la *variable transaction*.

```
public function getConnectionKey(): ?string
{
	return $this->connectionKey ?: parent::getConnectionKey();
}
```

Código 2. Sobreescritura del método getConnectionKey.

Para establecer el RequestContext del conector como la implementación a usar en lugar del RequestContext de la librería común, hay que configurar el *framework* para ello. Se ha de registrar el RequestContext del conector como un servicio y hay que informar al *framework* de que, cuando se solicite en un constructor una instancia de la interfaz RequestContextualizer, el inyector de dependencias de servicios de Symfony (*autowiring* [46]) provea al constructor una instancia del RequestContext del conector en lugar del RequestContext de la librería de código común.

```
# Archivo config/services.yaml
…
App\Service\RequestContext: ~
CPConnector\CommonBundle\Service\RequestContext\RequestContextualizer:
'@App\Service\RequestContext'
```

Código 3. Configuración del autowiring de Symfony

5.1.1.2 RequestReceivedSubscriber

Symfony, como el *framework* que es, aporta un gran conjunto de funcionalidades al proyecto. Una de ellas es el sistema de eventos. Gracias a él, se pueden crear piezas de código para realizar una acción cuando un evento se dispara (EventListener). Ese EventListener lo podemos vincular a cualquier otra pieza de código del proyecto. También podemos crear un EventSubscriber. Ese EventSubscriber es un código creado para realizar una acción cuando uno o más eventos específicos se disparan.

Se ha creado la clase RequestReceivedSubscriber. Esta escuchará todos los eventos lanzados cuando se ejecute un controlador. Cuando se detecte que el controlador implementa alguna de las interfaces que se han creado para la ocasión (SetUpRequestControler y SetUpAuthRequestController) realizará las siguientes acciones:

Cuando se reciba una llamada a un controlador que implemente la interfaz SetUpRequestControler, se ejecutará el servicio RequestContextFiller, rellenando las variables de RequestContext con la información de las cabeceras. También configurará LogService y creará una carpeta de *logs* específica para ese entorno y un archivo de *log* específico para esa llamada.

Cuando se reciba una llamada a un controlador que implemente la interfaz SetUpAuthRequestControler, se ejecutarán todos los *setters* del RequestContext del conector con la información recibida en las cabeceras. Si las cabeceras no contienen la información necesaria, se leerá del archivo de variables de entorno del conector. Si ninguno de los dos orígenes de datos contiene alguna variable de entorno crítica, se lanzará una excepción.

```
	private function setUpAuthRequest(ControllerEvent $event): void
	{
		$request = $event->getRequest();


$this->requestContext->setBearerToken($request->headers->get('x-auth-bearer-t
oken') ?? $_ENV['META_BEARER_TOKEN'] ?? throw new
\Exception('META_BEARER_TOKEN not set'));

$this->requestContext->setBusinessId($request->headers->get('x-auth-business-
id') ?? $_ENV['META_BUSINESS_ID'] ?? throw new \Exception('META_BUSINESS_ID
not set'));
		$this->requestContext->setApplicationSecret($request->headers->g
	et('x-auth-application-secret') ?? $_ENV['META_APPLICATION_SECRET'] ??
	throw new \Exception('META_APPLICATION_SECRET not set'));
		$this->requestContext->setWhatsappBusinessAccountId($request->he
	aders->get('x-auth-whatsapp-business-account-id') ?? throw new
	\Exception('Header "x-auth-whatsapp-business-account-id" not set'));
		$this->requestContext->setWhatsappPhoneNumberId($request->header
	s->get('x-auth-whatsapp-phone-number-id') ?? throw new
	\Exception('Header "x-auth-whatsapp-phone-number-id" not set'));
	}
```

Código 4. Configuración de RequestContext con la información de las cabeceras y del archivo de variables de entorno del conector.

Destaca el hecho de que un controlador puede implementar más de una interfaz. Por lo tanto, un controlador creado para recibir y procesar peticiones del CGC podrá implementar la interfaz SetUpRequestControler y ejecutar el flujo de código normal. Y, aparte, implementar la interfaz SetUpAuthRequestControler y tener acceso también a las credenciales de la empresa.

5.1.1.3 KernelExceptionSubscriber

Este suscriptor de eventos está diseñado para que, cuando haya una excepción en el código, ejecute ciertas acciones:

1. Obtener información del error.

2. Si el servicio de *logs* está preparado para escribir, anota el error en el log.
    a. Cualquier excepción lanzada durante la escritura será ignorada para no interrumpir el procesamiento del error original.
3. Crea una respuesta con la información del error.
4. Sobreescribe la respuesta a la llamada con la información del error.

```
        $exception = $event->getThrowable();
        $class = get_class($exception);
        $errorMessage = "Exception thrown ({$class}):
{$exception->getMessage()} ON
{$exception->getFile()}:{$exception->getLine()}";

        try {
                if ($this->logger->canWrite()) {
                        $this->logger->error($errorMessage);
                }
        } catch (\Exception $logError) {
        }

        $event->setResponse(new JsonResponse([
                'message' => 'DATA_KO',
                'code' => $exception->getCode(),
                'error' => $errorMessage,
        ]));
```

Código 5. Tratamiento y respuesta común para cualquier error detectado durante la ejecución del código.

5.2 Base de Datos

El gestor de bases de datos utilizado es MySQL. Para acceder a la base de datos del proyecto Symfony, recomienda utilizar Doctrine [45]. Doctrine es una librería de código que contiene, entre otras funcionalidades, un ORM[30] y un DBAL[31] para PHP. Nos proporciona una capa de abstracción sobre la base de datos que nos permite programar sin tener en cuenta sobre qué soporte se guarda la información.

Se han creado las entidades especificadas en el apartado [4.3] usando un comando que proporciona Symfony mediante la librería symfony/maker-bundle. Ese comando pregunta por consola el nombre de la entidad, los campos que tendrá y de qué tipo son. Cuando terminamos de escribir la información, crea automáticamente una clase de la nueva entidad y un repositorio para dicha entidad.

Como hecho remarcable, hace falta comentar que los campos de tipo enum de la entidad *webhook* se han definido como *string*. El control sobre si la información a guardar en la base de datos es válida recae sobre el *setter* de dichos campos en la entidad. Estos *setters* comprobarán si el argumento de la llamada corresponde con un valor de un enum que

[30] ORM: Object Relational Mapper.
[31] DBAL: Database Abstraction Layer.

contiene los valores válidos para cada campo. Este hecho facilita la actualización del código en producción cuando haya que añadir más valores aceptados en esos campos.

```
enum WebhookStatus: string
{
	case PENDING = 'pending';
	case PROCESSED = 'processed';
	case ERROR = 'error';
}

// setter del campo status de la entidad Webhook
public function setStatus(WebhookStatus|string $status): static
{
	if (! ($status instanceof WebhookStatus)) {
		$status = WebhookStatus::tryFrom($status);
	}
	$this->status = $status->value;

	return $this;
}
```

Código 6. Validación vía código de una propiedad de tipo enumeración.

Aparte, se han implementado los siguientes *setters* en sendas entidades que se ejecutan cuando el ORM de Doctrine va a crear un registro, en el caso de setCreatedAtValue o cuando lo va a actualizar, como en el caso de setUpdatedAtValue. Este código establecerá la fecha y hora exactos cuando se ejecuta en los campos “created_at” y “updated_at”.

```
#[ORM\PrePersist]
public function setCreatedAtValue(): void
{
	$this->createdAt = new \DateTime("now");
	$this->updatedAt = new \DateTime("now");
}

#[ORM\PreUpdate]
public function setUpdatedAtValue(): void
{
	$this->updatedAt = new \DateTime("now");
}
```

Código 7. Escritura automática de los campos de creación y modificación del registro.

5.3 Graph API y WhatsApp Cloud API

La comunicación con los servicios de Meta se realiza mediante la API Graph de Meta. Gracias a esta API, podemos gestionar todos los activos de Meta. Estos incluyen los activos de Instagram, Facebook, anuncios, campañas de *marketing*, Messenger, WhatsApp y muchos más. Se ha utilizado el SDK[32] oficial de Meta [47] para realizar la mayoría de peticiones.

[32] SDK: Software Development Kit.

Este SDK no incluye algunas funcionalidades necesarias para gestionar los números de teléfono de la cuenta de WhatsApp del cliente o similares. Por lo que se ha creado un servicio, ConnectionService, para realizar peticiones a la API de Meta. Este servicio implementa el servicio CurlService de la librería de código común para realizar las peticiones.

Se ha creado también una clase para poder enviar mensajes al cliente final usando la cuenta de WhatsApp de la empresa. Esta clase utiliza ConnectionService para poder realizar las llamadas.

```
class ConnectionService
{
	private string $version = 'v19.0';

	private CurlService $curlService;

	public function __construct(
		protected LogService $logService,
		protected RequestContext $requestContext
	) {
	}

	public function setUp(): void
	{
		$this->curlService = new JsonCurlService();
		$this->curlService->setHeaders([
			'Authorization: Bearer '.
$this->requestContext->getBearerToken()
		]);
	}

	public function get($path): array
	{
		if (!isset($this->curlService)) $this->setUp();

		$this->setUrl($path);

		return $this->curlService->get();
	}

	public function post($path, $body): array
	{
		if (!isset($this->curlService)) $this->setUp();

		$this->setUrl($path);

		return $this->curlService->post($body);
	}

	protected function setUrl($path): void
	{
```

```
$this->curlService->setUrl("https://graph.facebook.com/{$this->version}/{$path}");
            }
      }
```

Código 8. Servicio que encapsula las peticiones a Meta.

```
      class PhoneRepository
      {
            public function __construct(
                  protected ConnectionService $connectionService,
                  protected RequestContext $requestContext,
                  protected LogService $logService,
            ) {
            }

            public function sendMessage(array $message): array
            {
                  $response = $this->connectionService->post(
                        $this->requestContext
                              ->getWhatsappPhoneNumberId().'/messages',
                        $message
                  );

                  return $response;
            }
      }
```

Código 9. Clase que encapsula las peticiones a Meta del número de teléfono. El CDU 14. SendMessage se ha implementado como la función sendMessage.

5.4 Sincronizar Cuenta

Esta acción está formada por los casos de uso: CDU 01. SyncAccount; CDU 02. CheckWebhookSubscription, y CDU 03. SyncApplication, y solventa los requisitos funcionales RF1 y RF2. El caso de uso CDU 01. SyncAccount, por ahora solo ejecuta el caso de uso CDU 02. CheckWebhookSubscription y, después, el CDU 03. SyncApplication. En caso de que se necesiten añadir más acciones a realizar, se tendría que crear un nuevo caso de uso y llamarlo desde este.

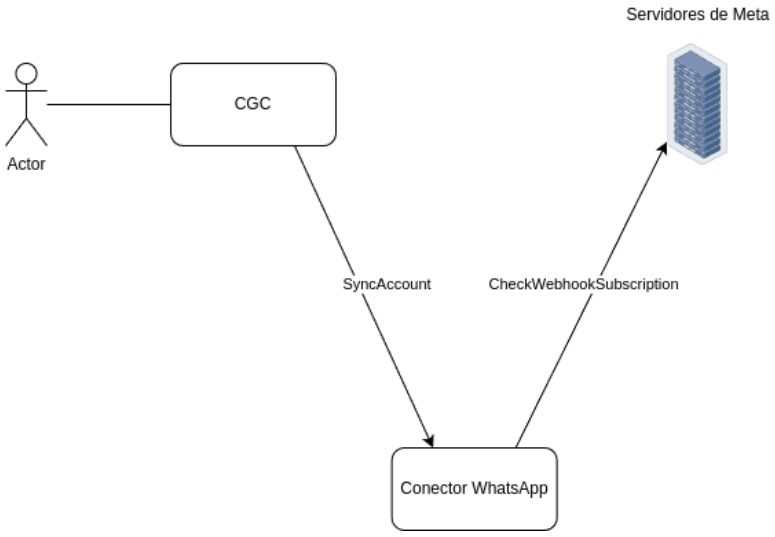

Figura 14. Diagrama de sincronización de cuenta.

5.4.1 CDU 02. CheckWebhookSubscription

Realiza una petición GET al siguiente endpoint [48]:

```
{whats-app-business-account-id}/subscribed_apps
```

y este nos devuelve las aplicaciones que están suscritas al *webhook* de la cuenta de WhatsApp del cliente. Se busca, dentro de la respuesta, una aplicación con el identificador de la aplicación de Meta de CatalogPlayer. Si está presente el caso de uso, ha terminado.

En caso de que no esté se realiza una petición POST al mismo *endpoint*. No se pasa parámetro alguno. Meta detectará la aplicación de CatalogPlayer por el identificador de acceso que utiliza el conector para realizar la petición y suscribirá la cuenta de Meta de CatalogPlayer a los *webhooks* de la cuenta de WhatsApp del cliente. Si todo funciona correctamente, se terminará el caso de uso. En caso de que no haya funcionado, se lanzará una excepción.

```
class CheckWebhookSubscription
{
	public function __construct(
		private RequestContext $requestContext,
		private LogService $logger
	) {
	}

	public function __invoke(): void
	{
		$cpApplicationId = $_ENV['META_APP_ID'] ?? null;
		if (!$cpApplicationId) throw new \Exception('META_APP_ID
not set');
		$wabaId =
$this->requestContext->getWhatsappBusinessAccountId();
```

```
                $adAccount = new AdAccount($wabaId);
                $subscribedApps = $adAccount->getSubscribedApps();

                $this->logger->info("AdAccount {$wabaId} is subscribed to
" . count($subscribedApps) . " applications");
                foreach ($subscribedApps as $subscribedApp) {
                        $appId =
$subscribedApp->getData()['whatsapp_business_api_data']['id'] ?? null;

                        if ($appId == $cpApplicationId) return;
                }

                $this->logger->info("Subscribing AdAccount {$wabaId} to
application $cpApplicationId...");
                $response = $adAccount->createSubscribedApp();

                if (!($response->getData()['success'] ?? true)) throw new
\Exception('Failed to subscribe to application');
            }
        }
```

Código 10. CDU 02. CheckWebhookSubscription.

5.5 Actualizar Catálogo de Meta

Esta acción está formada por los casos de uso: CDU 07. GetProduct; CDU 08. SetProduct, y CDU 13. GetProductCatalog, y solventa los requisitos funcionales RF3 y RF4. Para esta acción, necesitamos crear el catálogo en caso de que no exista, por lo que así se le indicará a GetProductCatalog.

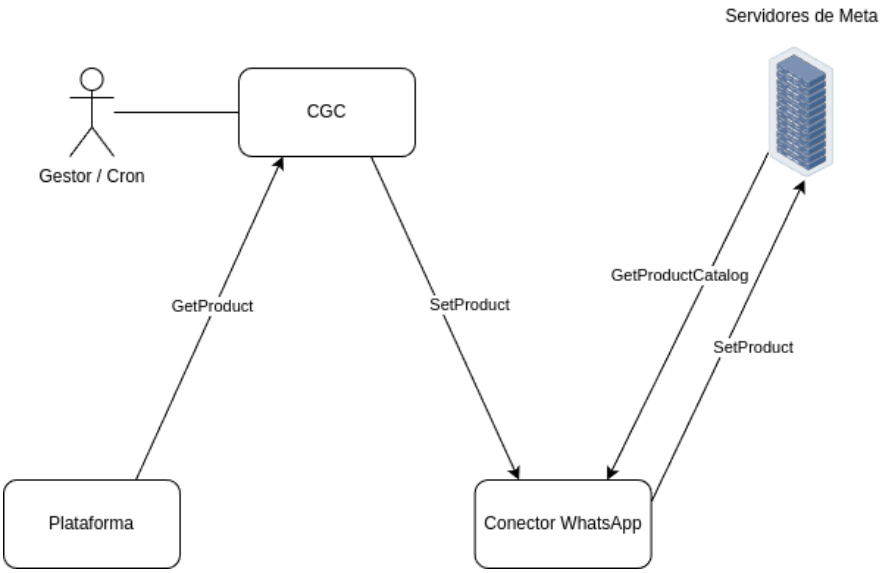

Figura 15. Diagrama de actualización de catálogo en Meta.

5.5.1 CDU 13. GetProductCatalog

Este caso de uso realiza una petición GET sobre el siguiente endpoint [49]:

```
{business-id}/owned_product_catalogs
```

la cual devolverá todos los catálogos de la aplicación en Meta de CatalogPlayer. Se buscará entre esos catálogos uno con el nombre de la empresa. Si no está, y se había indicado al caso de uso que se creara en caso de que no estuviera, se intentará crear mediante una petición POST al mismo *endpoint* y pasando como parámetro el nombre de la empresa.

5.5.2 CDU 08. SetProduct

Este caso de uso convierte los CPObject de producto a formato Meta. Una vez formateados, ejecuta una petición POST pasándole los productos al siguiente *endpoint* [50] a partir del catálogo devuelto por CDU 13. GetProductCatalog:

```
{product_catalog_id}/batch
```

Esta petición importará los productos y, cuando termine, se podrán enviar como oferta de productos en un mensaje de WhatsApp. Según algunas pruebas realizadas durante el desarrollo, puede tardar entre 20 y 30 minutos en estar disponibles. A partir de algunas búsquedas por internet y foros de atención de Meta, se llegó a la conclusión de que es el tiempo que tarda Meta en vincular su catálogo a la cuenta de WhatsApp [51]. Se realizará una encuesta continua, con un intervalo de 5 segundos, mediante una petición GET al siguiente *endpoint* para saber cuándo termina [52]:

```
{product-catalog-id}/check_batch_request_status
```

El código de la encuesta continua es el siguiente:

```
$response = $baseCatalog->createBatch([], $baseCatalogProducts);

if (isset($response->validation_status)) {
      $this->logger->error(json_encode($response->validation_status,
JSON_PRETTY_PRINT));
}

foreach ($response->handles ?? [] as $handle) {
      do {
            $this->logger->info("Waiting for batch request to finish.
ID = ".$handle);
            sleep(5);
            $finished = true;
            $batchRequestStatuses =
$baseCatalog->getCheckBatchRequestStatus([
                  CheckBatchRequestStatusFields::HANDLE,
                  CheckBatchRequestStatusFields::STATUS,
            ], [
                  CheckBatchRequestStatusFields::HANDLE => $handle,
```

```
CheckBatchRequestStatusFields::IDS_OF_INVALID_REQUESTS => true
                    ]);

                    foreach ($batchRequestStatuses as $batchRequestStatus) {
                          $finished &= $batchRequestStatus->status ==
'finished';
                          }
             } while (!$finished);

             $this->logger->info("Batch request finished. ID = ".$handle);
      }
```

Código 11. Encuesta continua para saber cuándo Meta ha importado todos los productos.

```
      public function __invoke(array $cpProducts): void
      {
             $this->logger->info("Updating ".count($cpProducts)."
products...");

             $baseCatalog = $this->getProductCatalog
                    ->createIfNotExists()
                    ->__invoke();

             $metaProducts = $this->cpProductsToMetaProducts($cpProducts);

             $this->fillCatalogWithProductsAsBatch($baseCatalog,
$metaProducts);
      }
```

Código 12. Flujo de ejecución principal de CDU 08. SetProduct.

La forma de configurar el CDU 13. GetProductCatalog para que cree el catálogo en caso de que no exista en lugar de lanzar una excepción es mediante la función createIfNotExists. Esta función tan solo establece un *booleano* interno del caso de uso a "Cierto" y devuelve la instancia como respuesta. Este retorno permite encadenar llamadas a funciones del mismo caso de uso (Method Chaining o Fluent Interface [53]) y facilita la lectura del código (Syntactic Sugar[33]).

5.6 Enviar Oferta de Productos o Cápsula *Microsite* al Cliente

Esta acción está formada por los casos de uso: CDU 09. GetProductOffer; CDU 10. SentProductOffer; CDU 11. GetCapsule; CDU 12. SendCapsule; CDU 13. GetProductCatalog, y CDU 14. SendMessage. Solventa los requisitos funcionales y no funcionales RF5, RF6 y RNF2.

[33] Sintaxis que permite una mejor lectura del código.

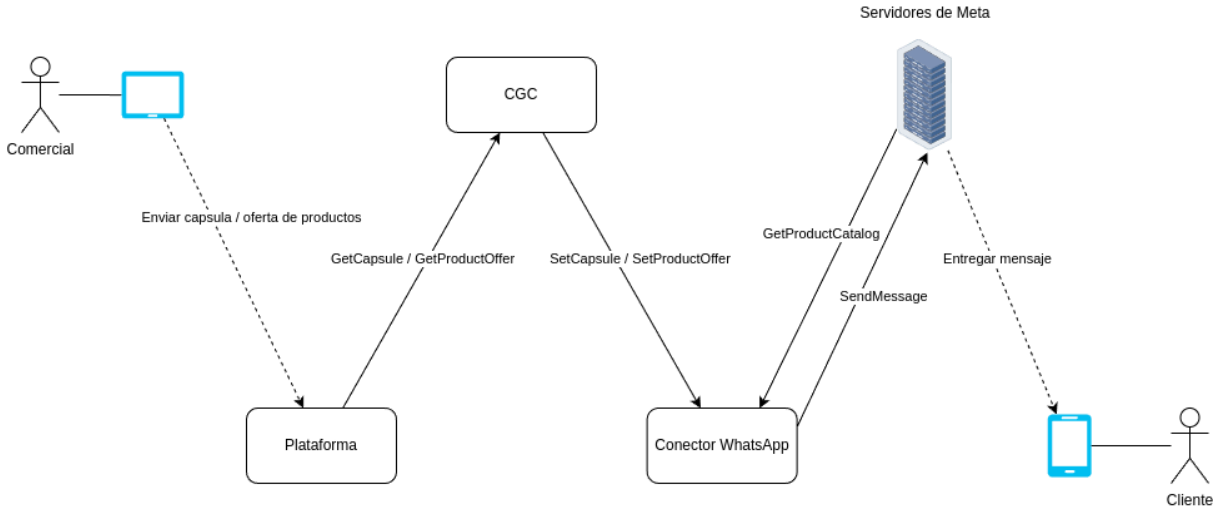

Figura 16. Diagrama de envío de cápsula o oferta de productos.

Los casos de uso 09 y 11 se han implementado en el código de la CPAPI. El resto, en el código del conector de WhatsApp. Su único propósito es convertir el formato CPObject en formato Meta y realizar una llamada al CDU 14. SendMessage.

5.6.1 CDU 14. SendMessage

Este caso de uso se ha implementado como una función de un repositorio [Código 9]. Su único objetivo es realizar una petición POST al siguiente *endpoint* [54] con el mensaje a enviar al cliente final:

```
{phone-number-id}/messages
```

Hay muchos campos que podemos enviar para formar un mensaje de WhatsApp. Los campos más importantes que usamos son:

- messaging_product: para especificar el servicio de mensajes de Meta a utilizar. Desde este conector siempre enviaremos “whatsapp”.
- recipient_type: siempre enviaremos “individual”
- to: número de teléfono de destino
- type: tipo de mensaje. Los valores aceptados son “text”, “interactive”, “audio”, “image”, “location”, “document”, “contacts”, “sticker” y “template”.

Falta un campo más, el cual depende del tipo seleccionado. Ese campo contendrá toda la información relativa al mensaje a enviar.

5.6.2 CDU 12. SendCapsule

El tipo de mensaje a enviar es “interactive - cta_url”. Contendrá, para la empresa que lo envía, una imagen, un texto y un enlace al *microsite* del cliente. Este *microsite* es un apartado de la plataforma específico para ese cliente y contiene los productos que el comercial le ha compartido.

```
	private function mapMicrositeToWhatsappMessage(array $capsule): array
	{
		return [
			'messaging_product' => 'whatsapp',
			'recipient_type' => 'individual',
			'to' => $capsule['Contact']['Phone'],
			'type' => 'interactive',
			'interactive' => [
				'type' => 'cta_url',
				'header' => [
					'type' => 'image',
					'image' => [
						'link' => $capsule['ImageThumb']
					]
				],
				'body' => [
					'text' => $capsule['Header']
				],
				'action' => [
					'name' => 'cta_url',
					'parameters' => [
						'display_text' =>
$this->buttonText($capsule['Language']),
						'url' => $capsule['Url']
					]
				]
			]
		];
	}

	private function buttonText(string $language): string
	{
		return match ($language) {
			'ca' => 'Veure més',
			'en' => 'See more',
			default => 'Ver más',
		};
	}
```

Código 13. Conversión de cápsula en formato CPObject a mensaje de WhatsApp

El cliente recibirá un mensaje como el siguiente:

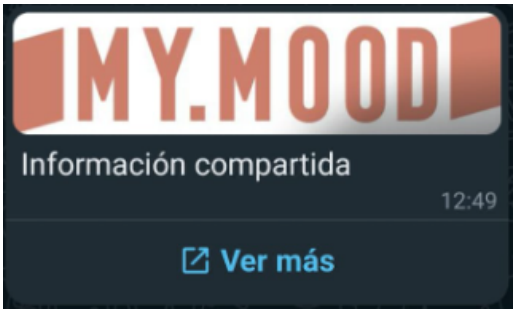

Figura 17. Cápsula recibida como mensaje de WhatsApp.

Se ve una imagen, obtenida de la plataforma de la empresa. También un texto que el comercial puede editar y un botón que redirige al *microsite* específico para el cliente en la plataforma de la empresa.

5.6.3 CDU 10. SentProductOffer

Este caso de uso llama al CDU 13. GetProductCatalog para conocer qué catálogo referenciar al enviar los productos. El tipo de mensaje es “interactive - product_list”. Puede contener hasta 30 productos en, como máximo, 10 secciones distintas.

```
      private function mapMessageToWhatsappMessage(mixed $productOffer,
string $catalogId): array
      {
            $sections = [
                  [
                        'title' => 'Productos',
                        'product_items' => [],
                  ],
            ];

            foreach ($productOffer['Products'] as $product) {
                  if (empty(array_filter($sections[0]['product_items'] ??
[], fn ($i) => $i['product_retailer_id'] ===
$product['ProductPrimaryToken']))) {
                        $sections[0]['product_items'][] = [
                              'product_retailer_id' =>
$product['ProductPrimaryToken']
                        ];
                  }
            }

            $message = [
                  "messaging_product" => "whatsapp",
                  "recipient_type" => "individual",
                  "to" => $productOffer['Contact']['Phone'],
                  "type" => "interactive",
                  "interactive" => [
                        "type" => "product_list",
                        "header" => [
                              "type" => "text",
                              "text" => $productOffer['Header'] ?? '',
                        ],
                        "body" => [
                              "text" => $productOffer['Body'] ??
$this->defaultBodyText($productOffer['Language']),
                        ],
                        "action" => [
                              "catalog_id" => $catalogId,
                              "sections" => $sections,
                        ],
                  ],
            ];
```

```
            if (!empty($productOffer['Footer'])) {
                  $message['interactive']['footer']['text'] =
$productOffer['Footer'];
            }

            return $message;
      }

      private function defaultBodyText(string $language): string
      {
            return match ($language) {
                  'ca' => 'Aquí tens els productes que has sol·licitat',
                  'en' => 'Here are the products you have requested',
                  default => 'Aquí tienes los productos que has solicitado',
            };
      }
```

Código 14. Conversión de oferta de productos en formato CPObject a mensaje de WhatsApp.

Figura 18. Ejemplo de envío de oferta de productos visto desde el WhatsApp de un posible cliente.

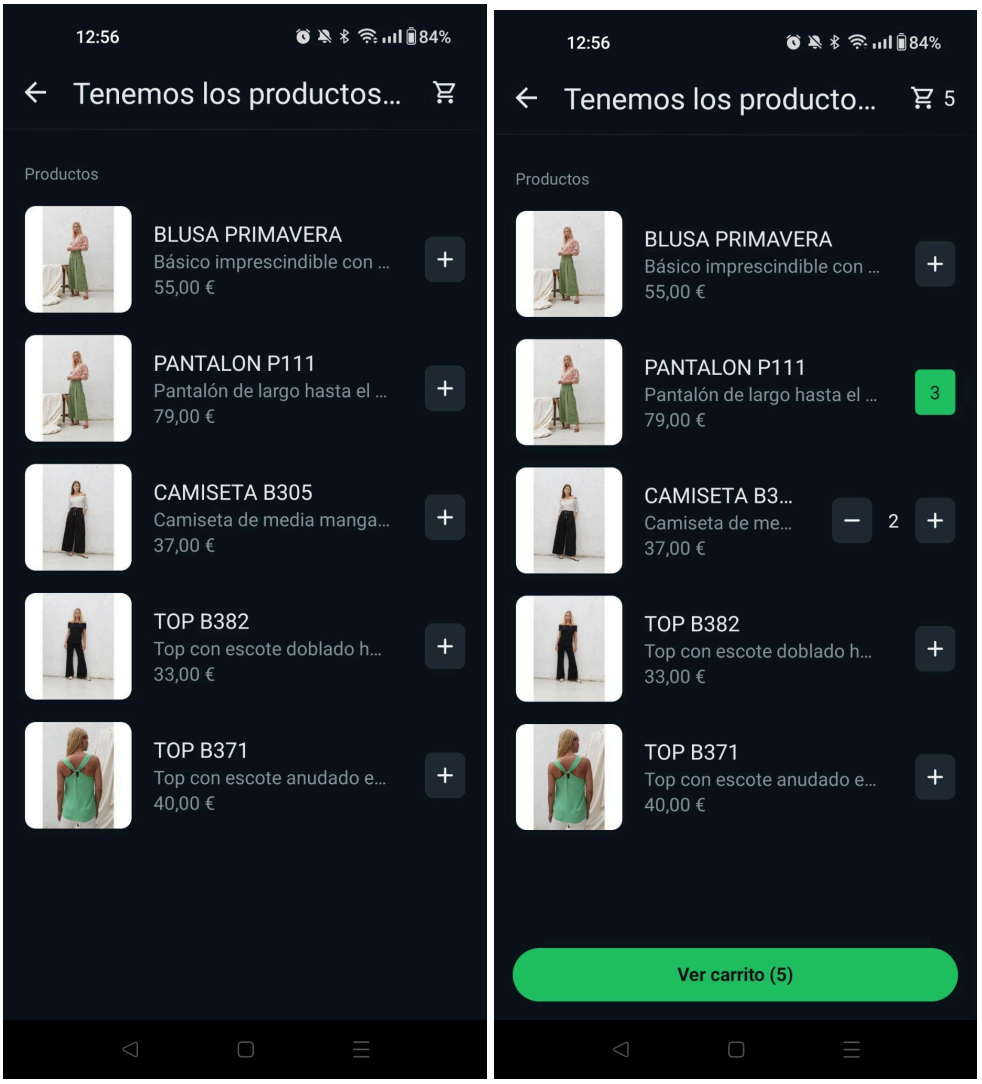

Figura 19. Ejemplo de listado de productos enviados en una oferta de productos y selección de los productos deseados.

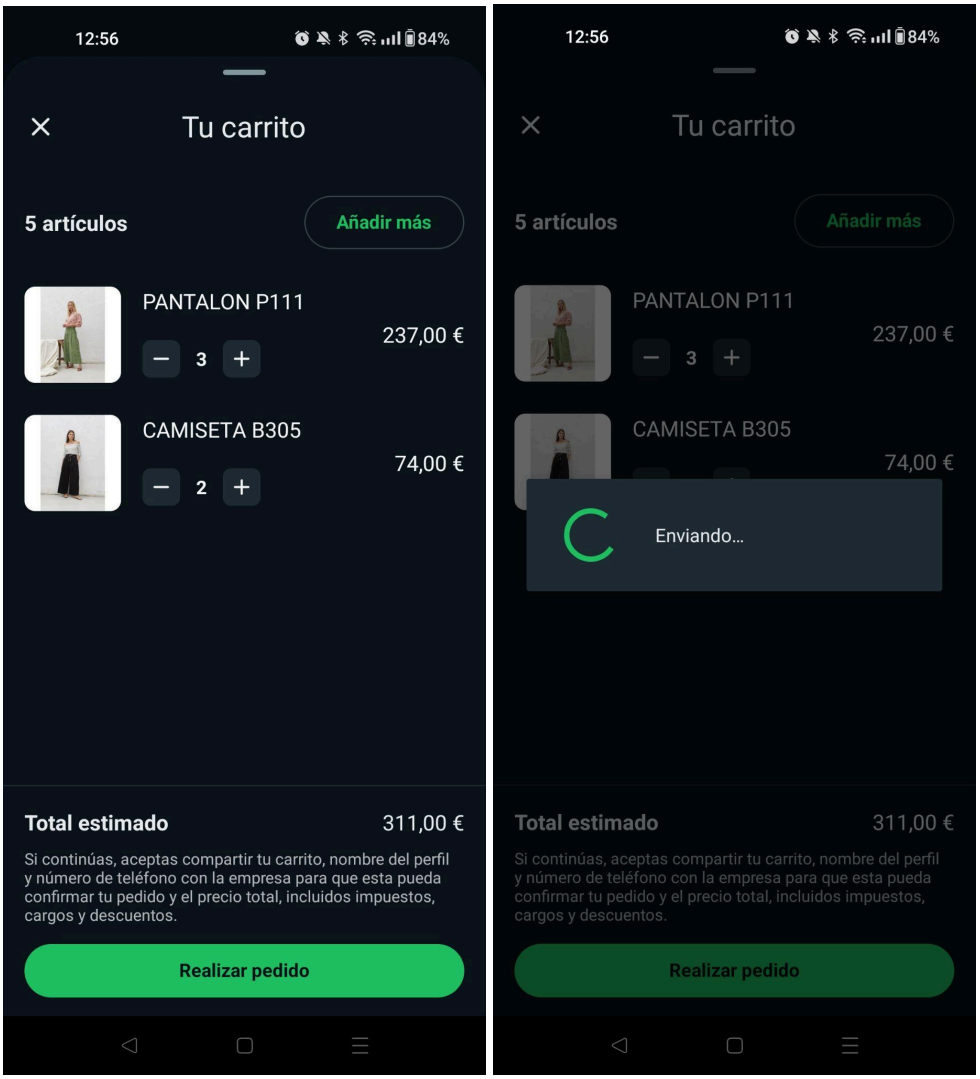

Figura 20. Ejemplo de visualización del carrito y envío de pedido.

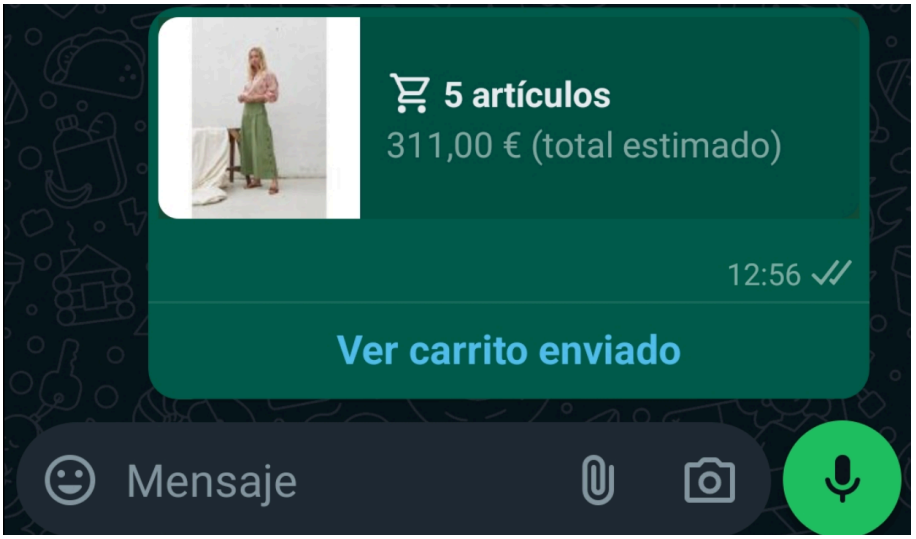

Figura 21. Ejemplo de pedido enviado desde el WhatsApp del cliente.

5.7 Procesar un *Webhook* de Meta

Esta acción está integrada por los casos de uso: CDU 15. ProcessWebhook; CDU 04. EntitySync; CDU 17. PersistWebhook, y CDU 16. ValidateOriginIp. Solventa los requisitos funcionales y no funcionales RF7 y RNF1.

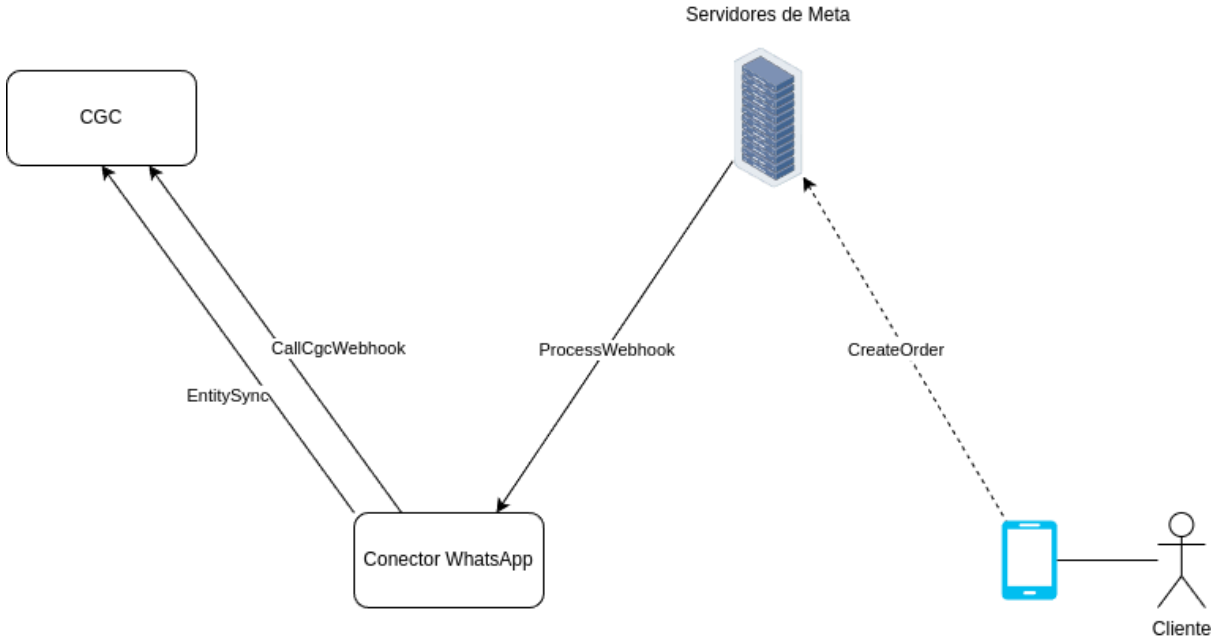

Figura 22. Diagrama de procesamiento de *webhooks* enviados por Meta.

Suscribir nuestra aplicación en Meta de CatalogPlayer a los *webhooks* de cada cuenta de cliente es un paso necesario para poder recibir los eventos y pedidos del cliente. Pero nuestro conector aún es ajeno a dichos eventos. El siguiente paso es hacer que la aplicación de Meta de CatalogPlayer redirija el *webhook* hacia un *endpoint* en nuestro proyecto.

Para poder recibir *webhooks* de Meta, hay que seguir las instrucciones que nos indican [55]. Hay que crear un *endpoint* que acepte una petición GET y pueda leer de la URL los argumentos hub_mode, hub_verify_token y hub_challenge. hub_mode ha de contener el valor “subscribe” y hub_verify_token debe contener un *token* secreto que solo debe conocer Meta y nuestro proyecto. Si el token enviado es el mismo que el que tenemos definido en nuestra variable de entorno, devolvemos como respuesta hub_challenge para informar a quien ha realizado la petición que ha pasado el reto.

```
	#[Route('/api/meta/webhooks', name: 'api_verify_request_webhook',
methods: ['GET'])]
	public function verifyRequestWebhook(Request $request)
	{
		$this->logger->info("Verify request received!");
		$this->logger->info("Body: ".$request->getContent());
		$this->logger->info("URL: ".$request->getRequestUri());
		$this->logger->info("Request:
".json_encode($request->request->all()));
```

```
            $query = $request->query->all();

            if ($query['hub_mode'] == 'subscribe' &&
                  $query['hub_verify_token'] ==
$_ENV['META_VERIFY_REQUEST_TOKEN']
            ) {
                  $this->logger->info("Challenge accepted!");
                  return new Response($query['hub_challenge'], 200);
            } else {
                  $this->logger->info("Challenge failed");
                  return
$this->json(ResponseService::createResponse('DATA_KO', 403, 'Challenge
failed', '"hub.challenge" does not match with META_VERIFY_REQUEST_TOKEN or
"hub.mode" is invalid', []));
            }
      }
```

Código 15. *Endpoint* para verificar el conector como un receptor válido de *webhooks* de Meta.

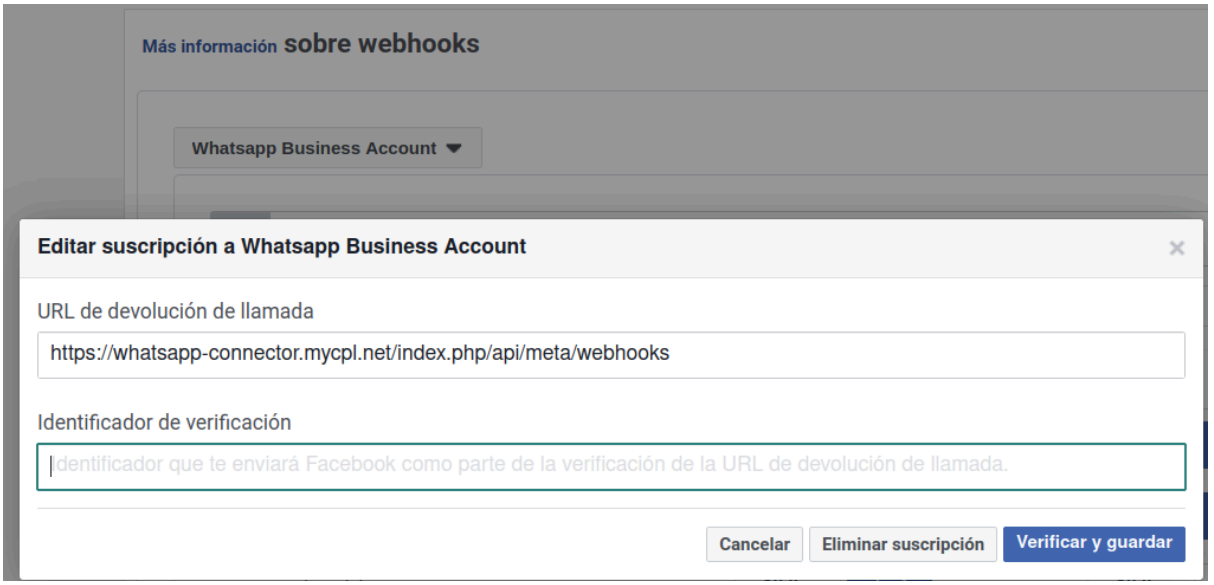

Figura 23. Configuración del endpoint creado como receptor de webhooks de Meta.

Una vez verificado nuestro endpoint podremos escoger a qué eventos nos queremos suscribir según su entidad. Las entidades son:

- User
- Application
- Whatsapp Business Account
- Page
- Permissions
- Instagram
- Certificate Transaprency
- Ad Account

Por el momento tan solo nos interesa suscribirnos a los eventos “messages” de Whatsapp Business Account. Otros eventos interesantes serían “flows” y “message_echoes”.

5.7.1 CDU 16. ValidateOriginIp

Para asegurar que el *webhook* proviene de un servidor de Meta, este nos ofrece la posibilidad de consultar de qué IP pueden provenir sus peticiones [56]. Esta acción, aunque no tarde un tiempo excesivo, no es inmediata. Por ello, se ha implementado una caché de Symfony con una caducidad de una hora.

```
	public function __invoke(string $ip): bool
	{
		$cache = new FilesystemAdapter('app.cache'); // TODO: Inject
this dependency

		$validIps = $cache->get('webhook_valid_origin_ips', function
(CacheItem $item) {
			$item->expiresAfter(3600);
			return $this->fetchValidOriginIps();
		});

		return in_array($ip, $validIps);
	}

	private function fetchValidOriginIps(): array
	{
		$this->logger->info('Getting valid origin IPs for webhooks...');

		exec("whois -h whois.radb.net — '-i origin AS32934' | grep
^route | awk '{print $2}' | sort | cut -d '/' -f 1", $output, $resultCode);

		$this->logger->info('Got valid origin IPs for webhooks: ' .
json_encode([
			'output' => $output,
			'resultCode' => $resultCode,
		]));

		return $resultCode == 0 ? $output : [];
	}
```

Código 16. CDU 16. ValidateOriginIp.

Este caso de uso recibe una IP en forma de *string* y devuelve “Cierto” en caso de que la IP provenga de un servidor de Meta o “Falso” en caso contrario.

Para implementar este caso de uso en el proyecto, se ha modificado RequestReceivedSubscriber para que el evento que se dispara cuando un controlador es llamado ahora también compruebe si el controlador implementa la interfaz ValidateOriginIp.

En caso de que la implemente, y la variable de entorno APP_ENV[34] no esté a "dev", se ejecutará el siguiente código que llamará al CDU 16. ValidateOriginIp.

```
	private function validateOriginIp(ControllerEvent $event): void
	{
		$request = $event->getRequest();

		if ($_ENV['APP_ENV'] != 'dev' &&
!$this->validateWebhookOriginIp->__invoke($request->getClientIp())) {
			$event->setController(fn() => new
JsonResponse(ResponseService::createResponse(
				'DATA_KO',
				403,
				'Invalid origin IP',
				'Client IP is not allowed to access this resource.',
				[]
			), 403));
		}
	}
```

Código 17. Llamada al CDU 16. ValidateOriginIp

Cuando se haya validado que la IP de origen de la petición es aceptada, se continuará con la ejecución del código. Acto seguido, se ejecutará el CDU 15. ProcessWebhook. Este, tal y como se ha descrito en el apartado 3.3.15, ya realizará las llamadas a los otros casos de uso involucrados -CDU 04. EntitySync, CDU 16. PersistWebhook y CDU 18. CallCgcWebhook- según el tipo de *webhook* que se reciba.

5.8 Enviar Pedidos a la Plataforma

Esta acción está compuesta por los casos de uso: CDU 05. GetOrder y CDU 06. SetOrder. Solventa el requisito funcional RF8.

[34] Esta variable especifica el entorno en el que se ejecuta el código del proyecto. Los valores comunes son "dev" para desarrollo, "prod" para producción o "test" para cuando se ejecuta la suite de tests.

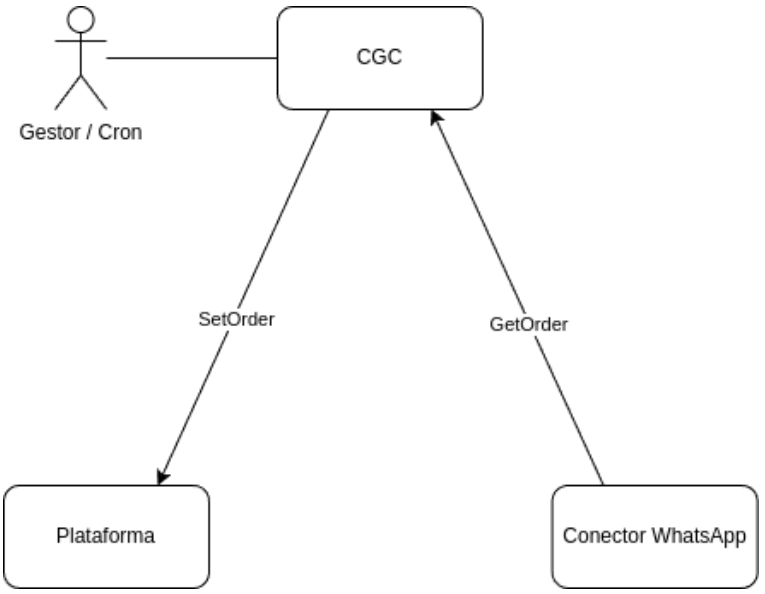

Figura 24. Diagrama de sincronización de pedidos.

Este es el paso final. Ya sea por sincronización manual, programada o por llamada al CGC para iniciar una sincronización (CDU 04. EntitySync), se realizará una petición HTTP GET sobre el *endpoint* creado específicamente para la entidad order. Este llamará el CDU 05. GetOrder y devolverá su retorno.

```
#[Route('/api/order', name: 'api_order', methods: ['GET'])]
public function getOrder(Request $request)
{
	$orders = $this->getOrder->__invoke();

	return $this->json(ResponseService::createResponse('DATA_OK',
200, '', count($orders).' orders retrieved', $orders));
}
```

Código 18. Endpoint GET /api/order.

5.8.1 CDU 05. GetOrder

A partir de las credenciales de las cabeceras de la petición, se obtendrá el identificador de la cuenta de WhatsApp de la empresa. Se consultarán los *webhooks* almacenados en la base de datos de tipo “order”, con estado “pendiente” y que correspondan a la aplicación vinculada al identificador de la cuenta de WhatsApp de la empresa. Todos los webhooks coindcidentes con la búsqueda se convertirán a CPObjects y se modificará su estado a “processed”.

```
public function __invoke(): array
{
	$wabaId = $this->requestContext
		->getWhatsappBusinessAccountId();
	$application = $this->applicationRepository
		->findOneBy(['wabaId' => $wabaId]);
	if (!$application) {
```

```
                $errorMessage = 'Application not found for WABA ID: ' .
                        $wabaId;
                $this->logger->error($errorMessage);
                throw new \Exception($errorMessage);
            }
            $orders = $this->webhookRepository
                ->findPendingOrdersByApplication($application);

            $cpOrders = $this->makeCpOrders($orders);

            $this->entityManager->flush();

            return $cpOrders;
        }
```

Código 19. Lógica principal de CDU 05. GetOrder.

```
        public function findPendingOrdersByApplication(Application
$application): array
        {
            return $this->createQueryBuilder('w')
                ->andWhere('w.type = :type')
                ->andWhere('w.status = :status')
                ->andWhere('w.application = :application')
                ->setParameter('type', WebhookType::ORDER->value)
                ->setParameter('status', WebhookStatus::PENDING->value)
                ->setParameter('application', $application)
                ->getQuery()
                ->getResult();
        }
```

Código 20. Filtro aplicado para buscar los pedidos a devolver.

6 Pruebas

Se han realizado pruebas de integración con el objetivo de verificar el correcto funcionamiento del conector. Se han creado, en la colección de Postman, ejemplos de peticiones que se pueden ejecutar directamente sobre el proyecto.

6.1 CDU 02. CheckWebhookSubscription - No hace nada al estar suscritos al *webhook*

El CDU no debe realizar la llamada a la API Graph suscribiendo la aplicación de CatalogPlayer al *webhook* de la aplicación de la empresa en caso de que ya lo esté.

6.2 CDU 02. CheckWebhookSubscription - Se suscribe al webhook

El CDU debe realizar la llamada a la API Graph suscribiendo la aplicación de CatalogPlayer al *webhook* de la aplicación de la empresa en caso de que no lo esté.

6.3 CDU 03. SyncApplication - Registra la aplicación en base de datos

Se insertan todos los datos recibidos por las cabeceras de la petición en la base de datos.

6.4 CDU 03. SyncApplication - Actualiza la aplicación de base de datos

En caso de que exista un registro con el mismo identificador de aplicación, se actualiza con los datos recibidos por las cabeceras de la petición.

6.5 CDU 13. GetProductCatalog - Obtiene el identificador del catálogo

En caso de que exista el catálogo correspondiente a la empresa, lo devuelve.

6.6 CDU 13. GetProductCatalog - Crea un catálogo

En caso de que no exista el catálogo correspondiente a la empresa, lo crea y se lo devuelve.

6.7 CDU 13. GetProductCatalog - No crea un catálogo

En caso de que no exista el catálogo correspondiente a la empresa, lanza una excepción informando del error.

6.8 CDU 12. SendCapsule - Envía el mensaje al cliente

Envía correctamente un mensaje de tipo *capsule* al cliente final.
Informa a la plataforma del identificador del mensaje de WhatsApp enviado.
El mensaje tarda menos de 5 segundos en llegar, en la mayoría de los casos.

6.9 CDU 12. SendCapsule - Informa del error a la plataforma

En caso de que ocurra algún error durante el envío del mensaje, se informa del error a la plataforma.

6.10 CDU 10. SendProductOffer - Envía una oferta de productos

Envía correctamente un mensaje de tipo product_offer al cliente final.
Informa a la plataforma del identificador del mensaje de WhatsApp enviado.
El mensaje tarda menos de 5 segundos en llegar, en la mayoría de los casos.

6.11 CDU 10. SendProductOffer - No envía productos repetidos

Descarta los identificadores de productos repetidos para evitar un error al enviar el mensaje.

6.12 CDU 10. SendProductOffer - Informa del error a la plataforma

En caso de que ocurra algún error durante el envío del mensaje, se informa del error a la plataforma.

6.13 CDU 15. ProcessWebhook - Registra los *webhook* de tipo order

Crea un registro en la base de datos con los datos recibidos del *webhook*.
No falla si no encuentra una cuenta con el identificador de la cuenta de WhatsApp Business.

6.14 CDU 15. ProcessWebhook - Registra los *webhook* de tipo order vinculados a una cuenta

Crea un registro en la base de datos con los datos recibidos del *webhook.*
Lo vincula a la cuenta de WhatsApp Business al encontrar un registro coincidente.

6.15 CDU 15. ProcessWebhook - Devuelve error al verificar la IP de origen

Devuelve un error si el proyecto se está ejecutando en modo prod y la IP de origen no está entre las aceptadas.

6.16 CDU 05. GetOrder - Devuelve los pedidos pendientes de una aplicación

Devuelve todos los pedidos en forma de *webhook* que hay registrados en la base de datos y vinculados a la aplicación que realiza la petición.
Actualiza cada registro consultado marcándolo como procesado.

6.17 CDU 05. GetOrder - No se ejecuta en caso de no encontrar la aplicación

En caso de no encontrar un registro coincidente con la aplicación que ejecuta la petición, devuelve un error informando del problema.

7 Paso a Producción

Para desplegar el conector en el servidor en producción, hay que realizar los siguientes pasos:

1. Clonar el repositorio git que contiene el conector en el servidor.
2. Crear el archivo .env con las variables de entorno del proyecto. Esto incluye el *token* de acceso *oauth*, el identificador de la aplicación de CatalogPlayer en Meta u otros parámetros importantes como las credenciales a la base de datos.
3. Instalar las dependencias del proyecto.
4. Crear una base de datos y un usuario específico para el conector.
5. Crear una ruta específica para el conector donde guardar los archivos de *log*.
6. Ejecutar las migraciones del conector. Se crearán las tablas en la base de datos según las entidades del apartado 5.2 .
7. Crear un *vhost* para el conector en la configuración de Apache. En este *vhost* se indicará la ruta con la que acceder al conector, la versión de PHP a usar y la ruta en el sistema de archivos donde guardar los *logs* de Apache correspondientes a las peticiones de este conector.
8. Crear el conector en el CGC. Se configuran su ruta, sus entidades, sus acciones y configuraciones.

Hasta este punto ya se pueden realizar peticiones al conector y se considera en producción. Cuando una empresa contrate el conector de WhatsApp, habrá que realizar el siguiente paso:

9. Realizar el onboarding que vinculará la plataforma con el conector. Para ello, se creará una aplicación de la CPAPI con las credenciales de la plataforma de la empresa. También una aplicación para el conector de WhatsApp con los identificadores de la cuenta de WhatsApp de la empresa. Y, para finalizar, una conexión entre ambas para poder realizar sincronizaciones.

8 Conclusiones

8.1 Costes

8.1.1 Costes de desarrollo y mantenimiento

Los costes de desarrollo agrupan todos los recursos estimados que CatalogPlayer ha destinado para desarrollar el canal de ventas vía WhatsApp. Estos incluyen los de las licencias de las herramientas de *software* para el desarrollo, el tiempo dedicado a *devops* y análisis, bienes materiales y de avituallamiento de los empleados de CatalogPlayer y, como no, el tiempo dedicado a desarrollar el código y su implementación.

Los podemos separar en:

- **Costes directos**
  - **160 horas de investigación, desarrollo y documentación = 3384 €**
    Si un desarrollador cobra, por ejemplo, un sueldo de 30.000 € brutos anuales, CatalogPlayer destina 39.420€ anuales a ese empleado [57].

    Poniendo como ejemplo 2024 (año bisiesto) [58]:
    366 días - 52 sábados - 52 domingos - 10 días festivos = 252 días hábiles
    256 días hábiles - 23 días de vacaciones = 233 días hábiles reales
    233 días hábiles reales x 8 horas / día = 1864 horas trabajadas / año
    coste trabajador anual / Horas trabajador anuales =
    39.420 / 1864 = 21,15 € / hora

  - **32 horas en *devops* y análisis = 676,8 €**
    Podemos considerar que para *devops*, análisis y mantenimiento del servidor se destina un 20% de las horas reservadas a investigación y desarrollo.
    160 x 0,2 = 32 horas

- **Costes indirectos**
  - **Avituallamiento = 50 € / mes**
    Incluye portátil, luz e internet de cada empleado.

  - **Postman = 320,11 € / año**
    Plan profesional para 7 usuarios:
    348 $ / año = 320,11 € / año

  - **Licencia PhpStorm = 99 € / año**
  - **Infraestructura empresa = 100 € / mes**
    Incluye costes de gestión como:
    - El tiempo destinado por parte de un gestor:
      - Atención al cliente
      - Configuración y mantenimiento de la plataforma

- Licencia del *software* de facturación
- Licencia del *software* de nóminas

8.1.2 Precio módulo WhatsApp

CatalogPlayer

El precio que una empresa pagará a CatalogPlayer por añadir el módulo de Whatsapp a su plataforma es de 3.000 € / año.

Meta

Meta cobra por cada conversación establecida con un cliente, no por mensajes individuales. Define como conversación una cadena de mensajes de 24 horas entre el emisor (la empresa) y los receptores (los clientes de la empresa). Se inician y se cobran cuando se entregan los mensajes a los receptores [59].

A parte Meta aplica diferentes tarifas según el tipo de conversación:

Tipo	Descripción	Precio por conversación
Marketing	Aquellas relacionadas con venta de productos y fidelización de clientes.	0,0509 €
Utilidad	Permiten hacer un seguimiento de las acciones o solicitudes de los usuarios.	0,0315 €
Autenticación	Permiten autenticar usuarios con códigos de acceso de un solo uso.	0,0283 €
Servicio	Ayudan a resolver las consultas de los clientes.	0,0305 €

Tabla 2. Tarifas de conversaciones para WhatsApp España.

El tipo de mensajes que utilizamos en el módulo es el tipo Marketing. La empresa no se beneficia, por lo tanto, del bono de 1.000 conversaciones de servicio gratuitas por mes que Meta ofrece para los mensajes de tipo Servicio.

8.1.3 Amortización de la inversión

En el momento de la redacción de este documento, existen 2 empresas que ya han contratado el módulo del canal de ventas vía WhatsApp. Hay 3 más que se han mostrado interesadas y lo contratarán este año. Teniendo en cuenta únicamente los costes directos de desarrollo, podemos deducir cuántos meses tardará CatalogPlayer en amortizar la inversión según el número de empresas que contraten el módulo.

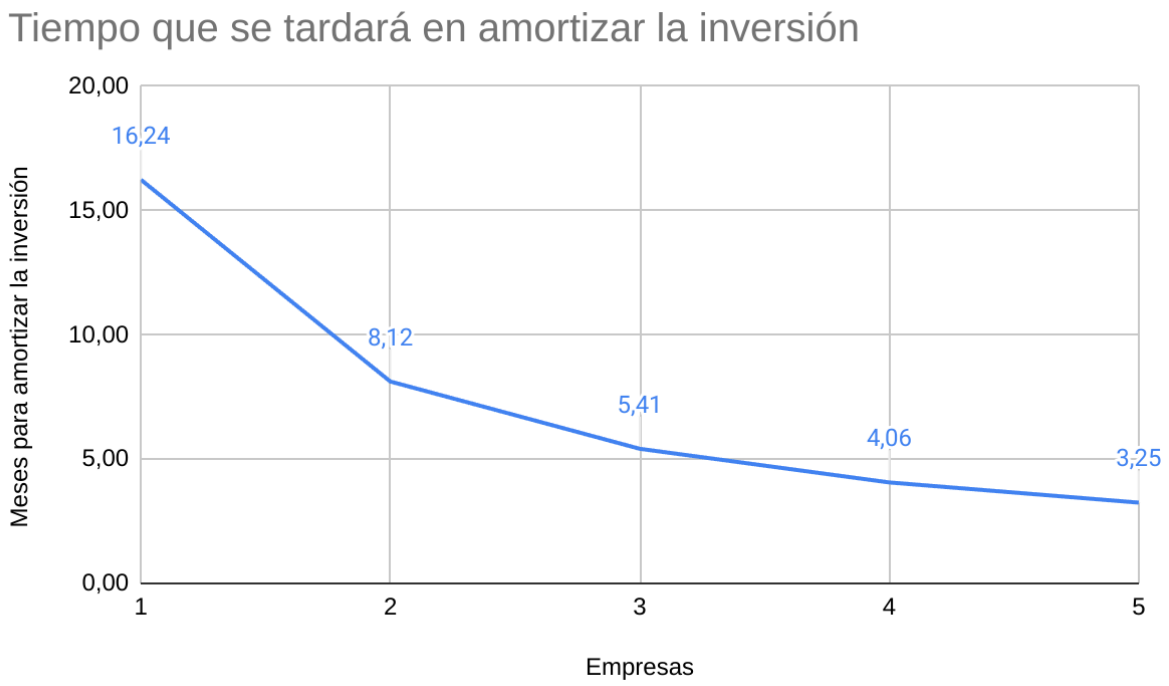

Figura 25. Gráfica que muestra el número de meses que se tardará en amortizar la inversión en desarrollo según el número de clientes que contraten el módulo.

8.2 Futuro del Conector de WhatsApp

8.2.1 Mantenimiento

Como está documentado en esta memoria, se han implementado buenas prácticas durante el desarrollo pensando en el trabajo en equipo y la sostenibilidad del código. Principios como SOLID, la definición de métodos de forma que sea fácil comprender qué hacen y la implementación de patrones de diseño ayudan a crear un código de calidad que facilita su comprensión y reduce las horas de depuración y desarrollo.

El hecho de localizar los servicios críticos y repetitivos en el conjunto de nuestros proyectos, y su posterior conversión a librería de código, agiliza los nuevos desarrollos sirviendo como punto de partida, y asegurando su correcto funcionamiento al ser implementado y probado en más ocasiones. Debemos tener en cuenta que esta librería y los servicios que comprende se convierten en un SPOF[35]. Cualquier error que se nos pasase por alto y lo desplegásemos como versión funcional, afectaría a todos los proyectos que implementen la librería.

En cuanto a la seguridad, en un futuro se plantea implementar una tarea automática que revise las dependencias del archivo composer.lock analizando sus vulnerabilidades. En cuanto se detecte un número excesivo de alertas o, directamente, un error de seguridad, se notificará a los desarrolladores para que se actualicen las dependencias.

[35] SPOF: Single Point of Failure.

8.2.2 CI / CD

Continuous Integration

La revisión automática de brechas de seguridad en las dependencias de los proyectos comentada en el apartado anterior es una de las tareas automáticas a realizar. También se ha planteado una revisión del estilo del código que realice un *commit* con los cambios planteados antes de realizar el *merge* a la rama principal. Este estilo se definirá con herramientas como StyleCI [60].

Continuous Delivery

En el momento de redacción de este documento, se han contratado 2 servidores nuevos en CatalogPlayer. Estos servidores tienen como objetivo sostener toda la infraestructura actual de conectores en forma de conectores Docker. Por el momento, hay 6 proyectos desplegados en contenedores, el conector de WhatsApp de desarrollo entre ellos. De esta forma, tendremos acotadas las dependencias de cada conector según sus características. Y, aunque por el momento sea un trabajo manual, en un futuro muy próximo se desplegará cada conector automáticamente cada vez que se realice un *merge* a la rama principal en su repositorio git.

8.2.3 IA

Mientras yo he desarrollado el nuevo canal de ventas para CatalogPlayer, otro de mis compañeros ha desarrollado, también para un TFG, un asistente de ventas mediante OpenAI. De momento, tan solo interactúa con la plataforma. Pero al estar diseñado como un conector, cualquier otro conector, como el de WhatsApp, puede interactuar con él.

Para este año, el conector de WhatsApp podrá realizar consultas al conector de OpenAI para atender las peticiones directas del cliente. El cliente podrá realizar preguntas como: cuánto stock queda de un artículo, obtener un artículo a partir de una descripción, buscar artículos similares a otro y muchas otras funciones que aún no se han planteado. Todo ello desde su cuenta de WhatsApp, ya sea desde su ordenador o desde su teléfono móvil.

Esta nueva conexión entre ambos conectores planteará nuevos retos para CatalogPlayer. Hasta ahora, los conectores interactúan entre sí mediante el orquestador de conectores, el CGC. Esto es así dado que, hasta ahora, las sincronizaciones de entidades eran de un tamaño de datos considerable y, normalmente, unidireccionales. Esta nueva conexión, a pesar de todo, sería de peticiones minúsculas propias de la mensajería instantánea. Con el añadido de que los datos con los que alimentar previamente mediante “prompt”[36] (como los productos y sus características) están en la plataforma y no en el conector de Open AI, por lo que se necesitaría otra comunicación con otro conector, la CPAPI. Otros métodos de comunicación alternativos a peticiones HTTP, como *websockets*, y evitando el CGC como actor intermediario, están sobre la mesa.

[36] Entrada inicial o la instrucción proporcionada al modelo para guiar la generación de texto. [61]

8.3 Análisis de Resultados

Viendo con retrospectiva lo conseguido en este proyecto, considero que ha resultado una experiencia muy educativa. Integrar una API moderna en un proyecto en producción conlleva algunos retos, pero poco a poco se han ido superando.

Desde que entré en CatalogPlayer, he podido participar en decisiones de diseño para replantear la forma en la que hacíamos las cosas. Este proyecto ha servido como conejillo de indias con el que probar algunas dinámicas de trabajo nuevas y arquitectura del código distinta, verificando su correcto funcionamiento para poder extrapolarlas a otros conectores.

He adquirido nuevos conocimientos con el *framework* de Symfony, aprendido a paquetizar código en librerías y paquetizar proyectos en Docker.

Considero que se han alcanzado los objetivos fijados al inicio del proyecto y se han cumplido sus requisitos, tanto funcionales como no funcionales.

Referencias

1. Sales Enablement: https://www.bricks.ai/es/sales-enablement-guia/
2. Popularidad WhatsApp: https://www.cm.com/es-es/blog/que-popular-es-whatsapp/
3. El uso del teléfono móvil en España. Programa Desconecta: https://www.programadesconecta.com/wp-content/uploads/2018/09/informe_moviles.compressed.pdf
4. WhatsApp Business: https://www.whatsapp.com/coronavirus/get-started-business?lang=es
5. WhatsApp Cloud API: https://developers.facebook.com/docs/whatsapp/cloud-api
6. Graph API: https://developers.facebook.com/docs/graph-api
7. Historia de Linux: https://ca.wikipedia.org/wiki/Hist%C3%B2ria_de_Linux
8. Linux: https://www.redhat.com/es/topics/linux/what-is-linux
9. Docker: https://aws.amazon.com/es/docker/
10. PHP: https://www.php.net/manual/es/intro-whatis.php
11. Composer: https://getcomposer.org/doc/00-intro.md
12. Nginx: https://kinsta.com/es/base-de-conocimiento/que-es-nginx/
13. Symfony: https://symfony.es/pagina/que-es-symfony/
14. Auditoria seguridad Symfony: https://symfony.com/blog/symfony2-security-audit
15. MySQL: https://www.computerweekly.com/es/definicion/MySQL
16. Sublenguajes SQL: https://learnsql.com/blog/what-is-dql-ddl-dml-in-sql/
17. HTTP: https://concepto.de/http/
18. Verbos HTTP utilizados en un sistema REST: https://codigofacilito.com/articulos/rails-verbos-http
19. Códigos de respuesta HTTP: https://blog.makeitreal.camp/el-protocolo-http/
20. URL: https://es.semrush.com/blog/que-es-una-url/
21. Plataforma: Documentación interna de CatalogPlayer
22. SAGE: https://www.sage.com/es-es/
23. JSON Schema: https://json-schema.org/overview/what-is-jsonschema
24. Crontab.guru: https://crontab.guru/
25. WhatsApp Business: https://leadsales.io/blog/diez-razones-whatsapp-business/
26. Xdebug: https://kinsta.com/es/blog/xdebug/#presentacin-de-xdebug
27. IDE: https://aws.amazon.com/es/what-is/ide/
28. WhatsApp Cloud API: https://www.postman.com/meta/workspace/whatsapp-business-platform/collection/13382743-84d01ff8-4253-4720-b454-af661f36acc2
29. Requisitos de software: https://www.northware.mx/blog/requerimientos-en-el-desarrollo-de-software-y-aplicaciones/
30. Draw.io: https://app.diagrams.net/
31. Robert C. Martin. Arquitectura Limpia, Anaya Multimedia, 2018, p. 67
32. Bertrand Meyer. Object Oriented Software Construction, Prentice Hall, 1988, p. 23
33. Robert C. Martin. Arquitectura Limpia, Anaya Multimedia, 2018, p. 93
34. Definición wrapper: https://developer.mozilla.org/es/docs/Glossary/Wrapper

35. cURL PHP: https://www.php.net/manual/en/book.curl.php
36. Estándares y recomendaciones PHP: https://www.php-fig.org/psr/
37. Definición UML: https://www.ctr.unican.es/asignaturas/mc_oo/doc/m_estructural.pdf
38. Definición repositorio git: https://desarrolloweb.com/home/git
39. Mermaid: https://mermaid.live/
40. Oauth 2.0: https://auth0.com/docs/authenticate/protocols/oauth
41. Diagrama Oauth: https://medium.com/codenx/oauth-2-0-4cddd6c7471f
42. Diagrama Oauth con Facebook: https://webstersprodigy.net/2013/05/09/common-oauth-issue-you-can-use-to-take-over-accounts/
43. Symfony releases: https://symfony.com/releases
44. PHP releases: https://www.php.net/supported-versions.php
45. Doctrine: https://www.doctrine-project.org/
46. Symfony autowiring: https://symfony.com/doc/6.4/service_container/autowiring.html
47. SKD Meta PHP: https://github.com/facebook/facebook-php-business-sdk
48. API Graph - WhatsApp/SubscribedApps: https://developers.facebook.com/docs/graph-api/reference/whats-app-business-account/subscribed_apps/
49. API Graph - Business/OwnedProductCatalogs: https://developers.facebook.com/docs/marketing-api/reference/business/owned_product_catalogs/
50. API Graph - ProductCatalog/Batch: https://developers.facebook.com/docs/marketing-api/reference/product-catalog/batch/
51. Catalog publish delay: https://developers.facebook.com/community/threads/856851922137568/
52. API Graph - ProductCatalog/CheckBatchRequestStatus: https://developers.facebook.com/docs/marketing-api/reference/product-catalog/check_batch_request_status/
53. Patrón de diseño Fluent Interface: https://designpatternsphp.readthedocs.io/es-mx/latest/Structural/FluentInterface/README.html
54. WhatsApp Cloud API - PhoneNumber/Messages: https://developers.facebook.com/docs/whatsapp/cloud-api/reference/messages
55. Suscribirse a un *webhook* de Meta: https://developers.facebook.com/docs/whatsapp/cloud-api/guides/set-up-webhooks
56. Comprobar IP de origen: https://developers.facebook.com/docs/whatsapp/cloud-api/guides/set-up-webhooks#direcciones-ip
57. Calculadora coste trabajador: https://factorialhr.es/calculadora-coste-trabajador
58. Días laborables 2024: https://www.dias-laborables.es/cuantos_dias_laborables_en_ano_2024_Andaluc%C3%ADa.htm

59. Tarifas WhatsApp por conversación: https://developers.facebook.com/docs/whatsapp/pricing?locale=es_LA#opening-conversations
60. StyleCI: https://styleci.io/
61. Definición “prompt”: Pregunta “¿Qué es un "prompt" para un modelo de lenguaje?” a ChatGPT 3.5