



Diseño y Desarrollo de un Sistema de Sincronización entre Shopify y STEL Order

Máster Universitario en Ingeniería del Software

Trabajo Fin de Máster

Autor:

Juan Bernal Miralles

Tutor/es:

Catalina Martínez Costa



Facultad
Informática
Universidad
Murcia

15 de Junio de 2025

Diseño y Desarrollo de un Sistema de Sincronización entre Shopify y STEL Order

Sistema de Integración de Entidades y Campos entre
Shopify y STEL Order

Autor

Juan Bernal Miralles

Tutor/es

Catalina Martínez Costa

Departamento de Información y Sistemas



Máster Universitario en Ingeniería del Software



Murcia, 15 de Junio de 2025



APROBADA EN JUNTA DE FACULTAD DE
6/6/2023

UNIVERSIDAD DE
MURCIA

Facultad de
Informática

Anexo V

Visto bueno de empresa a la presentación de TFG/TFM realizado por
estudiantes de la UMU en colaboración con la entidad

D./Dña. ISMAEL PUERTAS MARÍN, con DNI 48523669V,
representante de la empresa: STEL SOLUTIONS S.L.

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado en Consejo de Gobierno el día 28 de junio de 2022), así como la normativa interna para la oferta, asignación, elaboración y defensa de los Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 6 junio de 2023)

OTORGA SU VISTO BUENO A:

D./Dña. JUAN BERNAL MIRALLÉ, con DNI 49791257V, estudiante
de la titulación de Master de Ingeniería de Software de la Universidad de
Murcia y autor del TFG/TFM titulado:
Diseño y Desarrollo de un sistema de sincronización entre Shopify y Stel
para presentar el trabajo y la correspondiente memoria. Ortiz

Murcia, a de de 20

Fdo.: [Firma]

Representante de la empresa



STEL SOLUTIONS S.L.
NIF: B-73684292
C/ Mayor, 8, Puente
Tocinos, 30006, Murcia.
Tlf: +34 968 39 35 99

1/1

Campus Universitario de Espinardo. 30100 Murcia

www.um.es/informatica

Agradecimientos

El presente trabajo ha sido realizado en el marco del «PROGRAMA ENCUENTRO TF» dentro de las actividades del Campus de Excelencia Mare Nostrum 37/38, subvencionadas con la financiación de la Dirección General de Universidades e Investigación de la Consejería de Medio Ambiente, Universidades, Investigación y Mar Menor.

Es para mí un placer poder expresar mi agradecimiento a todas aquellas personas que han hecho posible la realización de este trabajo. En primer lugar, quiero agradecer a mis familiares por haberme apoyado en todo momento, especialmente a mis padres, por su paciencia e implicación en mi formación académica. También quiero agradecer a mi grupo de trabajo/amigos del Máster por un año donde he aprendido mucho con ellos.

En segundo lugar, quiero agradecer a mi tutora, Catalina Martínez Costa, por su profesionalidad y dedicación en la realización de este trabajo, así como por su paciencia y apoyo en todo momento. También quiero agradecer a mi tutor Juan Cifuentes Balsobre y al equipo de desarrollo de *STEL Order* que me han acogido desde el primer momento.

Resumen

La replicación de la información entre diferentes aplicaciones es un aspecto crucial para las empresas que buscan optimizar sus procesos y mantener una visión integral de su negocio. Muchas empresas que utilizan un Enterprise Resource Planning (ERP) también emplean uno o varios eCommerce, sistemas de mensajería u otras aplicaciones de gestión. Por ello, es fundamental que la información relevante esté sincronizada entre estas plataformas para facilitar la toma de decisiones y mejorar la eficiencia operativa.

Este Trabajo Fin de Máster documenta todo el proceso seguido para el diseño e implementación de un sistema de integración entre *Stel Order* y *Shopify*, una de las plataformas de *eCommerce* más utilizadas en la actualidad. *Stel Order* es un ERP en la nube que permite controlar distintos aspectos de un negocio, como la facturación, el inventario o la gestión de clientes. En primer lugar, se ha realizado un estudio de las dos herramientas a integrar, analizando sus características y capacidades. A partir de este análisis, se han revisado las soluciones existentes en el mercado junto con las tecnologías que se van a utilizar en el proyecto.

Se han definido los objetivos y requisitos del proyecto, junto con un plan de trabajo estructurado a través de un *RoadMap*. A continuación, se analizan las posibles arquitecturas de la aplicación, los diagramas de dominio y persistencia, y se presenta un prototipo de la aplicación. En la estructura de la aplicación se han seguido los principios de arquitectura limpia y el estilo arquitectónico *REST*, incluyendo una tabla con todas las rutas de la *API*.

En la fase de implementación se desarrolla la solución adoptada para la integración entre *Shopify* y *Stel Order*, la cual se compone de un microservicio y un plugin. El microservicio, encargado de la configuración, procesamiento de eventos y sincronización de datos, se ha implementado utilizando *Spring Boot* en *Java*. Por su parte, el plugin, responsable de la gestión de los *webhooks* y de la emisión de datos, se ha desarrollado con el *framework Remix*. Además, se ha incorporado un sistema de seguridad basado en JSON Web Token (JWT) para garantizar la autenticación y autorización de los usuarios, junto con diversas pruebas para verificar el correcto funcionamiento de la solución.

Índice general

Resumen	ix
1 Introducción	1
1.1 Proyecto General	1
1.2 Desafíos abordados por el Trabajo de Fin de Máster (TFM)	2
2 Estado del Arte	5
2.1 STEL Order	5
2.1.1 ¿Qué es?	5
2.1.2 ¿Qué existe para su integración?	6
2.2 Shopify	9
2.2.1 ¿Qué es?	9
2.2.2 ¿Qué existe para su integración?	9
2.2.3 Partner Shopify - Shopify App	12
2.3 Actualidad del Mercado	13
2.3.1 Integration Platform as a Service (iPaaS)	13
2.3.2 Conectores personalizados	15
2.3.3 Soluciones de los ERP	17
2.3.4 Análisis de las soluciones	18
2.4 Tecnologías y Herramientas	20
3 Análisis de Objetivos y Metodología	23
3.1 Objetivos	23
3.1.1 Objetivos Generales	23
3.1.2 Objetivos Específicos	24
3.2 Requisitos	25
3.2.1 Casos de Uso: Usuario	25
3.2.2 Casos de Uso: Sistema	27
3.3 Planificación Ágil del Proyecto	27
3.3.1 Scrum	27
3.3.2 RoadMap	30
4 Diseño del Trabajo Realizado	35
4.1 Diseño y Análisis de la Arquitectura del Sistema	35
4.1.1 Arquitectura Basada en Plugin	36
4.1.2 Arquitectura Basada en un Microservicio	38

4.1.3	Arquitectura Basada en Microservicios	39
4.1.4	Arquitectura Seleccionada	41
4.1.5	Arquitectura REST	42
4.2	Arquitectura Limpia	43
4.3	Diagrama Dominio	44
4.3.1	Diagrama Dominio Plantilla/Micro-Servicio Shopify	44
4.4	Diagrama de Persistencia	46
4.4.1	Diagrama de Persistencia Plugin	46
4.4.2	Diagrama de Persistencia Micro-Servicio Shopify	47
4.5	Prototipo de la Aplicación	47
5	Desarrollo e Implementación de la Aplicación	49
5.1	Desarrollo del Microservicio de Shopify	49
5.1.1	Descripción del Microservicio	49
5.1.2	Arquitectura del Proyecto	49
5.1.3	Patrones	54
5.2	Desarrollo del Plugin de Shopify	56
5.2.1	Descripción del Plugin	56
5.2.2	Plugin Backend	56
5.2.3	Plugin Frontend	58
5.2.4	Sistema de Monitoreo Log	64
5.3	Diagrama de Flujo del Sistema	64
5.4	Seguridad	66
5.4.1	Seguridad del Microservicio de Shopify	66
5.4.2	Seguridad del Plugin de Shopify	67
5.5	Pruebas	69
5.5.1	Pruebas de Integración/Contrato	69
5.5.2	Plan de Pruebas	71
6	Conclusiones y vías futuras	75
6.1	Conclusiones	75
6.2	Vías Futuras	76
6.2.1	Vías Futuras Proyecto Específico	76
6.2.2	Vías Futuras Proyecto General	76
	Bibliografía	79
	Lista de Acrónimos y Abreviaturas	85

Índice de figuras

2.1	Logo de STEL Order.	5
2.2	<i>Application Programming Interface (API)</i> de <i>STEL Order</i>	7
2.3	Logo de Shopify.	9
2.4	<i>WebHook</i> de <i>Shopify</i>	11
2.5	<i>Shopify App</i>	12
2.6	Zapier.	15
3.1	Casos de uso del usuario.	25
3.2	Tarjetas de los casos de uso del usuario.	26
3.3	Casos de uso del sistema.	27
3.4	Tarjetas de los casos de uso del sistema.	27
3.5	Historia de usuario en Jira.	30
3.6	<i>RoadMap</i> del proyecto.	31
4.1	Arquitectura Basada en Plugin.	36
4.2	Arquitectura Basada en Microservicio.	38
4.3	Arquitectura Basada en Microservicios.	39
4.4	Estructura de la arquitectura limpia.	43
4.5	Diagrama Dominio Plantilla.	44
4.6	Diagrama Dominio Microservicio Shopify.	45
4.7	Diagrama Persistencia Plugin.	46
4.8	Diagrama Persistencia Microservicio Shopify.	47
4.9	Prototipo Creación Integración.	48
4.10	Prototipo Historial Usuario.	48
5.1	Diagrama de vistas del plugin de Shopify.	60
5.2	Página de inicio de las integraciones.	61
5.3	Vista de la creación de la integración.	62
5.4	Página de configuración de la integración.	63
5.5	Vista del historial de eventos del usuario.	63
5.6	Diagrama de flujo del plugin de Shopify.	65
5.7	Diagrama de flujo del microservicio de Shopify - Parte 1.	65
5.8	Diagrama de flujo del microservicio de Shopify - Parte 2.	66
5.9	Proceso de autenticación OAuth 2.0 del plugin de Shopify.	68
5.10	Proceso de verificación de sesión del plugin de Shopify.	68

Índice de tablas

2.1	Comparación entre diferentes tipos de conectores personalizados. . . .	16
2.2	Ventajas e inconvenientes de las principales soluciones de integración. .	19
4.1	Operaciones REST del microservicio.	42
4.2	Operaciones del Plugin de Shopify.	43
5.1	Plan de pruebas de los recursos del proyecto.	71

1 Introducción

En la actualidad, las Pequeñas y Medianas Empresas (PYMES) necesitan herramientas de gestión adecuadas para seguir siendo relevantes en mercados competitivos. El uso de aplicaciones especializadas para empresas ha crecido exponencialmente en los últimos años Team [1]. Algunos de los tipos de aplicaciones [2] son *ERP*, *Customer Relationship Management (CRM)*, control horario y gestión de personal, gestión de proyectos, *eCommerce*, etc.

Existen diferentes aplicaciones que son capaces de cubrir varias de estas necesidades, pero aún así muchas empresas siguen utilizando diferentes aplicaciones para cada una de estas necesidades. Esto provoca que la información esté fragmentada y que no se pueda mantener y acceder de forma rápida y sencilla. Por lo tanto, es necesario que dichas aplicaciones tengan algún sistema capaz de integrar la información más relevante entre las diferentes herramientas.

El objetivo de este proyecto es la creación de un sistema que permita la integración de diferentes aplicaciones con un *ERP* de forma sencilla y rápida. Este sistema es un proyecto de una escala extremadamente grande, para ser abordado en su totalidad en un TFM. Por lo tanto, este trabajo se centrará en una implementación parcial del sistema, que permita la integración de un *ERP* con un *eCommerce* en concreto.

1.1 Proyecto General

Para comprender bien el alcance del TFM, es necesario entender primero el proyecto general. Éste consiste en la creación de uno/varios sistemas que permitan la integración entre *Stel Order*[3] y otras aplicaciones relevantes del mercado.

Esta integración surge de la necesidad de las empresas de contar con un sistema que mantenga actualizada y sincronizada la información entre sus diferentes aplicaciones, como un *ERP* y un *eCommerce*. Por ello, se ha decidido iniciar un ambicioso proyecto para integrar *Stel Order* con diversas plataformas de comercio electrónico, respondiendo a una de las funcionalidades más demandadas por los usuarios. Esta integración permite una gestión centralizada, eficiente y automatizada de los procesos comerciales, ofreciendo una visión global del negocio y facilitando la toma de decisiones informadas.

Esta integración debe permitir la sincronización de la información de manera bidireccional. Esto significa que cualquier cambio que se realice en una de las aplicaciones se reflejará en la otra. Por ejemplo, si se crea un nuevo producto en el *eCommerce*, este debe aparecer automáticamente en el *ERP*. De esta manera, no debe preocuparse por mantener la información actualizada en ambas aplicaciones.

1.2 Desafíos abordados por el TFM

El TFM se centrará en la integración de *Stel Order* con un *eCommerce* en concreto, *Shopify*[4]. Esta integración es una de las más demandadas por las empresas, ya que *Shopify* es una de las plataformas de *eCommerce* más utilizadas en la actualidad.

Sin embargo, en el alcance de este TFM, se ha optado por implementar inicialmente una sincronización unidireccional, en la que los datos únicamente fluyen desde *Shopify* hacia *Stel Order*. La dirección de la sincronización se ha elegido así por dos aspectos principales: es el flujo más lógico y es el más solicitado por las empresas.

De toda la información que se puede sincronizar entre ambas aplicaciones, se sincronizarán los productos, clientes, pedidos y facturas. Esto se debe a que esta información es la más relevante para las empresas y es la más usada en el día a día.

Es importante destacar que el TFM no solo se centrará en la creación de un sistema que permita la integración entre *Stel Order* y *Shopify* de manera unidireccional. Se diseñará con una arquitectura escalable y reutilizable, de modo que pueda ser utilizada en futuras integraciones con otras aplicaciones, y facilite la implementación de una sincronización bidireccional.

Resumiendo, el TFM se centrará en la creación de un sistema que permita la integración de *Stel Order* y *Shopify* de manera unidireccional. Este proyecto deberá manejar las tecnologías ofrecidas tanto por *Stel Order* como por *Shopify*, para poder realizar dicha integración de forma sencilla y rápida.

El resto del documento se divide en las siguientes secciones:

- **Sección 2 Estado del Arte:** Se realiza una investigación sobre las dos herramientas que se van a integrar, *Shopify* y *Stel Order*. Se estudian las diferentes alternativas que existen en el mercado y se describen las tecnologías que se van a utilizar en el proyecto.
 - **Sección 3 Análisis de Objetivos y Plan de Trabajo:** Se establecen los objetivos del proyecto, los casos de uso y los usuarios y se detalla la metodología de trabajo que se va a seguir.
 - **Sección 4 Diseño del Trabajo Realizado:** Se detallan las posibles arquitecturas de la aplicación, los diagramas de dominio y persistencia y un prototipo de la aplicación.
 - **Sección 5 Desarrollo e Implementación de la Aplicación:** Se explica la implementación y configuración de los diferentes componentes de la aplicación. Además, se detalla la seguridad, pruebas implementadas y un plan de pruebas para garantizar el correcto funcionamiento de la aplicación.
 - **Sección 6 Conclusiones y Vías Futuras:** Se presentan las conclusiones del proyecto y posibles vías futuras.
-

2 Estado del Arte

El estudio del estado del arte en el contexto de este proyecto requiere analizar las dos aplicaciones que se van a integrar, *STEL Order* y *Shopify*, así como examinar el mercado actual de los sistemas de sincronización. Además, se comentarán brevemente las tecnologías utilizadas para el proyecto.

De esta manera, se podrá tener una visión global de las diferentes soluciones que existen en el mercado y cómo se están abordando los problemas de sincronización de datos entre aplicaciones, con el fin de seleccionar las soluciones más adecuadas y alineadas con las necesidades del proyecto.

2.1 STEL Order

2.1.1 ¿Qué es?

Una de las herramientas a integrar en este proyecto es *STEL Order*. *STEL Order* fue desarrollado por *STEL Solutions S.L.*, una compañía tecnológica con sede en la ciudad de Murcia. *STEL Solutions* se fundó en 2010 para cubrir las necesidades de emprendedores, autónomos y pequeñas empresas [5].



Figura 2.1: Logo de STEL Order.

STEL Order es un software de gestión para autónomos y empresas que permite controlar todos los procesos de un negocio de una forma ordenada, rápida y segura. Combina funcionalidades de ERP, CRM y programa de facturación en un único software accesible desde PC, móvil y tablet [6].

STEL Solutions consta de un ecosistema de aplicaciones además de la herramienta ya mencionada [3]. Estas aplicaciones son:

- **STEL Shop:** Una plataforma de comercio electrónico que permite a las empresas crear su propia tienda online y un portal del cliente. Está completamente integrada con *STEL Order*, lo que facilita la gestión de productos, pedidos y facturación desde una única plataforma [7].
- **STEL Tiempo:** Una solución enfocada en la gestión de recursos humanos. Permite controlar las jornadas laborales, ausencias, vacaciones y fichajes del equipo. Esta herramienta ayuda a las empresas a cumplir con la normativa laboral y a optimizar la gestión del tiempo de sus empleados [8].

2.1.2 ¿Qué existe para su integración?

En este apartado se analizarán las tecnologías que ofrece esta herramienta para facilitar la integración con otras aplicaciones. Ahora bien, aunque el proyecto se implemente utilizando estas tecnologías específicas, es fundamental que el sistema se diseñe de forma desacoplada, evitando una dependencia directa de cualquier tecnología concreta. Esto se debe a que, en el ámbito del software, las tecnologías evolucionan constantemente, por lo que resulta esencial garantizar la flexibilidad y escalabilidad del sistema para permitir futuras adaptaciones o migraciones sin requerir una reestructuración completa.

Tecnologías disponibles

STEL API Publica

STEL Order ofrece una API *RESTful* que permite interactuar con la aplicación de manera programática. Esta API permite realizar operaciones *CRUD* (*POST*, *GET*, *DELETE*, *PUT*, *PATCH*) sobre los recursos disponibles en *STEL Order*, como productos, clientes, pedidos y facturas, como se muestra en la Figura 2.2. La API está diseñada para ser fácil de usar y está documentada para facilitar su integración con otras aplicaciones [9].

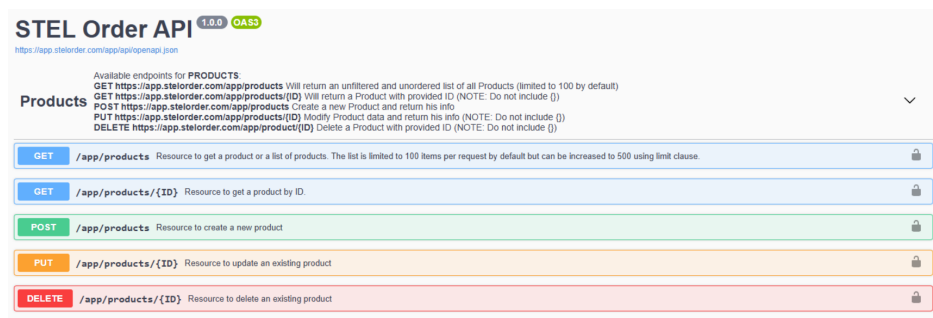


Figura 2.2: API de STEL Order.

STEL API Privada

STEL Order ofrece una API privada para los productos software de STEL Solutions S.L. Esta permite realizar operaciones CRUD más complejas que las que ofrece la API pública.

Para el desarrollo del proyecto general deberán de implementarse o modificarse funcionalidades que permitan la sincronización de entidades. Una de estas funcionalidades que ya está desarrollada es la sincronización parcial, que como su nombre indica permite obtener una gran cantidad de datos especificando las fechas de modificación de entidades.

Autenticación

Para poder utilizar la API de STEL Order, es necesario tener un API Key. Este API Key es creado por los usuarios desde la aplicación con unos permisos específicos. Estos permisos son los que permiten al usuario acceder a la API y realizar operaciones sobre los recursos disponibles.

Sincronizaciones Parciales

Esta es una funcionalidad implementada para las aplicaciones móviles de STEL Order (Android e iOS). Esta funcionalidad permite sincronizar los datos de la aplicación móvil con la base de datos de STEL Order. Se realiza de forma parcial, es decir, solo se sincronizan los datos que han sido modificados desde la última sincronización o fecha especificada. Con esta funcionalidad, se puede obtener una gran cantidad de datos en una llamada para sincronizar información con Shopify u otras aplicaciones.

Propiedad ExternalId

Actualmente STEL Order permite tener un campo adicional a las entidades denominado **ExternalId**. Este campo permite almacenar un identificador externo para cada entidad, lo que facilita la integración con otras aplicaciones. Este campo es muy útil para poder relacionar los datos de STEL Order con los datos de otras aplicaciones, como por ejemplo Shopify.

Ahora bien, este campo deberá ser modificado para poder almacenar una lista de identificadores externos, ya que en la actualidad solo permite almacenar un único identificador externo. Esto es un problema, ya que si se quiere integrar *STEL Order* con varias aplicaciones, es necesario tener un identificador externo para cada aplicación. También deberá implementarse un campo adicional para poder almacenar el nombre de la aplicación a la que pertenece el identificador externo.

Tecnologías no disponibles

STEL Webhooks

STEL Order actualmente no ofrece la posibilidad de crear *Webhooks* [10] para recibir notificaciones en tiempo real sobre eventos específicos. Sin embargo, esta funcionalidad deberá ser implementada en el futuro, ya que es una funcionalidad muy demandada por los usuarios de la aplicación.

Esto es muy útil para la integración con otras aplicaciones, ya que permite recibir notificaciones en tiempo real sobre eventos específicos, como la creación de un nuevo pedido o la modificación de un producto. Además, de que es mas eficiente que realizar consultas periódicas a la API para comprobar si ha habido cambios en los datos o utilizar la sincronización parcial.

STEL GraphQL

La arquitectura de *Stel Order* no permite el uso de *GraphQL*[11] para realizar consultas y modificaciones de las clases de dominio de la aplicación sino que se basa en una arquitectura *RESTful*. Esto es un inconveniente, ya que *GraphQL* ofrece una serie de ventajas frente a Representational State Transfer (REST), como la posibilidad de realizar consultas más complejas y obtener solo los datos necesarios.

Por ejemplo, si solo nos interesa el nombre del cliente y su correo electrónico, con *GraphQL* podemos realizar una consulta que nos devuelva solo esos datos, mientras que con REST tendríamos que realizar una consulta que nos devuelva todos los datos del cliente y luego filtrar los datos que nos interesan.

2.2 Shopify

2.2.1 ¿Qué es?

Shopify fue fundada en 2006 por Tobias Lütke, Daniel Weinand y Scott Lake, quienes originalmente buscaban una solución para vender equipos de snowboard online. Al no encontrar ninguna plataforma que cumpliera con sus expectativas, decidieron crear su propio software. Desde entonces, *Shopify* ha experimentado un crecimiento impresionante, convirtiéndose en una de las principales plataformas de comercio electrónico del mundo [12].



Figura 2.3: Logo de Shopify.

Shopify está diseñada para ser intuitiva y accesible, pero también ofrece la potencia y flexibilidad que las empresas en crecimiento necesitan. Algunas empresas que han encontrado el éxito utilizando Shopify como su plataforma de comercio electrónico son Gymshark, Allbirds y Heinz [13].

2.2.2 ¿Qué existe para su integración?

Esta herramienta ofrece diferentes formas para ser integrada con diferentes aplicaciones. Es importante diferenciar la comunicación entre el sistema de sincronización con *Shopify* y *Shopify* con el sistema de sincronización. Para estos intercambios de datos, Shopify dispone de tres sistemas: *REST*, *GraphQL* y *WebHook*. Se manejará en dos vías: una para enviar información y otra para recibirla.

Envío de datos

Shopify ofrece dos mecanismos para la comunicación: la herramienta *REST Admin API* y *GraphQL Admin API*. Pero actualmente ya no se le da soporte a *REST Admin API* porque se ha migrado todo a *GraphQL Admin API*.

- **REST Admin API:** ofrece una *API REST* con toda la funcionalidad disponible relacionada con los Customers, Inventory, Orders, Product, etc. Todos los *endpoints* que ofrecen necesitan autenticación a través de un *Access Token*. Sin embargo, esta solución desde el 1 de octubre de 2024 está obsoleta y debe ser migrada a *GraphQL* [14].
-

- **GraphQL Admin API:** ofrece la posibilidad de utilizar un cliente de GraphQL para acceder y modificar la información de la que dispone Shopify. También cuenta con las mismas funcionalidades disponibles que se encontraban en su API REST, y necesita un Access Token para el proceso de autenticación [15]. Esta tecnología ofrece una serie de mejoras frente a la tecnología API REST para este tipo de comunicación.

Recepción de datos

En este caso existen tres posibles soluciones. Las dos soluciones anteriores, *Rest Admin API* y *GraphQL Admin API*, permiten consultar la información de Shopify periódicamente. Al igual que antes, no se recomienda el uso de *Rest Admin API* ya se está migrando a *GraphQL Admin API*. La otra opción es el uso de *Webhooks* para suscribirse a eventos.

WebHook: nos permite suscribirnos a un número determinado de eventos del dominio que sucedan dentro de Shopify. Estos se crean indicando la *Uniform Resource Locator (URL)* de *callback* que responde al evento, junto con los datos correspondientes en un campo llamado *data*.

Sus principales ventajas frente a otras soluciones son:

- **Simplificación del flujo de trabajo:** Permiten a las aplicaciones recibir notificaciones en tiempo real sobre eventos específicos, lo que simplifica el flujo de trabajo al eliminar la necesidad de realizar consultas periódicas para comprobar si ha habido cambios en los datos.
- **Mejora del rendimiento:** Solo se intercambia información cuando se producen eventos específicos, lo que reduce la carga en el sistema y mejora el rendimiento general de la aplicación.
- **Tiempo real:** Permiten recibir notificaciones en tiempo real, lo que significa que las aplicaciones pueden reaccionar de inmediato a los cambios en los datos, mejorando la eficiencia y la capacidad de respuesta del sistema.

El principal inconveniente de los *Webhooks* es su limitada personalización: solo permiten reaccionar a los eventos disponibles en su suscripción y no ofrecen control sobre la información específica que se desea recibir.

Entre las características que se han comentado, resulta de mayor interés el uso de *Webhooks* para abordar la comunicación entre *Shopify* y el sistema de sincronización. El propio *Shopify* [16] en su documentación recomienda el uso de *Webhooks* para la comunicación entre *Shopify* y otras aplicaciones, como se muestra en la Figura 2.4.

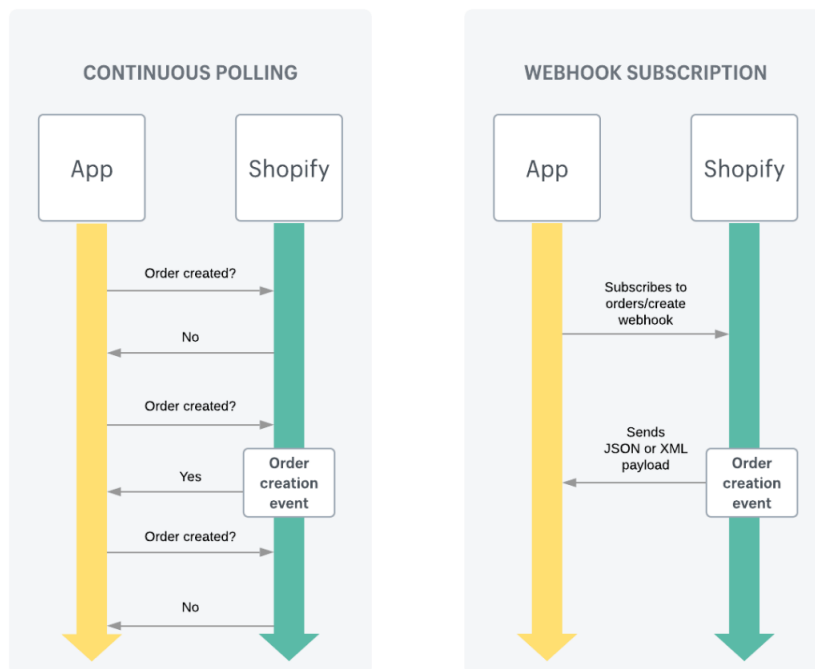


Figura 2.4: *WebHook* de *Shopify*.

2.2.3 Partner Shopify - Shopify App

Todas estas herramientas y tecnologías que ofrece *Shopify* son muy interesantes, pero hace falta un sistema que permita crear aplicaciones para poder integrar *Shopify* con otras aplicaciones. Para ello, *Shopify* ofrece la posibilidad de crear aplicaciones para su plataforma a través de su programa de *Partners*. Estas aplicaciones pueden abordar una gran cantidad de funcionalidades, desde la gestión de productos y pedidos, generación de facturas, hasta la integración con otras aplicaciones como se muestra en la Figura 2.5.

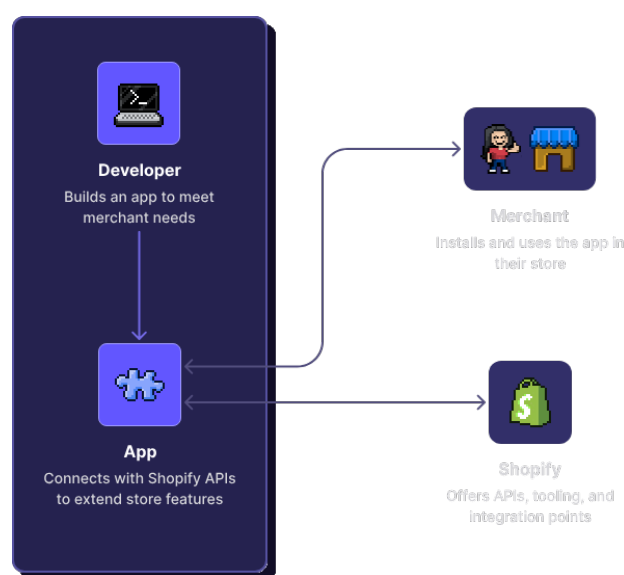


Figura 2.5: *Shopify App*.

Esta herramienta es gratuita y permite a los desarrolladores crear tiendas de prueba ilimitadas para realizar pruebas a sus aplicaciones. Además, *Shopify*, con esta funcionalidad, ofrece todos los *End-Point* de la plataforma, una gran cantidad de documentación y una serie de plantillas y librerías para facilitar el desarrollo de aplicaciones.

Shopify establece una serie de requisitos y prohibiciones para publicar aplicaciones en su plataforma, con el objetivo de garantizar su calidad y seguridad. Toda esta información se encuentra detallada en su documentación oficial [17].

A continuación, las tecnologías que *Shopify* recomienda son las siguientes [18]:

- **Remix**: un *framework* de *React* que permite crear aplicaciones web modernas y rápidas.
- **Prisma**: un Object-Relational Mapping (ORM) que permite interactuar con bases de datos de manera sencilla y eficiente.

- **Polaris:** una librería de *JavaScript* que permite crear interfaces de usuario modernas y atractivas.

2.3 Actualidad del Mercado

El mercado de la sincronización de datos entre aplicaciones está en constante evolución. Cada vez más empresas buscan soluciones que les permitan integrar sus sistemas y aplicaciones de manera eficiente y efectiva. Existen diferentes tipos de soluciones en el mercado, desde soluciones personalizadas hasta soluciones *iPaaS* [19].

Después de analizar las diferentes soluciones que existen, se describirán dos ERP para la sincronización con Shopify. Estas soluciones son muy interesantes, ya que nos ofrecen una visión de cómo se están abordando los problemas de sincronización de datos entre aplicaciones.

2.3.1 iPaaS

Una *Integration Platform as a Service (iPaaS)* es un conjunto de herramientas y soluciones de autoservicio basadas en la nube. Se utilizan para integrar datos de varias aplicaciones alojadas en distintos entornos informáticos [19].

Las características principales de una *iPaaS* son las siguientes:

- **Conectores y adaptadores:** Las *iPaaS* suelen ofrecer una amplia gama de conectores y adaptadores para facilitar la integración con diferentes aplicaciones y servicios.
 - **Desarrollo low-code y no-code:** Muchas *iPaaS* permiten a los usuarios crear integraciones utilizando interfaces gráficas y herramientas de arrastrar y soltar, lo que reduce la necesidad de codificación manual.
 - **Transformación y mapeo de datos:** Las *iPaaS* suelen incluir herramientas para transformar y mapear datos entre diferentes formatos y estructuras.
 - **Procesamiento en tiempo real y por lotes:** Las *iPaaS* pueden procesar datos en tiempo real o por lotes, lo que permite a las empresas elegir la opción que mejor se adapte a sus necesidades.
-

Esta es una posible solución para el desarrollo del proyecto, ya que permite integrar diferentes aplicaciones de manera sencilla y rápida. Sin embargo, tiene diferentes inconvenientes que hacen que no sea la mejor opción. Estos son los principales inconvenientes de las *iPaaS*:

- **Coste elevado:** Suelen tener un coste elevado, lo que puede ser un inconveniente para la escalabilidad del sistema.
- **Limitaciones en la personalización:** Aunque las *iPaaS* ofrecen herramientas de desarrollo *low-code* y *no-code*, pueden tener limitaciones en la personalización de las integraciones, lo que puede ser un inconveniente para empresas con necesidades específicas.
- **Dependencia del proveedor:** Se estaría dependiendo del proveedor para el mantenimiento y la actualización de la plataforma, lo que puede ser un inconveniente si el proveedor no cumple con sus expectativas.

Algunas de las *iPaaS* más conocidas en el mercado son *Latenode*, *Zapier*, *Boomi*, *Jitterbit*, y *IBM Connect* [20]. A continuación se describen algunas de ellas.

Zapier

Zapier [21] es una herramienta de automatización de procesos *low-code* que actúa como un intermediario entre dos o más aplicaciones, permitiendo que se comuniquen entre sí sin la necesidad de saber programación. Con *Zapier*, los usuarios pueden crear "Zaps", que son flujos de trabajo automatizados que conectan dos o más aplicaciones [22].

Una ilustración de cómo funciona *Zapier* se puede ver en la Figura 2.6. En esta Figura se puede ver cómo *Zapier* permite conectar diferentes aplicaciones y servicios para automatizar tareas y flujos de trabajo.

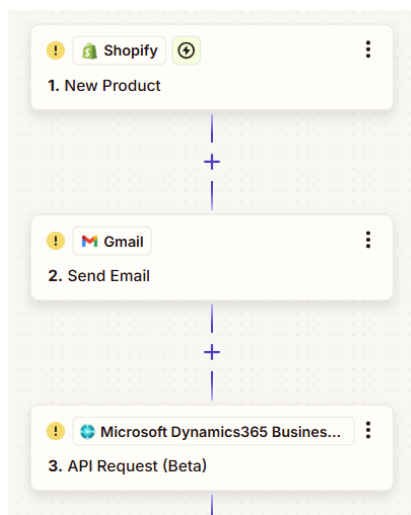


Figura 2.6: Zapier.

Como se puede ver en la Figura 2.6, cada vez que se produce la creación de un nuevo pedido en una cuenta de *Shopify*, se envía un correo a una cuenta de *Gmail* y se realizará una llamada a la API de *Business Central*. Este sería un ejemplo de caso de uso del proyecto.

Zapier no puede ser un ejemplo de sincronización para este proyecto, ya que el número de "Zaps" que debería crearse para la sincronización de datos entre *STEL Order* y *Shopify* sería elevado. Esto es debido a que cada nueva integración entre dos cuentas de *Shopify* y *STEL Order* tendría x "Zaps" para cada una de las entidades que se quieran sincronizar. Por lo tanto, el coste de la solución sería elevado y no sería escalable.

Realmente *Zapier* y las *iPaas* no están diseñadas para la sincronización total de aplicaciones, sino para la automatización de pequeñas tareas y flujos de trabajo específicos entre aplicaciones.

2.3.2 Conectores personalizados

Muchas empresas han optado por desarrollar sus propios conectores personalizados para integrar aplicaciones y sistemas. Estos conectores pueden ser desarrollados internamente o por terceros y pueden ofrecer una mayor flexibilidad y personalización

en comparación con las soluciones de *iPaas*. Sin embargo, el desarrollo de conectores personalizados puede ser costoso inicialmente y requerir un mayor esfuerzo de mantenimiento. Existen diferentes tipos de conectores personalizados, algunos de ellos son:

- **Integraciones punto a punto:** establece conexiones directas y específicas entre dos aplicaciones, transfiriendo datos de una a otra, opcionalmente con transformaciones in situ.
- **Middleware:** actúa como capa intermedia entre dos (o más) aplicaciones, ofreciendo servicios comunes de mensajería, gestión de transacciones, autenticación y enrutamiento de mensajes.
- **Conector nativo:** es un componente de software proporcionado por el proveedor de una aplicación que permite su integración directa con otras aplicaciones.
- **Enterprise Service Bus (ESB):** extiende el concepto de middleware al proporcionar una plataforma centralizada con capacidades avanzadas de enrutamiento, transformación de mensajes, orquestación y monitorización de servicios.

A continuación, en la tabla 2.1 se puede ver una comparativa entre las diferentes soluciones de integración de aplicaciones. Con esta comparativa se pueden apreciar los puntos fuertes y débiles de cada una de las soluciones.

Puntuación Alto Medio Bajo				
	Integraciones punto a punto	Middleware	Conector nativo	ESB
Escalabilidad	Bajo	Alta	Alta	Alta
Personalización	Alta	Media	Media	Alta
Mantenimiento	Alta	Bajo	Bajo	Bajo
Flexibilidad	Bajo	Alta	Media	Alta
Monitoreo y Fiabilidad	Bajo	Medio	Alto	Alto
Coste	Bajo	Medio	Medio	Alto

Tabla 2.1: Comparación entre diferentes tipos de conectores personalizados.

2.3.3 Soluciones de los ERP

Más adelante se describen las soluciones de integración ofrecidas por los diferentes ERP que existen en el mercado. A continuación, se va a analizar la solución propuesta por uno de los ERP más conocidos en el mercado, *Microsoft Dynamics 365 Business Central* [23] y la solución de un ERP español más pequeño como es *Holded* [24].

Microsoft Dynamics 365 Business Central

Microsoft Dynamics 365 Business Central es un software de gestión empresarial que permite a las empresas gestionar sus finanzas, ventas, servicios y operaciones. Este software no cuenta con un conector genérico para la integración de aplicaciones, sino que ofrece diferentes conectores personalizados para la integración con diferentes aplicaciones. Estos conectores son desarrollados por *Microsoft* y permiten la integración parcial con diferentes aplicaciones.

Microsoft y *Shopify* tienen un acuerdo para integrar las dos herramientas a través de un conector nativo diseñado en 2022. El conector permite conectar una misma instancia de *Business Central* con una o muchas instancias de *Shopify*. Actualmente solo está disponible para versiones en la nube, no en local.

Los elementos disponibles para la sincronización son:

- **Sincronización artículos/productos:** Permite sincronizar de manera bidireccional los productos entre *Shopify* y *Business Central*.
- **Sincronización de inventario:** Permite sincronizar los niveles de inventario entre *Shopify* y *Business Central*.
- **Sincronización clientes:** Permite sincronizar de manera bidireccional los clientes entre *Shopify* y *Business Central*.
- **Sincronización pedidos:** Permite importar los pedidos desde *Shopify* a *Business Central*.

Holded

Holded es un ERP español que ofrece una solución de gestión empresarial para PY-MES. Esta herramienta ofrece una solución de integración con *Shopify* a través de un conector nativo. Este conector es un plugin de *Shopify* que se puede instalar en la tienda de *Shopify* de manera gratuita.

Al instalar el plugin, se inicia sesión con una cuenta de *Holded* registrada y se despliega automáticamente un modal donde se pueden editar la configuración del conector. Esta configuración permite sincronizar los siguientes elementos:

- **Sincronización de Productos:** Permite sincronizar de manera unidireccional los productos entre *Shopify* y *Holded*. Estos son vinculados mediante el *SKU* de *Shopify* y no se sincroniza toda la información de los productos, sino solo la información básica como nombre, descripción, precio, impuestos, código de barras y coste del producto.
- **Sincronización de Stock:** Tiene la opción de sincronizar el *stock* de los productos de manera bidireccional. Para ello, es necesario que el usuario tenga activada y configurada la opción de sincronización de almacenes.
- **Sincronización de Ventas:** Permite importar los pedidos desde *Shopify* a *Holded*. Esta sincronización es unidireccional y transforma los pedidos de *Shopify* a facturas de *Holded*, si están pagados, y a pedidos de venta, si no están pagados. Para poder sincronizar los pedidos, es necesario que los usuarios introduzcan información sobre el tipo de pago.
- **Sincronización de Almacenes:** Permite sincronizar los almacenes de *Shopify* y *Holded*.

Esta solución es muy interesante, ya que permite sincronizar parte de los datos entre *Shopify* y *Holded* de manera sencilla y rápida. Claro que es una solución muy limitada, ya que no admite sincronizar algunos datos interesantes como los clientes y la mayoría de los datos no son bidireccionales. Además, no da la facultad a los usuarios de seleccionar qué campos de los datos quieren sincronizar.

2.3.4 Análisis de las soluciones

Antes de seleccionar la solución de integración más adecuada para el proyecto, es fundamental analizar las ventajas e inconvenientes de las principales alternativas existentes en el mercado. En la siguiente tabla se resumen los puntos fuertes y débiles de cada opción:

Solución	Ventajas	Inconvenientes
iPaaS	<ul style="list-style-type: none"> - Integración rápida y sencilla. - Desarrollo low-code/no-code. 	<ul style="list-style-type: none"> - Coste elevado, especialmente a gran escala. - Limitaciones en la personalización. - Dependencia del proveedor.
Integraciones punto a punto	<ul style="list-style-type: none"> - Sencillez de implementación. - Bajo coste inicial. 	<ul style="list-style-type: none"> - Escalabilidad limitada. - Baja flexibilidad ante cambios.
Middleware	<ul style="list-style-type: none"> - Mejora la flexibilidad y el mantenimiento. - Permite gestión centralizada de mensajes y autenticación. 	<ul style="list-style-type: none"> - Puede requerir personalización adicional.
Conector nativo	<ul style="list-style-type: none"> - Integración directa y optimizada. - Alto nivel de fiabilidad y soporte. 	<ul style="list-style-type: none"> - Escalabilidad limitada.
ESB	<ul style="list-style-type: none"> - Alta escalabilidad y flexibilidad. - Capacidades avanzadas de orquestación y monitorización. 	<ul style="list-style-type: none"> - Gran coste económico y técnico.

Tabla 2.2: Ventajas e inconvenientes de las principales soluciones de integración.

Tras analizar las herramientas iPaaS, los diferentes conectores personalizados y las soluciones de los ERP, el proyecto adoptará una solución de conectores nativos junto a *middleware*. Esta decisión se debe a que las soluciones de iPaaS no son escalables y no ofrecen la personalización de los datos a sincronizar. Por otro lado, la opción de desarrollar un ESB resulta demasiado costosa y no es necesaria para el alcance del proyecto, como demuestran las soluciones implementadas por otros ERP.

2.4 Tecnologías y Herramientas

En esta sección se enumeran las tecnologías y herramientas que se van a utilizar en el desarrollo del proyecto, así como algunas alternativas que podrían considerarse como mejoras futuras.

Las tecnologías y herramientas que se van a utilizar en el desarrollo del proyecto son las siguientes:

- **Remix:**[25] es un *framework* web integral creado sobre *React*. Esta es la mejor opción para el desarrollo de una aplicación de *Shopify*.
- **Spring Boot:**[26] es un *framework* de código abierto para el desarrollo de aplicaciones en *Java*. Este se ha convertido en el más popular para el desarrollo de aplicaciones web en *Java*. Este es el *framework* seleccionado por la empresa para el desarrollo del proyecto. Esta decisión se debe a que la empresa ya tiene experiencia en el desarrollo de aplicaciones con esta tecnología y es muy potente y escalable.
- **MySQL:**[27] es un sistema de código abierto de administración de base de datos, desarrollado y soportado por *Oracle*. Es la base de datos seleccionada para el desarrollo del proyecto. Y ello es así porque todo el entorno de *Stel* utiliza *MySQL* como base de datos principal y es la base de datos más utilizada en el mundo.
- **Prisma ORM:**[28] es un ORM de código abierto de nueva generación. Permite interactuar con bases de datos de manera sencilla y eficiente. *Prisma* es otra de las tecnologías recomendadas por *Shopify* para el desarrollo de aplicaciones junto al uso de *MySQL* como base de datos.
- **Visual Studio Code:** para el desarrollo de la aplicación de *Shopify* se va a utilizar *Visual Studio Code* debido a que tiene soporte para *Remix*.
- **IntelliJ IDEA:** Para el desarrollo de la aplicación basada en *Spring Boot* se empleará *IntelliJ IDEA*. Este entorno integrado de desarrollo, creado por *JetBrains*, está especialmente diseñado para *Java* y es ampliamente reconocido como una de las mejores opciones para el desarrollo de aplicaciones *Java* debido a su robustez, funcionalidades avanzadas y soporte para *frameworks* modernos.

La elección de estas tecnologías no ha sido arbitraria, sino que en gran parte ha estado condicionada por el entorno y las aplicaciones ya existentes en la empresa. Por ejemplo, *Spring Boot* y *MySQL* ya forman parte del stack tecnológico habitual de *Stel Order*, lo que garantiza compatibilidad, aprovechamiento del conocimiento interno y una integración más eficiente con los sistemas actuales. Asimismo, el uso de *Remix* y *Prisma ORM* responde tanto a las recomendaciones de *Shopify* como a su adecuación

técnica para los requerimientos del proyecto.

Además de estas tecnologías, también se va a utilizar las herramientas de *Atlassian* [29] para la gestión del proyecto. Estas herramientas son:

- **Jira:** es una aplicación web que con el tiempo se ha convertido en el estándar del mercado en las áreas de gestión de proyectos, gestión de tareas y gestión de errores [30]. Con esta herramienta se pueden gestionar los diferentes tickets del proyecto y asignar tareas a los diferentes miembros del equipo.
 - **Confluence:** es una herramienta de colaboración en equipo e intercambio de conocimientos. Está diseñada para ayudar a los equipos a crear, compartir y gestionar la documentación y el conocimiento de sus proyectos en un único espacio [31].
 - **Bitbucket:** es una plataforma de alojamiento de repositorios de código basada en la nube que proporciona a los equipos de desarrollo un entorno colaborativo en el que poder almacenar, organizar y gestionar su código fuente [32].
-

3 Análisis de Objetivos y Metodología

En este apartado se analizarán los objetivos y requisitos del proyecto y se presentará un plan de trabajo detallado para su desarrollo del TFM a través de un *RoadMap*.

3.1 Objetivos

Esta sección se dividirá en dos partes. En la primera parte se presentarán los objetivos generales de los proyectos de integraciones. En la segunda parte se mostrarán los objetivos específicos del TFM.

3.1.1 Objetivos Generales

Los objetivos generales del proyecto de integración entre *Stel Order* y diferentes aplicaciones son los siguientes:

- **OBG1:** Analizar el sistema y definir requisitos.
- **OBG2:** Diseñar la arquitectura del sistema.
- **OBG3:** Diseñar la arquitectura de la plantilla.
- **OBG4:** Desarrollar la plantilla de integración.
- **OBGN.1:** Investigar y estudiar la herramienta **N** a integrar.
- **OBGN.2*:** Diseñar la arquitectura que ofrece la herramienta **N** a integrar.
- **OBGN.3:** Desarrollar el microservicio partiendo de la plantilla de integración.
- **OBGN.4*:** Desarrollar plugin o sistema que permita la integración con la herramienta **N**.
- **OBGN.5:** Realizar pruebas de integración con la herramienta **N**.
- **OBGN.6:** Desarrollar la documentación del microservicio.

Los nombres de los objetivos generales están compuestos por el prefijo **OBG**, que significa objetivo general, seguido de un número. Los objetivos que empiezan por **OBGN** significa los pasos que se deben seguir para la integración de una herramienta **N**. Algunos de estos pasos son obligatorios para cada herramienta, como el **OBGN.1** y **OBGN.5**, pero otros son opcionales y dependen de la herramienta a integrar. Por ejemplo, el **OBGN.2** es obligatorio para algunas herramientas, pero no para otras.

No se han introducido objetivos más específicos sobre despliegue y pruebas porque la gran parte de estos objetivos son abordados por el equipo de Quality Assurance (QA) y otros compañeros encargados de estas tareas.

3.1.2 Objetivos Específicos

Los objetivos que se abordarán en el TFM son los siguientes:

- **OBS1:** Investigar y estudiar la herramienta *Shopify* para la integración con *Stel Order*.
- **OBS2:** Diseñar el *plugin* de integración con *Shopify*.
- **OBS3:** Desarrollar el *plugin* de integración con *Shopify*.
- **OBS4:** Desarrollar el microservicio de integración con *Shopify*.
- **OBS5:** Realizar pruebas de integración con *Shopify*.
- **OBS6:** Desarrollar la documentación del microservicio de integración con *Shopify*.

Es importante destacar que, además de los objetivos específicos del TFM, también se han desarrollado los 4 primeros objetivos generales. Se ha hecho para poder tener una base sólida sobre la que desarrollar el TFM y para poder reutilizar el trabajo realizado para futuras integraciones con otras aplicaciones.

Los objetivos generales han sido desarrollados de forma colaborativa con el equipo de desarrollo de *Stel Order*, cuya experiencia ha sido clave en la definición de la arquitectura y en la creación de la plantilla base para futuras integraciones. Este trabajo conjunto ha permitido establecer una base sólida y reutilizable. En cambio, los objetivos específicos del TFM han sido abordados íntegramente por el autor, por lo que se profundizará especialmente en esta parte del desarrollo a lo largo del documento.

3.2 Requisitos

Se ha elegido ilustrar los requisitos de la herramienta creada por medio de casos de uso en lenguaje natural para facilitar la comprensión y comunicación entre todas las partes interesadas [33]. En primer lugar, citaremos dos tipos de actores:

- **Usuario:** Representa a un usuario que va a integrar su Shopify con *Stel Order* desde el plugin de *Shopify*. El usuario no tiene conocimientos técnicos y solo quiere integrar su tienda de *Shopify* con *Stel Order*. Este usuario necesita dos cuentas, una de *Shopify* y otra de *Stel Order*, o puede crearlas en el proceso.
- **Sistema:** Corresponde al sistema de integración entre *Stel Order* y *Shopify*. Es el actor responsable de recibir los *Webhooks* enviados por *Shopify* y notificar los eventos correspondientes al sistema *Stel Order*, permitiendo así la sincronización entre ambas plataformas.

3.2.1 Casos de Uso: Usuario

Esta sección presenta los casos de uso del usuario, con su descripción, prioridad y puntos de historia. La figura 3.1 los ilustra.

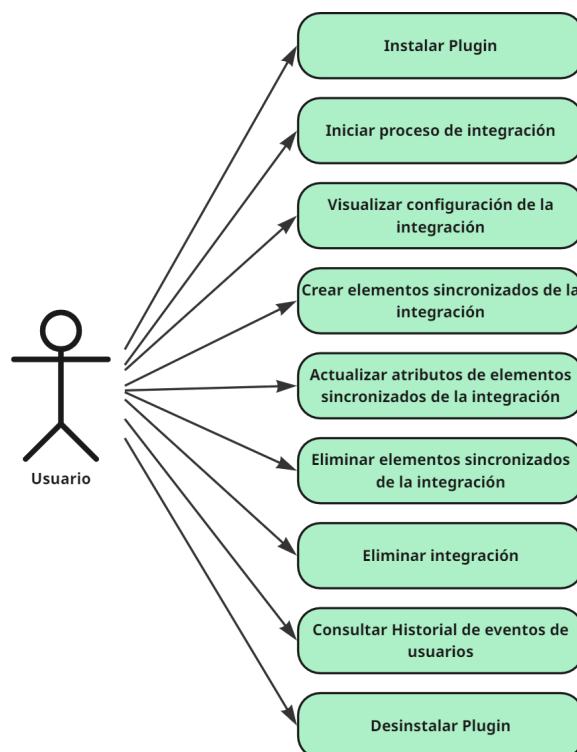


Figura 3.1: Casos de uso del usuario.

Las tarjetas de requisitos de los anteriores casos de uso del usuario son las que se muestran en la figura 3.2. Se puede ver la prioridad que representa cada uno de estos CDU y los puntos de historia asociados. También se puede observar como el CDU 1 (Instalar el PPlugin) fue la prioridad inicial, mientras que el CDU 7 (Consultar historial de eventos de usuarios) resultó ser una de las ultimas priorizadas.

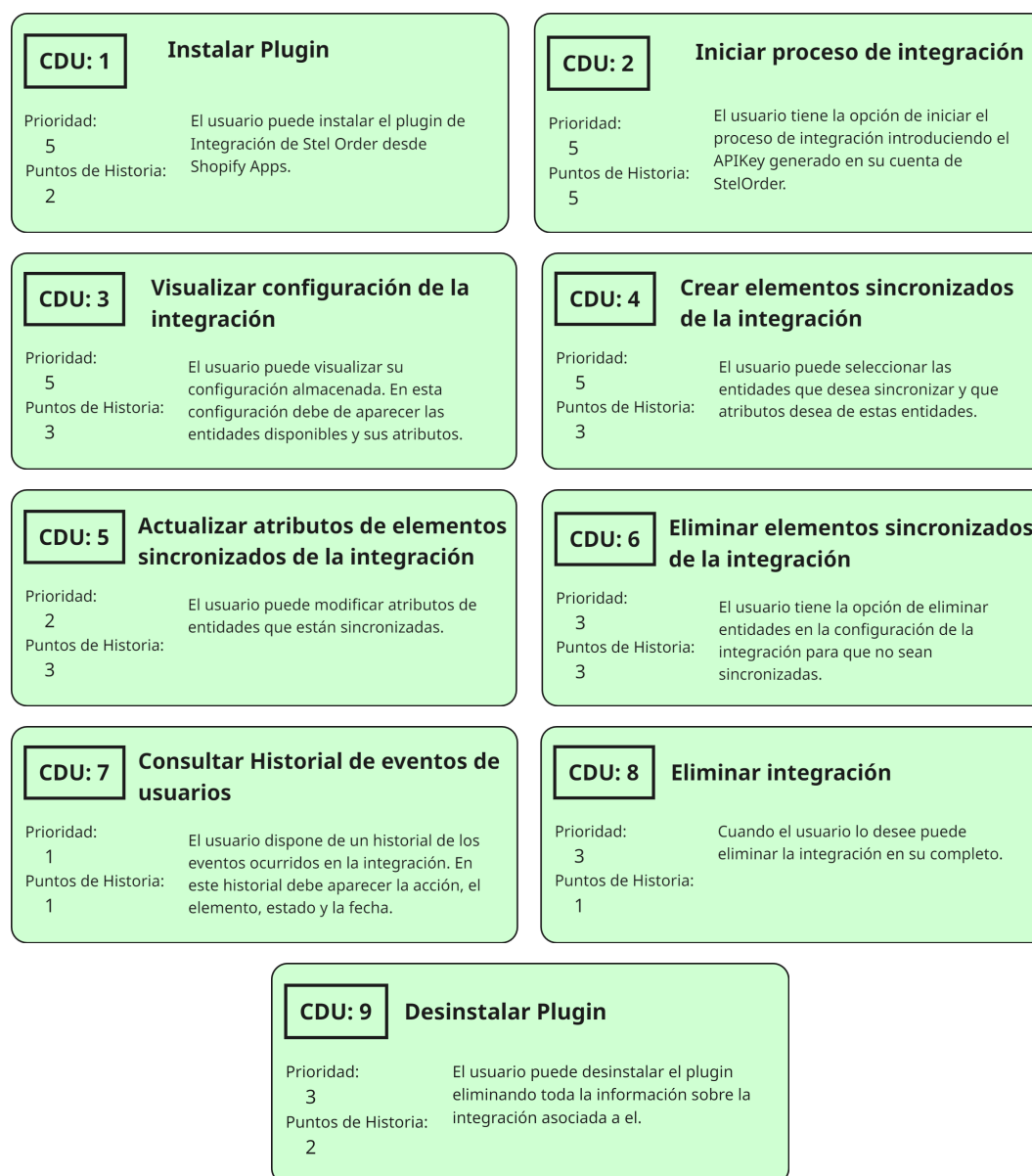


Figura 3.2: Tarjetas de los casos de uso del usuario.

3.2.2 Casos de Uso: Sistema

Para el desarrollo de este TFM este actor únicamente tiene un caso de uso (véase Figura 3.3).

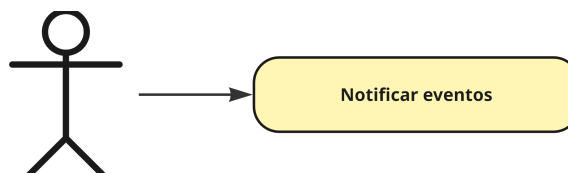


Figura 3.3: Casos de uso del sistema.

A continuación se muestra la tarjeta de requisito del anterior caso de uso.

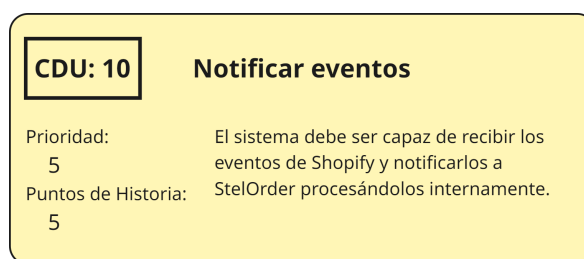


Figura 3.4: Tarjetas de los casos de uso del sistema.

3.3 Planificación Ágil del Proyecto

El desarrollo del TFM se ha realizado siguiendo la metodología ágil de *Scrum* [34]. Además de la metodología *Scrum*, se ha utilizado la herramienta de gestión de proyectos Jira y la herramienta *Confluence* para desarrollar un *RoadMap* del proyecto que ayude a la gestión del mismo.

3.3.1 Scrum

Scrum es un marco de gestión de proyectos ágil que ayuda a los equipos a estructurar y gestionar su trabajo a través de un conjunto de valores, principios y prácticas [35]. Se ha modificado el marco de trabajo de *Scrum* para adaptarlo a las necesidades del proyecto.

Eventos de Scrum

Scrum se basa en un conjunto de eventos que permiten a los equipos planificar, ejecutar y revisar su trabajo. Estos eventos son: *sprint*, la reunión de planificación, la reunión diaria, la revisión del *sprint* y la retrospectiva del *sprint*.

En este caso se ha suprimido el evento de *sprints* para el proyecto. Se ha trabajado de forma continua. Esto se ha realizado debido a que el equipo de desarrollo está integrándose con la gestión ágil y todavía no está preparado para el tratamiento de *Spring*. Tampoco se ha hecho uso de la retrospectiva del proyecto. Está proyectado realizarlo en el momento en que se acabe una primera versión.

Por otra parte, se han realizado diferentes eventos de *sprint*:

- **Reuniones de planificación:** Se han realizado reuniones de planificación al inicio del proyecto para definir los objetivos y requisitos. En estas reuniones se han definido los objetivos generales y específicos, así como los requisitos del sistema. Esta reunión se realizó junto con el *product owner* del proyecto.
- **Reuniones diarias:** Se han realizado reuniones diarias para monitorizar la evolución del proyecto y resolver dudas y bloqueos. Estas ocurren de forma diaria y han tenido una duración aproximada de 25 minutos. Los participantes son el equipo de desarrollo y el *scrum mánager*.
- **Reuniones de revisión:** Se han desarrollado reuniones de revisión al final de cada semana para revisar el avance del proyecto y realizar ajustes en la planificación. En estas reuniones participan los *product owners*, *scrum manager* y el equipo de desarrollo con un tiempo aproximado de 1 hora.

Artefactos de Scrum

Los artefactos de *Scrum* son los elementos que se utilizan para gestionar el trabajo del equipo. Estos artefactos son: el *backlog* del producto, el *backlog* del *sprint* y el incremento.

- **Backlog del producto:** Se corresponde con la lista priorizada de los requisitos y funcionalidades del producto. En este caso, el *backlog* del producto se ha realizado con la supervisión del *product owner* y se ha utilizado para gestionar el trabajo del equipo.
 - **Backlog del sprint:** es una lista de los requisitos y funcionalidades que se van a desarrollar a lo largo de la primera *release* del proyecto. Estos han sido desarrollados por el equipo de trabajo con la supervisión del *scrum master*.
-

- **Incremento:** la presentación de una parte del producto terminada y lista para usarse permite comprobar si se está generando valor. En este caso, el incremento se ha utilizado para presentar el avance del proyecto y recibir *feedback* por parte del *product owner* y de los demás integrantes del equipo de trabajo.

Kanban e Historias de Usuario

Kanban es una forma de ayudar a los equipos a encontrar un equilibrio entre el trabajo que necesitan hacer y la disponibilidad de cada uno de sus miembros del equipo [36]. Esta técnica ha sido muy útil para el desarrollo del TFM, permitiendo gestionar el trabajo de forma visual y adaptarse a los cambios de forma rápida y eficiente.

La herramienta *Jira* otorga la posibilidad de gestionar el trabajo utilizando la técnica *Kanban*. En este caso, se ha generado una épica para el proyecto genérico y se han ido añadiendo historias de usuario a la épica. Estas historias de usuario se han ido gestionando utilizando la técnica *Kanban* por los miembros del equipo. La unidad más pequeña de trabajo no ha sido la historia sino que éstas se dividen en tareas para mayor especificación.

Algunas de estas tarjetas no son historias de usuario, sino que son historias técnicas. Estas son aquellas que no son visibles para el usuario final, pero que son necesarias para el correcto funcionamiento del sistema.

Estas historias de usuario y sus subtareas son seleccionadas por el equipo de desarrollo para ser implementadas. Por cada historia de usuario se crea una rama en el repositorio de *Bitbucket* y se desarrolla la historia de usuario en esa rama. Los posibles estados de las historias de usuario son las siguientes *Backlog*, *Tarea por hacer*, *WIP*, *Committed*, *Revisar*, *Control de calidad*, *Finalizada* y *Producción*.

Un ejemplo visual de una historia de usuario en *Jira* es la que se muestra en la Figura 3.5. En esta figura se puede ver el estado de la historia de usuario, las tareas que componen la historia de usuario y sus estados.

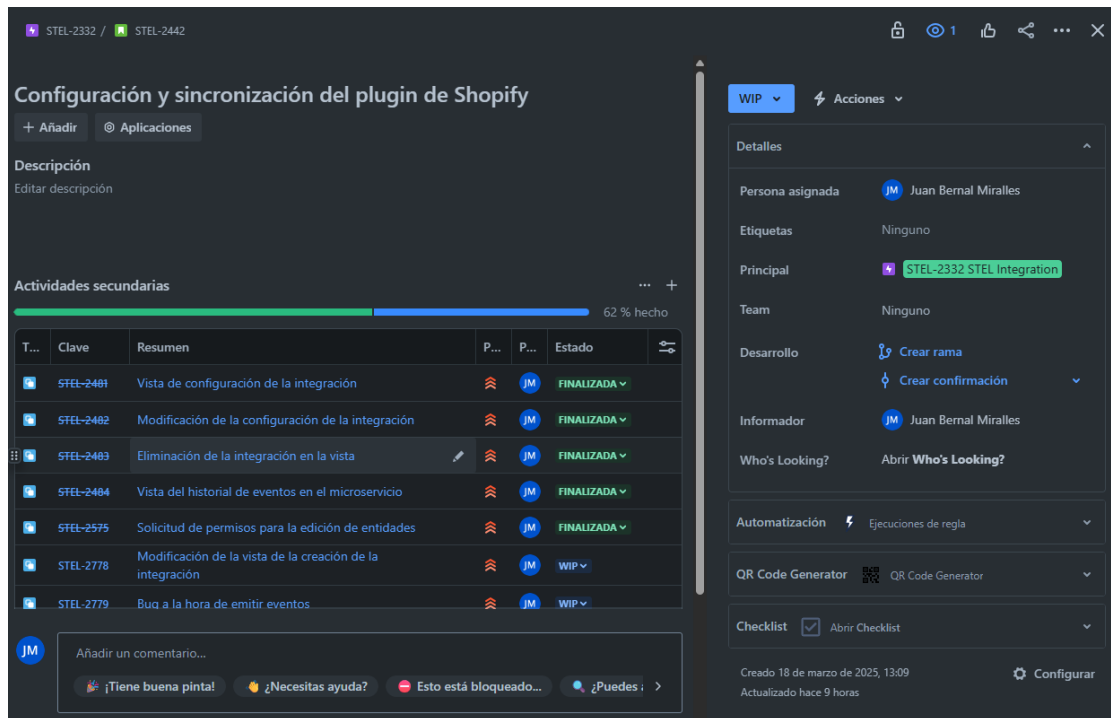


Figura 3.5: Historia de usuario en Jira.

3.3.2 RoadMap

El *RoadMap* tiene como finalidad la orientación del desarrollo del software mediante una estructura basada en *User Story Mapping*, la priorización de tareas con el método *MoSCoW* y la estimación del esfuerzo utilizando las tallas de camisetas y *Planning Poker*.

Para su elaboración, se ha empleado la herramienta *Confluence*, que ha permitido organizar el trabajo en diferentes *releases*. Tal y como se muestra en la figura 3.6, el *RoadMap* se ha estructurado en cuatro *releases* y siete épicas principales. Cada *release* tiene un objetivo específico y cada épica agrupa diferentes historias de usuario. En total, se han definido **18 historias** de usuario. Cada una de estas tiene diferentes versiones repartidas en diferentes *release*.

El *RoadMap* se ha dividido en cuatro *releases*, que son los siguientes:

- **Minimum Viable Product (MVP):** Este es el *release* correspondiente al TFM. En este *release* se van a desarrollar los objetivos específicos del TFM que consisten en una primera versión capaz de integrar *Shopify* con *Stel Order*. Este *release* es la base para empezar las demás *releases*.

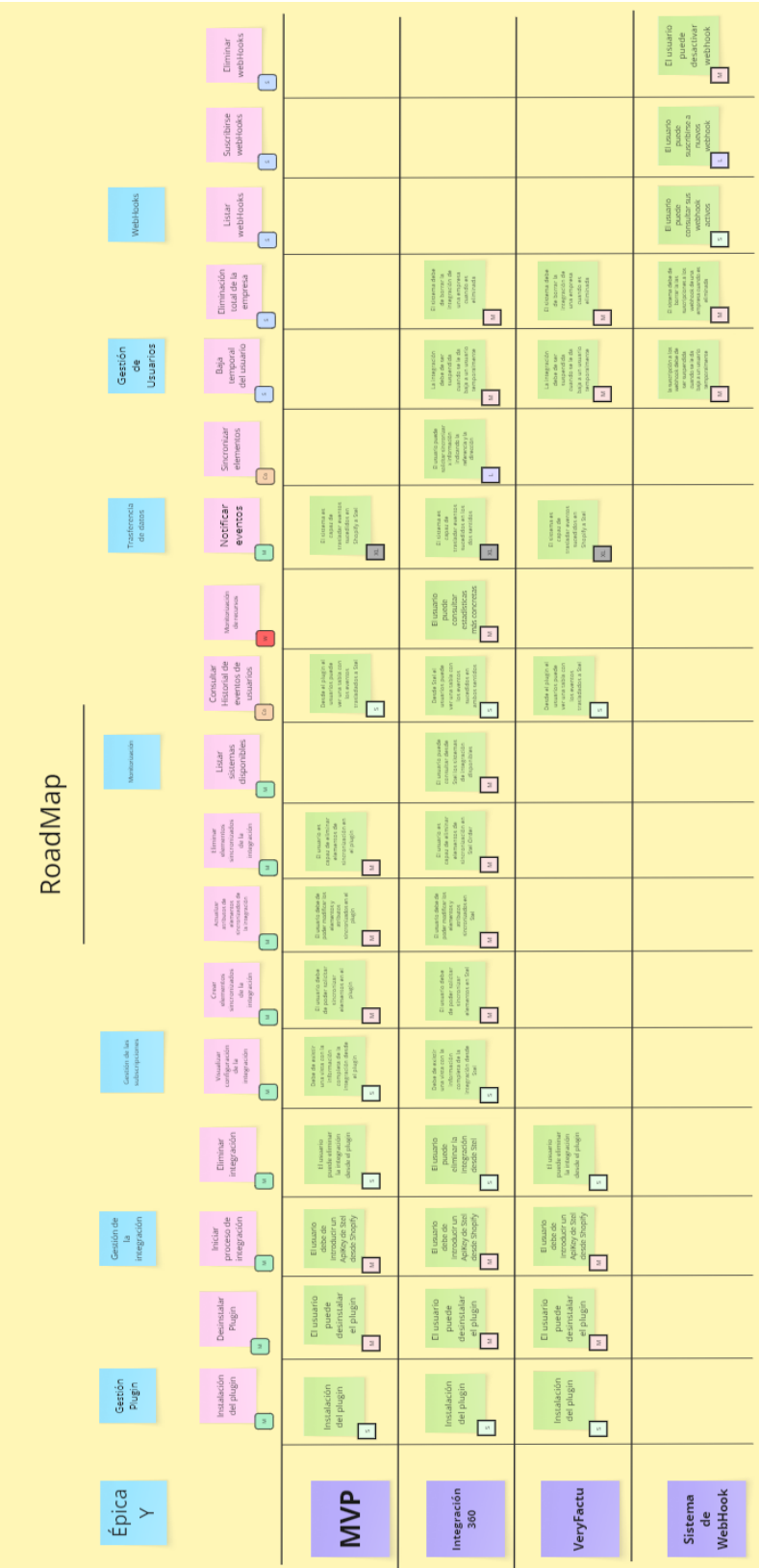


Figura 3.6: RoadMap del proyecto.

- **VeryFactu:** Esta versión del *release* amplía el MVP incorporando la integración entre *Shopify* y el sistema interno de *STEL Order* llamado *VeryFactu*.
- **Integración 360:** Consiste en la extensión del MVP a gran escala. Este *release* añade diferentes funcionalidades como la bidireccional, la sincronización automática o la edición desde *Stel Order*.
- **Sistema de WebHook:** Consiste en un sistema de *WebHook* que permita suscribirse a los eventos que sucedan en *Stel Order*. Este *release* está diseñado para ser utilizado por otros desarrolladores y empresas que quieran integrar sus aplicaciones con *Stel Order*.

Además de estos *releases*, se han definido **7 épicas** que agrupan parte de las historias de usuario. Estas épicas son las siguientes:

- **Gestión Plugin:** Esta épica agrupa las historias de usuario relacionadas con la gestión del plugin de integración con *Shopify*.
 - **Gestión de la integración:** Agrupa las historias de usuario que involucran la gestión de la integración.
 - **Gestión de suscripciones:** Las historias de usuario relacionadas con la gestión de suscripciones a los eventos de *Shopify*.
 - **Monitorización:** Corresponde a las historias de usuario relacionadas con la monitorización del sistema de integración.
 - **Trasferencia de datos:** Consiste en las historias de usuario relacionadas con la transferencia de datos entre *Shopify* y *Stel Order*.
 - **Gestión de Usuarios:** Se encarga de gestionar las integraciones cuando el usuario es dado de baja.
 - **Webhooks:** Esta épica agrupa las historias de usuario relacionadas con la gestión de *Webhooks*.
-

Se han definido exactamente 18 historias de usuario, cada una con diferentes versiones que representan sus respectivos incrementos. Algunos de estos incrementos no implican cambios reales, como en el caso de la historia sobre la instalación del *plugin*, que se desarrolla en la primera *release* y se reutiliza en las siguientes.

Las historias de usuario se han etiquetado con la técnica de *MoSCoW*. *MoSCoW* es una técnica inventada por Dai Clegg, que sirve para determinar la priorización dentro de proyectos con limitaciones de tiempo [37]. Esta técnica se basa en cuatro categorías *Must have*, *Should have*, *Could have* y *Won't have*.

Para las diferentes versiones de las historias de usuario se han utilizado las tallas de camisetas junto con la técnica de *Planning Poker*. Esta técnica consiste en asignar una talla a cada historia de usuario para estimar el esfuerzo necesario para desarrollarla. En este caso tenemos cuatro tallas de camisetas *S*, *M*, *L* y *XL*.

4 Diseño del Trabajo Realizado

En este capítulo se describen las decisiones adoptadas durante el diseño de la aplicación. Se analizan las distintas arquitecturas consideradas para el sistema y se justifica la elección de la arquitectura final. Además, se presentan los diagramas de entidad-relación y de persistencia, así como un prototipo inicial que permite visualizar el funcionamiento general de la aplicación.

4.1 Diseño y Análisis de la Arquitectura del Sistema

Una vez analizados los requisitos del sistema, se procede a diseñar distintas arquitecturas posibles. Para seleccionar la más adecuada, se evaluaron las ventajas e inconvenientes de cada alternativa.

Estas arquitecturas no solo se deben centrar en la integración con *Shopify*, sino también deben abarcar la integración con otras herramientas como pueden ser *WooCommerce*, *Prestashop* o *Magento*.

Antes de analizarlas, es importante identificar los elementos que pueden estar presentes en cada una de ellas.

- **Tienda/Herramienta:** Es la tienda online que se desea integrar con *STEL Order* en caso de que sea de interés integrar un *eCommerce*.
- **Plugin:** Es el componente que se instala el usuario en su tienda online para poder integrar *STEL* con la tienda.
- **Microservicio:** Es el componente que se encarga de almacenar la configuración de la integración y de gestionar la comunicación entre la aplicación y la tienda.
- **Monolito STEL Order:** Es la otra aplicación que debe ser integrada con la tienda online.
- **AWS:** Todo lo que aparece dentro del recuadro de *AWS* corresponde a la infraestructura desplegada en la nube, la cual es gestionada por *STEL Solutions*.

4.1.1 Arquitectura Basada en Plugin

Esta es la primera arquitectura analizada y propuesta. En esta se opta por crear un *plugin* para cada herramienta a integrar. Este debe ser capaz de transmitir la información entre las dos aplicaciones y de realizar las operaciones necesarias para que la aplicación funcione correctamente. Con esta arquitectura no se necesitaría la creación de un microservicio, ya que cada *plugin* es una aplicación independiente. En la siguiente Figura 4.1 se puede ver la arquitectura de esta opción.

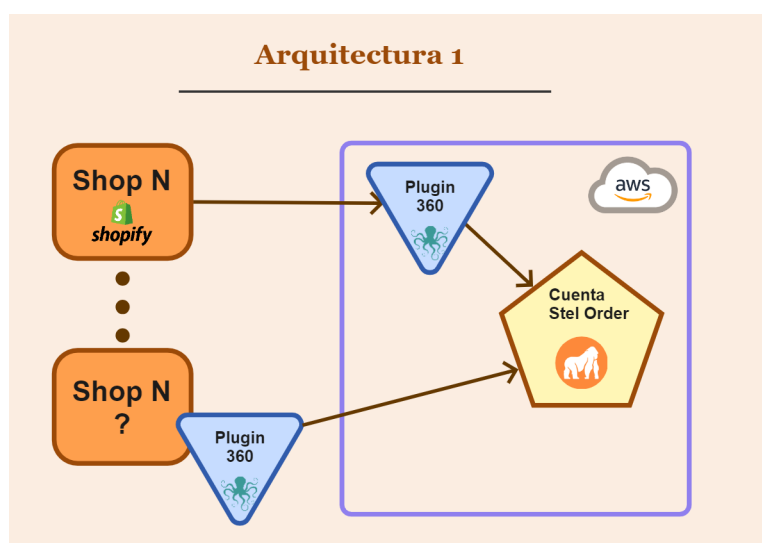


Figura 4.1: Arquitectura Basada en Plugin.

Ventajas

- **Velocidad de desarrollo:** Al ser específico para una herramienta, en un primer momento el desarrollo es más rápido y sencillo.
- **Más específico y personalizable:** Al ser diseñado específicamente para una herramienta, se puede personalizar y adaptar mejor a las necesidades de esa herramienta en particular.
- **Comunicación directa:** La comunicación entre la aplicación y la herramienta es más directa y sencilla. Esto puede hacer que la integración sea más rápida y eficiente.
- **Independencia frente a otras integraciones:** No se ve afectado por otras integraciones que se realicen en el futuro. Esto puede hacer que la integración sea más estable y menos propensa a caídas.

- **Menos complejidad:** La arquitectura de componentes es más sencilla y menos compleja. Esto puede hacer que la integración sea más fácil de entender y de desplegar.

Inconvenientes

- **Limitaciones de la herramienta a integrar:** Se ve limitado por las capacidades de la herramienta a integrar. Esto puede hacer que la integración sea menos flexible y escalable.
 - **Robo de propiedad intelectual:** En el caso de *Shopify* no habría este problema, ya que al ser un *plugin* de *Shopify* gestionado por la propia empresa, no se vería afectado. Sin embargo, en el caso de otras herramientas como *WooCommerce* o *Prestashop*, la empresa no tendría control sobre el *plugin* y la propiedad intelectual de la aplicación estaría en manos de los clientes, ya que el *plugin* se aloja en los clientes finales.
 - **Código no sostenible:** El código puede ser más difícil de mantener y de actualizar. Con el cambio de versiones de la herramienta a integrar, el *plugin* puede dejar de funcionar o puede ser necesario realizar cambios en el código para adaptarse a la nueva versión.
 - **Propensos a inconsistencias:** La integración puede ser más propensa a inconsistencias y errores. Toda la integración depende de un único *plugin* y si este *plugin* falla, la integración puede fallar también.
 - **No escalable:** La integración no es escalable. Esto significa que si se desea integrar más herramientas en el futuro, se tendría que crear un nuevo *plugin* para cada herramienta. Ello conlleva a que con cada nueva herramienta se deba partir desde cero sin ninguna base inicial
-

4.1.2 Arquitectura Basada en un Microservicio

Esta arquitectura representa una variación respecto a la anterior. En este enfoque, se incorpora un microservicio encargado de gestionar la comunicación entre la aplicación y la tienda online. Así, existe un único microservicio inteligente capaz de centralizar la integración de *STEL Order* con las distintas herramientas. Aunque sigue siendo necesario disponer de un *plugin* para cada herramienta, estos *plugins* ya no gestionan directamente la comunicación, sino que delegan esta función en el microservicio. La siguiente Figura 4.2 muestra la arquitectura correspondiente a esta opción.

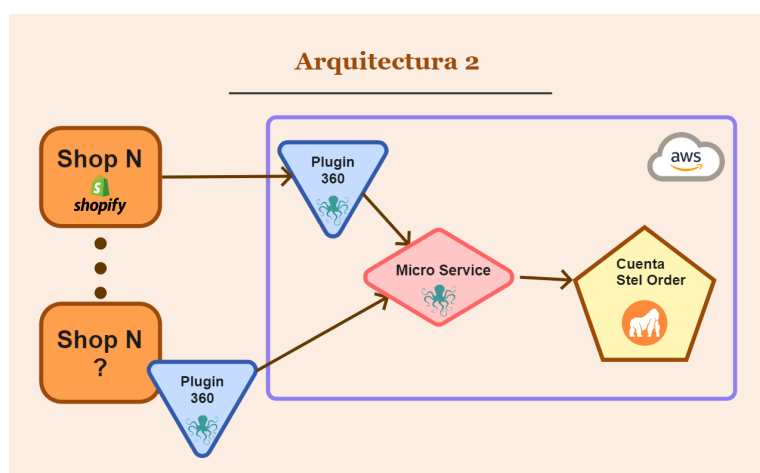


Figura 4.2: Arquitectura Basada en Microservicio.

Ventajas

- **Mayor Seguridad:** Al tener la configuración de la integración y el sistema de comunicación en un microservicio, ya no es un problema el robo de propiedad intelectual como en el caso de la arquitectura basada en *plugin*.
- **Desacoplamiento:** Se separa la generación de eventos de las aplicaciones del procesamiento de los mismos. Esto permite que cada parte del sistema pueda evolucionar de forma independiente y que se puedan realizar cambios en una parte del sistema sin afectar a las demás partes.
- **Mantenibilidad:** El código no depende de las actualizaciones de la herramienta a integrar. Esto hace que el código sea más fácil de mantener y de actualizar.
- **Enfoque orientado a microservicios:** Permite un enfoque más orientado a microservicios, lo cual es uno de los enfoques más modernos y escalables para el desarrollo de software.

Inconvenientes

- **Inversión AWS:** se debe invertir en la infraestructura de *AWS* para poder desplegar el microservicio.
- **Punto único de fallo:** Si este microservicio falla, toda la integración puede fallar.
- **Complejidad:** La arquitectura es más compleja y puede ser más difícil de desarrollar y configurar.
- **Acoplamiento de tecnologías:** Se ve obligado a adaptarse a cada una de las herramientas a integrar, las cuales pueden tener diferentes tecnologías, lógicas y necesidades.
- **Sobrecarga de responsabilidad:** Tener un microservicio tan inteligente puede hacer que la lógica de negocio se vuelva más compleja y difícil de mantener.

4.1.3 Arquitectura Basada en Microservicios

En esta arquitectura se opta por crear un microservicio y *plugin* para cada herramienta que se desea integrar. En este caso, cada microservicio es capaz de gestionar la comunicación entre la aplicación y la tienda online. Esta arquitectura se muestra en la siguiente Figura 4.3.

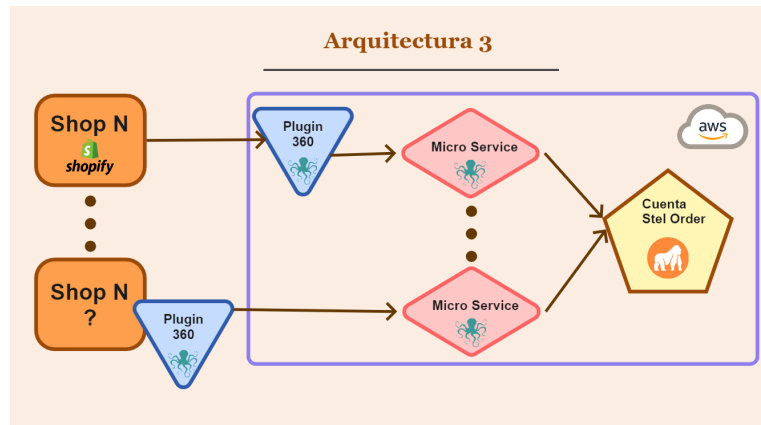


Figura 4.3: Arquitectura Basada en Microservicios.

El *plugin* sería el encargado de configurar la tienda online y de gestionar la comunicación entre la tienda online y el microservicio adaptándose a cada una de las herramientas. Mientras tanto, los microservicios compartirán la mayor parte de la lógica de negocio y la configuración de la integración. Dadas estas características, resulta

fundamental desarrollar una plantilla que sirva como base para los microservicios. Esta plantilla actuará como el esqueleto común, al que posteriormente se le añadirán las particularidades de cada microservicio.

Ventajas

Este tipo de arquitectura tiene todas las ventajas de la arquitectura basada en microservicio, pero además tiene otras:

- **Separación de responsabilidades:** Cada microservicio es responsable de una sola herramienta. Esto permite que cada parte del sistema pueda evolucionar de forma independiente y que se puedan realizar cambios en una parte del sistema sin afectar a las demás partes.
- **No hay punto único de fallo:** Si uno de ellos falla, el resto del sistema puede seguir funcionando. Además otra ventaja es que si no interesa económicamente mantener un microservicio, se puede eliminar sin afectar al resto del sistema.
- **Escalabilidad:** Al poder desarrollar una plantilla para los microservicios, se puede escalar la arquitectura de forma más sencilla. Cada vez que se desee integrar una nueva herramienta, solo es necesario crear un nuevo microservicio a partir de la plantilla y un nuevo *plugin*.
- **Complejidad:** La arquitectura es menos compleja que la arquitectura basada en un único microservicio.

Inconvenientes

- **Inversión AWS:** se debe invertir una mayor cantidad de recursos en la infraestructura de *AWS* para poder desplegar todos los microservicios.
 - **Costo de desarrollo:** Al tener que desarrollar tres componentes: plantilla, *plugin* y microservicio, el costo del desarrollo inicial es mayor. Sin embargo, a largo plazo, el costo de desarrollo se reduce al poder reutilizar la plantilla para cada nuevo microservicio.
-

4.1.4 Arquitectura Seleccionada

La primera arquitectura analizada no es una opción viable tras analizar que para otras herramientas como *WoCommerce* o *Prestashop*, el código del *plugin* no es propiedad de la empresa, sino que al instalarlo el cliente en su tienda online, el *plugin* pasa a ser propiedad del cliente.

Las otras dos arquitecturas son viables, pero tras analizar las ventajas e inconvenientes de cada una de ellas, se optó por la arquitectura basada en microservicios. Esto se debe a que tiene dos ventajas muy importantes, esto es, no hay un punto único de fallo y la escalabilidad del sistema. Estas ventajas son muy importantes para el futuro de la aplicación ya que se espera integrar más herramientas en el futuro.

Componentes de la Arquitectura

La arquitectura seleccionada consta de los siguientes componentes:

- **Shop de Shopify:** Tienda online de *Shopify* que se quiere integrar con *STEL Order*. Esta tienda debe instalar el *plugin* de *Shopify* y aceptar los permisos necesarios para que el *plugin* pueda acceder a la información de la tienda.
 - **Plugin de Shopify:** *Plugin* que se instala en la tienda online de *Shopify*. Encargado de gestionar los recursos de obtención de datos de la tienda y del acceso a la API. Al ser una aplicación de *Shopify*, este *plugin* es gestionado por *STEL Solutions*, si bien debe pasar la revisión de *Shopify* para poder ser publicado en la tienda de aplicaciones de *Shopify*.
 - **Microservicio de Shopify:** Un microservicio que se encarga de gestionar la configuración, comunicación y la lógica de negocio de la integración con *Shopify*. El microservicio es un elemento interno de *STEL Solutions* y es gestionado por la empresa.
 - **Monolito de STEL Order:** Cuenta de *STEL Order* que debe integrarse con la tienda online de *Shopify*. Este monolito es el encargado de gestionar la información de las empresas.
-

4.1.5 Arquitectura REST

REST es un estilo arquitectónico para aplicaciones distribuidas basado en los principios de la web. Se centra en el uso de recursos identificados por URL, manipulados mediante métodos estándar de Hypertext Transfer Protocol (HTTP) como **GET**, **POST**, **PUT**, **DELETE** y **PATCH** [38, 39].

Operaciones REST del Microservicio

El microservicio de *Shopify* utiliza REST para comunicarse con la aplicación de *STEL Order* y el *plugin* de *Shopify*. Esto permite una integración fluida y eficiente entre los diferentes componentes del sistema. En la siguiente tabla 4.1 se muestra todas las operaciones REST que se han utilizado en la aplicación.

Tipo	Ruta
Publico	
GET	/users/userId/integrations
POST	/integrations
Restringido	
GET	/integrations/integrationId
DELETE	/integrations/integrationId
GET	integrations/integrationId/subscriber
POST	integrations/integrationId/subscriber
DELETE	integrations/integrationId/subscriber
DELETE	integrations/integrationId/subscriber/subscriberId
PUT	integrations/integrationId/subscriber/subscriberId
PUT	integrations/integrationId/subscriber
GET	/integrations/integrationId/events
POST	/integrations/integrationId/events

Tabla 4.1: Operaciones REST del microservicio.

Operaciones del Plugin

A diferencia del microservicio, el *plugin* de *Shopify* no utiliza REST debido a que es una aplicación *Remix* de *Shopify*. Por lo tanto, las operaciones se realizan a través de la exposición de las rutas/vistas del *plugin* y de *endpoints* privados. De estos *endpoints*, el *plugin* de *Shopify* expone los siguientes:

Ruta	Descripción
Shop Shopify	
/webhooks/receive	Este endpoint es el encargado de recibir los eventos de Shopify.
/webhooks/app/uninstall	Endpoint para recibir el evento de desinstalación de la aplicación.
Microservicio Shopify	
/api/auth	Este endpoint es el encargado de autenticar al usuario y verificar si tiene permisos para acceder a la información de la tienda.

Tabla 4.2: Operaciones del Plugin de Shopify.

4.2 Arquitectura Limpia

La arquitectura limpia se basa en los principios de diseño y en las mejores prácticas en desarrollo [40]. Esta arquitectura 4.4 se centra en la separación de preocupaciones y en la independencia de los detalles de implementación, lo que facilita la prueba y el mantenimiento del software.

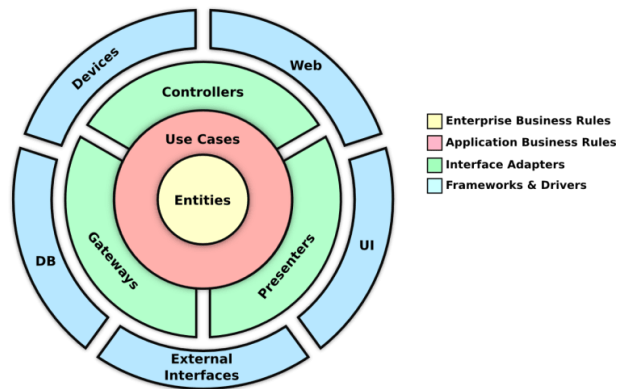


Figura 4.4: Estructura de la arquitectura limpia.

[41]

Para lograr una arquitectura limpia, se deben dividir las diferentes capas de la aplicación en diferentes módulos: Entities, Use Cases, Interface Adapters y Frameworks and Drivers. Cada una de estas capas tiene su propia responsabilidad y se comunica

con las demás capas a través de interfaces bien definidas [42].

4.3 Diagrama Dominio

El modelo de dominio puede ser tomado como el punto de partida para el diseño del sistema. Esto se debe a que, al aplicar la programación orientada a objetos, se parte de la premisa de que la estructura interna del software reflejará en cierta medida la realidad. Por ello, el mapa conceptual del modelo de dominio sirve como una primera aproximación o versión inicial del sistema.

4.3.1 Diagrama Dominio Plantilla/Micro-Servicio Shopify

En la Figura 4.5 se puede ver en verde el diagrama de dominio de la plantilla. Actualmente el microservicio de *Shopify* se encuentra en una fase del desarrollo inicial donde solo se sincroniza parte de información de la tienda a la aplicación. Por este motivo el microservicio de *Shopify* es el mismo que el de la plantilla salvo la parte en color azul.

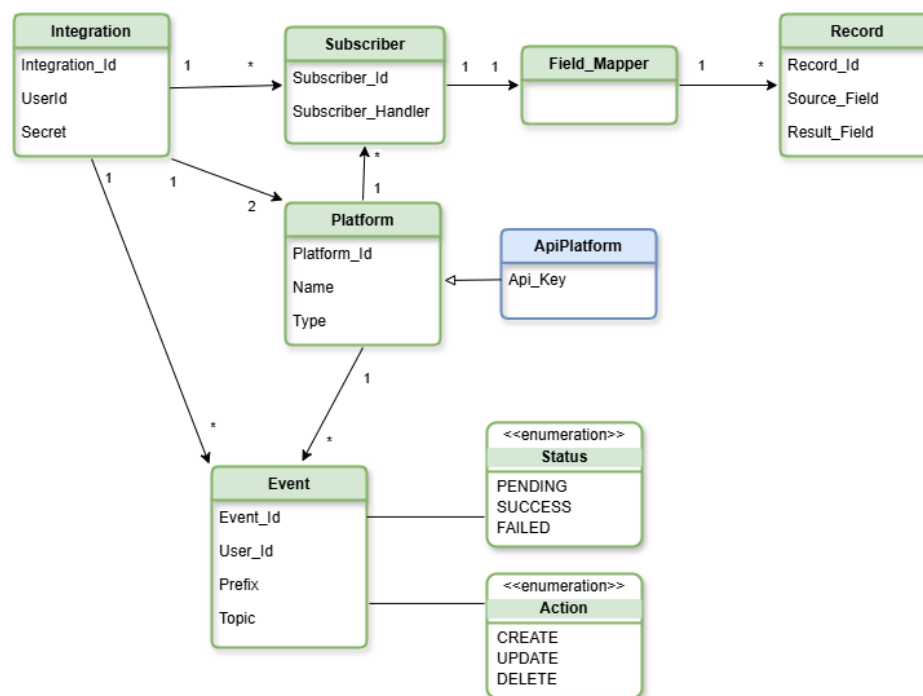


Figura 4.5: Diagrama Dominio Plantilla.

Tal y como se observa en la Figura 4.6, se ha definido una capa de dominio vinculada

a la infraestructura del microservicio para facilitar el mapeo de campos y la aplicación de la lógica de negocio. Esta capa incluye las entidades de *STEL Order* resultantes del proceso de mapeo. Como se mencionó al inicio de la memoria, las primeras entidades seleccionadas para este mapeo han sido pedidos de venta, clientes y productos, que son precisamente las representadas en el diagrama de dominio del microservicio.

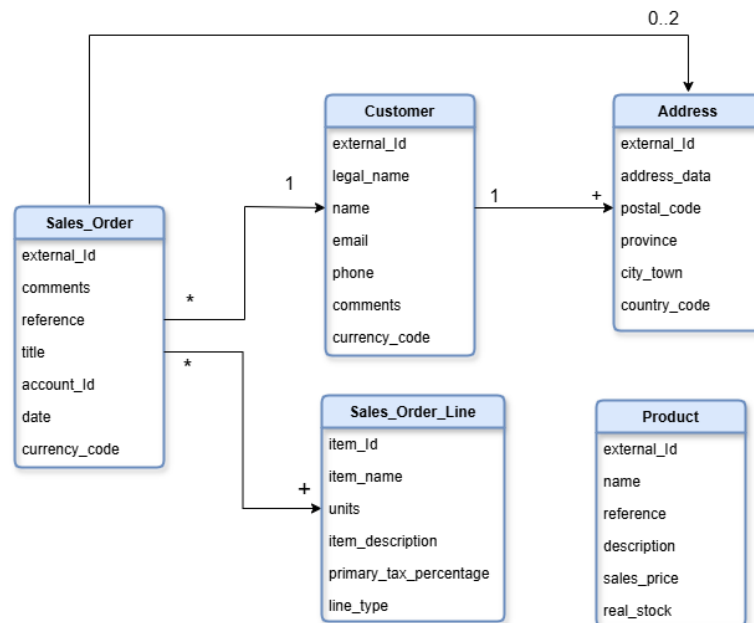


Figura 4.6: Diagrama Dominio Microservicio Shopify.

En el caso del *plugin* de *Shopify*, se optó por no implementar una capa de dominio propia, ya que la mayor parte de la lógica de negocio y el procesamiento de datos residen en el microservicio. El *plugin* actúa principalmente como intermediario entre la tienda de *Shopify* y el microservicio, gestionando la autenticación, la recepción de eventos y la comunicación con la API de *Shopify*, pero delegando la lógica de negocio y el tratamiento de la información en el *backend*. Por este motivo, el *plugin* no dispone de un diagrama de dominio específico, ya que no gestiona entidades de dominio complejas ni aplica reglas de negocio relevantes de forma autónoma..

4.4 Diagrama de Persistencia

El diagrama de persistencia es una representación gráfica de la estructura de la base de datos del sistema. Se han definido con la herramienta de diagramación de bases de datos *DBeaver*[43].

4.4.1 Diagrama de Persistencia Plugin

La Figura 4.7 muestra el diagrama de persistencia del *plugin* de *Shopify*, donde se representan las tablas empleadas por el *plugin* y las relaciones existentes entre ellas. En este caso, la base de datos utilizada es *MySQL* y la gestión de la persistencia se realiza mediante el ORM *Prisma*.

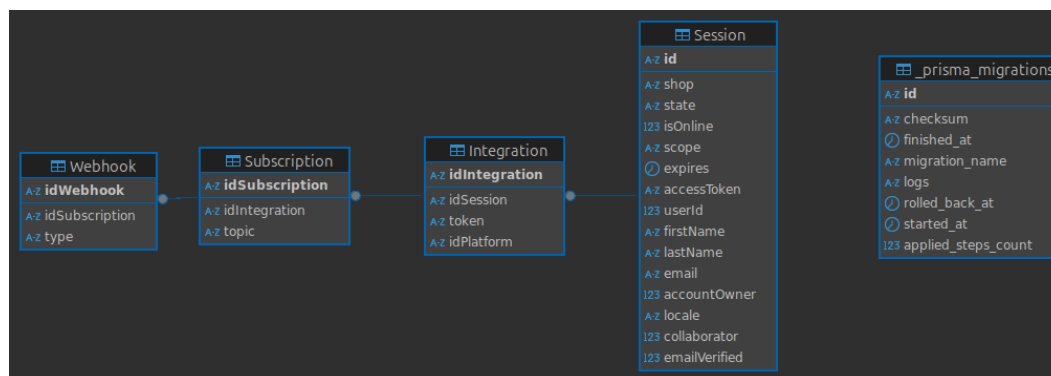


Figura 4.7: Diagrama Persistencia Plugin.

Cabe destacar que la tabla de migraciones se emplea para registrar y controlar los cambios realizados en la base de datos a través de *Prisma*. Por otro lado, la tabla *Session* es una tabla predeterminada proporcionada por el *plugin* de *Shopify*, utilizada para gestionar las sesiones y la autenticación de los usuarios de las tiendas de *Shopify*.

4.4.2 Diagrama de Persistencia Micro-Servicio Shopify

En la Figura 4.8 se ilustra el esquema de persistencia implementado para el micro-servicio de *Shopify*, detallando la estructura de tablas y sus relaciones. La solución tecnológica adoptada combina *Spring Boot JPA* como capa de abstracción sobre una base de datos relacional *MySQL*.

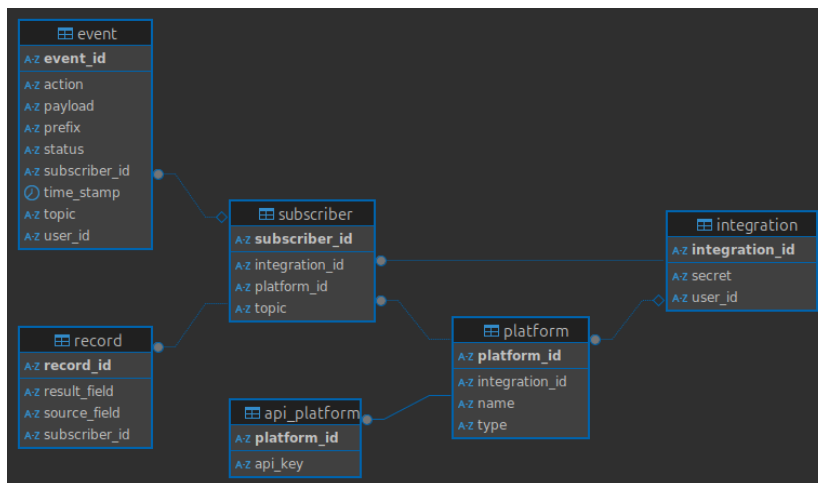


Figura 4.8: Diagrama Persistencia Microservicio Shopify.

4.5 Prototipo de la Aplicación

A continuación se muestra el prototipo de la aplicación de *Shopify*. Se han desarrollado los prototipos de la vista de creación de la integración y la vista del historial de eventos de un usuario. Estos prototipos se han desarrollado con la herramienta Canva[44].

En la Figura 4.9 se puede ver el prototipo de la vista de creación de la integración. En esta vista se puede ver el formulario para crear una nueva integración con *Shopify* la cual necesita el API Key generado por la aplicación de *STEL Order*.

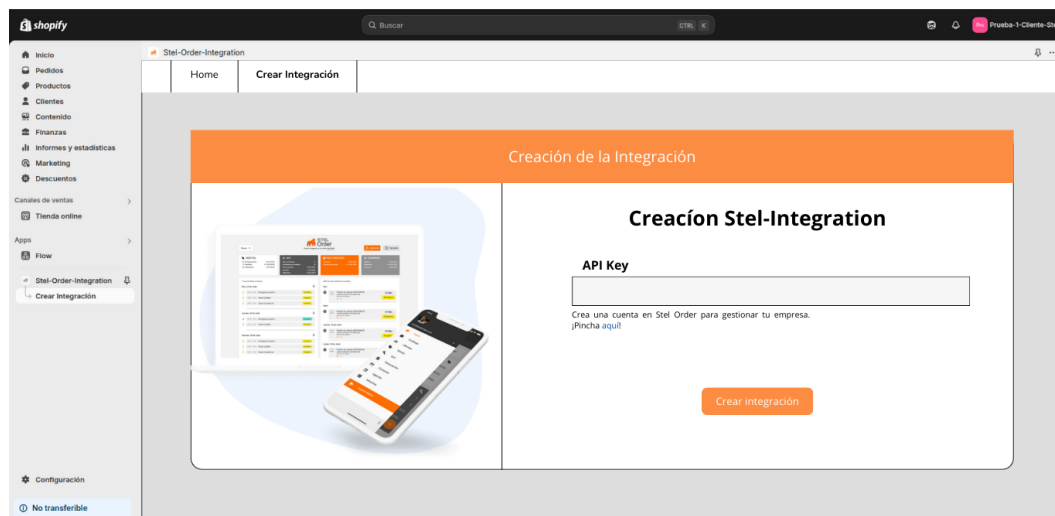


Figura 4.9: Prototipo Creación Integración.

En la Figura 4.10 se puede ver el prototipo de la vista del historial del usuario. En esta vista se pueden ver las diferentes opciones que tiene en el menú un usuario del *plugin* de *Shopify* en una primera versión. En la parte inferior se pueden ver una tabla con un historial de eventos. La información que se muestra de cada evento es su tópico, fecha, la acción realizada y el estado del evento.

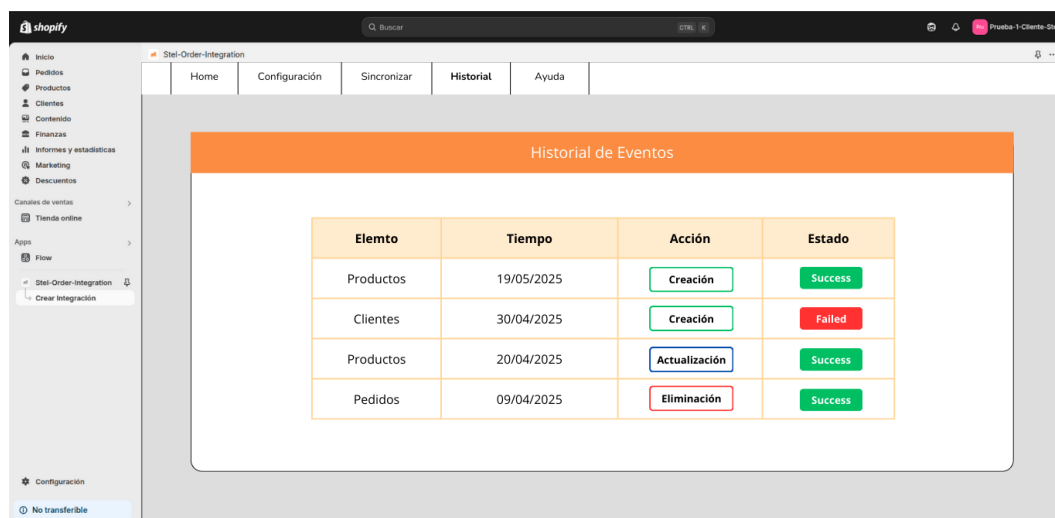


Figura 4.10: Prototipo Historial Usuario.

5 Desarrollo e Implementación de la Aplicación

En este capítulo se detallan los aspectos técnicos de la implementación del sistema de integración de *Shopify* con *STEL Order*. Este se divide en 4 secciones principales: el desarrollo del microservicio de *Shopify*, desarrollo del plugin de *Shopify*, la seguridad del sistema y las pruebas realizadas para garantizar su correcto funcionamiento y un futuro plan de pruebas.

5.1 Desarrollo del Microservicio de Shopify

5.1.1 Descripción del Microservicio

El microservicio de *Shopify* es una aplicación desarrollada en *Java* con el *framework* *Spring Boot*. Este se encarga de recibir los eventos de *Shopify*, procesarlos y enviarlos a *STEL Order*. También de la gestión de la configuración de la integración y de la autenticación de los usuarios. Este es un proyecto *Maven* donde se han utilizado varias librerías como *Spring Security*, *Spring Data JPA*, *Spring Web*, *Lombok*, entre otras.

5.1.2 Arquitectura del Proyecto

La arquitectura del microservicio está basada, como ya se ha explicado anteriormente, en una arquitectura por capas/arquitectura limpia. Con esta arquitectura se busca separar las responsabilidades de cada capa y hacer que cada una sea independiente de las demás. Las capas principales son:

- **Capa de Infraestructura:** Incluye todos los adaptadores y elementos técnicos necesarios para conectar y soportar las distintas capas del microservicio, como la configuración, la seguridad y la gestión de excepciones.
- **Capa de Dominio:** Contiene las entidades y lógica central del dominio, representando los conceptos y reglas de negocio fundamentales del microservicio.
- **Capa de Servicio:** Aloja la lógica de negocio principal. Está dividida en dos partes: la carpeta `service`, que contiene las interfaces (puertos) de la capa de servicio, y la carpeta `infrastructure/service`, que contiene las implementaciones y adaptadores de dichos puertos.

- **Capa de Persistencia:** Gestiona el acceso y almacenamiento de datos. Se organiza en dos partes: la carpeta **repository**, que define las interfaces de los repositorios, y la carpeta **infrastructure/repository**, que contiene las implementaciones y adaptadores de los repositorios.
- **Capa de Controlador:** Incluye los controladores responsables de exponer los *endpoints* HTTP, recibir las peticiones del cliente y devolver las respuestas adecuadas.

Capa de Infraestructura

Como se ha mencionado anteriormente, la capa de infraestructura contiene todos los adaptadores de las diferentes capas del microservicio. Además de los adaptadores de las capas de servicio y persistencia, también implementa parte de la funcionalidad:

- **Security:** Esta carpeta contiene la configuración de seguridad del microservicio. Es la responsable de la autenticación y autorización de los usuarios. Utiliza *Spring Security* para gestionar la seguridad del microservicio.
- **Exceptions:** Dentro de este directorio se encuentran el manejador global de excepciones y las excepciones personalizadas del microservicio. El manejador global de excepciones se encarga de capturar todas las excepciones que se producen en las diferentes capas y devolver una respuesta adecuada al cliente.
- **Config:** Este subdirectorio de infraestructura contiene la configuración de la aplicación. Aquí se encuentran la configuración de la capa de persistencia, la configuración de la capa de servicio, del publicador de eventos y la configuración de *Jackson* para la serialización y deserialización de los JavaScript Object Notation (JSON)

Adicionalmente, la capa de infraestructura también contiene una sección para la gestión de eventos internos del microservicio, el iniciador de *Spring Boot* de la aplicación y la infraestructura para el tratamiento de los eventos hacia *STEL*.

Esta infraestructura se ubica en la carpeta **infrastructure/STEL** subdividida en subcapas para gestionar de forma estructurada las entidades y la lógica asociada a *STEL Order*. Sus principales componentes son:

- **Data Transfer Object (DTO):** Incluye los DTO utilizados para recibir la información estructurada de los eventos de *Shopify*. Previamente esta información ha sido filtrada en la capa de dominio por el **FielMapper**. Además, estos DTO se encargan de validar los campos de las entidades antes de su procesamiento.
 - **Mapper:** Contiene los mapeadores responsables de transformar los DTO en entidades de dominio de *STEL Order*. Estos mapeadores encapsulan parte de la
-

lógica de negocio utilizada para adaptar los datos recibidos de *Shopify* al modelo de *STEL Order*. Por ejemplo, el mapeador de clientes elige si el nombre legal del cliente de *STEL Order* corresponde a los campos `first-name` y `last-name` o a `name-company` de *Shopify*.

- **Domain:** Tal y como se ilustra en la Figura 4.6, esta subcarpeta contiene las entidades de dominio de *STEL Order*. Estas entidades son las que, tras ser serializadas a JSON, se envían finalmente a la plataforma.
- **Services:** Esta subcapa agrupa los servicios que contienen la lógica de negocio para procesar los eventos a *STEL Order*. Los servicios hacen uso de los clientes de plataformas y de las entidades de dominio de *STEL*. Además, se estructura en puertos (interfaces que definen los casos de uso) y adaptadores (implementaciones concretas de dichos puertos), siguiendo los principios de la arquitectura limpia.

Capa de Dominio

La capa de dominio contiene las entidades del dominio mostradas en la Figura 4.5 en la sección anterior. Son las entidades sobre las que actúan la lógica de negocio. Las entidades del dominio principales son:

- **Integration:** Define la relación de integración entre *Shopify* y *STEL Order*. Actúa como la clase base para las distintas entidades de integración.
- **Platform:** Representa una jerarquía de clases para las diferentes aplicaciones. En este proyecto, se ha implementado un tipo de plataforma que incluye un dominio web y una API Key, permitiendo modelar tanto *Shopify* como *STEL Order*.
- **Subscription:** Modela la suscripción de un usuario a un tópico concreto. Almacena información sobre el usuario, la plataforma, el tópico y los campos asociados a la suscripción.
- **Event:** Describe un evento generado en *Shopify*. Incluye detalles como el tipo de evento, su contenido, estado y fecha de creación.

Capa de Servicio

Se encarga de la lógica de negocio del microservicio y se divide en dos partes: la carpeta `service`, que contiene los puertos de la capa de servicio y la carpeta `infrastructure/service` que contiene las implementaciones/adaptadores de los puertos de la capa de servicio.

Los puertos de la capa de servicio son interfaces que definen los casos de uso del microservicio. Estos puertos son:

- **EventService**: Encargado de gestionar las operaciones relacionadas con los eventos provenientes de *Shopify*.
- **SubscriptionService**: Responsable de manejar las operaciones asociadas a las suscripciones de los usuarios a diferentes tópicos.
- **IntegrationService**: Gestiona las funcionalidades vinculadas a la integración entre *Shopify* y *STEL Order*.
- **AuthService**: Se ocupa de los procesos de autenticación y autorización de los usuarios.
- **ClientPlatform**: Interfaz que define las operaciones necesarias para interactuar con plataformas externas.

Las implementaciones/adaptadores de los puertos de la capa de servicio se encuentran en la carpeta `infrastructure/service`. Estas implementaciones se dividen en dos partes:

- **Impl**: Contiene las implementaciones propias del microservicio anotadas con `@Service` y `@Transactional`. No solo se encuentran las implementaciones de los puertos de la capa de servicio, sino que también se encuentran estrategias de eventos que utilizan los servicios, DTO y mapeadores de STEL para enviar los eventos a *STEL Order*.
- **Clients**: Son las implementaciones de los clientes, necesarios para interactuar con plataformas externas. En el caso de *STEL Order*, se han desarrollado varios clientes específicos, uno para cada controlador principal de la plataforma (por ejemplo, **ProductClientSTEL**, **OrderClientSTEL** y **CustomerClientSTEL**), siguiendo el patrón de diseño **Factory** [45] para su instanciación dinámica según la necesidad. Además, para cada plataforma externa se ha implementado un cliente dedicado a la autenticación.

Un apartado importante de la capa de servicio es que tanto para las estrategias de eventos como para la autenticación de las plataformas externas se ha utilizado el patrón de diseño **Strategy** [46].

Capa de Persistencia

Al igual que la capa de servicio, la capa de persistencia se divide en dos partes: la carpeta `repository`, que contiene los puertos de la capa de persistencia; y la carpeta `infrastructure/repository`, que contiene las implementaciones/adaptadores de los puertos de la capa de persistencia.

En la carpeta que almacena los puertos de la capa de persistencia se encuentran las interfaces que definen los repositorios del microservicio. Estos repositorios son: **EventRepository**, **SubscriptionRepository** e **IntegrationRepository**.

Las implementaciones/adaptadores de los puertos de la capa de persistencia se encuentran en la carpeta **infrastructure/repository**. Estas implementaciones utilizan *Spring Data JPA* para interactuar con la base de datos y se encargan de realizar las operaciones *CRUD* sobre las entidades del dominio. Además de los repositorios, también existen más elementos:

- **Adapters:** Los adaptadores de repositorio actúan como puente entre el dominio y la capa de persistencia de manera transparente para el resto de la aplicación.
- **Mapper:** Los adaptadores anteriores utilizan los mapeadores para convertir las entidades del dominio a entidades de persistencia y viceversa.
- **Entities:** Representan las clases que modelan la estructura de los datos tal y como se almacenan en la base de datos. Estas entidades están anotadas con `@Entity` y representan las tablas de la base de datos.
- **Specification:** Las especificaciones son una forma de definir consultas complejas de manera declarativa. Se ha definido una especificación para el filtro de obtención de los eventos de *Shopify*.

Capa de Controlador

Los controladores son los responsables de exponer los *endpoints* REST del microservicio. Para ello, se ha utilizado *Spring Web* se anotaron con `@RestController`. Los componentes dentro de esta capa son:

- **Controllers:** Clases que definen los *endpoints* REST del microservicio recibiendo las peticiones HTTP y delegando las operaciones a los servicios correspondientes.
- **DTOs:** Los DTO son objetos que se utilizan para transferir datos entre el cliente y el servidor. Existen dos tipos de DTO: los que se utilizan para recibir los datos del cliente en la carpeta **request** y los que se utilizan para enviar los datos al cliente en la carpeta **response**.
- **Mappers:** Los mapeadores son responsables de convertir los DTO en entidades del dominio y viceversa. Estos mapeadores utilizan la librería para realizar la conversión de forma automática.

En esta capa se han implementado los siguientes controladores:

- **EventController:** Gestiona los *endpoints* REST relacionados con los eventos recibidos desde *Shopify*, delegando la lógica de negocio en el **EventService**.

- **SubscriptionController:** Administra las operaciones sobre las suscripciones de los usuarios a los distintos tópicos, utilizando el **SubscriptionService** para la gestión interna.
- **IntegrationController:** Se encarga de las operaciones vinculadas a la integración entre *Shopify* y *STEL Order*, apoyándose en el **IntegrationService** para la gestión de integraciones y en el **AuthService** para la autenticación de usuarios durante la creación de integraciones.

5.1.3 Patrones

En el desarrollo del microservicio de *Shopify* se han utilizado varios patrones de diseño para mejorar la calidad del código y la mantenibilidad del mismo, aspectos importantes en el desarrollo de esta aplicación.

Patrón de Diseño Strategy

El patrón de diseño **Strategy** consiste en definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables [46]. La estrategia permite que el algoritmo varíe, independientemente de los clientes que lo utilizan. Una de las ventajas de este patrón es que se resuelve en tiempo de ejecución, lo que permite cambiar el comportamiento del sistema sin necesidad de modificar el código.

Este patrón se ha utilizado en el microservicio de *Shopify* para definir dos estrategias:

- **Event Strategy:** Define las diferentes estrategias de eventos que se pueden utilizar para enviar los eventos a *STEL Order*. Cada estrategia implementa la interfaz **EventStrategy** y define su propio comportamiento para enviar los eventos.
- **Auth Platform Strategy:** Define las diferentes estrategias de autenticación para las plataformas externas. Cada estrategia implementa la interfaz **Auth Platform Strategy** y define su propio comportamiento para autenticar a los usuarios en las plataformas externas.

Patrón de Diseño Factory

El patrón de diseño **Factory** es un patrón que proporciona un método estático encapsulado en una clase llamada fábrica, para ocultar la lógica de implementación y hacer que el código del cliente se centre en el uso, en lugar de inicializar nuevos objetos [45].

En el microservicio de *Shopify* se hace uso de este patrón en varios puntos:

- **PlatformClientFactory**: Se encarga de instanciar los clientes necesarios para interactuar con las diferentes plataformas externas. Aplica el patrón *Factory* para seleccionar y crear el cliente adecuado según la plataforma requerida.
- **EventStrategyFactory**: Responsable de generar las estrategias de eventos apropiadas. Utiliza el patrón *Factory* para seleccionar la estrategia más adecuada en función del tópico, tipo de evento y plataforma final.
- **AuthPlatformStrategyFactory**: Gestiona la creación de estrategias de autenticación para plataformas externas, aplicando el patrón *Factory* para elegir la estrategia correcta según la plataforma correspondiente.
- **ServiceFactory**: Actúa como una fábrica para instanciar servicios asociados a las estrategias de eventos.
- **MapperFactory**: Se encarga de crear los mapeadores necesarios para transformar entidades del dominio en entidades de persistencia y viceversa, utilizando el patrón *Factory* para seleccionar el mapeador adecuado según la entidad a tratar.

Patrón de Diseño Adapter

El patrón de diseño de adaptador en Java convierte la interfaz de una clase en otra interfaz que los clientes esperan, lo que permite la compatibilidad [47].

Este patrón se ha implementado en la capa de persistencia del microservicio de *Shopify* para adaptar los repositorios. Con esta implementación se logra que las entidades del dominio puedan interactuar con la base de datos de manera transparente, sin necesidad de que el resto del microservicio conozca los detalles de la implementación de la persistencia.

Además de los patrones mencionados anteriormente, también se han empleado otros patrones de diseño como el patrón **DTO** para la transferencia de datos entre las distintas capas del microservicio, el patrón **Singleton** para garantizar que solo exista una instancia de ciertas clases y el patrón de **Inyección de Dependencias** para gestionar las dependencias de los diferentes componentes del sistema mediante el contenedor de dependencias de *Spring*.

5.2 Desarrollo del Plugin de Shopify

5.2.1 Descripción del Plugin

El *plugin* de *Shopify* se corresponde con una aplicación *Shopify* desarrollada en *TypeScript* con el *framework* de *Remix*. Este parte de una plantilla de *Shopify*, que se ha modificado para adaptarlo a las necesidades del proyecto.

El *plugin* tiene tres funcionalidades principales:

- **Gestión de acceso:** Permite a los usuarios instalar el *plugin* en su tienda de *Shopify* almacenando la sesión del usuario y la tienda en la base de datos de *Shopify*.
- **Gestión de Webhooks:** Es el responsable de gestionar los *Webhooks* de *Shopify*.
- **Frontend de la Aplicación:** Proporciona una interfaz de usuario para que los usuarios puedan gestionar sus integraciones con *STEL Order* desde el *plugin*.

5.2.2 Plugin Backend

A diferencia del microservicio de *Shopify*, el *plugin* tiene un mayor acoplamiento de sus capas ya que no es necesario que el *plugin* sea tan escalable como el microservicio de *Shopify*. Sin embargo, se han seguido algunas buenas prácticas para mantener una estructura organizada y modular.

Al utilizar *TypeScript* para el desarrollo del *Backend* del *plugin*, se ha optado por crear clases para representar a los servicios y repositorios necesarios para la gestión de la aplicación. Estas clases *TypeScript* son implementadas utilizando el patrón de diseño **Singleton** para garantizar que solo exista una instancia de cada servicio o repositorio a lo largo de la aplicación. Para ello, se ha instanciado un constructor privado y se ha proporcionado un método estático para obtener la instancia.

Además, para realizar un correcto manejo de los errores, se han implementado diferentes errores personalizados que heredan de la clase **Error** de *JavaScript*. Estos errores personalizados permiten manejar los errores de manera más específica y proporcionar mensajes de error más claros a los usuarios.

El *plugin* de *Shopify* se organiza en cuatro carpetas principales. Estas carpetas son: **persistence**, **module**, **service** y **routes**. Cada una de estas carpetas tiene una función específica dentro del *plugin*.

Capa/Carpeta de Persistencia

En esta aplicación, la capa de persistencia contiene los repositorios que se encargan de interactuar con el ORM Prisma. Como se ha mencionado anteriormente, se han implementado clases que representan los repositorios utilizando el patrón de diseño **Singleton**. Los repositorios principales son: **SessionRepository**, **WebhookRepository**, **IntegrationRepository** y **SubscriptionRepository**.

No existen entidades de persistencia como tal, ya que Prisma se encarga de generar las entidades a partir del esquema definido en el fichero `schema.prisma`. Este esquema define las tablas y relaciones de la base de datos. Estas entidades generadas por Prisma son las ilustradas en la Figura 4.7 en la sección anterior.

Carpeta de Módulo

Actualmente, esta carpeta centraliza la gestión de los *Webhooks* de *Shopify*, aunque en el futuro debería ser refactorizada, ya sea cambiando su ubicación y nombre o transformándola en un servicio independiente. Contiene tres archivos principales, cada uno encargado de una función específica:

- **WebhookCreate**: Gestiona la creación de *Webhooks* en *Shopify*, utilizando el repositorio correspondiente para almacenar la información del *Webhook* en la base de datos.
- **WebhookDelete**: Se encarga de eliminar los *Webhook* de *Shopify*, accediendo al repositorio de *Webhooks* para borrar la información asociada en la base de datos.
- **WebhookCreateAppUninstalled**: Configura la creación de un *Webhook* específico para el evento de desinstalación de la aplicación en *Shopify*. Este *Webhook* es fundamental para eliminar la integración entre *Shopify* y *STEL Order* cuando el usuario desinstala la aplicación.

Capa/Carpeta de Servicio

La capa de servicio del *plugin* de *Shopify* contiene las clases que implementan la lógica de negocio del *plugin*. Estas clases son responsables de gestionar las operaciones relacionadas con los *Webhooks*, las sesiones y las integraciones. Al igual que en la capa de persistencia, se han implementado clases que representan los servicios utilizando el patrón de diseño **Singleton**. Estos servicios se encuentran agrupados en dos carpetas principales:

- **PluginService**: Contiene los servicios relacionados con la gestión del *plugin* de *Shopify*. Estos servicios son:
-

- **IntegrationService**: Contiene la lógica de negocio relacionada con las integraciones entre *Shopify* y *STEL Order*. Entre sus funciones incluyen la creación, actualización y eliminación de integraciones.
- **SubscriptionService**: Se encarga de gestionar las suscripciones de los usuarios. Este servicio es capaz de tratar inconsistencias en las suscripciones. Por ejemplo, si en el microservicio de *Shopify* no se encuentra una suscripción que sí está en el *plugin* de *Shopify*, se elimina la suscripción del *plugin* de *Shopify*.
- **Webhookservice**: Gestiona la lógica de negocio relacionada con los *Webhooks* de *Shopify*. Este servicio es responsable de llamar a las funciones correspondientes para la creación y eliminación de *Webhooks*.
- **EventService**: Alberga la lógica de negocio relacionada con los eventos de *Shopify*.
- **STELService**: Contiene los servicios encargados de interactuar con el microservicio de *Shopify*. Estos servicios son responsables de gestionar la autenticación y la comunicación utilizando los *endpoints* del microservicio. Los servicios desarrollados son: **IntegrationSTELService**, **SubscriptionSTELService** y **EventSTELService**.

Capa/Carpeta de Rutas

La carpeta de rutas agrupa los *endpoints* disponibles en el *plugin* de *Shopify*. Estos *endpoints* permiten tanto a *Shopify* acceder a la interfaz de usuario del *plugin* como a herramientas externas interactuar con el mismo. Los principales *endpoints* expuestos para las herramientas externas son:

- **Api Auth**: Endpoint utilizado para autenticar la tienda desde el microservicio de *Shopify*. Recibe el dominio de la tienda y el token de acceso, devolviendo un error si no son correctos o un 200 OK en caso contrario.
- **Webhook App Uninstalled**: Endpoint encargado de recibir el *Webhook* de desinstalación de la aplicación.
- **Webhook App Scopes Updated**: Endpoint encargado de recibir el *Webhook* de actualización de los permisos de la aplicación.
- **Webhook Received**: Endpoint diseñado para recibir los *Webhooks* de *Shopify* generados para las suscripciones.

5.2.3 Plugin Frontend

El *plugin* de *Shopify* utiliza el *framework* de *Remix* para el desarrollo del *frontend*. Además, se ha utilizado la librería de componentes *Bootstrap* para el diseño de la interfaz

de usuario. El *frontend* del *plugin* se organiza en tres carpetas principales: **routes**, **context** y **components**.

Carpeta de Componentes

Estos componentes son utilizados en las diferentes vistas del *plugin*. Estos están diseñados para ser altamente reutilizables y configurables, lo que permite una mayor flexibilidad y consistencia en la interfaz de usuario.

Los dos componentes principales son:

- **Alert**: Se utiliza para mostrar mensajes a los usuarios. Permite mostrar mensajes de éxito, error o información de manera clara y concisa.
- **Switch**: Se utiliza para mostrar una tarjeta con un switch principal (tópico) y debajo una acordeón con los diferentes switches (atributos de los tópicos). Permite a los usuarios activar o desactivar los diferentes elementos de forma sencilla.

Carpeta de Context

Para comprender la lógica de la interfaz hay que diferenciar dos tipos de contexto: el contexto sin la integración y el contexto con la integración. El contexto `IntegrationContext` se utiliza para gestionar y compartir el identificador de la integración (`idIntegration`) entre los diferentes componentes del *frontend* del *plugin* de *Shopify*.

Este contexto se implementa utilizando la API de *Context* de *React* y proporciona tanto el valor actual del identificador como una función para actualizarlo (`setIdIntegration`) cuando la integración sea creada o eliminada.

Carpeta de Rutas

Las vistas del *plugin* de *Shopify* se organizan dentro de la carpeta **routes**. Cada vista se corresponde con una ruta del *plugin*. Antes de explicar las diferentes vistas, es importante mencionar el elemento principal de todo proyecto de *Shopify*, el `AppProvider`. Este componente es el encargado de configurar los menús de las vistas y de instanciar los componentes hijos a través de la propiedad `outlet`.

Las vistas actuales del *plugin* de *Shopify* son:

- **Home**: Esta vista se corresponde con el fichero `app._index.tsx` y es la interfaz inicial del *plugin*.
- **Crear Integración**: Ubicada en el fichero `app.create.tsx`, consiste en una interfaz donde el usuario puede introducir la `apiKey` de *STEL Order* para crear una integración.

- **Configuración de Integración:** Interfaz que permite al usuario gestionar la configuración de la integración con *STEL Order*. El fichero que contiene esta vista es `app.integration.tsx`.
- **Historial de Eventos:** Esta vista se encuentra en el fichero `app.history.tsx` y muestra una tabla con los eventos persistidos en el microservicio de *Shopify*.

Diagrama de Vistas

El diagrama de vistas del *plugin* de *Shopify* se muestra en la Figura 5.1. Este diagrama representa las diferentes vistas del *plugin* y cómo se relacionan entre sí.

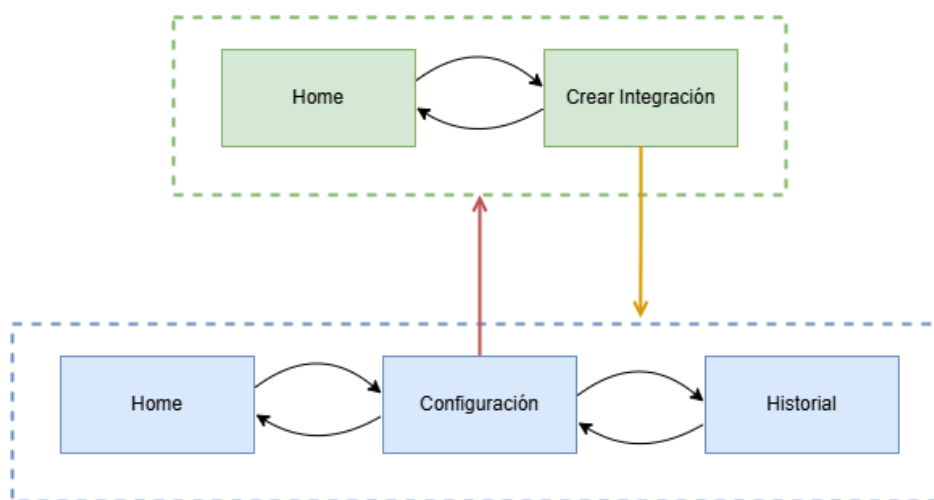


Figura 5.1: Diagrama de vistas del plugin de Shopify.

Existen dos grupos de vistas: En color verde, las vistas que se muestran antes de la creación de la integración; y, en color azul, las que se muestran después de la creación. La flecha con color rojo refleja que el usuario, desde la vista de la configuración, elimina la integración, en cuyo caso cambia de contexto y es redirigido a la vista de creación de la integración. La flecha amarilla representa el flujo que sigue un usuario cuando crea una nueva integración desde la vista correspondiente. Una vez completado el proceso de creación, el sistema redirige automáticamente al usuario a la pantalla de configuración de dicha integración, donde podrá ajustarla según sus necesidades.

Home

La vista Home 5.2 es la pantalla inicial que se muestra al usuario al acceder al *plugin* de *Shopify*. En ella se presenta un mensaje de bienvenida junto con enlaces para explorar el entorno de *STEL Order* y para iniciar el proceso de creación de una integración. Si el usuario ya dispone de una integración creada, también se le ofrece un acceso directo a la configuración de dicha integración.

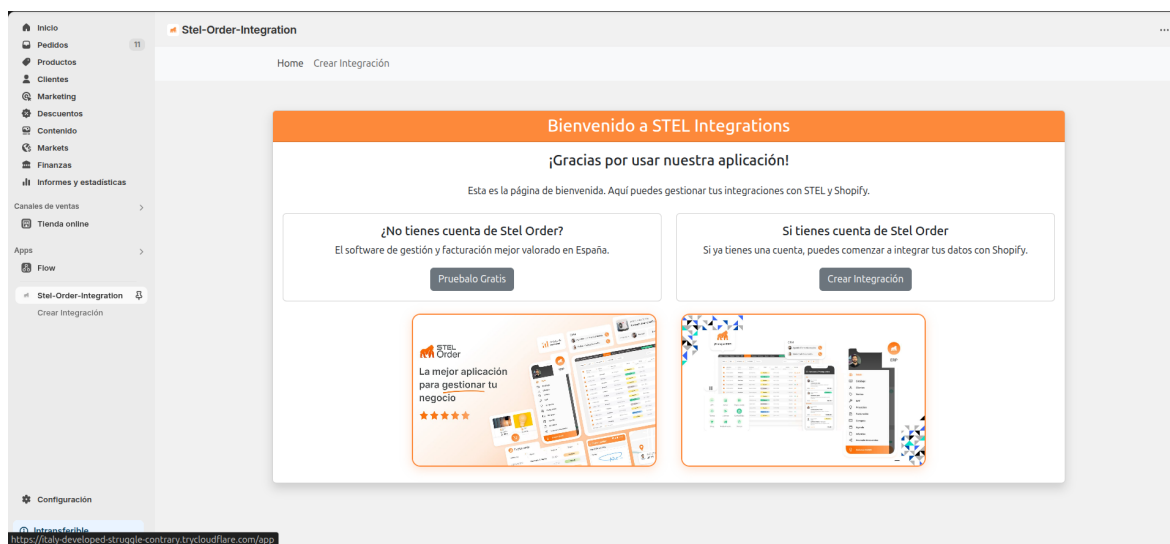


Figura 5.2: Página de inicio de las integraciones.

Crear Integración

Esta interfaz permite al usuario introducir la `apiKey` de *STEL Order* para crear una nueva integración. Actualmente, también se solicita el identificador de usuario, aunque este campo se eliminará en futuras versiones. Tras introducir los datos, si se produce algún error, se muestra un mensaje informativo al usuario. En caso de éxito, el usuario es redirigido automáticamente a la vista de configuración de la integración, cambiando de contexto. En la siguiente Figura 5.3 se muestra la vista desde una tienda de Shopify.

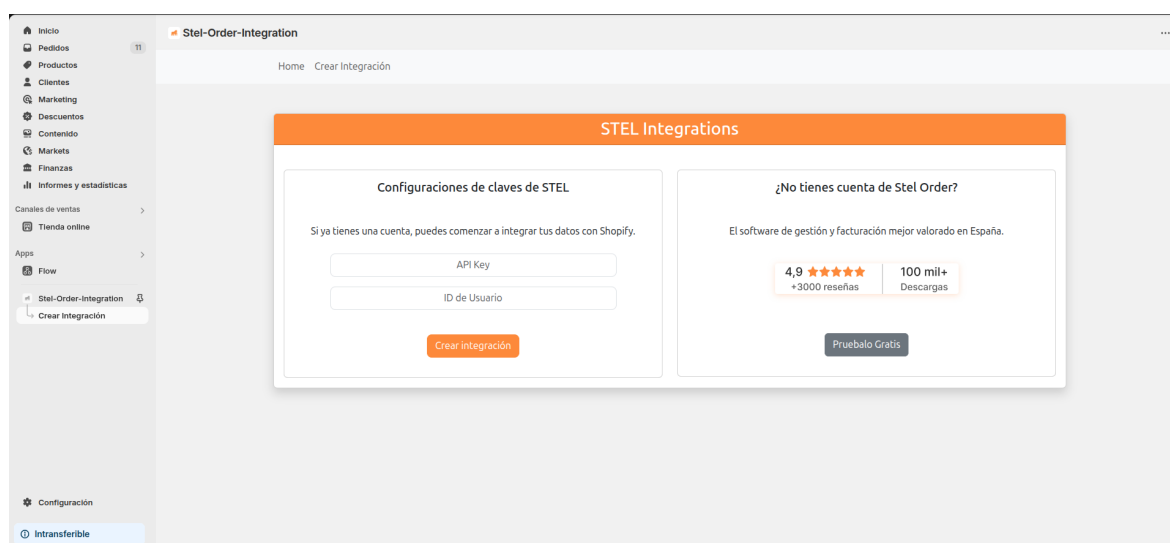


Figura 5.3: Vista de la creación de la integración.

Configuración de Integración

En la vista de configuración de la integración (véase Figura 5.4), el usuario puede gestionar diversas acciones relacionadas con la misma. Entre las opciones disponibles, se encuentra la posibilidad de eliminar la integración, lo que provoca la redirección a la vista de creación de integraciones, modifica el contexto del usuario y elimina los datos correspondientes tanto en la base de datos de *Shopify* como en el microservicio.

Además, se presentan diferentes entidades con interruptores (switches) para activar o desactivar los distintos tópicos de *Shopify*, como Productos, Pedidos en Borrador y Clientes. Al activar un tópico, se despliega un acordeón que permite seleccionar los campos opcionales a sincronizar, los cuales también pueden activarse o desactivarse individualmente mediante switches.

Una vez seleccionados los tópicos y campos deseados, el usuario puede actualizar la integración. Si ocurre algún error durante el proceso, se muestra un mensaje informativo. Si la actualización es exitosa, se informa al usuario sobre los tópicos eliminados,

creados o actualizados.

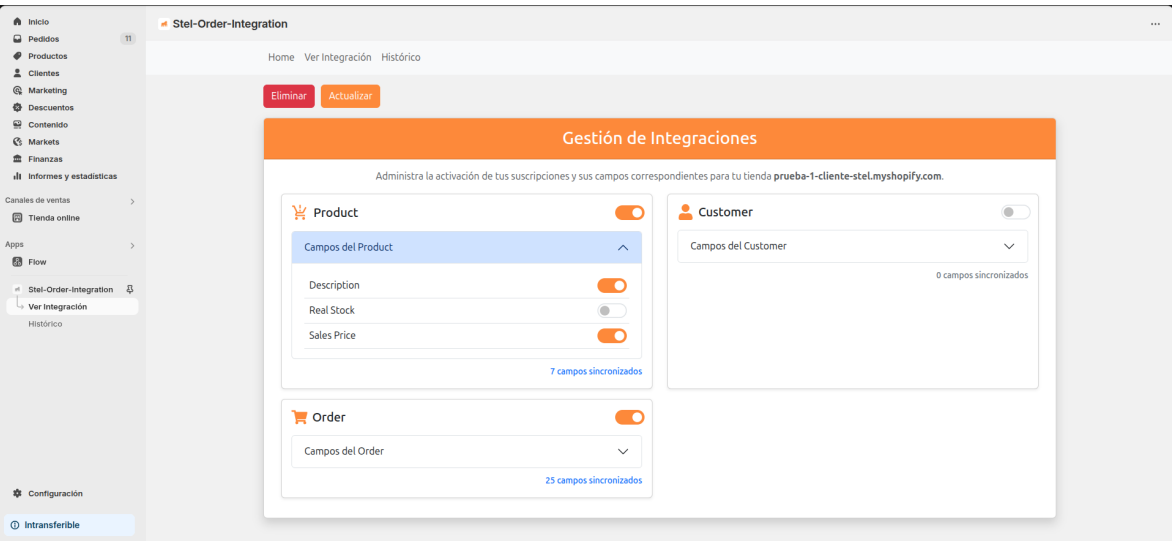


Figura 5.4: Pagina de configuración de la integración.

Historial de Eventos

Esta vista muestra una tabla con los eventos almacenados en el microservicio de *Shopify* (véase Figura5.5). Las columnas de la tabla incluyen el tópico o tipo de evento, la fecha y hora de creación, el estado del evento y la acción realizada sobre el mismo.

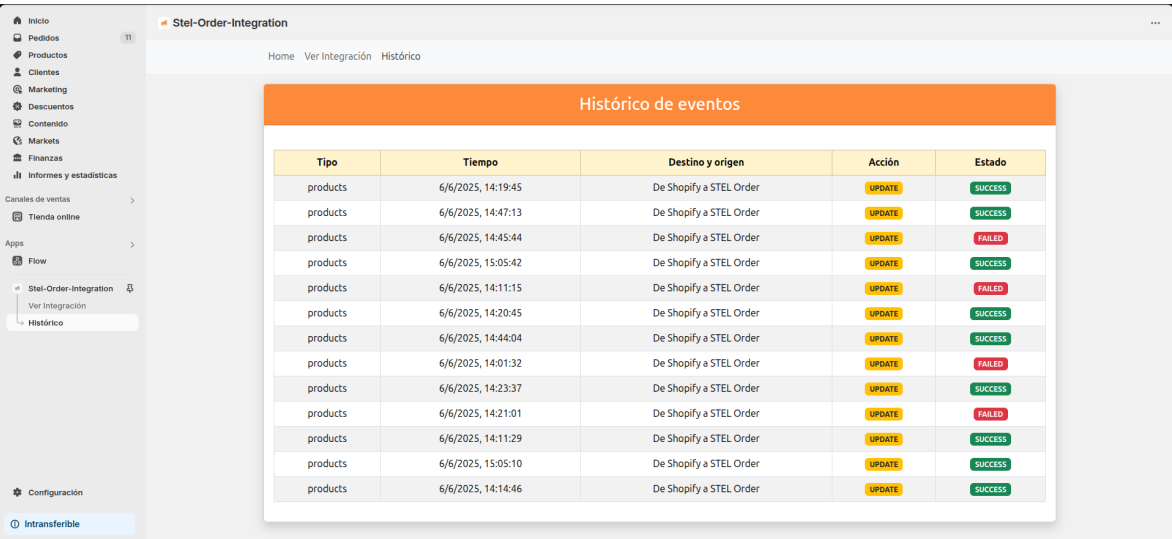


Figura 5.5: Vista del historial de eventos del usuario.

5.2.4 Sistema de Monitoreo Log

Para llevar un control de los eventos, interacciones, errores e inconsistencias en el *plugin* de *Shopify*, se ha implementado un sistema de monitoreo de logs. Para ello, se ha utilizado una biblioteca que permite registrar logs en diferentes ficheros según su nivel de severidad.

Se ha definido 3 tipos de logs:

- **Logs de Información:** Se utilizan para registrar información relevante sobre el funcionamiento del *plugin*, como la creación de integraciones, actualizaciones de suscripciones o eventos.
- **Logs de Errores:** Se emplean para registrar errores que ocurren durante la ejecución del *plugin*, como problemas de conexión con *Shopify* o errores al procesar eventos.
- **Logs de Inconsistencias:** Se emplean para registrar situaciones anómalas detectadas en el *plugin*. Por ejemplo, si ocurre un error al recibir el evento de desinstalación (*uninstalled*), el sistema podría quedar en un estado inconsistente, ya que la integración no se eliminaría correctamente de *STEL Order*.

Estos logs se almacenan en ficheros separados **app.log**, **errors.log** e **inconsistencies.log**, lo que permite una fácil identificación y análisis. Para tener una mayor escalabilidad y control se han centralizado los posibles mensajes de log en varios ficheros dentro de la carpeta **const**. También los mensajes de error utilizados dentro de la aplicación se han centralizado en un fichero dentro de este directorio.

5.3 Diagrama de Flujo del Sistema

Para poder comprender la lógica del sistema, se han incluido un diagrama de flujo sobre la emisión de eventos. Al ser uno de los flujos más importantes y largos del sistema, se ha dividido en dos partes. La primera parte muestra la lógica del *plugin* de *Shopify* y la segunda parte muestra la lógica del microservicio de *Shopify*.

En la Figura 5.6 se muestra el diagrama de flujo que ocurre en el *plugin* de *Shopify*, cuando se recibe un evento desde los *Webhooks* de *Shopify*. A grandes rasgos, el flujo en este componente se encarga de recibir el evento, verificar a qué integración pertenece a través del id del *Webhooks*, y enviar el evento al microservicio de *Shopify* para que este lo procese con el token de acceso.

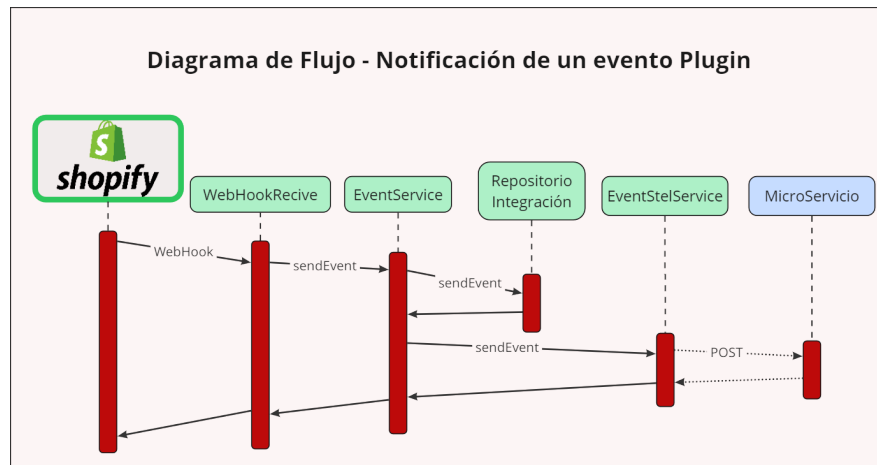


Figura 5.6: Diagrama de flujo del plugin de Shopify.

El diagrama de flujo del microservicio es demasiado extenso para mostrarlo en una sola imagen. Se ha dividido en dos partes. Además, hay partes del flujo que han sido omitidas para facilitar la comprensión. La primera parte, mostrada en la Figura 5.7, expone la recepción del evento desde el *plugin* de *Shopify*. Tras recibir el evento, se verifica que exista un suscriptor para el evento recibido y éste es emitido al bus interno de eventos.

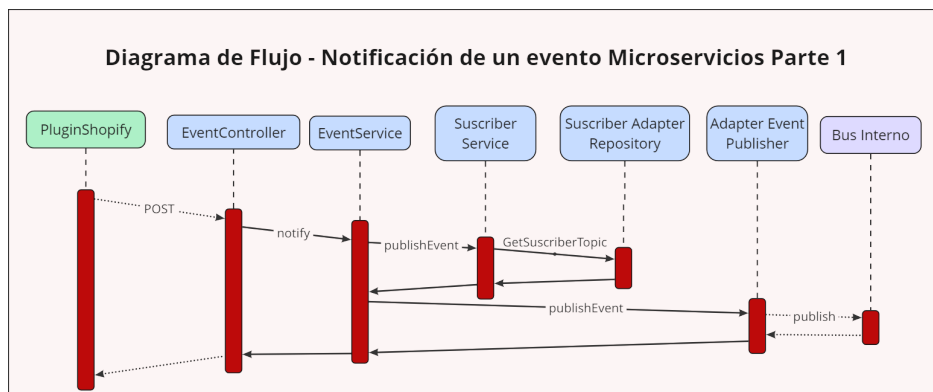


Figura 5.7: Diagrama de flujo del microservicio de Shopify - Parte 1.

Antes de su procesamiento, el evento se recupera del bus interno y se persiste en la base de datos del microservicio con el estado *PENDING*. Esto permite su reintento en caso de fallo. Una vez almacenado, se emite al bus de eventos para que sea gestionado por el listener.

La segunda parte, mostrada en la Figura 5.8, corresponde al procesamiento del evento recibido. Tras obtenerlo mediante el listener, se ejecuta la estrategia correspondiente

según el tópico, la plataforma y el tipo. Esta estrategia se encarga de transformarlo en un DTO, validarlo, mapearlo a una clase Plain Old Java Object (POJO) y llamar a la función correspondiente del servicio, pasando la clase del dominio con la información necesaria. El servicio ejecuta la lógica de negocio y utiliza el cliente para emitir el evento a *STEL Order*, donde será procesado. Finalmente, se actualiza su estado a *SUCCESS* o *FAILED*, según el resultado del envío.

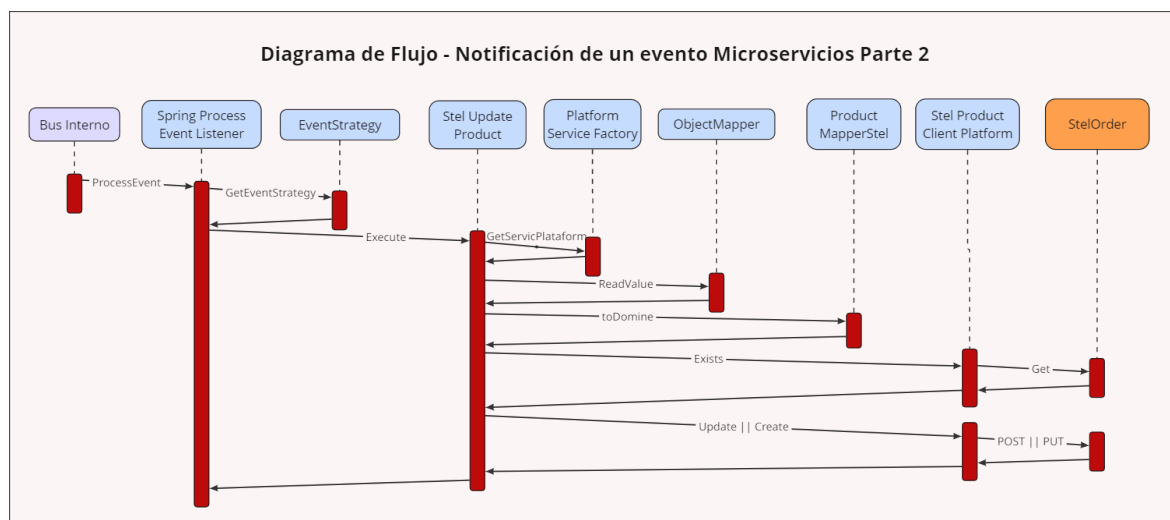


Figura 5.8: Diagrama de flujo del microservicio de Shopify - Parte 2.

5.4 Seguridad

Un aspecto fundamental en el desarrollo de este proyecto es la seguridad. Para garantizar la seguridad del sistema, se han implementado diferentes medidas tanto en el microservicio como en el *plugin*.

5.4.1 Seguridad del Microservicio de Shopify

En el microservicio de *Shopify*, se ha utilizado *Spring Security* para gestionar la seguridad del componente. Se ha implementado un sistema de autenticación y autorización basado en JWT para garantizar que solo los usuarios autorizados puedan acceder a los recursos del microservicio.

La generación del token JWT se realiza en la creación de la integración, porque es el punto donde el usuario se autentica. Para ello, se deben verificar tres aspectos:

- **API Key STEL Order:** Se verifica que la API Key de *STEL Order* sea correcta.

Y se obtiene el identificador de usuario de *STEL Order* para poder relacionar la integración con el usuario de *STEL Order*.

- **Permisos del Usuario:** Se comprueba que el usuario/empresa tenga los permisos necesarios para crear la integración. Si no tiene los permisos necesarios, se devuelve un error 403 Forbidden.
- **Token de acceso de Shopify:** Se revisa que el token de acceso de *Shopify* sea correcto haciendo uso de uno de los *endpoints* de la aplicación desarrollada en *Shopify*.

Una vez verificados estos aspectos, se genera el token JWT con el identificador de usuario de *STEL Order* y el identificador de la integración. Adicionalmente, se crea la integración en la base de datos del microservicio de *Shopify* y se almacena el token JWT y la integración en la base de datos de *Shopify*.

5.4.2 Seguridad del Plugin de Shopify

En este caso, la seguridad se gestiona como exige *Shopify* y viene establecida en la plantilla del *plugin*. Para ello, se utiliza la librería de *Shopify*, que tiene diferentes funciones para verificar si las peticiones provienen de *Shopify*.

Es importante explicar que cuando un usuario decide instalar el *plugin* de *Shopify*, esto inicia un sistema de autenticación **OAuth 2.0** [48]. Este sistema permite que el *plugin* de *Shopify* obtenga un token de acceso que se almacena en la base de datos del *plugin*. Este token de acceso se utiliza para autenticar las peticiones que se realizan entre el *plugin* y *Shopify*. El proceso de autenticación **OAuth 2.0** se muestra en la Figura 5.9.

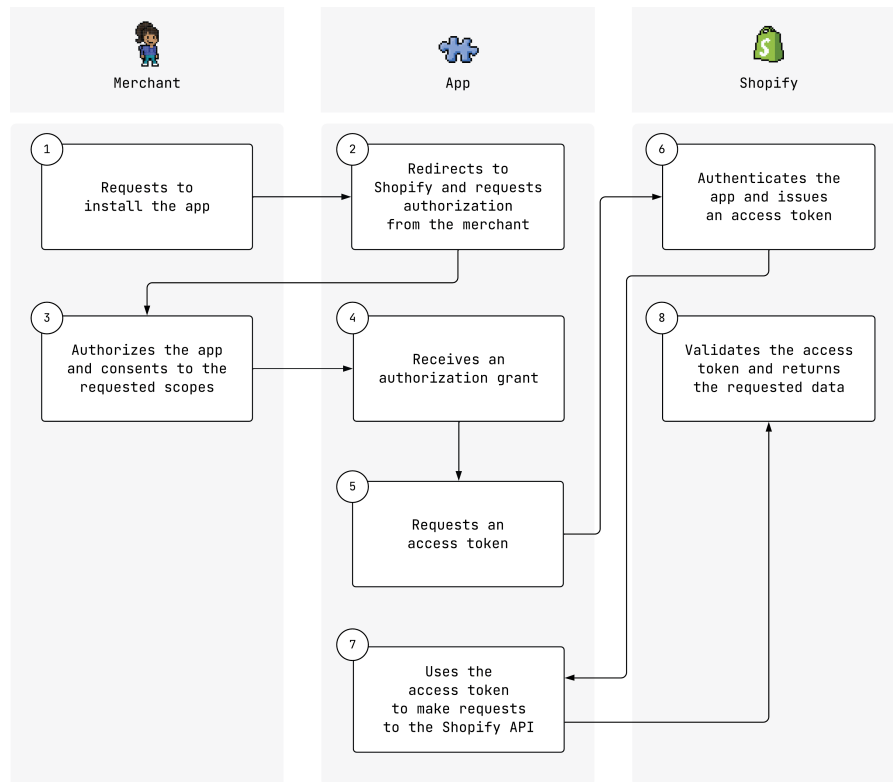


Figura 5.9: Proceso de autenticación OAuth 2.0 del plugin de Shopify.

Para garantizar que las peticiones realmente provienen de *Shopify*, se utiliza un sistema basado en tokens de sesión [49]. Este mecanismo se complementa con el sistema de autenticación **OAuth 2.0**, proporcionando una doble verificación de la autenticidad de las solicitudes. Tal y como se ilustra en la Figura 5.10, cuando *Shopify* realiza una petición al *plugin*, se envía un token de sesión que es validado en el propio *plugin* y, posteriormente, también se verifica el token de acceso a través de las APIs de *Shopify*.

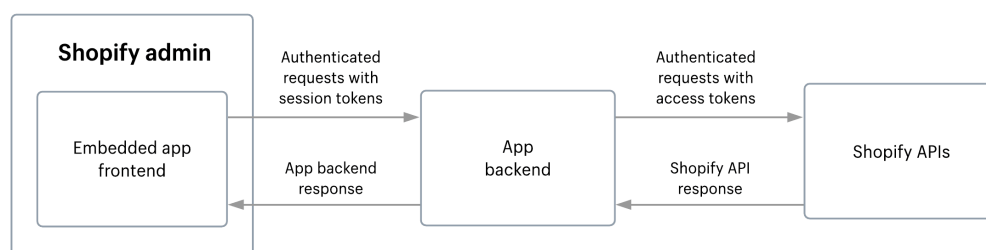


Figura 5.10: Proceso de verificación de sesión del plugin de Shopify.

Para llevar a cabo este proceso de verificación en el *plugin* de *Shopify*, se utiliza la librería oficial, que proporciona un objeto denominado **authenticate**. Este objeto ofrece cinco métodos principales que, tras autenticar la petición, devuelven la sesión del usuario correspondiente:

- **Admin:** Autentica las solicitudes realizadas desde la interfaz de administración de *Shopify*.
- **Flow:** Autentica las peticiones provenientes de extensiones de *Shopify Flow*.
- **FulfillmentService:** Autentica las solicitudes de servicios de logística de *Shopify*.
- **Public:** Se utiliza para autenticar peticiones públicas.
- **Webhook:** Autentica las peticiones enviadas por los *Webhooks*.

Una vez explicada la seguridad entre el *plugin* de *Shopify* y *Shopify*, es importante mencionar que este también se comunica con el microservicio. Para ello, se utiliza el token JWT generado en la creación de la integración. Este token se envía en las cabeceras de las peticiones al microservicio para autenticar las peticiones. Si el token no es correcto o ha expirado, se devuelve un error 401 Unauthorized.

Eso sí, el token JWT se genera en la creación la integración en el microservicio como se ha explicado anteriormente. Para asegurarse de que la solicitud de creación de la integración es segura, el microservicio consulta un *endpoint* del *plugin* para verificar que el token de acceso de *Shopify* es correcto para el dominio de la tienda.

5.5 Pruebas

En este capítulo se describen las pruebas realizadas al sistema y un plan de pruebas pendiente de realizar. Estas pruebas son fundamentales para garantizar el correcto funcionamiento del sistema y la calidad del mismo.

5.5.1 Pruebas de Integración/Contrato

Las pruebas de contrato son una técnica utilizada para verificar puntos de integración entre aplicaciones. Consisten en comprobar por separado que cada una de ellas envía y recibe mensajes que cumplen con un acuerdo previamente definido, conocido como “contrato” [50]. Se han realizado diferentes pruebas de integración sobre el *plugin* de *Shopify* y el microservicio de *Shopify*.

Ficheros Rest

Tanto para el microservicio de *Shopify* como para el *plugin* se han creado ficheros `.rest` que permiten realizar pruebas de contrato sobre los *endpoints* de ambos. Estas pruebas se pueden ejecutar desde los dos entornos de desarrollo, tanto en *Visual Studio Code* como en *IntelliJ IDEA*. Una ventaja de estas es que son muy fáciles de implementar y de ejecutar. De igual forma permiten probar los diferentes flujos de trabajo del sistema de manera rápida y sencilla.

Por contraparte, estas pruebas no se ejecutan de manera automática, por lo que es necesario ejecutarlas manualmente cada vez que se quiere probar el sistema. Asimismo, no son escalables a gran escala, ya que cada vez que se añade un nuevo *endpoint* o se modifica uno existente, es necesario modificar los ficheros `.rest` correspondientes.

Estas pruebas han resultado muy útiles para validar la aplicación durante las fases iniciales de desarrollo y para comprobar el funcionamiento de casos concretos. Sin embargo, no se han empleado para cubrir todos los flujos posibles ni de manera exhaustiva, por lo que no garantizan la detección de todos los errores o situaciones límite del sistema.

Pruebas Microservicio de Shopify

Para el microservicio de *Shopify* se han implementado pruebas de contrato utilizando las siguientes librerías: `Rest Assured`, `Spring Boot Test` y `Test containers`.

Con estas librerías se han implementado tres ficheros de pruebas de integración. Estos ficheros son: **EventControllerTest**, **IntegrationControllerTest** y **SubscriptionControllerTest**. Cada uno de estos ficheros contiene pruebas para todos los *endpoints* de su respectivo controlador.

La estructura de estos ficheros de pruebas es la siguiente:

- **Configuración:** En la configuración se define el contexto de la aplicación y se instancia el contenedor de *MySQL* para la base de datos. Esto permite que las pruebas se ejecuten en un entorno aislado y controlado, garantizando que no se afecten los datos reales de la base de datos.
 - **SetUp:** En este apartado se definen constantes y se inicializan los datos necesarios para las pruebas, métodos para refactorizar códigos de cada prueba y un método con la anotación `@BeforeEach`, que se ejecuta antes de cada prueba. Este método se encarga de eliminar los datos de la base de datos y de crear los datos necesarios para las pruebas. Esto garantiza que cada prueba se ejecute en un estado limpio y controlado, evitando interferencias entre pruebas.
-

- **Pruebas:** Las pruebas están divididas por grupos según el *endpoint* que se está probando. Cada grupo de pruebas contiene diferentes casos de prueba que verifican el correcto funcionamiento del *endpoint* en diferentes situaciones.

Estas pruebas se ejecutan de manera automática y se integran en el proceso de construcción del microservicio de *Shopify*. Esto permite que cada vez que se realice un cambio en el código, las pruebas se ejecuten automáticamente para verificar que todo sigue funcionando correctamente.

Uno de los principales problemas encontrados es que son muy lentas, ya que cada prueba se ejecuta de manera aislada y se conecta a la base de datos, son costosas de implementar y requieren un mantenimiento constante.

5.5.2 Plan de Pruebas

El plan de pruebas es un documento que describe las pruebas que se van a realizar sobre el sistema. Este se ha elaborado para garantizar que el sistema cumple con los requisitos funcionales y no funcionales establecidos en la fase de análisis.

A continuación, se describe en la tabla 5.1 con los roles a tener en cuenta en el plan de pruebas, las tareas que van a desempeñar y el porcentaje de participación dentro del plan.

Rol	Responsabilidad	Porcentaje
Ingeniero de Software	Responsable de identificar y corregir errores o bugs detectados, así como desarrollar las pruebas unitarias, seguridad, integración y End-to-End (E2E) necesarias para garantizar la calidad del sistema.	45%
Tester	Encargado de diseñar y ejecutar pruebas sobre el sistema, así como de reportar y dar seguimiento a los errores detectados. Responsable de las pruebas de usabilidad, aceptación, rendimiento y E2E.	40%
Scrum Master	Supervisión del proceso ágil, facilitación de priorización y resolución de bloqueos.	5%
Cliente	Proporcionar feedback sobre la usabilidad, funcionalidad y rendimiento del sistema y de las pruebas de aceptación.	10%

Tabla 5.1: Plan de pruebas de los recursos del proyecto.

Alcance del plan de pruebas

El plan de pruebas cubre todas las funcionalidades del sistema, tanto del microservicio de *Shopify* como del *plugin* de *Shopify*. Seguidamente se describen los tipos de pruebas que se llevarán a cabo:

- **Pruebas Unitarias:** Se desarrollarán pruebas unitarias para las clases del microservicio y del *plugin* de *Shopify*, enfocándose en verificar el correcto funcionamiento de cada clase y método de forma aislada. En aquellos servicios o clases con mayor complejidad o propensión a errores, se implementarán pruebas de caja blanca.
- **Pruebas de Seguridad:** Utilizando la herramienta *Owasp ZAP*[51], se analizarán posibles vulnerabilidades tanto en el microservicio como en el *plugin* de *Shopify*. Se realizarán escaneos pasivos y activos, filtrando los falsos positivos.
- **Pruebas de Integración:** Se contemplan diferentes enfoques de pruebas integración:
 - **Ficheros Rest:** Como se mencionó anteriormente, se crearán ficheros `.rest` para probar los distintos *endpoints* del microservicio y del *plugin* de *Shopify*.
 - **Pruebas de contrato del microservicio de Shopify:** Se implementarán pruebas automáticas para validar el comportamiento de los *endpoints*, tal y como se ha explicado previamente.
- **Pruebas de Usabilidad:** Se llevarán a cabo pruebas con usuarios reales para evaluar la facilidad de uso y la experiencia de usuario en el *plugin* de *Shopify*, centrándose en la navegación, comprensión de funcionalidades y satisfacción general.
- **Pruebas de Aceptación:** Se realizarán junto al cliente para comprobar que el sistema cumple con los requisitos funcionales y no funcionales definidos en la fase de análisis.
- **Pruebas de Rendimiento:** Se empleará la herramienta *JMeter*[52] para medir el tiempo de respuesta, la capacidad de carga y la escalabilidad tanto del microservicio como del *plugin* de *Shopify*.
- **Pruebas E2E:** Se implementarán pruebas E2E utilizando *Cypress*[53] para verificar el correcto funcionamiento del *plugin* de *Shopify* en su conjunto, incluyendo la interacción con el microservicio de *Shopify* y la gestión de eventos.

Fuera del alcance del plan de pruebas

El plan de pruebas excluye ciertos tipos de estas que no se consideran prioritarias para este sistema. Estas son:

- **Pruebas de Regresión:** Debido al elevado coste de ejecución de todas las pruebas del proyecto completo, se ha decidido no incluir pruebas de regresión en esta fase.
 - **Pruebas exhaustivas de base de datos:** No se realizarán pruebas detalladas sobre las bases de datos, salvo en aquellos casos en los que la funcionalidad lo requiera.
 - **Pruebas de Mutantes:** No se contemplan pruebas de mutantes, ya que su implementación resulta costosa y están orientadas a detectar errores en código crítico, lo cual no se considera necesario en este contexto.
-

6 Conclusiones y vías futuras

6.1 Conclusiones

Tras finalizar una primera versión del proyecto, se puede concluir que se han cumplido los objetivos planteados al inicio del mismo. Se ha desarrollado un plugin y un microservicio que permiten a los usuarios sincronizar parte de la información esencial de *Shopify* a *STEL Order*. Esta integración facilita a los usuarios seleccionar qué entidades y atributos desean transferir, lo que les permite personalizar la experiencia según sus necesidades. Además, se ha logrado subsanar problemas de compatibilidad entre entidades de *Shopify* y *STEL Order*, como ha sucedido con las variantes de productos, que no existían en *STEL Order* y se han adaptado para su correcta sincronización. Todo esto se ha realizado siguiendo el patrón de arquitectura limpia para garantizar la escalabilidad y mantenibilidad del código.

Es evidente que es de un alto interés el desarrollo del proyecto, ya que se han podido aplicar la mayoría de los conocimientos adquiridos durante el máster y la carrera, desde la gestión ágil de proyectos, pasando por el desarrollo *full stack* y arquitectura de datos, con tecnologías como *React*, *TypeScript*, *Java*, *MySQL*; hasta desarrollo de software seguro, control de calidad y pruebas. Además, he podido trabajar en un entorno profesional, lo que me ha permitido mejorar mis habilidades de comunicación y colaboración en equipo.

Aunque el proyecto ha cumplido con los objetivos iniciales, realmente es un proyecto en una fase inicial de desarrollo. Todavía quedan por implementar muchas funcionalidades y mejoras que se han identificado durante el proceso de desarrollo. Por ejemplo, la implementación de una vista de configuración para que el usuario introduzca campos necesarios para la transferencia de datos, la mejora de la interfaz de usuario del plugin, etc. De igual modo el proyecto debe de someterse al plan de pruebas implementado, para garantizar su correcto funcionamiento y seguridad.

Uno de los aspectos que no se ha podido abordar en esta primera versión ha sido el despliegue de los componentes del proyecto en la nube. Aunque se ha avanzado en la arquitectura y el diseño de los microservicios, no se ha llegado a implementar su despliegue en un entorno de pruebas o producción. Tampoco se ha podido aplicar Continuous Integration/Continuous Deployment (CI/CD) para automatizar el proceso de despliegue y pruebas, lo que sería fundamental para mejorar la eficiencia en el desarro-

llo y el mantenimiento del proyecto a largo plazo.

6.2 Vías Futuras

Como se ha mencionado anteriormente, la aplicación que se ha desarrollado es una versión inicial de un proyecto a gran escala. Por lo tanto, existen muchas vías futuras que se pueden explorar para mejorar y ampliar la funcionalidad del mismo. Voy a dividir estas vías en dos categorías: proyecto específico y proyecto general.

6.2.1 Vías Futuras Proyecto Específico

Para dar continuidad al proyecto denominado comercialmente como **Integración 360 Shopify STEL Order**, se pueden considerar las siguientes vías futuras:

- **Bidireccionalidad de la integración:** Implementar la capacidad de transferir datos desde *STEL Order* a *Shopify*, lo que permitiría una sincronización completa entre ambas plataformas.
- **Ampliación de entidades y atributos:** Incluir más entidades y atributos en la transferencia de datos, como facturas, categorías de productos, clientes potenciales, etc. Esto mejoraría la funcionalidad del plugin y permitiría una integración más completa entre *Shopify* y *STEL Order*.
- **Sincronización manual:** Permitir que los usuarios puedan solicitar una sincronización automática de los datos en cualquier momento.
- **Implementación de CI/CD:** Configurar un *pipeline* de integración y despliegue continuo para automatizar el proceso de pruebas y despliegue, lo que mejorará la eficiencia del desarrollo.

6.2.2 Vías Futuras Proyecto General

Amén de las vías específicas del proyecto, existen otros proyectos que parten de la base del trabajo realizado y que pueden ser de interés para el futuro:

- **Desarrollo del sistema Veryfactu:** Implementar un sistema que sea capaz de generar facturas y ticarlas en *STEL Order* a partir de los pedidos realizados en *Shopify*.
 - **Integración con otros sistemas:** Ampliar la integración a otras plataformas de comercio electrónico, como *Prestashop*, *Magento*, etc. Además, existe el interés de integrar *STEL Order* con otros sistemas de gestión de email, facturas, etc.
-

- **Desarrollo de un sistema de WebHooks:** Implementar un sistema de *WebHooks* que permita a los propios proyectos y *Partners* de *STEL Order* recibir notificaciones en tiempo real sobre eventos importantes sucedidos en la plataforma.
-

Bibliografía

- [1] Flowlu Team. Apps for small and medium business. 2025. URL <https://www.flowlu.com/es/blog/productivity/apps-for-small-and-medium-business/>. Accedido el 22 de abril de 2025.
- [2] Centac Team. Tipos de software indispensables para cualquier pyme o pequeño negocio. 2017. URL <https://www.centac.es/top-tecnologia/software/software-para-cualquier-pyme-o-pequeno-negocio>. Accedido el 22 de abril de 2025.
- [3] STEL Team. Stel order. software de gestión para empresas. 2025. URL <https://www.stelorder.com/>. Accedido el 26 de abril de 2025.
- [4] Shopify Team. Shopify. 2023. URL <https://www.shopify.com/es-es>. Accedido el 30 de mayo de 2025.
- [5] Murcia EP. Exalumnos de la upct y fundadores de stel solutions, ultiman el lanzamiento de una “app” que ayuda a emprendedores. 2014. URL <https://www.europapress.es/murcia/noticia-exalumnos-upct-fundadores-stel-solutions-ultiman-lanzamiento-app-ayuda-emprendedores-20140226104249.html>. Accedido el 26 de abril de 2025.
- [6] Oligal. Stel order: el software de gestión mejor valorado en España. 2021. URL <https://www.oligal.com/stel-order-software-gestion-mejor-valorado-espana>. Accedido el 26 de abril de 2025.
- [7] STEL Team. Mi stel shop. centro de ayuda de stel order. 2025. URL <https://help.stelorder.com/hc/es/articles/7105259750429-Mi-STEL-Shop>. Accedido el 26 de abril de 2025.
- [8] Tempo Team. Tempo. stel order. 2025. URL <https://www.stelorder.com/tempo/>. Accedido el 26 de abril de 2025.
- [9] STEL Team. Stel order api documentation. 2025. URL <https://app.stelorder.com/app/indexapi.html>. Accedido el 29 de abril de 2025.

- [10] Red Hat Team. ¿qué es un webhook y para qué sirve? 2024. URL <https://www.redhat.com/es/topics/automation-and-management/que-es-un-webhook>. Accedido el 30 de mayo de 2025.
 - [11] GraphQL Team. GraphQL | a query language for your api. 2024. URL <https://graphql.org/>. Accedido el 30 de mayo de 2025.
 - [12] Creacom Team. Shopify: Guía completa para principiantes en 2024. 2024. URL <https://www.creacom.es/blog/guia-shopify-principiantes/>. Accedido el 29 de abril de 2025.
 - [13] Alhsis Team. ¿qué es shopify y por qué es popular para ecommerce? 2025. URL <https://www.alhsis.com/faq/que-es-shopify-y-por-que-es-popular-para-ecommerce>. Accedido el 3 de mayo de 2025.
 - [14] Shopify Team. GraphQL admin api reference. 2025. URL <https://shopify.dev/docs/api/admin-graphql>. Accedido el 26 de abril de 2025.
 - [15] Shopify Team. GraphQL admin api reference. 2025. URL <https://shopify.dev/docs/api/admin-graphql>. Accedido el 26 de abril de 2025.
 - [16] Shopify Team. About webhooks. 2025. URL <https://shopify.dev/docs/apps/build/webhooks>. Accedido el 26 de abril de 2025.
 - [17] Shopify Team. Checklist of requirements for apps in the shopify app store. 2021. URL <https://shopify.dev/docs/apps/launch/app-requirements-checklist>. Accedido el 3 de mayo de 2025.
 - [18] Shopify Team. Build a shopify app using remix. 2021. URL <https://shopify.dev/docs/apps/build/build?framework=remix>. Accedido el 3 de mayo de 2025.
 - [19] IBM Team. ipaas. 2021. URL <https://www.ibm.com/es-es/topics/ipaas>. Accedido el 3 de mayo de 2025.
 - [20] Latenode Team. Las 5 mejores soluciones ipaas en 2024. 2024. URL <https://latenode.com/es/blog/top-5-ipaas-solutions>. Accedido el 3 de mayo de 2025.
 - [21] Zapier Team. Build automated workflows with zapier. 2025. URL <https://zapier.com/workflows>. Accedido el 30 de mayo de 2025.
 - [22] González D. Qué es, cómo funciona y ejemplos de uso. 2023. URL <https://davizgonzalez.com/blog/que-es-zapier/>. Accedido el 3 de mayo de 2025.
-

-
- [23] Microsoft Team. Business central | microsoft dynamics 365. 2024. URL <https://www.microsoft.com/es-es/dynamics-365/products/business-central>. Accedido el 30 de mayo de 2025.
- [24] Holded Team. Holded - el software de gestión para pymes y emprendedores. 2025. URL <https://www.holded.com/es>. Accedido el 30 de mayo de 2025.
- [25] GeeksforGeeks. Getting started with remix. 2024. URL <https://www.geeksforgeeks.org/getting-started-with-remix/>. Accedido el 4 de mayo de 2025.
- [26] Yanina M. Qué es spring framework y por qué usarlo. 2018. URL <https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/>. Accedido el 23 de abril de 2025.
- [27] Kinsta®. ¿qué es mysql? una explicación para principiantes. 2022. URL <https://kinsta.com/es/base-de-conocimiento/que-es-mysql/>. Accedido el 24 de abril de 2025.
- [28] Prisma Team. What is prisma orm? (overview) | prisma documentation. 2020. URL <https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma>. Accedido el 10 de mayo de 2025.
- [29] Russo F. Atlassian, el gigante del software para equipos de trabajo. 2018. URL <https://tentulogo.com/atlassian-el-gigante-del-software-para-equipos-de-trabajo/>. Accedido el 11 de mayo de 2025.
- [30] IONOS Team. ¿qué es jira? todo lo que necesitas saber sobre el software de gestión de proyectos. 2022. URL <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-jira/>. Accedido el 11 de mayo de 2025.
- [31] GetGuru Team. Confluence: A comprehensive guide. 2025. URL <https://www.getguru.com/reference/what-is-confluence>. Accedido el 11 de mayo de 2025.
- [32] GoDaddy Team. Bitbucket, ¿qué es, para qué sirve y cómo funciona? 2023. URL <https://www.godaddy.com/resources/es/crearweb/bitbucket-como-funciona>. Accedido el 11 de mayo de 2025.
- [33] Rodrigues A. Applying natural language processing (nlp) in user stories on agile software development: Uses, techniques and challenges. 2021. URL <https://medium.com/@anthonini/applying-natural-language-processing-nlp-in-user-stories-on-agile/-software-development-uses-ab857dde0e5a>. Accedido el 14 de mayo de 2025.
-

- [34] Scrum Manager. Scrum master. temario troncal 1. versión 4. 2024. URL <https://scrummanager.com/website/c/info/topic.php?keytheme=1000>. Accedido el 19 de mayo de 2025.
 - [35] Atlassian Team. Qué es scrum y cómo empezar. 2025. URL <https://www.atlassian.com/es/agile/scrum>. Accedido el 12 de mayo de 2025.
 - [36] Asana Team. ¿qué es la metodología kanban y cómo funciona? 2025. URL <https://asana.com/es/resources/what-is-kanban>. Accedido el 20 de mayo de 2025.
 - [37] Chiyana S. Moscow. ¿qué es y cómo priorizar en el desarrollo de tu aplicación? 2020. URL <https://www.itdo.com/blog/moscow-que-es-y-como-priorizar-en-el-desarrollo-de-tu-aplicacion/>. Accedido el 19 de mayo de 2025.
 - [38] Marcos M. Tema 2: Servicios rest - introducción. 2024. Accedido el 10 de mayo de 2025.
 - [39] Philipp Hauer. Restful api design. best practices in a nutshell. 2015. URL <https://phauer.com/2015/restful-api-design-best-practices/>. Accedido el 18 de mayo de 2025.
 - [40] Elbloddeldev. Arquitectura limpia: Fundamentos y aplicación en desarrollo. 2025. URL <https://elblogdelprogramador.com/posts/arquitectura-limpia-fundamentos-y-aplicacion-en-desarrollo/#gsc.tab=0>. Accedido el 23 de mayo de 2025.
 - [41] Jiménez F. Arquitecturas limpias y ddd, un recorrido práctico. parte 1. 2021. URL <https://sixphere.com/blog/arquitecturas-limpas-ddd-parte-1/>. Accedido el 10 de mayo de 2025.
 - [42] Uncle Bob. The clean architecture. 2012. URL <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Accedido el 23 de mayo de 2025.
 - [43] DBeaver Team. Dbeaver community | free universal database tool. 2025. URL <https://dbeaver.io/>. Accedido el 26 de mayo de 2025.
 - [44] Canva. Canva. 2020. URL <https://www.canva.com/>. Accedido el 14 de mayo de 2025.
 - [45] Iluwatar. Factory. patrones de diseño java. 2025. URL <https://java-design-patterns.com/es/patterns/factory/>. Accedido el 26 de mayo de 2025.
-

-
- [46] Iluwatar. Strategy. patrones de diseño java. 2025. URL <https://java-design-patterns.com/es/patterns/strategy/>. Accedido el 26 de mayo de 2025.
 - [47] Iluwatar. Adapter. patrones de diseño java. 2025. URL <https://java-design-patterns.com/es/patterns/adapter/>. Accedido el 27 de mayo de 2025.
 - [48] Shopify Team. About token acquisition. 2021. URL <https://shopify.dev/docs/apps/build/authentication-authorization/access-tokens/>. Accedido el 29 de mayo de 2025.
 - [49] Shopify Team. About session tokens. 2021. URL <https://shopify.dev/docs/apps/build/authentication-authorization/session-tokens>. Accedido el 29 de mayo de 2025.
 - [50] Pact Team. Introduction | pact docs. 2022. URL <https://docs.pact.io/>. Accedido el 2 de junio de 2025.
 - [51] ZAP Team. The zap homepage. 2025. URL <https://www.zaproxy.org/>. Accedido el 13 de junio de 2025.
 - [52] Apache JMeter Team. Apache jmeter - apache jmetertm. 2024. URL <https://jmeter.apache.org/>. Accedido el 13 de junio de 2025.
 - [53] Cypress Team. Testing frameworks for javascript | write, run, debug | cypress. 2025. URL <https://www.cypress.io/>. Accedido el 13 de junio de 2025.
 - [54] Shopify Team. Build. shopify. 2025. URL <https://shopify.dev/docs/apps/build>. Accedido el 26 de abril de 2025.
 - [55] LucusHost. Qué es node.js y para qué sirve. 2022. URL https://www.lucushost.com/blog/que-es-node-js/#Que_es_Nodejs. Accedido el 23 de abril de 2025.
 - [56] Java. ¿qué es java y para qué sirve? 2024. URL https://www.java.com/es/download/help/whatis_java.html. Accedido el 23 de abril de 2025.
 - [57] Deyimar A. Qué es react: definición, características y funcionamiento. 2020. URL <https://www.hostinger.es/tutoriales/que-es-react>. Accedido el 24 de abril de 2025.
 - [58] IBM. ¿qué es mongodb? 2024. URL <https://www.ibm.com/es-es/topics/mongodb>. Accedido el 24 de abril de 2025.
 - [59] IBM Team. Redis. 2021. URL <https://www.ibm.com/mx-es/topics/redis>. Accedido el 10 de mayo de 2025.
-

- [60] Microsoft. Visual studio: Ide y editor de código para desarrolladores de software y teams. 2020. URL <https://visualstudio.microsoft.com/es/#:~:text=Visual%20Studio%20Code,casi%20cualquier%20lenguaje%20de%20programaci%C3%B3n>. Accedido el 25 de abril de 2025.
- [61] Vidal S. ¿qué es intelliJ idea? 2023. URL <https://tecnobits.com/que-es-intellij-idea/>. Accedido el 10 de mayo de 2025.
-

Lista de Acrónimos y Abreviaturas

API	Application Programming Interface.
CI/CD	Continuous Integration/Continuous Deployment.
CRM	Customer Relationship Management.
DTO	Data Transfer Object.
E2E	End-to-End.
ERP	Enterprise Resource Planning.
ESB	Enterprise Service Bus.
HTTP	Hypertext Transfer Protocol.
IEEE	Institute of Electrical and Electronics Engineers.
iPaaS	Integration Platform as a Service.
JSON	JavaScript Object Notation.
JWT	JSON Web Token.
MVP	Minimum Viable Product.
ORM	Object-Relational Mapping.
POJO	Plain Old Java Object.
PYMES	Pequeñas y Medianas Empresas.
QA	Quality Assurance.
REST	Representational State Transfer.
SOLID	Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation and Dependency Inversion.
TFG	Trabajo de Fin de Grado.
TFM	Trabajo de Fin de Máster.
UI	User Interface.
URL	Uniform Resource Locator.