UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

Maestría en

SOFTWARE CON MENCIÓN EN DESARROLLO WEB Y MÓVIL Y DISEÑO DE ARQUITECTURA DE SISTEMAS

RPC-SO-34-NO.778-2021

Opción de Titulación:

Proyecto de titulación con componentes de investigación aplicada y/o de desarrollo

Tema:

Desarrollo de un prototipo de Microservicios con Clean Architecture, ASP.NET Core 8.0 y RabbitMQ

Autor(es)

Gisela Elizabeth Osorio Tibán

Luis Fernando Campos Sánchez

Director:

Daniel Giovanny Diaz Ortiz

Quito – Ecuador 2023

Autor(es):

Luis Fernando Campos Sánchez
Ingeniería en Software y Networking
Candidato a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
lcamposs@est.ups.edu.ec

Gisela Elizabeth Osorio Tibán
Ingeniería en Electrónica y Redes de Información
Candidata a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
gosoriot1@est.ups.edu.ec

Dirigido por:

Daniel Giovanny Diaz Ortiz
Ingeniería en Sistemas
Magister en Software Libre/ Magister en Redes de Computadoras
ddiaz@ups.edu.ec

Luis Fernando Campos Sánchez, Gisela Elizabeth Osorio Tibán
Desarrollo de un prototipo de Microservicios con Clean Architecture, ASP.NET Core 8.0 y RabbitMQ

DEDICATORIA

A mi familia.

Giss Osorio

A mi madre.

Luis Campos

AGRADECIMIENTO

Agradezco a Dios.

A mi familia.

A los profesores.

Al Ingeniero Daniel Diaz tutor de este Trabajo de Titulación.

Gis Osorio

Deseo expresar mi profundo agradecimiento a quienes me han brindado su apoyo y orientación a lo largo de mi Trabajo de Titulación. Su valiosa ayuda y conocimientos han sido esenciales en mi crecimiento académico.

Luis Campos

Tabla de Contenido

índice de figuras

índice de tablas

Desarrollo de un prototipo de Microservicios con Clean Architecture, ASP.NET Core 8.0 y RabbitMQ

Autor(es):

Gisela Elizabeth Osorio Tibán

Luis Fernando Campos Sánchez

Resumen

El presente trabajo de titulación se centra en el desarrollo de un prototipo de Microservicios utilizando Clean Architecture, ASP.NET Core 8.0 y RabbitMQ, destinado a la gestión eficiente de colas de tareas. El prototipo tiene como objetivo principal simplificar la administración de colas, superando los desafíos asociados con el uso convencional de RabbitMQ, que requiere la creación de consolas dedicadas para cada cola, lo que resulta en una gestión compleja y problemática, especialmente en entornos escalables.

El primer capítulo aborda la introducción al tema, estableciendo el contexto y la relevancia del proyecto. En el segundo capítulo, se determina el problema, donde se establecen los objetivos generales y específicos del proyecto. El tercer capítulo se enfoca en el marco teórico referencial, explorando el estado del arte y presentando conceptos clave con referencia a Microservicios, Arquitectura de Microservicios, RabbitMQ y Sistemas de Colas.

En el cuarto capítulo, se describen los materiales y metodología empleados, incluyendo el uso de Metodologías Ágiles, análisis de resultados previo a la aplicación de la metodología, requerimientos, tareas y spring backlog. Los resultados y la discusión se presentan en el quinto capítulo, incluyendo diagramas de arquitecturas como el C4 y el de flujo, dando una visión clara de la implementación del prototipo.

Finalmente, se presentan las conclusiones derivadas del proyecto en el sexto capítulo, destacando la importancia del desarrollo del prototipo para simplificar la administración de colas de tareas, así como las recomendaciones para futura investigaciones en este campo.

Palabras clave:

Microservicios, Clean Architecture, RabbitMQ, Escalabilidad

Abstract

This thesis project focuses on developing a Microservices prototype using Clean Architecture, ASP.NET Core 8.0, and RabbitMQ aimed at efficiently managing task queues. The prototype aims to simplify queue management, overcoming challenges associated with the conventional use of RabbitMQ, which requires creating dedicated consoles for each queue, resulting in complex and problematic management, especially in scalable environments.

The first chapter introduces the topic, establishing the project's context and relevance. In the second chapter, a detailed problem determination is conducted, outlining the project's general and specific objectives. The third chapter centers on the theoretical framework, exploring the state of the art and presenting key concepts related to Microservices, Microservices Architecture, RabbitMQ, and Queue Systems.

The fourth chapter describes the materials and methodology employed, including the use of Agile Methodologies, pre-methodology result analysis, requirements, tasks, and spring backlog. The fifth chapter showcases the findings and their analysis, including architecture diagrams such as the C4 diagram and flow diagram, providing a clear insight into the prototype implementation.

Finally, the sixth chapter outlines the conclusions reached from the study, highlighting the importance of developing the prototype to simplify task queue management, along with recommendations for future research in this field.

Palabras clave:
Microservices, Clean Architecture, RabbitMQ, Scalability

1. Introducción

En la actualidad, los equipos de desarrollo necesitan que las aplicaciones sean altamente escalables y de desarrollo rápido. Los microservicios son una arquitectura para el desarrollo de software, en la cual el sistema se conforma de componentes pequeños y autónomos que se comunican entre sí mediante interfaces de programación de aplicaciones (APIs). En comparación con las aplicaciones monolíticas, el diseño, la prueba, la implementación y la actualización de microservicios resultan más sencillos. Este enfoque fomenta la innovación y acorta el tiempo necesitado para sacar al mercado aplicaciones con nuevas funcionalidades.

La arquitectura limpia (Clean architecture) es un enfoque para diseñar sistemas de software que se centra en la separación de preocupaciones y la independencia de las capas. Esta arquitectura separa la lógica de negocio de su interfaz y su infraestructura. Así cada capa de una aplicación puede ser desarrollada y mantenida de forma independiente. Es decir, se pueden realizar cambios o mejoras en cualquier parte de la aplicación sin preocuparse por su impacto en otros lugares del código (Geekebrains, 2022).

ASP.NET Core es un marco de desarrollo moderno y de código abierto que facilita construir rápidamente aplicaciones web, para la nube, dispositivos móviles e IoT, ofreciendo un alto rendimiento. Es compatible con macOS, Windows, Linux, y Docker (Microsoft, s/f-b).

Los procesos encolados o ejecutados en segundo plano son esenciales para garantizar el funcionamiento eficiente y ágil de las aplicaciones. Estos procesos incluyen tareas como la descarga de documentos, envío de correos electrónicos, procesamiento de imágenes, entre otros. Para abordar esta necesidad, se han desarrollado numerosas tecnologías, siendo RabbitMQ una de las más usadas.

RabbitMQ es una herramienta de mensajería de código abierto que actúa como un intermediario para la comunicación entre sistemas separados. Utiliza colas para gestionar y transferir datos de manera asíncrona, lo que facilita la comunicación entre sistemas distribuidos. RabbitMQ implementa el protocolo AMQP (Advanced Message Queuing Protocol) y proporciona una infraestructura confiable y escalable para la entrega y el procesamiento de mensajes (Diego Coder, 2023b).

2. Determinación del Problema

Actualmente, el uso convencional de RabbitMQ presenta un inconveniente significativo: la necesidad de crear una consola dedicada para cada cola. Esto conlleva una gestión compleja y problemática, dificultando la comunicación asíncrona y la eficiente administración de mensajes en sistemas distribuidos. Este obstáculo podría agravarse aún más en entornos escalables.

Por estos motivos, se propone el desarrollo de un prototipo de Microservicios utilizando Clean Architecture y ASP.NET Core 8.0, que permita la creación y consumo de cualquier tipo de cola. En otras palabras, se plantea desarrollar una herramienta que simplifique de manera significativa la administración de colas. En lugar de depender de múltiples consolas, este enfoque requerirá únicamente dos microservicios: uno dedicado a la creación de colas y otro encargado de su consumo. Esta propuesta proporcionará flexibilidad y agilidad en la gestión de colas. Es importante destacar que, aunque existen herramientas de encolamiento en el mercado, muchas de ellas son de pago.

2.1 Objetivos

2.1.1 Objetivo general

Analizar el estado del arte de cuatro documentos relacionados a los problemas de encolamiento para identificar las necesidades y desafíos en el desarrollo de sistemas de colaboración.

2.1.1 Objetivos específicos

- Diseñar una arquitectura que permita la creación de un modelo para la gestión de procesos encolados.
- Desarrollar e implementar una arquitectura basada en Clean Architecture Microservices utilizando las tecnologías ASP.NET Core 8.0 y RabbitMQ para abordar los problemas de encolamiento.

- Ejecutar pruebas para validar la funcionalidad integral del sistema, incluyendo la gestión de procesos encolados, abordando y corrigiendo cualquier error identificado.

3. Marco teórico referencial

3.1 Estado del Arte

En el artículo Evaluation of Microservice Communication While Decomposing Monoliths, publicado el 2023 en Vilnius, Lituania, el objetivo es evaluar y comparar cinco tecnologías de comunicación, mediante los criterios de evaluación propuestos para la comunicación entre microservicios.

La idea principal de este estudio es evaluar tecnologías de comunicación sincrónica y asincrónica, y determinar casos particulares para su aplicación mientras se descompone un monolito en aplicaciones nativas de la nube. Define el contexto de la arquitectura de microservicios y las ventajas y desventajas de cada tecnología de comunicación adecuada ya que los microservicios deben interactuar utilizando tecnologías de comunicación entre procesos según los criterios de evaluación propuestos. Utiliza una metodología de Diseño experimental, criterios y métricas, topologías, herramientas como bibliotecas y equipos de TI utilizados para realizar experimentos, además de definir métricas de comparación con cinco tecnologías de comunicación, como: HTTP Rest, RabbitMQ, Kafka, gRPC y GraphQL.

En conclusión, se introdujo nuevos criterios como la influencia en la topología del microservicio, el rendimiento de la llamada a procedimiento remoto, el tamaño del mensaje, el consumo de memoria, el uso del almacenamiento, el tiempo de arranque y la disponibilidad de las bibliotecas correspondientes para evaluar aspectos que potencialmente pueden ser un desafío (Justas Kazanavičius & Dalius Mažeika, 2023).

En el artículo Microservice development using RabbitMQ message bróker publicado el 2022 en Zenica, el objetivo principal es demostrar cómo los microservicios pueden mejorar la agilidad y la eficiencia en el desarrollo de aplicaciones.

La idea principal del artículo es resaltar las ventajas de adoptar la arquitectura de microservicios en el desarrollo de aplicaciones, y cómo RabbitMQ facilita la comunicación entre estos microservicios. Además, el artículo presenta resultados que resaltan la eficacia de RabbitMQ en la reducción de la carga y el tiempo de entrega en aplicaciones web, además de su capacidad para garantizar la entrega de mensajes correctos, contribuyendo a la robustez del sistema. Este artículo se relaciona con el proyecto de titulación, ya que ambos se centran en la utilización de RabbitMQ en el contexto de la arquitectura de microservicios.

En conclusión, el artículo demuestra cómo la combinación de RabbitMQ y la arquitectura de microservicios ofrece una solución efectiva para el desarrollo ágil y escalable de aplicaciones (Amar Ćatović, Nevzudin Buzađija, & Samir Lemes, 2022)

El artículo A Highly Reliable Communication System for Internet of Robotic Things and Implementation in RT-Middleware With AMQP Communication Interfaces, publicado en el 2021 por la IEEE tiene como objetivo desarrollar un sistema IoRT (Internet of Robotic Things) confiable y robusto a través del uso del protocolo AMQP (Advanced Message Queuing Protocol).

La idea principal de este artículo es utilizar el protocolo de comunicación AMQP dentro de RT-Middleware para mejorar la confiabilidad de la comunicación en los sistemas IoRT. Además, enfatiza la elección de los protocolos de comunicación adecuados para construir sistemas robóticos robustos y eficientes que puedan operar en redes estables como inestables. Se realizaron dos tipos de pruebas comparando AMQP con otras interfaces de comunicación como CORBA y MQTT. Pruebas de rendimiento en tiempo real que miden el tiempo de retraso de ida-vuelta y pruebas de calidad de la mensajería que evalúan la cantidad de errores en el orden de llegada de los mensajes y mensajes faltantes en un entorno de alta carga.

Los resultados de las pruebas se resumieron en diagramas de caja y barras. Estos resultados muestran que la interfaz de comunicación AMQP puede mantener suficientemente mensajes de alta calidad en condiciones de red inestables y de alta

carga. Este artículo se relaciona con este Proyecto de titulación en el uso de AMQP, pero difieren en sus enfoques principales: uno se centra en la arquitectura de microservicios y ASP.NET Core, mientras que el otro se enfoca en la comunicación en tiempo real para IoRT utilizando AMQP y RT-Middleware. Se puede concluir que al usar AMQP se obtuvo ventajas como garantizar la confiabilidad y el orden de los mensajes, lo que lo hace eficiente en el contexto de Internet de las cosas robóticas como para el encolamiento de la data (Yoshino, Watanobe, & Naruse, 2021).

El artículo A Fair Comparison of Message Queuing Systems, publicado en el 2021 por la IEEE, evalúa y compara las características de sistemas de mensajería como Kafka, RabbitMQ, RocketMQ, ActiveMQ y Pulsar. La idea principal es analizar la funcionalidad y rendimiento de estos sistemas bajo condiciones específicas de prueba, para identificar los puntos fuertes y debilidades de la calidad de servicio y funcionalidades.

La metodología utilizada es la comparación estandarizada en un entorno experimental reproducible. Desarrolla un marco de prueba que garantiza una comparación justa del rendimiento. Además, diseña una herramienta de prueba de throughput/latencia personalizada para obtener resultados unificados y comparables. Se realizaron análisis cualitativos y cuantitativos de sistemas de cola de mensajes (Kafka, RabbitMQ, RocketMQ, ActiveMQ y Pulsar). Entre los resultados principales, destaca Kafka con resultados de throughput bajo en diferentes condiciones y RocketMQ con baja latencia.

Este artículo se relaciona con este Proyecto de titulación en evaluar los sistemas de colas más famosos en la actualidad y usarlos en sistemas que ofrecen gran transmisión y alto volumen de procesamiento de datos. Se destaca la importancia de elegir el sistema adecuado según los requisitos específicos de la aplicación. Finalmente, concluye que Kafka destaca en throughput debido a sus técnicas de optimización, como zero-copy, lectura/escritura secuencial de disco y compresión de datos. RocketMQ exhibe una notable superioridad en términos de latencia, atribuida a diversas técnicas de optimización, como la reducción de la latencia de

pausas de JVM, la reducción de la latencia de bloqueo y la reducción de la latencia de la caché de páginas (Fu, Zhang, & Yu, 2021).

3.2 Definiciones Previas

3.2.1 Microservicios

Los microservicios son un enfoque de arquitectura que divide una aplicación en pequeños servicios independientes que se comunican mediante interfaces definidas. Este enfoque, implica desplegar y escalar cada componente de manera independiente, promoviendo la flexibilidad y la eficiencia en el desarrollo (IBM, s/f).

En una arquitectura monolítica, todos los componentes están interconectados y funcionan como una sola unidad. Esto significa que, si un componente experimenta un aumento de la demanda, toda la arquitectura debe ser escalada. Además, agregar o mejorar características se vuelve complicado a medida que la aplicación crece. Esto incrementa el riesgo de errores, ya que cualquier fallo en un componente puede afectar la disponibilidad de toda la aplicación al depender entre sí.

En el enfoque de microservicios, una aplicación se desarrolla utilizando partes separadas llamadas servicios, cada uno de ellos funciona de manera independiente y se especializa en una tarea específica. Cada servicio puede desarrollarse, implementarse y escalarse sin afectar a los demás. Estos servicios intercambian información mediante interfaces y son diseñados para realizar tareas específicas (aws, s/f).

Ventajas

- Agilidad: Los microservicios impulsan la creación de equipos independientes, cada uno encargado de servicios específicos. Esto permite que los equipos trabajen de manera más rápida e independiente.

- Escalado flexible: Los microservicios promueven el crecimiento independiente de cada parte de la aplicación. Esto disminuye costos y garantiza la disponibilidad de los servicios.
- Simple implementación: Los microservicios facilitan la implementación continua y la integración sin problemas. Aumentando la velocidad del equipo en la entrega de nuevas características.
- Código reutilizable: Un servicio creado para una función específica puede usarse como base para otros servicios, facilitando el desarrollo de nuevas características sin tener que empezar desde cero.
- Resistencia a errores: La autonomía de los servicios hace que una aplicación sea más robusta frente a errores. En un sistema monolítico, un error en una parte puede afectar a la aplicación entera, pero en el caso de los microservicios, un error en un servicio puede ser aislado y manejado sin afectar el funcionamiento global.

3.2.2 Arquitectura Estándar de Microservicios

Los microservicios son como piezas pequeñas y autónomas de software, donde cada una es un conjunto de código separado y gestionado por un equipo de desarrollo independiente. (Microsoft, s/f-a).

Además de los servicios, en una arquitectura de microservicios típica hay otros elementos como:

- Administración e implementación. Este componente se encarga de tareas como distribuir servicios en nodos, detectar errores y redistribuir los servicios entre los nodos según sea necesario. Se puede comparar a un director de orquesta que utiliza tecnologías como Kubernetes para coordinar y gestionar el despliegue de los servicios (Microsoft, s/f-a).

- Puerta de enlace de API: Punto de acceso principal para los clientes. En vez de que los clientes se comuniquen directamente con los servicios, ellos llaman a esta puerta, que luego dirige la llamada al servicio correcto por detrás. Además

de esto, la puerta de enlace puede encargarse de la autenticación, el registro, la seguridad SSL y la distribución equitativa de la carga. (Microsoft, s/f-a).

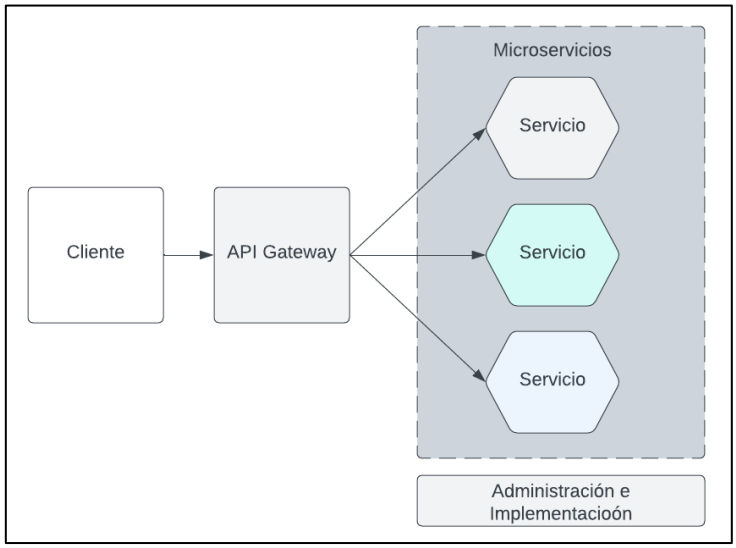


Ilustración 1 Diagrama de Arquitectura Estándar de Microservicios

3.2.3 Clean architecture

Las arquitecturas limpias, también llamadas "Clean Architectures", son un grupo de principios y patrones de diseño de software creados por Robert C. Martin. Destacan por su enfoque en la separación clara de las preocupaciones en capas definidas, con reglas estrictas para su interacción, creando software más fácil de mantener y escalar (Diego Coder, 2023a).

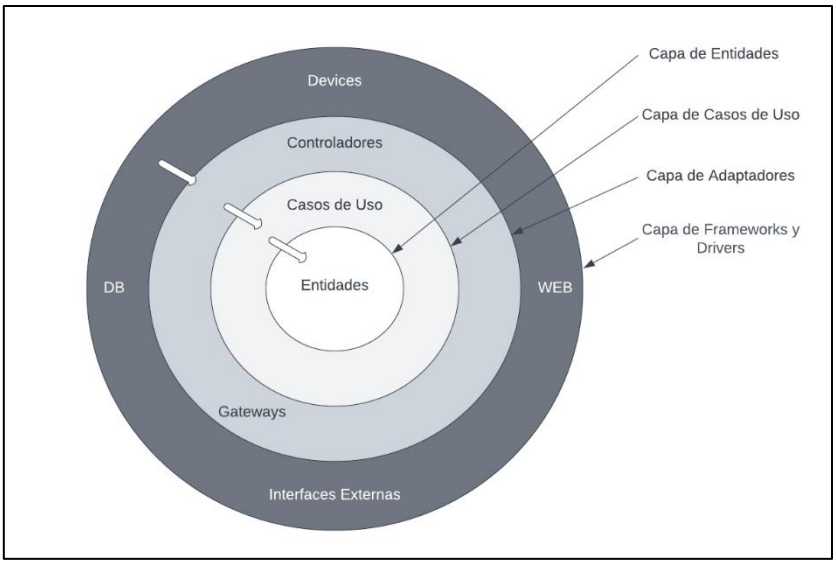


Ilustración 2 Diagrama de Arquitectura Limpia

Capas

- Capa de Entidades: Contiene las entidades de negocio o modelos que encapsulan el estado y la lógica fundamental de la aplicación. Las entidades no deben depender de ninguna otra capa.
- Capa de Casos de Uso: Define las reglas de negocio de la aplicación. Los casos de uso encapsulan la lógica específica para las funciones de la aplicación.
- Capa de Adaptadores: Sirve como punto de conexión con el mundo exterior, como la interfaz de usuario, base de datos, y otros servicios externos. Los adaptadores transforman datos entre las capas internas y externas.
- Capa de Frameworks y Drivers: Contiene el código que se conecta con bibliotecas externas, frameworks y drivers, siendo la capa más externa de la arquitectura.

Patrones de diseño en Clean Architecture

En Clean Architecture, se pueden utilizar diferentes patrones de diseño para conseguir modularidad y separación de responsabilidades. Algunos de estos patrones son:

- Principio de Inversión de Dependencias (DIP): Propone que los componentes de alto y bajo nivel se relacionen a través de abstracciones, como interfaces y la inyección de dependencias, en lugar de tener una dependencia directa entre sí.

- Inversión de Control (IoC): En lugar de que los componentes principales se encarguen directamente de crear y administrar los objetos que requieren, esta responsabilidad se delega a un contenedor de inversión de control (IoC). En este enfoque, el contenedor de IoC se encarga de crear y administrar la vida útil de los objetos, permitiendo que los componentes de alto nivel soliciten estos objetos cuando los necesitan. La IoC también facilita la inyección de dependencias, lo que significa que las dependencias requeridas por un componente no se crean dentro de ese componente, sino se inyectan desde fuera (Diego Coder, 2023a).

3.2.4 Arquitectura Basada en Eventos

La arquitectura basada en eventos facilita la comunicación entre servicios independientes al registrar y procesar eventos. Este enfoque permite que los sistemas operen de manera asincrónica mientras comparten información y completan tareas.

Muchas aplicaciones modernas se basan en esta arquitectura, lo que permite una integración flexible y un acoplamiento mínimo entre los sistemas. Los eventos capturan cambios en el estado del sistema, pueden provenir de diversas fuentes y se distribuyen a través de productores y consumidores de eventos. Plataformas como Apache Kafka se usan para procesar eventos en tiempo real, proporcionando alto rendimiento y baja latencia. Otros middlewares de administración de eventos también son comunes en el procesamiento de eventos distribuidos (Red Hat, 2019).

Elementos de una arquitectura basada en eventos:

- Productor de eventos (Event Producer): Este componente genera eventos en respuesta a acciones específicas dentro del sistema. El productor de eventos se

encarga de enviar estos eventos al enrutador de eventos para que sean procesados y distribuidos a los consumidores apropiados.

- Enrutador de eventos (Event Router): El enrutador de eventos es un intermediario que recibe eventos del productor y los envía a los consumidores adecuados. Puede filtrar, enriquecer y aplicar políticas de seguridad a los eventos para garantizar su entrega eficiente y segura.
- Consumidor de eventos (Event Consumer): Los consumidores de eventos reciben y procesan los eventos del enrutador. Pueden ser aplicaciones o servicios que utilizan los datos de los eventos para realizar acciones específicas, como enviar correos electrónicos de confirmación o generar informes de análisis.

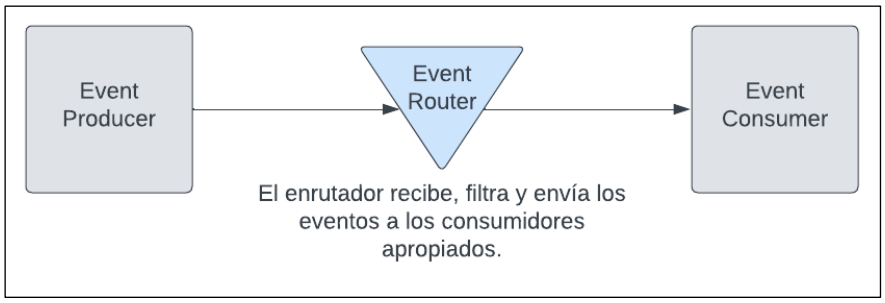


Ilustración 3 Diagrama de Arquitectura Basada en Eventos

Ventajas

- Escalado y errores por separado: En una arquitectura basada en eventos, al desvincular los servicios, cada uno puede escalar y manejar fallos de forma independiente, lo que mejora la resiliencia de la aplicación. Además, facilita la creación de sistemas en tiempo real y ofrece escalabilidad mediante el uso de servicios de mensajería para la transmisión de eventos.
- Desarrollar con agilidad: La arquitectura basada en eventos simplifica el desarrollo al automatizar la gestión de eventos y eliminar la necesidad de código personalizado. Los enrutadores de eventos dirigen automáticamente los eventos a los consumidores, lo que agiliza el proceso y reduce la coordinación entre los servicios. Además, al basarse en inserción, esta arquitectura permite una escalabilidad eficiente y reduce los costos operativos.

- Creación de sistemas extensibles: La arquitectura basada en eventos permite una fácil extensibilidad. Los equipos pueden agregar nuevas funciones sin afectar a los microservicios existentes. Además, al publicar eventos, las integraciones con sistemas actuales y futuros son sencillas, sin agregar dependencias.
- Auditaría sencilla: El enrutador de eventos es un centro de control que establece políticas de seguridad, controla el acceso a los datos y cifra los eventos para mayor protección (AWS, 2023).

3.2.5 Patrón Mediator

El patrón Mediator reduce las dependencias entre objetos al restringir las comunicaciones directas entre ellos y forzar su colaboración a través de un objeto mediador. Esto hace que los componentes dependan solo del mediador en lugar de estar acoplados entre sí. Este enfoque facilita la reutilización y modificación de las clases. Se aplica cuando las clases están demasiado acopladas entre sí y es difícil cambiarlas o reutilizarlas en diferentes contextos.

Los componentes que conforman el patrón se explican a continuación:

- Client: El componente que inicia la comunicación con otros componentes a través del mediador.
- Components: Forman parte de la comunicación utilizando el mediador como intermediario. Estos componentes pueden ser diferentes objetos que comparten el mismo mediador para intercambiar información entre ellos.
- Mediador: Es el componente que actúa como intermediario entre los demás componentes, encargándose de dirigir los mensajes recibidos al destinatario adecuado.

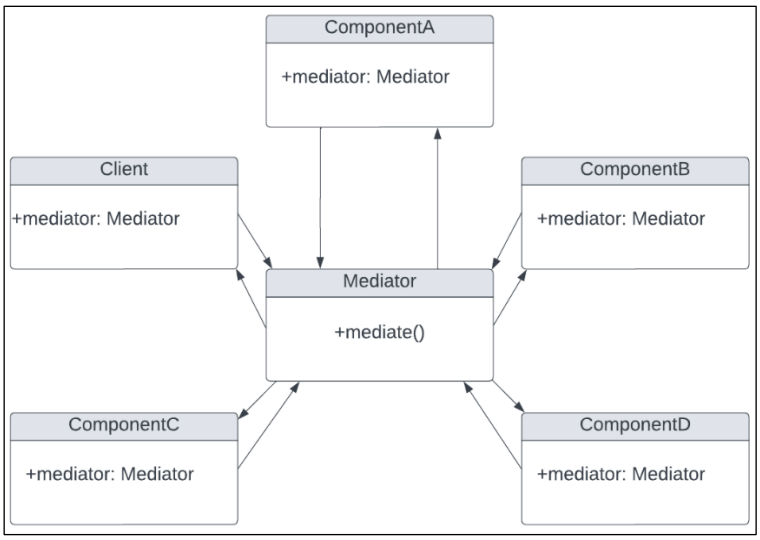


Ilustración 4 Diagrama del Patrón Mediator

Beneficios

- Principio de responsabilidad única: Centraliza las interacciones entre diferentes partes en un solo lugar, lo que simplifica su comprensión y mantenimiento.
- Principio de abierto/cerrado: Fexibilidad de agregar nuevos mediadores sin necesidad de realizar modificaciones en los componentes existentes.
- Se reduce el acoplamiento.
- Se reutilizar componentes individuales.

(Refactoring Guru, s/f)

MediatR es una poderosa herramienta en C# que implementa el patrón Mediator en .NET. Facilita la comunicación entre objetos o clases de forma indirecta, admitiendo una variedad de operaciones como solicitudes/respuestas, comandos, consultas, notificaciones y eventos, tanto síncronos como asíncronos. Además, utiliza la generic variance de C# para un envío inteligente de mensajes (Arbems, s/f).

3.2.6 Reflexión

La reflexión en C# mediante las clases dentro del namespace: System.Reflection y System.Type permite obtener información sobre ensamblados cargados y tipos definidos en ellos, como clases, interfaces y tipos de valor. También posibilita la

creación de instancias de tipos en tiempo de ejecución, invocar métodos y acceder a campos y propiedades.

Los ensamblados contienen módulos, que a su vez contienen tipos y miembros. La reflexión proporciona objetos que encapsulan ensamblados, módulos y tipos, permitiendo crear instancias de tipos, enlazar tipos a objetos existentes y acceder a miembros de tipos.

Usos:

- Definir y cargar ensamblados.
- Acceder a información de módulos, como el ensamblado que los contiene y las clases que contienen.
- Obtener detalles sobre constructores, métodos, campos, eventos, propiedades y parámetros de un tipo.
- Examinar atributos personalizados.
- Compilar tipos en tiempo de ejecución mediante System.Reflection.Emit.
- Además, se emplea para crear exploradores de tipos y es utilizada por compiladores de lenguajes y clases de serialización y remoting para diversos propósitos (Microsoft, 2024).

System Type

En C# se utilizan las declaraciones de tipos para representar diferentes estructuras de datos como clases, interfaces, matrices, tipos de valor y enumeraciones, así como también para definir tipos genéricos y sus parámetros (Microsoft, s/f-c).

3.2.7 RabbitMQ

RabbitMQ es un sistema de mensajería de código abierto que actúa como intermediario para permitir la comunicación y el envío eficaz de información entre diferentes aplicaciones y sistemas. RabbitMQ implementa el protocolo AMQP (Advanced Message Queuing Protocol) para la entrega y el procesamiento de

mensajes; permite a las aplicaciones enviar, recibir y procesar mensajes de manera segura y escalable (Diego Coder, 2023b).

Protocolo AMQP

Advanced Message Queuing Protocol es un protocolo de red para la comunicación entre aplicaciones y sistemas que necesitan enviar y recibir mensajes de manera eficiente y confiable. Creado con la intención específica de simplificar el intercambio de mensajes entre sistemas y aplicaciones que funcionan en diferentes plataformas. Su tarea principal es hacer más fácil la comunicación asíncrona entre los diferentes componentes. Esto implica que las aplicaciones pueden enviar mensajes sin tener que aguardar una respuesta inmediata, permitiéndoles operar de forma independiente mientras los mensajes se procesan en segundo plano (Ionos, s/f).

Características de RabbitMQ

- Colas de mensajes: Representan puntos de almacenamiento temporal para los mensajes antes de su procesamiento. RabbitMQ permite a las aplicaciones productoras (o publicadores) enviar mensajes a colas, y permite a las aplicaciones consumidoras (o suscriptores) recuperar y procesar estos mensajes.

- Intercambio de mensajes: RabbitMQ gestiona el envío de mensajes a través de intercambios, que son componentes encargados de dirigir los mensajes hacia colas específicas según reglas predefinidas de enrutamiento.

- Patrones de enrutamiento: RabbitMQ admite diversos patrones de enrutamiento, como el enrutamiento directo, el de topic y el fanout. Estos patrones permiten a las aplicaciones definir cómo son distribuidos los mensajes entre las colas según criterios específicos.

- Garantías de entrega: RabbitMQ asegura la entrega confiable de mensajes a las colas, incluso en situaciones como fallos de red. Esto minimiza el riesgo de pérdida de mensajes durante el proceso de envío y recepción.

- Modelo de mensajes: Los mensajes en RabbitMQ pueden contener datos en formato de texto o binario. Los usuarios pueden personalizar los encabezados y propiedades de los mensajes para añadir información adicional que facilite su procesamiento.

- Escalabilidad: RabbitMQ presenta alta escalabilidad, siendo capaz de gestionar una gran cantidad de mensajes y conexiones simultáneas. Esto incrementa su popularidad para sistemas distribuidos y aplicaciones que demandan una comunicación confiable y de alto rendimiento.

- Plugins y extensiones: RabbitMQ soporta diversos plugins y extensiones. Estas permiten incorporar funcionalidades extras, como la encoladura basada en prioridades, la gestión de usuarios y permisos, y la integración con otros sistemas.
- Almacenamiento de mensajes: Un aspecto clave de RabbitMQ es su capacidad para almacenar temporalmente mensajes en colas. Esta función posibilita el envío y recepción asíncronos de mensajes, liberando a las aplicaciones de la necesidad de estar activas y en espera de respuestas inmediatas para continuar su funcionamiento.

Arquitectura de RabbitMQ

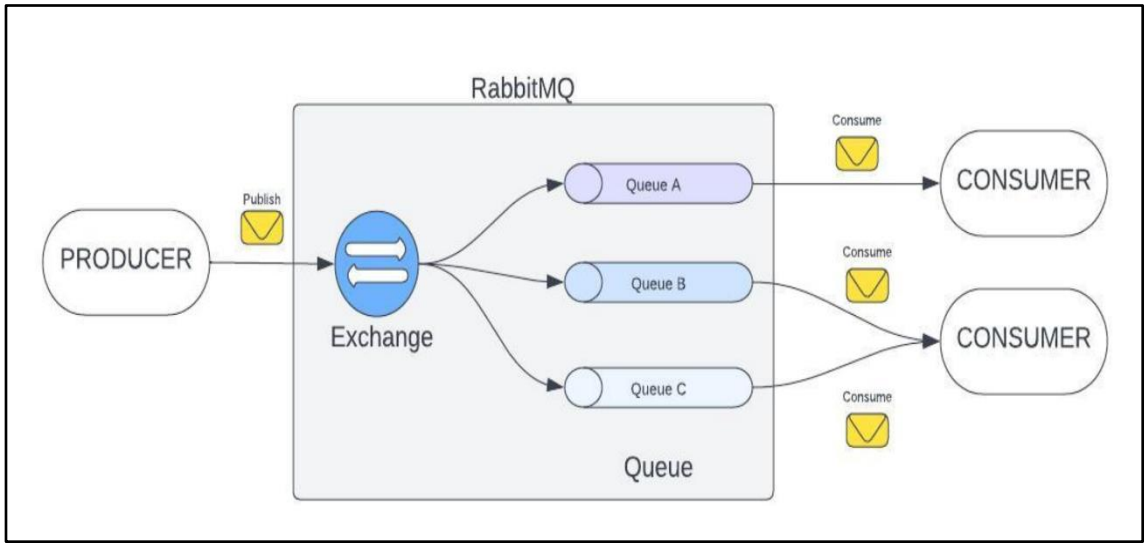


Ilustración 5 Diagrama de Arquitectura RabbitMQ

- RabbitMQ (Message Broker): RabbitMQ actúa como intermediario y consta de varios componentes. Recibe los mensajes de los productores y los almacena temporalmente en colas. Estas colas son como áreas de espera donde los mensajes esperan a ser entregados a los consumidores.
- Productor (Producer): Aplicación o componente que genera y envía mensajes al sistema de mensajería. En otras palabras, es la entidad que crea y publica mensajes en RabbitMQ. Los productores envían mensajes a través de intercambios, que son puntos de entrada donde los mensajes ingresan al sistema de RabbitMQ. Estos mensajes pueden contener información, datos o instrucciones que se destinarán a ser procesados por las aplicaciones consumidoras.
- Intercambio (Exchange): Recibe mensajes de los productores y los envía a las colas apropiadas basándose en reglas de enrutamiento. En otras palabras, es un punto de entrada para los mensajes en RabbitMQ que determina cómo se distribuyen a las colas.
- Cola (Queue): Estructura de almacenamiento temporal donde los mensajes son almacenados hasta que los consumidores estén listos para procesarlos. Por defecto, las colas en RabbitMQ son del tipo FIFO, lo que significa que los mensajes se manejan en el orden en que llegan a la cola.
- Canales (Channels): Conexión virtual dentro de una conexión TCP establecida entre un cliente y un servidor RabbitMQ. Los canales son una característica importante que permite a los clientes interactuar con el servidor de manera eficiente y escalable.
- Consumidor (Consumer): Aplicación o componente que recibe y procesa los mensajes enviados por los productores. Los consumidores están conectados a las colas, que actúan como áreas de almacenamiento temporal para los mensajes. Los consumidores obtienen mensajes de las colas y realizan acciones basadas en el contenido de los mensajes. Por ejemplo, pueden realizar cálculos, actualizar bases de datos o tomar decisiones según los datos en los mensajes (Diego Coder, 2023b).

3.2.8 Sistemas de Colas

Un sistema de colas, o fila, es una estructura que organiza datos para ser procesados en orden de llegada. La teoría de colas estudia cómo se comportan estas líneas de espera en diferentes situaciones. Se forma una cola o linea de espera cuando un servidor tiene capacidad limitada para atender solicitudes y no está disponible inmediatamente (Gabriel Esteban Vel´azquez, s/f).

Una cola es como una fila visual donde las personas esperan su turno. La teoría de colas, por otro lado, es un conjunto de modelos matemáticos que detallan cómo funcionan las colas en diferentes situaciones. Estos modelos son importantes para equilibrar el tiempo de espera y los costos (UM, s/f).

Componentes de un sistema de colas

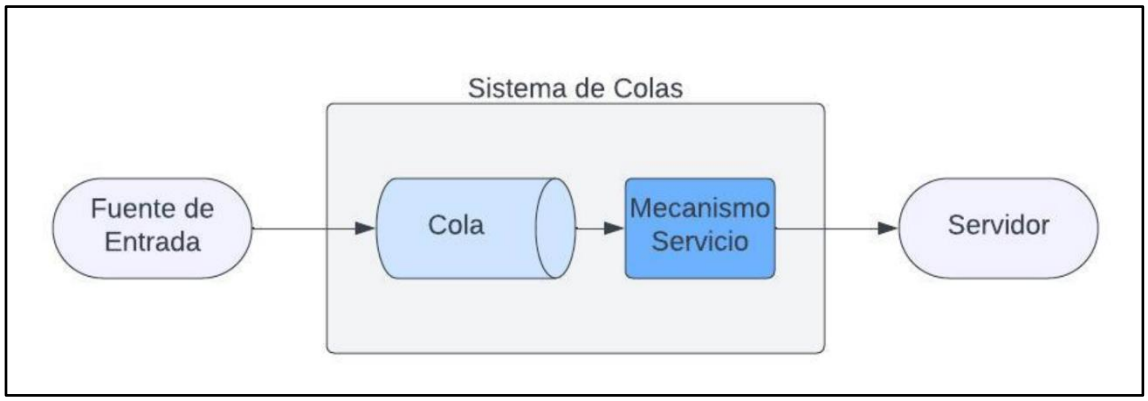


Ilustración 6 Componentes de sistemas de colas

- Fuente de entrada: Punto de inicio de los elementos que entran en el sistema.
- Cola: Estructura de datos que guarda los elementos que aguardan ser tratados. La cola se caracteriza por su capacidad (es la cantidad máxima de elementos que puede gestionar), y su disciplina, que determina el orden en que los elementos son seleccionados para recibir el servicio. Las disciplinas de cola más comunes son:
    - First In-First Out (FIFO): En este modelo, el primero que ingresa es el primero en ser atendido, manteniendo así la cola en el orden de llegada de los elementos.

- Last In-First Out (LAFO): El último que ingresa es el primero en ser atendido, lo que resulta en una cola ordenada de manera inversa.
- Service-In-Random-Order (SIRO): Se elige al azar qué elemento en espera será atendido a continuación.
- Servidor (o servidores): Es la entidad encargada de procesar los elementos de la cola.

4. Materiales y metodología

4.1 Metodologías Ágiles

En la siguiente tabla se comparan cinco metodologías ágiles para analizar y contrastar sus características y desempeño; así se obtendrá la metodología ágil a usarse en este proyecto.

Tabla 1

Cuadro Comparativo Metodologías Ágiles

Características	XP	SCRUM	LEAN	KANBAN	RUP
Presencia empresarial	X	X	X	X	X
Respeto de las fechas de entrega		X			X
Definición y cumplimiento de los requisitos	X	X	X	X	X
Respeto al nivel de calidad	X	X		X	X
Modelos de diseño	X	X			X
Roles de trabajo		X			
Ciclos de trabajo		X			
Entregas continuas		X	X		X
Menor probabilidad de errores	X	X		X	X
Sujeto a cambios	X	X	X		
Programación organizada	X	X	X	X	X
Efectividad en proyectos	X	X	X	X	X
Visualización de tareas		X		X	
Retroalimentación	X	X			X

4.2 Análisis de los resultados previa a la aplicación de la metodología

Basándonos en el análisis presentado en la Tabla 1, se concluye que la elección de la metodología Scrum es la más adecuada, ya que ofrece ventajas en cuanto a cumplimiento de plazos, definición de roles y ciclos de trabajo.

En Scrum, los períodos de tiempo están definidos por la duración de un Sprint. Un Sprint es un intervalo fijo y breve en el que se lleva a cabo el trabajo planificado. Los sprints tienen una duración de tiempo fija, un compromiso de entrega establecido lo que permite que el equipo tenga un ritmo constante. Scrum se basa en ciclos de desarrollo iterativos e incrementales. Esto significa que se entregan partes funcionales al final de cada iteración. En este caso al ser un proyecto con un plazo corto es fundamental tener entregas frecuentes y que añadan valor.

Scrum define tres roles principales y fundamentales para el desarrollo de un proyecto: Product Owner, Scrum Master y el Scrum Team. Estos roles son encargados de la buena ejecución de cada Sprint y del proyecto en general. Sin embargo, en este proyecto se contempla una adopción parcial de Scrum, centrándonos en la aplicación de ciertas prácticas esenciales. En este proyecto, se contemplan los siguientes roles: Product Owner: Tutor de Tesis, quién representará los requerimientos del cliente; Scrum Master: En este proyecto no existirá un Scrum Master, aunque sí se garantizará los principios y prácticas Scrum; Development Team: Gisela y Luis Fernando, quienes desarrollarán el proyecto.

Además, en este proyecto, se contemplan las siguientes reuniones: Weekly Scrum: Reunión semanal donde el equipo de desarrollo coordinará su trabajo y discutirá avances y obstáculos; Sprint Planning: Reunión al principio de cada sprint para establecer qué tareas se llevarán a cabo durante ese periodo y cómo se realizarán.; Sprint Review: Reunión al término de cada sprint para mostrar el producto desarrollado hasta el momento y obtener comentarios y sugerencias. Finalmente, en este proyecto, cada iteración tendrá una duración de un mes, durante la cual se

abordarán las funcionalidades o tareas establecidas para ese sprint. Al concluir cada sprint, se obtendrá un incremento que añadirá valor.

4.3 Requerimientos

Los requerimientos se centran en el diseño de Microservicios con Clean Architecture, ASP.NET Core 8.0 y RabbitMQ y el desarrollo de su prototipo. Se iniciará con un análisis del funcionamiento de RabbitMQ y se detallarán sus principales componentes como queues, routings keys, subscribers, publishers y exchanges, además, se analizarán los principios y mejores prácticas de la Clean Architecture (programación por capas de persistencia, presentación y dominio) así como también los patrones de diseño que se implementarán tales como MediatR e IoC (Inversion of Control) los cuales ayudarán en la comunicación entre capas y servicios.

Se creará y configurará contenedores docker usando WSL para los microservicios y el servicio de RabbitMQ, luego, con el uso de clean code se desarrollará interfaces genéricas reutilizables que entablará una comunicación asíncrona entre los microservicios (consumers y producers). Finalmente, se llevarán a cabo pruebas de funcionalidad de este prototipo al poder encolar todo tipo de llamados de endpoints de tipo API REST.

4.4 Tareas

- Analizar los microservicios y la arquitectura de microservicios.
- Analizar Clean Architecture.
- Analizar el funcionamiento de RabbitMQ.
- Describir los componentes clave de RabbitMQ como queues, routing keys, subscribers, publishers y exchanges.
- Investigar de los Sistemas de Colas.
- Crear diagramas de arquitecturas.

- Aplicar Clean Architecture.

- Implementar Clean Architecture Microservices con ASP.NET 8.
- Construir el Setup de Docker.
- Instalar y configurar un servidor de RabbitMQ.
- Realizar programación de interfaces genéricas reutilizables para comunicación entre NET y RabbitMQ.
- Entablar una comunicación Asíncrona entre Microservices
- Implementar Pattern IoC y MediaTR
- Encolar cualquier tipo de api RESTful
- Generar Queues en RabbitMQ
- Evaluación y pruebas de funcionalidad.

4.5 Spring Backlog

El conjunto de sprints pasan a ser parte del sprint backlog.

Sprint 1: Investigación y Fundamentos

Este primer sprint, se enfocará en realizar un análisis de los microservicios y de la arquitectura de microservicios. Paralelamente, se analizará Clean Architecture, explorando sus principios y beneficios. Además, se comprenderá el funcionamiento de RabbitMQ, examinando sus características, capacidades y componentes. Finalmente, se investigarán los Sistemas de Colas.

Sprint 2: Diseño e Implementación Inicial

Este sprint, se enfocará en la creación de diagramas detallados de arquitecturas para visualizar la estructura de nuestro sistema. Luego, se implementarán los principios de Clean Architecture. Además, se contruirán los contenedores Docker, se instalará y se configurará un servidor de RabbitMQ para establecer la base de la comunicación asíncrona entre nuestros microservicios.

Sprint 3: Desarrollo Avanzado

Este sprint comenzará implementando la programación de interfaces genéricas reutilizables para facilitar la comunicación entre .NET y RabbitMQ. A continuación, se establecerá una comunicación asíncrona efectiva entre los microservicios, utilizando las mejores prácticas y patrones de desarrollo.

Además, se implementará el patrón de Inversión de Control (IoC) para mejorar la modularidad y escalabilidad del sistema. Finalmente, se realizarán las modificaciones necesarias para encolar cualquier tipo de API RESTful, garantizando una gestión eficiente de las solicitudes.

Sprint 4: Evaluación y Validación

Este sprint marcará el cierre del desarrollo, centrándonos en la evaluación y pruebas exhaustivas de la funcionalidad del sistema. Se realizarán pruebas rigurosas para asegurarnos de que todas las características implementadas funcionen como se espera y cumplan con los requisitos establecidos.

5. Resultados y discusión

5.1 Diagramas de Arquitecturas

5.1.1 Diagrama de contexto del sistema C1

Proporciona un punto de partida y muestra cómo el sistema de software encaja de forma muy general.

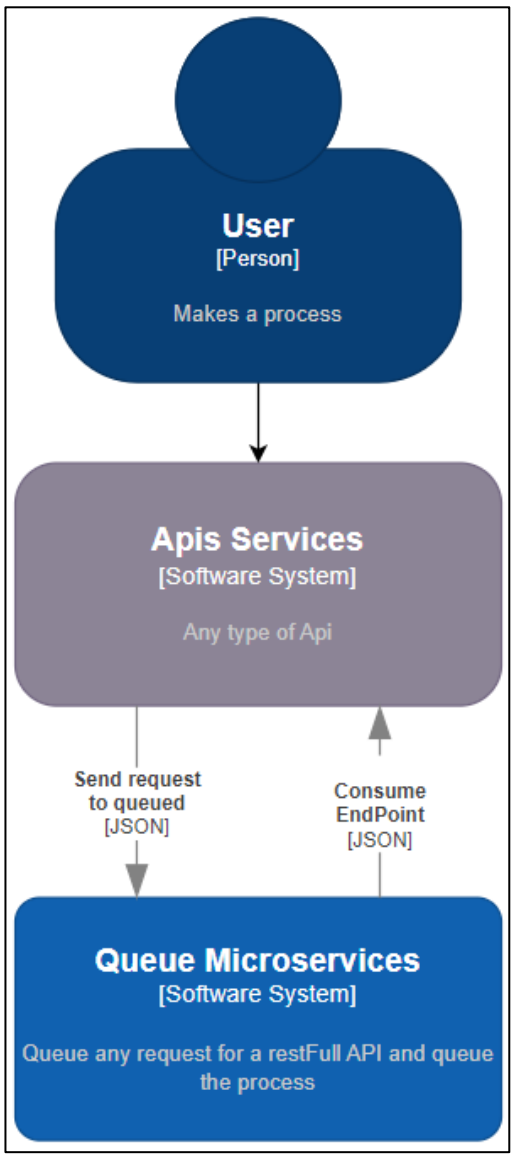


Ilustración 7 Diagrama de Contexto

5.1.2 Diagrama de contenedor C2

Se acerca al alcance del sistema de software y muestra los componentes técnicos de alto nivel.

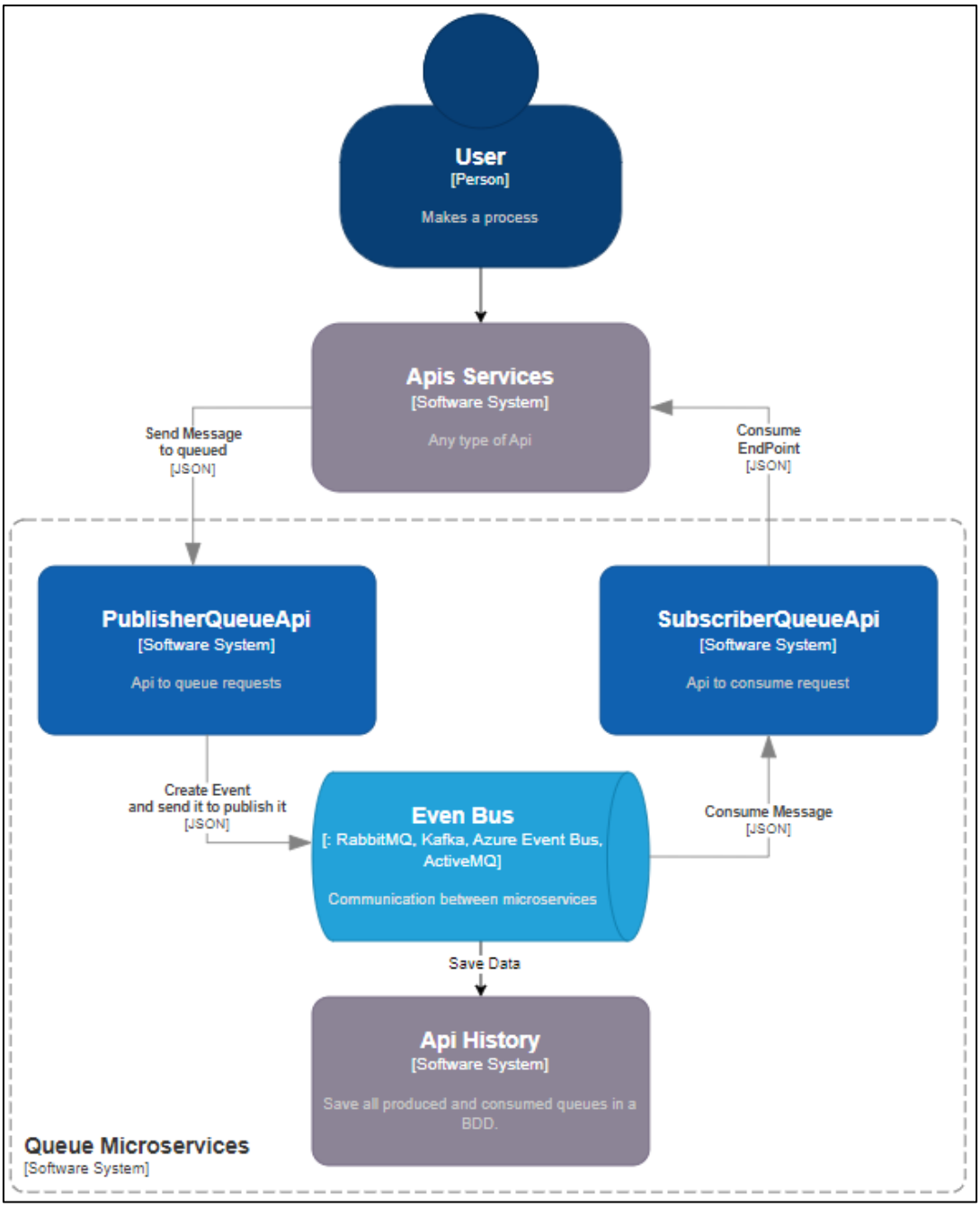


Ilustración 8 Diagrama de Contenedor

5.1.3 Diagrama de componentes C3

Se acerca a un contenedor individual y muestra los componentes que contiene.

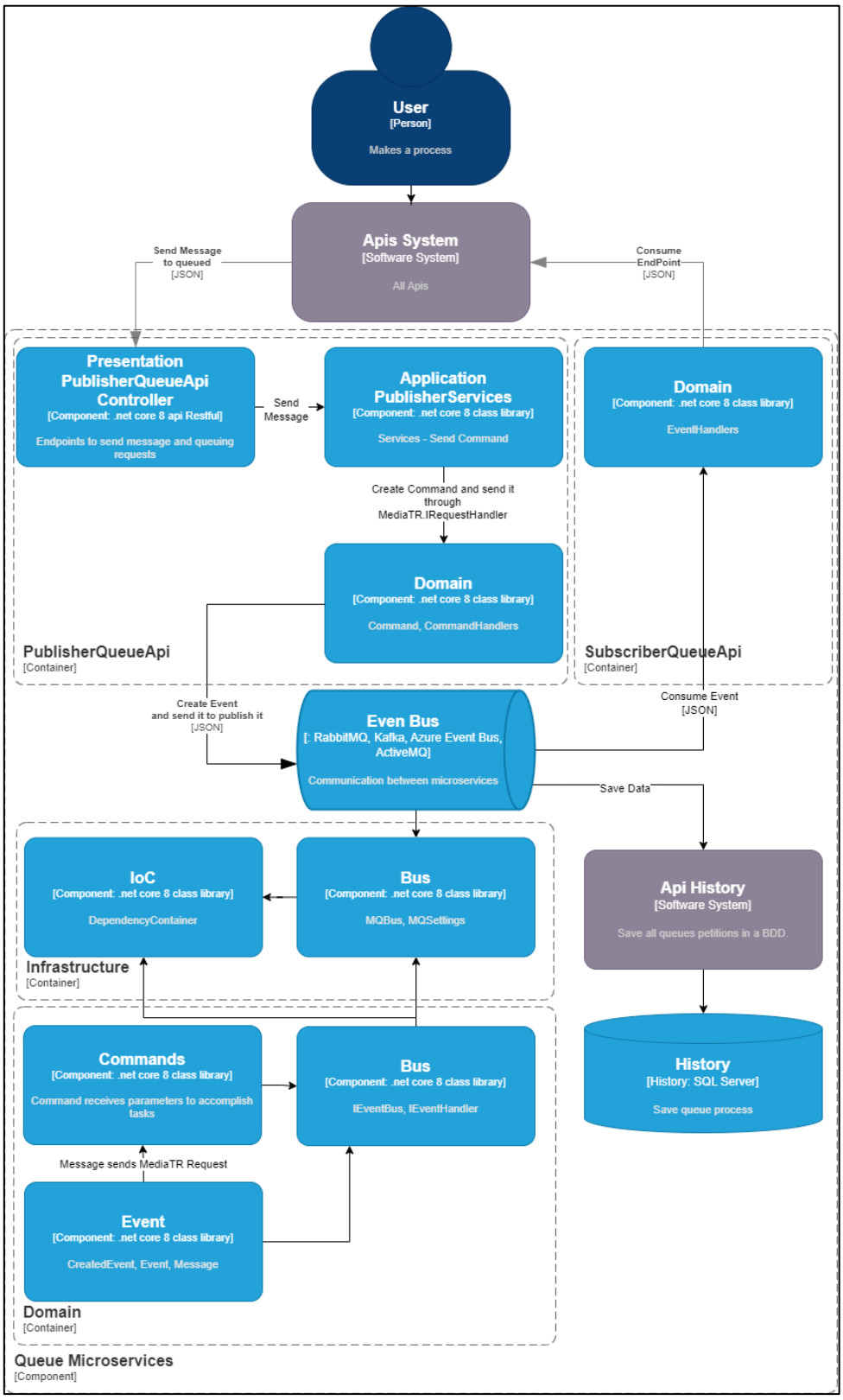


Ilustración 9 Diagrama de Componentes

5.1.4 Diagrama UML C4

Se usó un diagrama de código (clase UML) para ampliar un componente individual y mostrar cómo se implementa ese componente.

En la arquitectura limpia (clean architecture) y basada en eventos (event driven architecture), los diagramas de clases UML son fundamentales para detallar la estructura de los componentes. Estos diagramas son esenciales para entender cómo los mensajes son publicados y suscritos en el sistema de colas, se mostrará la interacción entre entidades, casos de uso, controladores de eventos y adaptadores de interfaz. Esta representación visual ayuda a clarificar el flujo de mensajes y la distribución de responsabilidades dentro del sistema.

Subscriber

En la capa de Presentación, dentro de la clase Program, se suscribe a los eventos que serán generados por el publicador. En la capa de Dominio, se manejan los eventos enviados por el publicador y se realiza la llamada al servicio encargado de procesar la tarea en segundo plano.

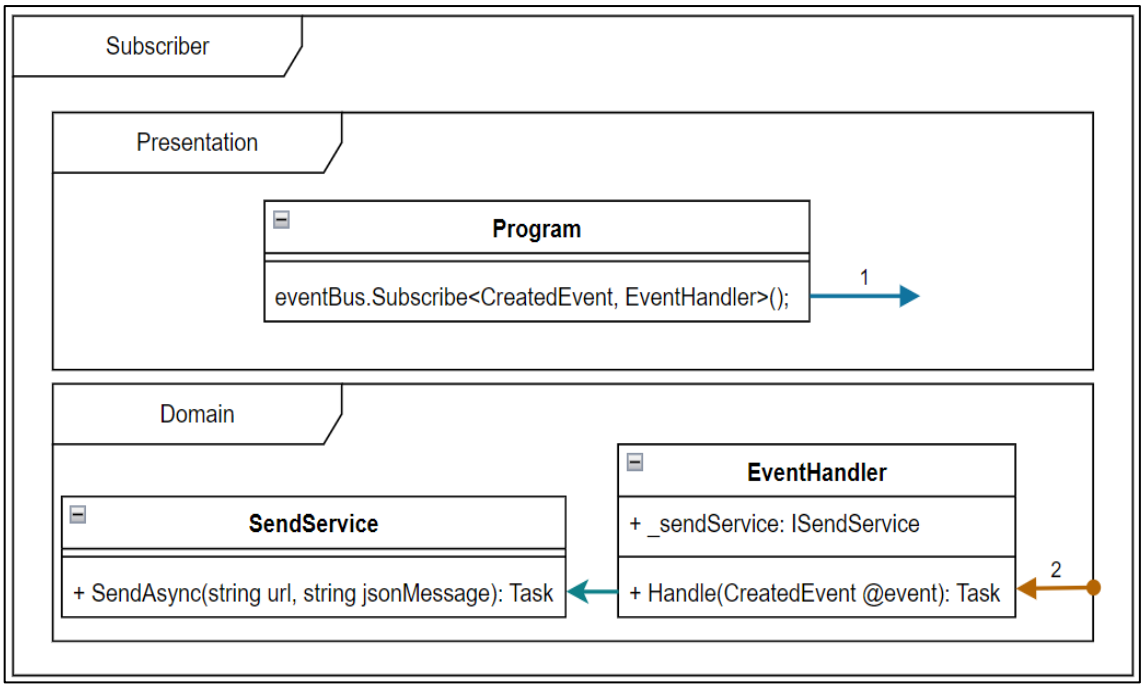


Ilustración 10 Diagrama UML Subscriber

Publisher

En la capa de Presentación, el controlador Publisher transmite el mensaje al suscriptor mediante el bus de eventos y selecciona la tecnología a utilizar. Por defecto, se emplea RabbitMQ, aunque cualquier tecnología configurada e implementada en la capa de infraestructura es igualmente válida. La entrega del mensaje ocurre en la capa de aplicación del servicio, que es responsable de crear el comando. A través del patrón de diseño Mediator, se gestiona este comando, lo que permite la creación y envío del evento al bus de eventos para su posterior publicación.

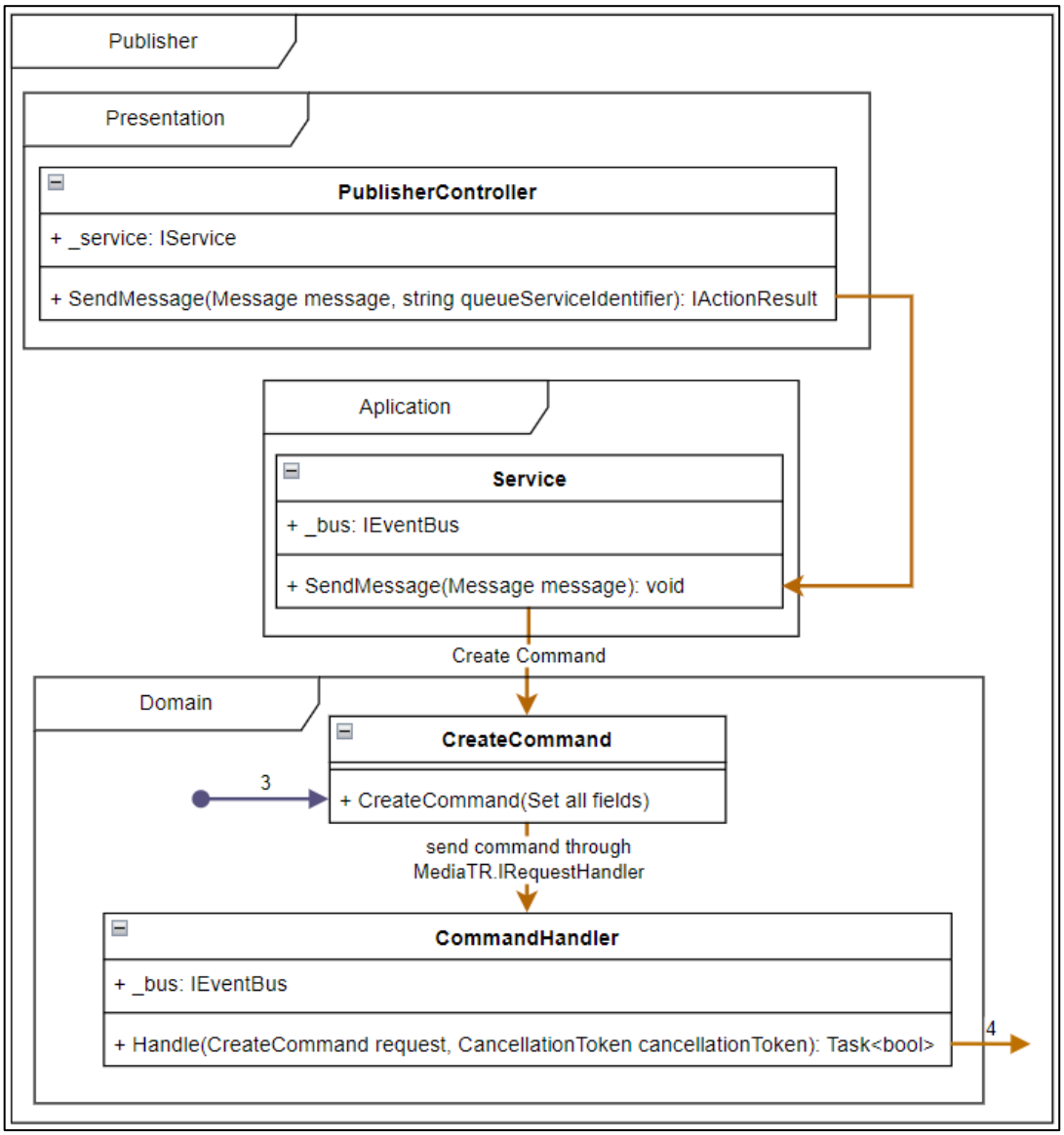


Ilustración 11 Diagrama UML Publisher

Infraestructura - Event Bus

El proceso más crítico es la implementación del bus de eventos, esencial para el sistema. Se comienza inyectando este bus en la clase DependencyContainer mediante el método RegisterServices, aplicando el principio de Inversión de Control (IoC) adicionalmente selecciona la tecnología de colas adecuada. Los comandos se transmiten usando MediatR, que activa la publicación del evento y la invocación del método Consumer_Received, donde se registran los eventos suscritos. Utilizando Reflection, se obtiene el nombre de los eventos de los modelos para nombrar las colas. En la clase Program del microservicio Subscriber se configura la suscripción a eventos, y las credenciales se establecen en appsettings.json, pasándolas a la clase MQSetting para definir las configuraciones del suscriptor y publicador. Al surgir un evento en el bus, se llama a la clase ProcessEvent, que a su vez convoca al manejador del evento para ejecutar el servicio en segundo plano, completando así el proceso.

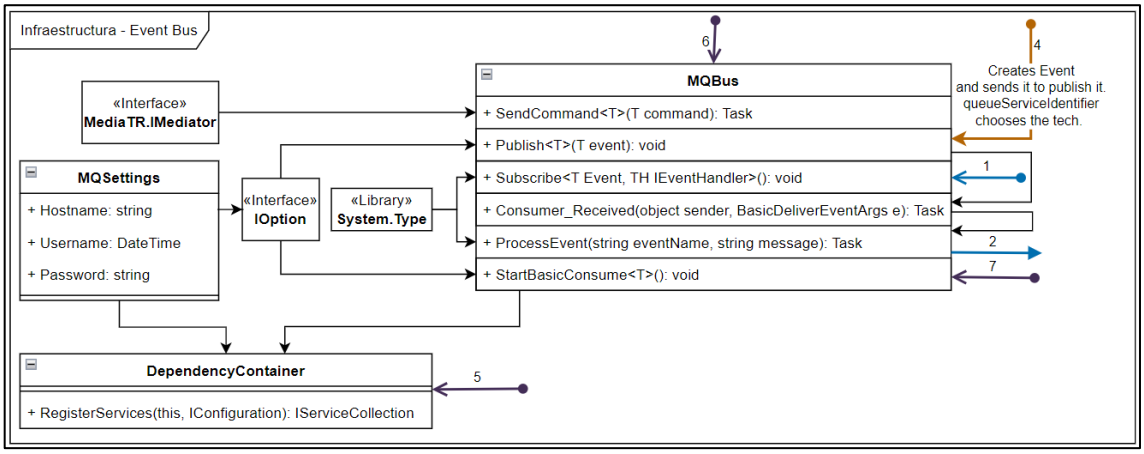


Ilustración 12 Diagrama UML Infraestructura Event Bus

Domain - Event Bus

En la capa de Dominio del bus de eventos, se encuentran todas las clases base que serán implementadas en el sistema. De esta manera, se completa la arquitectura del manejador de eventos, lo que permite una división clara de responsabilidades y tecnologías acorde con los principios de la arquitectura limpia.

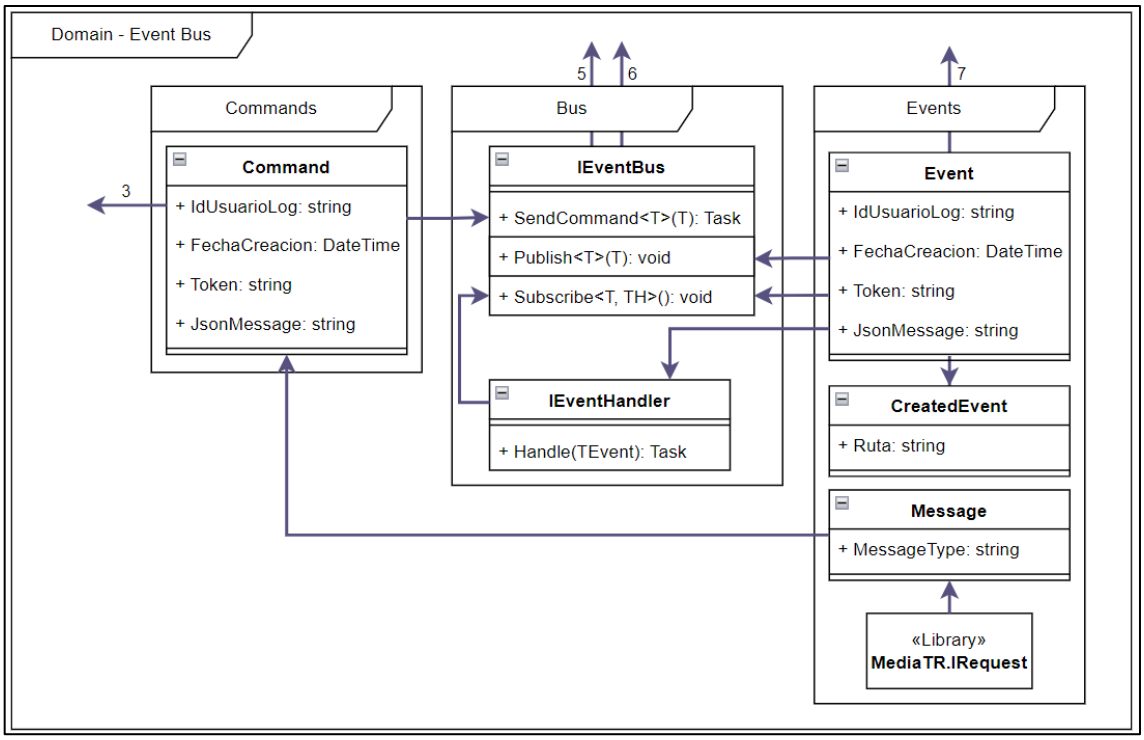


Ilustración 13 Diagrama UML Domain Event Bus

5.1.5 Diagrama Flujo

Los diagramas de flujo representan la relación de los microservicios y las colas, definen claramente los procesos y las interacciones entre los servicios. Con estos diagramas se facilita la comprensión de los flujos de trabajo.

Se presenta a continuación el flujo operativo de los microservicios implementados, destacando dos alternativas fundamentales basadas en la dirección especificada en el mensaje que se procede a encolar. En la primera alternativa, si la dirección enviada coincide con la dirección de origen, el proceso que se encola se ejecutará en un endpoint distinto dentro de la misma API de origen.

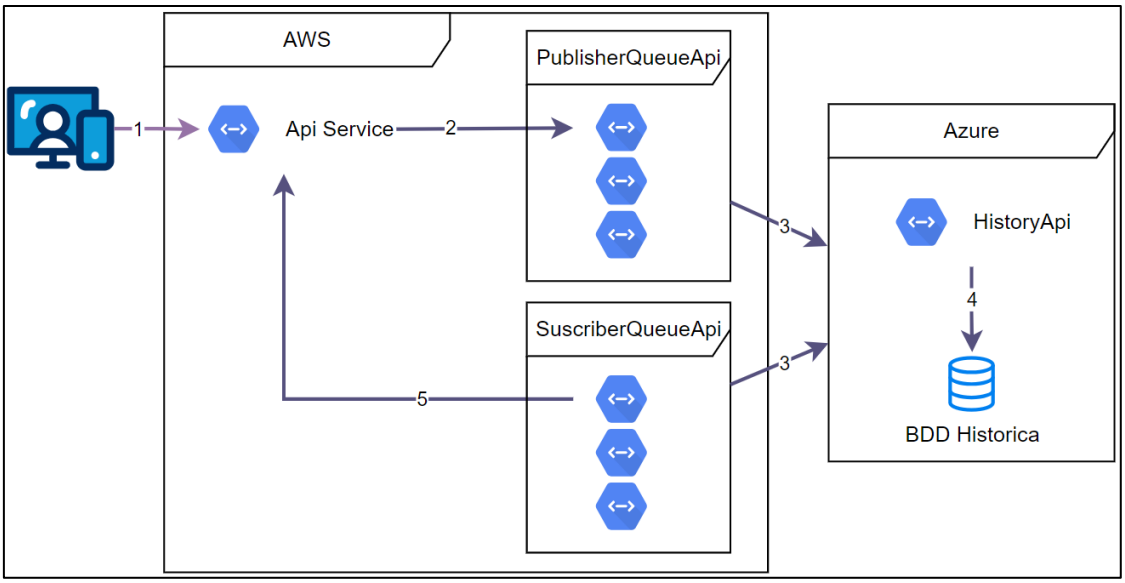


Ilustración 14 Diagrama de Flujo mismo destino

Por otro lado, en la segunda alternativa, si la dirección proporcionada difiere de la dirección de origen, el proceso en cuestión se encolará en un microservicio distinto al de origen.

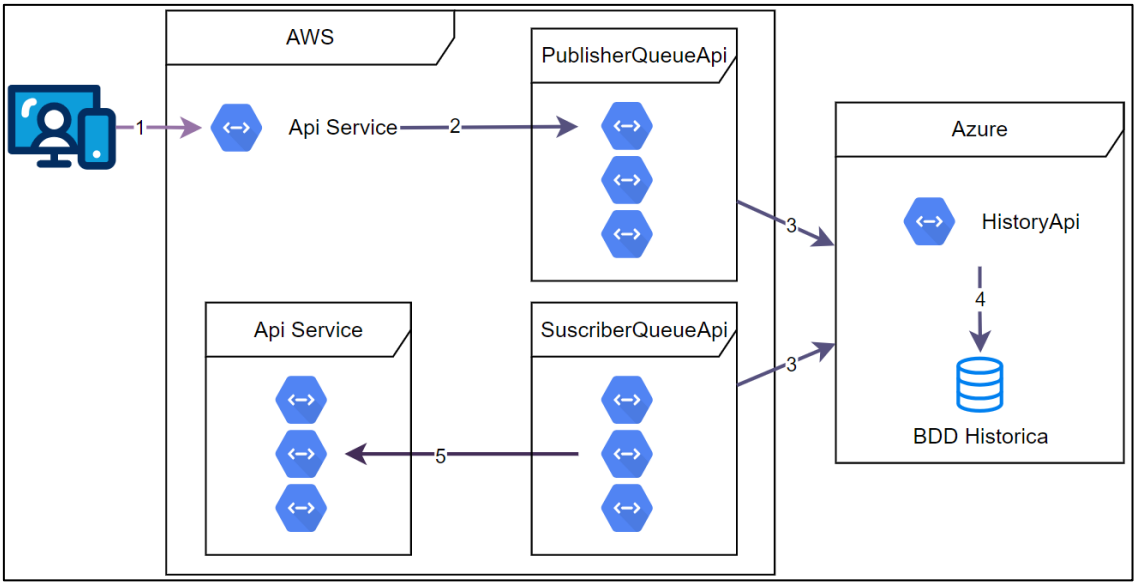


Ilustración 15 Diagrama de Flujo diferente destino

5.2 Partes clave del código

En el proyecto, se adoptaron los principios de la Arquitectura Limpia, priorizando la separación de responsabilidades y la independencia de marcos de trabajo y tecnologías específicas. La Arquitectura Limpia fomenta el desarrollo de sistemas modulares y mantenibles, con componentes de responsabilidades bien definidas y acoplamiento flexible. El proyecto se estructura en tres bibliotecas de clases principales: Domain.Core, Infra.Bus e Infra.IoC, que se alinean con la implementación de un bus de eventos siguiendo la arquitectura orientada a eventos.

La biblioteca Domain.Core alberga las entidades, objetos de valor y conceptos fundamentales del dominio de la aplicación, incluyendo comandos, eventos, mensajes y otras interfaces que representan acciones y eventos importantes en el sistema. Esta capa encapsula la lógica de negocio esencial y se mantiene aislada de cualquier tecnología de infraestructura.

Infra.Bus, por su parte, gestiona la implementación específica de la infraestructura para la comunicación mediante un bus de eventos, utilizando RabbitMQ. RabbitMQBus, encargada de la lógica para publicar y suscribirse a eventos, enviar comandos y gestionar la suscripción a eventos, utiliza RabbitMQ como mecanismo de mensajería. Esta capa interactúa con la infraestructura externa y adapta los eventos y comandos del dominio.

Finalmente, Infra.IoC ofrece métodos para el registro de dependencias del proyecto a través del contenedor de inversión de control (IoC) de .NET Core. La clase DependencyContainer juega un papel importante en el registro de servicios para la aplicación, como MediatR y la implementación del bus de eventos con RabbitMQ, facilitando la gestión de dependencias y promoviendo la modularidad y reutilización del código.

El proyecto está diseñado siguiendo los principios de la Arquitectura Limpia y la Arquitectura Orientada a Eventos. Esta estructuración promueve una clara separación de responsabilidades y dependencias entre las distintas capas y módulos

del sistema, lo que resulta en un diseño altamente flexible y mantenible. Cada componente del sistema tiene la capacidad de evolucionar de manera independiente, sin interferir con otros componentes. Además, se implementa un manejo de eventos eficiente, donde los eventos se publican y se gestionan a través de un suscriptor, permitiendo que la ejecución de procesos se realice en segundo plano y sean invocados por otros servicios según sea necesario.

A continuación, se examinarán los componentes fundamentales del proceso del bus de eventos, detallando sus clases base para entender cómo estas partes interactúan dentro del sistema y cómo contribuyen a su funcionalidad global.

Domain.Core

Dentro de la carpeta Domain se tiene una biblioteca que contiene las clases e interfaces fundamentales para el dominio de la aplicación.

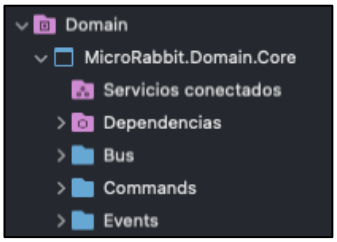


Ilustración 16 Contenido carpeta Domain

Carpeta Bus

La interfaz **IEventBus** define un contrato para un bus de eventos en el dominio. Incluye métodos para enviar comandos, publicar eventos y suscribirse a eventos. Esto proporciona una abstracción sobre la interacción entre los diversos elementos del sistema, facilitando la integración y el desacoplamiento.

IEventBus.cs

selección

```
1   using MicroQueue.Domain.Core.Commands;
2   using MicroQueue.Domain.Core.Events;
3
4   namespace MicroQueue.Domain.Core.Bus
5   {
6       public interface  IEventBus
7       {
8           Task SendCommand<T>(T command) where T : Command;
9
10          void Publish<T>(T @event) where T : Event;
11
12          void Subscribe<T, TH>()
13              where T : Event
14              where TH : IEventHandler<T>;
15
16      }
17  }
18
```

Ilustración 17 Interfaz IEventBus

IEventHandler<TEvent> es otra interfaz que define un contrato para los manejadores de eventos en el sistema. Es una interfaz genérica que obliga a las clases que la implementan a manejar eventos específicos que heredan de la clase base Event.

IEventHandler.cs

n selección

```
1   using MicroQueue.Domain.Core.Events;
2
3   namespace MicroQueue.Domain.Core.Bus
4   {
5       public interface IEventHandler<in TEvent> : IEventHandler
6           where TEvent : Event
7       {
8           Task Handle(TEvent @event);
9       }
10
11      public interface IEventHandler { }
12  }
13
```

Ilustración 18 Interfaz IEventHandler

Carpeta Commands

La clase abstracta Command sirve como representación de un comando dentro del dominio del bus de eventos. Esta clase incluye atributos esenciales tales como el identificador único del usuario, la fecha en que fue creado el comando, un token de

seguridad y un mensaje en formato JSON. Además, se especifica la ruta del proceso que será encolado y el tipo de tecnología de cola utilizada, información que se incluye en el mensaje. Heredando de la clase Message, la clase Command adquiere la capacidad de determinar el tipo de comando a encolar. Al implementar MediaTR, asegura que el comando sea efectivamente enviado a la cola correspondiente.

```
using MicroQueue.Domain.Core.Events;

namespace MicroQueue.Domain.Core.Commands
{
    6 references
    public abstract class Command : Message
    {
        6 references
        public string IdUsuarioLog { get; protected set; } = string.Empty;
        1 reference
        public DateTime FechaCreacion { get; protected set; }
        4 references
        public string Token { get; protected set; } = string.Empty;
        6 references
        public string JsonMessage { get; protected set; } = string.Empty;
        2 references
        public string QueueServiceIdentifier { get; protected set; } = string.Empty;
        2 references
        public string Ruta { get; protected set; } = string.Empty;

        0 references
        protected Command()
        {
            FechaCreacion = DateTime.Now;
        }
    }
}
```

Ilustración 19 Clase Abstracta Command

La clase **Message** es una abstracción que representa un mensaje en el dominio. Implementa la interfaz **IRequest<bool>** de MediatR y proporciona una estructura común para los mensajes que se envían entre diferentes partes del sistema.

```
using MediatR;

namespace MicroQueue.Domain.Core.Events
{
    2 references
    public abstract class Message : IRequest<bool>
    {
        1 reference
        public string MessageType { get; protected set; }

        0 references
        protected Message()
        {
            MessageType = GetType().Name;
        }
    }
}
```

Ilustración 20 Clase abstracta Message

Carpeta Events

La clase abstracta Event sirve como representación de un evento dentro del dominio. Esta clase alberga las propiedades de un Comando, que, al ser emitido, se convierte en un evento. La instanciación de un evento específico se lleva a cabo en la clase CreatedEvent, donde se asignan las propiedades pertinentes.

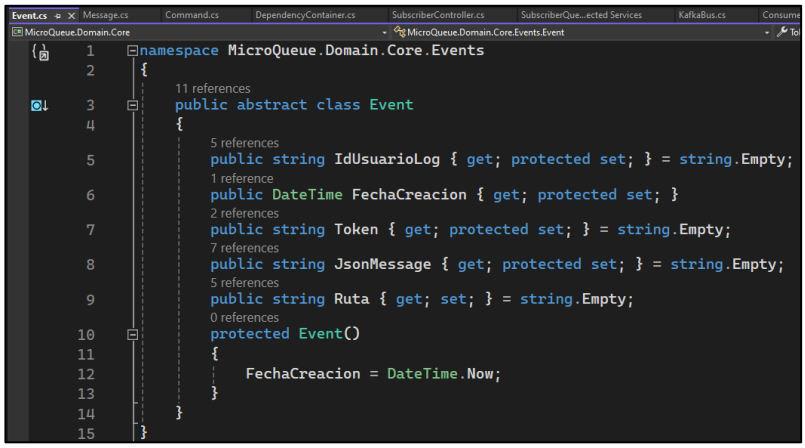


Ilustración 21 Clase Abstracta Event

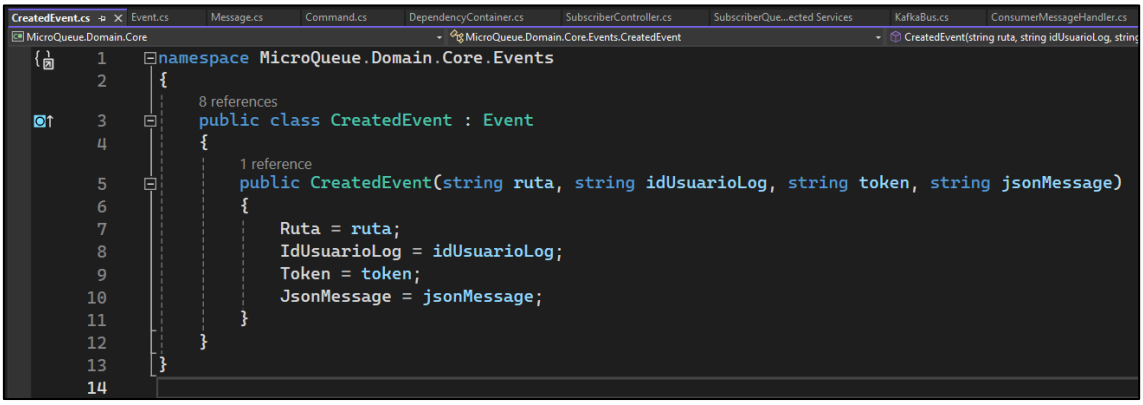


Ilustración 22 Clase CreatedEvent

Carpeta Histórico

HistoricoDTO es un DTO (Data Transfer Object) que representa un objeto de historial en el sistema. Contiene propiedades para detalles como el ID del usuario, la fecha de creación, una descripción, un tipo de evento y un mensaje. Este tipo de objetos se utilizan para transferir datos entre diferentes capas del sistema.

HistoricoDTO.cs

```
namespace MicroQueue.Domain.Core.Historico
{
    public class HistoricoDTO
    {
        public HistoricoDTO()
        {
            FechaCreacion = DateTime.Now;
        }

        public string? IdUsuario { get; set; }
        public Guid? IdRelacionGuid { get; set; }
        public string? IdRelacionVarchar { get; set; }
        public DateTime FechaCreacion { get; set; }
        public string? Descripcion { get; set; }
        public int TipoEvento { get; set; }
        public string? Mensaje { get; set; }
    }

    public class TipoEvento
    {
        public const int CreateEmailQueue = 5;
        public const int CreateCommonQueue = 6;
        public const int ConsumerEmailQueue = 7;
        public const int ConsumerCommonQueue = 8;
    }

    public static class HistoricoServices
    {
        public const string ApiLogsAlliance = "ApiLogsAlliance";
        public const string HeaderLogsAlliance = "Api-Key";
    }

    public class HistoricoSettings
    {
        public string URL { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }
}
```

Ilustración 23 Clase HistoricoDTO

La clase HistoricoExtensions proporciona un método de extensión para IServiceCollection, que se utiliza para configurar el cliente HTTP para el servicio de historial.

HistoricoExtensions.cs

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace MicroQueue.Domain.Core.Historico
{
    public static class HistoricoExtensions
    {
        public static void AddHistoricoClient(this IServiceCollection services, IConfiguration configuration)
        {
            services.AddHttpClient(HistoricoServices.ApiLogsAlliance, config =>
            {
                HistoricoSettings? settings = configuration.GetSection("HistoricoSettings").Get<HistoricoSettings>();

                config.BaseAddress = new Uri($"{settings?.URL}");
                config.DefaultRequestHeaders.Add(HistoricoServices.HeaderLogsAlliance, settings?.ApiKey);
            });
        }
    }
}
```

Ilustración 24 Clase estática HistoricoExtensions

IServicioHistorico es una interfaz que define un contrato para un servicio de historial en el sistema. Contiene un método para crear un historial, proporcionando una abstracción sobre la implementación concreta del servicio.

```cs
namespace MicroQueue.Domain.Core.Historico
{
    public interface IServicioHistorico
    {
        Task<HistoricoDTO> CrearHistoricoAsync(HistoricoDTO historico);
    }
}
```

Ilustración 25 Interfaz IServicioHistorico

La clase ServicioHistorico implementa la interfaz IServicioHistorico y ofrece la capacidad de registrar un historial de eventos o transacciones. Esta clase envia peticiones HTTP a un microservicio dedicado a la gestión de históricos, lo que permite una integración eficiente con sistemas de almacenamiento de datos o servicios de backend especializados en el seguimiento de la actividad histórica.

```cs
using Newtonsoft.Json;
using System.Net;
using System.Text;

namespace MicroQueue.Domain.Core.Historico
{
    public class ServicioHistorico : IServicioHistorico
    {
        private readonly IHttpClientFactory _httpClientFactory;

        public ServicioHistorico(IHttpClientFactory httpClientFactory)
        {
            _httpClientFactory = httpClientFactory;
        }

        public async Task<HistoricoDTO?> CrearHistoricoAsync(HistoricoDTO historico)
        {
            try
            {
                HttpClient cliente = _httpClientFactory.CreateClient(HistoricoServices.ApiLogsAlliance);

                HistoricoDTO completedHistoric = new()
                {
                    IdUsuario = historico.IdUsuario,
                    IdRelacionGuid = historico.IdRelacionGuid,
                    IdRelacionVarchar = historico.IdRelacionVarchar,
                    FechaCreacion = historico.FechaCreacion,
                    Descripcion = historico.Descripcion,
                    TipoEvento = historico.TipoEvento,
                    Mensaje = historico.Mensaje
                };

                var jsonHistorico = JsonConvert.SerializeObject(completedHistoric);
                var httpHistoric = new StringContent(jsonHistorico, Encoding.UTF8, "application/json");
                using HttpResponseMessage response = await cliente.PostAsync("api/Historico", httpHistoric);
                switch (response.StatusCode)
                {
                    case HttpStatusCode.Created:
                        var responseSuccess = await response.Content.ReadAsStringAsync();
                        HistoricoDTO responseSerialized = JsonConvert.DeserializeObject<HistoricoDTO>(responseSuccess);
                        return responseSerialized;
                    case HttpStatusCode.Unauthorized:
                        throw new Exception(HttpStatusCode.Unauthorized.ToString());
                    default:
                        throw new Exception();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                return null;
            }
        }

    }
}
```

Ilustración 26 Clase ServicioHistorico

Carpeta MessageModels

La clase Message representa los mensajes que se envían inicialmente y que se desean encolar para un procesamiento en segundo plano. A lo largo del flujo del proceso, estos mensajes se transforman en comandos mediante la adición de un tipo en sus atributos. Finalmente, se convierten en eventos.

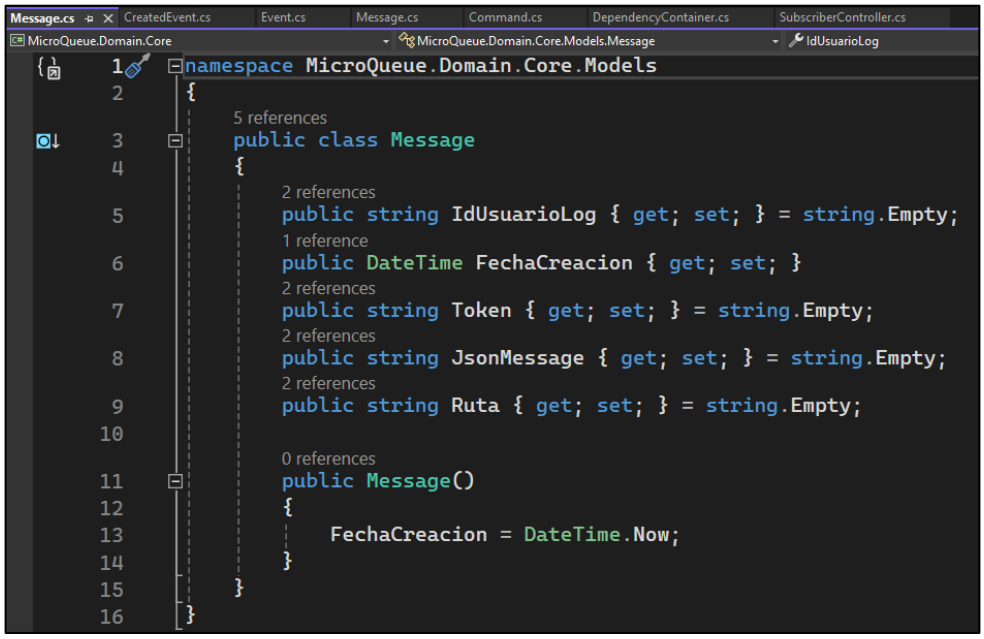


Ilustración 27 Clase Message

Infra.Bus

Esta biblioteca implementa la infraestructura para la comunicación a través de un bus de eventos utilizando RabbitMQ.

Ilustración 28 Contenido de la carpeta Infra.Bus

La clase **RabbitMQBus** proporciona una implementación de un bus de eventos utilizando RabbitMQ en un entorno de microservicios. En su constructor, recibe instancias de **IMediator**, **IServiceScopeFactory**, y **IOptions<RabbitMQSettings>**, necesarios para la inyección de dependencias y obtener la configuración de RabbitMQ.

El método **Publish<T>** permite la publicación de eventos en RabbitMQ. Aquí, el evento se serializa a formato JSON y se publica en RabbitMQ.

RabbitMQBus.cs

elección

```
using MediatR;
using MicroQueue.Domain.Core.Bus;
using MicroQueue.Domain.Core.Commands;
using MicroQueue.Domain.Core.Events;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using Newtonsoft.Json;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;

namespace MicroQueue.Infra.Bus
{
    public class RabbitMQBus : IEventBus
    {
        private readonly RabbitMQSettings _rabbitMQSettings;
        private readonly IMediator _mediator;
        private readonly Dictionary<string, List<Type>> _handlers;
        private readonly List<Type> _eventTypes;
        private readonly IServiceScopeFactory _serviceScopeFactory;

        public RabbitMQBus(IMediator mediator, IServiceScopeFactory serviceScopeFactory, IOptions<RabbitMQSettings> rabbitMQSettings)
        {
            _mediator = mediator;
            _serviceScopeFactory = serviceScopeFactory;
            _handlers = new Dictionary<string, List<Type>>();
            _eventTypes = new List<Type>();
            _rabbitMQSettings = rabbitMQSettings.Value;
        }

        public void Publish<T>(T @event) where T : Event
        {
            var factory = new ConnectionFactory
            {
                HostName = _rabbitMQSettings.Hostname,
                UserName = _rabbitMQSettings.Username,
                Password = _rabbitMQSettings.Password
            };

            using (var connection = factory.CreateConnection())
            using (var channel = connection.CreateModel())
            {

                var eventName = @event.GetType().Name;

                channel.QueueDeclare(eventName, false, false, false, null);

                var message = JsonConvert.SerializeObject(@event);

                var body = Encoding.UTF8.GetBytes(message);

                channel.BasicPublish("", eventName, null, body);

            }
        }
```

Ilustración 29 Clase RabbitMQBus

SendCommand<T> envía comandos utilizando MediatR, llamando al método Send de **IMediator**. El método **Subscribe<T, TH>** se utiliza para suscribirse a un tipo específico de evento y asociarlo con un manejador. Registra el tipo de evento y su

manejador correspondiente en una lista interna y comienza a consumir mensajes para ese evento de RabbitMQ.

```
public Task SendCommand<T>(T command) where T : Command
{
    return _mediator.Send(command);
}

public void Subscribe<T, TH>()
    where T : Event
    where TH : IEventHandler<T>
{
    var eventName = typeof(T).Name;
    var handlerType = typeof(TH);

    if (!_eventTypes.Contains(typeof(T)))
    {
        _eventTypes.Add(typeof(T));
    }

    if (!_handlers.ContainsKey(eventName))
    {
        _handlers.Add(eventName, new List<Type>());
    }

    if (_handlers[eventName].Any(s => s.GetType() == handlerType))
    {
        throw new ArgumentException($"El handler exception {handlerType.Name} ya fue registrado anteriormente por '{eventName}'", nameof(handlerType));
    }

    _handlers[eventName].Add(handlerType);

    StartBasicConsume<T>();

}
```

Ilustración 30 Método SendCommand y Subscribe

StartBasicConsume<T> inicia el consumo básico de mensajes de RabbitMQ para un tipo específico de evento, configurando un consumidor asíncrono para procesar los mensajes entrantes.

```
private void StartBasicConsume<T>() where T : Event
{
    var factory = new ConnectionFactory
    {
        HostName = _rabbitMQSettings.Hostname,
        UserName = _rabbitMQSettings.Username,
        Password = _rabbitMQSettings.Password,
        DispatchConsumersAsync = true
    };

    var connection = factory.CreateConnection();
    var channel = connection.CreateModel();

    var eventName = typeof(T).Name;

    channel.QueueDeclare(eventName, false, false, false, null);

    var consumer = new AsyncEventingBasicConsumer(channel);

    consumer.Received += Consumer_Received;

    channel.BasicConsume(eventName, true, consumer);

}
```

Ilustración 31 Método StartBasicConsume

Cuando se recibe un mensaje, **Consumer_Received** maneja su procesamiento, extrayendo el nombre del evento y el mensaje en formato JSON del cuerpo del mensaje y llamando a ProcessEvent para su procesamiento.

```
private async Task Consumer_Received(object sender, BasicDeliverEventArgs e)
{
    var eventName = e.RoutingKey;
    var message = Encoding.UTF8.GetString(e.Body.Span);

    try
    {
        await ProcessEvent(eventName, message).ConfigureAwait(false);
    }
    catch (Exception ex)
    {

    }
}
```

Ilustración 32 Método Consumer_Received

ProcessEvent procesa el evento recibido, encontrando los manejadores asociados con el evento y los invoca para manejar el evento.

```
private async Task ProcessEvent(string eventName, string message)
{
    if (_handlers.ContainsKey(eventName))
    {
        using (var scope = _serviceScopeFactory.CreateScope())
        {
            var subscriptions = _handlers[eventName];

            foreach (var subscription in subscriptions)
            {
                var handler = scope.ServiceProvider.GetService(subscription);
                if (handler == null) continue;
                var eventType = _eventTypes.SingleOrDefault(t => t.Name == eventName);
                var @event = JsonConvert.DeserializeObject(message, eventType);
                var concreteType = typeof(IEventHandler<>).MakeGenericType(eventType);

                await (Task)concreteType.GetMethod("Handle").Invoke(handler, new object[] { @event });

            }
        }
    }
}
```

Ilustración 33 Método ProcessEvent

La clase **RabbitMQSettings** simplemente contiene la configuración necesaria para conectar y autenticarse con RabbitMQ. Esto incluye la dirección del host, el nombre de usuario, y la contraseña. Estas propiedades se establecen utilizando las opciones proporcionadas en la configuración de la aplicación.

RabbitMQSettings.cs

lección

```
1   namespace MicroQueue.Infra.Bus
2   {
3       public class RabbitMQSettings
4       {
5           public string Hostname { get; set; } = string.Empty;
6           public string Username { get; set; } = string.Empty;
7           public string Password { get; set; } = string.Empty;
8       }
9   }
```

Ilustración 34 Clase RabbitMQSettings

Infra.IoC

Esta biblioteca proporciona métodos para registrar las dependencias del proyecto utilizando Microsoft.Extensions.DependencyInjection.

Ilustración 35 Contenido de la carpeta Infra.IoC

La clase DependencyContainer en la biblioteca de clases Infra.IoC proporciona un conjunto de métodos estáticos para registrar las dependencias del proyecto utilizando Microsoft.Extensions.DependencyInjection. En particular, el método RegisterServices registra los servicios necesarios para la aplicación. Primero, utiliza services.AddMediatR para registrar los servicios de MediatR, que facilitan la mediación de solicitudes y respuestas en el proyecto. Luego, configura el servicio del bus del dominio (IEventBus) utilizando RabbitMQ como implementación concreta. Esto se hace mediante services.AddSingleton, donde se pasa una fábrica de instancias personalizada que crea una nueva instancia de RabbitMQBus con las dependencias requeridas, como el proveedor de alcance de servicios (IServiceScopeFactory) y la configuración de RabbitMQ (RabbitMQSettings). Finalmente, devuelve la colección de servicios registrados para que puedan ser utilizados en el resto de la aplicación. En resumen, esta clase facilita la configuración

y registro de las dependencias del proyecto, continuando con los principios de la arquitectura limpia y la inyección de dependencias para los servicios.

DependencyContainer.cs

elección

```
using MediatR;
using MicroQueue.Domain.Core.Bus;
using MicroQueue.Infra.Bus;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using System.Reflection;

namespace MicroQueue.Infra.IoC
{
    public static class DependencyContainer
    {
        public static IServiceCollection RegisterServices(this IServiceCollection services, IConfiguration configuration)
        {
            //MediatR Mediator
            services.AddMediatR(Assembly.GetExecutingAssembly());

            //Domain Bus
            services.AddSingleton<IEventBus, RabbitMQBus>(sp => {
                var scopeFactory = sp.GetRequiredService<IServiceScopeFactory>();
                var optionsFactory = sp.GetService<IOptions<RabbitMQSettings>>();
                return new RabbitMQBus(sp.GetService<IMediator>(), scopeFactory, optionsFactory );
            });

            return services;
        }

    }
}
```

Ilustración 36 Clase estática DependencyContainer

5.3 Link del repositorio

https://github.com/DoomLuchin/MicroservicioRabbitMQ.git

5.4 Pruebas Funcionales

Las pruebas funcionales son esenciales para garantizar que los procesos encolados funcionen correctamente bajo condiciones similares a la producción. Utilizando Postman, se pueden simular solicitudes de servicios en un entorno local con múltiples usuarios, en este caso, 100 usuarios recurrentes, para evaluar el rendimiento y la fiabilidad del sistema antes de su despliegue. Esto ayuda a identificar y corregir problemas potenciales de manera proactiva.

Pasos:

1. Se desarrolló una colección en Postman diseñada para analizar el procedimiento de encolamiento. El endpoint de prueba, denominado queueProcess, requiere como parámetro la tecnología específica a emplear, que para este escenario es RabbitMQ, así como la ruta donde se ejecutará el proceso solicitado por el consumidor. Dicho endpoint activará al Publisher, y el evento generado será interceptado por el suscriptor. Finalmente, este suscriptor procederá a consumir los mensajes en la cola y efectuará una llamada a la ruta de la API encargada de llevar a cabo el proceso en cuestión.

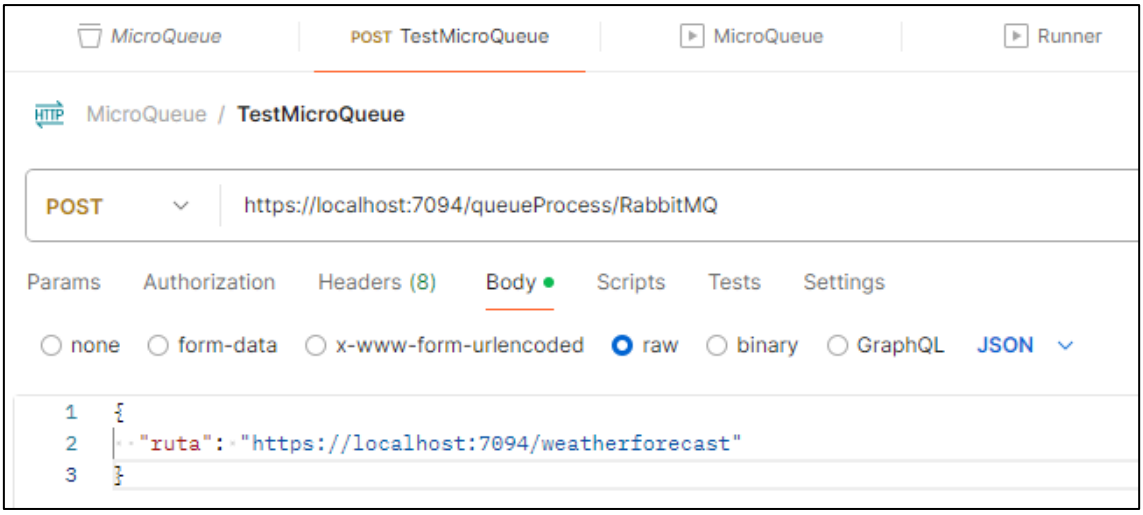


Ilustración 37 Llamada POST al api de pruebas con RabbitMQ

2. Se configuran las pruebas funcionales en postman.

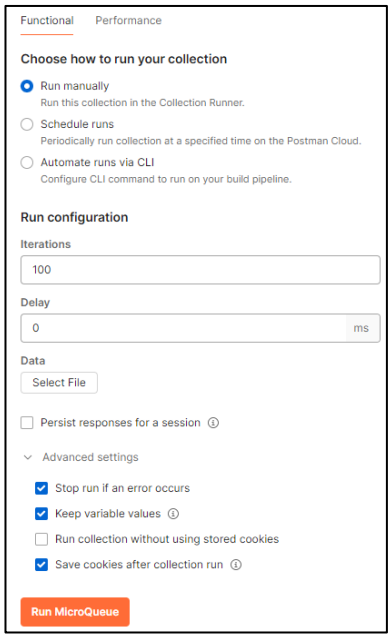


Ilustración 38 Configuración para pruebas funcionales en Postman

3. Se ejecutó las pruebas y se verificó el funcionamiento correcto. Se crean correctamente las 100 iteraciones.

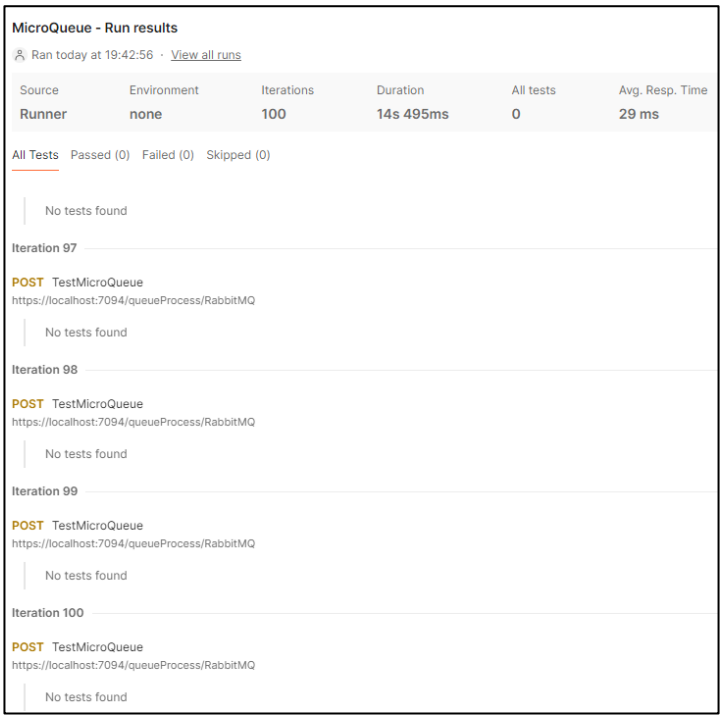


Ilustración 39 Pruebas ejecutadas

4. Se observa que las colas se crearon correctamente.

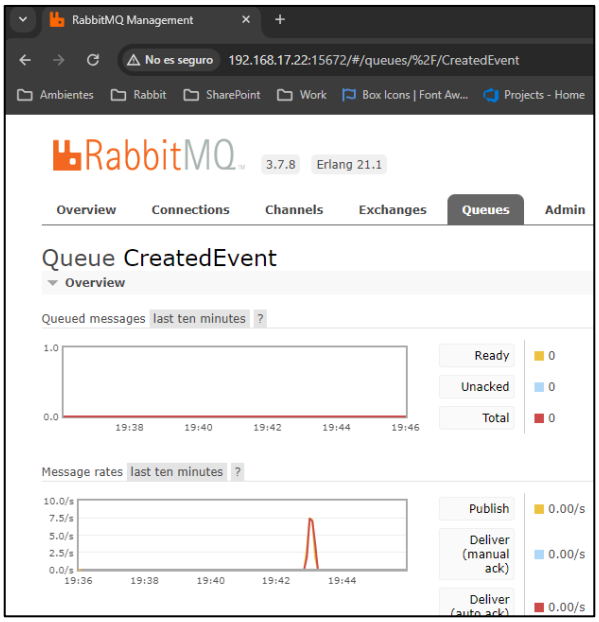


Ilustración 40 Consola RabbitMQ

5. El consumidor llama correctamente al servicio api que hace el proceso.

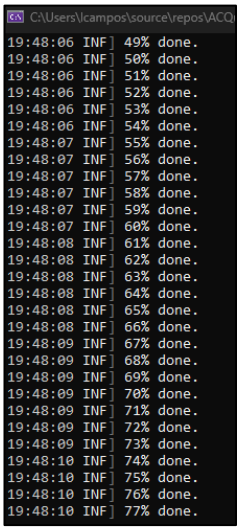
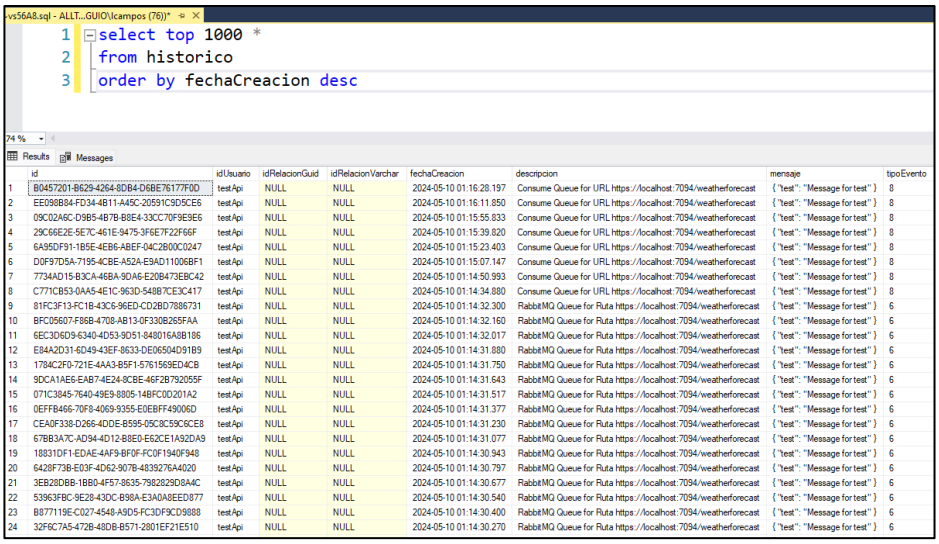


Ilustración 41 Consumidor llama al API correspondiente

6. Los registros históricos de la cola se crean exitosamente



```
select top 1000 *
from historico
order by fechaCreacion desc
```

74 %

Results | Messages

	id	idUsuario	idRelacionGuid	idRelacionVarchar	fechaCreacion	descripcion	mensaje	tipoEvento
1	B0457201-B629-4264-8DB4-D6BE76177F0D	testApi	NULL	NULL	2024-05-10 01:16:28.197	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
2	EE098B84-FD34-4B11-A45C-20591C9D5CE6	testApi	NULL	NULL	2024-05-10 01:16:11.850	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
3	09C02A6C-D9B5-4B7B-B8E4-33CC70F9E9E6	testApi	NULL	NULL	2024-05-10 01:15:55.833	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
4	29C66E2E-5E7C-461E-9475-3F6E7F22F66F	testApi	NULL	NULL	2024-05-10 01:15:39.820	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
5	6A95DF91-1B5E-4EB6-ABEF-04C2B00C0247	testApi	NULL	NULL	2024-05-10 01:15:23.403	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
6	D0F97D5A-7195-4CBE-A52A-E9AD11006BF1	testApi	NULL	NULL	2024-05-10 01:15:07.147	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
7	7734AD15-B3CA-46BA-9DA6-E20B473EBC42	testApi	NULL	NULL	2024-05-10 01:14:50.993	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
8	C771CB53-0AA5-4E1C-963D-548B7CE3C417	testApi	NULL	NULL	2024-05-10 01:14:34.880	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
9	81FC3F13-FC1B-43C6-96ED-CD2BD7886731	testApi	NULL	NULL	2024-05-10 01:14:32.300	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
10	BFC05607-F86B-4708-AB13-0F330B265FAA	testApi	NULL	NULL	2024-05-10 01:14:32.160	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
11	6EC3D6D9-6340-4D53-9D51-848016A8B186	testApi	NULL	NULL	2024-05-10 01:14:32.017	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
12	E84A2D31-6D49-43EF-8633-DE06504D91B9	testApi	NULL	NULL	2024-05-10 01:14:31.880	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
13	1784C2F0-721E-4AA3-B5F1-5761569ED4CB	testApi	NULL	NULL	2024-05-10 01:14:31.750	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
14	9DCA1AE6-EAB7-4E24-8CBE-46F2B792055F	testApi	NULL	NULL	2024-05-10 01:14:31.643	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
15	071C3845-7640-49E9-8805-14BFC0D201A2	testApi	NULL	NULL	2024-05-10 01:14:31.517	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
16	0EFFB466-70F8-4069-9355-E0EBFF49006D	testApi	NULL	NULL	2024-05-10 01:14:31.377	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
17	CEA0F338-D266-4DDE-B595-05C8C59C6CE8	testApi	NULL	NULL	2024-05-10 01:14:31.230	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
18	67BB3A7C-AD94-4D12-B8E0-E62CE1A92DA9	testApi	NULL	NULL	2024-05-10 01:14:31.077	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
19	18831DF1-EDAE-4AF9-BF0F-FC0F1940F948	testApi	NULL	NULL	2024-05-10 01:14:30.943	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
20	6428F73B-E03F-4D62-907B-4839276A4020	testApi	NULL	NULL	2024-05-10 01:14:30.797	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
21	3EB28DBB-1BB0-4F57-8635-7982829D8A4C	testApi	NULL	NULL	2024-05-10 01:14:30.677	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
22	53963FBC-9E28-43DC-B98A-E3A0A8EED877	testApi	NULL	NULL	2024-05-10 01:14:30.540	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
23	B877119E-C027-4548-A9D5-FC3DF9CD9888	testApi	NULL	NULL	2024-05-10 01:14:30.400	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
24	32F6C7A5-472B-48DB-B571-2801EF21E510	testApi	NULL	NULL	2024-05-10 01:14:30.270	RabbitMQ Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6

Ilustración 42 Registros históricos de la cola RabbitMQ

7. Se arma un caso de pruebas similar, pero ahora probando con Kafka

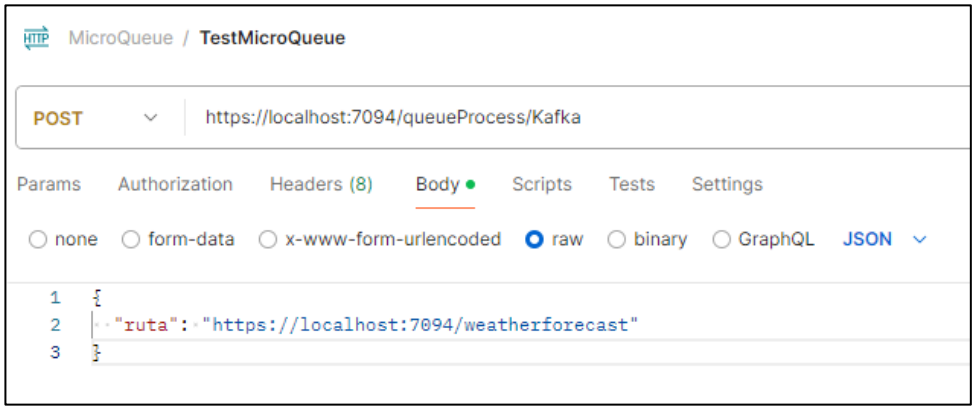
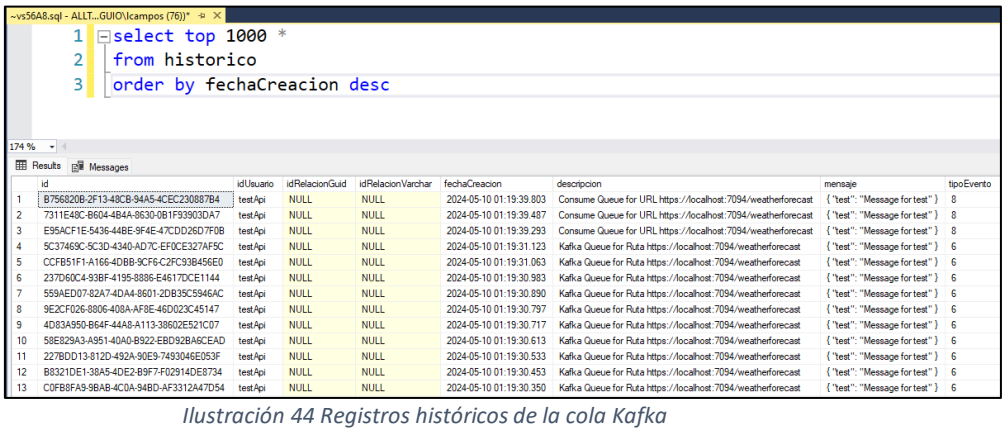


Ilustración 43 Llamada POST al api de pruebas con Kafka

8. Se evidencia el correcto envío.



```
select top 1000 *
from historico
order by fechaCreacion desc
```

174 %

Results | Messages

	id	idUsuario	idRelacionGuid	idRelacionVarchar	fechaCreacion	descripcion	mensaje	tipoEvento
1	B756820B-2F13-49CB-94A5-4CEC230887B4	testApi	NULL	NULL	2024-05-10 01:19:39.803	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
2	7311E48C-B604-4B4A-8630-0B1F93903DA7	testApi	NULL	NULL	2024-05-10 01:19:39.487	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
3	E95ACF1E-5436-44BE-9F4E-47CDD26D7F0B	testApi	NULL	NULL	2024-05-10 01:19:39.293	Consume Queue for URL https://localhost:7094/weatherforecast	{ "test": "Message for test" }	8
4	5C37469C-5C3D-4340-AD7C-EF0CE327AF5C	testApi	NULL	NULL	2024-05-10 01:19:31.123	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
5	CCFB51F1-A166-4DBB-9CF6-C2FC93B456E0	testApi	NULL	NULL	2024-05-10 01:19:31.063	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
6	237D60C4-93BF-4195-8886-E4617DCE1144	testApi	NULL	NULL	2024-05-10 01:19:30.983	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
7	559AED07-82A7-4DA4-8601-2DB35C5946AC	testApi	NULL	NULL	2024-05-10 01:19:30.890	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
8	9E2CF026-8806-408A-AF8E-46D023C45147	testApi	NULL	NULL	2024-05-10 01:19:30.797	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
9	4D83A950-B64F-44A8-A113-38602E521C07	testApi	NULL	NULL	2024-05-10 01:19:30.717	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
10	58E829A3-A951-40A0-B922-EBD92BA6CEAD	testApi	NULL	NULL	2024-05-10 01:19:30.613	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
11	227BDD13-812D-492A-90E9-7493046E053F	testApi	NULL	NULL	2024-05-10 01:19:30.533	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
12	B8321DE1-38A5-4DE2-B9F7-F02914DE8734	testApi	NULL	NULL	2024-05-10 01:19:30.453	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6
13	C0FB8FA9-9BAB-4C0A-94BD-AF3312A47D54	testApi	NULL	NULL	2024-05-10 01:19:30.350	Kafka Queue for Ruta https://localhost:7094/weatherforecast	{ "test": "Message for test" }	6

Ilustración 44 Registros históricos de la cola Kafka

5.5 Pruebas de Rendimiento

Para evaluar la funcionalidad del sistema, se implementaron pruebas de rendimiento con la participación de 100 usuarios concurrentes. Estos usuarios interactuaron con el servicio durante un período de 10 minutos, utilizando la tecnología de la cola de RabbitMQ. Este método fue crucial para simular un entorno de alta demanda, lo cual es esencial para asegurar que el servicio mantenga su estabilidad y eficiencia, incluso bajo condiciones de carga considerable.

Pasos:

1. Configuración de pruebas de rendimiento tipo Ramp up para 100 usuarios concurrentes.

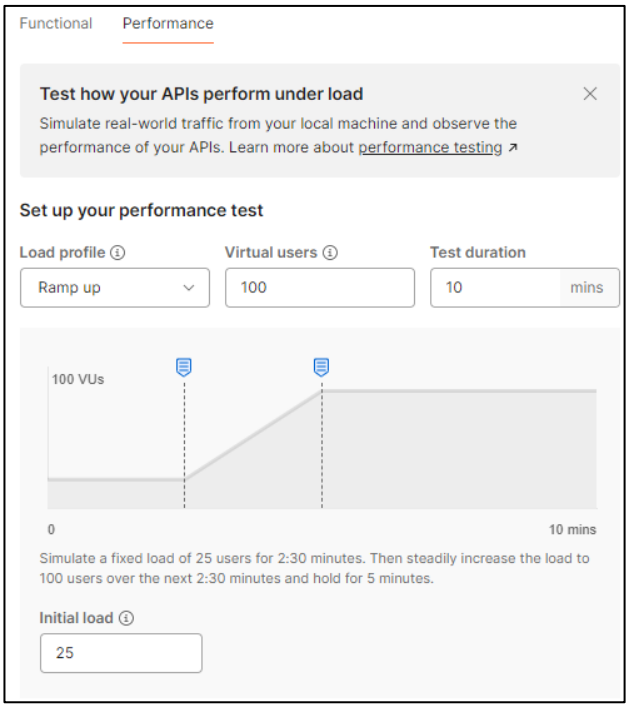


Ilustración 45 Configuración para pruebas de Rendimiento

2. Los resultados finales del Postman y la tabla históricos son correctos.

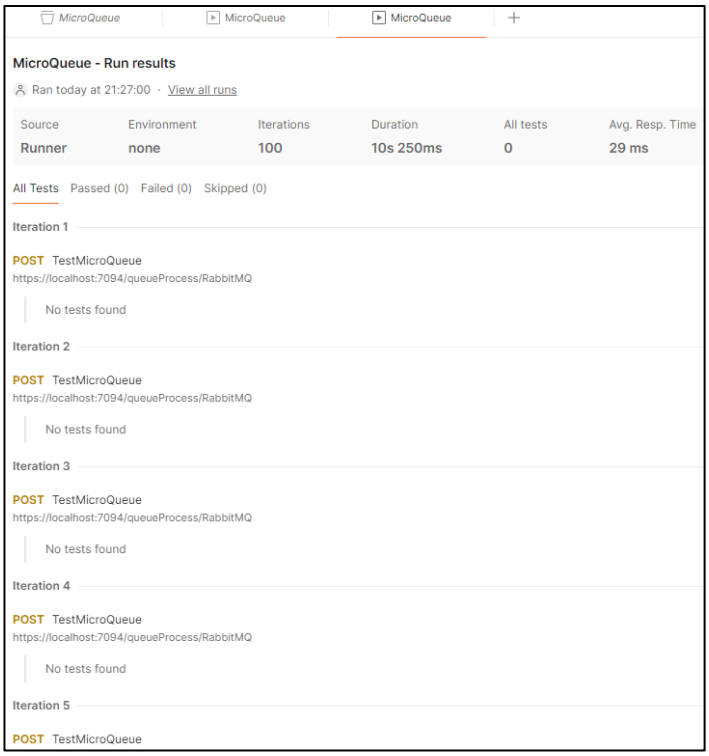
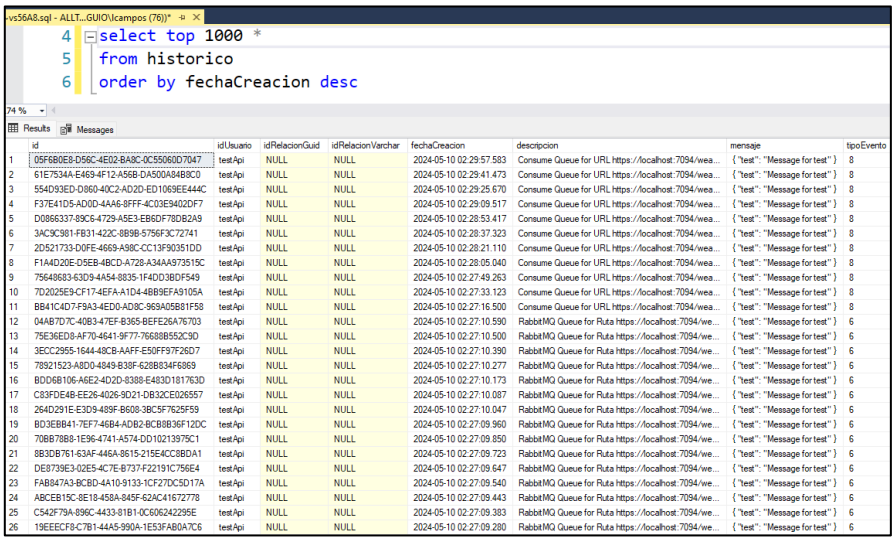


Ilustración 46 Pruebas ejecutadas

3. Los registros históricos de la cola se crean exitosamente.



```
4  select top 1000 *
5  from historico
6  order by fechaCreacion desc
```

Results | Messages

	id	idUsuario	idRelacionGuid	idRelacionVarchar	fechaCreacion	descripcion	mensaje	tipoEvento
1	05F6B0E8-D56C-4E02-BA8C-0C55060D7047	testApi	NULL	NULL	2024-05-10 02:29:57.583	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
2	61E7534A-E469-4F12-A56B-DA500A84B8C0	testApi	NULL	NULL	2024-05-10 02:29:41.473	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
3	554D93ED-D860-40C2-AD2D-ED1069EE444C	testApi	NULL	NULL	2024-05-10 02:29:25.670	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
4	F37E41D5-AD0D-4AA6-8FFF-4C03E9402DF7	testApi	NULL	NULL	2024-05-10 02:29:09.517	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
5	D0866337-89C6-4729-A5E3-EB6DF78DB2A9	testApi	NULL	NULL	2024-05-10 02:28:53.417	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
6	3AC9C981-FB31-422C-8B9B-5756F3C72741	testApi	NULL	NULL	2024-05-10 02:28:37.323	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
7	2D521733-D0FE-4669-A98C-CC13F90351DD	testApi	NULL	NULL	2024-05-10 02:28:21.110	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
8	F1A4D20E-D5EB-4BCD-A728-A34AA973515C	testApi	NULL	NULL	2024-05-10 02:28:05.040	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
9	75648683-63D9-4A54-8835-1F4DD3BDF549	testApi	NULL	NULL	2024-05-10 02:27:49.263	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
10	7D2025E9-CF17-4EFA-A1D4-4BB9EFA9105A	testApi	NULL	NULL	2024-05-10 02:27:33.123	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
11	BB41C4D7-F9A3-4ED0-AD8C-969A05B81F58	testApi	NULL	NULL	2024-05-10 02:27:16.500	Consume Queue for URL https://localhost:7094/wea...	{"test": "Message for test"}	8
12	04AB7D7C-40B3-47EF-B365-BEFE26A76703	testApi	NULL	NULL	2024-05-10 02:27:10.590	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
13	75E36ED8-AF70-4641-9F77-76688B552C9D	testApi	NULL	NULL	2024-05-10 02:27:10.500	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
14	3ECC2955-1644-48CB-AAFF-E50FF97F26D7	testApi	NULL	NULL	2024-05-10 02:27:10.390	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
15	78921523-A8D0-4849-B38F-628B834F6869	testApi	NULL	NULL	2024-05-10 02:27:10.277	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
16	BDD6B106-A6E2-4D2D-8388-E483D181763D	testApi	NULL	NULL	2024-05-10 02:27:10.173	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
17	C83FDE4B-EE26-4026-9D21-DB32CE026557	testApi	NULL	NULL	2024-05-10 02:27:10.087	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
18	264D291E-E3D9-489F-B608-3BC5F7625F59	testApi	NULL	NULL	2024-05-10 02:27:10.047	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
19	BD3EBB41-7EF7-46B4-ADB2-BCB8B36F12DC	testApi	NULL	NULL	2024-05-10 02:27:09.960	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
20	70BB78B8-1E96-4741-A574-DD10213975C1	testApi	NULL	NULL	2024-05-10 02:27:09.850	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
21	8B3DB761-63AF-446A-8615-215E4CC8BDA1	testApi	NULL	NULL	2024-05-10 02:27:09.723	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
22	DE8739E3-02E5-4C7E-B737-F22191C756E4	testApi	NULL	NULL	2024-05-10 02:27:09.647	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
23	FAB847A3-BCBD-4A10-9133-1CF27DC5D17A	testApi	NULL	NULL	2024-05-10 02:27:09.540	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
24	ABCEB15C-8E18-458A-845F-62AC41672778	testApi	NULL	NULL	2024-05-10 02:27:09.443	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
25	C542F79A-896C-4433-81B1-0C606242295E	testApi	NULL	NULL	2024-05-10 02:27:09.383	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6
26	19EEECF8-C7B1-44A5-990A-1E53FAB0A7C6	testApi	NULL	NULL	2024-05-10 02:27:09.280	RabbitMQ Queue for Ruta https://localhost:7094/we...	{"test": "Message for test"}	6

Ilustración 47 Registros Históricos

4. Se evidencian que los resultados del api de procesos son correctos.

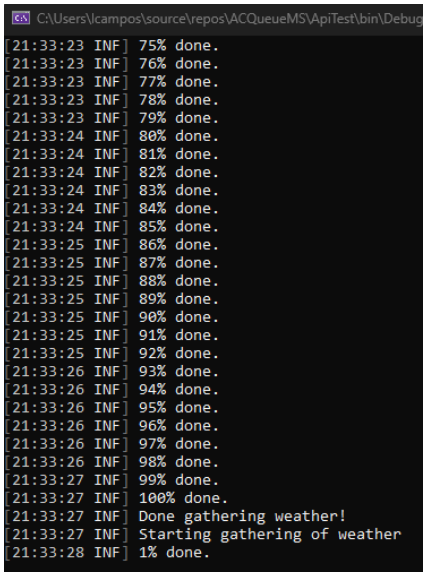


Ilustración 48 Llamada al API

5.6 Discusión

En el presente Trabajo de Titulación se eligió la arquitectura Clean Architecture con microservicios y la arquitectura basada en eventos debido a la necesidad de construir un sistema escalable, flexible y robusto. En la siguiente tabla se comparan cuatro arquitecturas para analizar y contrastar sus características; así se obtuvo la arquitectura usada en este proyecto.

Tabla 2 Cuadro comparativo de arquitecturas

Características	Microservicios	Arquitectura Monolítica	Arquitectura en Pares	Arquitectura Orientada a Servicios (SOA)
Escalabilidad	X	X		X
Independencia	X		X	X
Flexibilidad	X		X	X
Baja complejidad en el mantenimiento	X			
Tolerancia a fallos	X			
Seguridad	X			

Al comparar las arquitecturas mencionadas, los microservicios son una opción favorable en escalabilidad, independencia, flexibilidad, mantenimiento, tolerancia a fallos y seguridad.

En cuanto a escalabilidad, los microservicios permiten un escalado horizontal eficiente, lo que significa que cada servicio puede escalarse independientemente según la demanda, a diferencia de la arquitectura monolítica que se basa principalmente en el escalado vertical. En términos de independencia, los microservicios ofrecen una gran ventaja al permitir que cada componente del sistema se desarrolle y despliegue de forma independiente. Esto significa que los equipos pueden trabajar en paralelo en diferentes servicios sin afectar el resto del sistema, lo que no es posible en una arquitectura monolítica donde todos los componentes están interconectados.

La flexibilidad es otra ventaja de los microservicios, ya que los cambios en un servicio no afectan a otros servicios ni a la aplicación completa. Esto permite una mayor adaptabilidad a medida que los requisitos del negocio cambian con el tiempo. Los equipos realizan actualizaciones y correcciones rápidas sin afectar a otros componentes. La tolerancia a fallos es también un área en la que los microservicios destacan, ya que un fallo en un servicio puede ser aislado y manejado sin afectar a otros servicios. Esto mejora la estabilidad y la confiabilidad del sistema. Finalmente, en cuanto a seguridad, los microservicios ofrecen la posibilidad de implementar medidas de seguridad de manera eficiente y escalable en cada servicio, lo que mejora la protección del sistema contra posibles amenazas.

En resumen, los microservicios presentan como la mejor opción en comparación con otras arquitecturas debido a su capacidad para ofrecer escalabilidad, independencia, flexibilidad, mantenimiento, tolerancia a fallos y seguridad de manera efectiva y eficiente.

Además, en este Trabajo de Titulación se eligió la arquitectura Clean Architecture con microservicios y la arquitectura basada en eventos debido a la necesidad de construir un sistema escalable, flexible y robusto. La arquitectura Clean permite la separación clara de responsabilidades y la modularidad del sistema, lo que facilita su desarrollo, mantenimiento y evolución a lo largo del tiempo. Al adoptar este enfoque, se facilita la adaptación a los cambios en los requisitos del sistema, así como la integración con otras tecnologías y sistemas existentes. Además, Clean Architecture promueve la independencia de la tecnología, lo que permite la

sustitución de componentes como RabbitMQ en el futuro sin afectar la lógica de negocio subyacente.

Así también, se eligió RabbitMQ como la solución de mensajería para gestionar eficientemente las colas de tareas. Esta decisión fue tomada por las características y capacidades que brinda RabbitMQ que se alineaban con los objetivos del proyecto. En la siguiente tabla se comparan RabbitMq y Kafka para analizar y contrastar sus características:

Tabla 3 Cuadro comparativo de soluciones

Características	Kafka	RabbitMQ
Arquitectura	Distribuida, distribución de mensajes en particiones.	Basada en el modelo de colas de mensajes.
Escalabilidad	Altamente escalable y diseñado para manejar grandes volúmenes de datos.	Escalabilidad limitada en comparación con Kafka.
Rendimiento	Alto rendimiento y baja latencia.	Rendimiento robusto, pero puede ser inferior a Kafka en entornos de alto rendimiento.
Persistencia de mensajes	Almacena mensajes en disco para recuperación en caso de fallo.	Soporte para almacenamiento de mensajes en disco y en memoria.
Fiabilidad	Garantiza la entrega de mensajes a los consumidores y la tolerancia a fallos.	Enfoque en la entrega garantizada de mensajes y la consistencia de datos.
Facilidad de uso	Requiere una curva de aprendizaje para la configuración y administración.	Fácil de configurar y usar, especialmente para casos de uso simples.
Casos de uso	Aplicaciones en tiempo real, streaming.	Adecuado para aplicaciones que requieren entrega garantizada de mensajes y consistencia.

RabbitMQ, conocido por su facilidad de uso y su sólido soporte para la entrega garantizada de mensajes y la consistencia de datos, ofreció una solución confiable para simplificar la administración de colas de tareas. Su arquitectura basada en el modelo de colas de mensajes proporcionó una integración sencilla en conjunto con Microservicios y Clean Architecture.

Aunque RabbitMQ ofrece una escalabilidad limitada en comparación con Kafka y puede tener un rendimiento inferior en entornos de alto rendimiento, estas limitaciones no representaron obstáculos significativos para los objetivos del proyecto. Por el contrario, la facilidad de configuración y uso de RabbitMQ, junto con su capacidad para garantizar la entrega de mensajes y la tolerancia a fallos, fueron aspectos clave.

En resumen, la elección de RabbitMQ como solución de mensajería en tu trabajo de titulación se justificó por su capacidad para simplificar la administración de colas de tareas, su confiabilidad y su facilidad de integración.

6. Conclusiones

- La estructura de Scrum ofrece ventajas claras en términos de cumplimiento de plazos, definición de roles y establecimiento de ciclos de trabajo eficientes. La naturaleza de los Sprint proporciona un ritmo constante y predecible, mientras que la asignación de roles como Product Owner y Development Team garantiza una distribución clara de responsabilidades. Las reuniones planificadas facilitan la comunicación y la retroalimentación, asegurando un enfoque centrado en los objetivos del proyecto. En conjunto, la adopción parcial de Scrum promete maximizar la eficiencia y la calidad del trabajo, aumentando así las posibilidades de éxito en la culminación de la titulación.
- El desarrollo de este prototipo de microservicios utilizando Clean Architecture, ASP.NET Core 8.0 y RabbitMQ ha demostrado la flexibilidad y modularidad inherentes a este enfoque. La adopción de Clean Architecture ha facilitado la creación de un sistema altamente escalable y mantenible, con la posibilidad de cambiar fácilmente de un broker a otro, como Kafka, en el futuro. De todos modos, es importante mencionar que este trabajo se ha centrado específicamente en el uso de RabbitMQ y no ha explorado el uso de otros brokers.
- La arquitectura diseñada ha demostrado ser eficaz para la creación de un modelo robusto y escalable que facilita la gestión de procesos encolados, asegurando una ejecución fluida y ordenada de las tareas.
- La implementación de Clean Architecture y Microservices ha permitido una separación clara de responsabilidades, lo que resulta en un sistema más mantenible y flexible frente a cambios y nuevas funcionalidades.
- El uso de la arquitectura orientada a eventos y la integración de ASP.NET Core 8.0 y RabbitMQ han proporcionado una solución sólida para los desafíos del encolamiento, mejorando la eficiencia en el procesamiento y la comunicación entre servicios.

- Las pruebas ejecutadas han validado la funcionalidad integral del sistema, confirmando que la gestión de procesos encolados se realiza de manera efectiva.
- El sistema desarrollado ofrece una base sólida para futuras expansiones y optimizaciones, gracias a su diseño modular y su capacidad de adaptarse a diferentes escenarios de carga y complejidad de procesos.

Referencias

[1]. Amar Ćatović, Nevzudin Buzađija, & Samir Lemes. (2022). Microservice development using RabbitMQ message broker.

[2]. Arbems. (s/f). CQRS con MediatR en .NET 6.

[3]. AWS. (2023). ¿Qué es la arquitectura basada en eventos (EDA)?

[4]. aws. (s/f). Microservicios.

[5]. Diego Coder. (2023a, agosto 1). Introducción a las “Clean Architectures”.

[6]. Diego Coder. (2023b, agosto 7). Introducción a RabbitMQ.

[7]. Fu, G., Zhang, Y., & Yu, G. (2021). A Fair Comparison of Message Queuing Systems. IEEE Access, 9, 421–432. https://doi.org/10.1109/ACCESS.2020.3046503

[8]. Gabriel Esteban Vel´azquez. (s/f). Modelos de Teorías de Colas.

[9]. Geekebrains. (2022, diciembre 10). Clean Architecture.

[10]. IBM. (s/f). ¿Qué son los microservicios?

[11]. Ionos. (s/f). AMQP: conoce el Advanced Message Queuing Protocol.

[12]. Justas Kazanavičius, & Dalius Mažeika. (2023). The Evaluation of Microservice Communication While Decomposing Monoliths.

[13]. Microsoft. (2024, abril 2). Reflexión en .NET.

[14]. Microsoft. (s/f-a). Estilo de arquitectura de microservicios.

[15]. Microsoft. (s/f-b). ¿Qué es ASP.NET Core?

[16]. Microsoft. (s/f-c). Type Clase.

[17]. Red Hat. (2019, septiembre 27). ¿Qué es la arquitectura basada en eventos?

[18]. Refactoring Guru. (s/f). Mediator.

[19]. UM. (s/f). Características de un sistema de colas.

[20]. Yoshino, D., Watanobe, Y., & Naruse, K. (2021). A Highly Reliable Communication System for Internet of Robotic Things and Implementation in RT-Middleware with AMQP Communication Interfaces. IEEE Access, 9, 167229–167241. https://doi.org/10.1109/ACCESS.2021.3136855