Hromek, Emil

Aportes a la mejora en la certificación del estado final de un envío en logística

2024

Instituto de Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática

Práctica Profesional Supervisada

Informe final

Aportes a la mejora en la certificación del estado final de un envío en logística

Hromek, Emil

Florencio Varela, noviembre de 2024

PRÁCTICA PROFESIONAL SUPERVISADA (PPS)

Aportes a la mejora en la certificación del estado final de un envío en logística

DATOS DEL ESTUDIANTE

Apellido y Nombres: Hromek, Emil

DNI: 36763595

N.º de Legajo: 34864

Correo electrónico: emil233@live.com.ar

Cantidad de materias aprobadas al comienzo de la PPS: 44

PPS enmarcada en el artículo 7 de la Resolución (CS) 103/16

DOCENTE SUPERVISOR

Apellido y Nombres: Dr. Ing. Gross, Patricio Martín

Correo electrónico: pgross@unaj.edu.ar

DOCENTE TUTOR DEL TALLER DE APOYO A LA PRODUCCIÓN DE TEXTOS ACADÉMICOS DE LA UNAJ

Apellido y Nombres: N/A

DATOS DE LA ORGANIZACIÓN DONDE SE REALIZÓ LA PPS

Nombre o Razón Social: Correo Andreani S.A.

Dirección: Av. Leandro N. Alem 639 P. 7 Dpto. 1 (1001) CABA

Teléfono: 0800-122-1112

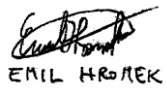
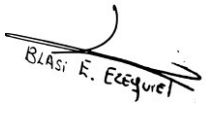
Sector: Integraciones

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: Blasi, Eduardo Ezequiel

Correo electrónico: eblasi@andreani.com

FIRMA DEL COORDINADOR DE LA CARRERA

Firma Estudiante:	Firma Docente Supervisor:	Firma Tutor Organizacional:
		

RESUMEN

En el presente informe se detalla todo el proceso de la PPS (Práctica Profesional Supervisada) realizada en la empresa Correo Andreani S. A., cuyo título elegido para la misma es «Aportes a la mejora en la certificación del estado final de un envío en logística».

El objetivo de este trabajo fue un proceso de migración tecnológica por razones de eficiencia operativa, aplicado a un sistema generador de certificados del estado final de un envío, llamado «Constancias Electrónicas», los cuales se utilizan principalmente para auditorías. En caso de ser necesario, también pueden ser solicitados por clientes. Se puede entender la migración como una reingeniería del sistema, ya que tanto la entrada del sistema (los datos de los envíos) como la salida (las constancias electrónicas) se mantienen intactas, en primera instancia.

El sistema legado, que al momento de redactar este informe todavía sigue en funcionamiento de forma paralela al nuevo sistema y está a la espera de ser apagado, está desarrollado enteramente utilizando tecnologías web, y lo mismo se hizo para el nuevo desarrollo. Los aportes hechos en este trabajo corresponden a la implementación de la parte de *backend* (del inglés, es un término referido a la parte del sistema que maneja la lógica de negocio, la base de datos y la comunicación con el servidor) [1].

Lo primero que se hizo fue relevar el sistema actual, luego diseñar el nuevo en base a las necesidades del negocio, para luego finalmente realizar los procesos de desarrollo y puesta en producción.

ABSTRACT

This report details the entire process of the SPP (Supervised Professional Practice) carried out at Correo Andreani S.A., whose chosen title is “Contributions to the improvement of the certification of the final state of a shipment in logistics”.

The objective of this work was a technology migration process for reasons of operational efficiency, applied to a system that generates certificates of the final state of a shipment, called “Electronic Certificates”, which are mainly used for audits. If necessary, they can also be requested by clients. The migration can be understood as a reengineering of the system, since both the input of the system (the shipment data) and the output (the electronic certificates) remain the same, at least initially.

The legacy system, which at the time of writing this report is still running alongside the new one and is awaiting shutdown, is fully developed using web technologies, and the same approach has been applied to the new development. The contributions made in this work are focused on the backend implementation. Backend is a term referring to the part of the system that handles business logic, the database, and communication with the server.

The first step was to analyze the current system, then design the new one based on the needs of the business, and then finally carry out the development and deployment processes.

DEDICATORIAS Y AGRADECIMIENTOS

Mi agradecimiento y dedicatoria de este trabajo es a quienes contribuyeron a que el mismo sea posible.

A mi tutor Dr. Ing. Patricio Gross por haberme orientado en la realización de este, su guía ha sido invaluable e indispensable, así como también al coordinador de la carrera, Dr. Ing. Martín Morales, por su respaldo constante y compromiso con los estudiantes.

A los docentes de la carrera, que con su dedicación, paciencia y conocimiento fueron una parte fundamental y decisiva en mi formación profesional.

Al personal de Grupo Logístico Andreani y de OpenDev Pro, quienes participaron y colaboraron conmigo en este proyecto, con todo su apoyo e intercambio de conocimientos que fueron fundamentales para su culminación.

A mis compañeros de la universidad, con quienes tuve gratos intercambios de ideas y conocimientos en estos años, así como también momentos de compañerismo y apoyo mutuo, que hicieron el recorrido académico más llevadero.

Por último, a mi familia y amigos de toda la vida, quienes me respaldaron en todo sentido durante estos años. Su aliento constante fue fundamental para llegar a esta instancia.

Todos fueron parte de este recorrido, de diversas maneras, contribuyendo a mi crecimiento profesional y personal. Por eso, les extiendo mi más sincero agradecimiento.

CONTENIDO

INTRODUCCIÓN

Grupo Logístico Andreani es una empresa conocida de nuestro país, con presencia desde el año 1945, cuyo negocio se basa principalmente en soluciones logísticas integrales.

En los últimos tiempos ha experimentado una transformación en lo que respecta a tecnología, lo que incluye un desarrollo constante para cumplir con las necesidades del negocio y hacerlo competitivo. Este desarrollo implica la implementación de sistemas informáticos que permitan cumplir con este objetivo, así como también mantenerlos actualizados. En el presente trabajo se realizó la migración de tecnologías de un sistema llamado «Constancias Electrónicas», cuya función es generar una certificación del estado final de un envío. Este sistema está en funcionamiento desde el año 2018. Puede parecer poco, pero es un periodo significante en los tiempos que se manejan en el ámbito tecnológico. El objetivo general fue actualizar las tecnologías y arquitectura que utiliza el mismo, por cuestiones de eficiencia operativa, incluyendo costos de licencias (puntualmente de una tecnología que utiliza, y de esto se hablará más adelante en detalle).

A grandes rasgos, lo que hace el sistema es capturar ciertos eventos de la cadena de envíos (esto es el conjunto de eventos que conforman un envío), enriquecerlos con datos provenientes de diversos sistemas que tiene la empresa y finalmente generar archivos con información relevante, los cuales se guardan en almacenamientos.

Estos certificados tienen un uso tanto para auditorías por parte de la empresa como de parte de los clientes, ya que en ciertos casos los requieren o pueden requerirlos.

En este informe, se presentará primero una descripción del sistema actual, para que se entienda de qué se está hablando, en lo que respecta a tecnologías, arquitectura y funcionamiento general del mismo. Luego se explicarán las motivaciones que impulsaron la migración de este, profundizando lo mencionado al principio de esta sección.

Concluido esto se explicará el proceso general del desarrollo del nuevo sistema, seguido por un resumen de características de este y una conclusión del trabajo.

El informe pretende ser principalmente cualitativo, ya que es una descripción a grandes rasgos de todo el proceso realizado. No se pretendía hacer un informe que incluya aportes cuantitativos, no obstante, se incluyeron algunos datos numéricos relacionados a métricas del sistema.

Los términos y expresiones marcados en *itálica (o cursiva)* hacen referencia a términos en inglés, y su traducción se proporciona inmediatamente a continuación de estos. Los que están marcados con letra **negrita**, en la medida que se consideró conveniente, se refieren a terminología, entidades y tecnologías utilizadas por el sistema.

DESCRIPCIÓN DEL SISTEMA ACTUAL

INTRODUCCIÓN AL SISTEMA ACTUAL

Las Constancias Electrónicas son documentos digitales que certifican que un envío llegó a un determinado estado de finalización, ya sea que fue entregado exitosamente o que la entrega fracasó definitivamente. Los datos para generarlas provienen de diversas fuentes (por ejemplo: eventos provenientes de la cadena de envío, bases de datos, sistemas de la empresa, contenedores de archivos, entre otras) y pueden generarse en los formatos **HTML**, **JPG** y **PDF**. Definiciones de estos formatos:

- **HTML** es la sigla de *HyperText Markup Language* (Lenguage de marcado de hipertexto). Es un lenguaje de marcado típicamente usado para el desarrollo de páginas web, las cuales se pueden abrir con cualquier navegador web [2].
- **JPG** es la sigla de *Joint Photographic Experts Group* (Grupo conjunto de expertos en fotografía). Este grupo es el creador de este popular formato de imágenes [3].
- **PDF** es la sigla de *Portable Document Format* (Formato de documento portable), y es un popular formato para documentos en general [4].

Los archivos se guardan en contenedores de **Azure Blob Storage** (una tecnología de almacenamiento de archivos, explicada más adelante), así como su contenido de imágenes, firmas y datos adicionales. También se almacenan en servidores a los que se accede mediante el protocolo **FTP** (*File Transfer Protocol*, Protocolo de transferencia de archivos) [5].

El servicio actualmente genera constancias para cada envío, en base a ciertos eventos:

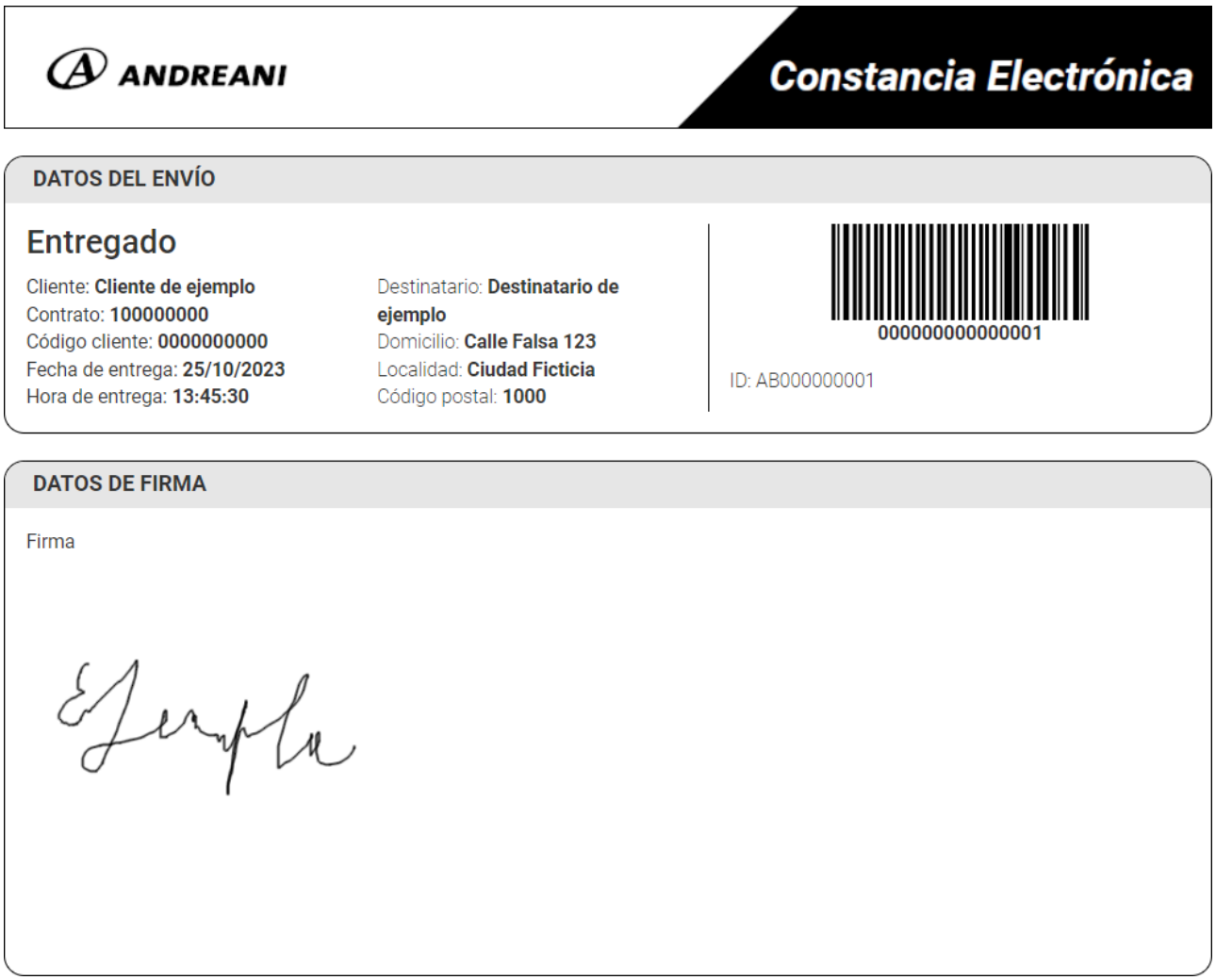
- **Visita**
- **Envío entregado**
- **Envío no entregado**

- **Rectificación de motivo**

Es importante aclarar que no todos los eventos de los tipos mencionados pueden generar una constancia. Por ejemplo: si en una visita el envío no fue entregado y quedan más instancias de esta, entonces no se genera una constancia. Similarmente, si al rectificar el motivo de un evento no se confirma la entrega, tampoco se genera una constancia.

Las constancias contienen una estructura de secciones (en forma de módulos), las cuales muestran información del envío. Algunos de estos módulos son de inclusión obligatoria. Otros son opcionales, y su inclusión depende de lo especificado en el contrato con un cliente puntual [6].

Un ejemplo típico de constancia electrónica es el de la siguiente página (los datos son a modo de ejemplo).


ANDREANI

Constancia Electrónica

DATOS DEL ENVÍO

Entregado

Cliente: **Cliente de ejemplo**
Contrato: **100000000**
Código cliente: **0000000000**
Fecha de entrega: **25/10/2023**
Hora de entrega: **13:45:30**

Destinatario: **Destinatario de ejemplo**
Domicilio: **Calle Falsa 123**
Localidad: **Ciudad Ficticia**
Código postal: **1000**

000000000000001

ID: AB000000001

DATOS DE FIRMA

Firma

Figura 1: Ejemplo típico de constancia, primera parte

Figura 2: Ejemplo típico de constancia, segunda parte

Detalles de los módulos:

- **Módulo datos de envío**: es obligatorio para todas las constancias. Especifica si el envío fue entregado o no, motivo, cliente, contrato, código de cliente, fecha y hora de entrega, destinatario, su domicilio (incluyendo localidad y código postal), también un código de barras que identifica al número de envío, así como también identificadores asociados al cliente. Se muestra un ejemplo en la figura 1.
- **Módulo firma**: muestra imágenes de la firma del destinatario. Se muestra un ejemplo en la figura 1.
- **Módulo geolocalización** (opcional): muestra una imagen tomada de **Google Maps** [7], con el domicilio marcado, también las coordenadas geográficas y el enlace al mapa. Se muestra un ejemplo en la figura 2.

- **Módulo detalle del envío**: incluye información adicional del envío, que no viene en los eventos consumidos por el sistema. Si bien no es opcional para el cliente, se incluye solamente si en la información del envío hay al menos un campo relacionado a este módulo. Ejemplo:

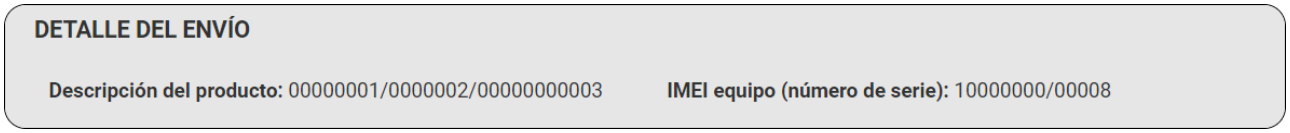


Figura 3: Módulo de detalle del envío

- **Módulo POD** (*Proof Of Delivery*, Prueba de distribución) (opcional): incluye información adicional sobre el contrato asociado al envío. Ejemplo:

Figura 4: Módulo POD

- **Módulo información adicional** (opcional): incluye información adicional proveniente desde el dispositivo móvil del distribuidor o de la sucursal. Se muestra un ejemplo en la figura 2.
- **Módulo observaciones** (opcional): incluye información sobre observaciones del envío. Ejemplo:

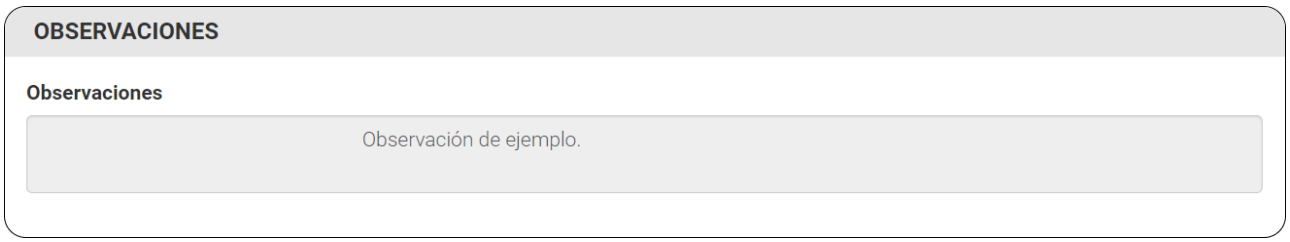


Figura 5: Módulo de observaciones

- **Módulo imágenes** (opcional): incluye imágenes adicionales que puede requerir el cliente, como fotos del documento del destinatario. Ejemplo:

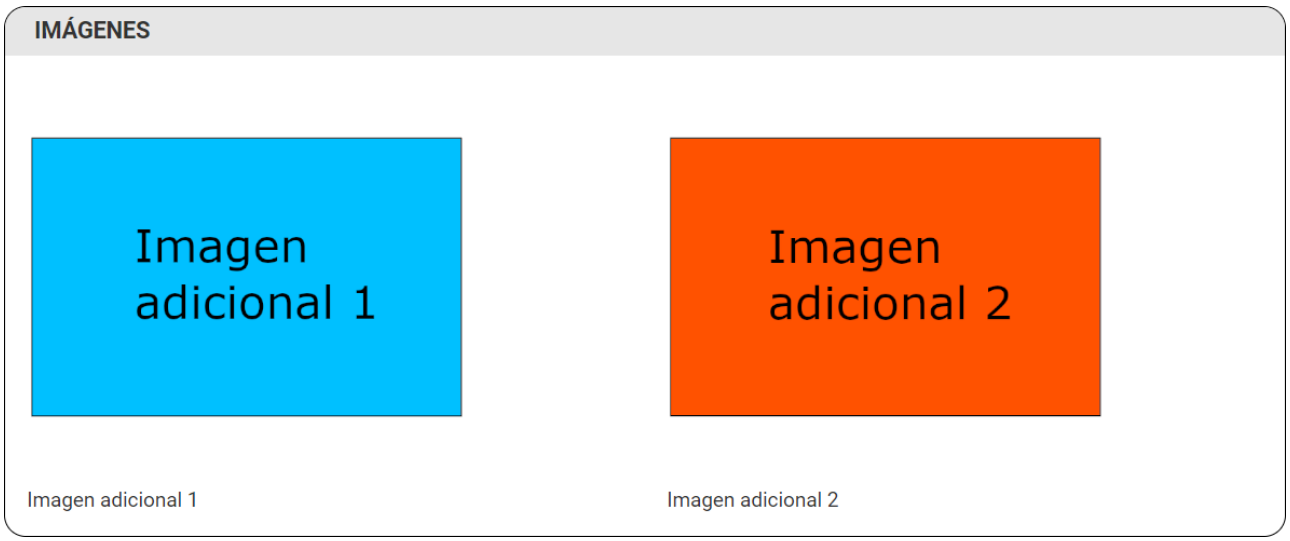


Figura 6: Módulo de imágenes

El sistema posee una web de configuración, que también tiene la opción para volver a generar una constancia, en caso de error. Los detalles sobre la misma se explican más adelante.

En la actualidad, aproximadamente se generan 130 mil constancias por día. Los siguientes números corresponden a 2 semanas (correspondientes a S_1 y S_2 en la tabla 1), empezando desde el 30 de septiembre de 2024 hasta el 13 octubre del mismo año:

	Lunes	**Martes**	**Miércoles**	**Jueves**	**Viernes**	**Sábado**	**Domingo**
S_1	114282	120336	123039	126015	134420	45609	706
S_2	124001	137479	141902	147419	28008	10661	894

Tabla 1: Número de constancias generadas día a día

ARQUITECTURA DEL SISTEMA ACTUAL

El sistema consta de un servicio monolítico (en otras palabras, es una única aplicación), desplegado en varias instancias que están distribuidas en varios servidores, que está continuamente escuchando eventos provenientes de la cadena de envíos. El proceso comienza con la recepción de un evento publicado en el sistema de transmisión de eventos

de la empresa, el cual desencadena la ejecución del procesamiento de los datos para la creación de la constancia.

Este es el listado principal de tecnologías que utiliza el sistema:

- **IBM MQ** para recepción y publicación de eventos
- **Windows Server** para el despliegue (en otras palabras: donde está en ejecución)
- **Angular** para el *frontend* (es un término que refiere a la parte visible de una aplicación web, con la que el usuario interactúa) [1]
- **Redis** para almacenar información de forma temporal
- **.NET Framework 4.6.1** como marco de desarrollo backend
- **SQL Server** como base de datos relacional (esto significa que sus entidades, en este caso las tablas, pueden estar relacionadas mediante ciertos parámetros)
- **NHibernate** como herramienta de **mapeo objeto-relacional** (expresión definida más adelante)
- **Log4Net** para la generación de registros
- **Azure Blob Storage** para almacenar las constancias generadas

Cada instancia del sistema se puede configurar como publicadora o suscriptora:

- En el caso de que sea publicadora la misma escucha eventos y los almacena en Redis.
- En caso de que sea suscriptora está a la escucha de los eventos publicados en Redis, para procesarlos.

El sistema posee una **API**, mediante la cual expone diversos **endpoints**. Definiciones de estos 2 términos:

- **API** es la sigla de *Application Programming Interface* (Interfaz de programación de aplicaciones), y se puede considerar como un contrato de servicio entre 2

aplicaciones, el cual define cómo deben comunicarse, sin necesidad de entender cómo funcionan internamente [8].

- Un **endpoint** (punto final) es la URL, o *Uniform Resource Locator* (Localizador de recursos uniforme, básicamente es la dirección web), mediante la cual se accede a un servicio de un sistema [9]. En este caso, la comunicación es a través del protocolo **HTTP** o *HyperText Transfer Protocol* (Protocolo de transferencia de hipertexto) el cual es utilizado en la navegación web [10]. Un endpoint devuelve respuestas con un código numérico, según lo ocurrido: 200 (OK, solicitud aceptada), 201 (Creado, si se guardó algo en base de datos), 404 (Objeto no encontrado), 400 (Solicitud incorrecta), entre otras.

La web de configuración del sistema se comunica al mismo mediante varios endpoints hechos a tal fin.

También existe un endpoint para descargar constancias generadas, al cual se le debe especificar un número de envío y el formato de archivo.

TECNOLOGÍAS UTILIZADAS POR EL SISTEMA ACTUAL

IBM MQ

MQ o *Message Queue* (Cola de mensajes) de IBM es un sistema de mensajería, orientado al uso empresarial, para el intercambio de mensajes entre aplicaciones y con ello la integración de activos de tecnología. El mismo envía y recibe datos entre aplicaciones y a través de redes.

La entrega de mensajes es segura y está desacoplada de la aplicación, porque el intercambio de mensajes se hace de manera transaccional, evitando que las aplicaciones verifiquen que los mensajes se entregaron de manera segura.

La implementación de MQ se basa en uno o más gestores de estructuras de datos, del tipo colas, cuyo envío de mensajes es asincrónico. En una estructura del tipo cola, el primer

elemento (en este caso un mensaje) en ser insertado es el primero que sale. Los gestores son el lugar donde se configuran los recursos de mensajería y a los que se conectan las aplicaciones, para consumir los mensajes publicados en las colas y tópicos configurados (los tópicos funcionan como colas, pero para un tipo de mensaje particular). El sistema se puede gestionar mediante una variedad de herramientas: desde la GUI o *Guided User Interface* (Interfaz guiada de usuario) de IBM MQ Explorer, a través de herramientas de línea de comandos interactivas y *scripts* (se puede traducir como secuencia de comandos).

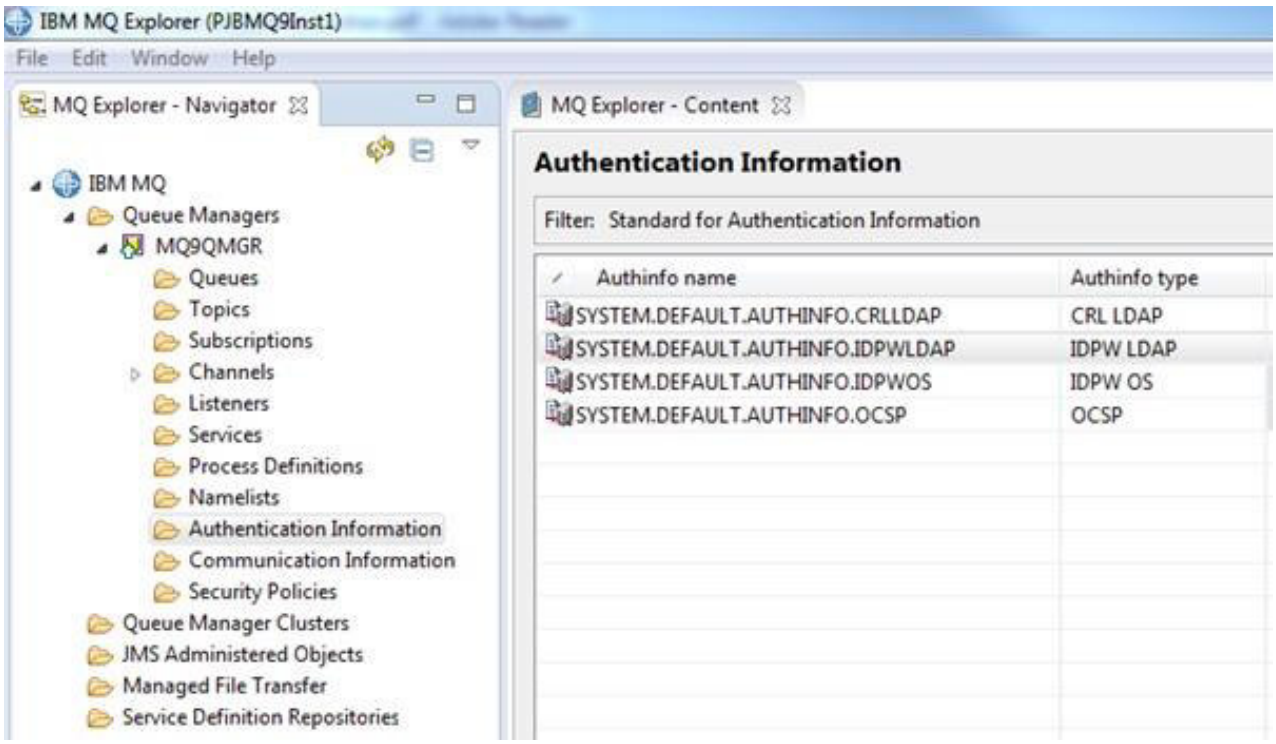


Figura 7: GUI de MQ[1]

El funcionamiento general es el siguiente:

1. En primer lugar, una aplicación de mensajería debe conectarse a un gestor de colas.
2. Segundo, cuando una aplicación desea transferir datos a otra aplicación, crea un mensaje y coloca los datos en el gestor.

[1] Imagen tomada de [52]

3. Finalmente, el sistema disponibiliza el mensaje en una cola o un tópico para que se entregue a los suscriptores.

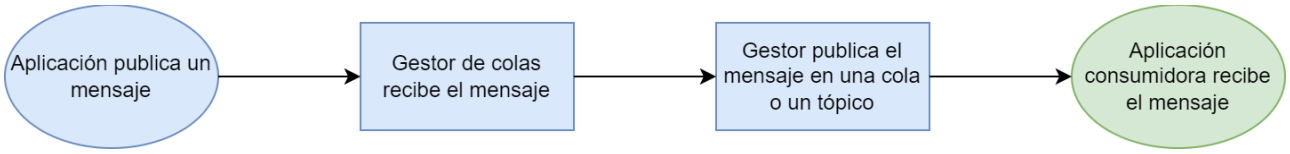


Figura 8: Flujo general de un mensaje en MQ

La cola o las suscripciones pueden estar en el mismo gestor de colas o en otros gestores conectados. Las aplicaciones no se comunican entre sí, lo hacen los gestores [11].

En el caso de este sistema, actualmente existen diversos tópicos en donde se publican los eventos de la cadena de envíos, que son consumidos por los publicadores de constancias. Estos, a su vez, publican los eventos en una base de datos en memoria (se explica más adelante), de la cual los suscriptores consumen los eventos para procesarlos.

Windows Server

Windows Server es una línea de sistemas operativos de Microsoft. En general, se instala en servidores de uso intensivo que sirven como columna vertebral para la mayoría de las empresas, aplicaciones y servicios de TI (Tecnología de la Información). El servidor maneja las actividades relacionadas con el grupo administrativo en una red. En este caso, se utiliza la versión 2022, en máquinas virtuales [12].

Actualmente el sistema está desplegado en 3 servidores. Cada uno contiene varias instancias del sistema de constancias en funcionamiento.

Angular

Angular es un marco de desarrollo para frontend, de código abierto, de **JavaScript**, escrito en **TypeScript** (ambos son lenguajes de programación) [13]. Google lo mantiene y su propósito principal es desarrollar aplicaciones de una sola página. Como marco de desarrollo, Angular tiene ventajas y, al mismo tiempo, proporciona una estructura estándar

con la que los desarrolladores pueden trabajar. Permite crear aplicaciones grandes de una manera fácil de mantener [14].

Este marco de desarrollo se utilizó para desarrollar la web de gestión del sistema.

Figura 9: Interfaz web de configuración del sistema

REDIS

Redis es una base de datos en memoria, del tipo no-relacional (significa que las entidades que guarda no necesitan tener referencias unas a otras). Proporciona soluciones locales y en la nube, y se adapta a cualquier conjunto de tecnologías, lo que facilita la creación, el escalado y la implementación de aplicaciones rápidas. También maneja diversos tipos de datos, posee características de replicación y persistencia en disco. [15]

Como se mencionó anteriormente, en la descripción de MQ, el uso de Redis actúa como capa intermedia entre este y el sistema. Esto tiene varias ventajas:

- Desacople de funcionalidad: publicadores y suscriptores trabajan de manera independiente.
- Reducción de carga en MQ: al tener menos instancias comunicándose directamente a MQ, se logra una reducción de la carga de trabajo sobre este.
- Velocidad: Redis es muy rápida, tanto para lectura como para escritura, lo cual da una ventaja de rendimiento al sistema.
- Escalamiento: en caso de necesidad, se puede aumentar la cantidad de publicadores manteniendo igual la cantidad de suscriptores, y viceversa.

.NET Framework

.NET Framework es un marco de desarrollo de software para crear y ejecutar aplicaciones en Windows.

Para explicar qué es .NET Framework se debe empezar por .NET, que es una plataforma de desarrollo compuesta por herramientas, lenguajes de programación y bibliotecas, para crear diversos tipos de aplicaciones. Existen varias implementaciones de .NET. Hoy en día, cada implementación permite que el código .NET se ejecute en diferentes sistemas operativos (no aplica para .NET Framework): Linux, macOS, Windows, iOS, Android y otros. Admite la ejecución de sitios web, servicios, aplicaciones de escritorio y más en Windows. Es multiplataforma y sirve para crear aplicaciones como sitios web, servicios y aplicaciones de consola, en Windows, Linux y macOS.

Las aplicaciones .NET (incluidas las de .NET Framework) están escritas los lenguajes de programación C#, F# o Visual Basic. El código se compila en un **lenguaje intermedio común** (CIL, *Common Intermediate Language*) independiente de estos. El código compilado se almacena en ensamblados: archivos con una extensión de archivo .dll o .exe. Cuando se ejecuta una aplicación, un motor de ejecución toma el ensamblado y utiliza un compilador **justo a tiempo** (JIT, *Just In Time*) para convertirlo en código de máquina, que se puede ejecutar en la arquitectura específica de la computadora en la que se ejecuta.

.NET Framework es la implementación original de .NET. Sus 2 componentes principales son **Common Language Runtime** y **.NET Framework Class Library**:

- Entorno de Ejecución Común (CLR, *Common Language Runtime*): es el motor, mencionado anteriormente, que maneja las aplicaciones en ejecución. Proporciona servicios como administración de subprocesos, recolección de elementos no utilizados, seguridad de tipos (de datos), manejo de excepciones y más.

- Biblioteca de Clases (*Class Library*): proporciona un conjunto de interfaces y tipos para funcionalidades comunes. Proporciona tipos para cadenas, fechas, números, etc. La biblioteca de clases incluye interfaces para leer y escribir archivos, conectarse a bases de datos, dibujar y más [16].

En este caso, el sistema fue desarrollado en C#, que es el más común de los 3 lenguajes mencionados.

SQL Server

Microsoft SQL Server es un sistema de gestión de bases de datos relacionales que admite una amplia variedad de aplicaciones de procesamiento de transacciones, inteligencia empresarial (BI, *Business Intelligence*) y análisis de datos en entornos de TI corporativos.

SQL Server se basa en el lenguaje de consulta estructurado o SQL (*Structed Query Language*), un lenguaje de programación estandarizado que los administradores de bases de datos o DBA (*Database Administrators*) y otros profesionales de TI utilizan para administrar bases de datos y consultar los datos que contienen. SQL Server está vinculado a Transact-SQL (T-SQL), el lenguaje de consulta patentado de Microsoft que permite que las aplicaciones y herramientas se comuniquen y también se conecten a una instancia o base de datos de SQL Server [17].

El sistema de constancias posee su base de datos SQL Server, en la cual existen las tablas que guardan la configuración de este. Al ser relacional, se observa como ciertas tablas hacen referencia a otras, a la hora de guardar datos (por ejemplo, en vez de guardar el nombre de un elemento, se guarda su identificador, que tiene su correspondencia en otra tabla). También hay una tabla utilizada a modo de registro, en donde el sistema guarda un registro de las constancias procesadas. Posee más tablas, que actualmente están en desuso.

NHibernate

Es una herramienta de **mapeo objeto-relacional** para entornos .NET. El término mapeo objeto-relacional u ORM (*Object Relational Mapping*) se refiere a la técnica de mapear una representación de datos de un modelo de objetos (lo que se define como clase) a un modelo de datos relacional con un esquema basado en SQL. Entre sus características más importantes se incluyen:

- Mapea clases .NET a tablas de bases de datos
- Proporciona facilidades de consultas (tanto de lectura como de escritura) de datos
- Reduce significativamente el tiempo de desarrollo que de otro modo se dedicaría al manejo manual de datos (característica típica de un ORM) [18]

En el caso de este proyecto, se utiliza para trabajar con la base de datos SQL Server. En el código legado se puede observar cómo esta herramienta permite tratar a las tablas de la base de datos como objetos, en un contexto de Programación Orientada a Objetos [19].

Log4Net

Log4net es una biblioteca de *logging* (se puede traducir como registro de eventos) que permite controlar qué declaraciones de registro se generan en una aplicación. Es totalmente configurable, en tiempo de ejecución, mediante archivos de configuración externos. Características principales:

- Proporciona un registro sobre la ejecución de la aplicación. Mientras una aplicación está en ejecución, la generación de la salida del registro no requiere intervención humana.
- La salida del registro se puede guardar en un medio persistente (como archivos de texto), y por esto mismo se lo puede considerar como una herramienta de auditoría.

Tiene algunas desventajas, como ralentizar una aplicación. Si es demasiado detallado, puede ser difícil leer los detalles [20].

Se observa como esta librería es utilizada en el proyecto para hacer capturas de mensajes de todo tipo, desde información general hasta errores y advertencias, lo cual se guarda en archivos del tipo .log, los cuales se pueden leer con cualquier aplicación visualizadora de texto.

```
2024-10-25 14:36:24,353 | INFO  | Se procesa la visita del envio: 0000001.
2024-10-25 14:36:24,368 | INFO  | Envio 0000001 valido y con contrato 002 configurado -- ENVIO ACEPTADO
2024-10-25 14:36:24,368 | INFO  | Datos desde traza cargados correctamente, envio 0000001.
2024-10-25 14:36:24,368 | INFO  | Opcionales de constancia configurados correctamente, envio 0000001.
2024-10-25 14:36:24,699 | INFO  | Imagen de geolocalizacion obtenida correctamente para el envio 0000001.
2024-10-25 14:36:24,701 | INFO  | Datos de geolocalización cargados correctamente para el envio 0000001.
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional ESCANEO DEL ENVIO - 0000001
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional Tenes el DNI / Mi Argentina? - NO
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional NOMBRE - Fulano
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional APELLIDO - Qwerty
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional DNI
2024-10-25 14:36:24,886 | INFO  | Se agrega el adicional ¿Puede registrar su firma? - SI
2024-10-25 14:36:24,896 | INFO  | Datos desde mobile cargados correctamente, envio 0000001.
2024-10-25 14:36:24,896 | INFO  | Inicia carga de Datos de API Envio, envio: 0000001.
```

Figura 10: Ejemplo de archivo generado por Log4Net

Azure Blob Storage

Azure Blob Storage es la solución de almacenamiento de objetos de Microsoft para la nube. El término *blob* es un acrónimo que proviene de *Binary Large Object* (Objeto binario grande). Blob Storage está optimizado para almacenar cantidades masivas de datos no estructurados. Los datos no estructurados son datos que no se adhieren a un modelo de datos o definición en particular, como texto o datos binarios.

Blob Storage permite:

- Servir imágenes o documentos directamente a un navegador
- Almacenar archivos para acceso distribuido
- Transmisión de video y audio
- Escribir en archivos de registro
- Almacenar datos para copia de seguridad y restauración
- Almacenar datos para análisis por parte de un servicio local o alojado en Azure

Los usuarios o las aplicaciones cliente pueden acceder a los objetos en Blob Storage a través de diversas opciones: Azure Storage Explorer, API de Azure Storage, Azure PowerShell, Azure CLI (*Command Line Interface,* Interfaz por línea de comandos) o una biblioteca de cliente de Azure Storage [21].

En el caso de este sistema, esta herramienta se utiliza para guardar y también para leer datos. Constancias Electrónicas guarda los documentos generados en un contenedor. Pero también necesita leer otros contenedores para acceder a datos necesarios para la generación de estos, como información en texto e imágenes.

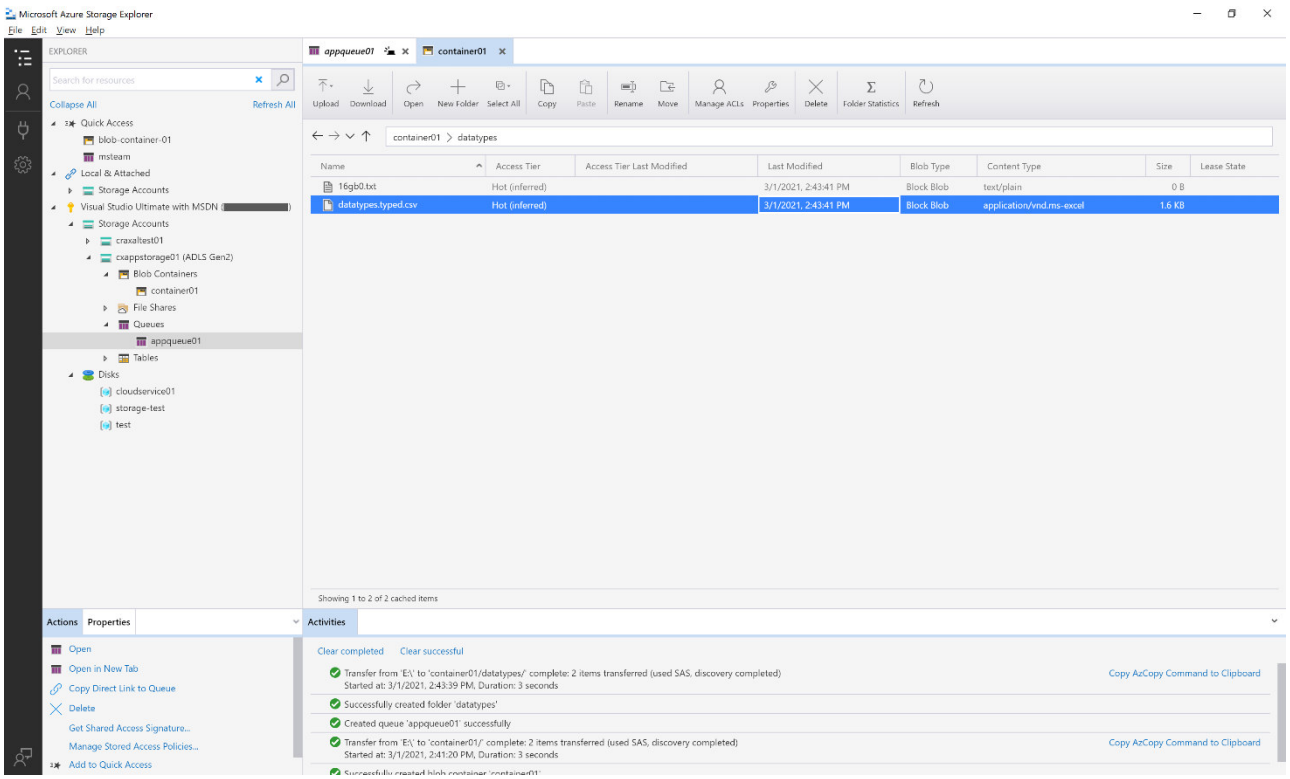


Figura 11: Azure Storage Explorer[2]

[2] Imagen tomada de [53]

Firma Estudiante:	Firma Docente Supervisor:	Firma Tutor Organizacional:
EMIL HROMEK		

FLUJO GENERAL DE LA GENERACIÓN DE CONSTANCIAS

El proceso inicia con la generación de eventos pertenecientes a la cadena de envíos. Estos eventos son generados cuando ocurre un evento en el movimiento de un paquete (ejemplos de estos son altas, llegadas a sucursal, visitas, entre otros), y son reportados a los sistemas de procesamiento de estos. Hay 2 sistemas de los cuales Constancias Electrónicas consume eventos:

- **Integra**: procesa los envíos B2C (menores a 50 kg). B2C es la sigla de *Business to Consumer* (Empresa a consumidor).
- **Alertran**: procesa los envíos B2B (a partir de 50 kg). B2B es la sigla de *Business to Business* (Empresa a empresa) [22].

Una vez que se genera un evento por estos sistemas es publicado en una cola de MQ, con el objetivo de que pueda ser consumido por otros sistemas que tiene la empresa. Constancias Electrónicas tiene suscripción a MQ y, como se mencionó, es uno de los consumidores. Tal como se explicó en la introducción, los eventos disparadores son lo que están relacionados al estado final de un envío, ya sea entregado o no.

Como se mencionó anteriormente, el sistema recibe los mensajes y los almacena temporalmente en Redis, en vez de consumirlos directamente de MQ, para que los suscriptores después los consuman.

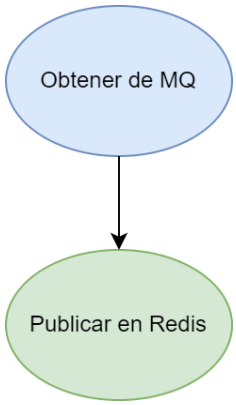


Figura 12: Obtención de mensajes de la cadena de envío

El siguiente paso es el enriquecimiento de información sobre el mismo. Para esto se clasifica el evento según el sistema remitente del mismo (Integra o Alertran). La necesidad de esto radica en que cada uno de estos tiene su API:

- **API Integra**: provee información sobre un envío procesado por Integra
- **API Alertran**: ídem para Alertran

Estas APIs proveen información adicional que no está incluida en el evento de MQ.

El sistema le hace consultas a estas APIs para obtener más información. Si el envío viene de Integra, se consultan 2 APIs adicionales:

- **API de Información Adicional de Contrato**
- **API de Información Adicional de Envío**

Si viene de Alertran, se hace una consulta adicional a una API llamada **Clientes de Alertran**.

En ambos casos, estos procesos son de enriquecimiento de información sobre el envío.

En la siguiente página se muestra un diagrama de flujo de todo este procedimiento.

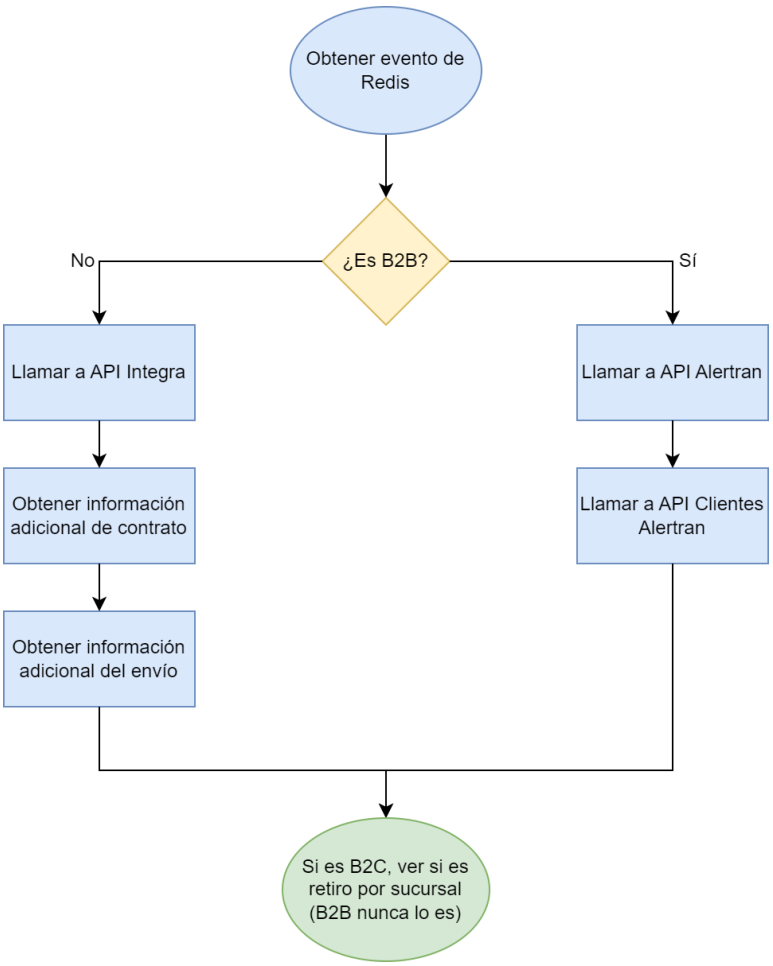


Figura 13: Proceso de enriquecimiento de un evento

Durante el recorrido anterior, si la constancia a generar requiere módulos adicionales (exceptuando geolocalización) y la información para completarlos proviene de las fuentes mencionadas anteriormente, entonces se guarda dicha información.

Una vez superado este proceso, se verifica si el paquete fue entregado a domicilio o en sucursal, o si es un evento de rectificación de motivo (de ocurrencia de un evento). En el primer caso se consulta a la **API Mobile**, que es una API que provee información proveniente de los dispositivos móviles utilizados por los transportistas a lo largo de su recorrido. En el segundo se obtiene la información desde un contenedor de imágenes de Azure Blob Storage. En el último caso se consulta a ambas fuentes.

Figura 14: Una sucursal de Andreani[3]

Si la constancia requiere el módulo de geolocalización se hace adicionalmente la llamada a la API de Google Maps, para generar el mapa. Si alguno o algunos de los otros módulos adicionales requieren información de Mobile y/o del contenedor de imágenes, se guarda dicha información.

Finalmente, con toda la información recopilada, se genera la constancia y se la disponibiliza en un contenedor de Azure Storage, también en un servidor FTP en caso de que así sea requerido por el cliente. También se genera un registro en la base de datos del sistema y se publica en MQ un evento llamado **Constancia Electrónica Generada**, el cual contiene información sobre la constancia generada, incluida la URL de esta (para descargarla desde el contenedor), así como también las URLs de documentos asociados (como las imágenes de las firmas), para ser consumido por otros sistemas de la empresa [6].

[3] Imagen tomada de [51]

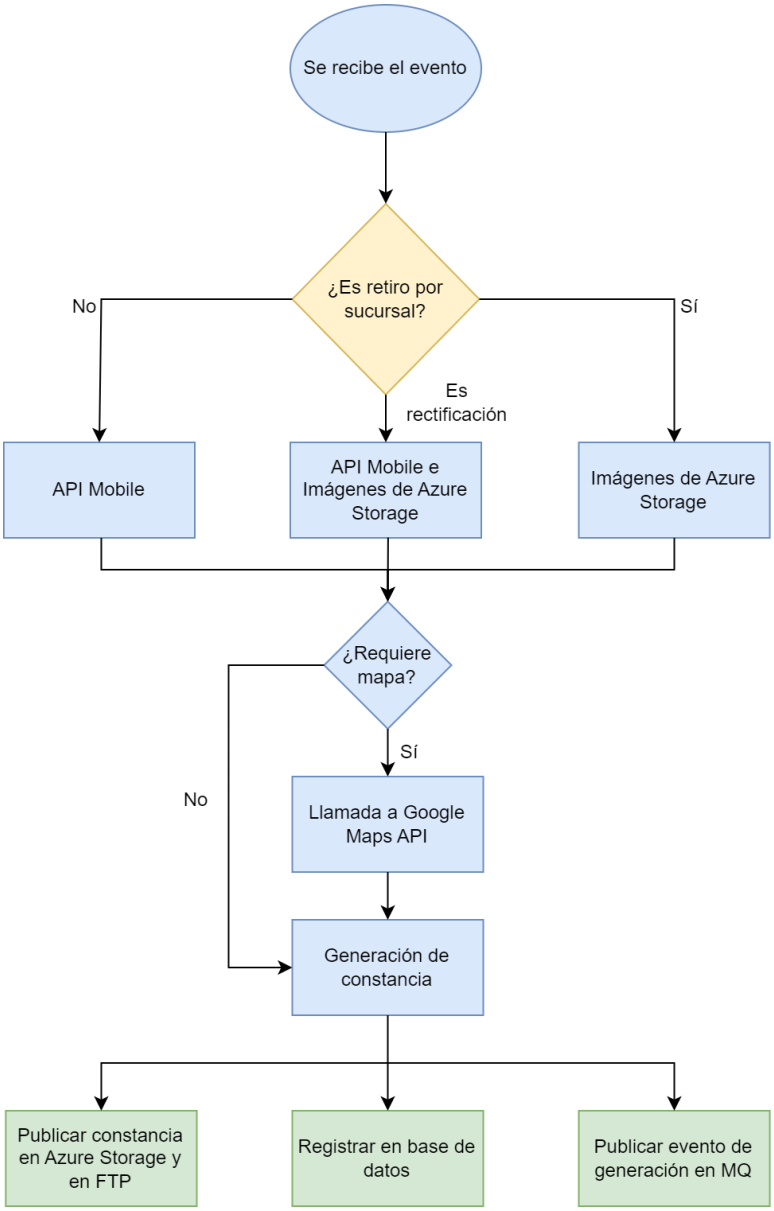


Figura 15: Proceso de generación de la constancia

CASOS PUNTUALES DE USO

Como se mencionó anteriormente, el sistema tiene una web de configuración. En la misma se puede:

- Crear un nuevo tipo de constancia y ver los ya existentes.

- Agregar contratos al sistema, así como también ver los que ya existen y modificar la información asociada. Esta función se nutre de una API que provee información sobre contratos.

- Republicar constancias, pasando un número de envío o un listado de estos. En este último caso, el sistema almacena el listado, para después en un horario de la madrugada ejecutar las republicaciones.

Endpoint de descarga de constancias: como se mencionó anteriormente, existe un endpoint al cual se le pasa un número de envío y el formato de archivo, para descargar una constancia.

IMPLEMENTACIÓN GENERAL DEL CÓDIGO

No es objetivo de este informe describir de manera exhaustiva la estructura del código legado. Sí se pretende que haya un entendimiento, y de hecho es necesario que lo haya, para entender sobre qué se basa el proceso migratorio. Sin embargo, no se considera necesario que se sepa de manera minuciosa cómo está escrito el código legado, porque el valor de este trabajo está en la nueva implementación que se hizo. Lo que se pretende es que se entienda bien el flujo del sistema, y por eso se lo explicó anteriormente.

Se observan las siguientes características en el código legado:

- Como se dijo anteriormente, el código está en desarrollado en .NET Framework y C#, con todo lo que esto típicamente implica.
- Clases del tipo gestor: son las encargadas de ejecutar la lógica de negocio del sistema, así como también la relacionada a conexiones externas.
- Clases del tipo modelo: representan las diversas entidades que componen el sistema, incluidas las tablas de la base de datos, y los métodos para operar con las primeras.
- Clases del tipo repositorio: proveen los métodos para operar con la base de datos.

- Clases del tipo servicios: especifican la lógica de negocio de los endpoints. Los mismos están implementados utilizando el estándar REST o *Representational State Transfer* (Transferencia de estado representacional), que especifica una estandarización en la transferencia de información, comunicación entre cliente y servidor sin estado, entre otras cuestiones que limitan la reglas y especificaciones para diseñar endpoints [23].
- Clases del tipo utilidades: proveen métodos de uso general.
- Archivos de configuración: contienen los parámetros de configuración del sistema, no son parte del código fuente.
- Existencia de código deprecado: hasta este punto se hizo una descripción general del sistema, considerando lo que está en uso actualmente. La realidad es que el sistema tiene código en desuso, por ejemplo, para generar otros tipos de constancias que actualmente no se generan más, entre otras características.

MÉTRICAS DE RENDIMIENTO

Actualmente se puede tener una idea del rendimiento basándose en varios elementos:

- Los logs del sistema: con estos archivos (generados mediante Log4Net) se guarda un registro en tiempo real de los eventos que van ocurriendo en el sistema. Estos contienen mensajes de información, incluyendo errores y su horario exacto.
- Visualización en MQ Explorer: se puede observar en tiempo real la cantidad de mensajes que están encolados, con esto se puede tener una idea acerca de la velocidad de procesamiento de estos.
- Herramientas del equipo de Infraestructura: este equipo posee herramientas más especializadas para el monitoreo de MQ, como por ejemplo los gráficos diarios de la cantidad de mensajes.

- Revisión del estado de los servidores: con esto se puede observar la carga de memoria y del procesador.

Se realizó un análisis de los logs actuales, mediante un script hecho a tal fin. El algoritmo realizó los siguientes pasos:

1. Para cada evento que generó una constancia, calcular la diferencia entre el momento de generación del evento en la vida real y el momento en el que se generó el evento **Constancia Electrónica Generada** correspondiente.
2. Agrupar estas diferencias en bandas horarias, de 6 horas. El horario que se le asigna a la diferencia calculada es el de generación del evento en la vida real.
3. Calcular el promedio para cada banda horaria.

Estos fueron los resultados:

Fecha	**Rango horario**	**Promedio de diferencia (minutos)**
7/10/2024	12:00 a 17:59:59	14343,06
8/10/2024	12:00 a 17:59:59	21667,6
9/10/2024	12:00 a 17:59:59	11832,35
9/10/2024	06:00 a 11:59:59	13340,5
10/10/2024	06:00 a 11:59:59	19265,76
10/10/2024	12:00 a 17:59:59	13836,29
14/10/2024	06:00 a 11:59:59	4055,77
14/10/2024	12:00 a 17:59:59	7638,24
14/10/2024	18:00 a 23:59:59	10845,43
15/10/2024	06:00 a 11:59:59	3306,17
15/10/2024	12:00 a 17:59:59	56,43
15/10/2024	00:00 a 05:59:59	772,19
15/10/2024	18:00 a 23:59:59	1,7
16/10/2024	06:00 a 11:59:59	260,3
16/10/2024	12:00 a 17:59:59	318,49

Fecha	Rango horario	Promedio de diferencia (minutos)
16/10/2024	18:00 a 23:59:59	204,16
16/10/2024	00:00 a 05:59:59	5,22
17/10/2024	06:00 a 11:59:59	54,83
17/10/2024	12:00 a 17:59:59	54,75
17/10/2024	18:00 a 23:59:59	3,74
17/10/2024	00:00 a 05:59:59	5,19
18/10/2024	12:00 a 17:59:59	52,56
18/10/2024	06:00 a 11:59:59	20,72
18/10/2024	18:00 a 23:59:59	7,32
18/10/2024	00:00 a 05:59:59	4,13
19/10/2024	06:00 a 11:59:59	2,79
19/10/2024	12:00 a 17:59:59	1,14
19/10/2024	18:00 a 23:59:59	1,02
19/10/2024	00:00 a 05:59:59	3,19
20/10/2024	00:00 a 05:59:59	15,93
20/10/2024	06:00 a 11:59:59	0,36
20/10/2024	12:00 a 17:59:59	0,37
20/10/2024	18:00 a 23:59:59	0,37

Tabla 2: Tiempos de generación de una constancia en el sistema actual

Estos números representan cotas mínimas de tiempos promedio. En una fecha posterior a las que aparecen en la tabla 2, podría aparecer algún evento que se publique con retraso significativo en MQ. Un caso así aumentaría el promedio de tiempo calculado para el rango horario correspondiente. Ejemplo:

- Aparece un evento A, generado el 13/10, a las 14:00. La constancia correspondiente al mismo se generó a las 15:00. Por lo tanto, la demora es el resultado de hacer la diferencia entre estos 2 valores: fue de 1 hora.

- Aparece otro evento B, generado el 13/10, a las 14:10. Su constancia se generó a las 14:30. Por lo tanto, la demora fue de 20 minutos.
- El promedio de generación para la banda horaria del día 13/10 entre las 12:00 y 17:59:59 es $\frac{1\text{ hora} + 20\text{ minutos}}{2}$ = 40 minutos.
- Aparece un evento C: generado en la vida real el 13/10, 14:30, pero llegó al sistema el día 14/10. La constancia se generó a las 16:00 del mismo día. Entonces la demora fue de 1 día y 30 minutos.
- Por lo tanto, el nuevo promedio para la banda horaria es: $\frac{1\text{ hora} + 20\text{ min} + 1\text{ día y }30\text{ min}}{3}$ = 516,66 minutos.

Por otra parte, las demoras considerables que se observan en las primeras bandas horarias de la tabla tienen una interpretación (y está relacionado con lo explicado anteriormente): el análisis se realizó sobre una cantidad acotada (según una fecha mínima) de logs y ocurrió que llegaron algunos eventos atrasados. El promedio para las bandas horarias de esos eventos justamente se está calculando exclusivamente sobre los mismos, lo cual está sesgando el resultado para dichos horarios. La manera de evitar esto sería incluir archivos anteriores en el proceso, los cuales no siempre se guardan, porque se eliminan automáticamente.

MOTIVOS DE LA MIGRACIÓN

El tener motivos sólidos para migrar el sistema fue el puntapié para la realización de este trabajo, ya que se requería una justificación importante por la cantidad de horas de trabajo y recursos económicos invertidos. A continuación, se detalla el porqué de la decisión.

CONTEXTO ACTUAL Y NECESIDADES DEL NEGOCIO

La empresa, si bien pertenece al sector logístico, desde hace años tiene su sector de tecnología, el cual está en constante innovación. Esto involucra la búsqueda de nuevas tecnologías que se alineen con las necesidades del negocio, lo cual se traduce en un proceso transversal de modernización de los sistemas que tiene el mismo. El uso de tecnologías obsoletas tiene consecuencias como limitaciones en el rendimiento, dificultades para hacerles soporte técnico, aparición de problemas de seguridad, limitaciones en la integración con otras tecnologías que se quieran implementar y pérdida de competitividad del negocio, entre otras.

Existe también la posibilidad de que una tecnología no sea obsoleta, sin embargo, se puede requerir reemplazarla por otra que tenga un menor costo de mantenimiento y/o licencia, como es el caso actual de IBM MQ, cuyo costo es algo más de 200 mil dólares por año.

Por lo tanto, desde la Dirección de Tecnología se impulsa el desarrollo constante, bajo la guía del equipo de Arquitectura, que define tecnologías y estándares a aplicar en los desarrollos para el negocio.

LIMITACIONES IDENTIFICADAS

Luego de un análisis para determinar qué limitaciones y problemas en general tiene el sistema original, se arribó a la siguiente lista:

- Escalabilidad de los componentes: el sistema, debido a que es monolítico, sería mucho más fácilmente escalable si tuviese sus funcionalidades distribuidas en microservicios. En otras palabras, sería mejor que sus funcionalidades se repartan en varias aplicaciones.
- Desacople de funciones y complejidad del monolito: un sistema monolítico puede llegar a tener una complejidad considerable, la cual se puede reducir mediante una implementación de microservicios. Con esto se logra un sistema con funciones desacopladas, que además permite más flexibilidad en el desarrollo.
- Rendimiento: al ser un sistema antiguo (en lo que respecta a *stack* tecnológico, es decir, las tecnologías que utiliza), se espera que tenga menor rendimiento en comparación con uno más moderno, que cumpla la misma función. Habitualmente las tecnologías más modernas suelen tener mejor rendimiento que las más antiguas. Por otra parte, un monolito requiere un cierto número de instancias corriendo al mismo tiempo, y eso aplica para el sistema entero. Con una arquitectura de microservicios, el número de instancias puede variar según el microservicio, lo cual implica que se puede variar el número de instancias para cierta funcionalidad del sistema. Si un microservicio necesita baja demanda, entonces puede tener una instancia. Otros servicios que requieran más demanda pueden tener más instancias.
- Aislamiento de fallos: si el monolito falla se cae todo el sistema. Esto puede mitigarse con una arquitectura de microservicios.
- Stack tecnológico: como se mencionó en la parte de rendimiento, está desactualizado y hay tecnologías que se pretenden reemplazar por un tema de costos o conveniencia en general.
- Limitaciones de seguridad: actualmente el sistema no cumple los estándares de seguridad informática requeridos por el negocio [6].

BENEFICIOS ESPERADOS Y OPORTUNIDADES TECNOLÓGICAS

De esta migración se espera obtener beneficios, tales como:

- Mejoras en el rendimiento, debido a que se implementaron cambios tales como tecnologías más nuevas, de las cuales se espera que impacten positivamente en el rendimiento del sistema.
- Mayor escalabilidad funcional, debido a la posibilidad de agregar microservicios que desempeñen otras funciones actualmente no contempladas.
- Reducción de costos, la cual se logrará principalmente por el apagado de MQ, así como también por eliminar tecnologías obsoletas, cuyo mantenimiento puede ser costoso a largo plazo.
- Mejoras en seguridad, al incorporar los últimos estándares de seguridad definidos por el negocio.
- Mejor apoyo a la toma de decisiones sobre el sistema. Se incorporó una tecnología que permite visualizar métricas del sistema en un *dashboard* (es un tablero del tipo panel de visualización).
- Flexibilidad tecnológica. Si bien el desarrollo del backend se hizo enteramente en .NET, nada impide hacer un nuevo microservicio en otro lenguaje y/o marco de desarrollo.
- Competitividad en el mercado, gracias a contar con un sistema moderno, que mejora la eficiencia operativa y responde más rápidamente.

CONSECUENCIAS DE NO MIGRAR

Evitar la migración del sistema conlleva una consecuencia importante: a medida que pasa el tiempo, el sistema se hace cada vez más obsoleto. Esto implica la pérdida de soporte oficial de las tecnologías que posee, así como también la oportunidad de tener un sistema que tenga un mejor rendimiento, frente a uno más moderno. Si se mantiene el sistema

original, la empresa tiene que hacer el esfuerzo por seguir dándole soporte, y se puede complicar cada vez más. Esto implica un costo adicional. También hay otras desventajas:

- Riesgos de seguridad: un sistema más nuevo puede ser más seguro (como se mencionó anteriormente, actualmente el sistema no cumple con los estándares mínimos que solicita la empresa).
- La dependencia de personal cada vez más especializado en las tecnologías que posee.
- La dificultad para integrarse con otros sistemas, debido a su obsolescencia.
- Además, visto desde el punto de vista del negocio, el mismo pierde competitividad frente a otros que sí tengan sistemas modernos.

RECOMENDACIONES Y CONSIDERACIONES INICIALES

Algunas recomendaciones y consideraciones iniciales que se tuvieron en cuenta previo al desarrollo del sistema fueron:

- Entender los ambientes de desarrollo (o de despliegue) de la empresa, que son: **desarrollo**, **test** (prueba), **QA** (*Quality Assurance*, Aseguramiento de la calidad) y **producción**, y saber cómo trabajar en cada uno de ellos.
- Definición de los requerimientos funcionales y no funcionales del sistema.
- Estandarización de las herramientas y tecnologías que se utilizarán. Si bien hay puntos que están definidos por el equipo de Arquitectura y se debían respetar, otros quedaron a elección personal.
- Definición inicial de la arquitectura del sistema, para saber qué servicios desarrollar.
- Contar con todo el personal necesario, no solo técnico sino también que tenga conocimiento del negocio.
- Aplicar buenas prácticas de desarrollo, para maximizar la calidad de este.

- Diseño del plan de migración, con el listado correspondiente de tareas.

PROCESO DE DESARROLLO DEL NUEVO SISTEMA

Luego de realizar un análisis del sistema original, que incluyó el relevamiento de sus limitaciones y las soluciones (o mitigaciones) a las mismas, se procedió a desarrollar el código del nuevo sistema. En esta sección se detalla el paso a paso sobre este proceso, con el objetivo de que se lo pueda entender desde su etapa inicial hasta la finalización.

METODOLOGÍA DE TRABAJO

SCRUM

Para la migración, y como metodología estándar de trabajo, no sólo en la empresa sino también comúnmente utilizada en el ámbito de desarrollo de software, se utilizó Scrum.

Scrum es una manera hacer el trabajo en equipo en pequeñas partes a la vez, con experimentación constante y ciclos de retroalimentación a lo largo del camino para aprender y mejorar a medida que se avanza. Esto crea valor de manera incremental y colaborativa. Como metodología ágil, Scrum proporciona la estructura para que el personal y los equipos se integren en su forma de trabajar, al mismo tiempo que implementa las prácticas adecuadas para cumplir de manera óptima sus necesidades.

El nombre de la metodología está inspirado en un scrum del rugby. En el rugby, el equipo se reúne en lo que llaman un scrum, para trabajar juntos y hacer avanzar la pelota. En este contexto, Scrum es donde el equipo se reúne para hacer avanzar el producto.

Scrum es un proceso empírico. Es importante que en un equipo Scrum haya confianza. Si no la hay, es probable que haya tensión y obstáculos que impidan realizar el trabajo, algo totalmente esperable.

Características importantes:

- Se entregan incrementos de trabajo en ciclos cortos (de un mes o menos), que se denominan *sprints* (como en una carrera). Durante el sprint se produce una

retroalimentación continua, lo que permite inspeccionar y adaptar el proceso a las circunstancias.

- El equipo Scrum está formado por:
  - Un *Scrum Master* (Maestro de Scrum), que es quien administra todo el proceso de Scrum.
  - Un *Product Owner* (Dueño de producto), que es el responsable del producto a desarrollar y habitualmente es alguien cercano al negocio.
  - Los desarrolladores, que son responsables de convertir las tareas elegidas para un sprint en un incremento de valor.
- El equipo Scrum y otros miembros de la organización, negocio, usuarios o base de clientes, conocidos como partes interesadas, inspeccionan los resultados de un sprint y realizan los ajustes necesarios para el próximo [24].

Scrum tiene ceremonias, cada una con un propósito, limitaciones de tiempo y participantes:

- Planificación del sprint (*Sprint Planning*): el sprint comienza con una reunión de planificación, en la que las partes planifican el trabajo a realizar en el mismo. Este plan crea una comprensión y una alineación de objetivos compartidas en el equipo.
- Reunión diaria (*Daily Scrum*): reunión que se hace todos los días, en la cual típicamente se comenta qué se hizo ayer y en qué se va a trabajar hoy.
- Revisión del sprint (*Sprint Review*): Al final del sprint, el equipo se reúne con las partes interesadas para mostrar lo logrado y obtener retroalimentación.
- Retrospectiva del sprint (*Sprint Retro*): por último, el equipo Scrum se reúne al final del sprint, para analizar cómo fue este y si hay cosas que se podrían hacer de manera diferente, así se puede mejorar en el próximo.

La duración del sprint no debe ser mayor a un mes, aunque normalmente dura 2 semanas. La reunión diaria dura habitualmente 15 minutos (no se pretende que dure mucho y se debe respetar eso).

Scrum también tiene otros elementos, que son:

- Trabajo pendiente del producto (*Product Backlog*): es el listado total de tareas del proyecto.
- Trabajo pendiente del sprint (*Sprint Backlog*): es el listado de tareas a realizar en un cierto sprint. Se definen en la reunión de planning.
- Gráfico de trabajo pendiente (*Burndown Chart*): una manera gráfica de visualizar el progreso de trabajo, en función del tiempo [25].

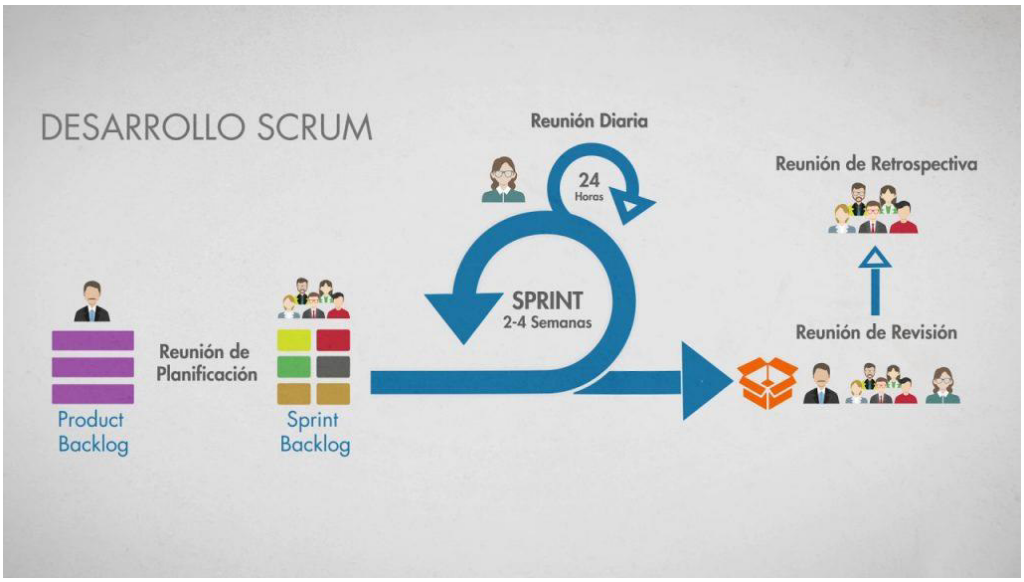


Figura 16: Proceso de Scrum[4]

Todo lo anterior es la descripción teórica de la metodología Scrum. En el caso de este trabajo, los sprints tenían una duración de 2 semanas y se respetaban todas las ceremonias. Las reuniones diarias duraban entre 15 y 30 minutos, salvo casos excepcionales.

[4] Imagen tomada de [54]

JIRA

Jira es una herramienta de gestión de proyectos basados en metodologías ágiles (como lo es Scrum), utilizada para planificar, realizar un seguimiento, publicar y dar soporte a software. Es una fuente de información para todo el ciclo de vida del desarrollo, lo que permite que un equipo cuente con el contexto necesario para avanzar rápidamente y, al mismo tiempo, mantenerlo conectado con el objetivo. Ya sea que se utilice para gestionar proyectos sencillos o lo que se requiera, facilita que los equipos avancen en el trabajo y se mantengan alineados [26].

En este caso, se le solicitó al equipo de Gestión de Proyectos la creación de un proyecto en Jira para esta migración, en el cual se cargaron inicialmente todas las tareas a realizar (product backlog). A medida que se creaban los sprints, se elegían las tareas a realizar (sprint backlog) y se les asignaba el usuario correspondiente. Jira tiene la función de asignarle un estado a cada tarea, por ejemplo: sin empezar, en progreso, en revisión y finalizado.

A medida que progresaba el proyecto, hubo la necesidad de cargas tareas nuevas y se lo hizo, ya que no es un limitante para Jira.

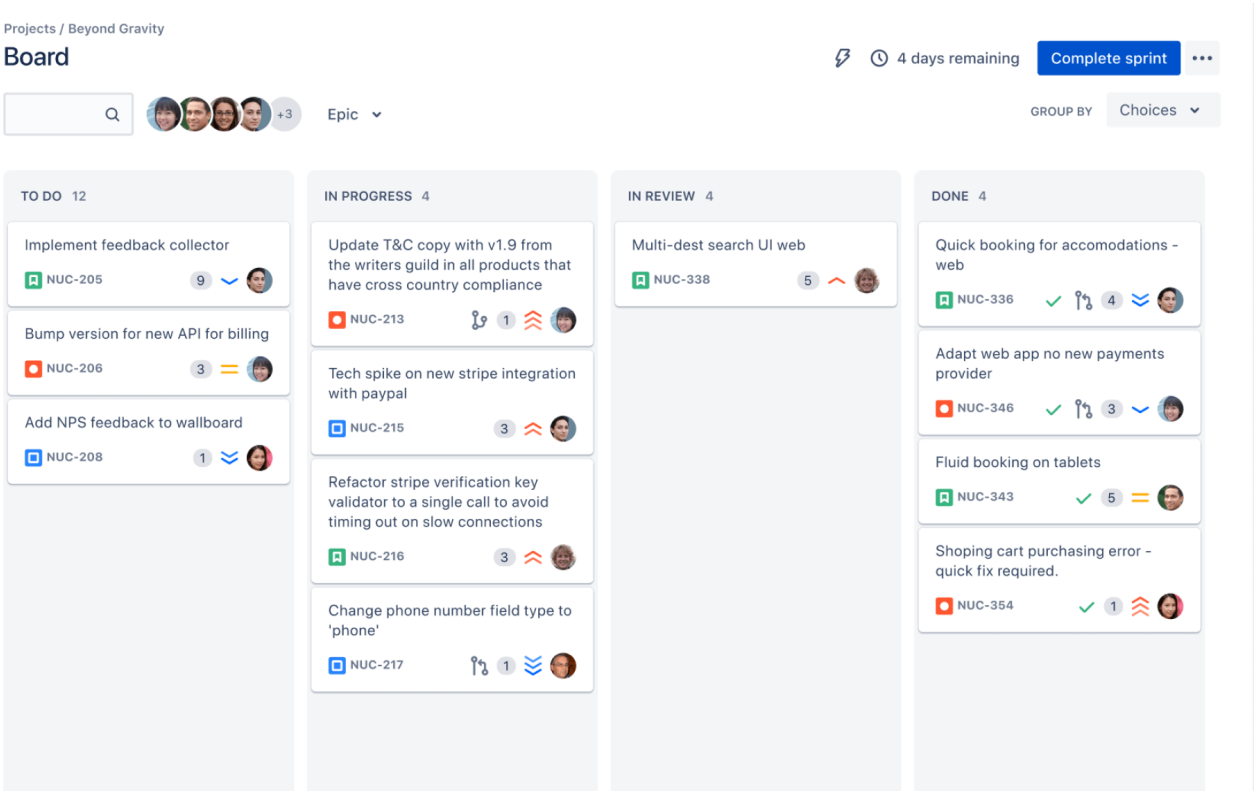


Figura 17: Ejemplo de proyecto en Jira[5]

RELEVAMIENTO DE CÓDIGO EXISTENTE Y DOCUMENTACIÓN ACTUAL

El primer paso para comenzar a escribir el nuevo código fue revisar el ya existente, así como también su documentación. Ambos elementos estaban a mano.

El código estaba disponible en los repositorios de la empresa, alojados en GitHub. GitHub es una plataforma donde se puede almacenar, compartir y trabajar junto con otros usuarios para escribir código, la cual implementa el sistema Git, para control de versiones de este [27].

[5] Imagen tomada de [55]

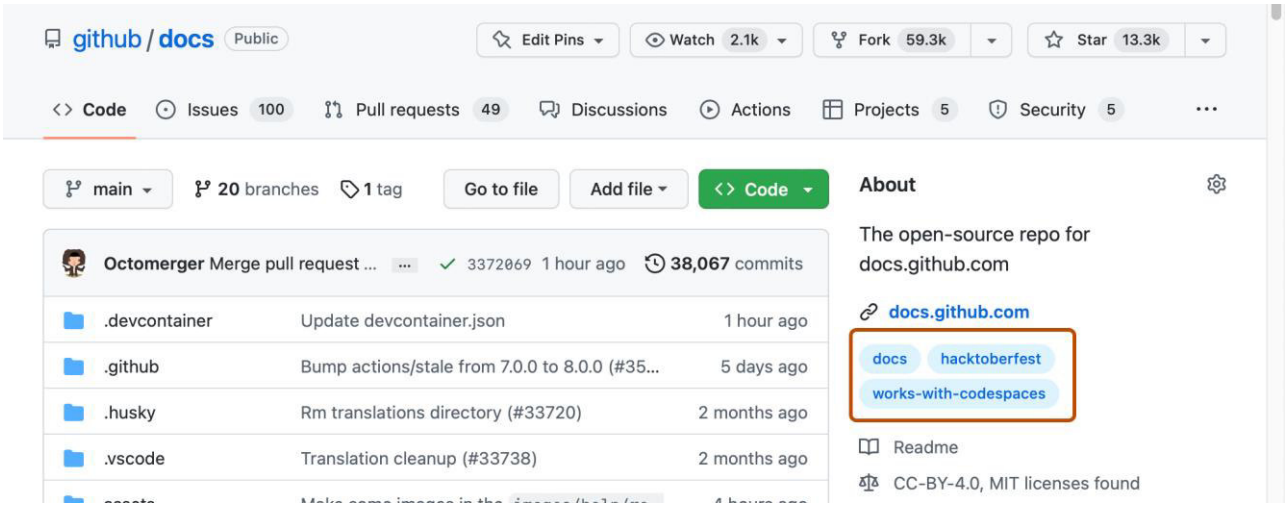


Figura 18: Ejemplo de proyecto en GitHub[6]

La documentación se encontraba disponible en fuentes internas de la empresa, escrita de una manera accesible, brindando un entendimiento general del sistema.

Por otra parte, se probó el código localmente, utilizando eventos de prueba. Con ello se pudo observar el funcionamiento de este y ver cómo generaba constancias (en este caso, de prueba), para entender el proceso.

DEFINICIÓN DE LA ARQUITECTURA DEL NUEVO SISTEMA

Para definir la arquitectura del nuevo sistema se llevó a cabo una reunión con el equipo de Arquitectura, debido a que, como se mencionó anteriormente, tiene la responsabilidad sobre este tema.

En la misma, después de varias horas de debate, se arribó a las siguientes definiciones:

- Implementar una arquitectura de microservicios, para distribuir las funciones del sistema.
- No incluir una capa intermedia entre el sistema y el publicador de eventos de la cadena de envíos, por considerarla innecesaria.

[6] Imagen tomada de [56]

- Almacenar las constancias solamente en Azure Blob Storage.
- Utilizar 2 bases de datos: una relacional para guardar la configuración del sistema y otra no-relacional para guardar registros relacionados al procesamiento de las constancias. Esta última opción proporciona una ventaja, al tener mayor flexibilidad para guardar campos relacionados al registro.

Si bien hubo una propuesta inicial sobre los microservicios, la misma se fue refinando con el paso del tiempo y se arribó a una arquitectura definitiva. A continuación, se describen los microservicios propuestos.

- **Templates** (Plantillas):

Este microservicio está conectado a la base de datos relacional del sistema, llamada **CE_Templates**, la cual contiene toda la configuración de este. Implementa una API mediante la cual provee endpoints tanto para la web de gestión así como también para proveerle información a otro microservicio llamado **Collector** (ya que la necesita, como se explicará a continuación). No procesa eventos. Al igual que el legado, consume información de una API que provee información sobre contratos, para hacer la configuración de estos.

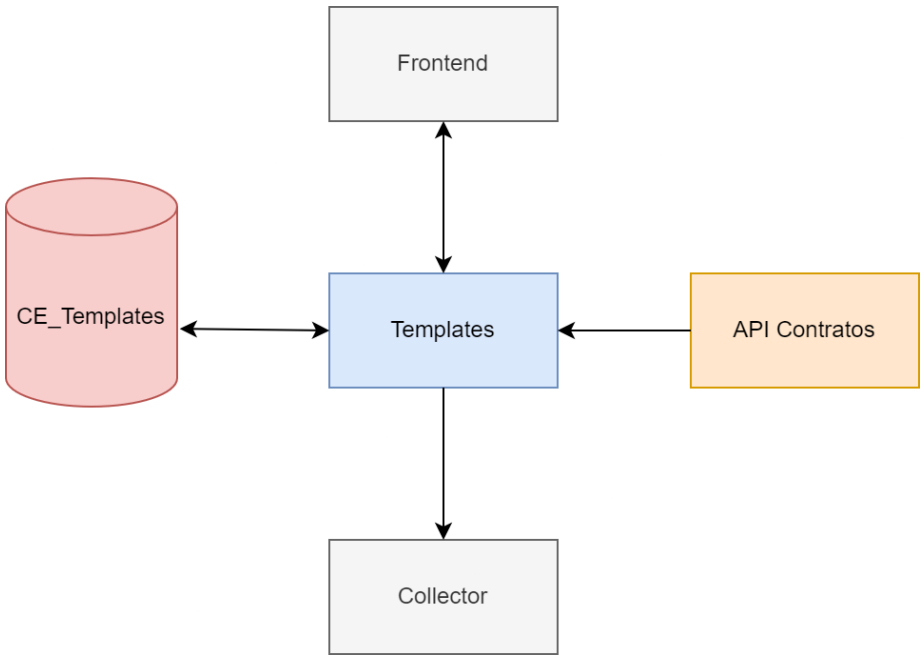


Figura 19: Diagrama del microservicio Templates

- **Collector** (Recolector):

El **Collector**, como indica su nombre, es el encargado de recolectar datos. Recibe los eventos de la cadena de envío y se encarga de hacer todas las llamadas necesarias a la base de datos de configuración (mediante un endpoint provisto por el microservicio **Templates**) y a las mencionadas APIs de los diversos sistemas que proveen información, para luego enviar los datos recolectados al microservicio **Builder** (encargado de generar las constancias, explicado más adelante).

Este servicio se conecta a la base de datos no-relacional del sistema, llamada **CE_Collector**, en la que crea un registro asociado a cada evento procesado, que contiene información general sobre el mismo, incluido un campo llamado **TrackStatus** (Estado de seguimiento), que indica el estado del procesamiento de este.

Cuando se crea el registro, **TrackStatus** se marca inicialmente como **Incomplete**, indicando que todavía está en procesamiento:

- En caso de que un evento se procese correctamente se genera un evento llamado **ReadyToBuild** (Listo para construir), que será consumido por el microservicio **Builder**. Este evento posee información para generar la constancia.
- En caso de que ocurra un error se genera un evento llamado **NotReadyToBuild** (No está listo para construir) para ser consumido por el mismo **Collector**, el cual actualiza el estado de la traza en base de datos a **Failed**, indicando que el procesamiento del envío falló. También consume el evento **GeneradaNotOk** (Generada no OK) generado por el **Builder**, replicando la lógica recién explicada.

Al igual que el sistema original, este microservicio también expone endpoints para descarga de constancias así como también para republicarlas, en caso de que sea necesario (ya sea por error o por necesidad en general).

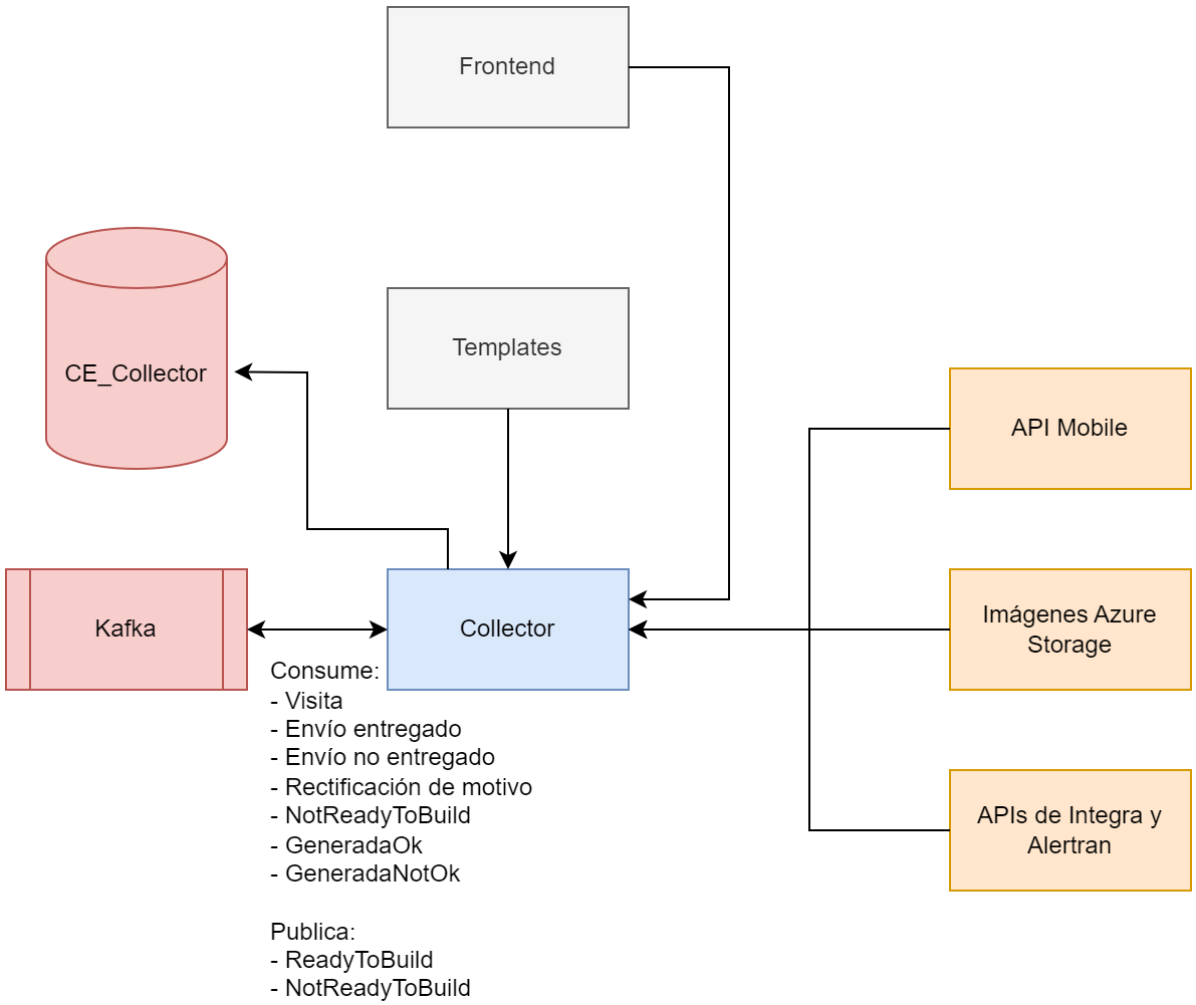


Figura 20: Diagrama del microservicio Collector

- **Builder** (Constructor):

Este es el servicio encargado de generar las constancias en términos físicos. Recibe la información de la constancia del **Collector**, mediante el evento **ReadyToBuild**, la cual utiliza para descargar todos los archivos que necesite (como imágenes de firmas), para luego generar el archivo y subirlo a Blob Storage. También está conectado directamente a la base de datos **CE_Templates** (si bien esto es un anti-patrón, se explica al final del párrafo el porqué de esta implementación), para consultar información sobre los módulos de las constancias. En caso de que la constancia se haya generado correctamente genera un evento **GeneradaOk**, con información sobre la misma. Este evento es consumido por el **Collector**, que actualiza el **TrackStatus** del envío a **Successful**, indicando que el

procesamiento del envío se completó exitosamente. En caso de error genera un evento **GeneradaNotOk** que, como se dijo anteriormente, es consumido por el **Collector** para actualizar el estado del procesamiento a **Failed**.

Como se dijo, este microservicio está conectado a la base de datos de **Templates**, lo cual se considera un anti-patrón, según la especificación de la arquitectura de microservicios. Sin embargo, se hizo una salvedad en este caso, ya que esto permite que el servicio siga funcionando en caso de que **Templates** falle. En un futuro, se podría hacer una mejor implementación.

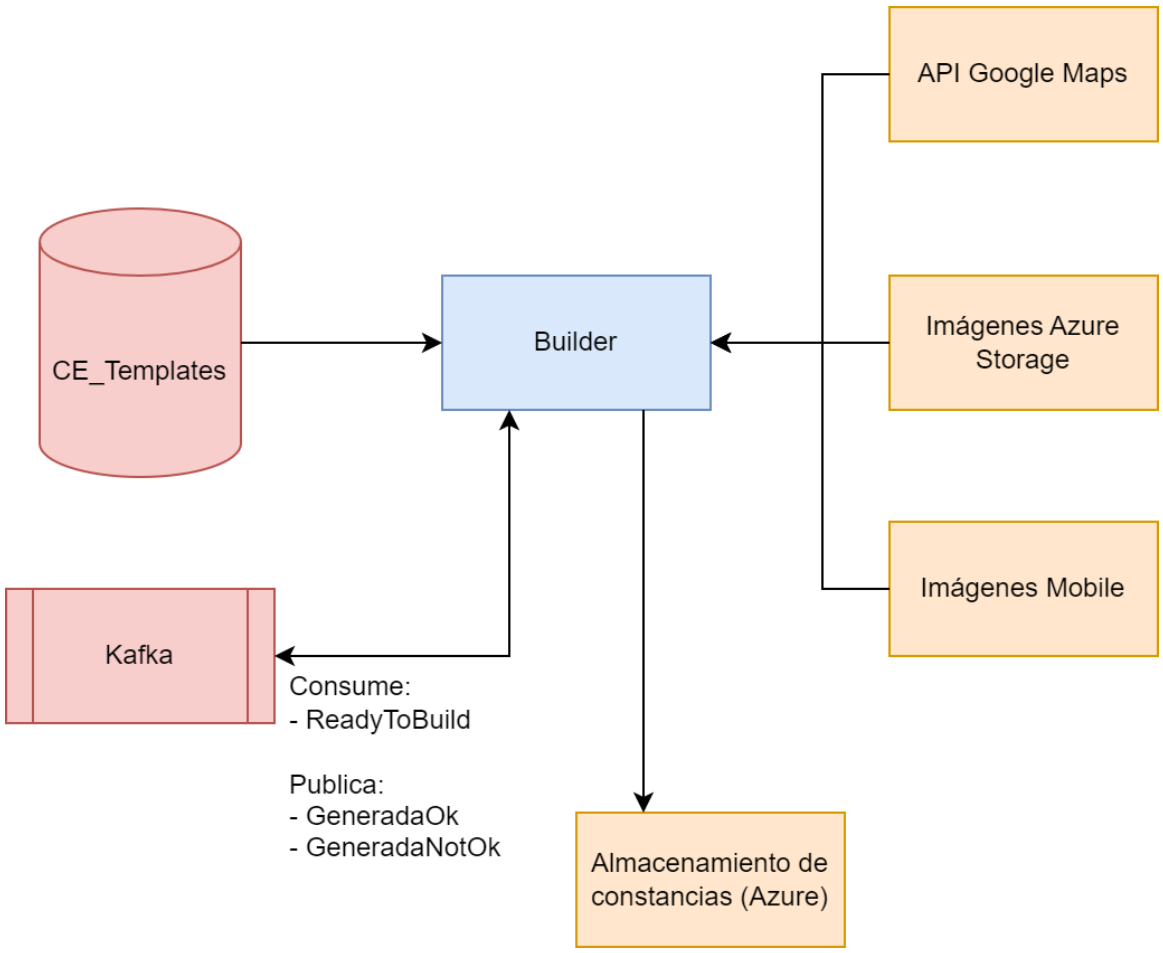


Figura 21: Diagrama del microservicio Builder

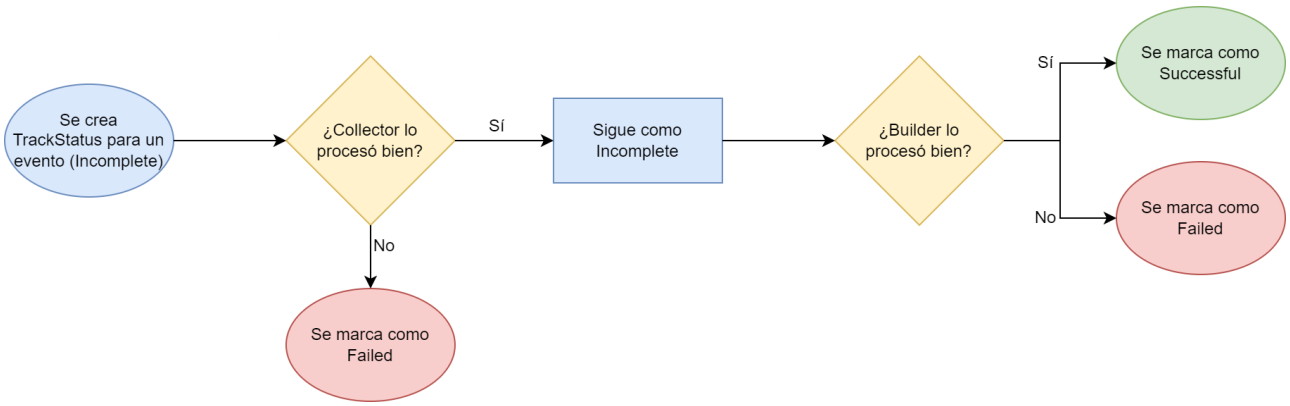


Figura 22: Diagrama de flujo de TrackStatus

- **Consultas**:

Servicio de apoyo al sistema, utilizado para descargar constancias desde el almacenamiento de estas.

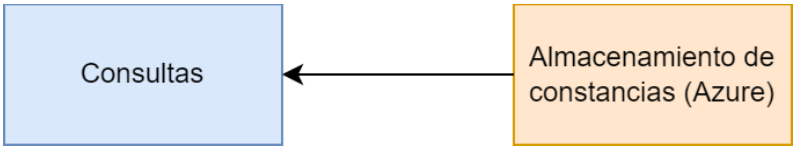


Figura 23: Diagrama del microservicio Consultas

TECNOLOGÍAS UTILIZADAS

.NET 8

Al igual que .NET Framework, .NET Core (o simplemente .NET, a partir de la versión 5) es un marco de desarrollo .NET (el cual se explicó anteriormente) para crear diversas aplicaciones, con los lenguajes C#, F# y Visual Basic. Presenta diferencias y mejoras:

- Soporte de plataforma: como se había mencionado, .NET Framework está diseñado para ejecutarse solo en Windows, mientras que .NET Core es multiplataforma, incluyendo macOS y Linux.
- Modelo de implementación: las aplicaciones .NET Core se pueden implementar mediante un modelo de implementación autónomo que incorpora el tiempo de ejecución

y las bibliotecas necesarias para ejecutarlas. En cambio, las aplicaciones de .NET Framework requieren que el tiempo de ejecución de .NET esté instalado en la máquina de destino.

- Rendimiento: .NET Core es más rápido que .NET Framework debido a su arquitectura liviana.
- Hay más, pero se mencionaron las más importantes [28].

KAFKA

Apache Kafka es una plataforma de transmisión de eventos, como mensajes.

Posee 3 capacidades importantes:

- Publicación y suscripción: permite publicar eventos y suscribirse a flujos de estos.
- Almacenamiento de eventos: permite almacenar transmisiones de eventos durante el tiempo que se necesite.
- Procesamiento de eventos: permite procesar eventos a medida que ocurren o retrospectivamente.

Estas funcionalidades se proporcionan de manera distribuida, escalable, elástica, tolerante a fallas y segura. Kafka se puede implementar en diversas plataformas, tanto en servidores locales como en la nube.

Kafka es un sistema distribuido, que consta de servidores y clientes, que se comunican a través de un protocolo de red TCP o *Transmission Control Protocol* (Protocolo de control de transmisión) de alto rendimiento [29]:

- Servidores: Kafka se ejecuta como un clúster (una agrupación) de uno o más servidores. Un clúster de Kafka es altamente escalable y tolerante a fallas.
- Clientes: permiten escribir aplicaciones distribuidas que leen, escriben y procesan flujos de eventos en paralelo, a escala y de manera tolerante a fallas.

Terminología:

- Evento: es un registro o mensaje que registra la ocurrencia de un evento en el mundo real (en este caso, en el negocio). Un evento tiene una clave, un valor, una marca de tiempo y metadatos opcionales. Ejemplo de logística:
    - **Clave de evento**: «0000001» (número de envío)
    - **Valor del evento**: «Comienzo de distribución»
    - **Marca de tiempo del evento**: «3 de octubre de 2024, a las 06:00»
- Productores: son las aplicaciones cliente que publican eventos en Kafka.
- Consumidores: son aquellas que se suscriben (leen y procesan) eventos. Los productores y los consumidores no necesitan conocerse.
- Tópicos: los eventos se organizan y almacenan de forma duradera en tópicos, que conceptualmente son similares a una carpeta en un sistema de archivos, y los eventos son los archivos de esa carpeta. Un ejemplo de nombre de tópico podría ser **EventosDeDistribucion**, para seguir el ejemplo anterior. Los tópicos en Kafka siempre son de múltiples productores y múltiples suscriptores. Sus números, en ambos casos, pueden ser 0 o muchos. A diferencia de los sistemas de mensajería tradicionales, los eventos no se eliminan inmediatamente después del consumo (como sí ocurre en MQ).

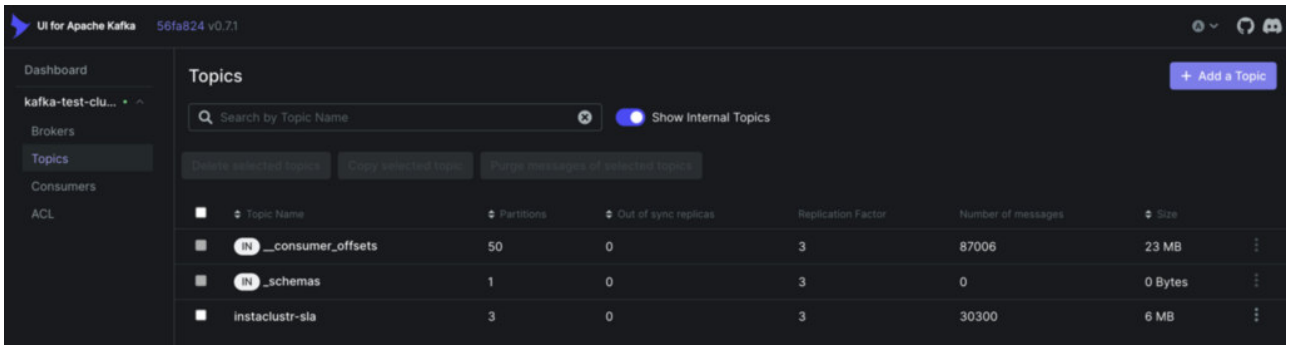


Figura 24: Ejemplo de interfaz web para Kafka[7]

[7] Imagen tomada de [57]

Los tópicos se distribuyen en varios almacenamientos, ubicados en diferentes *brokers* (gestores) de Kafka. Cuando se publica un nuevo evento en un tópico en realidad se agrega a una de las particiones del tópico (un tópico permite particiones). Se garantiza que cualquier consumidor de una partición de tópico determinada siempre leerá los eventos en el mismo orden en que fueron escritos (en este caso, funciona como una estructura del tipo cola).

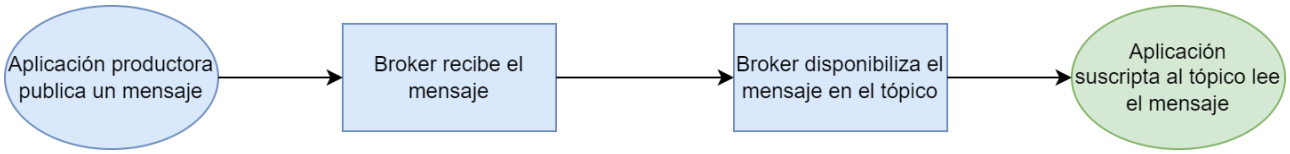


Figura 25: Flujo general de un mensaje en Kafka

Para que los datos sean tolerantes a fallas y de alta disponibilidad, todos los tópicos se pueden replicar a lo largo de varios brokers, siendo habitualmente 3 el factor de replicación [30].

Se implementó Kafka como reemplazo de MQ, debido a que está mejor alineado con respecto a las necesidades del negocio.

OpenShift

Es una plataforma empresarial de despliegue de aplicaciones, la cual se diseñó especialmente para formar parte de una estrategia de nube híbrida. Ofrece operaciones automatizadas integrales, una experiencia uniforme en todos los entornos (ambientes de despliegue) y la implementación de autoservicio para los desarrolladores. Los equipos pueden trabajar en conjunto sobre la misma, para llevar a cabo las ideas de la etapa de desarrollo a la de producción de manera eficiente.

Está disponible de 2 formas:

- Como servicio de nube, gestionado en las nubes públicas más utilizadas.
- Como sistema de software autogestionado para las empresas, que necesitan un mayor grado de personalización.

La empresa dispone de OpenShift *on-premise*, lo cual significa que está instalado en hardware propiedad de la empresa. También se dispone de OpenShift en la nube, el cual originalmente se comenzó a utilizar como estrategia de contingencia [31].

OpenShift está basado en Kubernetes, que es una plataforma portátil, extensible y de código abierto para gestionar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Tiene un ecosistema grande y de rápido crecimiento [32].

Con esto se puede desplegar una aplicación en un contenedor, en vez de ejecutarla como un servicio local.

OpenShift está implementado con una interfaz web, desde la cual se puede gestionar todo el sistema. El mismo permite el despliegue de un servicio en un contenedor y tiene la opción de replicar instancias de este (escalamiento horizontal), así como también cambiar la cantidad de recursos asignados (escalamiento vertical). Posee también una funcionalidad importante y de interés para el negocio, que es la posibilidad de hacer un auto-escalamiento horizontal, útil en casos de mucha demanda.

Por último, permite visualizar los logs de la aplicación en tiempo real y también diversas métricas relacionadas a la misma.

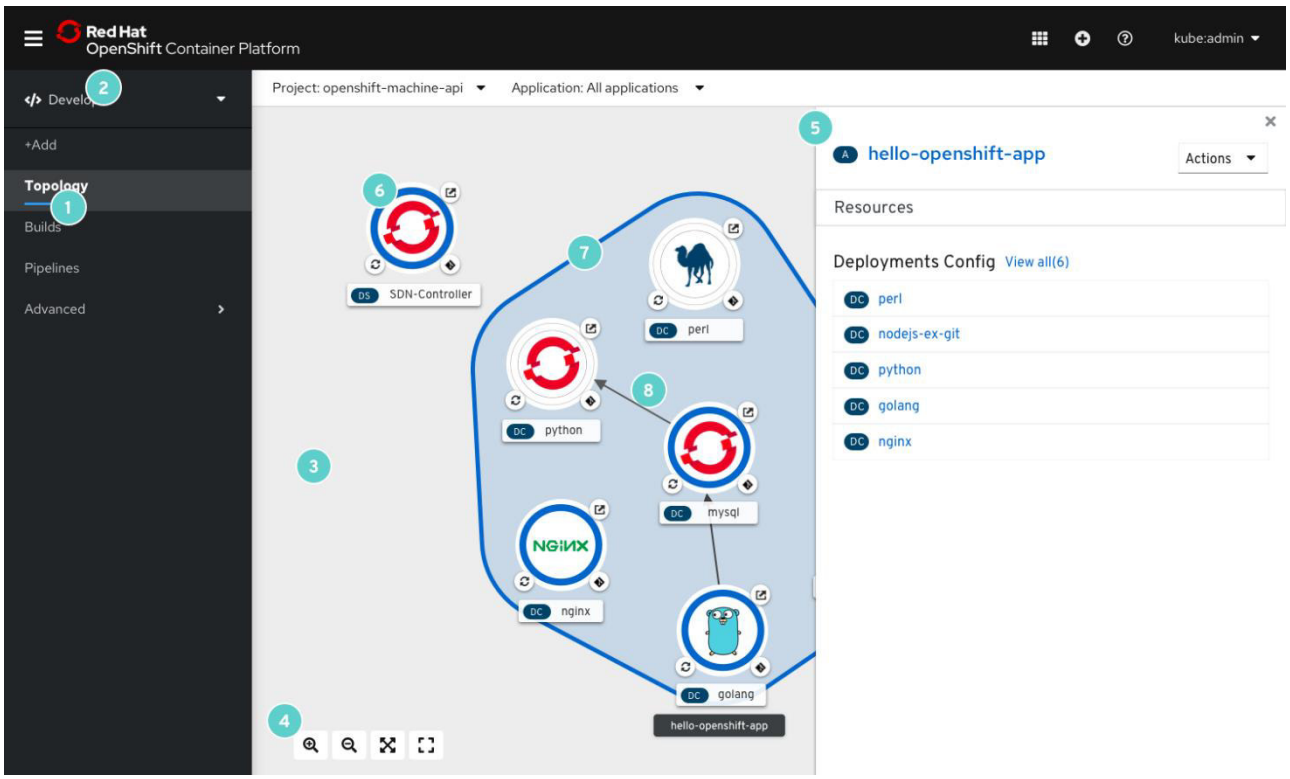


Figura 26: Interfaz web de OpenShift[8]

SQL Server

Esta tecnología ya fue explicada anteriormente, en la sección donde se describe el sistema legado. No presenta cambios. Lo que sí se hizo fue un pedido de creación de una nueva base de datos (**CE_Templates**), para no utilizar la original.

Azure Blob Storage

Ídem punto anterior. De nuevo, se hizo un pedido de creación de un nuevo contenedor para guardar las constancias.

[8] Imagen tomada de [58]

MongoDB

MongoDB es una base de datos de documentos (no-relacional) que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado. Características principales:

- MongoDB almacena datos en documentos flexibles, en formato BSON (*Binary JSON*), el cual está basado en el conocido formato JSON (*JavaScript Object Notation*, Notación de objetos en JavaScript) [33].
- El modelo de documento se asigna a los objetos en el código de una aplicación para facilitar el trabajo con los datos, similar a un sistema de ORM.
- Es una base de datos distribuida en su núcleo, por lo que la alta disponibilidad, la escalabilidad horizontal y la distribución geográfica están integradas y son fáciles de usar [34].

Se implementó MongoDB para el microservicio Collector, solicitando la creación de la base de datos **CE_Collector**.

Entity Framework Core

Entity Framework (EF) Core es un ORM, y es una versión ligera, extensible, de código abierto y multiplataforma de la popular tecnología de acceso a datos Entity Framework.

EF Core actúa como asignador relacional de objetos [35].

Se implementó EF Core en este sistema para las llamadas a la base de datos SQL Server, del tipo escritura, porque para las de lectura se implementó otro ORM (Dapper), explicado a continuación.

DAPPER

También es un ORM, simple, para .NET. A diferencia de Entity Framework Core, las consultas a ejecutar se deben escribir manualmente en el código. Pero tiene la ventaja de que es más rápido para consultas de lectura, por lo que está incorporado en el stack tecnológico de la empresa [36].

ELASTIC APM

Es un agente que mide automáticamente el rendimiento de una aplicación y realiza un seguimiento de los errores. La sigla APM significa *Application Performance Monitoring* (Monitoreo del rendimiento de aplicaciones). Registra los métodos llamados por una aplicación para los eventos de fuente de diagnóstico integrados. Con esto, los eventos admitidos activan el código del agente para medir su duración y recopilar metadatos, como declaraciones de base de datos, llamadas a APIs, errores, y más.

Estos eventos, llamados transacciones y tramos, se envían al servidor APM. El servidor los convierte a un formato adecuado para Elasticsearch (un tipo de base de datos no-relacional utilizada en la empresa) y los envía a un clúster de esta. Se utiliza APM en Kibana (una aplicación web para visualizar los datos) para obtener información diversa y de manera visual sobre la aplicación, incluyendo los problemas de latencia y causas de errores [37].

La implementación de APM permite acceder a un sinfín de datos, con los cuales se pueden tener métricas y registros del sistema.

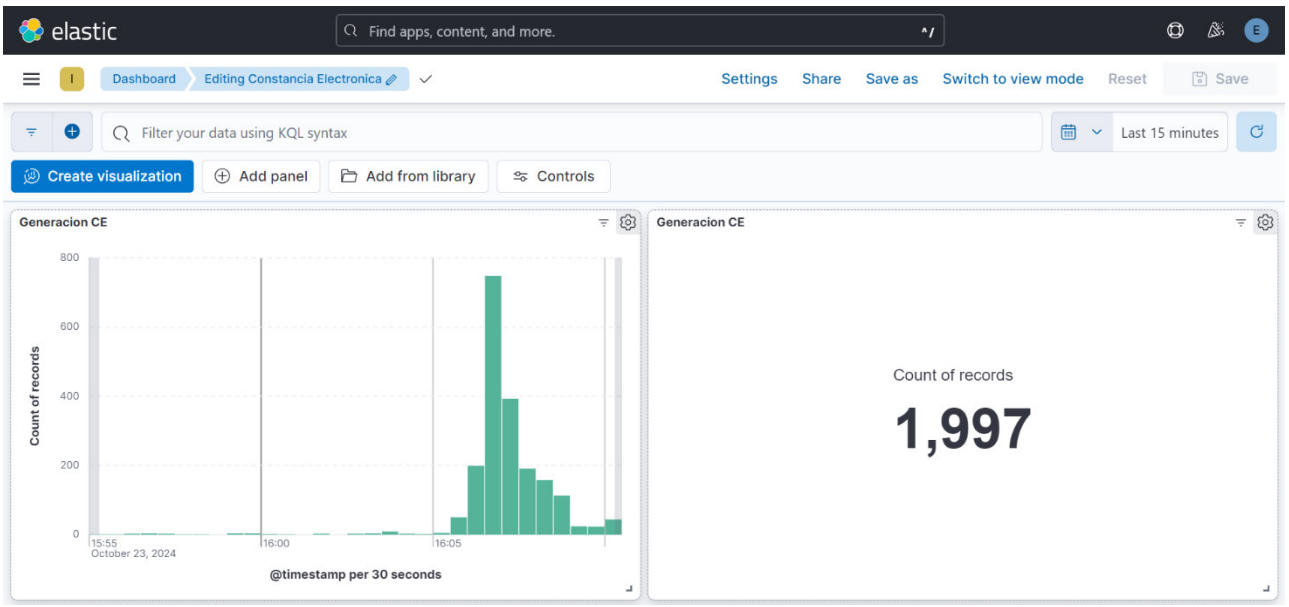


Figura 27: Interfaz web de Kibana, se muestra información de constancias generadas

SonarQube

Es una herramienta web que realiza un análisis estático del código de una aplicación. Lo que hace es identificar los puntos susceptibles de mejora.

Información que muestra:

- Puerta de calidad (*Quality Gate*): son una serie de condiciones (reglas) que el proyecto analizado debe cumplir para poder pasar a una siguiente etapa (por ejemplo, la cobertura sobre pruebas unitarias, que se explicarán más adelante).
- *Bugs* (bichos, en referencia a errores) y vulnerabilidades: hacen referencia tanto a errores reales o potenciales en el software como a riesgos de seguridad, que pueden ser usados como foco de un ataque (por ejemplo, una cadena de texto que debe estar en una sección de contraseñas).
- Olores en el código (*Code smells*): es un indicador de que posiblemente no se está escribiendo código de una manera óptima, lo que puede ocasionar algún problema en el futuro, normalmente problemas de mantenibilidad del código. Los code smells no

implican que el código contenga errores o bugs, pero sí pueden aumentar el riesgo a errores o fallos (ejemplo de code smell: un constructor que tiene muchos parámetros).

- Cobertura (*Coverage*): la cobertura de código es una medida que permite conocer el porcentaje de código que ha sido probado o validado por pruebas. Usualmente se desea obtener una mayor cobertura para tener menos probabilidad de introducir errores en el código [38].

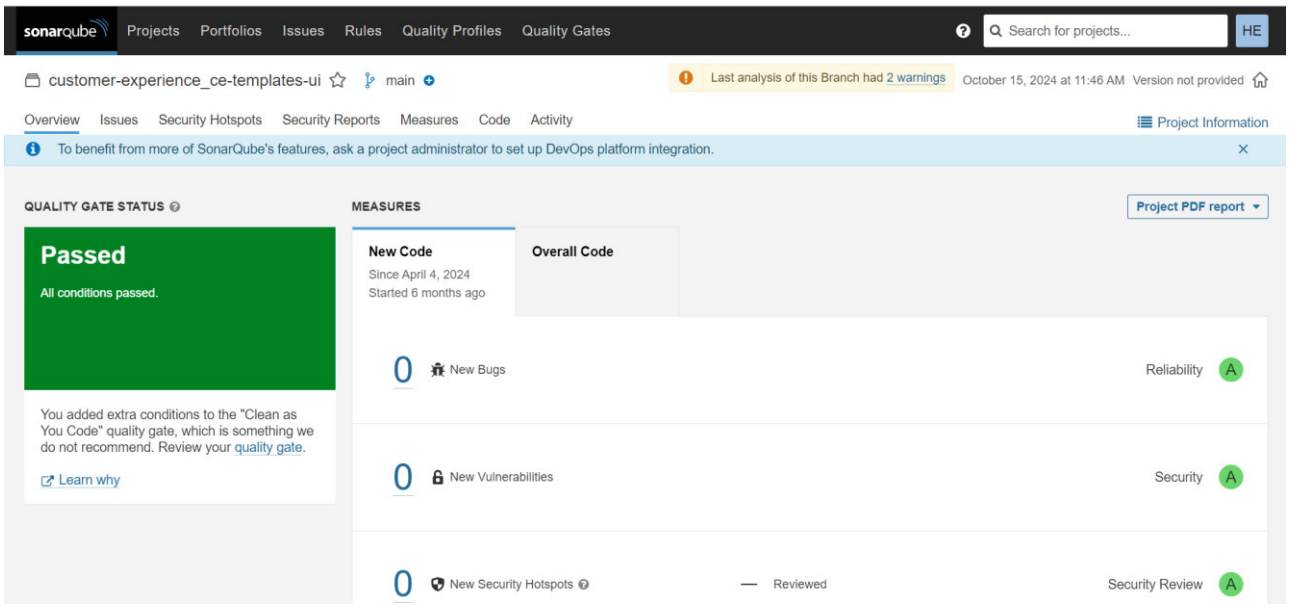


Figura 28: Interfaz web de SonarQube, se muestra información del Collector

En este caso, SonarQube está integrado con GitHub. Cada vez que se sube código a un repositorio se hace un análisis automático del mismo.

DESARROLLO GENERAL DE LOS MICROSERVICIOS

En esta sección se explica el desarrollo general de los microservicios, paso a paso. Después, más adelante, se detallan cada uno de estos pasos.

- Creación de repositorios en GitHub, para almacenamiento del código.

- Creación de los proyectos vacíos usando la plantilla **Platform** (la explicación sobre esto se da más adelante): una vez creados los repositorios, se procedió a crear el proyecto inicial para cada uno. Acto seguido, se hizo la subida del código a los repositorios.
- Integración con Kafka (publicadores, consumidores y configuración de estos).
- Creación de endpoints, inicialmente siempre devolviendo un estado correcto: se crearon los endpoints como *mocks*, esto es, que el microservicio exponga los mismos de manera simulada, y siempre devolviendo un resultado correcto.
- Agregado de validaciones para los parámetros de entrada de los endpoints.
- Creación de lógica de negocio para los endpoints.
- Conexiones a las bases de datos.
- Conexiones a las diversas APIs que utiliza el sistema y a Azure Blob Storage.
- Integración con Elastic APM.
- Creación de lógica de negocio para los eventos.
- Escritura de pruebas unitarias del código.
- Aplicación de medidas de seguridad.
- Configuración general del sistema.
- Corrección de errores y vulnerabilidades en el código, ya sea encontrados manualmente así como también por SonarQube.
- Intercambio con el equipo de QA: correcciones en el código, debido la retroalimentación proporcionada por este equipo.
- Intercambio con el equipo de frontend, cuya tarea fue desarrollar la nueva web de configuración.
- Despliegue en producción.

BUENAS PRÁCTICAS APLICADAS SOBRE EL CÓDIGO

PRINCIPIOS SOLID

Los principios SOLID, tal como indica Robert C. Martin en su artículo «Getting a SOLID start» son principios recomendables para basarse a la hora de escribir código. Son 5:

1. S: *Single Responsibility Principle* (SRP). Principio de Responsabilidad Única.
2. O: *Open/Closed Principle* (OCP). Principio de Abierto/Cerrado.
3. L: *Liskov Substitution Principle* (LSP). Principio de Sustitución de Liskov.
4. I: *Interface Segregation Principle* (ISP). Principio de Segregación de Interfaces.
5. D: *Dependency Inversion Principle* (DIP). Principio de Inversión de Dependencias.

Sus objetivos:

- Crear un software, en términos generales, eficaz: que cumpla con su propósito, que sea robusto y estable.
- Escribir un código limpio y flexible ante cambios: que sea fácilmente modificable según necesidad, que sea reutilizable y mantenible.
- Permitir escalabilidad: que pueda ser ampliado con nuevas funcionalidades de manera ágil.
- En definitiva: desarrollar un software de calidad.

Definiciones de 2 términos pertinentes al tema:

- Acoplamiento: es el grado de interdependencia que tienen 2 unidades de software entre sí. Ejemplos de unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etc.
- Cohesión: es el grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajan por separado.

La aplicación de los principios SOLID permite mantener una alta cohesión y, por lo tanto, un bajo acoplamiento de código.

Definiciones de los principios:

1. **Principio de Responsabilidad Única**:

«Una clase debería tener una, y solo una, razón para cambiar». La «razón para cambiar», es lo que se identifica, en este caso, como responsabilidad. Consiste en reunir las cosas que cambian por las mismas razones y separar aquellas que cambian por razones diferentes.

2. **Principio de Abierto/Cerrado**:

«Se debe ser capaz de extender el comportamiento de una clase, sin modificarla». Las clases deberían estar abiertas para poder extenderse y cerradas para modificarse.

3. **Principio de Sustitución de Liskov**:

Este principio dice que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema, o equivalentemente: si en un programa se utiliza cierta clase se debería poder usar cualquiera de sus subclases, sin interferir en su funcionalidad.

4. **Principio de Segregación de la Interfaz**:

Es preferible contar con muchas interfaces que definan pocos métodos, en vez de tener una interfaz que implemente muchos.

5. **Principio de Inversión de Dependencias**:

Los módulos de alto nivel no deben depender de los de bajo nivel y ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles, sí debe ocurrir la inversa. El objetivo es reducir las dependencias entre los módulos del código, para alcanzar un bajo acoplamiento de las clases.

Críticas a SOLID:

SOLID no es ajeno a los debates del desarrollo del software. Entre las críticas a los principios están:

- Son ambiguos
- Son confusos
- Complican el código
- Demoran el proceso de desarrollo
- Están equivocados y son innecesarios

Se debe tener en cuenta que los principios SOLID son eso: principios, es decir, buenas prácticas que pueden ayudar a escribir un mejor código. No son reglas, ni leyes, ni verdades absolutas, sino más bien soluciones heurísticas, basadas en la experiencia [39].

Diseño Orientado al Dominio

Diseño Orientado al Dominio (*Domain Driven Design*), es un enfoque de desarrollo de software utilizado por Eric Evans en su libro «Domain-Driven Design — Tackling Complexity in the Heart of Software, 2004». Consiste en distintas claves, terminología y patrones utilizados para desarrollar software donde el concepto de dominio representa, de cierto modo, el negocio de una organización.

Sus principios son:

- Colocar los modelos y reglas de negocio (el dominio) en el centro de la aplicación.
- Basar un dominio en un modelo de software.
- Lograr un nivel adecuado de colaboración entre expertos del dominio y los desarrolladores, para desarrollar software con objetivos claros.

Beneficios:

- Comunicación efectiva entre expertos del dominio y expertos técnicos (los desarrolladores) a través de un lenguaje en común, ubicuo (*Ubiquitous Language*).

- Posibilidad de focalizar el desarrollo de un área dividida del dominio (subdominio) a través de los llamados contextos divididos o acotados (*Bounded Contexts*).
- El software es más cercano al dominio, y por lo tanto más cercano al cliente.
- El código está organizado de tal manera que permite probar las distintas partes del dominio, de manera aislada.
- La lógica de negocio reside en un solo lugar y está dividida por contextos.
- Mantenibilidad a largo plazo.

Inconvenientes:

- Aislar la lógica puede llevar tiempo.
- Se necesita un experto en el dominio.
- Una curva de aprendizaje alta.

Definición de dominio:

Es el asunto específico que se aborda. Debe ser representado por un modelo. Representa el conocimiento de un experto en el negocio. El primer y principal requerimiento de un modelo de dominio es que tenga reglas consistentes. Consideraciones sobre los conceptos de dominio y subdominio:

- **Dominio principal**: es lo que marca la diferencia clave para el negocio.
- **Subdominio**: cada subdominio representa una característica que debe cumplir el software. Se lo puede entender como un dominio más acotado, cohesivo y comprensible.

El dominio debe ser una capa (o más de una) de la arquitectura de una aplicación, y no debe tener intrusiones externas: el dominio se debe pensar como el corazón de la aplicación. La o las capas de dominio son responsables de representar la información del negocio, su lógica y situaciones, sin importar la implementación de la parte de infraestructura.

Lenguaje ubicuo:

La comunicación sin ambigüedades entre los desarrolladores y los expertos del dominio es esencial para llevar adelante el desarrollo. Un lenguaje común es necesario, para evitar problemas futuros y tener éxito.

Como los desarrolladores están acostumbrados a hablar en términos técnicos y los expertos de dominio no tienen por qué entenderlos, se debe encontrar un lenguaje común entre ambas partes.

Elementos comunes del dominio:

- **Entidades** (*Entities*): son objetos que tienen identidad, y por lo tanto tienen un identificador único. Dos entidades del mismo tipo siempre tienen diferente identificador, aunque tengan los mismos valores. Tienen la capacidad de ser buscadas y almacenadas.
- **Objetos de valor** (*Value Objects*): son objetos sin identificador que describen medidas, cantidad, dinero, etc. Además de carecer de identidad, tienen características como la inmutabilidad (no se modifican sus atributos, solamente se puede crear uno nuevo).
- **Servicios** (*Services*): representan lógica de negocio que no pertenece a ninguna entidad ni objeto de valor. Ejemplo: la transferencia de dinero entre 2 cuentas bancarias. Los servicios que solo representan lógica de negocio e interactúan con objetos del dominio se deben encontrar en la capa de dominio o aplicación, mientras los que interactúan con entidades externas deben estar en la capa de infraestructura, por ejemplo: envío de e-mail, imprimir, etc.
- **Agregaciones** (*Aggregates*): conjunto de objetos que tienen una relación y todas sus operaciones deben realizarse a través de esta raíz. Ningún objeto del exterior puede tener referencias a los objetos dentro de la agregación, toda operación pasa por la raíz.
- **Fábricas** (*Factories*): las fábricas son objetos con la responsabilidad de fabricar otros objetos complejos, centralizando en las mismas el conocimiento de la fabricación.

- **Repositorios** (*Repositories*): objetos que encapsulan la lógica de persistencia de las entidades [40].

ARQUITECTURA LIMPIA

Considerando que la empresa posee un amplio espectro de equipos y tecnologías para el desarrollo, donde se evidencian diversas prácticas y conocimientos para desarrollar y resolver las necesidades del negocio, el equipo de Arquitectura decidió implementar un estándar de arquitectura de software (llamado Arquitectura Limpia), que sirva para que todas las aplicaciones desarrolladas tengan un formato común. Para eso se implementó en una primera instancia Platform, una herramienta que disponibiliza esta arquitectura para los diversos proyectos de código. Con esto, se asegura control, calidad, agilidad y mantenibilidad en el tiempo.

Definición de Arquitectura Limpia:

Es un término introducido por Robert C. Martin. Él recopiló los modelos de capas más utilizados en una versión mejorada a la que llamó *Clean Architecture*.

Sus 5 principios:

1. **Independencia de cualquier marco de desarrollo**: debe ser capaz de aplicarse a cualquier marco de desarrollo, sin importar lenguaje o librerías. Las capas deben quedar separadas al punto de sobrevivir independientemente.
2. **Verificable**: entre más pura sea una unidad de software (función, módulo, etc.) más fácil será predecir el resultado que se obtiene de la misma. En este caso, algo puro es algo que no tiene efectos colaterales. Las unidades deben ser capaces de ser probadas individualmente.
3. **Independiente de la interfaz de usuario**: ningún cambio en la interfaz de usuario (UI, *User Interface*) debe alterar el resto del sistema. Por ejemplo, se debe poder cambiar una UI móvil por una en modo consola.

4. **Independiente de la base (o las bases) de datos**: las capas deben tener tanta independencia al punto que se deben poder agregar múltiples fuentes de datos, e incluso múltiples fuentes del mismo tipo de dato.
5. **Independiente de cualquier elemento externo**: si el sistema necesita de una librería, otro sistema o cualquier otro elemento a conectar, debería ser fácilmente ensamblado y también debería ser modularizado. Esta capa externa debería ser transparente.

Motivos para utilizar Arquitectura Limpia:

- Se aplica a múltiples tecnologías.
- Desacoplamiento de capas.
- Flexibilidad en la mantenibilidad del software (a la hora de añadir o remover funcionalidades).
- Diseño basado en componentes con responsabilidades bien definidas.
- Pruebas: oportunidad de hacer pruebas de manera rápida y fácil.
- Calidad: producto sólido, de calidad y escalable [41].

Es importante destacar que el concepto de Arquitectura Limpia va más allá de una estructura de carpetas estándar (que obviamente es importante utilizarla), también involucra el cumplimiento de los principios mencionados (el cual es coadyuvado por la estructura mencionada).

Código Limpio

El término *Clean Code* (Código limpio) se utiliza para referirse a un código que es fácil de leer, comprender y mantener. Robert C. Martin popularizó este término al escribir «Clean Code: A Handbook of Agile Software Craftsmanship» en 2008. En este libro, presentó un conjunto de principios y prácticas recomendadas para escribir código limpio, tales como:

- Uso de nombres significativos

- Funciones cortas
- Comentarios claros
- Formato coherente

El objetivo del código limpio es crear software funcional, legible, mantenible y eficiente durante todo su ciclo de vida.

Principios de Clean Code:

1. **Evitar los números (o cadenas de texto) codificados de forma rígida (*hardcoding*)**: utilizar constantes con nombres significativos y a las mismas asignarles los valores, en vez de hacer hardcoding de estos.
2. **Usar nombres significativos y descriptivos**: elegir nombres para las variables, funciones y clases que reflejen su propósito y comportamiento. De cierto modo esto hace que el código se autodocumente (esto implica que se hace más fácil de entender).
3. **Usar comentarios con moderación y que sean significativos**: no comentar cosas obvias. El exceso de comentarios o los comentarios poco claros saturan el código.
4. **Escribir funciones cortas que solo hagan una cosa**: esto se deriva del Primer Principio SOLID, el de Responsabilidad Única (SRP). Las funciones son más comprensibles, legibles y fáciles de mantener si solo tienen una tarea, además hace que probarlas sea más fácil.
5. **Seguir el principio DRY o *Don't Repeat Yourself* (No se repita a usted mismo) y evitar la duplicación de lógica o código**: evitar escribir el mismo código más de una vez. Es mejor reutilizar el código usando funciones, clases, módulos, bibliotecas u otras abstracciones, esto reduce el riesgo de errores y fallas, ya que solo se necesita modificar el código en solo un lugar en caso de necesidad de cambio o actualización de este.
6. **Respetar los estándares establecidos para la escritura de código**: seguir las convenciones de un lenguaje de programación en términos de espaciado, comentarios

y nombres. La mayoría de los lenguajes de programación tienen estándares de codificación y guías de estilo, aceptados por la comunidad.

7. **Encapsular condicionales anidados en funciones**: una forma de mejorar la legibilidad y claridad de las funciones es encapsular bucles *if-else* anidados en otras funciones con nombres descriptivos, aclarando su propósito. También facilita la reutilización, modificación y prueba de la lógica sin afectar al resto de la función.
8. **Refactorizar continuamente**: revisar y refactorizar regularmente el código para mejorar su estructura, legibilidad y facilidad de mantenimiento.
9. **Usar el control de versiones**: los sistemas de control de versiones (como Git) rastrean cada cambio realizado en el código, lo que permite comprender la evolución del código y volver a versiones anteriores si es necesario [42].

IMPLEMENTACIÓN Y COMBINACIÓN DE LAS BUENAS PRÁCTICAS

La combinación de los Principios SOLID, Diseño Orientado al Dominio, Arquitectura Limpia y Código Limpio permite maximizar la calidad del código, con cada concepto haciendo su contribución a la misma. No solo eso, sino que también se complementan, no son principios independientes.

Por ejemplo:

- Una de las capas de Arquitectura Limpia es la de dominio, cuyos objetos representan las entidades del dominio en términos de negocio.
- Otra de las capas es la de aplicación, que contiene la lógica de negocio. Cada caso de uso representa un contexto acotado.
- El uso de interfaces en la capa de aplicación, cuyas implementaciones se encuentran en la capa de infraestructura, es un ejemplo del Principio SOLID de Inversión de Dependencias (DIP).

Estructura de capas utilizadas en este desarrollo:

La capa **Dominio** tiene la responsabilidad de almacenar todas las clases comunes a la aplicación. Contenido de cada uno de sus directorios:

- **Common** (Común): uso común (por ejemplo, enumeraciones).
- **Entities** (Entidades): clases que representan las entidades.
- **Dtos** (*Data transfer objects*, Objetos de transferencia de datos): clases utilizadas para transferencia de datos.
- **Exceptions** (Excepciones): clases de excepciones.
- **Interfaces**: son interfaces que se consideran parte del dominio.

La capa **Aplicación** contiene la lógica de negocio para los diversos casos de uso del sistema. Interactúa con la de dominio para obtener las entidades. No debe tener dependencias ni lógica con recursos externos. Para interactuar con estos se debe utilizar el Principio de Inversión de Dependencias de SOLID. Contenido de cada directorio:

- **Bootstrap** (Arranque): configuración de las dependencias que se necesitan inyectar.
- **Common**: almacenamiento de clases comunes, como los servicios (que solo interaccionan con el dominio), interfaces, modelos, excepciones, clases de ayuda, etc.
- **UseCase** (Caso de uso): directorio principal de la capa, almacena todos los casos de uso de la aplicación. Las consultas se separan por versión y también según sean de lectura o de escritura.

La capa **Infraestructura** vincula el sistema con las dependencias externas, como bases de datos, recepción de eventos y conexiones a APIs. Contenido de sus directorios:

- **Persistence** (Persistencia): carpeta donde se almacena la lógica de interacción con la base de datos.
- **Bootstrap**: ídem capa de aplicación, pero para infraestructura.
- **Kafka**: consumidores y publicadores de Kafka.

- **Provider** (Proveedor): configuraciones de conexiones externas (Azure Storage, Refit).
- **Services**: en el caso del servicio Templates se colocaron las clases repositorio acá, aunque esta parte del código podría ir en **Persistence**.

Web API/Worker:

Esta capa es la más externa de los servicios, contiene **Program.cs** (un archivo principal en aplicaciones .NET). En el caso de que exponga una API interactúa directamente con la capa de aplicación para resolver los pedidos que le llegan. Si no expone una API entonces no. Contenido:

- **Controllers** (Controladores): carpeta con las clases de controladores de endpoints. Hay versionado y división por método REST. Los métodos REST, basados en HTTP, son **Get** (obtener), **Update** (actualizar), **Delete** (borrar), entre otros. Está solo disponible si es un servicio que expone una API.
- **Properties** (Propiedades): carpeta con archivos de configuración.

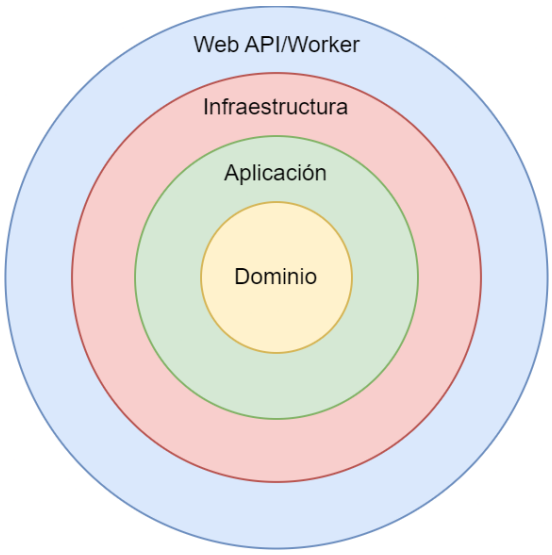


Figura 29: Estructura de capas en los proyectos

CREACIÓN DE REPOSITORIOS Y USO DE PLANTILLAS DE APLICACIÓN

Para el desarrollo inicial de los microservicios se utilizó **Platform**, la cual es una herramienta compuesta por diversas librerías estándares para el desarrollo, que permiten un desarrollo ágil, estándar y de calidad.

El objetivo del concepto de Platform es que, a través de un conjunto de librerías, se tenga una metodología de desarrollo estándar para toda la empresa, no solo arquitectónicamente sino también en cuanto a uso e implementación de las librerías y herramientas en general.

Platform permite generar un proyecto definido y además dispone de un conjunto de librerías estándares para los desarrollos.

Tiene 2 plantillas: API y *Worker* (trabajador). La primera se utiliza cuando se necesita que un microservicio exponga endpoints. La segunda se utiliza cuando el servicio solo corre en el fondo (por eso se denomina worker, porque trabaja sin exponer una API) [43].

Los párrafos anteriores pretenden explicar qué es Platform. Lo primero que se hizo fue crear los repositorios correspondientes en GitHub. Acto seguido, por cada microservicio se creó el proyecto con Platform y se hizo la subida del código.

CREACIÓN DE ENDPOINTS: MOCKS, VALIDADORES Y LÓGICA DE NEGOCIO

El primer paso para la creación de los endpoints fue hacer *mocks* de los mismos. Un *mock* (imitación) es básicamente una simulación de algo. En este caso, lo que se pretendía era crear inicialmente los endpoints de tal manera que, tengan la entrada que tengan, siempre devuelvan una respuesta HTTP favorable, ya sea 200 (OK) o 201 (Creado).

1. Los endpoints se definieron mediante la creación de clases del tipo **controlador** (*controller*) en la capa de aplicación, y se definió una clase por cada método REST. En cada una de estas clases se definieron los métodos para cada endpoint.

2. Cada uno de estos métodos reciben parámetros de entrada y envían una **solicitud** (*request*) con los mismos vía **MediatR** (una librería que implementa el patrón **Mediador**, el cual se utiliza para mediar entre capas [44]) a la capa de aplicación, donde la misma es procesada por un **manejador** (*handler*).
3. Originalmente, como se pretendía hacer mocks, los manejadores se escribieron de tal manera que devolvieran 200 o 201 como código de respuesta, cualquiera fuesen sus entradas (lo que se denomina *happy path*, ruta feliz).
4. Se implementaron clases del tipo **respuesta** (*response*), que devuelven una respuesta estandarizada en formato JSON.
5. Luego se hicieron pruebas manuales de los mismos, para asegurarse de que funcione todo lo mencionado anteriormente.
6. Pasada la etapa anterior, se continuó con el añadido de validaciones a los parámetros de los endpoints.

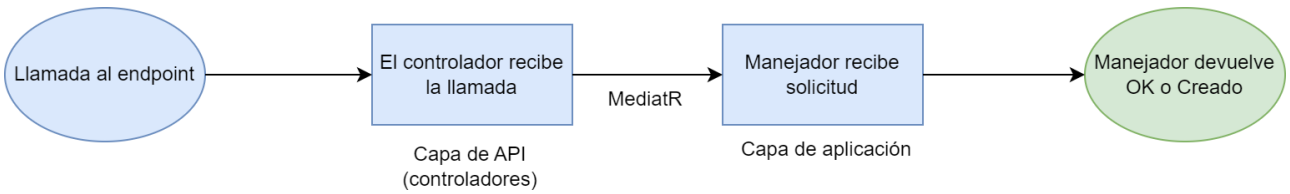


Figura 30: Flujo de llamada a un endpoint, happy path

Implementación de validadores:

1. Se hizo mediante la librería **FluentValidation**, la cual sirve para implementar validaciones de forma estandarizada.
2. La librería permite diversas reglas sobre los parámetros: por ejemplo, si es un número que esté en cierto rango, si es una cadena de texto que cumpla con una expresión regular (básicamente, que siga un patrón definido).
3. Permite también la especificación de errores, en caso de que cierta validación no se cumpla.
4. Se configuraron clases que funcionan como catálogos de errores.

5. Finalmente, se hicieron pruebas manuales de los validadores.

Finalizada la etapa anterior, se procedió a construir la lógica de negocio para los endpoints:

1. El primer paso fue crear clases del tipo repositorio en la capa de infraestructura, con los métodos de llamadas a bases de datos, para cada endpoint.
2. Se hizo una separación entre las operaciones de lectura y las de escritura (lo que se llama *Command and Query Responsibility Segregation*, Segregación de responsabilidades de comandos y consultas) ya que, como se explicó anteriormente, se utilizan distintas librerías según el tipo de consulta.
3. Se crearon interfaces en la capa de aplicación correspondientes a estos repositorios y se modificaron los manejadores para que las utilicen.

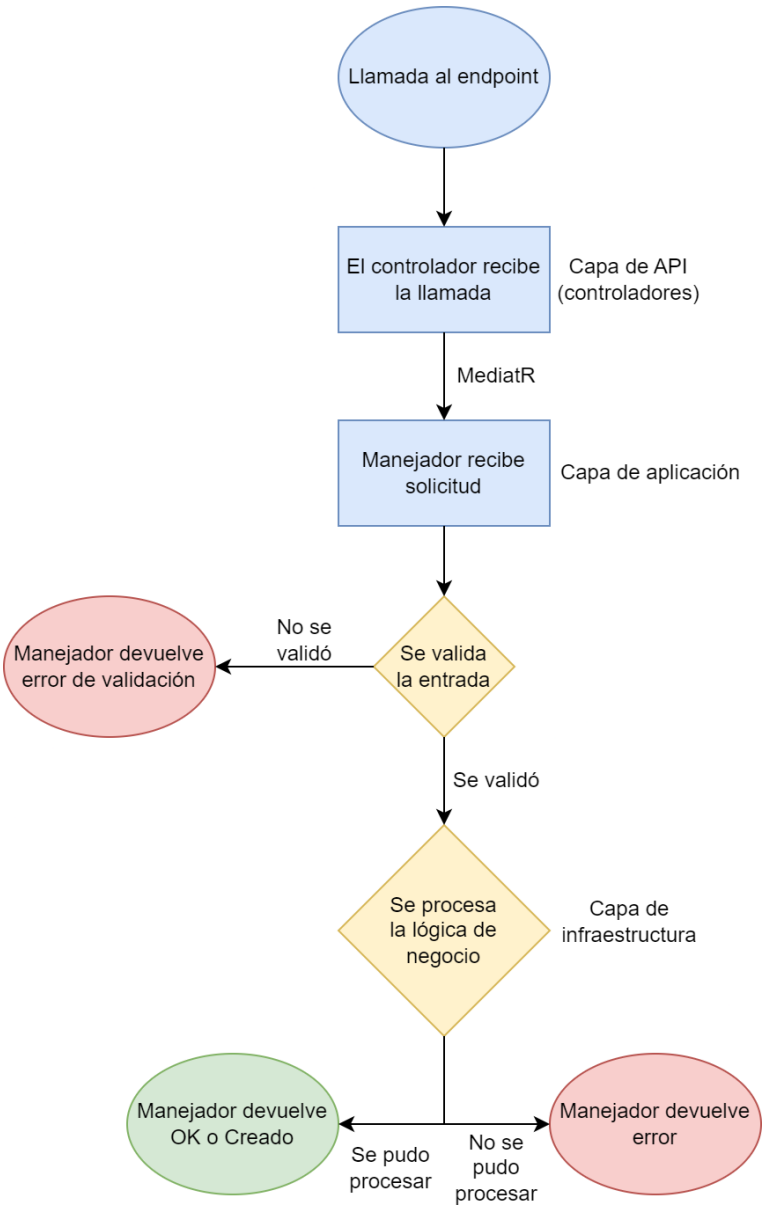


Figura 31: Flujo de llamada a un endpoint, sin happy path

Integración con Kafka

Para integrar con Kafka se utilizaron las librerías que provee Platform. Las mismas están basadas en **Confluent**, una librería para el manejo de eventos de Kafka.

Consumo de eventos:

1. El consumo de eventos se basa en suscriptores. Los suscriptores son clases que implementan una interface llamada **ISubscriber**, ya definida en las librerías de Platform. En C#, una interface es una entidad que contiene métodos definidos pero no sus implementaciones.
2. Los suscriptores tienen un método llamado **ConsumerAsync** (consumidor asíncrono), el cual consume un evento definido proveniente de Kafka.
3. Se envía mediante **MediatR** una solicitud a la capa de aplicación con la información del evento, para procesamiento de este.
4. Finalmente, el evento se continúa procesando según la lógica de negocio definida para el mismo.

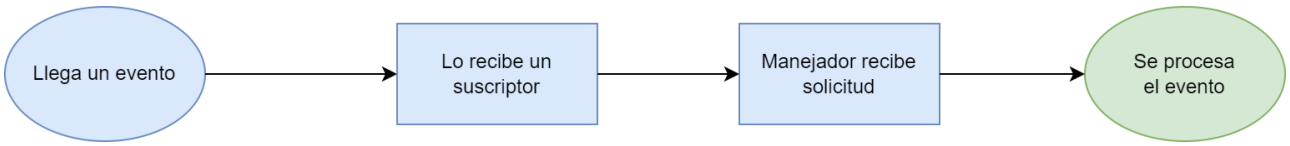


Figura 32: Flujo de consumo de un evento en Kafka

La publicación de eventos se basa en publicadores. Platform provee una interface llamada **IPublisher**, la cual se inyecta en la clase que necesita publicar un evento y como parámetro admite un objeto del tipo evento.

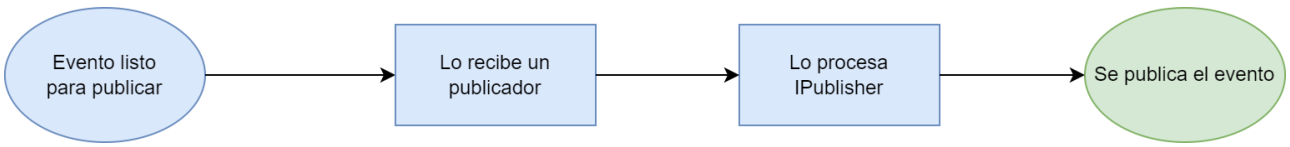


Figura 33: Flujo de publicación de un evento en Kafka

Por último, para que Kafka funcione, todos los publicadores y suscriptores se configuraron como servicios de la aplicación. Además, se agregó la configuración correcta en los archivos de configuración.

CONEXIONES A BASES DE DATOS

Para realizar las conexiones a bases de datos se utilizaron librerías a tal fin.

Conexión a SQL Server:

- En el microservicio Templates se implementó el estándar CQRS para separar las actividades de escritura en la base de datos (creación, actualización y borrado) de las de lectura.
- Ídem microservicio Builder.
- El uso de CQRS permite la posibilidad de tener 2 bases de datos (una de solo lectura y otra de escritura), ya sea de forma física o lógica. En este caso, la base de datos es la misma, pero aun así el uso del patrón tiene una justificación: para cada proceso se utilizan 2 librerías diferentes: Entity Framework Core para escritura y Dapper para lectura, y esto a su vez se justifica porque Dapper es mucho más rápido que EF Core para estas operaciones.

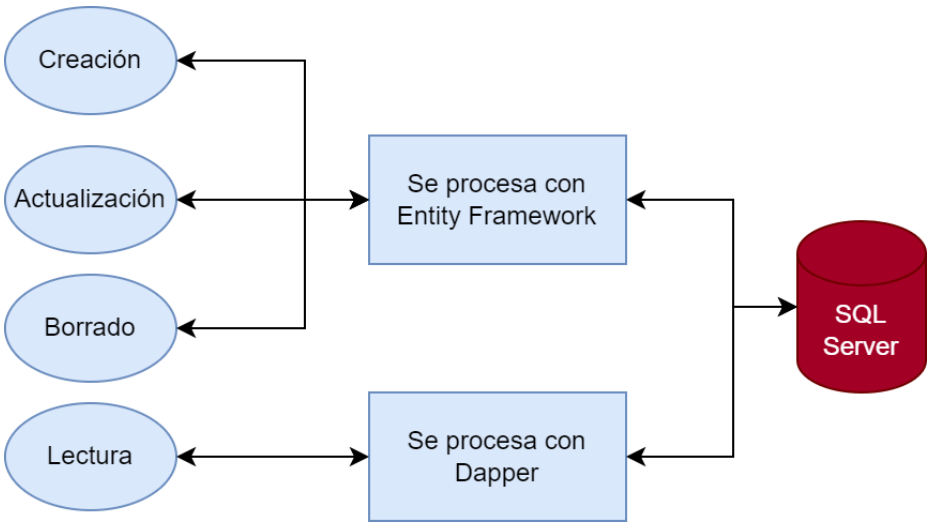


Figura 34: Flujo de conexión a SQL Server

Conexión a MongoDB:

- Para el microservicio Collector, se utilizó también una librería ya provista, basada en MongoDB Driver, que es la librería oficial de MongoDB para .NET.
- Se crearon 2 interfaces, con sus implementaciones correspondientes, para operar con la base de datos. Una con métodos para lectura y la otra para operaciones de escritura, para respetar también el patrón CQRS.

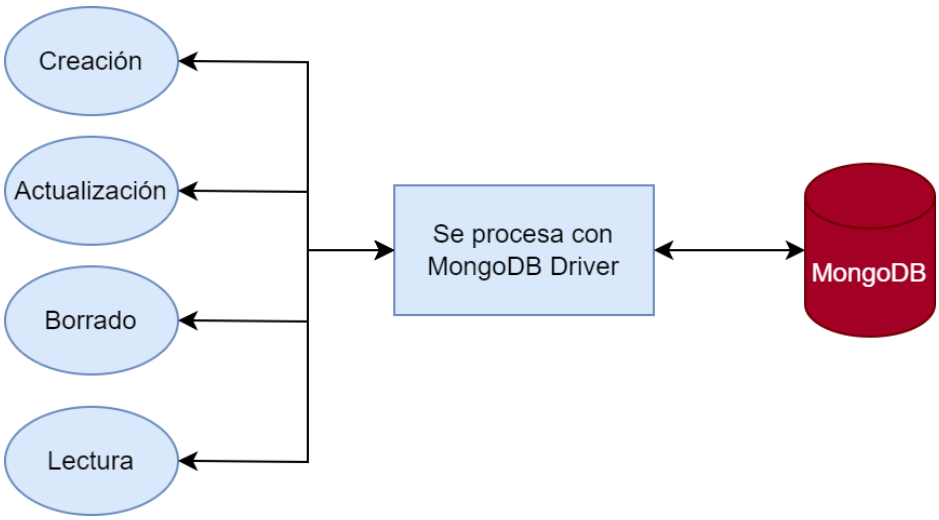


Figura 35: Flujo de conexión a MongoDB

IMPLEMENTACIÓN PARA LLAMADAS A ENDPOINTS EXTERNOS

Para realizar las llamadas a APIs externas e incluso entre microservicios (por ejemplo, cuando Collector cuando llama a un endpoint de Templates), se implementaron las librerías Refit y Polly.

Refit es una biblioteca REST automática, con seguridad de tipos para .NET, la cual convierte una API REST en una interface. Características:

- Reduce la cantidad de código necesario para llamar a una API, ya que automatiza la construcción de llamadas HTTP y sus respuestas.
- Proporciona serialización y deserialización automática de datos [45].

Polly es una biblioteca para .NET que ayuda a gestionar fallos y mejorar la resiliencia de aplicaciones. Puede definir y aplicar fácilmente diversas estrategias para gestionar fallos y ralentizaciones de forma fluida y segura. Permite utilizar y combinar diferentes estrategias de resiliencia para hacer frente a diversos escenarios, como:

- *Retry* (Reintentar): intentar de nuevo si algo falla (ante un problema temporal).

- *Circuit Breaker* (Interruptor de circuito): dejar de intentarlo si algo está roto o está ocupado.
- *Timeout* (Tiempo de espera): detenerse si algo tarda demasiado.
- *Rate Limiter* (Limitador de tasa): limitar la cantidad de solicitudes que realiza o acepta, permitiendo un control de carga.
- Respaldo: hacer otra tarea si algo falla.
- Cobertura: hacer varias tareas al mismo tiempo y elegir la más rápida [46].

Implementación de Refit:

Se crearon interfaces con los métodos y parámetros necesarios. También se especificaron para cada método:

- Los métodos HTTP
- Las rutas
- Parámetros adicionales que fuesen necesarios, como las cabeceras y la autorización

Adicionalmente se configuró la URL base de los endpoints.

Polly se implementó aplicando **Facade** (Fachada), el cual es un patrón de diseño que provee una interfaz simplificada a una librería o marco de desarrollo, o cualquier otro conjunto complejo de clases [47]:

- Se creó una interface con métodos (la fachada) y su respectiva implementación.
- En la misma, cada llamado a un endpoint se hace mediante Refit.
- Cada método aplica una clase llamada **ResiliencePipeline** (Tubería de resiliencia), propia de Polly, la cual maneja los errores automáticamente al hacer las llamadas.

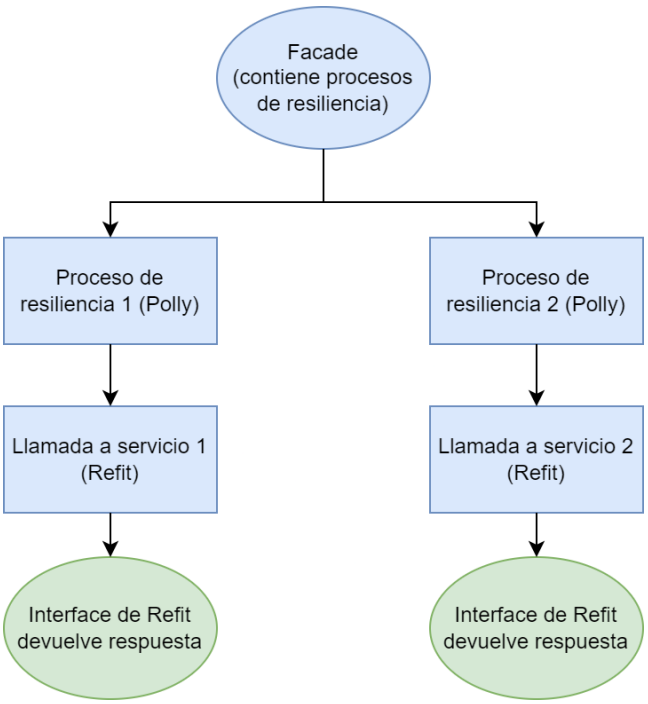


Figura 36: Implementación de Polly con Refit

Para la conexión con Azure Storage se utilizaron las librerías propias de Microsoft, con una diferencia considerable respecto al legado: se implementaron librerías actualizadas. Las mismas permiten crear una clase en donde se pueden implementar métodos para el manejo de archivos y el contenedor en general.

Se respetaron los métodos originales observados en el sistema legado:

- **Descargar**: para descarga de archivos
- **Eliminar**: para eliminación de archivos
- **InicializarContainer**: para trabajar con un contenedor
- **ObtenerUrl**: obtener la URL para descarga de un archivo
- Entre otros

Integración con el agente Elastic APM

La integración con Elastic APM se hizo simplemente agregando una línea en los archivos de configuración. En los archivos se especifica que la observabilidad para el servicio está activada, así como también:

- Nombre del servicio
- Entorno
- Cualquier otro parámetro necesario

Con esto, el servicio tiene conexión y registra los eventos, los cuales después se pueden visualizar en la UI de Kibana.

Lógica de negocio de los eventos

Teniendo en cuenta que fue un proceso migratorio que en principio no debía alterar la funcionalidad final del sistema, la lógica de negocio se mantuvo fiel a la original. No era objetivo de esta migración alterar la salida del sistema.

Aun así, se implementó la misma de la siguiente manera:

1. En primer lugar, el proceso inicia en el Collector con la recepción de parte de un evento, que ocurre gracias a un **consumidor** (*consumer*) de Kafka en la capa de infraestructura.
2. Después, de la misma manera que ocurre con los endpoints, se genera una solicitud que se le pasa a un manejador utilizando MediatR.
3. Finalmente, cuando el evento llega al manejador, se ejecutan todas las operaciones correspondientes a la generación de la constancia para el evento.

Como se explicó en la descripción de los microservicios, en el caso del Collector cada evento recibido contiene información que se utiliza para crear una constancia, sumado al

enriquecimiento de las diversas APIs. Además, se crea un registro en **CE_Collector**, para hacer seguimiento del estado de cada evento procesado. Finalmente, ya sea que el evento se procesa correctamente o no, se genera un evento que se publica en Kafka.

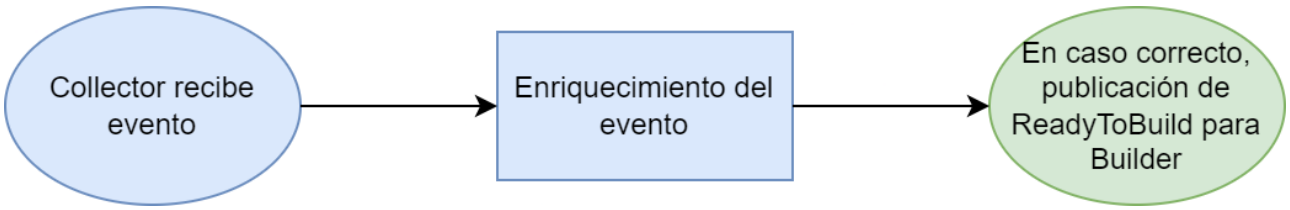


Figura 37: Flujo de un evento en el Collector

El proceso es similar para el Builder: recibe el evento **ReadyToBuild** y en base al mismo genera la constancia, publicando **GeneradaOk** (o **GeneradaNotOk** en caso de error) y escribiendo el registro en **CE_Collector**.

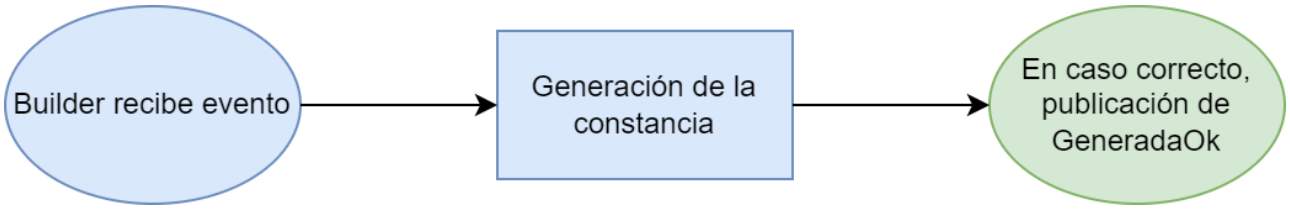


Figura 38: Flujo de un evento en el Builder

PRUEBAS UNITARIAS

Las pruebas unitarias (o *tests* unitarios) son un tipo de pruebas automatizadas. Las mismas prueban partes muy pequeñas de la aplicación, aislándolas, comparando su comportamiento real con el esperado.

Motivos para realizarlas:

- Las pruebas unitarias demuestran que la lógica del código es apropiada y que funcionará en todos los casos.
- Aumentan la legibilidad del código y ayudan a los desarrolladores a entender el código base, lo que facilita hacer cambios más rápidamente.
- Bien realizadas sirven como documentación del proyecto.

- Se ejecutan en pocos milisegundos, por lo que se pueden realizar cientos de ellas en muy poco tiempo.
- Permiten al desarrollador refactorizar el código más adelante y tener la garantía de que el módulo sigue funcionando correctamente. Para ello se escriben casos de prueba para todas las funciones y métodos, para que cada vez que un cambio provoque un error sea posible identificarlo y repararlo rápidamente.
- La calidad final del código mejorará ya que, al estar realizando pruebas de manera continua, al finalizar el código será limpio y de calidad.
- Como las pruebas unitarias dividen el código en pequeños fragmentos, se pueden probar distintas partes del proyecto sin tener que esperar a completar otras.

El patrón de diseño recomendado para una prueba unitaria se denomina **AAA**:

1. *Arrange* (organizar): en esta parte se definen los requisitos que debe cumplir el código.
2. *Act* (actuar): es momento de ejecutar la prueba. Esto genera los resultados que se deben analizar.
3. *Assert* (afirmar): es el momento de comprobar si los resultados obtenidos son los que se esperaban. En caso negativo, se corrige el error.

Características de una buena prueba unitaria:

- Rápida: las pruebas unitarias deberían tomar muy poco tiempo para ejecutarse.
- Aislada: las pruebas unitarias son independientes, se pueden ejecutar de forma aislada y no dependen de ningún factor externo, como una base de datos.
- Repetible: la ejecución de una prueba unitaria siempre devuelve el mismo resultado, si no cambia nada entre ejecuciones.
- Autocomprobación: la prueba debería poder detectar automáticamente si pasó o falló.

- Oportuna: una prueba unitaria no debería tardar un tiempo desproporcionadamente largo en escribirse en comparación con el código que se está probando [48].

En el caso de este proyecto, se escribieron pruebas unitarias para cubrir la mayor cantidad posible de la lógica de este. Puntualmente se puso a prueba la lógica de los endpoints y de los eventos.

Se utilizaron las librerías **xUnit** y **Moq**, que son librerías para trabajar con pruebas unitarias en .NET:

- **xUnit**: para implementar las mismas
- **Moq**: para hacer mocks de los objetos del sistema

Pruebas para los endpoints:

- Se pusieron a prueba tanto los manejadores como los validadores.
- Se escribieron pruebas según las diversas respuestas que devolvían los mismos, ya fuesen correctas o incorrectas.

En cuando a la lógica de negocio se escribieron pruebas que validen los métodos contenidos en las clases relacionados a la misma, ya que estos devuelven diversas respuestas según la entrada. Ejemplo: para un publicador de Kafka se espera una respuesta según la entrada. Si se pretende que esta devuelva un cierto resultado, ya sea válido o no, entonces se configura la prueba para eso. Lo mismo si la entrada no es válida, se espera que arroje error.

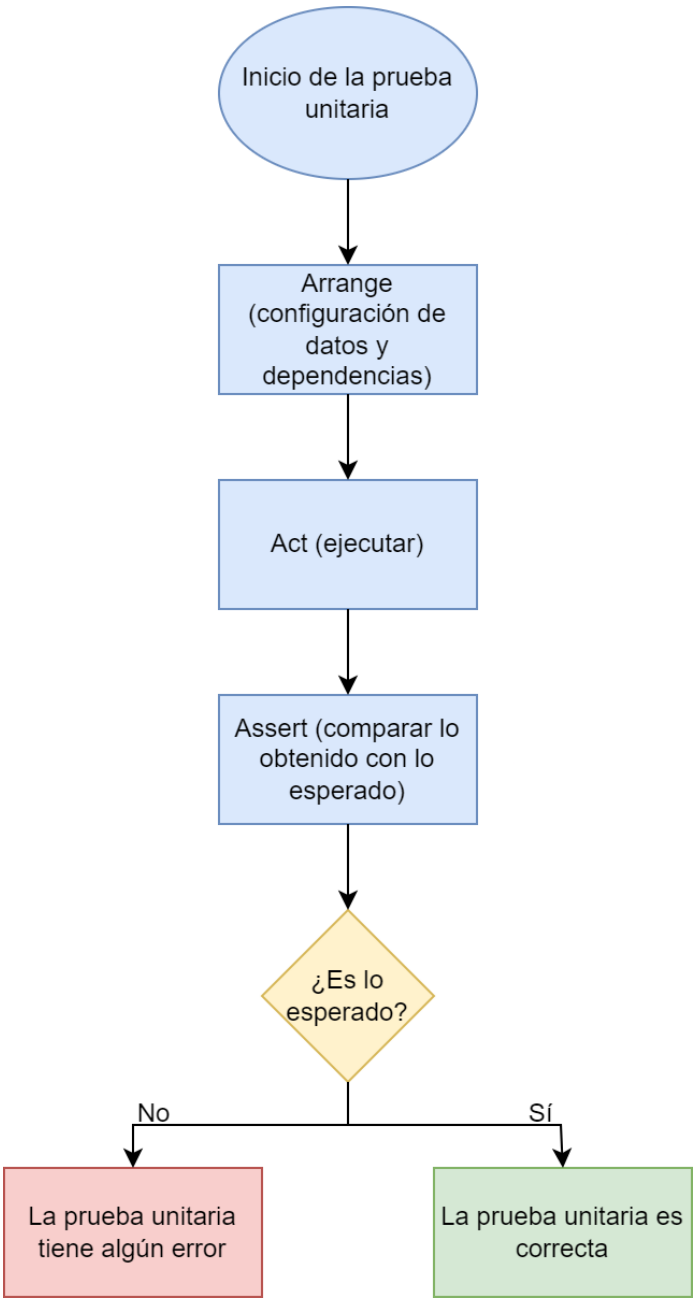


Figura 39: Diagrama de flujo de una prueba unitaria

SEGURIZACIÓN DEL SISTEMA

Como parte de la necesidad de cumplir con los estándares de seguridad requeridos por la empresa, se aplicaron medidas de seguridad informática sobre el código:

- Autenticación y autorización de endpoints:

Casi todos los endpoints necesitan un *token* para ser accedidos. Un token es una cadena de texto que se otorga si se realiza una operación exitosa de autenticación (por ejemplo, suministrando usuario y contraseña correctos). El mismo se obtiene mediante la llamada a una API **Azure Active Directory B2C** (es un servicio que tiene la empresa para autenticación de usuarios, a la cual se le suministra un identificador de cliente). Con el token correcto se obtiene acceso a los endpoints [49].

- Parametrización de las consultas SQL:

Se aseguró que todas las consultas SQL que se manejan mediante Dapper estén parametrizadas con *placeholders* (parámetros precedidos por una arroba), para evitar ataques del tipo inyección de SQL. Una inyección de SQL ocurre cuando se inserta código malicioso a través de alguna entrada (en este caso, un endpoint) que permita hacer una consulta a la base de datos.

ARCHIVOS DE CONFIGURACIÓN

Se configuraron los diversos parámetros requeridos por cada uno de los microservicios en archivos de configuración. Según lo requerido por los estándares de **DevOps** (*Development Operations*, Operaciones de desarrollo) de la empresa, cada ambiente de despliegue en OpenShift requiere un archivo de configuración para el mismo, donde se especifican diversos parámetros del sistema (conexiones a bases de datos, conexión a Kafka, etc). El formato requerido es **YAML** (acrónimo recursivo de YAML *Ain't Markup Language*, YAML no es un lenguaje de marcado), el cual permite guardar datos de manera jerárquica (en base a la indentación) [50].

Para la ejecución local esto no era requerido, pero aun así se requiere tener una configuración mínima, ya que de lo contrario la ejecución falla. También se utilizaron archivos YAML.

```
food:
  - vegetables: tomatoes #first list item
  - fruits: #second list item
      citrics: oranges
      tropical: bananas
      nuts: peanuts
      sweets: raisins
```

Figura 40: Ejemplo de archivo YAML[9]

CORRECCIONES DE ERRORES Y REFACTORIZACIÓN DEL CÓDIGO

Durante el desarrollo del código se hicieron correcciones sobre el mismo, tanto para solucionar errores como para su refactorización. En esta sección se explican las correcciones hechas desde la parte del desarrollo. Hubo colaboración con los equipos de QA y de frontend, los cuales informaron correcciones para hacer, pero esto se explica más adelante.

Pruebas iniciales del desarrollo:

1. Lo primero que se hizo fue probar localmente el código para observar su funcionamiento.
2. Se verificó el funcionamiento de los endpoints y de la lógica de negocio de los eventos.

Segunda fase, despliegue en ambiente de prueba:

1. Se hizo despliegue de los microservicios en OpenShift de prueba, para que estén en ejecución constante.
2. Se configuraron los mismos con parámetros correspondientes al ambiente de prueba (bases de datos, contenedores de Azure, eventos de Kafka, APIs).

[9] Imagen tomada de [50]

Después del despliegue de prueba, se evidenció algo importante: no venían suficientes eventos de Kafka (a comparación de lo que ocurría en el ambiente de producción) ni tampoco las APIs proveían suficiente información para el enriquecimiento de eventos. Se probó, en primera instancia, una posible solución: la inyección manual de eventos. Pero debido a que el nuevo sistema podía correr en paralelo con el original, se optó (con autorización del negocio) por desplegar los servicios en el ambiente de producción y consumir los eventos de este, así como también la información de las APIs. Esto permitió observar el funcionamiento completo del mismo.

Por otra parte, se hicieron refactorizaciones de código y añadido de pruebas unitarias según las indicaciones de SonarQube, para cumplir el requisito de cobertura de código y también lograr un código más legible.

COLABORACIÓN CON EL EQUIPO DE QA

Las tareas del equipo de QA o *Quality Assurance* (Aseguramiento de la calidad) consistieron en hacer pruebas integrales del sistema, del tipo automatizadas y manuales:

- Para probar el funcionamiento de los endpoints se le proveyó diversos casos de entrada y de salida.
- Para evaluar la lógica de los eventos comparó el resultado final de la generación de una constancia del nuevo sistema con la constancia generada por el sistema original. Se le proveyó un listado de constancias a comparar, correspondientes a los diversos tipos de constancias.

En ambos casos, a medida que el equipo reportaba correcciones a realizar, las mismas se agregaban como tareas y se las llevaba a cabo.

La colaboración con este equipo fue necesaria y valiosa, ya que se le delegó la tarea de realizar pruebas sobre el sistema, para validar que su funcionamiento fuese consistente con lo especificado.

COLABORACIÓN CON EL EQUIPO DE FRONTEND

El equipo de frontend tuvo la tarea de desarrollar, utilizando React (una biblioteca para desarrollo de frontend, también requerida dentro de la empresa), la nueva web de gestión. Se debía respetar la funcionalidad de la web original. Desde el lado del desarrollo de backend se colaboró con este equipo, ya que era necesario suministrar endpoints para el frontend, que funcionaran correctamente, tanto en cuanto a lógica de negocio como en lo que respecta a la devolución de mensajes de estado.

Se le especificó al equipo cómo se debían procesar los endpoints tanto de lectura como de escritura de datos. Por otro lado, el mismo reportó errores a medida que los iba encontrando, los cuales, al igual que el caso anterior, se cargaban como tareas de corrección.

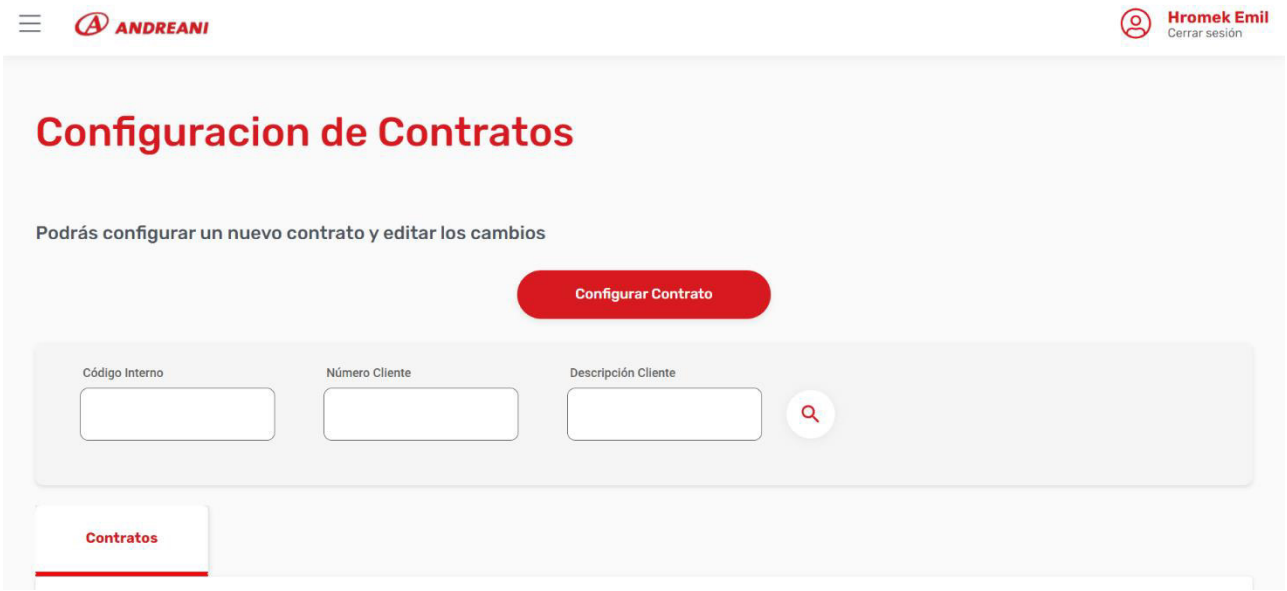


Figura 41: Nueva interfaz web de configuración

DESPLIEGUE EN PRODUCCIÓN

Como se mencionó en la sección que habla sobre las correcciones durante el desarrollo, la puesta en producción se hizo mientras se desarrollaba el código. Ya que el nuevo

sistema no entraba en conflicto con el original, y como se consideró necesario consumir eventos e información de APIs de producción, entonces se tuvo la autorización para subir los servicios a producción y observar su funcionamiento.

En base a esto, se pudieron hacer todas las correcciones necesarias en tiempo real, contando con la colaboración de los equipos de QA y de frontend.

CARACTERÍSTICAS E IMPACTO DEL NUEVO SISTEMA

Al margen de la explicación del proceso de desarrollo, se presenta un panorama general de las características del nuevo sistema, que lo diferencian del anterior.

ARQUITECTURA DEL SISTEMA Y SUS SERVICIOS

Con una arquitectura de microservicios no sólo se está a la altura de los estándares requeridos por el negocio, sino también a los estándares habitualmente aplicados hoy en día en el ámbito del desarrollo backend. Como se había mencionado, esta arquitectura permite un gran grado de escalabilidad del sistema y un mejor aprovechamiento de los recursos del sistema. Por otra parte, cada microservicio fue diseñado teniendo en cuenta buenas prácticas de arquitectura para aplicaciones y escritura de código, lo cual maximiza la calidad de los desarrollos.

RENDIMIENTO Y EFICIENCIA

Más allá de que gracias a la implementación de una arquitectura de microservicios se espera un mayor rendimiento y eficiencia del sistema, hay más características que coadyuvan a este propósito y son:

- Despliegue en OpenShift: se posee un sistema que corre en una infraestructura mucho más rápida que la de Windows Server. No solo se pueden asignar más recursos, sino que también es mucho más viable hacer el escalamiento de los servicios, tanto de manera horizontal como vertical.
- Escalamiento automático en OpenShift: el número de contenedores de cada microservicio puede variar automáticamente, lo cual puede ser útil en momentos de demanda alta.

- Uso de Kafka: permite manejar un mayor volumen de datos con menor latencia, permitiendo un procesamiento más rápido de los eventos. La mejora en la eficiencia no es solo operativa sino también económica, ya que esta tecnología es mucho menos costosa que MQ, en términos de licencia.
- Eliminación de código en desuso: sin código fuente que no tenga utilidad, hay una menor carga de recursos sobre el sistema.

De la misma manera que se hizo un cálculo del tiempo que se tarda para generar una constancia en el sistema original, también se lo hizo para el nuevo. Se hizo el promedio de diferencias de tiempo entre el momento en el que se generó el evento en la vida real y el momento en el que el servicio Collector recibió el evento **GeneradaOk**:

Fecha	**Rango horario**	**Promedio de diferencia (minutos)**
7/10/2024	00:00 a 05:59:59	0,64
7/10/2024	06:00 a 11:59:59	3,12
7/10/2024	12:00 a 17:59:59	35,11
7/10/2024	18:00 a 23:59:59	21,68
8/10/2024	00:00 a 05:59:59	2,75
8/10/2024	06:00 a 11:59:59	1,44
8/10/2024	12:00 a 17:59:59	34,11
8/10/2024	18:00 a 23:59:59	63,7
9/10/2024	00:00 a 05:59:59	2,73
9/10/2024	06:00 a 11:59:59	1,22
9/10/2024	12:00 a 17:59:59	97,23
9/10/2024	18:00 a 23:59:59	193,26
10/10/2024	00:00 a 05:59:59	17,61
10/10/2024	06:00 a 11:59:59	0,92
10/10/2024	12:00 a 17:59:59	20,54
10/10/2024	18:00 a 23:59:59	12,36

Fecha	Rango horario	Promedio de diferencia (minutos)
11/10/2024	00:00 a 05:59:59	1,14
11/10/2024	06:00 a 11:59:59	0,69
11/10/2024	12:00 a 17:59:59	1,95
11/10/2024	18:00 a 23:59:59	1,26
12/10/2024	00:00 a 05:59:59	1,65
12/10/2024	06:00 a 11:59:59	0,64
12/10/2024	12:00 a 17:59:59	1,4
12/10/2024	18:00 a 23:59:59	0,64
13/10/2024	00:00 a 05:59:59	18,87
13/10/2024	06:00 a 11:59:59	20,31
13/10/2024	12:00 a 17:59:59	1,04
13/10/2024	18:00 a 23:59:59	0,66
14/10/2024	00:00 a 05:59:59	0,7
14/10/2024	06:00 a 11:59:59	1,16
14/10/2024	12:00 a 17:59:59	42,66
14/10/2024	18:00 a 23:59:59	66,65
15/10/2024	00:00 a 05:59:59	1,13
15/10/2024	06:00 a 11:59:59	0,74
15/10/2024	12:00 a 17:59:59	29,16
15/10/2024	18:00 a 23:59:59	35,75
16/10/2024	00:00 a 05:59:59	4,99
16/10/2024	06:00 a 11:59:59	8,48
16/10/2024	12:00 a 17:59:59	208,27
16/10/2024	18:00 a 23:59:59	235,6
17/10/2024	00:00 a 05:59:59	29,52
17/10/2024	06:00 a 11:59:59	5,06
17/10/2024	12:00 a 17:59:59	136,23
17/10/2024	18:00 a 23:59:59	153,15

Fecha	Rango horario	Promedio de diferencia (minutos)
18/10/2024	00:00 a 05:59:59	9,43
18/10/2024	06:00 a 11:59:59	5,22
18/10/2024	12:00 a 17:59:59	80,57
18/10/2024	18:00 a 23:59:59	95,83
19/10/2024	00:00 a 05:59:59	2,69
19/10/2024	06:00 a 11:59:59	0,69
19/10/2024	12:00 a 17:59:59	1,78
19/10/2024	18:00 a 23:59:59	0,74
20/10/2024	00:00 a 05:59:59	2,98
20/10/2024	06:00 a 11:59:59	26,79
20/10/2024	12:00 a 17:59:59	0,66
20/10/2024	18:00 a 23:59:59	0,68
21/10/2024	00:00 a 05:59:59	0,64
21/10/2024	06:00 a 11:59:59	4
21/10/2024	12:00 a 17:59:59	71,35
21/10/2024	18:00 a 23:59:59	80,06

Tabla 3: Tiempos de generación de una constancia en el sistema nuevo

Como se observa, en comparación a la tabla 2, los tiempos de generación son aproximadamente iguales o incluso mejores que los del sistema original.

MANEJO Y MONITOREO DE ERRORES

El nuevo sistema posee características que sirven para manejar las ocurrencias de errores, así como también monitorear el rendimiento:

- Registro de eventos en MongoDB: cada vez que se procesa un evento, se crea un registro en **CE_Collector**. Al principio, este registro para un cierto número de envío se marca como incompleto. Si para dicho envío se genera un mensaje del tipo **NotReadyToBuild** o **GeneradaNotOk**, se marcará como **Failed**. Esto permite saber

qué eventos fallaron y después decidir qué hacer con los mismos (ya sea un reprocesamiento u otra opción).

- Implementación de Refit y Polly: el uso de estas librerías permite manejar de manera tanto configurable como automatizada los fallos al realizar una llamada HTTP.
- Visualización de métricas en los tableros de Kibana: permite hacer un seguimiento de los errores de sistema, incluyendo la cantidad y el tipo, facilitando la toma de decisiones.

TRABAJO A FUTURO

Más allá de que el objetivo principal del trabajo era la migración de tecnologías sin alterar la lógica de funcionamiento del sistema, eso no significa que no se puedan agregar nuevas características, así como también mejoras o incluso algo que quedó pendiente y no se lo desarrolló por no considerarlo prioritario para la puesta final en producción.

Como deudas técnicas, están pendientes:

- La creación de un microservicio para hacer las republicaciones masivas. La idea para el mismo es que provea un endpoint que tome como parámetro un listado de envíos y que se encargue de llamar al endpoint de republicación del Collector (una llamada para cada envío), en un horario conveniente, mediante la utilización de alguna librería para ejecutar tareas en un horario específico (preferentemente a la madrugada, como en el sistema original).
- El armado de los tableros para visualización en Kibana.

En cuando a mejoras propuestas, se pensó en la posibilidad de reprocesar automáticamente un envío en caso de que falle. Como se dijo anteriormente, cada vez que se genera un error en el procesamiento de un envío, ya sea en el Collector (generando un mensaje **NotReadyToBuild**) o en el Builder (**GeneradaNotOk**), el mismo es consumido por el Collector y simplemente se marca como **Failed** en **CE_Collector**. Pero se puede ir un paso más allá y hacer que los manejadores para esos eventos no solo hagan el registro, sino que además inicien la republicación del envío. Esto sería muy útil para evitar reprocesamientos manuales.

Es necesario también mantener actualizado el sistema, principalmente en lo que respecta a la actualización de librerías. Periódicamente se deberá revisar el código fuente y asegurarse que se estén utilizando las últimas versiones de las librerías, así como realizar cualquier otra actualización de código que se considere necesaria.

Por último, en lo que respecta a nuevas características, las mismas serán definidas por el negocio a futuro. No es tarea del desarrollador definirlas, sí se pueden proponer mejoras, pero es importante atenerse a lo que requiera el negocio.

CONCLUSIONES

Se pudo completar el objetivo migratorio, cumpliendo punto por punto los requisitos solicitados por la gerencia tecnológica de la empresa.

Principalmente se pudo implementar el consumo y producción de eventos utilizando Kafka, para contribuir al objetivo transversal del negocio del apagado de MQ, el cual se requiere por un tema de costo económico. También se pudo implementar el reemplazo de tecnologías antiguas, como .NET Framework, que tiene varios años y era conveniente migrar a .NET 8, por todas las ventajas de utilizar un marco de desarrollo más moderno.

También se pudieron incorporar diversas tecnologías que no estaban presentes, como MongoDB, Kibana y SonarQube, que suman ventajas y eran también requeridas.

Se pudo reemplazar la arquitectura monolítica por una de microservicios, así como también aplicar buenas prácticas de desarrollo en el código de estos. Esto permite un mayor escalamiento del sistema, así como también una mejor mantenibilidad y legibilidad del código.

Para finalizar, y a partir de acá me permito escribir en primera persona, estoy mucho más que conforme con el trabajo realizado. El mismo contribuyó plenamente a mi desarrollo profesional, ya que representó un desafío constante, en donde tuve que no sólo aplicar lo aprendido a lo largo de la carrera, sino que también tuve que aprender a lo largo del transcurso de este, empezando por el entendimiento inicial del sistema (no solo al leer la documentación, sino también al examinar el código), para luego ir adquiriendo diversos conocimientos sobre la marcha e ingeniarme sobre cómo avanzar con las tareas de desarrollo. Como dice un famoso proverbio: «Ningún mar en calma hizo experto a un marinero».

Por otra parte, es motivante y grato saber que el trabajo realizado es inmediatamente aprovechado por el negocio, debido a su valor. Gracias a esto, uno siente que su trabajo es realmente útil.

Y, por último, también es grato saber que lo que se hizo es un auténtico trabajo de ingeniería de software, un tema fundamental que se estudia en la carrera. En este caso, hablamos puntualmente de una reingeniería, al optimizar y mejorar el proceso de un sistema ya existente, sin alterar la salida ni la entrada de este.

ÍNDICE DE FIGURAS Y TABLAS

FIGURAS

TABLAS

BIBLIOGRAFÍA

[1] Amazon. «¿Cuál es la diferencia entre el front end y back end en el desarrollo de aplicaciones?» [En línea]. Disponible en: https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend [Último acceso: 2024]

[2] Mozilla. «HTML: HyperText Markup Language» [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/HTML [Último acceso: 2024]

[3] Adobe. «JPEG files» [En línea]. Disponible en: https://www.adobe.com/creativecloud/file-types/image/raster/jpeg-file.html [Último acceso: 2024]

[4] Adobe. «What does PDF mean?» [En línea]. Disponible en: https://www.adobe.com/acrobat/about-adobe-pdf.html [Último acceso: 2024]

[5] Fortinet. «Protocolo de transferencia de archivos (FTP): significado y definición» [En línea]. Disponible en: https://www.fortinet.com/lat/resources/cyberglossary/file-transfer-protocol-ftp-meaning [Último acceso: 2024]

[6] Equipo Custom de Andreani. «Documentación de Equipo Custom». [Última consulta: 2024]

[7] Google. «Crea apps increíbles con el conocimiento de Google del mundo real» [En línea]. Disponible en: https://developers.google.com/maps?hl=es-419 [Último acceso: 2024]

[8] Amazon. «¿Qué es una interfaz de programación de aplicaciones (API)?» [En línea]. Disponible en: https://aws.amazon.com/es/what-is/api/ [Último acceso: 2024]

[9] H. F. Nielsen. «Service Points - a Flexible Endpoint Architecture for the Net», 1999 [En línea]. Disponible en: https://www.w3.org/Protocols/Design/Endpoints [Último acceso: 2024]

[10] Mozilla. «HTTP» [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/HTTP [Último acceso: 2024]

[11] IBM. «Introduction to IBM MQ», 24 de octubre de 2024 [En línea]. Disponible en: https://www.ibm.com/docs/en/ibm-mq/9.4?topic=mq-introduction [Último acceso: 2024]

[12] SolarWinds. «What Is Windows Server?» [En línea]. Disponible en: https://www.solarwinds.com/resources/it-glossary/windows-server [Último acceso: 2024]

[13] Microsoft. «What is TypeScript?» [En línea]. Disponible en: https://www.typescriptlang.org/ [Último acceso: 2024]

[14] C. Deshpande. «What Is Angular? A Guide to the Angular Framework», 2024 [En línea]. Disponible en: https://www.simplilearn.com/tutorials/angular-tutorial/what-is-angular [Último acceso: 2024]

[15] Redis. «Introduction to Redis» [En línea]. Disponible en: https://redis.io/about/ [Último acceso: 2024]

[16] Microsoft. «What is .NET Framework?» [En línea]. Disponible en: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework [Último acceso: 2024]

[17] R. Awati, A. Hughes y C. Stedman. «Microsoft SQL Server», marzo de 2024 [En línea]. Disponible en: https://www.techtarget.com/searchdatamanagement/definition/SQL-Server [Último acceso: 2024]

[18] NHibernate. «NHibernate Preface» [En línea]. Disponible en: https://nhibernate.info/doc/nhibernate-reference/preface.html [Último acceso: 2024]

[19] IBM. «Programación orientada a objetos», 17 de agosto de 2021 [En línea]. Disponible en: https://www.ibm.com/docs/es/spss-modeler/saas?topic=language-object-oriented-programming [Último acceso: 2024]

[20] Apache Software Foundation. «Apache log4net™ Manual - Introduction» [En línea]. Disponible en: https://logging.apache.org/log4net/release/manual/introduction.html [Último acceso: 2024]

[21] Microsoft. «Introduction to Azure Blob Storage», 10 de octubre de 2023 [En línea]. Disponible en: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction [Último acceso: 2024]

[22] Salesforce Authors. «B2B vs B2C Ecommerce: What's the Difference?», 1 de noviembre de 2021 [En línea]. Disponible en: https://www.salesforce.com/eu/blog/b2b-vs-b2c-ecommerce-difference/ [Último acceso: 2024]

[23] Red Hat. «¿Qué es una API de REST?», 13 de julio de 2023 [En línea]. Disponible en: https://www.redhat.com/es/topics/api/what-is-a-rest-api [Último acceso: 2024]

[24] Scrum.org. «What is Scrum?» [En línea]. Disponible en: https://www.scrum.org/resources/what-scrum-module [Último acceso: 2024]

[25] Scrum.org. «Introduction to the Scrum Events» [En línea]. Disponible en: https://www.scrum.org/resources/introduction-scrum-events [Último acceso: 2024]

[26] Atlassian. «What is Jira?» [En línea]. Disponible en: https://www.atlassian.com/software/jira/guides/getting-started/introduction#what-is-jira-software [Último acceso: 2024]

[27] GitHub. «Acerca de GitHub y Git», 2024 [En línea]. Disponible en: https://docs.github.com/es/get-started/start-your-journey/about-github-and-git [Último acceso: 2024]

[28] Rootstack. «.Net core vs .Net framework: diferencias clave entre los dos», 13 de julio de 2023 [En línea]. Disponible en: https://rootstack.com/es/blog/net-core-vs-net-framework-diferencias-clave-entre-los-dos [Último acceso: 2024]

[29] Fortinet. «¿Qué es TCP?» [En línea]. Disponible en: https://www.fortinet.com/lat/resources/cyberglossary/tcp-ip [Último acceso: 2024]

[30] Apache Kafka. «Kafka Introduction» [En línea]. Disponible en: https://kafka.apache.org/intro [Último acceso: 2024]

[31] Equipo de Arquitectura de Andreani. «Openshift overview». [Última consulta: 2024]

[32] The Linux Foundation. «Kubernetes overview» [En línea]. Disponible en: https://kubernetes.io/docs/concepts/overview/ [Último acceso: 2024]

[33] MongoDB. «JSON and BSON» [En línea]. Disponible en: https://www.mongodb.com/resources/basics/json-and-bson [Último acceso: 2024]

[34] MongoDB. «¿Qué es MongoDB?» [En línea]. Disponible en: https://www.mongodb.com/es/company/what-is-mongodb [Último acceso: 2024]

[35] Microsoft. «Entity Framework Core», 24 de agosto de 2023 [En línea]. Disponible en: https://learn.microsoft.com/es-es/ef/core/ [Último acceso: 2024]

[36] Learn Dapper. «Welcome to Learn Dapper» [En línea]. Disponible en: https://www.learndapper.com/ [Último acceso: 2024]

[37] Equipo de Arquitectura de Andreani. «Observabilidad». [Última consulta: 2024]

[38] Equipo de Arquitectura de Andreani. «SonarQube». [Última consulta: 2024]

[39] Equipo de Arquitectura de Andreani. «SOLID». [Última consulta: 2024]

[40] Equipo de Arquitectura de Andreani. «Domain Driven Design». [Última consulta: 2024]

[41] Equipo de Arquitectura de Andreani. «Clean Architecture». [Última consulta: 2024]

[42] Codacy. «What Is Clean Code? A Guide to Principles and Best Practices», 19 de diciembre de 2023 [En línea]. Disponible en: https://blog.codacy.com/what-is-clean-code [Último acceso: 2024]

[43] Equipo de Arquitectura de Andreani. «Platform». [Última consulta: 2024]

[44] Refactoring.Guru. «Mediator» [En línea]. Disponible en: https://refactoring.guru/design-patterns/mediator [Último acceso: 2024]

[45] S. Kılıçarslan. «Using Refit in .NET», 1 de enero de 2024 [En línea]. Disponible en: https://medium.com/net-core/using-refit-in-net-0843bb199987 [Último acceso: 2024]

[46] .NET Foundation. «Meet Polly: The .NET resilience library» [En línea]. Disponible en: https://www.pollydocs.org/ [Último acceso: 2024]

[47] Refactoring.Guru. «Facade» [En línea]. Disponible en: https://refactoring.guru/design-patterns/facade [Último acceso: 2024]

[48] Equipo de Arquitectura de Andreani. «Unit Testing». [Última consulta: 2024]

[49] Microsoft. «¿Qué es Azure Active Directory B2C?», 3 de septiembre de 2023 [En línea]. Disponible en: https://learn.microsoft.com/es-es/azure/active-directory-b2c/overview [Último acceso: 2024]

[50] Red Hat. «YAML: qué es y usos», 3 de marzo de 2023 [En línea]. Disponible en: https://www.redhat.com/es/topics/automation/what-is-yaml [Último acceso: 2024]

[51] WebPicking. «Sucursal Andreani», 24 de agosto de 2017 [En línea]. Disponible en: https://webpicking.com/andreani-inaugura-tres-nuevas-sucursales/sucursal-andreani/ [Último acceso: 2024]

[52] P. Bareham. «MQ V8.0 Connection Authentication and MQ Explorer» [En línea]. Disponible en: https://www.ibm.com/support/pages/mq-v80-connection-authentication-and-mq-explorer [Último acceso: 2024]

[53] Microsoft. «Get started with Storage Explorer», 4 de abril de 2024 [En línea]. Disponible en: https://learn.microsoft.com/en-us/azure/storage/storage-explorer/vs-azure-tools-storage-manage-with-storage-explorer [Último acceso: 2024]

[54] Honduras Digital Challenge. «Metodología Scrum, una herramienta útil para agilizar tus proyectos» [En línea]. Disponible en: https://hondurasdigitalchallenge.com/2020/05/21/metodologia-scrum-una-herramienta-util-para-agilizar-tus-proyectos/ [Último acceso: 2024]

[55] Atlassian. «La colaboración en equipo y la entrega incremental comienzan en el tablero de Jira» [En línea]. Disponible en:

https://www.atlassian.com/es/software/jira/features/scrum-boards [Último acceso: 2024]

[56] GitHub. «Clasificar tu repositorio con temas» [En línea]. Disponible en https://docs.github.com/es/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/classifying-your-repository-with-topics [Último acceso: 2024]

[57] A.-M. Minda. «How to Use UI for Apache Kafka® with Instaclustr for Apache Kafka: Part 2», 16 de enero de 2024 [En línea]. Disponible en: https://www.instaclustr.com/blog/how-to-use-ui-for-apache-kafka-with-instaclustr-for-apache-kafka-part-2/ [Último acceso: 2024]

[58] OpenShift Design. «OpenShift Topology» [En línea]. Disponible en: https://openshift.github.io/openshift-origin-design/designs/developer/topology/ [Último acceso: 2024]