DISEÑO E IMPLEMENTACIÓN DEL PROCESO DE MODERNIZACIÓN DEL SERVICIO WEB DE CONTABILIZACIÓN

Autores:

BRIAN ALEJANDRO MARTÍNEZ ROJAS
Banco Central de Cuba

ELOY CRUZ MARTIN
Banco Central de Cuba

GABRIEL LAMBERT DE LA ROSA
Banco Central de Cuba

Luis Alberto Justiz
Banco Central de Cuba

"Por una banca cubana eficiente y revolucionaria al servicio de la sociedad"

7 al 9 de diciembre de 2022
Centro Nacional de Superación Bancaria
La Habana, Cuba

Resumen

El trabajo que a continuación se presenta, tiene como objetivo plasmar todo el proceso que se ha venido realizando en el Banco Central de Cuba (BCC) por modernizar y centralizar su tecnología haciendo uso de las Tecnologías de la Información y las comunicaciones (TIC) que en este se utilizan. En específico, su sistema contable SABIC que se realiza a través de un servicio web en funcionamiento (WEB SERVICE SABIC), el cual presenta disimiles dificultades y contratiempos, además de un estado obsoleto en cuanto a tecnologías informáticas se refiere.

El sistema automatizado para la banca internacional de comercio (SABIC), ha sido desarrollado para satisfacer las necesidades de procesamiento de datos de los bancos pequeños y medianos que operen en este ámbito. Funciona en tiempo real con el objetivo de conocer en cada momento cuál es su posición respecto a sus corresponsales y clientes.

El Web Service SABIC es un servicio web que permite el registro contable que se recibe en el SABIC. Mediante procedimientos almacenados contabiliza todas las operaciones que se reciben de todos los bancos e instituciones financieras.

La solución que se diseña e implementa consta de tres componentes individuales que interactúan entre ellos: API Web Service, Bus de mensajería RabbitMQ y el Worker Web Service. Esta arquitectura mitiga problemas fundamentales como la perdida de los mensajes que llegaban al sistema y una mayor rapidez de su procesamiento.

Palabras clave: sistema contable SABIC, Web Service SABIC, modernizar y centralizar tecnología.

Introducción

La presente investigación se desarrolla en el marco del Taller Científico Raúl León Torras organizado por la ANEC del BCC a desarrollarse entre los días 7 y 9 del mes de diciembre.

Hoy en día la necesidad de modernizar los servicios que componen los procesos dentro de nuestra sociedad es tarea primordial en nuestro país. El objetivo sigue siendo llevar ofertas de calidad hacia el público que estén acordes a las exigencias de este. La modernización tecnológica del Sistema Bancario y Financiero Cubano (SBFC) es una de estas tareas a cumplir. El diseño y ejecución de dicha tarea es consecuente con el proyecto “Modernización de la infraestructura tecnológica e implementación de nuevas tecnologías” del macro programa “Institucionalidad y Macroeconomía” mediante el programa “Desarrollo del Sistema Financiero”. Va de la mano con otras políticas y estrategias, como son: la política para el desarrollo de los sistemas de pago en Cuba, aprobada el 18 de marzo de 2020; la estrategia nacional del SBFC y los Objetivos de Desarrollo Sostenible (ODS) de la agenda 2030.

Se conoce las arquitecturas y tecnologías ya obsoletas de los sistemas en uso, que tienen dificultades para garantizar un servicio eficiente a los cada vez más crecientes usuarios y operaciones que suceden como parte de las relaciones entre los diversos actores económicos de nuestra sociedad por las vías digitales. Las dificultades que comprende asumir nuevos retos y funcionalidades que demanda el país, pasan por la falta de personal calificado y la necesidad de introducir nuevas tecnologías que acompañen el proceso de transformación digital de las instituciones financieras como el análisis de grandes volúmenes datos que faciliten la alerta temprana, la información y el análisis necesario para la toma de decisiones. (Iriarte, L. et al., 2021)

La modernización del SBFC está orientada, al desarrollo de nuevos productos y procesos financieros en base a las tecnologías, la cual ofrece respuestas eficientes a todos los actores de la economía y en su interacción con el mercado internacional. Esto por supuesto, requiere que sus procesos estén optimizados al máximo nivel y estos cuenten con una constante innovación tecnológica.

Estas nuevas tecnologías giran alrededor del procesamiento de grandes volúmenes de datos (*BigData*), la inteligencia artificial en la actividad bancaria, las interfaces de programa de aplicación (*API* por sus siglas en inglés), monederos digitales, contratos inteligentes, biometría, nuevas plataformas de pago, el uso de la tecnología de libro mayor distribuido (*DLT* por sus siglas en inglés) basado en cadenas de bloque

(*Blockchain*) y aportando soluciones para minimizar la obsolescencia tecnológica (Programa de Desarrollo del Sistema Financiero, 2021)

El nuevo sistema está diseñado para cumplir con este proceso de modernización, por lo tanto, está comprometido con estas tecnologías anteriormente mencionadas. Se implementan tecnologías innovadoras tales como sistema de autenticación y autorización a través del sistema Keycloack, bus de mensajerías permanente como es el RabbitMQ, sistemas para hacer trazas y monitoreo en tiempo real a través de herramientas como Grafana y Jaeger. Sistema de notificación y emisiones de alertas a través de correo electrónico. Además, es importante recalcar que todo el proceso de implementación se llevó a cabo mediante la planificación de las tareas a través de los llamados *sprints* del Microsoft TFS, optimizando el trabajo colaborativo y la organización del mismo.

El sistema en cuestión, Servicio web del sistema SABIC (WSS) del Banco Central de Cuba (BCC) se encuentra implementado en tres componentes individuales que interactúan entre ellos: API Web Service, Bus de mensajería RabbitMQ y el Worker Web Service. Este sistema se encarga de la recepción de mensajes de determinados clientes (diferentes entidades del sistema bancario) para su posterior proceso de verificación, contabilización y respuesta hacia dichos clientes.

El punto de entrada al sistema será el componente API, el cual enviará el mensaje hacia el bus de mensajería RabbitMQ, quien lo almacenará hasta que el componente Worker consuma dicho mensaje para la contabilización del mismo, luego de que tenga respuesta el Worker publicará en el RabbitMQ y la API se la devolverá al cliente una vez que este lo solicite consumiendo dicho mensaje de respuesta.

Metodología

Para la realización del proyecto se utilizó la metodología Scrum. Esta consiste en un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos. En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

Como técnica de captura de información se realizaron entrevistas para identificar los requisitos que deben cumplir el sistema y obtener información de varios especialistas del Banco Central de Cuba. Además, la revisión de documentos y proyectos pasados desarrollados en el Banco Central de Cuba permitió un análisis previo y chequear fuentes de información.

Conceptos asociados al dominio del problema

API

Como se mencionó anteriormente el punto de entrada del sistema es a través de un *Endpoint* (*API*). Una API (Interfaz de Programación de Aplicaciones) es un conjunto de definiciones y protocolos que se utilizan para desarrollar e integrar el software de las aplicaciones. Las API permiten que sus productos y servicios se comuniquen con otros sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero. Las API le otorgan flexibilidad, simplifican el diseño, la administración y el uso de las aplicaciones, y proporcionan oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales)

Microservicios

Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. Los propietarios de estos servicios son equipos pequeños independientes. Las arquitecturas de microservicios hacen que las aplicaciones sean más fáciles de escalar y más rápidas de desarrollar. Esto permite la innovación y acelera el tiempo de comercialización de las nuevas características

Herramientas Utilizadas

Azure DevOps

En el proyecto se utilizan herramientas como el Microsoft TFS (Azure DevOps Server), es un producto de Microsoft que proporciona funciones de control de versiones, informes, gestión de requisitos, gestión de proyectos, compilaciones automatizadas, pruebas y gestión de versiones. Cubre todo el ciclo de vida de la aplicación y habilita las capacidades de DevOps.

C#

Es un lenguaje de programación moderno, basado en objetos y con seguridad de tipos, desarrollado y estandarizado por la empresa Microsoft. Su sintaxis básica

deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes. Permite a los desarrolladores crear muchos tipos de aplicaciones seguras y sólidas. Entre sus principales ventajas esta la seguridad del lenguaje, que es orientado a objetos, seguro y flexible, posee un mejor manejo de memoria, mayor accesibilidad y soporte con otros lenguajes

Visual Studio

Microsoft Visual Studio es un entorno de desarrollo integrado compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby y PHP, al igual que entornos de desarrollo web, como ASP.NET MVC, Django, etc., a lo cual hay que sumarle las nuevas capacidades en línea bajo Windows Azure en forma del editor Mónaco. Permite a los desarrolladores crear sitios y aplicaciones web, así como servicios web en cualquier entorno compatible con la plataforma .NET. Así, se pueden crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, dispositivos embebidos y videoconsolas, entre otros.

SQL Server 2019

Microsoft SQL Server es un sistema de gestión de bases de datos relacionales (RDBMS) que admite una amplia variedad de aplicaciones de procesamiento de transacciones, inteligencia empresarial y análisis en entornos informáticos corporativos. Microsoft SQL Server es una de las tres tecnologías de bases de datos líderes del mercado, junto con Oracle Database y DB2 de IBM.

REST

REST (*Representational State Transfer*) es una arquitectura de desarrollo para conectar varios sistemas basados en el protocolo HTTP y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON siendo este último uno de los más usados ya que es más ligero y legible incluso para las personas. REST se apoya en HTTP y entre los verbos más importantes que utiliza están GET, POST, PUT y DELETE. Es una alternativa en auge a otros protocolos estándar de intercambio de datos como SOAP (*Simple Object Access Protocol*), que disponen de una gran capacidad, pero también mucha complejidad. Los objetos en REST se manipulan a partir de la URI. Es la URI y ningún otro elemento el identificador único de cada recurso de ese sistema REST. La URI nos facilita acceder a la información para su modificación o borrado, o, por ejemplo, para compartir su ubicación exacta con terceros.

Librería Serilog

Es un *framework* de *Logging* para *.NET*. Ha sido construido con el registro de datos estructurado en mente. Serilog utiliza lo que se llama *sinks* para enviar sus registros a un archivo de texto, base de datos, o soluciones de gestión de registros, o cualesquiera que sean otros lugares, todo ello sin cambiar tu código.

Grafana

Es una herramienta de interfaz de usuario centralizada en la obtención de datos a partir de consultas, como también del almacenamiento de estos y su visualización. Es completamente de código abierto, y está respaldada por una gran comunidad entusiasta y dedicada.

Librería OpenTelemetry

Proporciona un estándar único de código abierto y un conjunto de tecnologías para capturar y exportar métricas, seguimientos y registros (en el futuro) desde tu infraestructura y aplicaciones nativas de la nube.

Herramienta Jaeger

Jaeger es una herramienta de software dedicada a detectar operaciones entre servicios distribuidos. Para cumplir con sus funciones, este sistema emplea recursos como trace en Jaeger, que representa la ruta de datos o ejecución mediante la plataforma.

RabbittMQ Client

RabbitMQ es un software de negociación de mensajes de código abierto que funciona como un middleware de mensajería. Implementa el estándar *Advanced Message Queuing Protocol* (AMQP). El servidor RabbitMQ está escrito en Erlang y utiliza el *framework Open Telecom Platform* (OTP) para construir sus capacidades de ejecución distribuida y conmutación ante errores.

El proyecto RabbitMQ consta de diferentes partes:

- El servidor de intercambio RabbitMQ, que recibe los mensajes y los almacena en colas de intercambio. Los clientes podrán después recuperarlos.
- Pasarelas para los protocolos HTTP, XMPP y STOMP.
- Bibliotecas de clientes para Java, framework .NET y Python.
- El plugin Shovel (en inglés: pala) que se encarga de replicar mensajes entre corredores de mensajes.

Resultados

El sistema se encuentra programado en C#, usando como entorno de desarrollo integrado (IDE) de programación Visual Studio 2019 y Visual Studio Code. Las bases de datos se encuentran en SQLServer y corre sobre un servidor Linux, aunque en algunas pruebas realizadas se ha montado sobre Windows 10.

La entrada al sistema está determinada por una API que tiene como tarea brindar servicio a los clientes del nuevo Web Service, con el fin de atenderlos asincrónicamente y sin pérdida de datos que se pueda generar mediante: fallos en el sistema, desconexión de los servidores, o algún problema que se pueda generar en el proceso.

La API también tiene como objetivo aligerar la carga del Web Service, validando los mensajes entrantes en cuanto a estructura, pero no su contenido; almacenar en una cola física los mensajes y la identificación del usuario, en la espera de ser procesado.

Para su identificación y por medidas de seguridad, el cliente solicita un token al proveedor de identidad Keycloak, proporcionando un identificador de cliente y una clave secreta. Este token tiene un tiempo de expiración, el cual una vez termina su tiempo el cliente debe volver a generar un nuevo token.

Mediante una solicitud http (Http Request), el cliente envía un mensaje que contiene un arreglo contable con estructura xml, y el identificador del mensaje, para esto se hace la solicitud tipo Post al endpoint (Punto Final) con la ruta "Message". La API mediante un Middleware (Software Intermedio), aplica la respectiva seguridad determinando si el cliente está autorizado, si es así comienza el proceso el cual consiste en: validar la estructura del arreglo, conformar un nuevo mensaje que contiene el id del mensaje, el id del cliente y el arreglo contable. Este mensaje es publicado en un software de cola física llamada RabbitMQ. Estos mensajes son procesados por el Web Service, el cual se encarga de contabilizarlos.

Cuando el cliente necesite obtener el estado de los mensajes puede acceder al endpoint tipo Get con la ruta "Message", el cual solo necesita proporcionar el token. Aquí se obtiene un mensaje tipo Json que contiene un contador que muestra cuantos mensajes hay procesados en la cola y un listado con los códigos de respuesta de cada mensaje y el identificador del mensaje. Para esto la Api debe acceder a otra cola, la cual es única para cada cliente, y obtener todas estas respuestas que proporciona el Web Service.

El bus de mensajería RabbitMQ es el encargado de recibir desde la API el mensaje con ciertas características puntuales, el mensaje entra al bus con un formato .xml, con las validaciones de los campos especificadas anteriormente. En el sistema se

usan interfaces pertenecientes a la misma para lograr este objetivo como son *IModel* y *IConnection.*

Los procesos de envío y recepción de mensajes manejan con los métodos *BasicConsume* y *BasicPublish* que pertenecen a la clase *IModel*.

```
public void ReceiveMessage()
{
    try
    {
        if (channel.MessageCount(queueName) != 0)
        {
            var consumer = new EventingBasicConsumer(channel);
            Log.Information(consumer.Model.ToString());
            consumer.Received += (ch, bodyMessage) =>
            {
                WorkerOrchestator.ProcessMessage(bodyMessage);
                channel.BasicAck(bodyMessage.DeliveryTag, false);
            };
            channel.BasicConsume(queueName, false, consumer);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception caught: {0}", ex);
        Log.Error(ex, ex.Message);
    }
}
```

Figura 1 Método Consumer.

```
    propagator.Inject(new PropagationContext(contextToInject, Baggage.Current), props, OpenTelemetryService.In
    OpenTelemetryService.AddMessagingTags(activity, routingKey, routingKey);

    //OpenTelemetryService.openTel("SendMessage", props, System.Diagnostics.ActivityKind.Producer);

    if (resp == "00")
        //logger.LogContabilizo();
        Log.Information("Contabilizó");
    //this.logger.LogInformation("Contabiliz");
    else
        Log.Information("No Contabilizó. Respuesta: " + resp);
    //   logger.LogNoContabilizo();
    //this.logger.LogInformation("No Contabiliz");

    var body = Encoding.UTF8.GetBytes(message);
    channel.BasicPublish(exchange: "ws-sabic", routingKey: routingKey, basicProperties: props, body: body);
}
catch (Exception ex)
{
    Console.WriteLine("Exception caught: {0}", ex);
    Log.Error(ex, "Message publishing failed. " + ex.Message);
    //this.logger.LogError(ex, "Message publishing failed.");
    activity.SetStatus(ActivityStatusCode.Error, "Message publishing failed. " + ex.Message);
```

Figura 2 Método Publicar.

El sistema usa un *Worker Service* como *backend* ya que es necesario que por las características del mismo la aplicación que se encarga de consumir del bus de mensajes, esté consumiendo constantemente de dicho bus. Este *Background Service* se realizará como su nombre indica en segundo plano. Al configurar la instancia de Host, hay un método de extensión llamado *ConfigureServices*. Aquí es donde podemos añadir servicios para nuestra aplicación, como nuestros servicios alojados, o cualquier servicio que deseemos utilizar la inyección de dependencia.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseSerilog()
        .ConfigureServices((hostContext, services) =>
        {
            Env.Load();
            services
            .AddHostedService<Worker>();
            OpenTelemetryService.AddCustomOpenTelemetry();
        });
```

Figura 3 Inyección del Worker en el Host.

El servicio alojado Worker envía la hora actual a la instancia del registrador. Luego duerme la tarea durante un minuto antes de repetir la tarea. La tarea se repite hasta que se solicita el token de cancelación.

```
public class Worker : BackgroundService
{
    1 reference
    private readonly ILogger<Worker> _logger;
    1 reference
    private readonly IHost _host;

    0 references
    public Worker(ILogger<Worker> logger, IHost host)
    {
        _logger = logger;
        _host = host;
    }

    0 references
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {                    You, hace 1 segundo • Uncommitted changes
            RabbitMQService rbmq = RabbitMQService.GetInstance();

            rbmq.ReceiveMessage();
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Figura 4 Implementación del Worker.

Una vez que se obtiene un mensaje comienza el proceso de contabilización, este proceso comprende de desglosar la información contenida en el mensaje de entrada en diferentes variables y llamar a procedimientos almacenados de una base de datos (SABIC) para realizar el proceso. Una vez finalizado este se obtendrá una serie de código de error los cuales constituyen la respuesta que se publicara en el bus de mensaje para que la API se la devuelva al cliente que la solicite.

Bibliotecas utilizadas en la solucion

En el sistema se implementa usando las bibliotecas de Open Telemetry una serie de métricas que luego se exportan hacia la herramienta Jaeger, esto con el fin de visualizar y seguir el funcionamiento de la aplicación de forma completa de principio a fin y encontrar errores de ejecución o males funcionamientos del sistema.

```
<PackageReference Include="OpenTelemetry" Version="1.3.0" />
<PackageReference Include="OpenTelemetry.Exporter.Console" Version="1.3.0" />
<PackageReference Include="OpenTelemetry.Exporter.Jaeger" Version="1.3.0" />
<PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.0.0-rc9.4" />
<PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore" Version="1.0.0-rc9.4" />
<PackageReference Include="OpenTelemetry.Instrumentation.Http" Version="1.0.0-rc9.4" />
<PackageReference Include="OpenTelemetry.Instrumentation.SqlClient" Version="1.0.0-rc9.5" />
```

Figura 5 Bibliotecas de OpenTelemetry.

Las métricas se generan a partir de implementar ActivitySource en las clases que se quieren seguir.

```
activitySource = new ActivitySource(nameof(RabbitMQService));
propagator = Propagators.DefaultTextMapPropagator;
```

Figura 6 Implementación en clases.

Luego queda exportar dichas métricas hacia el Jaeger como se mencionó anteriormente, referenciando el host y el puerto donde se encuentre el Jaeger.

```
.AddJaegerExporter(opts =>
{
    opts.AgentHost = Env.GetString("JAEGERHOST");
    opts.AgentPort = Env.GetInt("JAEGERPORT");
})
```

Figura 7 Configuración del exportador a Jaeger.

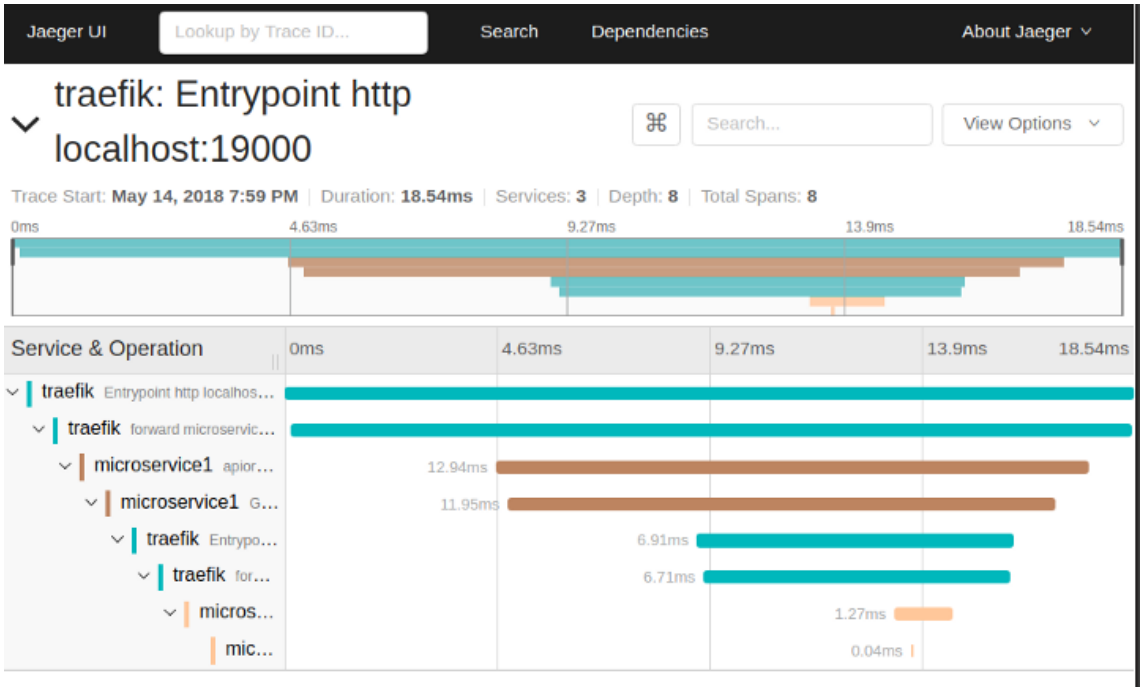

Figura 8 Vista previa de la interfaz.

Se utiliza SINK MSSQLSERVER, una biblioteca de Serilog para exportar trazas hacia una base de datos relacional SQL Server, donde se almacenarán y serán la base de las consultas posteriores para el monitoreo de la aplicación a través de la herramienta Grafana. También se utiliza la *Sink Console* para exportar los logs hacia una consola de ejecución.

```
<PackageReference Include="Serilog.AspNetCore" Version="6.0.1" />
<PackageReference Include="Serilog.Settings.Configuration" Version="3.3.1-dev-00337" />
<PackageReference Include="Serilog.Sinks.Console" Version="4.0.2-dev-00890" />
<PackageReference Include="Serilog.Sinks.MSSqlServer" Version="5.8.0-dev-00020" />
```

Figura 9 Bibliotecas de Serilog.

Una vez que las trazas se encuentren almacenadas, la herramienta Grafana permite el diseño y ejecución de consultas SQL a bases de datos para complementar sus llamados *dashboard*. Cada una de estas graficas estarán configuradas con alertas personalizadas, las cuales mandarán notificaciones vía correo electrónico a los encargados de la monitorización.

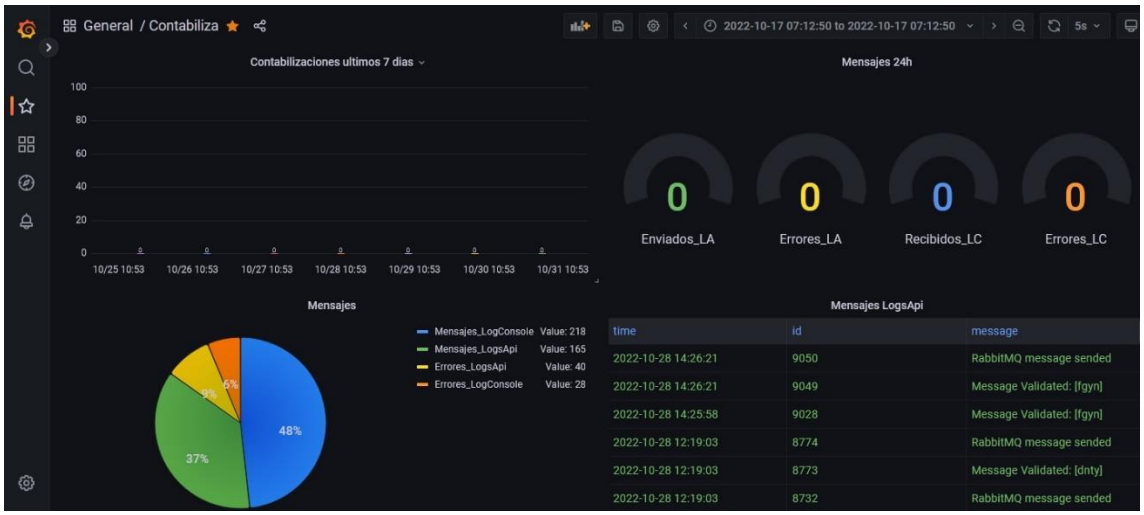

Figura 10 Dashboard de Grafana.

Discusión

La modernización del sistema demostró los beneficios que esta nueva arquitectura dotó al Servicio Web Sabic. La implementación de microservicios permitió la escalabilidad del sistema y una forma más rápida de desarrollo. Se añadieron funcionalidades tales como seguridad a través del sistema Keycloack el cual permitió la autenticación y autorización de los clientes para interactuar con la aplicación; trazabilidad y monitoreo a través de Jaeger y Grafana las cuales permiten comprobar de forma constante y eficiente el correcto funcionamiento del mismo; persistencia de datos a través de RabbitMQ lo cual era uno de los problemas principales que presentaba la versión anterior, la perdida de información valiosa.

Referencias