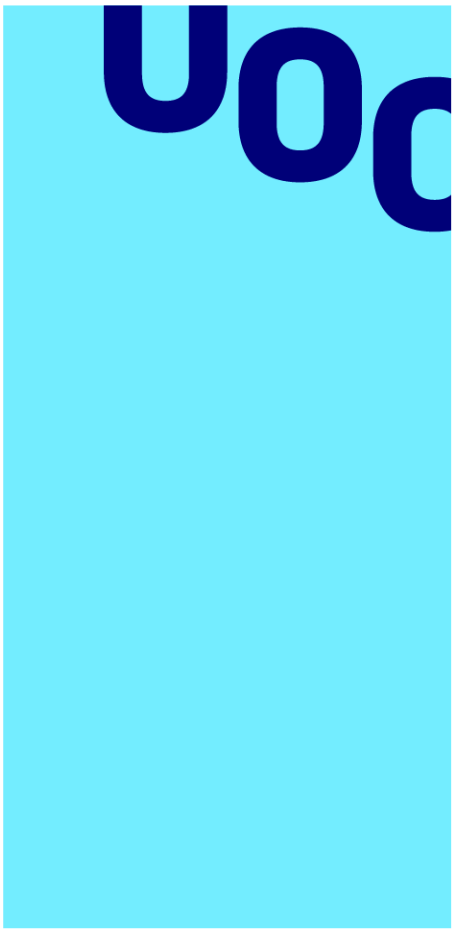
PEC 4
Notify News

Automatización de Microservicios, CI/CD y Kubernetes: una solución eficiente para el despliegue en la nube



Universitat Oberta de Catalunya

Diego Javier Ríos Sánchez

Grado en Ingeniería informática

Aplicaciones y sistemas distribuidos

Nombre Tutor/a de TF

Amadeu Albós Raya

Profesor/a responsable de la asignatura

Joan Manuel Marquès Puig

Fecha Entrega

13 de junio de 2023

Índice

Lista de figuras y tablas

“Many things can put a project off course: bureaucracy, unclear objectives, and lack of resources, to name a few. But it is the approach to design that largely determines how complex software can become. When complexity gets out of hand, developers can no longer understand the software well enough to change or extend it easily and safely. On the other hand, a good design can create opportunities to exploit those complex features… Agile processes, such as XP, emphasize the ability to cope with change and uncertainty”

Domain-driven Design: Tackling Complexity in the Heart of Software Eric Evans, Eric J. Evans

1. Introducción

1.1. Contexto y justificación del Trabajo

En los últimos años, la tecnología de computación en la nube ha experimentado un notable crecimiento y evolución. Esto ha dado lugar a la creación de diversas herramientas que facilitan el análisis, implementación y despliegue de proyectos. Estas nuevas tecnologías están definiendo el funcionamiento de sistemas y componentes distribuidos, y destacan especialmente en tres áreas clave: sistemas Serverless en la nube, integración y despliegue continuo, y tecnologías de contenerización con Docker.

Los sistemas Serverless en la nube permiten desarrollar y ejecutar aplicaciones sin la necesidad de gestionar servidores. Esto se logra mediante el aprovechamiento de proveedores de servicios en la nube como AWS Lambda, Azure Functions y Google Cloud Functions. Al utilizar estos sistemas, el trabajo se centra en el diseño y en la lógica de las aplicaciones en lugar de preocuparse por la infraestructura subyacente. Este enfoque puede resultar en una mayor eficiencia, calidad y productividad en los procesos de desarrollo de software.

La integración continua implica la automatización del proceso de construcción y prueba de aplicaciones, mientras que el despliegue continuo automatiza la implementación de aplicaciones en diferentes entornos. Esto puede mejorar significativamente la calidad del software y reducir el tiempo de entrega.

Por último, las tecnologías de dockerización permiten empaquetar aplicaciones y sus dependencias en contenedores ligeros y portátiles. Esto simplifica la implementación y administración de aplicaciones en diferentes entornos, lo que a su vez agiliza los procesos y reduce los costes asociados a la gestión de infraestructuras. La dockerización, se utiliza ampliamente junto con herramientas como Kubernetes para la orquestación y gestión eficiente de los contenedores.

1.2. Objetivos del Trabajo

Este proyecto tiene como objetivo implementar un sistema de microservicios heterogéneo, escalable y tolerante a fallos. Los microservicios estarán desarrollados en diferentes lenguajes de programación y se orquestarán mediante un sistema de mensajería asíncrona, lo que permitirá la interacción y la independencia de los componentes. Desde la fase inicial de la toma de requisitos hasta el despliegue en plataformas de proveedores en la nube, se abarcará la definición, arquitectura, desarrollo de la lógica y comunicación de los componentes, utilizando la automatización de procesos de CI/CD.

Con ello se pretende profundizar en los conceptos de independencia de tareas por servicios, en la dockerización y en la orquestación, utilizando para ello sistemas de mensajería asíncrona y su coordinación y exposición gracias al uso de Kubernetes, dentro de un proceso de despliegue automatizado haciendo uso de herramientas de CI/CD en diferentes proveedores de la nube.

Para ello, se pretende desarrollar las funcionalidades básicas que permitan la comunicación entre al menos 3 de estos microservicios escritos en diferentes lenguajes de programación, que interactúen entre ellos y con otros servicios externos de forma síncrona y asíncrona y que envíen correos de forma recurrente a las personas registradas.

Por tanto, se pretende trabajar los conceptos de independencia de componentes mediante la dockerización, la comunicación de los componentes mediante colas, la orquestación y exposición mediante Kubernetes y la automatización de tareas de compilación, validación, dockerización y despliegue de forma automatizada según los principios de la entrega continua. Gracias al uso de Kubernetes se consiguen óptimos resultados con relación a la seguridad de los aplicativos, la escalabilidad de los componentes, la tolerancia a fallos, así como la facilitación de la comunicación y transparencia, elementos básicos y necesarios de los sistemas distribuidos.

Los objetivos del trabajo, por tanto, también incluyen la configuración de los servicios para que puedan comunicarse entre sí, la creación de imágenes Docker y el registro de las mismas en proveedores de internet, la configuración y creación de un clúster de Kubernetes en proveedores de servicios en la nube y el despliegue de estos servicios en el clúster de Kubernetes mediante la implementación de procesos de CI/CD.

1.3. Impacto en sostenibilidad, ético-social y de diversidad

La competencia de compromiso ético y global, implica actuar de forma honesta, ética, sostenible y socialmente responsable y respetuosa con los derechos humanos y la diversidad.

Hay que considerar, pues, el abordar tres grandes dimensiones:

I. Sostenibilidad

En el marco del proyecto, se considerará la sostenibilidad ambiental en todas sus dimensiones, tanto en la elección consciente de los proveedores de servicios que involucra consideraciones de impacto ambiental, como en el diseño e implementación de los microservicios dockerizados en la nube mediante el uso de Kubernetes. Estos objetivos tienen un impacto **POSITIVO** en la reducción de emisiones de gases de efecto invernadero y en la disminución del consumo de energía y la huella de carbono de los centros de datos, contribuyendo así a la sostenibilidad medioambiental. Por otro lado, se realizará un estudio que tendrá en cuenta las consideraciones de impacto ambiental de los proveedores de servicios, lo que se considera un impacto **POSITIVO,** ya que podría permitir tener en cuenta esta cuestión a la hora de realizar una elección consciente de la plataforma de trabajo.

II. Comportamiento ético y responsabilidad social

Para garantizar un enfoque adecuado en este aspecto, el aplicativo se desarrollará utilizando código abierto, el cual será publicado en un repositorio público con una licencia que permita su uso y modificación sin ánimo de lucro. Este impacto **POSITIVO** del uso del código abierto permite la colaboración y el intercambio de conocimientos con la comunidad de software, lo que a su vez fomenta la innovación y la mejora continua de los productos, según se destaca en el informe *de* Red Hat *The State of Enterprise Open Source: A Red Hat report*. (n.d.)

Por otro lado, la utilización de tecnologías y herramientas modernas para garantizar un desarrollo y despliegue eficiente, escalable y seguro se considera que tiene un impacto **POSITIVO,** ya que la implementación de tecnologías y herramientas modernas puede mejorar la eficiencia y la seguridad de los sistemas, y reducir los costos y los riesgos asociados.

Por último, el envío de correos se considera que tiene impacto **NEUTRO,** puesto que este objetivo en sí mismo no tiene un impacto directo en los ODS, pero puede contribuir a la educación y la conciencia pública sobre temas relevantes.

III. Diversidad y derechos humanos

Aunque este proyecto es un trabajo técnico centrado en la creación y despliegue de componentes durante todo el ciclo de desarrollo de software, se prestará especial atención al uso del lenguaje inclusivo. Este impacto se entiende como un impacto **POSITIVO**.

Para respaldar esta idea, se puede citar la *Guía para el uso del lenguaje inclusivo en el INSST, O.A., M.P.* publicada por el instituto nacional de seguridad y salud en el trabajo, que destaca la importancia de utilizar un lenguaje inclusivo en todos los ámbitos de la sociedad. Además, diversas investigaciones han demostrado que el uso de un lenguaje inclusivo puede contribuir a la creación de entornos más equitativos e inclusivos para todas las personas, independientemente de su identidad de género, orientación sexual, raza o religión.

1.4. Enfoque y método seguido

Para lograr los objetivos del proyecto, se ha escogido una metodología de tipo *Agile*, ya que esta es adecuada para resolver problemáticas asociadas al desarrollo de microservicios en contextos de integración y entrega continua y sistemas distribuidos.

Para ello, se han tenido las siguientes consideraciones:

- Enfoque en iteraciones cortas: *Agile* se basa en la entrega temprana y frecuente de pequeñas piezas de funcionalidad. Esto se alinea con el enfoque de los microservicios, que dividen una aplicación en componentes más pequeños y autónomos. Al trabajar en iteraciones cortas, se puede entregar nuevos microservicios y mejorar los existentes de forma rápida.

- Flexibilidad para cambios: Los microservicios se construyen para ser independientes y escalables, concepto básico de los sistemas distribuidos. *Agile* permite la flexibilidad necesaria para realizar cambios en el proceso de desarrollo a medida que surgen nuevas necesidades, como agregar o quitar funcionalidades. Además, *Agile* también permite adaptarse a los cambios del mercado y a las necesidades de los clientes.

- Automatización: CI/CD se basa en la automatización para lograr entregas frecuentes y consistentes. *Agile* también enfatiza la automatización en las pruebas y la integración continua. Esta sinergia entre *Agile* y CI/CD es especialmente útil para el desarrollo de microservicios en entornos de sistemas distribuidos, donde la escalabilidad y la consistencia son críticas, manteniendo la independencia de cada componente.

El modelo metodológico se basará en una implementación simplificada de *Scrum*, con un enfoque tipo pull y la inclusión de un *Backlog* de producto. Se utilizarán *Sprints***,** que incluirán la planificación, la gestión a través de un Sprint Backlog, el uso de incrementos en la concepción del producto y el uso de *Sprint Reviews* y retrospectivas cuya finalidad será la de mantener un historial de los desafíos enfrentados y las lecciones aprendidas, lo que permitirá mejorar continuamente el proceso y la calidad del producto.

Este sistema híbrido también considerará dos factores clave. En primer lugar, se utilizará un *Backlog*, un Kanban de tareas y una hoja de ruta implementados en **Jira** para obtener una visión clara y establecer hitos durante cada entrega. En segundo lugar, se implementará un segundo Kanban en **GitLab** para asociar las historias de usuario con incidencias y tareas específicas, relacionándolas con un repositorio **GIT**.

Además, se utilizarán las siguientes herramientas de *Agile*:

- **Épicas**, para englobar un conjunto de objetivos comunes dentro del proyecto.
- **Sprints**, para trabajar en un sistema de entrega de valor incremental e iterativo.
- **Retrospectivas**, con el fin de reflexionar y mejorar la calidad de las entregas.
- **Historias de persona usuaria**, para identificar los elementos a desarrollar desde la perspectiva de la persona usuaria final.

Contenido de las Épicas del proyecto

Las tareas se han clasificado dentro de épicas según su tipo. A continuación se detalla el contenido de las épicas y tareas del proyecto:

- Planificación del producto: se realiza la planificación inicial, se definen los objetivos y los requerimientos, se diseña la arquitectura y se establece un plan de trabajo detallado.
- Walking skeleton, servicios y dockerización: se establece la arquitectura básica del proyecto y se crea un esqueleto básico de la aplicación para validar su viabilidad técnica. También se definen los microservicios necesarios y se establece la dockerización de la misma para garantizar la portabilidad y escalabilidad.
- Creación de CI/CD: se crea la infraestructura para implementar la integración continua y la entrega continua. Se establece un proceso automatizado para realizar pruebas y desplegar los cambios de forma rápida y eficiente.
- Creación de funcionalidades: se desarrollan las funcionalidades de acuerdo con el plan de trabajo definido. Se trabajará en iteraciones cortas para entregar funcionalidades tempranamente y obtener retroalimentación constante.
- Pruebas unitarias y de integración: se realizan pruebas para validar el correcto funcionamiento de las funcionalidades desarrolladas.

- Pruebas funcionales y de usabilidad: En este punto se realizan pruebas para validar la experiencia de usuario y la funcionalidad completa de la aplicación. Se evalúa la facilidad de uso, el rendimiento y capacidad de respuesta.

- Documentación y entrega: se realiza la documentación necesaria para el proyecto, incluyendo anexos, vídeos y cualquier otra documentación relevante. También se realiza la entrega final del trabajo y su presentación.

1.5. Planificación del Trabajo

Para la planificación del proyecto, se ha considerado un tiempo de trabajo equivalente a 4,5 horas diarias, es decir, 22,5 horas a la semana, para un solo recurso en *Sprints* de 1 semana de duración. El Sprint 0, de planificación, tiene una duración de dos semanas, mientras que el *Sprint* final utilizará el tiempo restante para finalizar la preparación de la entrega del trabajo de final de grado.

Las tareas se han valorado utilizando ***Story Points***, que representan la complejidad e importancia de cada tarea. Se asigna un valor aproximado de 1 '5 horas de trabajo a cada punto. Este valor puede ajustarse a medida que el equipo se acerque a su rendimiento óptimo, lo que ayudará a identificar la cantidad de tareas asumibles.

El resultado de la planificación incluye una planificación de *Sprints* y una hoja de ruta similar a un diagrama de Gantt. Esta planificación detalla las tareas, sus relaciones y posibles bloqueos.

Tareas a realizar

Cada microservicio en este proyecto tiene un propósito único y exclusivo, definido dentro de su área de responsabilidad. El primer servicio se encarga del registro de personas usuarias y de gestionar su información de interés temático. El segundo servicio tiene la responsabilidad de conectarse a una API pública de noticias y enviar noticias con las temáticas deseadas a las personas usuarias a través de correos electrónicos recurrentes. Por último, el tercer servicio se encarga de guardar un historial de notificaciones que las personas usuarias pueden consultar.

Planificación de los Sprints

A continuación se muestra la planificación por *Sprints:*

Sprint 0	**Fecha** - 1 de marzo de 2023 - 14 de marzo de 2023 **Objetivo del sprint** - El Sprint 0 define el alcance y el ámbito del proyecto. En este *Sprint* se identifica el problema, la metodología de trabajo que vamos a utilizar para resolver el mismo, así como una idea aproximada de la relación de tareas. También se

	identifican los principales riesgos del proyecto y se definen las medidas de contingencia que limiten el impacto y las desviaciones en el mismo. La duración de este *Sprint* es de dos semanas.
Sprint 1 y Sprint 2	**Fecha** - 15 de marzo de 2023 - 22 de marzo de 2023 **Fecha** - 22 de marzo de 2023 - 29 de marzo de 2023 **Objetivo de los *Sprints*** - Diseño arquitectónico inicial y creación de walking Skeleton de servicios. Se incluyen los primeros tests y pruebas de dockerización.
Sprint 3	**Fecha** - 29 de marzo de 2023 - 5 de abril de 2023 **Objetivo del sprint** - Estudiar las plataformas de servicios en la nube. Buscar información y estudiar el despliegue de CI/CD en las plataformas de servicios en la nube. También se preparan los primeros builds básicos de CI/CD.
Sprint 4	**Fecha** - 4 de abril de 2023 - 12 de abril de 2023 **Objetivo del sprint** - En este sprint se continúa con la Dockerización de los servicios, se trabaja en la creación de clúster local y despliegue inicial.
Sprint 5	**Fecha** - 12 de abril de 2023 - 19 de abril de 2023 **Objetivo del sprint** - En este *Sprint* se comienza la implementación de lógica de servicios y la creación de tests
Sprint 6	**Fecha** - 12 de abril de 2023 - 19 de abril de 2023 **Objetivo del sprint** - En este Sprint se continúa con la implementación de lógica de servicios y creación de tests. También se dan los primeros pasos para la configuración del sistema de mensajería asíncrona.
Sprint 7	**Fecha** - 26 de abril de 2023 - 3 de mayo de 2023 **Objetivo del sprint** - Se implementan los procesos automatizados de despliegue desde el repositorio gitlab a las diferentes plataformas. Se registran los contenedores en plataformas de servicios en la nube.
Sprint 8	**Fecha** - 3 de mayo de 2023 - 10 de mayo de 2023 **Objetivo del sprint** - Este Sprint se dedica a la realización de pruebas funcionales y a la optimización del clúster de kubernetes.
Sprint 9	**Fecha** - 10 de mayo de 2023 - 17 de mayo de 2023 **Objetivo del sprint** - Recopilar información sobre impacto ambiental de los despliegues en las distintas plataformas
Sprint 10	**Fecha** - 17 de mayo de 2023 - 14 de junio de 2023 **Objetivo del sprint** - Este es el *Sprint* final y tiene una duración excepcional de dos semanas. Es el momento de la finalización de la documentación TFG y de la preparación de la entrega

Tabla 1 Planificacion de Sprints

Hoja de ruta

Todo el proyecto se encuadra dentro de una hoja de ruta, que agrupa las tareas dentro de las épicas. A continuación se muestra la hoja de ruta del proyecto.

Figura 1. Hoja de ruta global

Figura 2. Hoja de ruta tareas 1

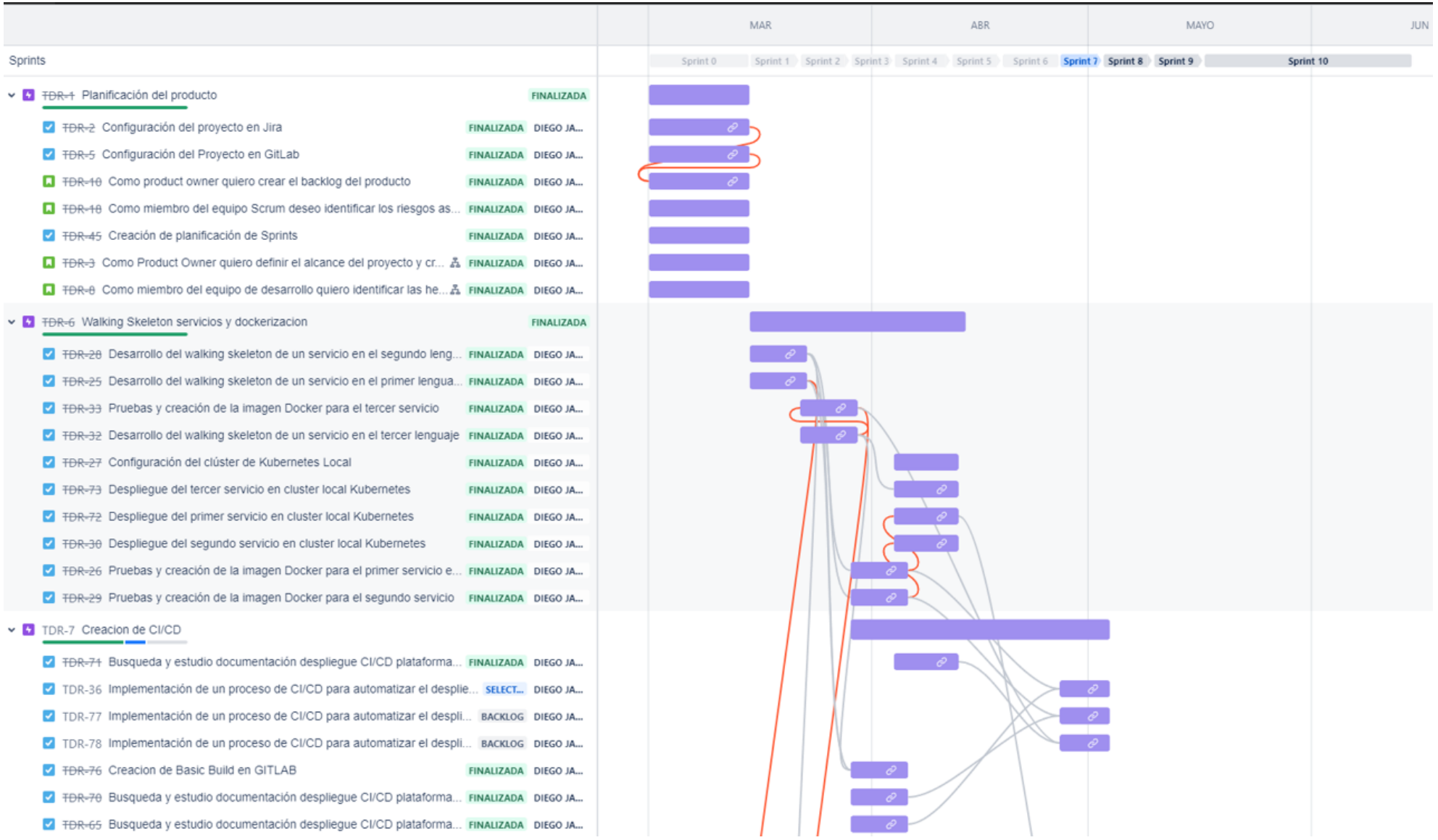
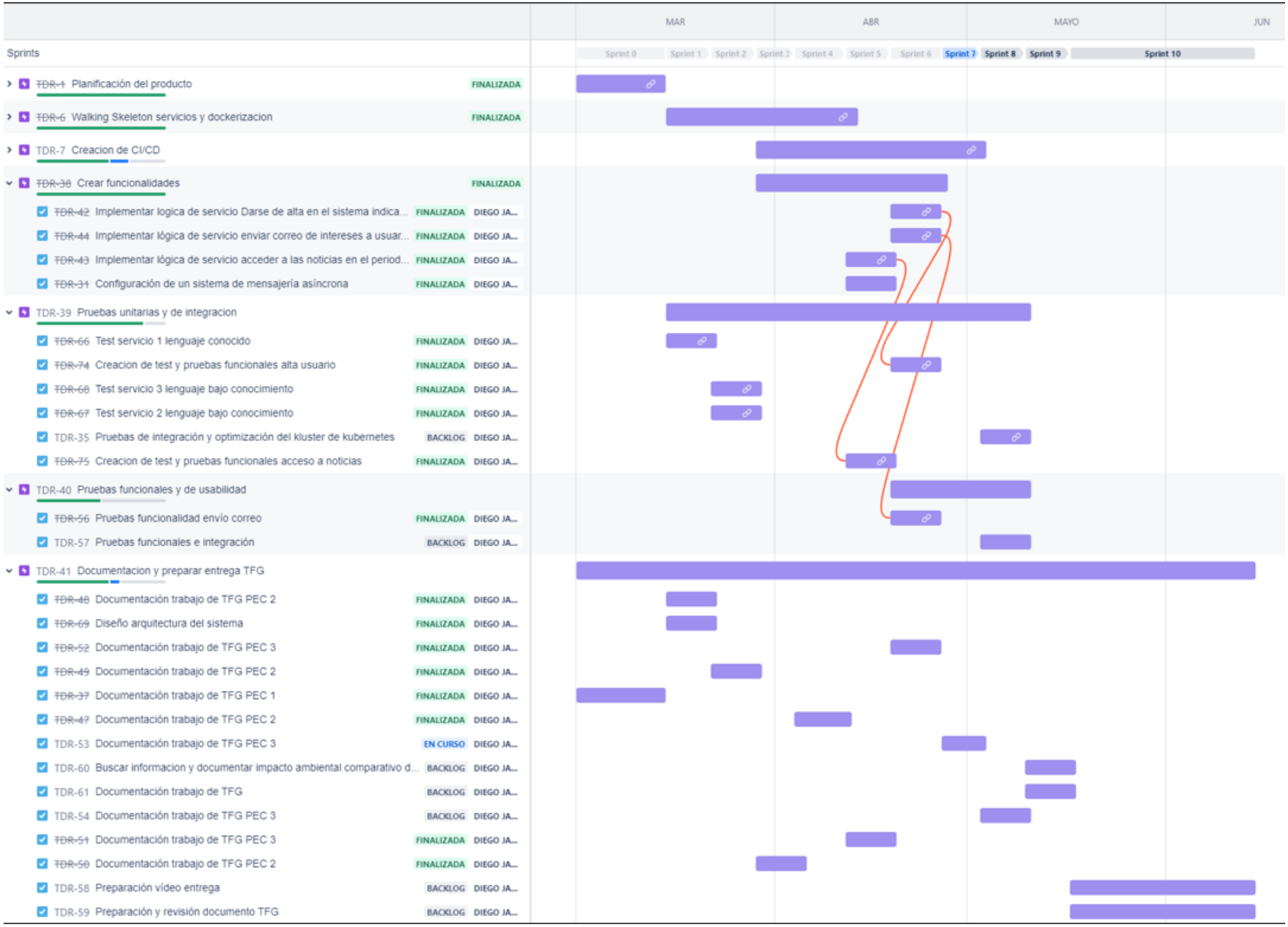


Figura 3. Hoja de ruta tareas 2



1.6. Evaluación de Riesgos

Para realizar la evaluación de riesgos definimos una serie de escenarios estructurados por áreas, a la vez que se incluye una serie de medidas de contingencia concretas que pueden servir para paliar los mismos.

ALCANCE

Riesgo: El ámbito del proyecto no está bien definido, lo que puede llevar a la inclusión de características no requeridas y estimaciones inexactas.

Prevención: Para evitar esto, se han subdividido las tareas del proyecto en tareas más pequeñas para clarificar el ámbito y las características. Además, se realizará un grooming de tareas en cada Sprint para asegurarse de que las tareas estén bien definidas.

Corrección: Se ha reservado un margen de error del 30% en la duración de las tareas para hacer frente a la contingencia. Además, la planificación de Sprints futuros no es estricta y se realizarán retrospectivas de Sprint para corregir la incertidumbre en las tareas. En caso de ser necesario, se planificarán los Sprints Backlogs y se eliminarán las funcionalidades que aporten menos valor de producto.

Riesgo: El equipo del proyecto añade características al producto que no son requerimientos o solicitudes de cambio, lo que puede llevar a un aumento en el alcance del proyecto y una posible desviación del presupuesto y plazo.

Prevención: Para prevenir esto, se debe establecer una clara definición de los requerimientos y solicitudes de cambio del proyecto. Se debe establecer un proceso de control de cambios para evaluar si una característica propuesta debe ser agregada o no al alcance del proyecto.

Corrección: Si se agregan características no requeridas o sin una solicitud de cambio formal, se deben evaluar los impactos en el presupuesto y plazo del proyecto. Si es necesario, se debe replanificar el proyecto para ajustarlo a las nuevas características. En el futuro, se debe hacer énfasis en el proceso de control de cambios y en la comunicación clara de los requerimientos y solicitudes de cambio.

Riesgo: Faltan actividades necesarias en la definición del alcance, lo que puede llevar a una falta de comprensión clara del alcance del proyecto y a una posible desviación del presupuesto y plazo.

Prevención: Para prevenir esto, se debe establecer un proceso claro y completo de definición del alcance de proyecto. Esto puede incluir la identificación y definición de los objetivos, requerimientos y entregables del proyecto, así como la definición de los límites y exclusiones.

Corrección: Si se descubren actividades faltantes en la definición del alcance, se debe realizar un esfuerzo p completar la definición del alcance. Se debe evaluar el impacto en el presupuesto y plazo del proyecto y replanificar si es necesario. En el futuro, se debe hacer énfasis en la definición completa del alcance desde el inicio del proyecto y en la comunicación clara de los límites y exclusiones del proyecto.

GESTIÓN DEL CAMBIO

Riesgo: Problemas en la definición y gestión de cambios, priorización de los cambios no esenciales, solicitudes de cambios de baja calidad y solicitudes de cambio que no tienen sentido en el contexto de los requisitos pueden llevar a una desviación del presupuesto y plazo del proyecto, así como a una pérdida de enfoque en los objetivos del proyecto.

Prevención: Para prevenir estos problemas, se debe establecer un proceso claro de gestión del cambio que incluya la identificación, evaluación y priorización de las solicitudes de cambio. Esto debe incluir la definición clara de los criterios de evaluación y la comunicación de los cambios a todas las partes interesadas relevantes.

Corrección: Si se descubren problemas en la gestión de cambios, se debe realizar una evaluación de su impacto en el proyecto. Se deben tomar medidas para corregir los cambios no esenciales y las solicitudes de cambio de baja calidad, y se debe reevaluar la priorización de los cambios restantes. Si es necesario, se debe replanificar el proyecto y ajustar el presupuesto y plazo en consecuencia. En el futuro, se debe hacer énfasis en la definición clara de los criterios de evaluación de cambios y la comunicación clara de los cambios a todas las partes interesadas relevantes

EQUIPO

Riesgo: Debilidades de los miembros del equipo, formación inadecuada, poca experiencia, problemas de rendimiento.

Prevención: Asegurar que los miembros del equipo tengan la formación adecuada y suficiente experiencia para llevar a cabo sus tareas.

Fomentar un ambiente de trabajo positivo y motivador.

Establecer prácticas de descanso regular para evitar el agotamiento y el *Burnout*.

Corrección: Identificar las debilidades del equipo y proporcionar la formación y el apoyo necesarios para superarlas.

ARQUITECTURA

Riesgo: La arquitectura carece de flexibilidad, es de baja calidad o es inviable.

Prevención: Seguir los principios SOLID al diseñar la arquitectura.

Definir una arquitectura basada en componentes para reducir el costo de desarrollo, facilitar pruebas, extensión, reutilización y flexibilidad.

Corrección: Reducir la complejidad de la arquitectura eliminando alguno de los componentes

PROYECTO

Riesgo: El proyecto carece de flexibilidad.

Prevención: Realizar un diseño flexible y escalable. Estar dispuesto a realizar cambios en el proyecto si es necesario. Realizar entregas incrementales dentro del proceso iterativo de los Sprints de Scrum de forma que se pueda percibir el incremento de valor de producto en las entregas.

Corrección: Si el proyecto carece de flexibilidad, es necesario realizar cambios en el diseño para hacerlo más flexible.

Riesgo: El proyecto falla, tiene muchos bugs e impide el avance.

Prevención: Aplicar planes de prueba desde los estadios iniciales del proyecto. Realizar una planificación adecuada de los recursos y del tiempo de desarrollo del proyecto.

Establecer procesos de gestión de calidad y pruebas rigurosas para identificar y solucionar los errores de manera temprana.

Aplicar metodologías de desarrollo ágiles que permitan una iteración constante y una mejora continua del proyecto.

Corrección: Establecer un proceso de resolución de problemas que permita identificar los errores y encontrar soluciones eficaces de manera rápida y eficiente.

Realizar pruebas exhaustivas en todas las etapas del proyecto para garantizar que se cumplan los requisitos y las expectativas.

Ajustar la planificación del proyecto.

Enfocarse en la solución de los errores más críticos que están afectando directamente el avance del proyecto

TÉCNICO

Riesgo: Los componentes técnicos no son adecuados, no son escalables, no tienen interfaces estándar, no son compatibles con las normas y violan las mejores prácticas, tienen vulnerabilidades de seguridad, características innecesarias, falta de estabilidad y no son extensibles.

Prevención: Seleccionar componentes técnicos maduros y altamente probados, que cumplan con las normas y las mejores prácticas de la industria. Realizar pruebas rigurosas de seguridad y estabilidad antes de implementar los componentes en el proyecto. Asegurarse de que los componentes tengan interfaces estándar y sean escalables y extensibles.

Corrección: Si se detectan problemas con los componentes técnicos durante el proyecto, se deben realizar las correcciones necesarias, que pueden incluir la sustitución de los componentes problemáticos por otros que cumplan con los requisitos del proyecto. Si los componentes son vulnerables a la seguridad, se deben tomar medidas de seguridad adicionales, como la implementación de cortafuegos y sistemas de detección de intrusiones. Si los componentes no son escalables o extensibles, se pueden agregar módulos adicionales para aumentar la capacidad y la flexibilidad del sistema.

Riesgo: Interrupciones de sistemas críticos, base de datos, entorno de test, etc.

Prevención: Utilización de herramientas de gestión de código fuente actualizadas con las últimas versiones en cada momento. Realizar copias de seguridad de la máquina de desarrollo.

Corrección: Si se produce una interrupción en el sistema, se deben tomar medidas inmediatas para identificar y solucionar el problema. Esto puede incluir la restauración de una copia de seguridad, la implementación de parches o actualizaciones, o la reconfiguración del sistema para minimizar el impacto de la interrupción.

Riesgo: Los componentes o productos no son mantenibles.

Prevención: Seleccionar componentes técnicos y productos que sean fáciles de mantener y actualizar. Asegurarse de que la documentación del proyecto esté actualizada y sea fácilmente accesible.

Corrección: Si se detecta que los componentes o productos no son mantenibles durante el proyecto, se deben tomar medidas para mejorar la documentación y la estructura del código para que sea más fácil de mantener en el futuro.

Riesgo: Los componentes o productos son demasiado costosos a nivel computacional.

Prevención: Realizar pruebas rigurosas de rendimiento antes de implementar el proyecto en producción para identificar cuellos de botella y problemas de rendimiento.

COSTES

Riesgo: El aumento de los costes del proyecto debido a la utilización de servicios en la nube puede provocar que se supere un límite mínimo de uso, lo que genera costes inasumibles.

Prevención: Establecer un presupuesto claro y realista para el uso de servicios en la nube y monitorizar el consumo de manera regular para evitar costes inesperados.

Considerar la posibilidad de utilizar servicios de nube alternativos que puedan ofrecer una mejor relación calidad-precio.

Corrección: Implementar un plan de reducción de costes, que podría incluir la eliminación de servicios no utilizados, la optimización del uso de recursos y la identificación de opciones más económicas.

Evaluar cuidadosamente los costes y beneficios de utilizar servicios locales en lugar de servicios en la nube para reducir los costes asociados con el uso de la nube. Antes de tomar esta decisión, es importante considerar cómo afectarán al proyecto en general y evaluar la responsabilidad en la gestión de los recursos y la seguridad de la infraestructura.

INTEGRACIÓN

Riesgo: La imposibilidad de integrar con procesos de negocio.

Imposibilidad de integración entre sistemas.

Los entornos de prueba e integración no están disponibles.

Incapacidad para integrar componentes.

Prevención:

Utilizar estándares en la comunicación entre sistemas para facilitar la integración.

Diseñar sistemas abiertos al cambio, con elementos modulares, para facilitar la integración.

Corrección:

Reducir el número de dependencias o escenarios para simplificar la integración en caso de limitaciones.

Disponer de sistemas alternativos en caso de imposibilidad de integración.

Establecer entornos de prueba e integración para garantizar la integración adecuada entre sistemas.

REQUISITOS

Riesgo: Los requisitos tienen problemas de cumplimiento.

Los requisitos son ambiguos.

Los requisitos no son aptos para el propósito.

Los requisitos son incompletos.

Prevención: Establecer un proceso de revisión de requisitos para detectar y corregir problemas de cumplimiento, ambigüedad, falta de aptitud y de completitud.

Corrección: Utilizar elementos de análisis y diseño para visualizar y entender el proyecto, las entidades, relaciones y procesos, y así mejorar la calidad de los requisitos y su visibilidad.

Crear documentos de prueba que permitan validar los requisitos y asegurar su cumplimiento.

DECISIONES Y SOLUCIÓN DE PROBLEMAS

Riesgo: Las decisiones no son las adecuadas para el propósito del proyecto.

Las decisiones incompletas crean más problemas.

Prevención: Involucrar a los stakeholders y a los expertos en la toma de decisiones para asegurar que sean adecuadas para el propósito del proyecto. Realizar un análisis detallado de las opciones disponibles antes de tomar una decisión.

Corrección: Realizar una valoración del estado del proyecto durante cada Sprint *review* para detectar decisiones erróneas y corregirlas.

Utilizar las retrospectivas como herramienta para identificar y solucionar problemas de decisiones incompletas.

Establecer medidas de seguimiento mediante el control de ejecución del plan de ruta y la ejecución de Sprints según planificación, y utilizar otras herramientas de control como un *Burndown chart*.

1.7. Breve sumario de productos obtenidos

Se entregará como producto final una aplicación compuesta por al menos tres microservicios, desplegados en diferentes plataformas de la nube. Gracias al uso de Kubernetes estos microservicios serán implementados mediante un proceso automatizado, escalable, configurable y portable Además serán resilientes y con una alta tolerancia a fallos, así como con una alta disponibilidad.

La aplicación permitirá a las personas usuarias extraer noticias de internet a través de una API pública en https://newsapi.org. Las personas usuarias podrán suscribirse a temas de su interés y recibirán un correo diario con una lista de noticias relacionadas con dichos temas.

Los entregables del proyecto incluirán el diseño detallado de la aplicación y su arquitectura, así como una prueba de concepto para validar la viabilidad técnica y funcionalidad del diseño final de implementación. Además de los componentes específicos de la aplicación, se entregarán los repositorios de código fuente resultantes para posibles futuras evoluciones, los manuales de creación de componentes, la descripción de las herramientas utilizadas y los scripts necesarios para su uso.

1.8. Recursos necesarios

Los recursos necesarios para llevar a cabo el proyecto son los siguientes:

- Un ordenador con capacidad suficiente para ejecutar las herramientas necesarias y desarrollar y probar aplicaciones en los lenguajes seleccionados.
- Un entorno de desarrollo integrado (IDE) para cada uno de los lenguajes utilizados.
- Docker, para la creación y gestión de contenedores.
- Cuentas y créditos en los proveedores de servicios en la nube para implementar y ejecutar el clúster de Kubernetes.
- Kubernetes, para la orquestación y despliegue de los servicios.

1.9. Breve resumen de capítulos

El presente documento describe el proceso de desarrollo de un sistema basado en microservicios para la extracción y envío de noticias de internet. En el capítulo de análisis se han definido las necesidades del sistema, mientras que en el capítulo correspondiente se ha identificado el *stack* tecnológico y las herramientas a utilizar. En el tercer capítulo se describe la arquitectura de microservicios, y en el cuarto capítulo se presentan las decisiones de diseño e implementación. Posteriormente, se detallan las pruebas realizadas y los resultados obtenidos en el capítulo de pruebas y validación, y se finaliza con las conclusiones y posibles líneas de trabajo futuro.

Análisis	Definición de las necesidades principales del sistema.
***Stack* tecnológico y herramientas**	Identificación de los lenguajes de programación y las herramientas utilizadas para la implementación del proyecto.
Arquitectura de microservicios	Descripción de la arquitectura de microservicios que se utilizan en el proyecto.
Diseño e implementación	Decisiones del proceso de desarrollo.
Pruebas y validación:	Descripción de las pruebas que se han realizado para validar el correcto funcionamiento de los microservicios, así como los resultados obtenidos.
Conclusiones	Presentación de las conclusiones del proyecto describiendo los productos obtenidos, se discuten

	las limitaciones y posibles mejoras, y se plantean posibles líneas de trabajo futuro

Tabla 2 Resumen de capítulos

2. Análisis

El producto que se pretende crear es el de un sistema de microservicios coordinados que gestionan la suscripción a noticias diarias mediante el cual las personas interesadas puedan solicitar el envío de correos diarios según sus intereses. Para ello se propone un sistema distribuido en diferentes componentes con responsabilidad única y que funcionen de forma autónoma, cubriendo cada uno de ellos un elemento fundamental dentro del contexto global del aplicativo. Como elementos mínimos, el sistema permitirá en primer lugar la suscripción. En segundo lugar, constará de un componente encargado de realizar consultas a una API externa para extraer los intereses de las personas usuarias. Por último, implementará también un componente encargado del envío de las comunicaciones a los clientes mediante el envío de correos. Por otra parte, se intentará proveer de funcionalidades adicionales que permitan la consulta tanto de usuarios como de notificaciones históricas. La idea fundamental del proyecto incide en el concepto del desarrollo de componentes desacoplados que se comuniquen mediante colas de de *Brokers* de mensajería, componentes con funcionalidades independientes que permitan la extensión futura del sistema sin que el mismo se vea afectado por la introducción, por una parte, de cualquier nueva funcionalidad o cuya afección por el cambio de funcionalidad en otros componentes sea mínima.

En este capítulo, se lleva a cabo el análisis de necesidades del proyecto. Para ello, se comienza por identificar los principales Stakeholders de la aplicación, entre los que se encuentran la persona usuaria y la persona desarrolladora. A continuación, se describe el ámbito y alcance de la aplicación para poder establecer las funcionalidades y requisitos necesarios para su implementación.

2.1 Stakeholders

Con el fin de determinar el ámbito y alcance de la aplicación se procede a identificar a los principales *Stakeholders* de la aplicación.

2.1.1 Persona Usuaria

La persona usuaria es la visitante de la aplicación cuyo principal interés es suscribirse al servicio de noticias y recibir actualizaciones sobre las noticias de su interés.

2.1.2 Persona Desarrolladora

La persona desarrolladora se identifica como el profesional cuyo objetivo se centra en la implementación del aplicativo mediante el uso de Kubernetes y su despliegue en la nube.

Todos los *Stakeholders* tienen intereses comunes asociados a la usabilidad de la aplicación. Desean que la aplicación sea estable, sencilla, intuitiva y rápida. También desean que la aplicación implemente un sistema de suscripción y comunicación de noticias publicadas.

2.2 Historias de usuario

Esta aplicación deberá cumplir las siguientes especificaciones definidas como historias de usuario. Estas historias de usuario se enfocan en las necesidades principales de la persona usuaria, como

la recepción de correos personalizados con noticias de su interés, la consulta de la lista de temáticas a los que está suscrita y el acceso al histórico de noticias recibidas. Además, se contempla la necesidad por parte de la persona desarrolladora de preparar despliegues automatizados hacia diferentes plataformas de servicios en la nube.

CÓDIGO	DESCRIPCIÓN
US1	Como persona usuaria quiero recibir correos personalizados con las noticias de las temáticas en las que estoy interesada
US2	Como persona usuaria quiero realizar una consulta de la lista de temáticas a las que estoy suscrita.
US3	Como persona usuaria quiero consultar el histórico de noticias que he recibido
US4	Como persona desarrolladora quiero preparar despliegues automatizados desde un repositorio de código hacia diferentes plataformas de servicios en la nube

Tabla 3 Historias de usuario

2.3 Requisitos funcionales

En esta sección se presentan los requisitos funcionales, que describen las funciones que el sistema debe proporcionar para satisfacer las necesidades de los Stakeholders. En la tabla 2 se detallan los requisitos funcionales identificados para la aplicación de noticias.

CÓDIGO	DESCRIPCIÓN	VALOR
FR1	El sistema debe permitir que la persona usuaria se suscriba mediante su correo electrónico a la temática de noticias que desee	10
FR2	El sistema debe enviar un correo diario a la persona usuaria con las noticias asociadas a la temática deseada.	10
FR3	El sistema debe permitir a la persona usuaria consultar la temática de noticias a la que está suscrita	5
FR4	El sistema debe permitir a la persona usuaria consultar el histórico de noticias que le ha sido enviado.	8

Tabla 4 Requisitos funcionales

2.4 Requisitos no funcionales

En esta sección se detallan los requisitos no funcionales que debe cumplir el sistema, importantes para su correcto funcionamiento y su uso a largo plazo. En este caso, se detallan tres requisitos no funcionales que se consideran fundamentales para el desarrollo y la implementación del sistema.

CÓDIGO	DESCRIPCIÓN	VALOR

NFR1	El sistema debe permitir la escalabilidad horizontal de forma sencilla	8
NFR2	El producto resultante debe tener alta disponibilidad	9
NFR3	El software debe ser desarrollado siguiendo una cultura de colaboración, automatización y entrega continua, que cumpla con los principios de la metodología DevOps y de las prácticas de Integración Continua (CI) y Entrega Continua (CD),	10

Tabla 5 Requisitos no funcionales

2.5 Requisitos descartados

De momento en esta primera entrega de producto se desecha FR3, ya que se considera que es el requisito que menos valor aporta el producto.

FR3	El sistema debe permitir a la persona usuaria consultar la temática de noticias a la que está suscrita

2.6 Casos de uso

En esta sección, se presentan los casos de uso para el sistema de suscripción a noticias, que permitirán entender cómo las personas usuarias podrán utilizar el sistema de manera efectiva y eficiente. El sistema estará dividido en componentes que interactúan entre ellos para mantener la coordinación y la coherencia global del sistema.

2.6.1 Suscribirse a noticias

Este caso de uso representa claramente un componente de responsabilidad única que será desarrollado en más profundidad durante el diseño de la arquitectura.

Caso de uso UC1 Suscribirse a noticias	
Código	UC1
Nombre	Suscribirse a noticias
Precondición	
FR	FR1
Actor Principal	Persona usuaria
Garantías Éxito	La persona usuaria se suscribe a una temática específica de noticias.
Escenario Principal de Éxito	1. El sistema espera que la persona usuaria introduzca su correo electrónico y una temática de noticias 2. La persona usuaria introduce su correo electrónico y una temática de noticias 3. El sistema registra los datos de correo y la temática asociada al correo
Extensiones	3.a El correo ya existe

	3.a.1. El sistema solo guarda los datos de temática
Postcondición	La persona usuaria está suscrita a la temática específica de noticias y recibirá correos electrónicos periódicos con información relevante sobre la temática seleccionada. El sistema registra la suscripción de la persona usuaria de manera correcta y efectiva. La persona usuaria cursa alta en el sistema.

Tabla 6 Caso de uso UC1 Publicar suscribirse a noticias

2.6.2 Enviar correo diario de noticias a la persona usuaria

Este caso de uso a su vez también puede representarse en al menos dos componentes de responsabilidad única, uno encargado del seguimiento diario y extracción de las noticias y otro encargado del envío de las mismas.

Caso de uso UC2 Enviar correo a persona usuaria	
Código	UC2
Nombre	Enviar correo diario de noticias a la persona usuaria
Precondición	La persona usuaria está dada de alta en el sistema y tiene un listado de temáticas sobre las que desea recibir información
FR	FR2
Actor Principal	Persona usuaria
Garantías Éxito	La persona usuaria recibe un correo con las noticias de la temática deseada.
Escenario Principal de Éxito	1. El sistema realiza un seguimiento diario de las noticias publicadas en los temas a los que la persona usuaria está suscrita mediante la conexión a la API externa de noticias. 2. Cuando se publica una noticia relevante en uno de los temas de la suscripción de la persona usuaria, el sistema la identifica y la agrega a una lista de noticias para esa persona usuaria. 3. El sistema envía un correo electrónico a la persona usuaria con un resumen de los temas de noticias a los que está suscrito y con la lista de noticias destacadas. 4. La persona usuaria recibe el correo electrónico y puede hacer clic en los enlaces proporcionados para leer las noticias completas.

Tabla 7 Caso de uso UC2 Enviar correo diario de noticias a la persona usuaria

2.6.3 Consultar historial de noticias

La consulta de noticias permite agregar otra componente de funcionalidad desacoplada.

Caso de uso UC3 Consultar historial de noticias	
Código	UC3
Nombre	Consultar historial de noticias
Precondición	La persona usuaria está dada de alta en el sistema y tiene un listado de temáticas

	sobre las que desea recibir información
FR	FR4
Actor Principal	Persona usuaria
Garantías Éxito	La persona usuaria recibe un correo con las noticias de la temática deseada.
Escenario Principal de Éxito	1. El usuario selecciona la opción de "Historial de noticias". 2. El sistema muestra una lista de noticias previamente publicadas en orden cronológico inverso, es decir, de la más reciente a la más antigua
Postcondición	La persona usuaria puede visualizar las noticias previamente enviadas en el sistema

Tabla 8 Caso de uso UC3 Consultar historial de noticias

2.6.4 Automatización de despliegues a servicios en la nube

Mediant este caso de uso se potencia el uso de la dockerizacion, de los procesos de CI/CD y se profundiza en el conocimiento de toda la automatización de procesos y el despliegue de microservicios en las diferentes plataformas de servicios en la nube.

Caso de uso UC4 Automatización de despliegues a servicios en la nube	
Código	UC4
Nombre	Automatización de despliegues a servicios en la nube
Precondición	La persona desarrolladora tiene acceso a un repositorio de código que contiene el código fuente del software a desplegar. La persona desarrolladora tiene acceso a diferentes plataformas de servicios en la nube donde se desplegará el software
US	US4
Actor Principal	Persona desarrolladora
Garantías Éxito	Los despliegues automatizados a diferentes plataformas de servicios en la nube se realizan con éxito a partir del repositorio de código especificado
Escenario Principal de Éxito	1. La persona desarrolladora define los requisitos y especificaciones para el despliegue automatizado del software a través de un archivo de configuración. 2. La persona desarrolladora configura y establece las credenciales de acceso a las diferentes plataformas de servicios en la nube donde se desplegará el software. 3. La persona desarrolladora establece un flujo de integración y entrega continua (CI/CD) que permita el despliegue automatizado desde el repositorio de código hacia las diferentes plataformas de servicios en la nube. 4. La persona desarrolladora verifica que el software se ha desplegado correctamente en cada plataforma de servicios en la nube. 5. El sistema confirma el despliegue exitoso del software en cada plataforma de servicios en la nube.

	6. La persona desarrolladora realiza las pruebas necesarias para verificar el correcto funcionamiento del software desplegado en cada plataforma de servicios en la nube.
Extensiones	3.a. Si la configuración de las plataformas de servicios en la nube no es correcta, el sistema notifica a la persona desarrolladora y detiene el proceso de despliegue automatizado hasta que se resuelva el problema. 4.a. Si el software no se ha desplegado correctamente en alguna plataforma de servicios en la nube, el sistema notifica a la persona desarrolladora y detiene el proceso de despliegue automatizado hasta que se resuelva el problema.
Postcondición	Los servicios en la nube seleccionados se encuentran actualizados con la última versión del código del repositorio y están en pleno funcionamiento

Tabla 9 Caso de uso UC4 Automatización de despliegues a servicios en la nube

El siguiente diagrama muestra los casos de uso que son necesarios para que el sistema funcione correctamente.

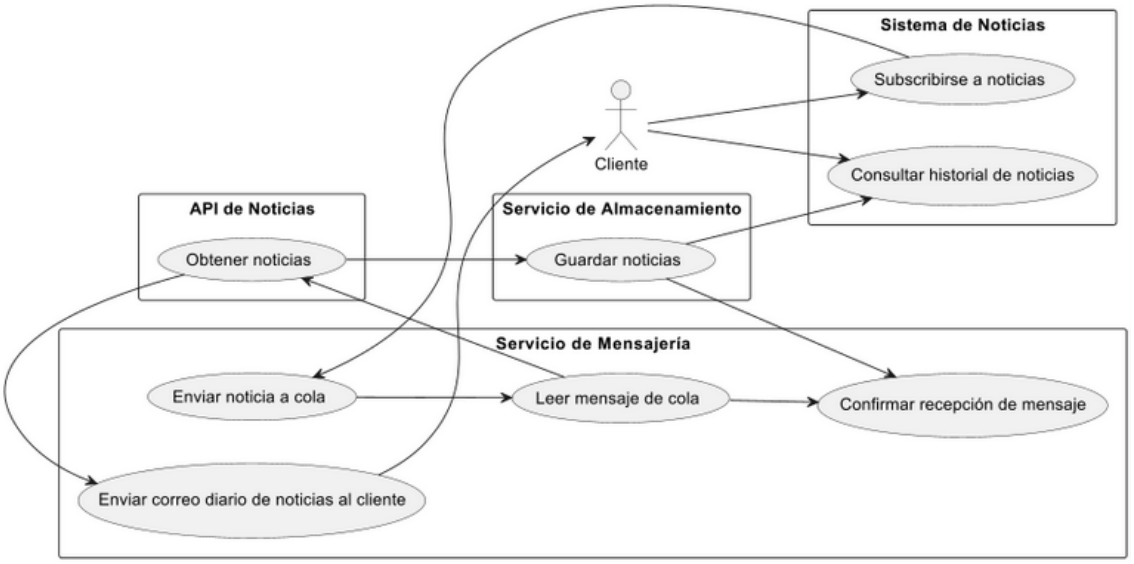


figura 4 Diagrama de casos de uso

3. Stack tecnológico y herramientas

3.1 Lenguajes de Programación

Para el desarrollo de los microservicios, se han elegido tres lenguajes de programación principales: Python, Java y C#. Se han seleccionado estos lenguajes debido a que son populares y están bien documentados.

Python **facilita el desarrollo de microservicios debido a su facilidad de uso** y su amplia variedad de bibliotecas y frameworks. **Python es particularmente adecuado para el procesamiento de datos** y la construcción de aplicaciones de aprendizaje automático. Para el desarrollo de microservicios en Python, se ha elegido Visual Studio Code como entorno de desarrollo integrado (IDE) debido a su sencillez.

Java es otro lenguaje de programación popular para el desarrollo de microservicios. Ofrece una amplia gama de bibliotecas y frameworks, y su integración perfecta con **Spring Boot permite desarrollar rápidamente servicios REST** con una configuración básica sin escribir mucho código y es especialmente adecuado para la construcción de aplicaciones empresariales complejas. **También es el lenguaje en el que se tiene un mayor dominio**. Para el desarrollo de microservicios en Java, se ha elegido IntelliJ como IDE debido a su amplia gama de funciones y herramientas integradas, como maven para la gestión de librerías, lombok para la simplificación del proceso de creación de clases y su integración directa con repositorios GIT.

Por último, se ha elegido **C#** como tercer lenguaje, usando como herramienta de desarrollo Visual Studio Community 2022 como IDE debido a su amplia popularidad, recursos de documentación, así como su capacidad de integración con otras herramientas de Microsoft.

3.2 Herramientas y Tecnologías

Hay una gran variedad y combinación de herramientas que pueden utilizarse para el desarrollo de este proyecto, por lo que **se ha primado** la búsqueda de **herramientas populares, bien documentadas y sobre las que el equipo de desarrollo tenga algo de conocimiento**, teniendo en cuenta que el desarrollo se realiza en un sistema operativo Windows.

En lo que respecta a los contenedores, se ha optado por Docker para Windows, que es una herramienta ampliamente utilizada para el despliegue de aplicaciones en contenedores, no necesita configuración y trabaja de forma directa montando un clúster de Kubernetes. Docker **puede trabajar en diversos sistemas operativos y plataformas en la nube**, ofrece ventajas para **trabajar con swarm y compose** al ofrecer herramientas de **gestión de clústeres** y tiene una gran cantidad de imágenes disponibles en internet, lo que permite la **puesta en marcha** de cualquier servidor **en pocos minutos**. Es perfecto para hacer pruebas, de fácil registro en diversas plataformas y posteriormente su uso **con kubernetes es prácticamente automático**, mediante unas sencillas anotaciones declarativas. En fin, su gran popularidad hace que esté muy bien documentado y que disponga de una gran cantidad de imágenes que permiten cubrir sobradamente las necesidades del proyecto.

Para el despliegue de los microservicios, se ha optado por los proveedores de servicios en la nube **Azure, Google Cloud y Amazon Web Services**. Estos proveedores ofrecen una amplia variedad de servicios en la nube, como el alojamiento de aplicaciones, bases de datos, redes y servicios de análisis de datos. Son ampliamente conocidos, **un estándar en la industria** y cualquier proceso a realizar está muy **bien documentado,** además de ofrecer una muy calidad en cuestiones de **soporte**.

En el stack tecnológico y herramientas también se deben incluir otras tecnologías y herramientas que son necesarias para el desarrollo de los microservicios. Por ejemplo, un broker de mensajería y un sistema de bases de datos.

En cuanto al sistema de mensajería, la implementación de una cola de mensajes utilizando RabbitMQ puede ser una excelente opción para mejorar la comunicación entre microservicios. RabbitMQ es un sistema **altamente confiable y escalable** que ofrece una amplia gama de características que permiten una comunicación asincrónica segura entre los microservicios.

Además de permitir el envío seguro de los mensajes, RabbitMQ garantiza un procesamiento rápido y confiable de los mismos, lo que puede ser esencial en un entorno de microservicios en el que la **velocidad y la confiabilidad** son críticas.

RabbitMQ **se puede dockerizar fácilmente**, es sencillo y directo en su uso, lo que significa que puede implementarse en poco **tiempo** y con poco **esfuerzo**. Además, RabbitMQ **se integra de forma sencilla** con la mayoría de lenguajes de programación y su **uso es muy simple**, lo que permite que las aplicaciones configuren el diseño de comunicación con un gran nivel de abstracción. Es otro **estándar de la industria**, un sistema **maduro** y ampliamente documentado. Se ha valorado trabajar con Kafka, pero el uso de zookeeper exige un componente extra, por lo que en este caso se ha optado por la sencillez.

PostgreSQL y MySQL pueden ser buenas opciones para la persistencia de datos. PostgreSQL y MySQL son dos de las bases de datos relacionales más **populares, gratuitas y de código abierto**. Ambas proporcionan una gran cantidad de funcionalidades y herramientas para la gestión de bases de datos, y **son ampliamente utilizadas en aplicaciones web y en la nube**. PostgreSQL es conocido por su soporte de características avanzadas como la integridad referencial, las transacciones anidadas y los procedimientos almacenados, mientras que MySQL se destaca por su velocidad y escalabilidad.

Además de las herramientas mencionadas, se utiliza **PLANTUML** como una herramienta de ayuda al análisis para la creación de esquemas UML. PLANTUML es una herramienta de **software libre** que permite la **creación de diagramas** UML utilizando una sintaxis simple y legible. Esta herramienta **facilita la visualización de la arquitectura del sistema** y ayuda a las personas desarrolladoras a entender y comunicar de manera efectiva el diseño del mismo. Por otra parte, es una herramienta que también **está muy bien documentada** y es **sencilla** de usar.

Por último, se utilizará **Swagger** para probar el correcto funcionamiento de los servicios. Swagger es una herramienta popular para documentar y probar APIs. Proporciona una interfaz fácil de usar que permite a las personas desarrolladoras probar y depurar los servicios en tiempo real. Además, Swagger **genera automáticamente la documentación** de la API a partir de los comentarios en el código, lo que **facilita la documentación y mantenimiento de la misma**. El haber utilizado

swagger con anterioridad supone otra ventaja para la justificación de su uso en el proyecto. Es **fácil de configurar y es efectiva**.

3.3 Gestión del proyecto

Para la gestión de proyectos, se ha optado por JIRA. JIRA permite la asignación de tareas, el **seguimiento del progreso** y la **visualización de informes detallados**, lo que facilita tanto la gestión del proyecto como la colaboración y coordinación en equipos de desarrollo, todo dentro de procesos *Agile*. Además, JIRA es **altamente adaptable** y cuenta con un ecosistema que permite su integración con otras herramientas como Confluence y GitLab. En comparación con otras herramientas como Easy Redmine resulta más amigable, o más potente que otras como Trello, y además tiene una versión gratuita.

En cuanto a la gestión del código fuente, se ha elegido GitLab. GitLab no solo ofrece una sencilla gestión de ramas y edición de código en línea, sino que también permite la integración con herramientas de integración continua y despliegue continuo (CI/CD), la posibilidad de **alojar repositorios privados** y la **automatización de flujos de trabajo**. También ofrece características avanzadas como la gestión de tickets, la revisión de código y la **integración con JIRA**. Otra buena opción podría haber sido GitHub, el cual se centra también dentro de una gran comunidad colaborativa, ofreciendo herramientas para la discusión y la contribución al código abierto. En este caso han primado más los factores tanto de integración como de conocimiento las herramientas.

La elección de esta combinación de herramientas se basa en la capacidad de abordar proyectos tanto desde el punto de vista de la gestión a nivel abstracto e integración con otras herramientas como desde una perspectiva más técnica y potente. Ambas herramientas son maduras, ofrecen una amplia gama de funciones y están en constante crecimiento, además de contar con versiones gratuitas. En el anexo VII se puede consultar la integración entre Jira y GitLab.

GitLab también facilita el despliegue a proveedores de la nube al proporcionar integraciones con proveedores como Azure, Google Cloud Platform y Amazon Web Services, lo que permite la implementación y el despliegue de aplicaciones directamente automatizando el proceso de implementación en la nube y asegurando que el código se pruebe, compile y despliegue sin errores en diferentes entornos.

En Gitlab se definirá un proyecto que contendrá diferentes repositorios para cada microservicio. Según la idea de que cada equipo de desarrollo trabaja con un microservicio, idealmente cada repositorio debe llevar la gestión de sus incidencias. En este caso, la gestión de las incidencias se llevará únicamente desde uno de estos repositorios para así simplificar todo el proceso de gestión y desarrollo.

Además de un repositorio para cada microservicio se agrega un repositorio extra para la documentación, que incluye plantillasUML con las definiciones de la arquitectura del sistema.

Otra decisión importante que se ha tomado es la de crear pequeños anexos que indiquen el proceso de creación de los servicios y la infraestructura necesaria. Los anexos definidos se pueden encontrar al final de éste documento, incluyendo las configuraciones utilizadas para la implementación del sistema.

4. Arquitectura de microservicios

4.1. Principios generales de la arquitectura

El concepto de microservicios está muy relacionado con el principio **SOLID** de Responsabilidad Única (SRP), ya que uno de los objetivos principales de esta arquitectura es la de crear pequeños servicios independientes y autónomos que se encarguen de una única responsabilidad o funcionalidad específica. En este sentido es muy importante definir el nivel de granulado de los mismos. Hay sistemas que pueden estar funcionando incluso con miles de microservicios, por ejemplo Netflix.

Cada microservicio dentro de esta arquitectura es una unidad de negocio independiente, lo que significa que está diseñado para manejar una sola tarea o responsabilidad. Esto permite una mayor flexibilidad y escalabilidad, ya que cada uno de ellos puede ser desarrollado, probado y desplegado por separado, sin afectar a los demás.

Además, el principio de **SRP** se aplica también a nivel de código dentro de cada microservicio. Cada microservicio debe estar diseñado de tal manera que cada clase o módulo tenga una única responsabilidad. Esto permite un mejor mantenimiento y evolución del sistema, ya que los cambios se pueden hacer de manera más sencilla y segura dentro de una única responsabilidad.

Otro principio importante en la arquitectura de microservicios es el Domain-Driven Design (DDD), que establece que el diseño de un sistema debe estar basado en el modelo de negocio y en los términos y conceptos utilizados en dicho modelo. Esto implica que cada microservicio debe estar enfocado en una parte específica del modelo de negocio y debe estar diseñado de tal manera que refleje dicha parte del modelo. Al utilizar DDD, los equipos de desarrollo pueden asegurarse de que están creando servicios que se alinean con los objetivos y necesidades del negocio.

4.2 Descubrimiento de las operaciones del sistema

Derivados de las historias de usuario y de los requisitos funcionales del sistema, se identifican las siguientes operaciones:

1. Suscribirse a noticias
2. Obtener noticias de sistema externo
3. Consultar historial de noticias
4. Enviar correo

Con esta información se puede empezar a definir los subdominios que podrían encuadrarse dentro del siguiente sistema de microservicios.

Servicio 1, Client Service, topicsubscriber

Recibe información del cliente, el correo electrónico y la temática a la que se desea suscribir. Esta información se envía al broker de mensajería.

Servicio 2 News Service, msgquery

La función principal de este servicio es tomar las preferencias de la persona usuaria de una cola del broker de mensajería y guardar un registro de la temática seleccionada. Además, diariamente se conecta a un servicio de noticias externo para consultar la temática seleccionada deseada y guarda estas noticias enviándolas a otra cola en el broker de mensajería.

Servicio 3 Historic Service, topichistory

Este servicio se conecta al broker de mensajería y guarda las noticias que han sido enviadas a la persona usuaria. Al guardar la información permite exponer una interfaz de conexión que permite a su vez consultar el histórico de noticias.

Servicio 4 Email Service, msgdispatch

Este servicio se encarga de enviar un correo cuando encuentra noticias pendientes de enviar en el broker de mensajería. Para simplificar el sistema, la versión inicial del proyecto incluye esta funcionalidad dentro del servicio 2, msgquery

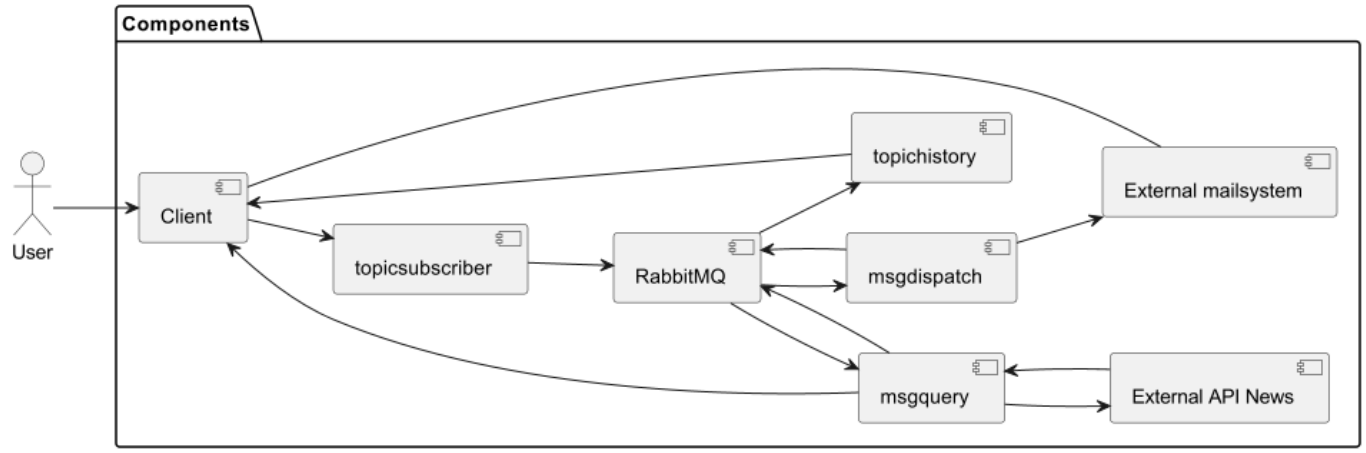


figura 5 Diagrama de componentes del sistema

4.3 Descripición de los procesos

Proceso de suscripción y entrega diaria de noticias por correo electrónico.

Siguiendo las prácticas y principios de la arquitectura de microservicios, se establece que uno de los elementos clave es el desacoplamiento de los componentes. Para lograr esto, se ha identificado que la comunicación entre los diferentes microservicios se debe realizar a través de un *broker* de mensajería, lo que permite una comunicación asíncrona entre los componentes.

Este enfoque de comunicación asíncrona tiene un gran impacto en la escalabilidad, la concurrencia y la estabilidad del sistema. Al permitir que los diferentes microservicios se comuniquen de manera asíncrona, se evita la necesidad de una comunicación directa entre ellos, lo que puede dar lugar a cuellos de botella, a una baja escalabilidad y un pobre desacoplamiento

debido a las interdependencias. Por tanto, el sistema se convierte en resiliente, independiente en sus partes, tolerante a errores y escalable.

Además, la comunicación asíncrona permite una mayor concurrencia en el sistema al permitir que varias instancias de los componentes estén en ejecución al mismo tiempo, lo que puede mejorar significativamente el rendimiento. Por último, al desacoplar los diferentes microservicios, se aumenta la estabilidad del conjunto del sistema, ya que si un microservicio falla, los demás microservicios no se ven afectados y pueden continuar funcionando correctamente. Por otra parte, el uso del clúster de Kubernetes impactará de manera directa en el control de la escalabilidad y disponibilidad de los microservicios.

El siguiente diagrama de secuencia muestra el proceso de suscripción a noticias y su posterior entrega diaria a la persona usuaria mediante el envío de correos.

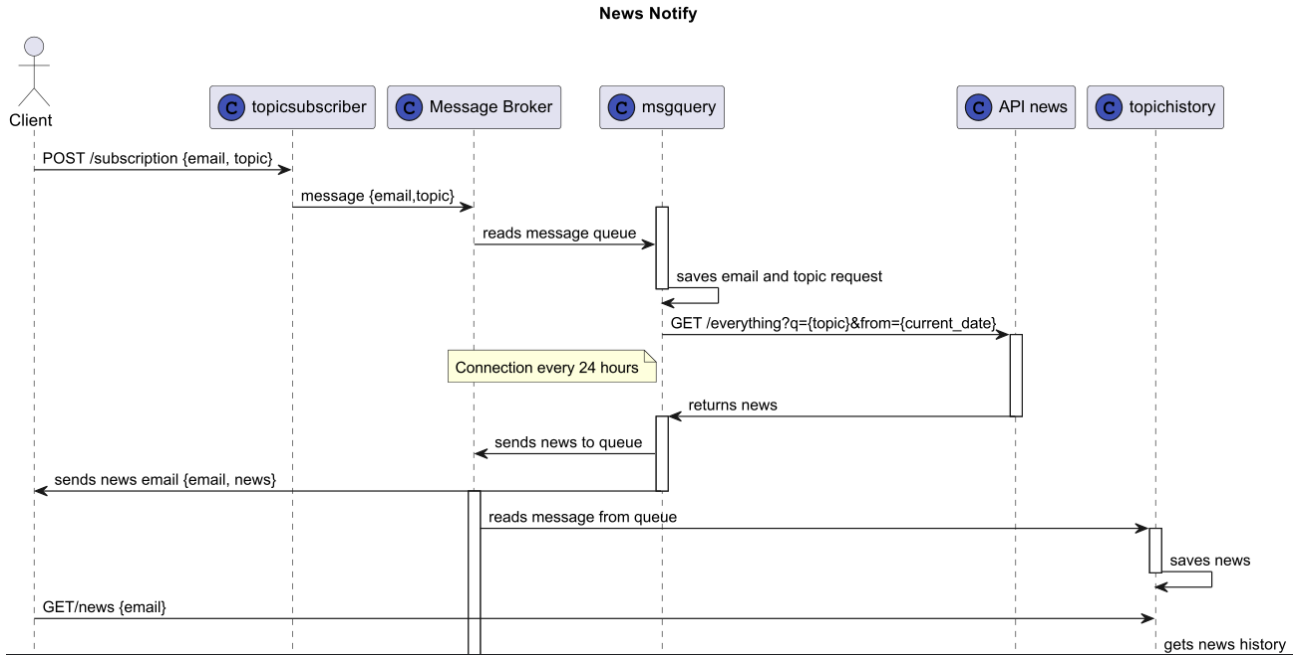


figura 6 Diagrama de secuencia del sistema

El proceso se inicia con el actor "Cliente" que se comunica con el microservicio **topicsubscriber** mediante una API Rest para solicitar una suscripción indicando su correo electrónico y una temática específica. A continuación, **topicsubscriber** envía un mensaje a **RabbitMQ** con los datos del cliente, correo electrónico y el tema elegido. El broker de mensajería, a su vez, envía el mensaje a **msgquery** para ser procesado.

msgquery almacena la petición de correo y temática y confirma al sistema de mensajería la recepción del mensaje. Este sistema tiene dos procesos principales. El primero es un proceso que se encarga de atender las peticiones que llegan por medio de una cola directa y les da persistencia. Un sistema programado que se ejecuta en intervalos regulares de tiempo realiza consultas al repositorio de nuevas entradas de clientes y basándose en ello ejecuta consultas a la API externa y envía el mensaje. Una vez confirmada la recepción del mensaje, marca el mismo como enviado y espera hasta el día siguiente para volver a realizar una nueva consulta a la API externa y enviar así de forma diaria a la persona usuaria las noticias de las temáticas de su interes. Así, cuando **msgquery** obtiene las noticias, las envía al Sistema de mensajería y queda a

la espera de la recepción de confirmación de entrega del mensaje. Por su parte, el sistema de mensajería lee el mensaje y lo envía a **topichistory** para ser archivado.

A su vez, **topichistory** guarda las noticias en su base de datos y confirma al sistema de mensajería la recepción del mensaje. Finalmente, el Cliente puede solicitar su historial de noticias a **topichistory** mediante una petición API Rest, que retorna el historial de noticias.

Diagrama de contenedores del modelo C4 para la arquitectura del sistema

El siguiente diagrama de contenedores de nivel 1 del modelo C4 (Context, Container, Component, Code) muestra la arquitectura de alto nivel de este sistema, identificando los principales contenedores y sus relaciones. Los contenedores representan componentes de software que pueden ser ejecutados de manera independiente y que se comunican a través de interfaces.

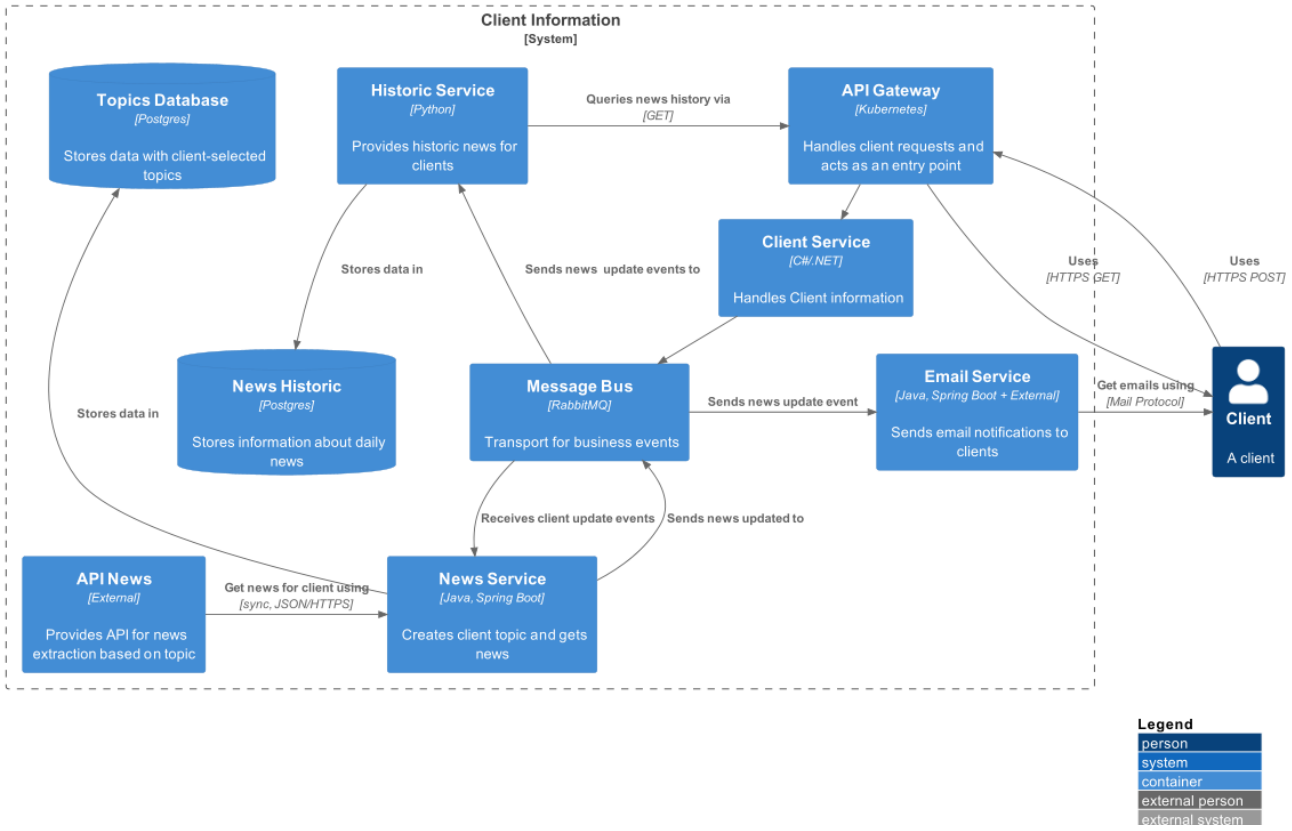


figura 7 Diagrama de contenedores de nivel 1 del modelo C4

4.4 Uso de patrones arquitectónicos en microservicios

El uso de patrones de microservicios es parte tanto de la arquitectura como del diseño de una aplicación basada en microservicios. Estos patrones proporcionan una guía para la implementación de características comunes en microservicios, como la comunicación entre servicios, el manejo de errores, la escalabilidad y la persistencia de datos.

La elección y aplicación de patrones de microservicios es un aspecto importante del diseño de la arquitectura de una aplicación basada en microservicios. Sin embargo, la arquitectura en sí misma

también puede incluir decisiones más amplias sobre la organización de los servicios, la infraestructura y las tecnologías utilizadas.

A continuación se muestran algunos de los patrones escogidos para el diseño arquitectónico de la aplicación

4.4.1 Patrón: Database per service

El patrón "Database per service" se refiere a la práctica de asignar una base de datos dedicada a cada microservicio en lugar de utilizar una base de datos compartida por todos los servicios. La finalidad de esto es evitar un acoplamiento fuerte entre los servicios y permitir una mayor independencia y escalabilidad, ya que cada transacción solo afecta al servicio que la posee. Además, los datos son privados a nivel de microservicio y solo pueden ser accedidos mediante una API.

Cada microservicio tiene su propio esquema de base de datos, lo que permite que los equipos de desarrollo trabajen de forma independiente sin interferir con otros servicios. También brinda una mayor flexibilidad en cuanto a la elección de tecnologías de bases de datos.

En el caso de este proyecto, no se implementan consultas complejas que usen joins ni que operen a nivel de transacciones, por lo que no existen problemas importantes, salvo la complejidad de administrar múltiples bases de datos SQL y NoSQL. Si fuera necesario usar datos de diferentes orígenes, se puede manejar la unión mediante el API gateway o utilizando el patrón de CQRS (Command Query Responsibility Segregation).

Para la implementación del patrón, una buena opción puede ser utilizar una base de datos noSQL para almacenar los mensajes que se enviarán a RabbitMQ. Por ejemplo, una base de datos de clave-valor como Redis, que es conocida por su alto rendimiento y escalabilidad. En este caso, se podrían almacenar los mensajes que se deben enviar a RabbitMQ en Redis y luego implementar un proceso en segundo plano que lea estos mensajes de Redis y los publique en RabbitMQ. Sin embargo, y debido a la sencillez del sistema, se opta por el uso de una base de datos SQL cualquiera de las indicadas en la selección de herramientas del sistema.

5. Diseño e implementación

Durante la fase de diseño e implementación, se materializa la concepción del sistema a través de la elaboración de un plan minucioso que describe la construcción del software. Esta etapa se inicia con el modelado de los servicios y, posteriormente, se detallan aspectos de la implementación y se aplican patrones de diseño pertinentes para el proyecto.

5.1 Servicio topicsubscriber

Este microservicio es muy sencillo. Simplemente, expone un *endpoint* que acepta peticiones POST vía API Rest con los datos del correo y la temática. A continuación, envía un mensaje a una cola directa al broker de mensajería.

figura 8 Diagrama servicio topicsubscriber

5.2 Servicio msgquery

Este microservicio se conecta a una base de datos PostgreSQL y realiza consultas de información de correo electrónico y temas para cada usuario. Utilizando esta información, realiza una consulta diaria a una API externa para obtener noticias relevantes y envía los resultados a una cola de mensajería. Una vez que se envía el mensaje, este microservicio recibe una comunicación y marca el correo electrónico como procesado para ese día en la base de datos. Este microservicio se ejecuta de manera autónoma sin interacción directa del usuario. La secuencia de interacciones para este microservicio se ilustra en el siguiente diagrama de secuencia.

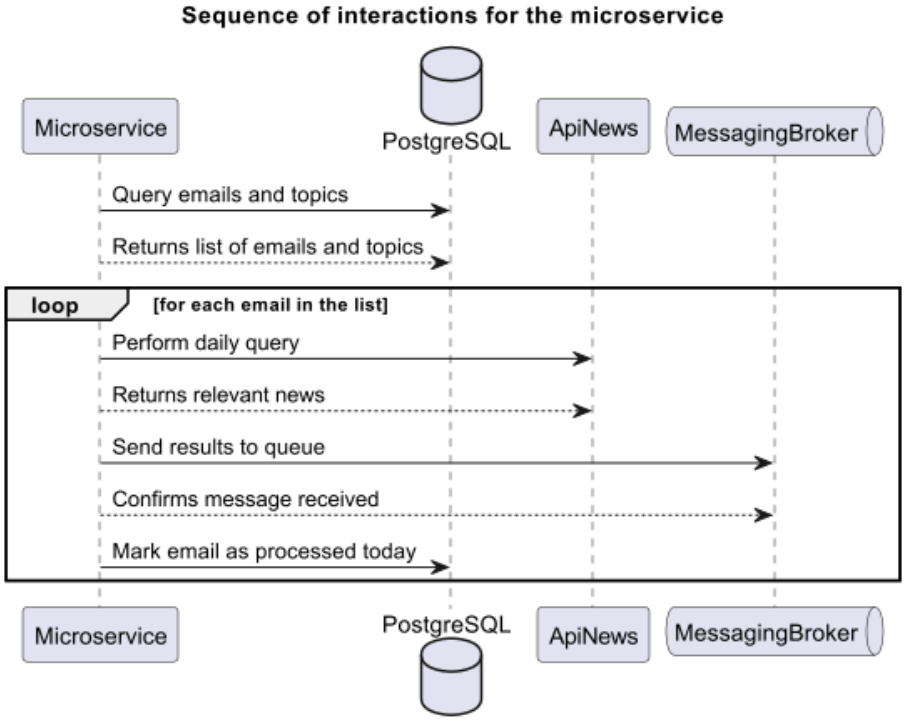


figura 9 Diagrama servicio msgquery

El proceso principal es manejado por la clase MessageSender que conecta con un broker de mensajería representado por la clase MessagingQueueService. El proceso se activa con el método privado queryNewsPeriodically() que realiza las comprobaciones pertinentes y realiza las consultas periódicas a la API de noticias.

La clase MessageSender tiene cuatro métodos principales:

- queryUsersEmailAndTopics() para obtener la información de correo electrónico y tema de personas usuarias.

- queryUsersEmailAndTopicsSent() para comprobar si se ha enviado correo de tema a las personas usuarias.
- verifyMessageStatus() para verificar el estado del mensaje.
- sendDailyNews() para enviar las noticias diarias a las personas usuarias.

La clase UserEmailAndTopic representa la información del correo electrónico y la temática escogida por un usuario.

A su vez, la clase MessagingQueueService tiene métodos para conectarse, enviar y recibir mensajes.

Por último, la clase DailyEmail guarda registro del envío de mensajes a los usuarios.

5.3 Servicio topichistory

Este microservicio sirve de almacenaje y consulta de los resultados históricos de noticias enviadas a las personas usuarias. Recibe la información mediante una cola de mensajería y expone una API REST para consultas GET. Una vez recibida la información, envía una confirmación a la cola.

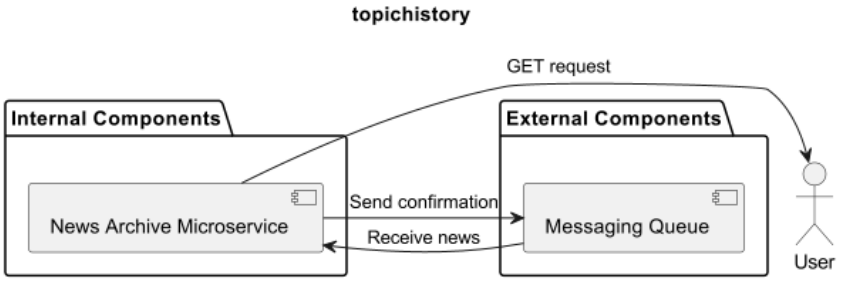


figura 10 Diagrama servicio topichistory

5.4 Identificación de los endpoints de la API

El diseño de los endpoints es crucial, ya que los endpoints son las interfaces a través de las cuales los usuarios y otros sistemas interactúan con el software. La correcta definición de los endpoints permite definir las operaciones disponibles para el usuario, los parámetros necesarios para su correcta ejecución y las respuestas esperadas. Por lo tanto, el diseño de los endpoints debe realizarse de manera cuidadosa y precisa para asegurar que el software sea fácil de usar y que cumpla con los requisitos específicos del negocio.

topicsubscriber:

- Endpoint: POST /subscribe Recibe información del cliente, el correo electrónico y el tópico al que se desea suscribir y la envía al broker de mensajería.

Ejemplo de uso

```
POST http://localhost:8080/api/user
```

```
{
  "email": "adream0@gmail.com",
  "name": "test",
  "topic": "internacional"
}
```

msgquery

- Endpoint: GET /topics/{id} Obtiene las preferencias de la persona usuaria identificada por el ID especificado.

Ejemplo de uso

```
GET http://localhost:8080/api/preferences/identificadorcorreo
```

topichistory

- Endpoint: GET /history/{id} Obtiene el historial de noticias de la persona usuaria identificada por el ID especificado.

Ejemplo de uso

```
GET http://localhost:8080/api/history/identificadorcorreo
```

5.5 Identificación de patrones a utilizar en la implementación

La siguiente lista de patrones son los principales que se van a usar en la implementación de la aplicación.

- Patrón de comunicación asincrónica mediante el uso de un broker de mensajería, que permite una mayor escalabilidad, concurrencia y estabilidad del sistema.

- Patrón de suscripción y publicación (publish/subscribe) para la comunicación entre los servicios y el broker de mensajería.

- Patrón de persistencia en base de datos relacional para los servicios que requieren almacenamiento de datos estructurados, como el histórico de noticias y las preferencias de los usuarios.

- Patrón de envío de correo electrónico mediante un servicio dedicado que se encarga de esta funcionalidad.

- Patrón API Gateway, para enrutar las peticiones desde un solo punto de acceso y facilitar la gestión de la conectividad y permisos a los servicios.

- Patrón DTO para la conversión de objetos para un uso más simplificado

- Patrón de inyección de dependencias para que un componente externo suministre las dependencias que necesita un objeto en lugar de que el objeto las cree por sí mismo.

- Patrón de inversión de dependencias, mediante el uso de interfaces.

- Patrón Builder, suministrado por Lombok, para facilitar la creación de objetos.

- Patrón de arquitectura hexagonal:se crea una capa de dominio que contiene las entidades que definen el modelo de negocio y los servicios que se encargan de llevar a cabo las acciones sobre ese modelo. En esta capa se utilizan interfaces para definir los casos de uso, de forma que su implementación pueda ser independiente de la tecnología utilizada para la persistencia de datos o para la conexión con servicios externos.

  A su vez se crea una capa de infraestructura que se encarga de implementar las interfaces definidas en la capa de dominio. Esta capa contiene los adaptadores que se encargan de conectar con las diferentes tecnologías utilizadas, como bases de datos, servicios externos o colas RabbitMQ.

  Por último, en el sistema se utiliza la inyección de dependencias para proporcionar los adaptadores necesarios para su uso. Además, se crea una capa de aplicación que expone los casos de uso identificados, a través de una API REST, utilizando controladores que transforman las peticiones HTTP en llamadas a los casos de uso definidos en la capa de dominio. Para procesar las solicitudes de manera asíncrona, se implementan colas con RabbitMQ.

6. Implementación

Durante el proceso de implementación el diseño del aplicativo ha sufrido algunos cambios. Debido a la extensión del proyecto, más grande de lo esperado se ha tenido que tomar decisiones importantes al respecto con el fin de obtener el producto mínimo viable que se deseaba desarrollar con el máximo valor añadido.

6.1 Servicio msgdispatch

Este servicio se implementa con Spring Boot y resulta del desacople del componente de correo del diseño original del sistema de consulta de API'S en dos componentes, **msgquery** para las consultas de la API externa de noticias y **msgdispatch**, para el envío del correo. Es más lógico que estas funcionalidades estén separadas, ya que presentaban problemáticas diferentes que no deben afectar entre sí y esto ha sido un hecho constatado durante el desarrollo del proyecto. Como contrapartida, se ha decidido no desarrollar el componente de consulta histórica, puesto que no aporta tanto valor como la separación de estos dos microservicios. El servicio msgdispatch utiliza dos colas, una para recibir las peticiones de envío de correo y otra para confirmar el envío. Escucha en un listener los mensajes, les da formato html y los envía a una plataforma de correo llamada **mailtrap** que se encarga de gestionar los envíos. Ambos implementan las colas de RabbitMQ mediante un archivo de configuración.

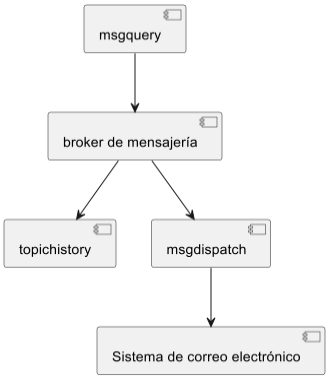


figura 11 Diagrama desacoplamiento msgdispatch

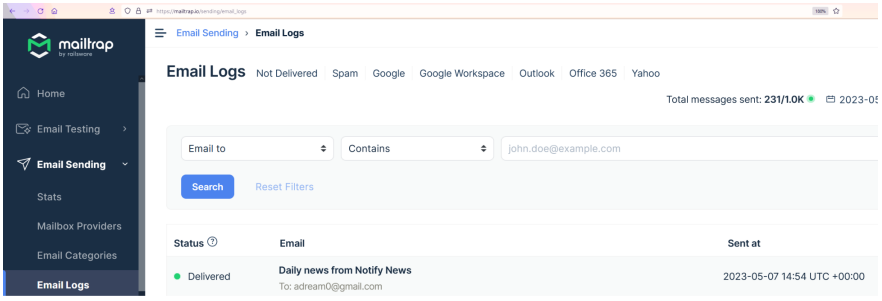


figura 12 imagen mailtrap gestor de correo

Por último, este servicio hace uso de perfiles, lo que permite mediante interfaces la inyección directa del proveedor de correo deseado según el parámetro de configuración que se utilice en el build del docker. En este caso se usa como parámetro mailtrap. También presenta un perfil para la elección de diferentes idiomas en el envío del correo, perfiles para el uso local del aplicativo en un

entorno local o en Kubernetes y dos implementaciones para la generación del mensaje de correo, una creada originalmente mediante Stringbuilders y otra que se crea con posterioridad en la que se agrega Thymeleaf como plantilla de generación del html. Se mantienen las dos implementaciones para mostrar la bondad de la utilización de las interfaces en la evolución de un proyecto.

En la de docker se explicita el uso de los perfiles indicados, mailtrap para el correo, kubernetes para el acceso a los servidores y sp como idioma mostrado en las plantillas.

A continuación se muestra el diagrama de clases del microservicio.

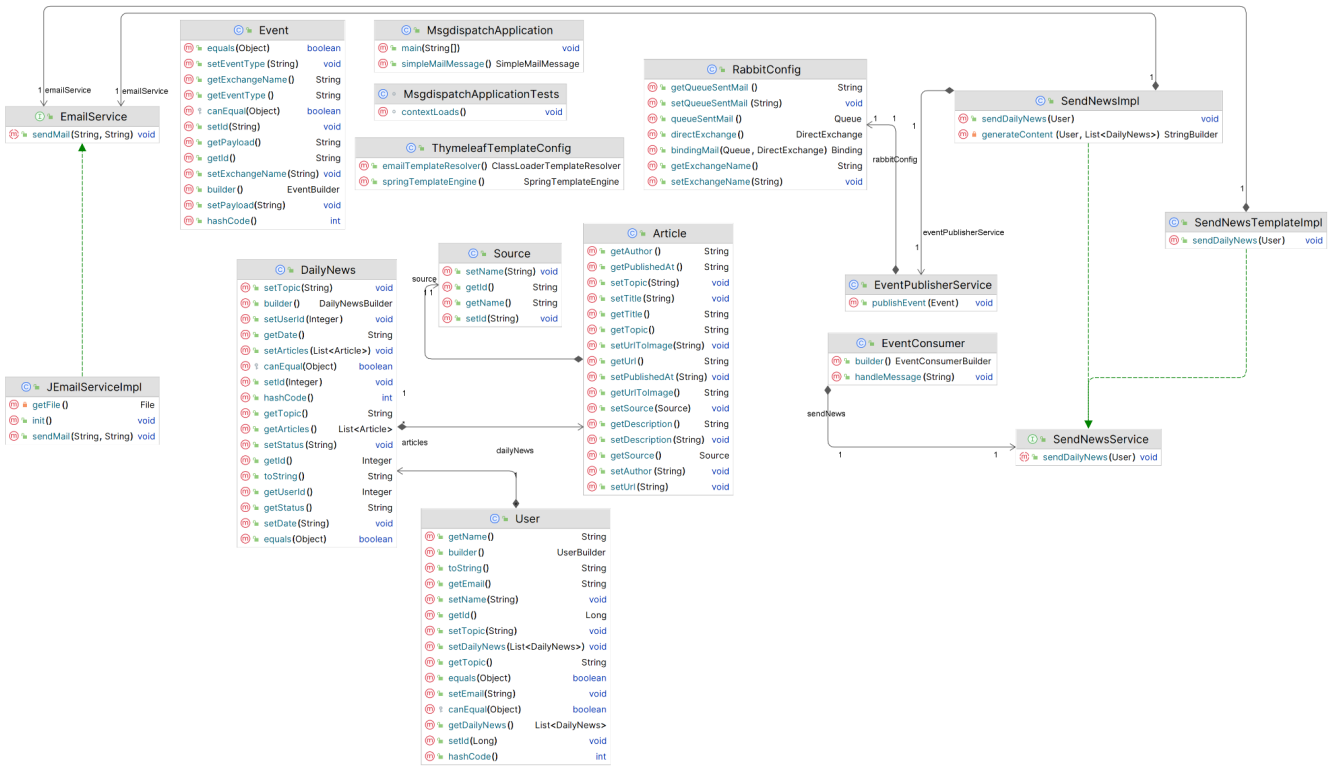


figura 13 Diagrama de clases de msgdispatch

6.2 Servicio topicsubscriber

El componente topicsubscriber ha sido implementado en C# de acuerdo con el diseño original. Este componente expone un *endpoint* que recibe peticiones HTTP POST para dar de alta a un usuario, utilizando un JSON que contiene los valores de correo electrónico, temática de interés y un atributo adicional para el nombre de usuario. Para permitir la flexibilidad en la elección del broker de mensajería, se han diseñado interfaces que permiten la inyección de diferentes implementaciones. En este caso, se han desarrollado dos posibles implementaciones, una para Kafka y otra para RabbitMQ. Finalmente, se realiza la inyección de la implementación de RabbitMQ.

En la estructura de proyecto se muestra la separación entre modelo e infraestructura, el uso de controladores y un proyecto de test. Además, se muestra el uso de interfaces en toda la aplicación.

En el anexo II se presenta en detalle la configuración. así como la activación de Swagger en el microservicio y el uso de la interfaz del productor de RabbitMQ en el archivo Program.cs.

Como extra se ha desarrollado el resto de *endpoints* CRUD. El servicio también tiene una base de datos *in memory* para poder debuguear con mayor facilidad las solicitudes.

6.3 Servicio msgquery

Este servicio representa el componente central de la orquestación y es el más complejo. Su función principal es recibir de forma asíncrona, a través de colas directas, las solicitudes provenientes de otros servicios y actuar en consecuencia. Entre sus tareas se encuentra el registro de usuarios y sus temáticas de interés, así como la realización de consultas a una API externa. Además, utiliza colas de tipo *fanout* a las cuales se suscriben otros servicios, lo que permite implementar un método de difusión para los suscriptores interesados.

En este punto, se han tomado dos decisiones de diseño relevantes. La primera ha sido realizar un cambio en la arquitectura, convirtiendo este componente en el orquestador central del sistema. En el modelo original, aunque se indicaba que no habría un desacoplamiento entre los procesos de consulta de la API y el envío de mensajes, se establecía que el componente de envío de correo **msgdispatch** sería responsable de informar al servicio histórico **topichistory** sobre el envío de mensajes.

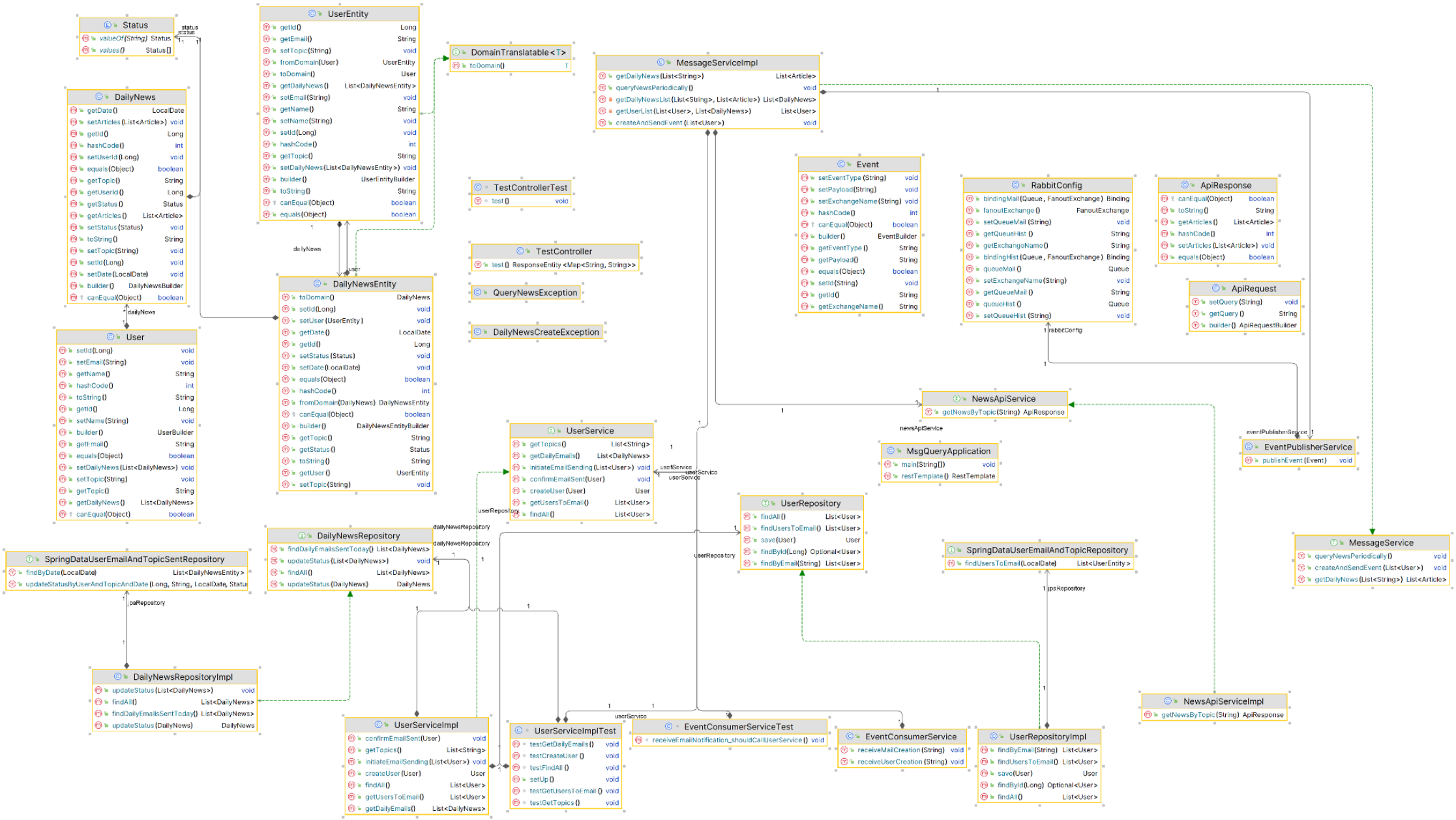
La decisión final de no incorporar el componente **topichistory** en esta iteración influyó en la toma de la decisión de mantener a este servicio como orquestador central, aunque a nivel arquitectónico considero que esta decisión no es la mejor, pues implica confirmaciones de vuelta y reenvíos de mensajes innecesarios, pero en este contexto no había otras opciones.

Por otro lado, como un elemento de valor y debido a esta decisión, se implementaron dos tipos de colas. Se utilizó una cola directa para que los servicios pudieran confirmar el resultado de sus operaciones de manera individual. Además, se incorporó otra cola que funciona en modo de difusión (broadcast), permitiendo que otros servicios se suscriban a ella. De esta manera, se envía un solo mensaje que es recibido tanto por el componente **msgdispatch** como por el futuro componente **topichistoric**. Esta configuración asegura una comunicación eficiente y efectiva entre los diferentes componentes del sistema.

En un entorno real, no tiene sentido guardar la confirmación en el histórico antes de recibir la confirmación de envío del mensaje. Sin embargo, a nivel teórico, se ha aprovechado la oportunidad de implementar dos tipos de colas diferentes simplemente a nivel ilustrativo. Es importante tener en cuenta que en un entorno real, se debería haber respetado el diseño original, donde la confirmación se registra en el histórico después de recibir la confirmación de envío del mensaje.

Debido a dificultades experimentadas en la integración con el sistema RabbitMQ dentro del clúster de kubernetes, el servicio **msgquery** funciona actualmente con una base de datos *in memory*. El cambio a otro tipo de base de datos es sencillo a nivel de programa pero, necesita una configuración extra en el clúster que también llevaría mayor tiempo de desarrollo

figura 14 Diagrama msgquery


Status
valueOf(String) Status
values() Status[]
UserEntity
getId() Long
getEmail() String
setTopic(String) void
fromDomain(User) UserEntity
toDomain() User
getDailyNews() List<DailyNewsEntity>
setEmail(String) void
getName() String
setName(String) void
setId(Long) void
hashCode() int
getTopic() String
setDailyNews(List<DailyNewsEntity>) void
builder() UserEntityBuilder
toString() String
canEqual(Object) boolean
equals(Object) boolean
DomainTranslatable<T>
toDomain() T
MessageServiceImpl
getDailyNews(List<String>) List<Article>
queryNewsPeriodically() void
getDailyNewsList(List<String>, List<Article>) List<DailyNews>
getUserList(List<User>, List<DailyNews>) List<User>
createAndSendEvent(List<User>) void
DailyNews
getDate() LocalDate
setArticles(List<Article>) void
getId() Long
hashCode() int
setUserId(Long) void
equals(Object) boolean
getTopic() String
getUserId() Long
getStatus() Status
getArticles() List<Article>
setStatus(Status) void
toString() String
setTopic(String) void
setId(Long) void
setDate(LocalDate) void
builder() DailyNewsBuilder
canEqual(Object) boolean
TestControllerTest
test() void
TestController
test() ResponseEntity<Map<String, String>>
QueryNewsException
DailyNewsCreateException
Event
setEventType(String) void
setPayload(String) void
setExchangeName(String) void
hashCode() int
canEqual(Object) boolean
builder() EventBuilder
getEventType() String
getPayload() String
equals(Object) boolean
setId(String) void
getId() String
getExchangeName() String
RabbitConfig
bindingMail(Queue, FanoutExchange) Binding
fanoutExchange() FanoutExchange
setQueueMail(String) void
getQueueHist() String
getExchangeName() String
bindingHist(Queue, FanoutExchange) Binding
queueMail() Queue
setExchangeName(String) void
getQueueMail() String
queueHist() Queue
setQueueHist(String) void
ApiResponse
canEqual(Object) boolean
toString() String
getArticles() List<Article>
hashCode() int
setArticles(List<Article>) void
equals(Object) boolean
ApiRequest
setQuery(String) void
getQuery() String
builder() ApiRequestBuilder
User
setId(Long) void
setEmail(String) void
getName() String
hashCode() int
toString() String
getId() Long
setName(String) void
builder() UserBuilder
getEmail() String
equals(Object) boolean
setDailyNews(List<DailyNews>) void
setTopic(String) void
getTopic() String
getDailyNews() List<DailyNews>
canEqual(Object) boolean
DailyNewsEntity
toDomain() DailyNews
setId(Long) void
setUser(UserEntity) void
getDate() LocalDate
getId() Long
setStatus(Status) void
setDate(LocalDate) void
equals(Object) boolean
hashCode() int
fromDomain(DailyNews) DailyNewsEntity
canEqual(Object) boolean
builder() DailyNewsEntityBuilder
getTopic() String
getStatus() Status
toString() String
getUser() UserEntity
setTopic(String) void
NewsApiService
getNewsByTopic(String) ApiResponse
UserService
getTopics() List<String>
getDailyEmails() List<DailyNews>
initiateEmailSending(List<User>) void
confirmEmailSent(User) void
createUser(User) User
getUsersToEmail() List<User>
findAll() List<User>
MsgQueryApplication
main(String[]) void
restTemplate() RestTemplate
EventPublisherService
publishEvent(Event) void
UserRepository
findAll() List<User>
findUsersToEmail() List<User>
save(User) User
findById(Long) Optional<User>
findByEmail(String) List<User>
SpringDataUserEmailAndTopicRepository
findUsersToEmail(LocalDate) List<UserEntity>
MessageService
queryNewsPeriodically() void
createAndSendEvent(List<User>) void
getDailyNews(List<String>) List<Article>
SpringDataUserEmailAndTopicSentRepository
findByDate(LocalDate) List<DailyNewsEntity>
updateStatusByUserAndTopicAndDate(Long, String, LocalDate, Status)
DailyNewsRepository
findDailyEmailsSentToday() List<DailyNews>
updateStatus(List<DailyNews>) void
findAll() List<DailyNews>
updateStatus(DailyNews) DailyNews
DailyNewsRepositoryImpl
updateStatus(List<DailyNews>) void
findAll() List<DailyNews>
findDailyEmailsSentToday() List<DailyNews>
updateStatus(DailyNews) DailyNews
UserServiceImpl
confirmEmailSent(User) void
getTopics() List<String>
initiateEmailSending(List<User>) void
createUser(User) User
findAll() List<User>
getUsersToEmail() List<User>
getDailyEmails() List<DailyNews>
UserServiceImplTest
testGetDailyEmails() void
testCreateUser() void
testFindAll() void
setUp() void
testGetUsersToEmail() void
testGetTopics() void
EventConsumerServiceTest
receiveEmailNotification_shouldCallUserService() void
EventConsumerService
receiveMailCreation(String) void
receiveUserCreation(String) void
UserRepositoryImpl
findByEmail(String) List<User>
findUsersToEmail() List<User>
save(User) User
findById(Long) Optional<User>
findAll() List<User>
NewsApiServiceImpl
getNewsByTopic(String) ApiResponse

6.4 Proceso CI/CD

El proceso de Pipelines funciona según lo esperado. Cuando se realiza un commit en la rama principal, se lleva a cabo la compilación y ejecución de pruebas tanto para C# como para Java. Posteriormente, se registran las imágenes Docker y se realiza la integración en caso de que el proceso sea exitoso.

Para el registro de las imágenes, se emplea un proceso de doble etapa. Primero se crea la imagen inicial y luego se genera una nueva imagen a partir de ella, utilizando el tag "latest". Esto permite mantener un registro histórico de las imágenes que se han creado a lo largo del tiempo.

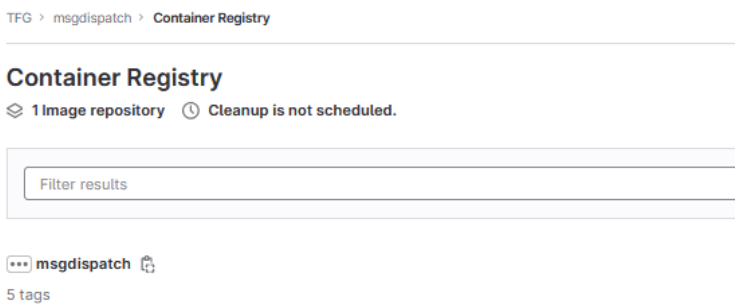


figura 15 Registro de containers en gitlab

Hay que tener en cuenta que en GitLab también se debe crear un access token con permiso de lectura sobre el registro de contenedores para que así Kubernetes pueda tener acceso a la imagen.

figura 16 Creación de access token

Las imágenes dockers creadas por el pipeline son usadas por el clúster de Kubernetes. En este punto del desarrollo falta la configuración del deploy en producción en las plataformas de la nube que sería el último paso. Para ello se intentará hacer uso de **Terraform.**

En el anexo II se muestra un ejemplo con el Pipeline de C#. El Pipeline de Java que se puede encontrar en el anexo III es similar pero utiliza Maven.

6.5 RabbitMQ

Para el broker de mensajería se utiliza la imagen docker rabbitmq:3-management que da acceso a un panel de administración. Las colas se crean y gestionan de forma automática por los servicios que se conectan a RabbitMQ

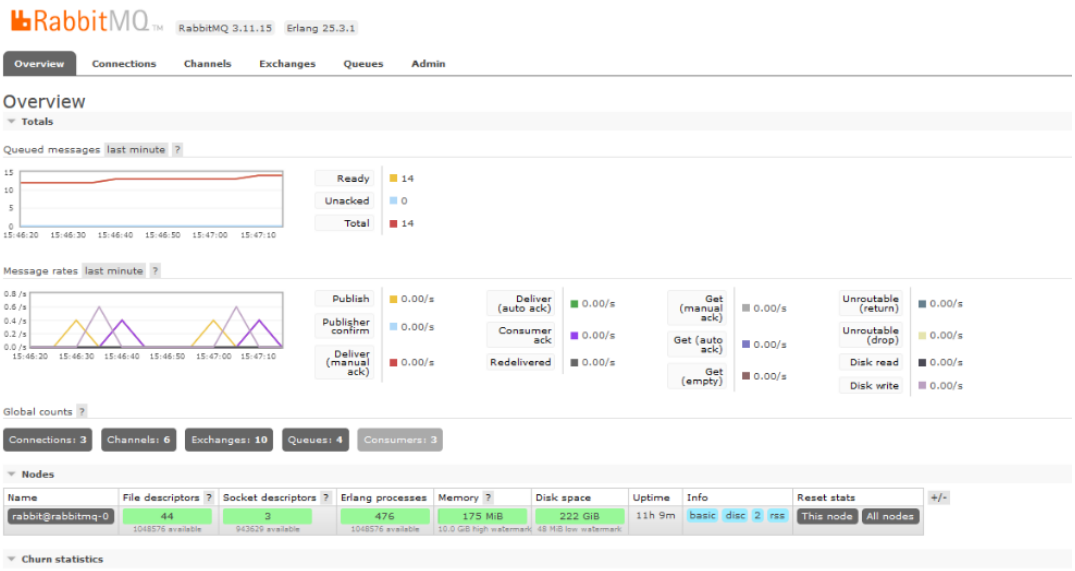


figura 17 Imagen panel administración de RabbitMQ

En rabbitMQ se implementan las siguientes colas

dailynews.createuser.queue: cola directa de topicsubscriber a msgquery con los datos del usuario creado.

dailynews.news.sendmail.queue, dailynews.news.sendhist.queue: dos colas que funcionan por suscripción a un *fanout exchange* y que es por donde se envían las noticias obtenidas de la API al servicio de correo y al servicio de históricos.

queue-sent-mail: dailynews.news.sentmail.queue: una cola directa de vuelta al servicio msgquery donde se informa de que el email ha sido enviado para que msgquery lo marque como procesado.

En Spring Boot, además de la configuración del servidor de RabbitMQ, se utiliza una clase de configuración con Beans que toma la parametrización de su archivo properties tal y como se puede ver en el anexo III.

En C# el proceso de creación de colas es algo diferente, tomando la configuración del archivo appsettings.json tal y como se puede ver en el anexo II.

6.6 Clúster de Kubernetes

El sistema se implementa en windows por lo que el clúster de Kubernetes se puede hacer correr directamente habilitando la opción de Kubernetes de Docker Desktop. Sin embargo, esta elección mostró cierta inestabilidad durante la ejecución del proyecto, así que finalmente se optó por la instalación de Kind.

Originalmente, se realizan las operaciones en el clúster mediante consola, pero una vez se conoce el entorno se comienza a usar el Dashboard de Kubernetes y así las operaciones pasan

de hacerse desde línea de comandos al entorno gráfico. La mayor dificultad que tiene este es el de que cada cierto tiempo se necesita crear un token de usuario para autenticarse.

Debido a la inexistencia de balanceador de carga local, la implementación los services ha sido creada en modo clusterIP, utilizando posteriormente una técnica de *portforwarding* para realizar las pruebas. Sin en embargo, en el despliegue en la nube el tipo se establece como loadbalancer, obteniendo de esta forma un acceso directo por IP a los servicios.

Actualmente, en el clúster implementado corren todos los servicios desarrollados, topicsubscriber, msgdispatch, msgquery y también una instancia de RabbitMQ.

En cuanto a RabbitMQ, se ha tomado la decisión de que corra también dentro del clúster en su propio Pod. Una de las principales razones de esta decisión es que el hecho de correr el Pod en el clúster es más económico y probablemente más sencillo de configurar en contraposición al uso de un servidor externo. La parte más compleja de la configuración ha sido precisamente poner RabbitMQ en funcionamiento, tarea para la que barajaron diferentes estrategias.

Una de las estrategias posibles era el uso de Cluster Kubernetes Operator, una extensión de Kubernetes que provee de un controlador que permite la definición y administración de forma prácticamente automatizada mediante una personalización usando un modelo de aplicación declarativa que automatiza y simplifica la gestión del clúster. Localmente, no he sido capaz de montar este sistema y quizás se deba al hecho de que Kubernetes esté enfocado al uso en la nube, complica en ciertos momentos la gestión los procesos de pruebas en contextos locales, en este caso no ha sido posible montar el Physical Volume Provisioner localmente.

Aun así, hay mucha documentación al respecto, tanto en el sitio oficial de RabbitMQ como en Github. Por otro lado, un recurso excelente, que explica de manera clara y sencilla la instalación del clúster de RabbitMQ, es el que se puede visionar en RabbitMQ on Kubernetes for beginners. Este ejemplo se ha probado y efectivamente monta un clúster de RabbitMQ con tres nodos que puede estar operativo con poco trabajo. El ejemplo también da acceso a un repositorio GIT con todos los fuentes que permiten el estudio de este caso de uso.

Sin embargo, la implementación de un sistema más complejo escapa al alcance de la prueba conceptual que se desea realizar en el presente trabajo, por lo que se ha optado por usar simplemente un pod con una imagen de RabbitMQ con un kind StateFulset expuesto mediante un servicio que hace uso del proveedor de persistencia local.

Para la implementación del clúster se han creado los siguientes tipos de objetos:

- **Secrets:** para crear el secret de GitLab se usa el access token con permiso de lectura sobre el registro de contenedores que se creó en GitLab.
  También se puede aplicar un fichero secret en formato declarativo. Para crear los secretos, estos se pueden codificar en base64.

- **Deployments**: un archivo típico de **Deploy** presenta un formato que indica básicamente su nombre, sus puertos internos, selectores para su identificación, la imagen con la que trabaja y variables de entorno, en este caso asociadas a los secretos que permiten su conexión con Gitlab y RabbitMQ

- **Services:** para exponer los servicios fuera del clúster en una configuración típica en la nube, un service sería de tipo loadbalancer. Para un clúster local sin balanceador se puede usar ClusterIP para a continuación poder acceder a ellos mediante portforwarding.

- **Pods**: definidos los deployments y los servicios que los exponen, se puede acceder a los pods y escalar el sistema para exponer el número de instancias que se desee para cada microservicio. Un POD eventualmente podría contener varios microservicios ejecutando varios contenedores, ya que trabajan como una unidad lógica. En la definición de este sistema se desea que cada POD funcione de forma operativa independiente.

- **StatefulSet**: para montar RabbitMQ se establece una configuración de tipo StatefulSet. Al utilizar esta configuración, se garantiza la alta disponibilidad y escalabilidad del servicio. Un StatefulSet es un controlador de Kubernetes que se utiliza para gestionar aplicaciones con estado, como RabbitMQ. Proporciona una identidad única y persistente para cada instancia del servicio, lo que permite el almacenamiento persistente de los datos y facilita la replicación y escalado de las instancias. Para más información, consultar el anexo asociado de RabbitMQ

- ***Persistent Volume y Persistent Volume Claims***: utilizados para mantener la persistencia de RabbitMQ y para una base de datos mysql de uso futuro. En este caso se utiliza un proveedor local

Por simplicidad se trabaja en el namespace por defecto.

En resumen, el sistema utiliza un archivo de deployment y otro de services por cada microservicio más un statefull para RabbitMQ junto a un PV y un PVC. También utiliza dos Secrets, uno para gestionar las credenciales de acceso a RabbitMQ y otro para permitir el acceso mediante un token al registro de contenedores de GitLab.

Actualmente, el clúster funciona tanto en modo local como en Google Cloud, e incluye la creación del clúster en GCD mediante Terraform y la creación y actualización de los pods mediante el sistema de CI/CD. Tanto la creación de RabbitMQ como la de los secrets se realiza de forma manual y se mantiene en directorios privados, ya que su implementación en el clúster se realiza únicamente de forma eventual, en la creación del sistema o en un caso de cambio de credenciales.

Para más información se sobre la configuración de Kubernetes se puede consultar el Anexo VIII Kubernetes

7. pruebas y validación

Prueba del sistema de CI/CD

A continuación, se procede a realizar una prueba integral de todo el proceso de CI/CD realizando un cambio en el código para mostrar la velocidad con la que se puede realizar un despliegue nuevo ejecutando un cambio en uno de los servicios.

Caso de uso: Según los logs la ejecución del proceso de consulta de mensajes se realiza cada 30 segundos y el equipo de desarrollo considera que este intervalo temporal es demasiado pequeño. Como estamos en un entorno de CI/CD vamos a trabajar directamente sobre la rama main del servicio. Este proceso es gestionado por el microservicio msgquery y según los logs su estado es el siguiente:

```
2023-05-09T18:43:07.244Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T18:43:37.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T18:43:37.243457800
2023-05-09T18:43:37.243Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_se
2023-05-09T18:43:37.244Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T18:44:07.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T18:44:07.243385
2023-05-09T18:44:07.243Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_se
2023-05-09T18:44:07.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T18:44:37.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T18:44:37.243496500
2023-05-09T18:44:37.243Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_se
2023-05-09T18:44:37.244Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T18:45:07.242Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T18:45:07.242822800
2023-05-09T18:45:07.243Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_se
2023-05-09T18:45:07.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T18:45:37.243Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T18:45:37.243103100
```

Directamente en GitLab se efectúa el cambio del valor temporal de ejecución del proceso y se realiza un commit sobre la rama master.

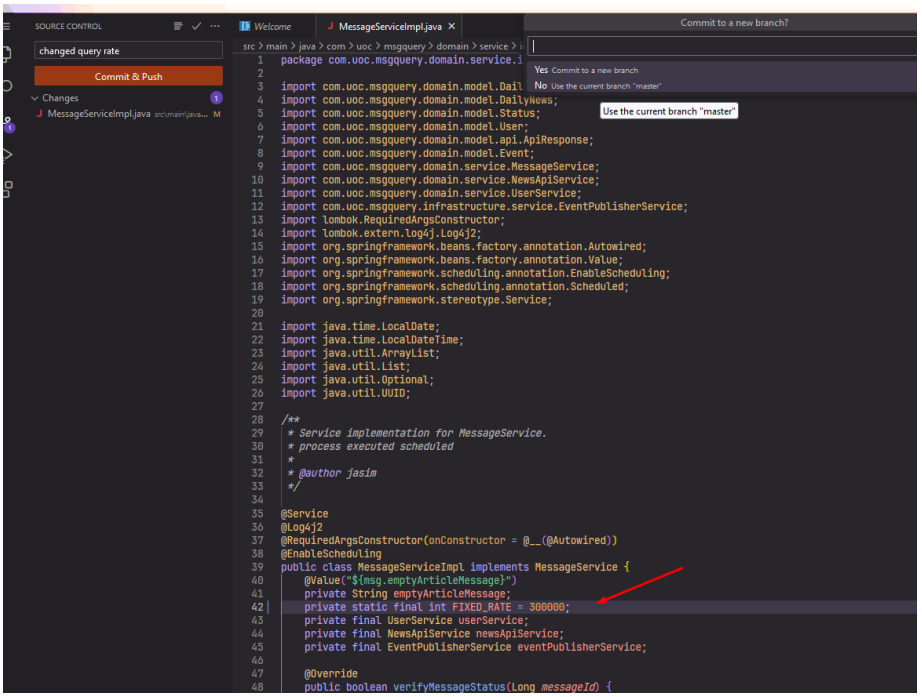


figura 19 Imagén con valor de variable en msgquery a modificar

En este momento se activa el proceso de CI/CD de forma automática.

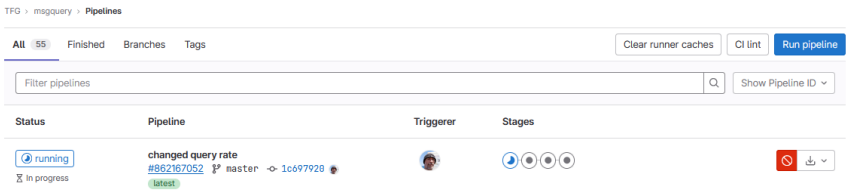


figura 20 Imagen del proceso de pipeline activado

Comienza el proceso de ejecución de todos los jobs. En primer lugar, el sistema recompila.

```
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-lang3/3.7/commons-lang3-3.7.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/checkerframework/checker-compat-qual/2.5.5/checker-compat-qual-2.5.5.jar (5.9 kB at 6.3 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/com/google/errorprone/error_prone_annotations/2.3.4/error_prone_annotations-2.3.4.jar (14 kB at 15 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/com/google/j2objc/j2objc-annotations/1.3/j2objc-annotations-1.3.jar (8.8 kB at 9.2 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-lang3/3.7/commons-lang3-3.7.jar (500 kB at 493 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/com/google/guava/guava/28.2-android/guava-28.2-android.jar (2.6 MB at 2.5 MB/s)
[INFO] Replacing main artifact with repackaged archive
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:23 min
[INFO] Finished at: 2023-05-09T18:58:09Z
[INFO] ------------------------------------------------------------------------
Uploading artifacts for successful job
Uploading artifacts...
target/*.jar: found 1 matching artifact files and directories
Uploading artifacts as "archive" to coordinator... 201 Created  id=4254222604 responseStatus=201 Created token=64_ggScj
Cleaning up project directory and file based variables
Job succeeded
```

A continuación se pasan los tests positivamente.

```
pic_sent d1_0 on u1_0.email=d1_0.email and u1_0.topic=d1_0.topic where d1_0.id is null or d1_0.date!=? or d1_0.status!='IN_PROGRESS'
Hibernate: select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_sent d1_0 on u1_0.email=d1_0.email and u1_0.topic=d1_0.topic w
r d1_0.date!=? or d1_0.status!='IN_PROGRESS'
2023-05-09T19:00:12.041Z DEBUG 82 --- [           main] org.hibernate.SQL                        : select d1_0.id,d1_0.date,d1_0.email,d1_0.status,d1_0.topic from u
1_0 where d1_0.date=?
Hibernate: select d1_0.id,d1_0.date,d1_0.email,d1_0.status,d1_0.topic from user_email_topic_sent d1_0 where d1_0.date=?
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.797 s - in com.uoc.msgquery.domain.service.impl.UserServiceImplTest
2023-05-09T19:00:12.092Z  INFO 82 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:20 min
[INFO] Finished at: 2023-05-09T19:00:12Z
[INFO] ------------------------------------------------------------------------
Cleaning up project directory and file based variables
Job succeeded
```

Ahora, en el siguiente paso, se crea la imagen de docker nueva y se le agrega el tag.

```
8df19c10a6b2: Preparing
34f7184834b2: Preparing
5836ece05bfd: Preparing
72e830a4dff5: Preparing
5836ece05bfd: Layer already exists
34f7184834b2: Layer already exists
72e830a4dff5: Layer already exists
8df19c10a6b2: Pushed
1c697920-4254222608: digest: sha256:3fdcf4a803cc6f595ae3d2ec42513fda5b576c651b44941c605ba3c3f07a5c43 size: 116
$ docker tag $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG $CI_REGISTRY/${CI_PROJECT_PATH}:latest
$ docker push $CI_REGISTRY/${CI_PROJECT_PATH}:latest
The push refers to repository [registry.gitlab.com/tfg1018711/msgquery]
8df19c10a6b2: Preparing
34f7184834b2: Preparing
5836ece05bfd: Preparing
72e830a4dff5: Preparing
5836ece05bfd: Layer already exists
34f7184834b2: Layer already exists
72e830a4dff5: Layer already exists
8df19c10a6b2: Layer already exists
latest: digest: sha256:3fdcf4a803cc6f595ae3d2ec42513fda5b576c651b44941c605ba3c3f07a5c43 size: 1163
Cleaning up project directory and file based variables
Job succeeded
```

En último lugar, se envía el cambio al proveedor externo, proceso que no está implementado.

```
Using docker image sha256:849a2a2d4242ade1b2f39545bb08af9d6cf6d94596b1b33743260751f78e1f6c for maven:3.8.4-openjdk-17-slim with digest
887b637c869a50c8019cab86f8 ...
Preparing environment
Running on runner-j1aldqxs-project-44340989-concurrent-0 via runner-j1aldqxs-shared-1683658866-57a574bc...
Getting source from Git repository
$ eval "$CI_PRE_CLONE_SCRIPT"
Fetching changes with git depth set to 20...
Initialized empty Git repository in /builds/tfg1010711/msgquery/.git/
Created fresh repository.
Checking out 1c697920 as detached HEAD (ref is master)...
Skipping Git submodules setup
Downloading artifacts
Downloading artifacts for build-job (4254222604)...
Downloading artifacts from coordinator... ok        host=storage.googleapis.com id=4254222604 responseStatus=200 OK token=64_Xshhe
Executing "step_script" stage of the job script
Using docker image sha256:849a2a2d4242ade1b2f39545bb08af9d6cf6d94596b1b33743260751f78e1f6c for maven:3.8.4-openjdk-17-slim with digest
887b637c869a50c8019cab86f8 ...
$ echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
This job deploys something from the master branch.
Cleaning up project directory and file based variables
Job succeeded
```

En GitLab se comprueba que la imagen se ha creado de forma correcta.

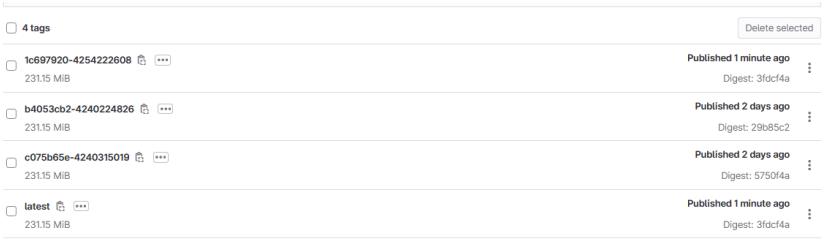


figura 21 Captura de histórico de contenedores creados en el registro de contenedores

Ahora, se reinicia el deployment en el clúster para que el sistema tome la nueva imagen docker

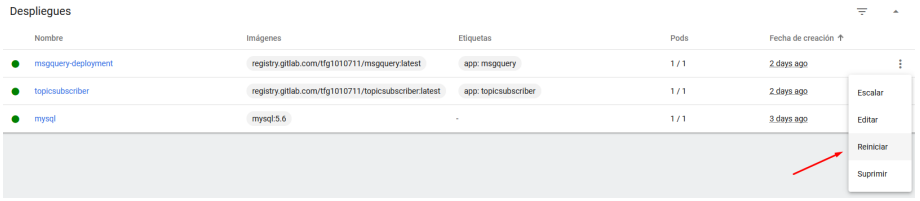


figura 22 Captura redespliegue del POD

Como resultado vemos que ahora el sistema realiza las consultas cada 5 minutos.

::

```
 :: Spring Boot ::                (v3.0.4)
2023-05-09T19:05:42.697Z  INFO 1 --- [           main] com.uoc.msgquery.MsgQueryApplication     : Starting MsgQueryApplication v0.0.1-SNAPSHOT using Java 17-ea with PID 1 (/app.jar started by root in
2023-05-09T19:05:42.700Z  INFO 1 --- [           main] com.uoc.msgquery.MsgQueryApplication     : The following 1 profile is active: "sp"
2023-05-09T19:05:43.713Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2023-05-09T19:05:43.788Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 67 ms. Found 2 JPA repository interfaces.
2023-05-09T19:05:44.568Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8087 (http)
2023-05-09T19:05:44.584Z  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-05-09T19:05:44.584Z  INFO 1 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.5]
2023-05-09T19:05:44.725Z  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-05-09T19:05:44.727Z  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1982 ms
2023-05-09T19:05:45.357Z  INFO 1 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2023-05-09T19:05:45.429Z  INFO 1 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.1.7.Final
2023-05-09T19:05:45.924Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2023-05-09T19:05:46.248Z  INFO 1 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:testdb user=SA
2023-05-09T19:05:46.250Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2023-05-09T19:05:46.281Z  INFO 1 --- [           main] SQL dialect                              : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2023-05-09T19:05:47.216Z DEBUG 1 --- [           main] org.hibernate.SQL                        : drop table if exists user_email_topic cascade
2023-05-09T19:05:47.220Z DEBUG 1 --- [           main] org.hibernate.SQL                        : drop table if exists user_email_topic_sent cascade
2023-05-09T19:05:47.223Z DEBUG 1 --- [           main] org.hibernate.SQL                        : create table user_email_topic (id bigint generated by default as identity, email varchar(255), topic
2023-05-09T19:05:47.231Z DEBUG 1 --- [           main] org.hibernate.SQL                        : create table user_email_topic_sent (id bigint generated by default as identity, date date, email varc
2023-05-09T19:05:47.233Z DEBUG 1 --- [           main] org.hibernate.SQL                        : alter table if exists user_email_topic add constraint UKd2l4dd7y61yisjtg4oedd8pjo unique (email, topi
2023-05-09T19:05:47.234Z DEBUG 1 --- [           main] org.hibernate.SQL                        : alter table if exists user_email_topic_sent add constraint UKl7y84wjmr77fme2heh04e5hkk unique (email,
2023-05-09T19:05:47.237Z  INFO 1 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.
2023-05-09T19:05:47.245Z  INFO 1 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-05-09T19:05:48.163Z  WARN 1 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during vi
2023-05-09T19:05:48.804Z  WARN 1 --- [           main] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (please add some templates, check your Thymeleaf
2023-05-09T19:05:49.025Z  INFO 1 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint(s) beneath base path '/actuator'
2023-05-09T19:05:49.140Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8087 (http) with context path ''
2023-05-09T19:05:49.144Z  INFO 1 --- [           main] o.s.a.r.c.CachingConnectionFactory       : Attempting to connect to: [rabbitmq.default.svc.cluster.local:5672]
2023-05-09T19:05:49.211Z  INFO 1 --- [           main] o.s.a.r.c.CachingConnectionFactory       : Created new connection: rabbitConnectionFactory#504497fa:0/SimpleConnection@6b8bdcc6 [delegate=amqp:/
2023-05-09T19:05:49.273Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T19:05:49.272063300
2023-05-09T19:05:49.273Z  INFO 1 --- [           main] com.uoc.msgquery.MsgQueryApplication     : Started MsgQueryApplication in 7.041 seconds (process running for 7.53)
2023-05-09T19:05:49.361Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_sent d1_0
2023-05-09T19:05:49.393Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
2023-05-09T19:10:49.271Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : Querying news periodically: 2023-05-09T19:10:49.271078600
2023-05-09T19:10:49.273Z DEBUG 1 --- [   scheduling-1] org.hibernate.SQL                        : select u1_0.id,u1_0.email,u1_0.topic from user_email_topic u1_0 left join user_email_topic_sent d1_0
2023-05-09T19:10:49.273Z  INFO 1 --- [   scheduling-1] c.u.m.d.service.impl.MessageServiceImpl  : No users found for sending emails.
```

Si se revisan los logs se comprueba que en el log inicial del pod se muestra la ejecución inicial a las 18:45:37 segundos. Realizando la tarea del cambio del desarrollo, las comprobaciones y el reinicio del pod vemos que tras el nuevo cambio el sistema está operativo a las 19:05:49.

Prueba funcional del sistema en entorno local

A continuación se procede a realizar una prueba integral del proceso de envío de mensaje.
En primer lugar, se toma nota de los puertos que utilizan los servicios para hacer un *portforward,* ya localmente no existe un balanceador y los servicios funcionan en modo ClusterIP.

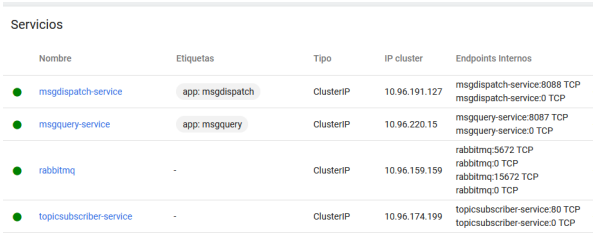
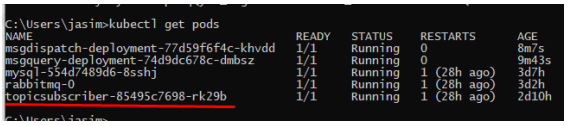
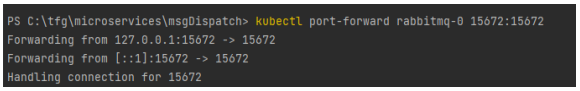


figura 23 Imagen servicios operativos

Se comprueba que se puede acceder al panel de administración de Rabbit por el puerto 15672 y que el puerto 80 es el que sirve de acceso para el servicio topicsubscriber, del cual tenemos que conocer su identificador, el cual obtenemos a partir del nombre del POD.



Mediante un par de consolas se realizan las redirecciones de puertos pertinentes.



Ahora se comprueba si hay alguna noticia sobre la UOC. Se procede a la suscripción en el sistema mediante el uso de Postman enviando una solicitud al puerto 8080 en la noticia de interés deseado indicando una dirección de correo.

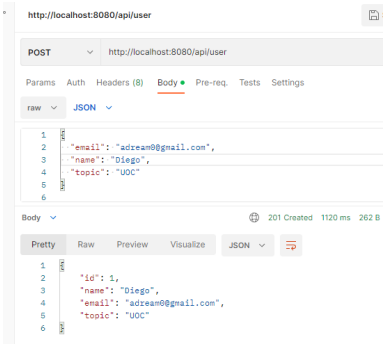
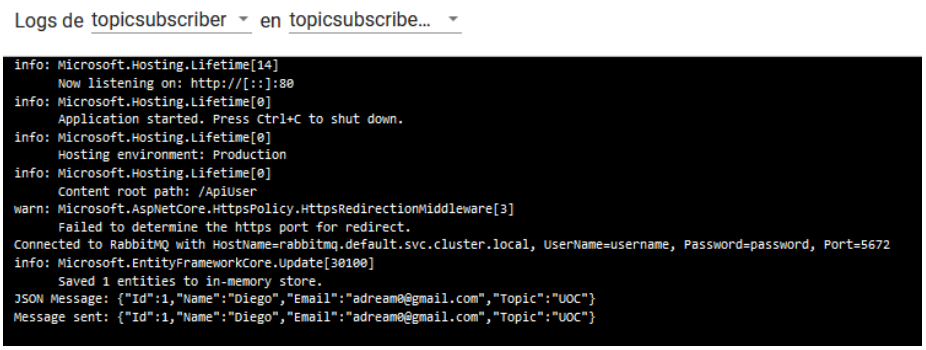


figura 24 Imagen solicitud JSON con POSTMAN

Para validar la bondad del proceso, en primer lugar se comprueba el log de topicsubscriber para ver que se ha realizado el envío con exito y se comprueba que el resultado es positivo.



Ahora se comprueba el log de gestión de API msgquery y se comprueba que se ha recibido el mensaje de topicsubscriber, que este se ha enviado msgdispatch y que se ha recibido respuesta de envío.

También se comprueba el log del Pod del servicio de envío de correo y se valida el envío correcto del mismo.

Por último, se comprueba la recepción del correo electrónico con la información de la temática solicitada en el correo electrónico.

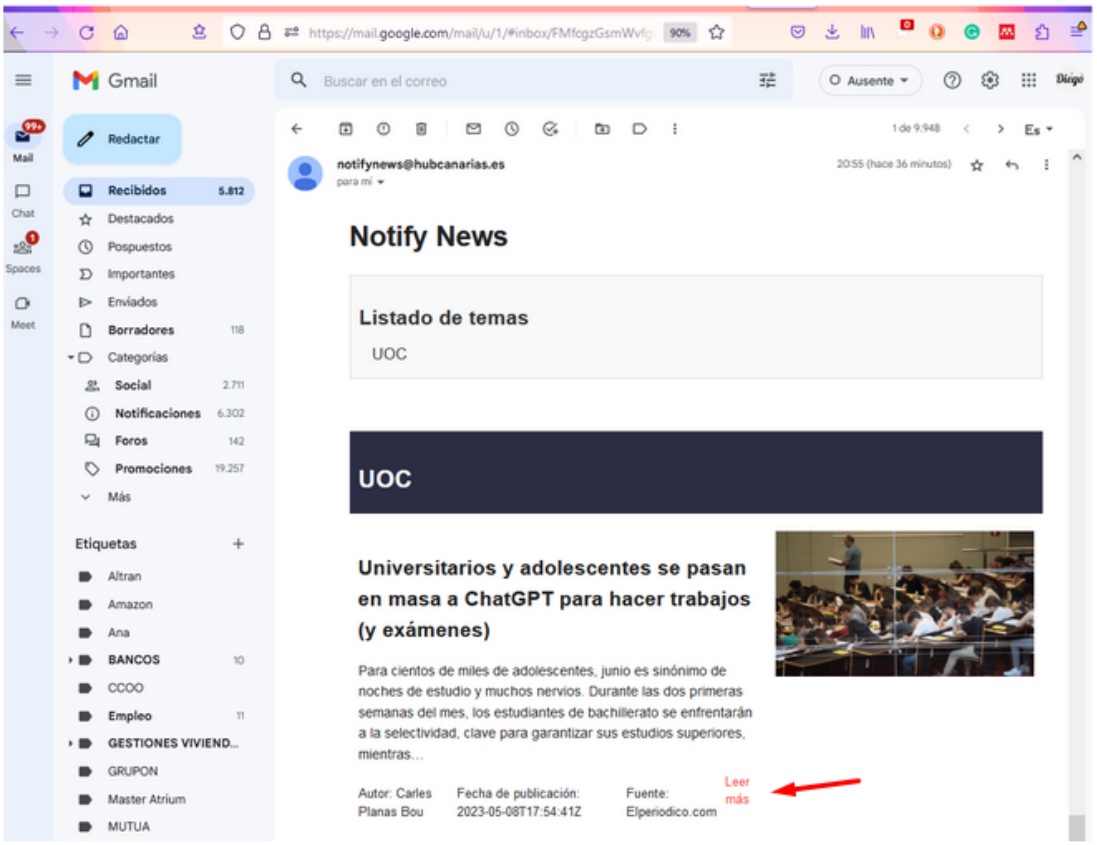


figura 25 Imagen correo enviado a detinatario

Se puede acceder a la fuente original de la noticia siguiendo el Link indicado en la imagen anterior. https://www.elperiodico.com/es/sociedad/20230508/chatgpt-universidad-escuelas-inteligencia-artificial-estudiantes-deberes-examenes-86837251

Las pruebas del sistema en la nube quedan registradas en los vídeos adjuntos al proyecto.

8. Conclusiones

La integración continua y la entrega continua (CI/CD) se han convertido en una parte esencial del proceso de desarrollo de software en la actualidad. Con la creciente popularización de la dockerización, ha habido una proliferación de herramientas que permiten la combinación e integración de todos los elementos necesarios para construir un producto con arquitecturas modernas. Las personas que trabajan en el ámbito del diseño y arquitectura del software, tienen ahora la libertad de elegir entre varias plataformas para alojar sus repositorios de código fuente, tanto las clásicas que se han dedicado a la gestión de repositorios de código fuente y que ahora han avanzado un paso más y permiten la creación de pipelines y registro de imágenes como los grandes proveedores de servicios en la nube Azure, Amazon, Google u Oracle, por nombrar algunos, los cuales han desarrollado herramientas para cubrir todo el proceso de diseño y creación del software incluyendo también la gestión de metodologías ágiles.

Además, la flexibilidad que ofrece la dockerización permite a los equipos de desarrollo ubicar los componentes necesarios de forma flexible en diferentes entornos según sus necesidades. Por ejemplo, un equipo de desarrollo puede tener su repositorio de código fuente en Azure, sus imágenes de Docker registradas en Docker Hub y utilizar un pipeline en GitLab para orquestar todo el proceso de construcción y entrega continua. En este sentido, se puede configurar un sistema a modo de puzle, donde se pueden ubicar los componentes necesarios en diferentes plataformas según las necesidades y requisitos y posteriormente relacionarlos para crear un proceso de desarrollo de software eficiente y escalable.

La planificación original del proyecto ha sido fundamental, tanto en el establecimiento de los hitos como en las fases de diseño y definición de la arquitectura, la elección de los componentes y las herramientas. Esta planificación ha tenido resultados francamente positivos, pues tener una hoja de ruta concreta y un punto de destino claro permite poder tomar decisiones delicadas en momentos cruciales del proyecto y así poder diferenciar lo que es fundamental de lo que es accesorio. Visto con perspectiva, el proyecto ha sido muy ambicioso, con muchas tareas exigentes, necesitadas de procesos de aprendizaje, de prueba error, de búsqueda de recursos, de ejemplos y de adaptación. Ha habido momentos de *burnout,* pero aun así se han intentado tomar en cada momento las decisiones más adecuadas para conseguir un producto mínimo funcional que cumpliera con los fundamentos básicos del ejercicio que se deseaba realizar. En ese sentido estoy profundamente satisfecho tanto del esfuerzo realizado, así como de los resultados, aun reconociendo que no se cumplieran todos los objetivos, pero en sí, todo este trabajo ha sido un camino de superación, de aprendizaje y de resultados tangibles que me impulsan a profundizar en el estudio de las tecnologías que se han tratado durante todo el desarrollo del proyecto.

Por otro lado, hay que reseñar la importancia del control de los riesgos en el proyecto y valorarlos en su justa medida. Durante todo el proceso de desarrollo nos hemos encontrado con decisiones complicadas que tomar y algunas veces ha sido a causa de la no contemplación de los riesgos y contingencias, por ejemplo creando desviaciones, dedicando recursos al conocimiento de nuevas características que no estaban identificadas en el proyecto original, o bien otras veces, por una infravaloración del coste de las tareas debido al desconocimiento de las herramientas. Este es un recordatorio muy importante para las tareas futuras.

Otra Conclusión importante es la metodología del uso de GIT en un proyecto con CI/CD. Mi perspectiva inicial sobre la necesidad de creación de ramas ha evolucionado y veo que su uso debe limitarse a proyectos en los que se deba supervisar tareas de *Juniors* o en los que se desee probar funcionalidades alternativas en sistemas no productivos, puesto que el propio proceso de CI/CD establece procedimientos de calidad e impide el despliegue si la implementación del pipeline y los test no son óptimos y que en cualquier caso se pueden establecer planes de contingencia para la puesta en producción de nuevo de versiones estables basadas en imágenes anteriores de forma muy rápida.

Por otro lado, y gracias a Terraform he empezado a tener conocimiento directo de *IAC,* algo que no se contemplaba en el proyecto original, pero que sin duda ha dado gran valor al mismo.

Con la consecución de estos elementos clave, se ha logrado cumplir los objetivos primarios del proyecto. Estos elementos incluyen:

1. Creación de un producto operativo: El objetivo principal del proyecto ha sido desarrollar un producto funcional y operativo. Esto ha implicado implementar la lógica de negocio y los requisitos especificados, utilizando la orquestación mediante un sistema de mensajería asíncrona como RabbitMQ de componentes desarrollados en diferentes lenguajes en el clúster de Kubernetes y utilizando además la comunicación a través de API REST para la interacción con la aplicación. Por último, el proyecto ha incluido funcionalidades como el envío de correos electrónicos a los usuarios, siguiendo las especificaciones definidas durante la concepción del producto.

2. Conocimiento de Kubernetes: El proyecto ha permitido adquirir conocimiento sobre los procedimientos necesarios para implementar y administrar un entorno de clúster de Kubernetes, incluyendo la creación y gestión de Pods, Servicios y otros recursos mediante el uso de las herramientas y comandos utilizados en el ecosistema de Kubernetes, tales como kubectl para interactuar con el clúster y otras herramientas relacionadas como kind o el panel de administración, tanto local como en Google Cloud.

3. CI/CD: El proyecto ha permitido explorar la integración continua y la entrega continua (CI/CD) en un entorno de Kubernetes, incluyendo la automatización de la construcción, pruebas, registro de contenedores y despliegue de la aplicación. Esto ha derivado en una metodología mixta en la que se combina Agile con prácticas de DevOps.

4. Gestión de proyecto: Durante el desarrollo del proyecto, se ha obtenido experiencia en la gestión y organización del proyecto, incluyendo la planificación, seguimiento y coordinación de las tareas relacionadas con la implementación mediante el uso básico de una metodología SCRUM y el uso de Jira y GitLab.

5. Plataformas de la nube: El proyecto ha implicado el despliegue del clúster de Kubernetes en una plataforma de la nube, en este caso en Google Cloud Platform (GCP). Esto ha proporcionado la oportunidad de trabajar con estas plataformas y aprovechar sus servicios y características específicas.

6. Arquitectura de software y diseño de patrones: Durante el desarrollo del proyecto, se ha abordado la arquitectura de software y se ha aplicado el diseño de patrones para

garantizar una estructura sólida y modular. Se han considerado principios como la separación de responsabilidades, la reutilización de componentes y la escalabilidad. Además, se han utilizado patrones de diseño reconocidos, como la arquitectura hexagonal, o la Inyección de Dependencias, para promover la flexibilidad y la mantenibilidad del código. La elección y aplicación adecuada de la arquitectura y los patrones de diseño han contribuido a la creación de un sistema robusto y escalable en el entorno del clúster de Kubernetes.

9. Glosario

Agile: Desarrollo iterativo e incremental.

AWS: Amazon Web Services. Servicios en la nube de Amazon.

Azure: Microsoft Azure, servicios en la nube de Azure.

BackLog: Lista priorizada de características de un producto.

CI/CD: Integracion y entrega continua.

Docker/Dockerización: abstracción virtual y empaquetado de aplicaciones y servicios en contenedores independizándolos del hardware subyacente y proporcionando un entorno de ejecución consistente.

GCP: Google Cloud Platform. Servicios en la nube de Google.

Kubernetes: Plataforma de orquestación de contenedores

Microservicios: Arquitectura de software que se basa en la descomposición en pequeños servicios independientes y autónomos.
Scrum: Tipo de metodología Agile

Serverless: Modelo de ejecución en la nube, sin uso de recursos de computación local.

Sprint: Unidades de tiempo fija en las que se realizan las iteraciones de un proyecto que utiliza metodología ágil.
SRP: Principio de responsabilidad única.
Walking skeleton: versión inicial, simplificada y funcional de un sistema.

Domain-Driven Design (DDD): Es una metodología que se enfoca en el análisis y diseño del dominio del problema y busca desarrollar un lenguaje común entre los equipos de desarrollo y el negocio. DDD puede ser útil para la creación de microservicios, ya que ayuda a identificar los límites de los servicios y las interacciones entre ellos.

DevOps: DevOps es una metodología que busca la colaboración entre los equipos de desarrollo y operaciones para automatizar y mejorar el proceso de entrega de software. La metodología DevOps puede ser útil para la creación de microservicios, ya que ayuda a automatizar la implementación y pruebas de los servicios.

10. Bibliografía

Guía para el uso del lenguaje inclusivo en el INSST, O.A., M.P. (n.d.). Retrieved March 11, 2023, from http://cpage.mpr.gob.es

Los 100 riesgos en la gestión de proyectos | LinkedIn. (n.d.). Retrieved March 11, 2023, from https://www.linkedin.com/pulse/los-100-riesgos-en-la-gesti%C3%B3n-de-proyectos-luciano-moreira-da-cruz/?originalSubdomain=es

Scrum: qué es, cómo funciona y por qué es excelente. (n.d.). Retrieved March 11, 2023, from https://www.atlassian.com/es/agile/scrum

The New User Story Backlog is a Map – Help your organization focus on successful outcomes. (n.d.). Retrieved March 11, 2023, from https://www.jpattonassociates.com/the-new-backlog/

Desarrollo ágil de software - Wikipedia, la enciclopedia libre. (n.d.). Retrieved March 12, 2023, from https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software

Product backlog - Wikipedia. (n.d.). Retrieved March 12, 2023, from https://en.wikipedia.org/wiki/Product_backlog

Kubernetes. (n.d.). Retrieved March 12, 2023, from https://kubernetes.io/es/

Welcome to Jira Software | Atlassian. (n.d.). Retrieved March 12, 2023, from https://www.atlassian.com/software/jira/guides/getting-started/introduction#what-is-jira-software

What are microservices? (n.d.). Retrieved March 12, 2023, from https://microservices.io/

Introducing Assemblage - a microservice architecture definition process. (n.d.). Retrieved March 12, 2023, from https://microservices.io/post/architecture/2023/02/09/assemblage-architecture-definition-process.html

Projects · Dashboard · GitLab. (n.d.). Retrieved March 12, 2023, from https://gitlab.com/

Kubernetes en AWS | Amazon Web Services. (n.d.). Retrieved March 12, 2023, from https://aws.amazon.com/es/kubernetes/

Kubernetes - Google Kubernetes Engine (GKE) | Kubernetes Engine | Google Cloud. (n.d.). Retrieved March 12, 2023, from https://cloud.google.com/kubernetes-engine?hl=es

Introduction to Azure Kubernetes Service - Azure Kubernetes Service | Microsoft Learn. (n.d.). Retrieved March 12, 2023, from https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes

News API – Search News and Blog Articles on the Web. (n.d.). Retrieved March 12, 2023, from https://newsapi.org/

The State of Enterprise Open Source: A Red Hat report. (n.d.). Retrieved March 30, 2023, from https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022

SOLID - Wikipedia, la enciclopedia libre. (n.d.). Retrieved April 5, 2023, from https://es.wikipedia.org/wiki/SOLID

SOLID Microservices Design. Applying SOLID principles to… | by Adrien Nortain | Zenika. (n.d.). Retrieved April 5, 2023, from https://medium.zenika.com/solid-microservices-design-dc6a4044a050

Principles for Microservice Design: Think IDEALS, Rather than SOLID. (n.d.). Retrieved April 5, 2023, from https://www.infoq.com/articles/microservices-design-ideals/

Domain-Driven Design: Tackling Complexity in the Heart of Software" de Eric Evans.

Agile Software Development, Principles, Patterns, and Practices" de Robert C. Martin y Micah Martin.

The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations" de Gene Kim, Jez Humble, Patrick Debois y John Willis.

Clean Code: A Handbook of Agile Software Craftsmanship" por Robert C. Martin

Raghavan, B., & Ma, J. (2011). *The Energy and Emergy of the Internet*. Retrieved April 2, 2023, from https://raghavan.usc.edu/papers/emergy-hotnets11.pdf

L4D Learning for Democracy: The Cloud begins with Coal - Big Data, Big Networks, Big Infrastructure, and Big Power - An Overview of the Electricity used by the Global Digital Ecosystem - Mark P. Mills. (n.d.). Retrieved April 11, 2023, from http://www.ict-21.ch/l4d/pg/file/read/878855/the-cloud-begins-with-coal-big-data-big-networks-big-infrastructure-and-big-power-an-overview-of-the-electricity-used-by-the-global-digital-ecosystem-mark-p-mills

How Much Energy Do Data Centers Really Use? - Energy Innovation: Policy and Technology. (n.d.). Retrieved April 11, 2023, from https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/

Amazon, Google, Microsoft: Here's Who Has the Greenest Cloud | WIRED. (n.d.). Retrieved April 11, 2023, from https://www.wired.com/story/amazon-google-microsoft-green-clouds-and-hyperscale-data-centers/

Cloud Computing, Server Utilization, & the Environment | AWS News Blog. (n.d.). Retrieved April 11, 2023, from https://aws.amazon.com/es/blogs/aws/cloud-computing-server-utilization-the-environment/

Usa, G. (n.d.). *Studie: Clicking Clean | Greenpeace*.
Clicking Clean - Greenpeace International. (n.d.). Retrieved April 11, 2023, from https://www.greenpeace.org/international/publication/6826/clicking-clean-2017/

How dirty is your data? (n.d.).
Retrieved April 11, 2023, from
https://www.greenpeace.org/static/planet4-international-stateless/2011/04/4cceba18-dirty-data-report-greenpeace.pdf

Documentation | Redis. (n.d.). Retrieved April 12, 2023, from https://redis.io/docs/

RabbitMQ tutorial - "Hello World!" — RabbitMQ. (n.d.). Retrieved April 12, 2023, from https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html

rabbitmq/cluster-operator: RabbitMQ Cluster Kubernetes Operator. (n.d.). Retrieved April 20, 2023, from https://github.com/rabbitmq/cluster-operator

RabbitMQ Cluster Kubernetes Operator Quickstart — RabbitMQ. (n.d.). Retrieved May 9, 2023, from https://www.rabbitmq.com/kubernetes/operator/quickstart-operator.html

RabbitMQ on Kubernetes for beginners - YouTube. (n.d.). Retrieved May 15, 2023, from https://www.youtube.com/watch?v=_lpDfMkxccc

Deploying RabbitMQ to Kubernetes: What's Involved? | RabbitMQ - Blog. (n.d.). Retrieved May 9, 2023, from
https://blog.rabbitmq.com/posts/2020/08/deploying-rabbitmq-to-kubernetes-whats-involved/
Quickstart · moq/moq4 Wiki. (n.d.). Retrieved April 1, 2023, from
https://github.com/Moq/moq4/wiki/Quickstart

Introducción a las pruebas unitarias - Visual Studio (Windows) | Microsoft Learn. (n.d.). Retrieved April 1, 2023, from
https://learn.microsoft.com/es-es/visualstudio/test/getting-started-with-unit-testing?view=vs-2022&tabs=dotnet%2Cmstest

Tutorial: Create a web API with ASP.NET Core | Microsoft Learn. (n.d.). Retrieved April 1, 2023, from
https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio

Unit testing C# with NUnit and .NET Core - .NET | Microsoft Learn. (n.d.). Retrieved April 1, 2023, from https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit

Containerize an app with Docker tutorial - .NET | Microsoft Learn. (n.d.). Retrieved April 1, 2023, from https://learn.microsoft.com/en-us/dotnet/core/docker/build-container?tabs=windows

Deployments | Kubernetes. (n.d.). Retrieved April 5, 2023, from
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Get started with Swashbuckle and ASP.NET Core | Microsoft Learn. (n.d.). Retrieved April 12, 2023, from https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-7.0&tabs=visual-studio

Deploy and Access the Kubernetes Dashboard | Kubernetes. (n.d.). Retrieved May 25, 2023, from https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Spring Initializr. (n.d.). Retrieved March 30, 2023, from https://start.spring.io/

Netflix OSS, Spring Cloud, or Kubernetes? How About All of Them! – Software Blog. (n.d.). Retrieved March 30, 2023, from https://blog.christianposta.com/microservices/netflix-oss-or-kubernetes-how-about-both/

Guide to Spring Email | Baeldung. (n.d.). Retrieved May 25, 2023, from https://www.baeldung.com/spring-email

Tutorial: Create and run your first GitLab CI/CD pipeline | GitLab. (n.d.). Retrieved April 2, 2023, from https://docs.gitlab.com/ee/ci/quick_start/

`.gitlab-ci.yml` keyword reference | GitLab. (n.d.). Retrieved April 2, 2023, from https://docs.gitlab.com/ee/ci/yaml/index.html

Reliability Guide — RabbitMQ. (n.d.). Retrieved April 17, 2023, from https://www.rabbitmq.com/reliability.html

RabbitMQ in Microservices - Step by Step Tutorial with SpringBoot - Microservices . (n.d.). Retrieved April 17, 2023, from https://www.kindsonthegenius.com/microservices/rabbitmq-in-microservices-step-by-step-tutorial-with-springboot/

11. Repositorios de Fuentes

El código fuente y la documentación UML de los microservicios se puede consultar en los siguientes repositorios.

git@gitlab.com:tfg1010711/documentation.git → imagenes y uml del proyecto.
git@gitlab.com:tfg1010711/topicsubscriber.git → c# con fuentes, punto de entrada al sistema.
git@gitlab.com:tfg1010711/msgquery.git → java con fuentespara consultas a la API externa.
git@gitlab.com:tfg1010711/topichistory.git → python con fuentes.
git@gitlab.com:tfg1010711/msgdispatch.git → java envio de correo con fuentes.
git@gitlab.com:tfg1010711/kubernetes.git → archivos para la creacion y el despliegue de los secrets del clúster.
git@gitlab.com:tfg1010711/rabbitmq.git → archivos de despliegue.

Anexo I Detalle de la planificación

Sprint 0

Fecha - 1 de marzo de 2023 - 14 de marzo de 2023
Objetivo del sprint - El Sprint 0 define el alcance y el ámbito del proyecto. En este *Sprint* se identifica el problema, la metodología de trabajo que vamos a utilizar para resolver el mismo, así como una idea aproximada de la relación de tareas. También se identifican los principales riesgos del proyecto y se definen medidas de contingencia que limiten el impacto y las desviaciones en el mismo. La duración de este *Sprint* es de dos semanas.

ID	Actividad	t/h	Duración
TDR-37	Documentación y preparación para entrega de TFG	Tarea	12
TDR-2	Configuración del proyecto en Jira	Tarea	3
TDR-5	Configuración del proyecto en GitLab	Tarea	2
TDR-3	Definición del alcance del proyecto y creación de historias de usuario	Historia	3
TDR-8	Identificación de herramientas necesarias para el proyecto	Historia	3
TDR-18	Identificación de riesgos y propuesta de acciones de mitigación	Historia	3
TDR-10	Creación del backlog del producto	Historia	3
TDR-45	Creación de planificación de *Sprints*	Tarea	10

Sprint 1 y Sprint 2

Fecha - 15 de marzo de 2023 - 22 de marzo de 2023
Fecha - 22 de marzo de 2023 - 29 de marzo de 2023

Objetivo de los *Sprints* - Diseño arquitectónico inicial y creación de walking Skeleton de servicios

ID	Actividad	t/h	Duración
TDR-25	Desarrollo del walking skeleton de un servicio en el primer lenguaje conocido	Tarea	2
TDR-48	Documentación trabajo de TFG PEC 2	Tarea	3
TDR-69	Diseño arquitectura del sistema	Tarea	6
TDR-66	Test servicio 1 lenguaje conocido	Tarea	1

TDR-28	Desarrollo del walking skeleton de un servicio en el segundo lenguaje	Tarea	3

ID	Actividad	t/h	Duración
TDR-49	Documentación trabajo de TFG PEC 2	Tarea	6
TDR-32	Desarrollo del walking skeleton de un servicio en el tercer lenguaje	Tarea	3
TDR-67	Test servicio 2 lenguaje bajo conocimiento	Tarea	3
TDR-68	Test servicio 3 lenguaje bajo conocimiento	Tarea	3
TDR-33	Pruebas y creación de la imagen Docker para el tercer servicio	Tarea	2

Sprint 3

Fecha - 29 de marzo de 2023 - 5 de abril de 2023

Objetivo del sprint - Estudiar las plataformas de servicios en la nube

ID	Actividad	t/h	Duración
TDR-65	Búsqueda y estudio documentación despliegue CI/CD plataforma 1	Tarea	3
TDR-70	Búsqueda y estudio documentación despliegue CI/CD plataforma 2	Tarea	3
TDR-71	Búsqueda y estudio documentación despliegue CI/CD plataforma 3	Tarea	3
TDR-76	Creacion de Basic Build en GITLAB	Tarea	3
TDR-50	Documentación trabajo de TFG PEC 2	Tarea	3

Sprint 4

Fecha - 4 de abril de 2023 - 12 de abril de 2023

Objetivo del sprint - Dockerización, creación de clúster local y despliegue inicial

ID	Actividad	t/h	Duración
TDR-26	Pruebas y creación de la imagen Docker para el primer servicio entorno conocido	Tarea	1
TDR-29	Pruebas y creación de la imagen Docker para el segundo servicio	Tarea	2

TDR-33	Pruebas y creación de la imagen Docker para el tercer servicio	Tarea	2
TDR-27	Configuración del clúster de Kubernetes Local	Tarea	2
TDR-72	Despliegue del primer servicio en clúster local Kubernetes	Tarea	2
TDR-30	Despliegue del segundo servicio en clúster local Kubernetes	Tarea	2
TDR-73	Despliegue del tercer servicio en clúster local Kubernetes	Tarea	2
TDR-47	Documentación trabajo de TFG PEC 2	Tarea	3

Sprint 5

Fecha - 12 de abril de 2023 - 19 de abril de 2023

Objetivo del sprint - Implementación de lógica de servicios y creación de tests

ID	Actividad	t/h	Duración
TDR-42	Implementar lógica de servicio Darse de alta en el sistema indicando correo para poder identificar al usuario y topic en el que está interesado.	Tarea	4
TDR-43	Implementar lógica de servicio acceder a las noticias en el periodo de tiempo que se indique y guardarlas	Tarea	4
TDR-74	Creación de test y pruebas funcionales alta usuario	Tarea	2
TDR-75	Creación de test y pruebas funcionales acceso a noticias	Tarea	2
TDR-51	Documentación trabajo de TFG PEC 3	Tarea	3

Sprint 6

Fecha - 12 de abril de 2023 - 19 de abril de 2023

Objetivo del sprint - Implementación de lógica de servicios y creación de tests

ID	Actividad	t/h	Duración
TDR-44	Implementar lógica de servicio enviar correo de intereses a usuarios	Tarea	6
TDR-56	Pruebas funcionalidad envío correo	Tarea	1

TDR-31	Configuración de un sistema de mensajería asíncrona	Tarea	5
TDR-52	Documentación trabajo de TFG PEC 3	Tarea	3

Sprint 7

Fecha - 26 de abril de 2023 - 3 de mayo de 2023

Objetivo del sprint - Se implementan los procesos automatizados de despliegue desde el repositorio gitlab a las diferentes plataformas

ID	Actividad	t/h	Duración
TDR-36	Implementación de un proceso de CI/CD para automatizar el despliegue y la actualización de los servicios en plataforma 1	Tarea	5
TDR-77	Implementación de un proceso de CI/CD para automatizar el despliegue y la actualización de los servicios en plataforma 2	Tarea	5
TDR-78	Implementación de un proceso de CI/CD para automatizar el despliegue y la actualización de los servicios en plataforma 3	Tarea	5
TDR-53	Documentación trabajo de TFG PEC 3	Tarea	3

Sprint 8

Fecha - 3 de mayo de 2023 - 10 de mayo de 2023

Objetivo del sprint - Este Sprint se dedica a la realización de pruebas funcionales y a la optimización del clúster de kubernetes

ID	Actividad	t/h	Duración
TDR-57	Pruebas funcionales e integración	Tarea	6
TDR-35	Pruebas de integración y optimización del clúster de kubernetes	Tarea	8
TDR-54	Documentación trabajo de TFG PEC 3	Tarea	5

Sprint 9

Fecha - 10 de mayo de 2023 - 17 de mayo de 2023

Objetivo del sprint - Recopilar información sobre impacto ambiental de los despliegues en las distintas plataformas

ID	Actividad	t/h	Duración
TDR-60	Buscar información y documentar impacto ambiental comparativo de los diferentes proveedores	Tarea	6
TDR-61	Documentación trabajo de TFG	Tarea	6

Sprint 10

Fecha - 17 de mayo de 2023 - 14 de junio de 2023

Objetivo del sprint - Finalización de documentación TFG y preparación entrega

ID	Actividad	t/h	Duración
TDR-58	Preparación video entrega	Tarea	24
TDR-59	Preparación y revisión documento TFG	Tarea	48

Backlog

Aquí se muestra el Backlog, donde quedan las historias de usuario sobre las que se trabajará durante el proyecto y a las que asignan tareas específicas una vez se defina la arquitectura

ID	Actividad
TDR-20	Como usuario quiero darme de alta en un servicio de noticias que me envíe diariamente un correo con información de un topic de noticias de internet en el que esté interesado
TDR-46	Como usuario quiero realizar una consulta de la lista de topics a las que estoy suscrito
TDR-62	Como desarrollador quiero preparar despliegues automatizados desde gitlab hacia diferentes plataformas de servicios en la nube
TDR-63	Como usuario quiero recibir correos personalizados con las noticias de las temáticas en las que estoy interesado
TDR-64	Como usuario quiero consultar el histórico de noticias que he recibido

Anexo II Creación de un servicio en C# topicsubscriber

Por qué C#

C# es uno de los principales lenguajes de programación utilizados para crear aplicaciones en la plataforma .NET de Microsoft. Esta plataforma proporciona un marco sólido y extensible para crear aplicaciones web y servicios de API REST, es ampliamente conocida y está bien documentada. Además, C# se integra de forma sencilla con otras tecnologías de Microsoft, como bases de datos SQL Server y Azure, lo que facilita la creación de una infraestructura completa de aplicaciones.

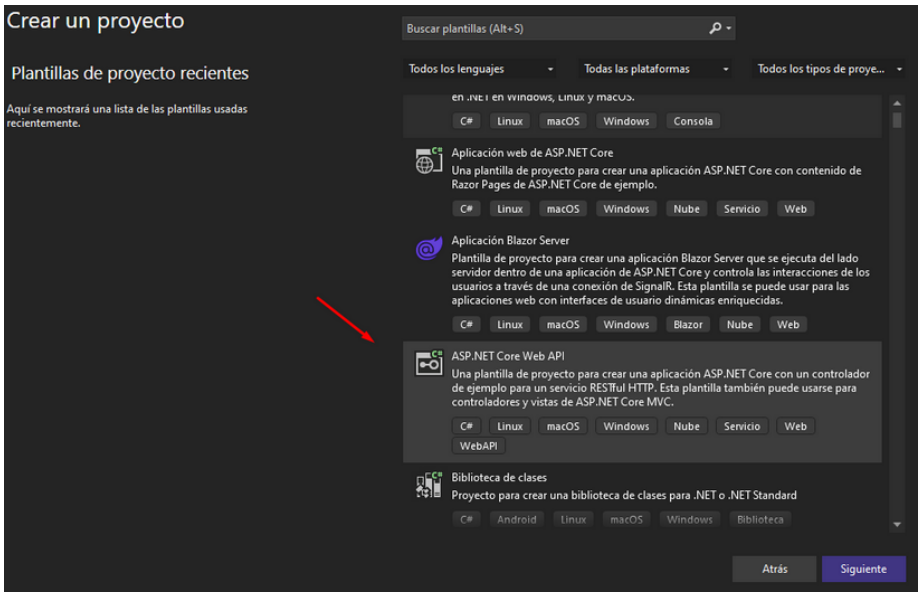
Cabe también destacar que el framework .NET ofrece una gran cantidad de herramientas y bibliotecas que permiten acelerar el desarrollo de aplicaciones.

Qué hace este servicio

Este servicio se encarga de mostrar una API de acceso a los clientes que permite subscribirse mediante la introducción de su correo electrónico a una temática de noticias

Proceso de creación de un servicio en C#

Para crear el servicio web se utiliza la aplicación vstudio 2022 community. Para ello, en primer lugar se crea el proyecto utilizando la plantilla ASP.NET Core Web API.



En la pantalla siguiente se selecciona el Framework .NET 7.0 y a continuación se comprueba que estén activas las siguientes propiedades:

- Configuración para HTTPS
- Habilitar Docker, sistema operativo de Docker: windows
- Usar controladores
- Habilitar compatibilidad con OpenAPI

Ahora, una vez creado el proyecto se debe instalar los paquetes necesarios desde el administrador de paquetes NuGet. Los paquetes instalados son los siguientes:

- RabbitMQClient, para la mensajería asíncrona.
- Microsoft.EntityFrameworkCore, como OMR para que funcione el Entity Framework en un proyecto.
- Microsoft.EntityFrameworkCore.InMemory, para tener una base de datos en memoria para realizar tests y configurar inicialmente el servicio.
- Newtonsoft.Json, para funcionalidades con serialización y deserialización de json

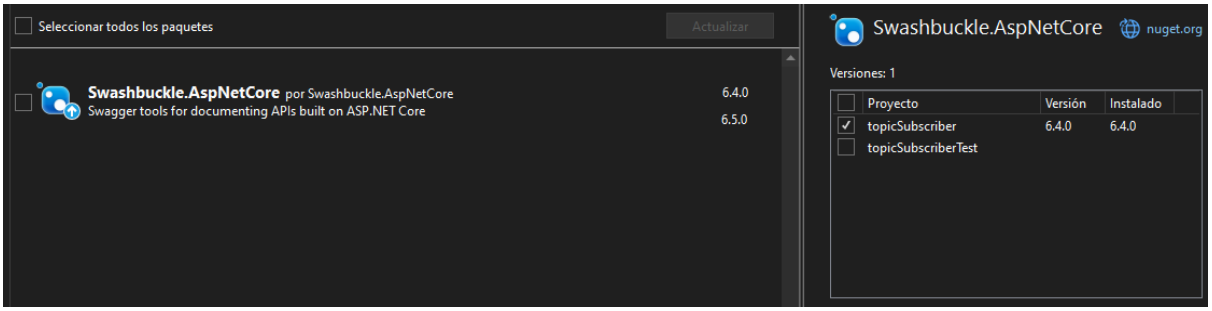
Probar el servicio C#

Para la creación de tests se usa la plantilla de Proyecto de prueba NUnit. Una vez creado el proyecto de test se agrega la referencia de proyecto ApiUser y desde el administrador de paquetes se agrega una referencia a Moq.

Por último, y para simular la interacción con los datos, en este entorno se utiliza una base de datos en memoria.

Documentar el proyecto con swagger

Para preparar la documentación con swagger el paquete Swashbuckle.AspNetCore debe estar instalado en el sistema.



Idealmente, se debe seleccionar esta opción en la creación del proyecto, de esta forma el archivo Program.cs tendrá toda la información necesaria para que swagger esté operativo y solo se necesita agregar el siguiente código al método `AddSwaggerGen`.

```
builder.Services.AddSwaggerGen(options =>
{
```

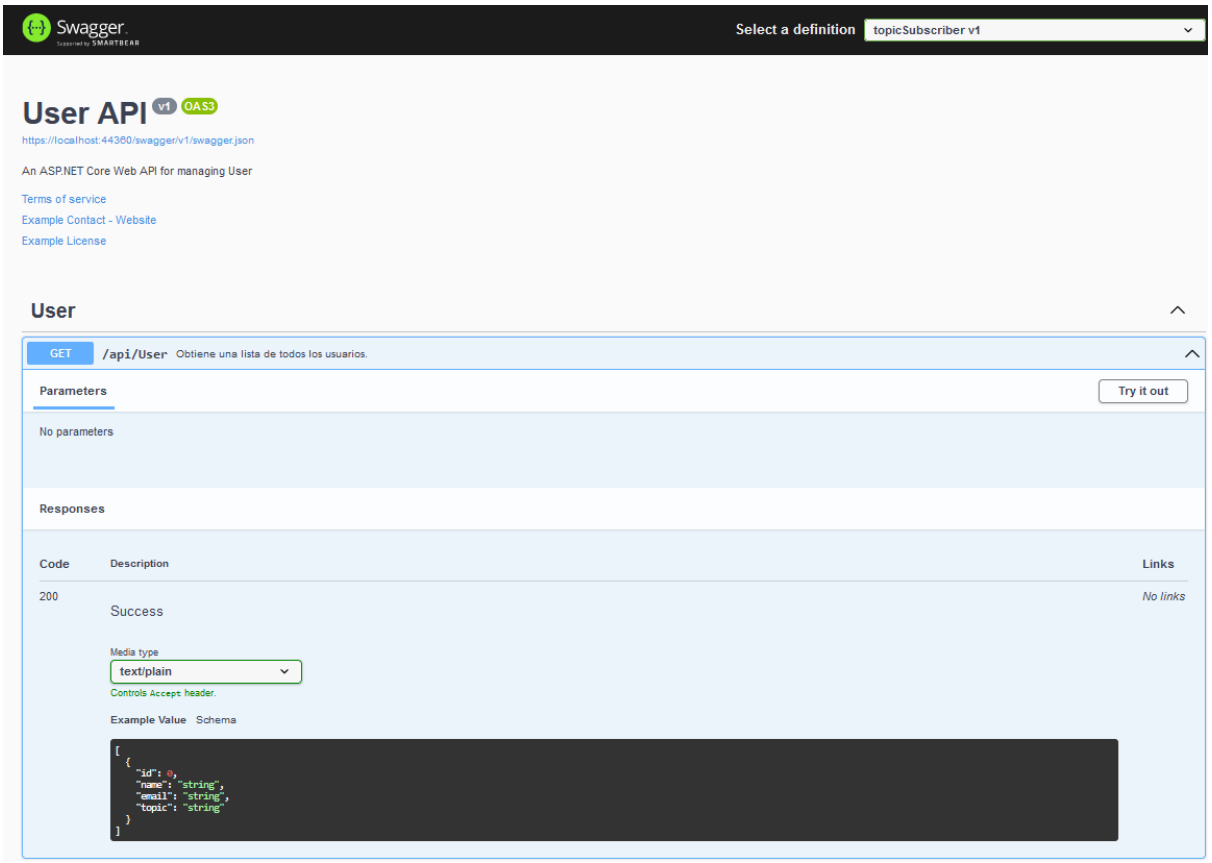
```
       options.SwaggerDoc("v1", new OpenApiInfo
       {
       Version = "v1",
       Title = "User API",
       Description = "An ASP.NET Core Web API for managing User",
       TermsOfService = new Uri("https://example.com/terms"),
       Contact = new OpenApiContact
       {
              Name = "Example Contact",
              Url = new Uri("https://example.com/contact")
       },
       License = new OpenApiLicense
       {
              Name = "Example License",
              Url = new Uri("https://example.com/license")
       }
       });
       // using System.Reflection;
       var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
       options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));
});
```

A continuación, agregamos el siguiente código en nuestro archivo de proyecto.

```
<PropertyGroup>
        <GenerateDocumentationFile>true</GenerateDocumentationFile>
    </PropertyGroup>
```

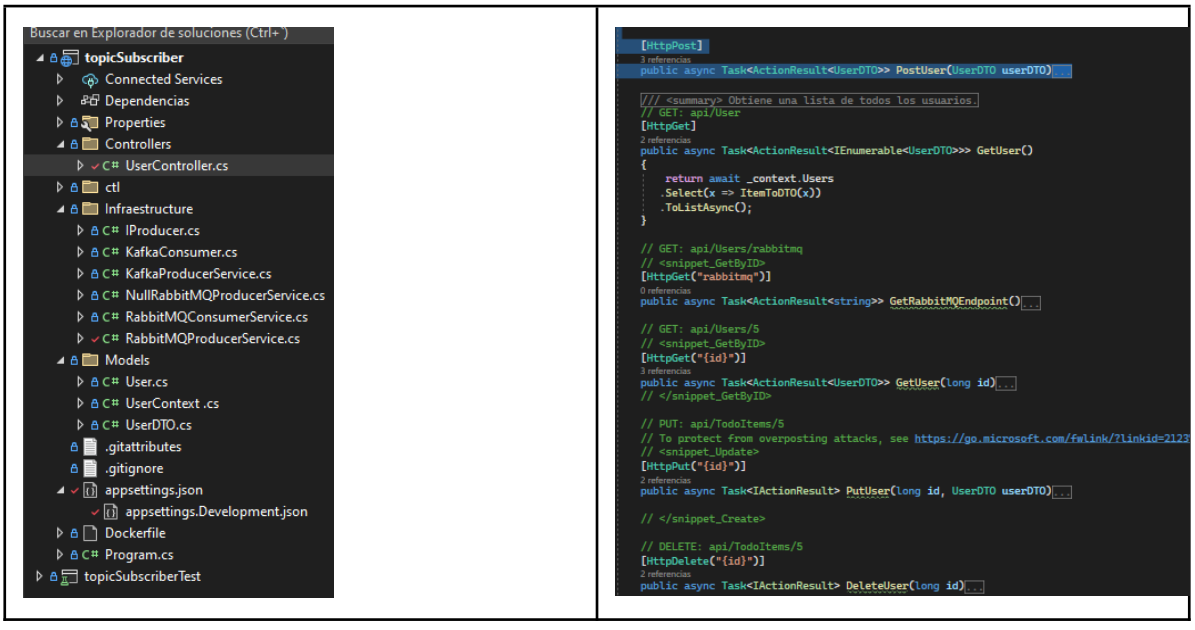
En este punto, ya se puede agregar los comentarios que se desee en los controladores con el siguiente formato.

```
/// <summary>
       /// Obtiene una lista de todos los usuarios.
       /// </summary>
       /// <returns>Una lista de objetos UserDTO que representan a los
usuarios.</returns>
       // GET: api/User
       [HttpGet]
       public async Task<ActionResult<IEnumerable<UserDTO>>> GetUser()
       {
       return await _context.Users
       .Select(x => ItemToDTO(x))
       .ToListAsync();
       }
```



Estructura del proyecto topicsubscriber

En la estructura de proyecto se muestra la separación entre modelo e infraestructura, el uso de controladores y un proyecto de test. Además, se muestra el uso de interfaces en toda la aplicación.



Proceso de CI/CD

El proceso de despliegue automatizado es fundamental para garantizar una entrega continua y eficiente de software en entornos de producción. Utilizando GitLab CI/CD, podemos configurar un pipeline que automatice las tareas de construcción, pruebas, registro de imágenes Docker y despliegue en un entorno de producción.

El pipeline se compone de varias etapas clave:

1. Etapa de construcción (build): En esta etapa, se utiliza una imagen de Microsoft .NET SDK para realizar las tareas de compilación del proyecto. Se restauran las dependencias, se compila en modo Release y se publica el proyecto en una carpeta específica. Los artefactos generados en esta etapa se conservan para su uso posterior.

2. Etapa de pruebas (test): En esta etapa, se ejecutan las pruebas del proyecto utilizando la misma imagen de Microsoft .NET SDK. Se realiza la restauración de dependencias y se ejecutan las pruebas en modo Release. Esto garantiza que el software se someta a un riguroso proceso de pruebas antes de avanzar hacia el siguiente paso.

3. Etapa de registro de imágenes Docker (docker-register): Esta etapa se encarga de registrar las imágenes Docker generadas en un registro de contenedores. Utilizando la imagen "docker:latest" y el servicio "docker:dind", se llevan a cabo las tareas relacionadas con Docker. Se crea una etiqueta de imagen única basada en el commit actual y el ID del trabajo. A continuación, se realiza el inicio de sesión en el registro de contenedores y se construye la imagen Docker del proyecto. La imagen se etiqueta y se envía tanto con la etiqueta generada como con la etiqueta "latest". Este proceso permite mantener un registro histórico de las imágenes generadas y facilita su posterior despliegue.

4. Etapa de despliegue (deploy): En esta etapa final, se implementa el proyecto en un entorno de producción, en este caso en Google, mediante el uso de la imagen google/cloud-sdk, y la aplicación con kubectl de los archivos de deployment y service.

```
stages:
  - build
  - test
  - docker-register
  - deploy

build:
  stage: build
  image: mcr.microsoft.com/dotnet/sdk:7.0
  script:
       - dotnet restore
       - dotnet build -c Release
       - dotnet publish -c Release -o out
  artifacts:
       paths:
       - out/

test:
  stage: test
  image: mcr.microsoft.com/dotnet/sdk:7.0
  script:
       - dotnet restore
```

```
      - dotnet test -c Release

docker-register-job:
  image: docker:latest
  stage: docker-register
  services:
      - docker:dind

  before_script:
      # Create a tag using the commit and the job ID
      - export TAG=${CI_COMMIT_SHORT_SHA}-${CI_JOB_ID}

      # Login into the containers registry
      - docker login $CI_REGISTRY -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"

  script:
      # Build and push the real image
      - docker build -t $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG .
      - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG

      # Retag and push the image to have a latest tag
      - docker tag $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG
$CI_REGISTRY/${CI_PROJECT_PATH}:latest
      - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:latest
  only:
      - main

deploy-gke:
  stage: deploy
  image:
    name: google/cloud-sdk
  script:
    - echo "This job deploys to Google Cloud from the $CI_COMMIT_BRANCH branch."
    - echo "$BASE64_GOOGLE_CREDENTIALS" | base64 -d > google-credentials.json
    - gcloud auth activate-service-account --key-file=google-credentials.json
    - gcloud config set project $TF_VAR_gcp_project
    - gcloud container clusters get-credentials $TF_VAR_cluster_name --region
$TF_VAR_gcp_region
    - >
      curl --header "Authorization: Bearer $CI_JOB_TOKEN" -o topicsubscriber-deployment.yaml
"https://gitlab.com/tfg1010711/topicsubscriber/-/raw/main/ApiUser/ctl/topicsubscriber-deployme
nt.yaml"
    - >
      curl --header "Authorization: Bearer $CI_JOB_TOKEN" -o topicsubscriber-service.yaml
"https://gitlab.com/tfg1010711/topicsubscriber/-/raw/main/ApiUser/ctl/topicsubscriber-service.
yaml"
    - kubectl apply -f topicsubscriber-deployment.yaml
    - kubectl apply -f topicsubscriber-service.yaml
  dependencies:
    - build
  only:
    - main
```

Configuración de RabbitMQ para C#

La clase que implementa el broker de RabbitMQ se encarga de establecer la conexión con el servidor de RabbitMQ y configurar los intercambios (exchanges) y colas necesarias para la comunicación. Es responsable de crear la conexión a través de la clase ConnectionFactory y configurar los parámetros necesarios, como el host, nombre de usuario, contraseña y puerto.

En el fragmento de código proporcionado, se establece la conexión y se crea un canal de comunicación con RabbitMQ. Luego, se declara un intercambio de tipo "Direct" y una cola en ese canal. Además, se realiza la vinculación entre la cola y el intercambio.

Esta implementación concreta del broker de RabbitMQ se utiliza para la comunicación en la aplicación. En Program.cs, se inyecta mediante una interfaz, lo que significa que se proporciona una abstracción que define la funcionalidad del broker. Esto permite una mayor flexibilidad y modularidad en la aplicación, ya que se puede intercambiar fácilmente la implementación del broker por otra que cumpla con la misma interfaz sin afectar el resto del código.

```
string rabbitMQHostName = configuration["RabbitMQ:HostName"];
             string rabbitMQUserName = configuration["RabbitMQ:UserName"];
             string rabbitMQPassword = configuration["RabbitMQ:Password"];
             string rabbitMQPort = configuration["RabbitMQ:Port"];
             try

             {
             var factory = new ConnectionFactory()
             {
                    HostName = rabbitMQHostName,
                    Port = Int32.Parse(rabbitMQPort),
                    UserName = rabbitMQUserName,
                    Password = rabbitMQPassword
             };

             _connection = factory.CreateConnection();
             _channel = _connection.CreateModel();

             _channel.ExchangeDeclare(EXCHANGE_NAME, ExchangeType.Direct);
             _channel.QueueDeclare(QUEUE_NAME, true, false, false, null);
             _channel.QueueBind(QUEUE_NAME, EXCHANGE_NAME, "");
```

Creación de Docker y Kubernetes

Preparación del Dockerfile

En primer lugar se crea en la raíz de nuestro proyecto el fichero Dockerfile al que agregamos el código siguiente.

```
# syntax=docker/dockerfile:1

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build-env
WORKDIR /ApiUser

# Copy everything
COPY . ./
# Restore as distinct layers
RUN dotnet restore
# Build and publish a release
RUN dotnet publish -c Release -o out

# Build runtime image
FROM mcr.microsoft.com/dotnet/aspnet:7.0
WORKDIR /ApiUser
COPY --from=build-env /ApiUser/out .
ENTRYPOINT ["dotnet", "topicSubscriber.dll"]
```

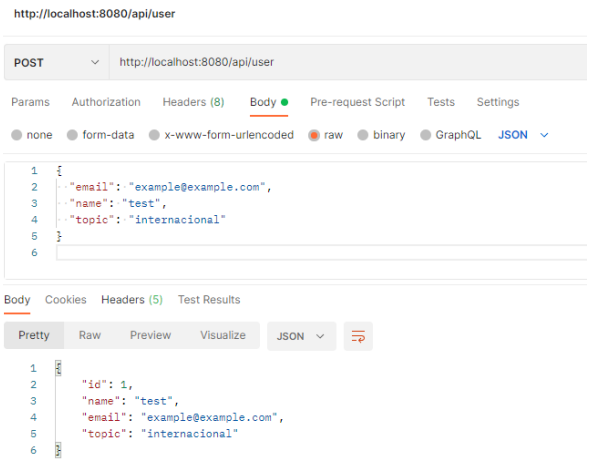
A continuación se prepara el docker.

```
$ docker build -t topicsubscriber:v1 .
```

Por último, se procede a arrancar el servicio docker en segundo plano

```
$ docker run --name topic-subscriber -d -p 8080:80 topicsubscriber:v1
```

Como resultado, tenemos la aplicación disponible localmente en el puerto 8080



Preparación del Pod de kubernetes

Mediante el siguiente archivo, se especifica que se desea un único pod ("replica: 1") y que la imagen del contenedor que se utilizará es la imagen registrada en gitlab registry.gitlab.com/tfg1010711/topicsubscriber:latest. El contenedor se expone internamente a través del puerto 80. El selector "matchLabels" se utiliza para emparejar el pod con el servicio correspondiente. Con anterioridad se debe haber creado el secret con los datos de acceso a rabbitmq. También se crea otro secret para acceder al registro de contenedores de gitlab y que la imagen pueda ser descargada.

```
kubectl create secret docker-registry gitlab-registry
--docker-server=registry.gitlab.com  --docker-username=@drios5
--docker-password=glpat-rUXCBdW56ymH-nhUsmw2   --docker-email=drios5@uoc.edu
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: topicsubscriber
  labels:
       app: topicsubscriber
spec:
  selector:
       matchLabels:
       app: topicsubscriber
```

```
  replicas: 1
  template:
       metadata:
       labels:
       app: topicsubscriber
       spec:
       containers:
       - name: topicsubscriber
       image: registry.gitlab.com/tfg1010711/topicsubscriber:latest
       imagePullPolicy: Always
       ports:
              - containerPort: 80
       env:
              - name: RABBIT_HOST
              value: rabbitmq.default.svc.cluster.local
              - name: RABBIT_PORT
              value: "5672"
              - name: RABBITMQ_DEFAULT_USER
              valueFrom:
              secretKeyRef:
                     name: rabbitmq-credentials
                     key: RABBITMQ_DEFAULT_USER
              - name: RABBITMQ_DEFAULT_PASS
              valueFrom:
              secretKeyRef:
                     name: rabbitmq-credentials
                     key: RABBITMQ_DEFAULT_PASS
       imagePullSecrets:
       - name: gitlab-registry
```

A continuación se indica este nuevo deploy a Kubernetes mediante la línea de comandos con la siguiente instrucción.

```
kubectl apply -f topicsubscriber-deployment.yaml
```

Para exponer el servicio, se necesitaría un archivo adicional de tipo Service, ya que en este punto solo es visible desde dentro del clúster. El fichero service expone el pod al exterior, en este caso publica el puerto 80 interno, haciéndolo visible mediante el puerto 8080. El campo type está establecido en "LoadBalancer", lo que significa que el Servicio se expondrá externamente utilizando el balanceador de carga del proveedor de la nube. Si se realiza un despliegue local sin balanceador se utiliza un tipo ClusterIP. El campo selector especifica la etiqueta de los pods con las que está relacionada este servicio para que se enrute el tráfico hacia ellos.

```
apiVersion: v1
kind: Service
metadata:
  name: topicsubscriber-service
spec:
  selector:
       app: topicsubscriber
  ports:
       - name: http
       port: 80
       targetPort: 80
```

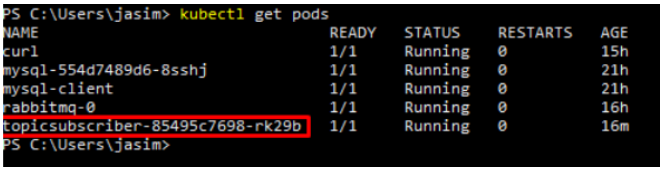
```
  type: ClusterIP
```

Por último se activa el servicio en el clúster.

```
kubectl apply -f topicsubscriber-service.yaml
```

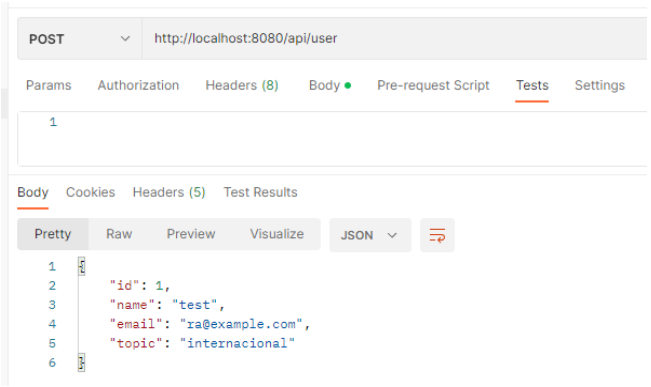
Prueba del envío de mensajes

Para probarlo el despliegue en el clúster local se consulta el nombre del pod para obtener su identificador y se hace un *portforward*

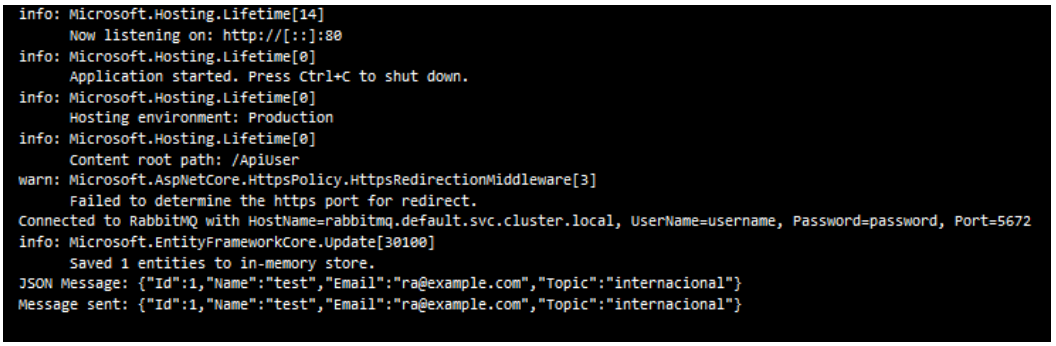



```
kubectl port-forward topicsubscriber-85495c7698-rk29b 8080:80
```

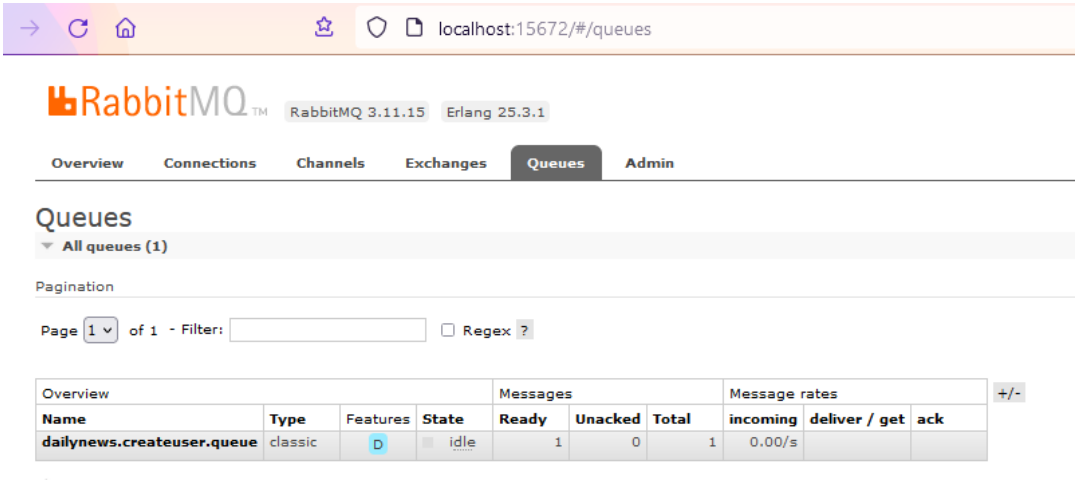
Se realiza una petición con postman para realizar el envío



Ahora, se comprueba el envío del mensaje en los logs del cliente



Por último, también se hace un *portforward* para acceder al panel de administración de rabbit y comprobar la recepción del mensaje.



Anexo III Creación de servicio msgquery en Java de consulta de API externa

Por qué Java

Java es un lenguaje de programación que se puede ejecutar en cualquier sistema operativo que tenga una máquina virtual Java (JVM). La portabilidad es un elemento deseable, ya que permite desarrollar una aplicación en una plataforma y ejecutarla en otra plataforma sin tener que reescribir o compilar el código fuente.

Por otro lado, Java es un lenguaje de programación orientado a objetos que se puede utilizar para crear aplicaciones escalables y robustas, presentando una gran variedad de herramientas y bibliotecas te permiten escalar una aplicación de manera eficiente.

También hay que tener en cuenta que Java cuenta con una amplia comunidad muy activa, con actualizaciones periódicas y una gran cantidad de herramientas, bibliotecas y frameworks que aceleran el proceso de desarrollo.

Por último, el entorno de las herramientas de Java y el lenguaje son bastante populares en los estudios de grado, así que no es ningún desconocido y esto permite rebajar los umbrales de incertidumbre a la hora de encarar un proyecto de esta tipología.

Qué hace este servicio

Este servicio se encarga de tener un registro con los intereses a los que se ha suscrito un usuario. De manera recurrente realiza una conexión con una API externa de noticias en la que consulta estos intereses y a continuación guardando registro de los mismos.

Proceso de creación de un servicio en Java

Para la creación del servicio en java se va a utilizar el framework de Spring boot, lo que acelera mucho el desarrollo al implementar una aplicación de servicios completa con una configuración por defecto. Para ello, se genera la aplicación mediante Spring initializer, configurando el proyecto mediante la selección de dependencias que van a utilizarse. Como gestor de dependencias se va a utilizar Maven, una herramienta bastante común en este entorno.

El proceso de desarrollo de este servicio con Spring Boot sigue la estructura siguiente:

Creación del modelo de datos: En este paso, se implementan las clases que se utilizarán en el servicio para representar los objetos. Además, se implementa el patrón DTO para comunicar y mostrar solo la parte del modelo que sea necesaria en cada contexto, lo que reduce la cantidad de datos en cada transacción y mejora la seguridad. Para facilitar la creación de objetos, se utiliza el patrón Builder. Además, se aplican interfaces a las clases para cumplir con el principio SOLID de inversión de dependencias mediante la utilización del patrón de inyección de dependencias.
Repositorios: Se crean los repositorios, que son los encargados de interactuar con la base de datos. Estos repositorios están definidos como interfaces que extienden de la interfaz JpaRepository de Spring Data, lo que facilita mucho su implementación.

Servicios: Se definen los servicios que albergan la lógica de negocio, enviando los datos obtenidos desde los repositorios a los controladores.

Controladores: Para recibir las peticiones se crean los controladores, los cuales exponen una interfaz que permite realizar consultas mediante HTTP.

Pruebas: Se realizan pruebas unitarias y de integración para asegurarse de que el servicio funciona correctamente. Además, se hace uso de Swagger, que permite documentar y validar el servicio.

Despliegue: Una vez que el servicio ha sido desarrollado y probado, se procede a su despliegue en el entorno de producción. Para ello se puede utilizar una amplia variedad de herramientas, como por ejemplo Docker o Kubernetes, que son las que se van a utilizar en este proyecto y que permiten desplegar el servicio de manera sencilla y escalable.

Creación del servicio con Spring initializr

Para la creación del servicio se utiliza la siguiente configuración:

- El tipo de proyecto creado es Maven.
- Lenguaje Java versión 17.

- La versión de spring Boot utilizada es la 3.0.4.
- Para el empaquetado del proyecto se utiliza un archivo tipo Jar

A continuación se agregan las siguientes dependencias:

- **Spring Boot DevTools**, para mejorar el proceso de desarrollo.
- **Lombok**, que nos permite utilizar anotaciones que simplifican la codificación del servicio.
- **Spring Web**, elemento básico para la creación de aplicaciones RESTful.
- **Thymeleaf**, que permite tener un sistema de plantillas html.
- **Spring Data JPA SQL**, para tener acceso a una interfaz que permite activar la persistencia.
- **H2 Database SQL**, para tener una base de datos en memoria para realizar pruebas.
- **PostgreSQL Driver SQL**, un driver que permite conectarse a PostgreSQL.
- **Spring for RabbitMQ**, para la publicación, subscripción, almacenamiento y proceso de flujos de registros.
- **Spring Boot Actuator**, para monitorizar y tener métricas del estado del servicio.
- **Spring Boot DevTools**, para mejorar el proceso de desarrollo.
- **Lombok**, que nos permite utilizar anotaciones que simplifican la codificación del servicio.
- **Spring for RabbitMQ**, para la publicación, subscripción, almacenamiento y proceso de flujos de registros.
- **Spring Boot Actuator**, para monitorizar y tener métricas del estado del servicio.

Una vez construido el proyecto, se agrega swagger al POM. Para agregar swagger se utiliza la siguiente instrucción. También se agrega un validador de hibernate para que no se generen errores.

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.6.12</version>
</dependency>
<dependency>
               <groupId>org.hibernate.validator</groupId>
               <artifactId>hibernate-validator</artifactId>
</dependency>
```

Proceso de CI/CD

A continuación, se muestra la implementación de un sistema del despliegue automatizado para una aplicación Java utilizando GitLab CI/CD.

El pipeline se compone de las siguientes etapas:

1. Etapa de construcción (build-job): En esta etapa, se utiliza la imagen "maven:3.8.4-openjdk-17-slim" como base para compilar y empaquetar la aplicación Java. Se ejecuta el comando "mvn package" para generar el archivo JAR de la aplicación. Los artefactos generados en esta etapa, en este caso, el archivo JAR, se conservan para su uso posterior.

2. Etapa de pruebas (test-job): En esta etapa, se ejecutan las pruebas de la aplicación utilizando la misma imagen "maven:3.8.4-openjdk-17-slim". Se ejecuta el comando "mvn test" para realizar las pruebas unitarias y de integración. Esto garantiza que la aplicación cumpla con los criterios de calidad definidos.

3. Etapa de registro de imágenes Docker (docker-register-job): Esta etapa se encarga de registrar las imágenes Docker generadas en un registro de contenedores. Se utiliza la imagen "docker:latest" y se configura el servicio "docker:dind" para interactuar con Docker. Primero, se crea una etiqueta de imagen única utilizando el commit actual y el ID del trabajo. Luego, se realiza el inicio de sesión en el registro de contenedores especificado en las variables de entorno. A continuación, se construye la imagen Docker de la aplicación y se empuja al registro utilizando la etiqueta generada. Además, se crea una etiqueta adicional llamada "latest" para facilitar el despliegue de la última versión de la imagen.

4. Etapa de despliegue (deploy-prod): En esta etapa final, se implementa la aplicación en un entorno de producción.

```
image: maven:3.8.4-openjdk-17-slim

stages:
  - build
  - test
  - docker-register
  - deploy

build-job:
  stage: build
  script:
       - echo "Hello, $GITLAB_USER_LOGIN! Let's maven package."
       - mvn package
  artifacts:
       paths:
       - target/*.jar

test-job:
  stage: test
  script:
       - echo "This job tests"
```

```
      - mvn test

docker-register-job:
  image: docker:latest
  stage: docker-register
  services:
      - docker:dind

  before_script:
      # Create a tag using the commit and the job ID
      - export TAG=${CI_COMMIT_SHORT_SHA}-${CI_JOB_ID}

      # Login into the containers registry
      - docker login $CI_REGISTRY -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"

  script:
      # Build and push the real image
      - docker build -t $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG .
      - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG

      # Retag and push the image to have a latest tag
      - docker tag $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG
$CI_REGISTRY/${CI_PROJECT_PATH}:latest
      - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:latest
  only:
      - master

deploy-gke:
  stage: deploy
  image:
    name: google/cloud-sdk
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."

    - echo "$BASE64_GOOGLE_CREDENTIALS" | base64 -d > google-credentials.json
    - gcloud auth activate-service-account --key-file=google-credentials.json

    - gcloud config set project $TF_VAR_gcp_project

    - gcloud container clusters get-credentials $TF_VAR_cluster_name --region
$TF_VAR_gcp_region
    - >
      curl --header "Authorization: Bearer $CI_JOB_TOKEN" -o msgquery-deployment.yaml
"https://gitlab.com/tfg1010711/msgquery/-/raw/master/ctl/msgquery-deployment.yaml"
    - >
      curl --header "Authorization: Bearer $CI_JOB_TOKEN" -o msgquery-service.yaml
"https://gitlab.com/tfg1010711/msgquery/-/raw/master/ctl/msgquery-service.yaml"
    - kubectl apply -f msgquery-deployment.yaml
    - kubectl apply -f msgquery-service.yaml
  dependencies:
    - build
  only:
    - master
```

Creación de Docker y Kubernetes

Preparación del Dockerfile

En primer lugar, se crea en la raíz de nuestro proyecto el fichero Dockerfile al que agregamos el código siguiente

```
FROM openjdk:17-alpine
ADD target/msgquery-*.jar app.jar
EXPOSE 8082
ENTRYPOINT
["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=docker","-j
ar","app.jar"]
```

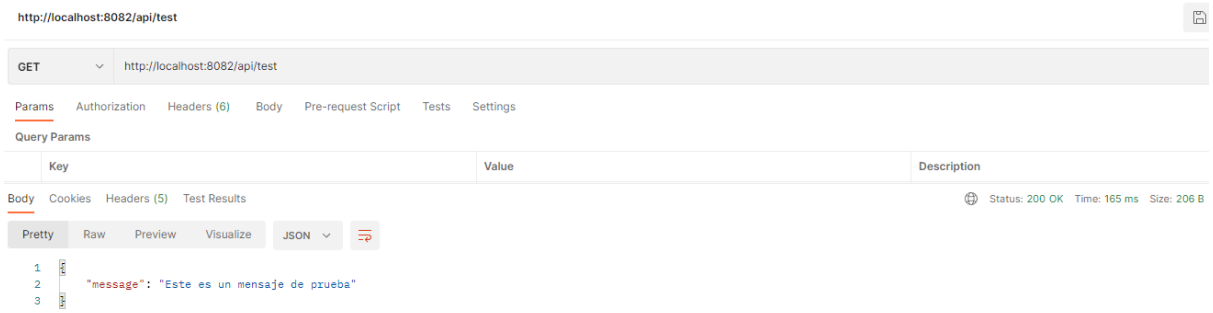
A continuación, se prepara el docker

```
$ docker build --tag msgquery:v1 .
```

Por último, se procede a arrancar el servicio docker en segundo plano

```
$ docker run --name msgquery -d -p 8082:8082 msgquery:v1
```

Como resultado, tenemos la aplicación disponible localmente en el puerto 8082



Preparación del Pod de kubernetes

Mediante el siguiente archivo se especifica que se desea un único pod ("réplica: 1") y que la imagen del contenedor que se utilizará es "msgquery:v1". El contenedor se expone internamente a través del puerto 8082. El selector "matchLabels" se utiliza para emparejar el pod con el servicio correspondiente.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: msgquery-deployment
  labels:
       app: msgquery
spec:
  replicas: 1
  selector:
       matchLabels:
       app: msgquery
  template:
       metadata:
       labels:
```

```
      app: msgquery
      spec:
      containers:
      - name: msgquery
      image: msgquery:v1
      ports:
      - containerPort: 8082
```

A continuación, se indica este nuevo deploy a Kubernetes mediante la línea de comandos con la siguiente instrucción.

```
kubectl apply -f msgquery-deployment.yaml
```

Para exponer el servicio, se necesitaría un archivo adicional de tipo Service, ya que en este punto solo es visible desde dentro del clúster. El fichero service expone el POD al exterior, en este caso publica el puerto 8082 interno haciéndolo visible mediante el puerto 8082. El campo type está establecido en "LoadBalancer", lo que significa que el Servicio se expondrá externamente utilizando el balanceador de carga del proveedor de la nube. El campo selector especifica la etiqueta de los pods con las que está relacionada este servicio para que se enrute el tráfico hacia ellos.

```
apiVersion: v1
kind: Service
metadata:
  name: msgquery-service
  labels:
       app: msgquery
spec:
  type: LoadBalancer
  ports:
       - name: http
       port: 8082
       protocol: TCP
       targetPort: 8082
  selector:
       app: msgquery
```

Por último se activa el servicio en el clúster.

```
kubectl apply -f msgquery-service.yaml
```

Configuración de RabbitMQ en Java

La clase que contiene este código se puede considerar como la clase principal de configuración en el sistema entre el microservicio y RabbitMQ. Es aquí donde se definen los componentes necesarios para el sistema de notificaciones, como el intercambio, las colas y los enlaces.

Además de esta clase principal, es importante mencionar que también existe un archivo de propiedades que complementa la configuración. En ese archivo, se pueden especificar diferentes

valores de configuración, como URL del broker, puertos, credenciales y parámetros personalizables. Estos valores se pueden cargar en la aplicación durante su inicialización.

La combinación de la clase principal de configuración y el archivo de propiedades permite establecer y personalizar la configuración del sistema de manera modular y centralizada. Esto facilita la adaptación del sistema a diferentes entornos y requisitos específicos, sin necesidad de modificar directamente el código fuente.

```
@Bean
public FanoutExchange fanoutExchange() {
   return new FanoutExchange(exchangeName);
}

/**
* Bean method to create a queue for email notifications.
*
* @return The {@link Queue} instance.
*/
@Bean
public Queue queueMail() {
   return new Queue(queueMail);
}

/**
* Bean method to create a queue for historical notifications.
*
* @return The {@link Queue} instance.
*/
@Bean
public Queue queueHist() {
   return new Queue(queueHist);
}

/**
* Bean method to create a binding between the historical notification queue and the exchange.
*
* @param queueHist        The {@link Queue} instance representing the historical notification
queue.
* @param fanoutExchange The {@link FanoutExchange} instance representing the fanout exchange.
* @return The {@link Binding} instance representing the binding.
*/
@Bean
public Binding bindingHist(Queue queueHist, FanoutExchange fanoutExchange) {
   return BindingBuilder.bind(queueHist).to(fanoutExchange);
}

/**
* Bean method to create a binding between the mail notification queue and the exchange.
*
* @param queueMail      The {@link Queue} instance representing the mail notification queue.
* @param fanoutExchange The {@link FanoutExchange} instance representing the fanout exchange.
* @return The {@link Binding} instance representing the binding.
*/
@Bean
public Binding bindingMail(Queue queueMail, FanoutExchange fanoutExchange) {
   return BindingBuilder.bind(queueMail).to(fanoutExchange);
}
```

Anexo IV Creación del servicio msgdispatch en Java para el envió de correo

Qué hace este servicio

Este servicio se encarga de leer del broker de mensajería los correos que debe enviar a las personas usuarias.

Proceso de creación del servicio en Java

Para la creación de este servicio en java se va a utilizar el framework de Spring boot, Para ello, se genera la aplicación mediante Spring initializer, configurando el proyecto mediante la selección de dependencias que van a utilizarse. Como gestor de dependencias se va a utilizar Maven, una herramienta bastante común en este entorno.

El proceso de desarrollo de este servicio con Spring Boot sigue la estructura siguiente:

Creación del modelo de datos: En este paso, se implementan las clases que se utilizarán en el servicio para representar los objetos.

Servicios: Se definen los servicios que albergan la lógica de negocio, enviando los datos obtenidos desde los repositorios a los controladores.

Pruebas: Se realizan pruebas unitarias y de integración para asegurarse de que el servicio funciona correctamente. Además, se hace uso de Swagger, que permite documentar y validar el servicio.

Despliegue: Una vez que el servicio ha sido desarrollado y probado, se procede a su despliegue en el entorno de producción. Para ello se puede utilizar una amplia variedad de herramientas, como por ejemplo Docker o Kubernetes, que son las que se van a utilizar en este proyecto y que permiten desplegar el servicio de manera sencilla y escalable.

Creación del servicio con Spring initializr

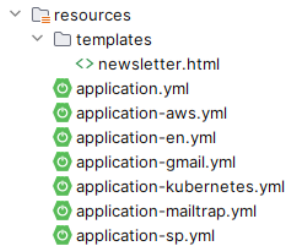
Para la creación del servicio se utiliza la siguiente configuración:

- El tipo de proyecto creado es Maven.
- Lenguaje Java versión 17.
- La versión de spring Boot utilizada es la 3.0.5.
- Para el empaquetado del proyecto se utiliza un archivo tipo Jar

A continuación se agregan las siguientes dependencias:

- **Lombok**, que nos permite utilizar anotaciones que simplifican la codificación del servicio.
- **Spring for RabbitMQ**, para la publicación, subscripción, almacenamiento y proceso de flujos de registros.
- **Spring Boot Actuator,**, para monitorizar y tener métricas del estado del servicio.
- **Java Mail Sender I/O** Send email using Java Mail and Spring Framework's JavaMailSender.
- **Spring Boot DevTools**, para mejorar el proceso de desarrollo.
- **Spring Boot Actuator,** para monitorizar y tener métricas del estado del servicio.

El proyecto utiliza diferentes *profiles,* de los que se han seleccionado mailtrap para la selección del proveedor de correo, kubernetes para el acceso a los servidores y sp como idioma mostrado en las plantillas.



Creación de Docker y Kubernetes

Preparación del Dockerfile

En primer lugar se crea en la raíz de nuestro proyecto el fichero Dockerfile al que agregamos el código siguiente.

```
FROM openjdk:17-alpine
ADD target/msgdispatch-*.jar app.jar
EXPOSE 8088
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=mailtrap,kubernetes,sp","-jar","app.jar"]
```

A continuación se prepara el docker.

```
$ docker build --tag msgdispatch:v1 .
```

Por último, se procede a arrancar el servicio docker en segundo plano

```
$ docker run --name msgdispatch-d -p 8088:8088 msgdispatch:v1
```

Como resultado, tenemos la aplicación disponible localmente en el puerto 8088

Preparación del Pod de kubernetes

Mediante el siguiente archivo se especifica que se desea un único pod ("réplica: 1") y que la imagen del contenedor que se utilizará es "msgquery:v1". El contenedor se expone internamente a través del puerto 8088. El selector "matchLabels" se utiliza para emparejar el pod con el servicio correspondiente.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: msgdispatch-deployment
  labels:
       app: msgdispatch
spec:
  replicas: 1
  selector:
       matchLabels:
       app: msgdispatch
  template:
       metadata:
       labels:
       app: msgdispatch
       spec:
       containers:
       - name: msgdispatch
       image: msgdispatch:v1
       ports:
       - containerPort: 8088
```

A continuación se indica este nuevo deploy a Kubernetes mediante la línea de comandos con la siguiente instrucción.

```
kubectl apply -f msgdispatch-deployment.yaml
```

Para exponer el servicio, se necesitaría un archivo adicional de tipo Service, ya que en este punto solo es visible desde dentro del clúster. El fichero service expone el POD al exterior, en este caso publica el puerto 8088 interno haciéndolo visible mediante el puerto 8088. El campo type está establecido en "LoadBalancer", lo que significa que el Servicio se expondrá externamente utilizando el balanceador de carga del proveedor de la nube. El campo selector especifica la etiqueta de los pods con las que está relacionada este servicio para que se enrute el tráfico hacia ellos.

```
apiVersion: v1
kind: Service
metadata:
  name: msgdispatch-service
  labels:
       app: msgdispatch
```

```
spec:
  type: LoadBalancer
  ports:
       - name: http
       port: 8088
       protocol: TCP
       targetPort: 8088
  selector:
       app: msgdispatch
```

Por último se activa el servicio en el clúster.

```
kubectl apply -f msgdispatch-service.yaml
```

Anexo V Creación del pipeline

Qué es un pipeline

Un pipeline es una secuencia de pasos automatizados que se ejecutan en un orden específico para compilar, probar y desplegar una aplicación de manera continua y se enmarca dentro de las prácticas de CI/CD. Cada paso en el pipeline es una tarea y cada tarea realiza una acción específica, como compilar el código fuente, ejecutar pruebas unitarias, construir contenedores de Docker o desplegar la aplicación en un entorno de producción.

La ventaja de usar pipelines es que permiten automatizar todo el proceso de desarrollo de software, lo que puede ayudar a:

Ahorrar tiempo y reducir errores: Al automatizar tareas, como la compilación, pruebas y despliegue, se ahorra tiempo y se reducen los errores manuales.

Facilitar la colaboración: Un pipeline permite que todos los miembros del equipo trabajen en el mismo proceso de desarrollo y, por lo tanto, puedan colaborar fácilmente.

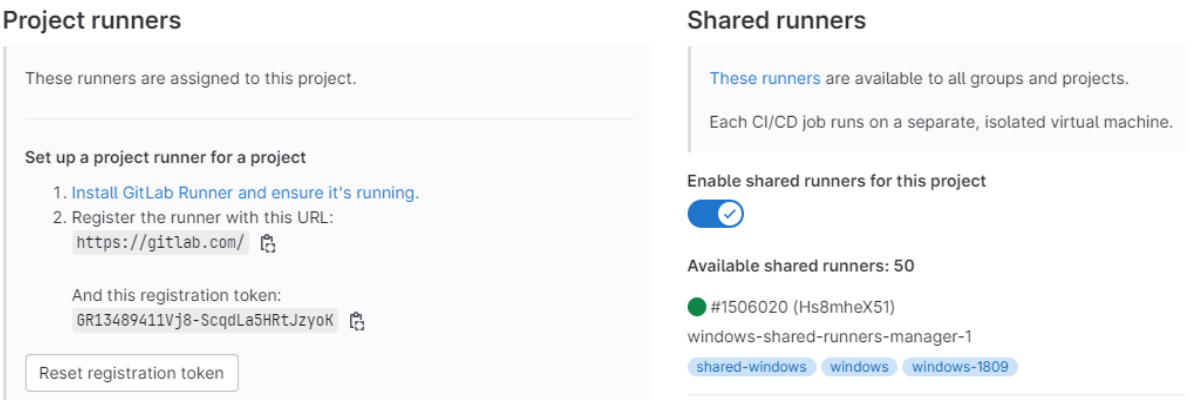
Mejorar la calidad del software: Al automatizar las pruebas, se comprueba que el código funciona correctamente y que cumple con los estándares de calidad.

Desplegar de manera más rápida: Los pipelines automatizados permiten una entrega continua de software, lo que significa que se puede desplegar rápidamente nuevas versiones de un aplicativo.

Facilitar la escalabilidad: Los pipelines automatizados permiten escalar la infraestructura de manera eficiente, lo que significa que se pueden agregar más recursos para satisfacer las demandas de la aplicación sin comprometer la calidad del software.

Creación de un pipeline en gitlab

Para ejecutar un pipeline necesitamos un runner, un agente que se encarga de ejecutar las tareas que se definen en el proceso de CI/CD. Para comprobar si existen runners activos en el proyecto se accede a settings, CI/CD y se comprueba que exista alguno en verde.



Existen dos tipos de runner. Por una parte, están los propios del proyecto, que se ejecutan solo en este ámbito y permiten una mayor personalización, aunque son algo más complejos de utilizar, pues necesitan una instalación local o en un servidor externo. Por otro lado, están los runners compartidos, que se ejecutan para cualquier proyecto, pero a cambio de una menor flexibilidad.

Una vez que el runner está activado, es necesario agregar un archivo .gitlab-ci.yml en la raíz del repositorio. Este archivo contiene el script de ejecución que se ejecutará automáticamente cada vez que se produzcan cambios en el repositorio.

Este archivo es un ejemplo de cómo se puede configurar un archivo .gitlab-ci.yml para un proyecto en GitLab. El archivo especifica las etapas y trabajos que se deben ejecutar para el pipeline CI/CD.

El siguiente archivo es un ejemplo de pipeline en el que se definen varios trabajos o jobs y diferentes stages, o etapas. Los *jobs* indican qué hacer y los *stages* cuándo hacerlo. En este ejemplo, cada trabajo tiene asignado un stage, pero en un archivo de GitLab CI se pueden definir múltiples stages. Cada trabajo se ejecutará en el *stage* correspondiente y se puede controlar el orden de ejecución de los stages mediante la asignación de nombres alfanuméricos a cada uno de ellos. Además, los stages también pueden definir dependencias entre sí, lo que permite ejecutar los trabajos en un orden específico.

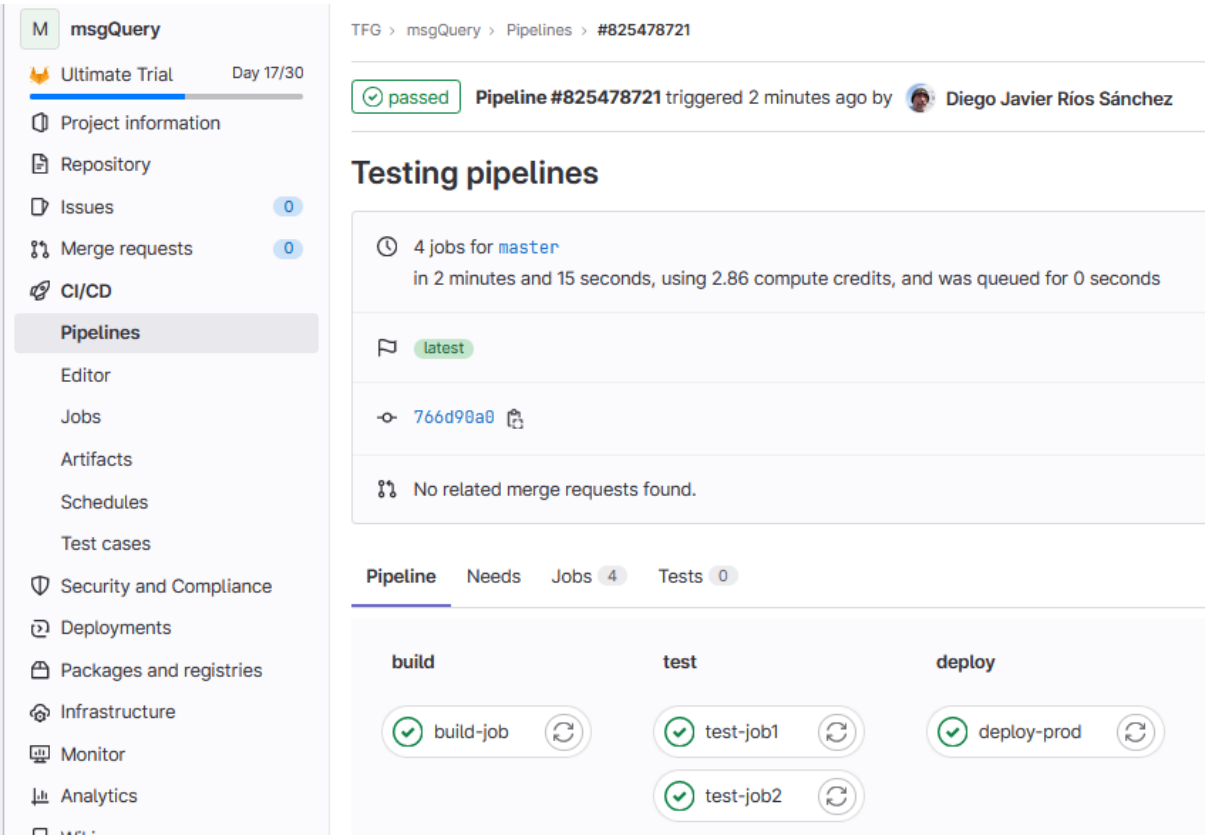
El primer trabajo, build-job, se ejecuta en la etapa de "build" y simplemente imprime un mensaje de saludo que incluye el nombre de usuario que inició sesión en GitLab.

Los dos trabajos de prueba, test-job1 y test-job2, se ejecutan en la etapa de "test". Ambos trabajos imprimen mensajes de prueba, pero el segundo trabajo incluye una espera de 20 segundos para simular una prueba que lleva más tiempo.

El último trabajo, deploy-prod, se ejecuta en la etapa de "deploy" y es responsable de implementar el código en el entorno de producción. En este caso, el trabajo imprime un mensaje que indica que está implementando algo desde la rama actual.

```
build-job:
  stage: build
  script:
      - echo "Hello, $GITLAB_USER_LOGIN!"
```

```
test-job1:
  stage: test
  script:
       - echo "This job tests something"

test-job2:
  stage: test
  script:
       - echo "This job tests something, but takes more time than test-job1."
       - echo "After the echo commands complete, it runs the sleep command for 20
seconds"
       - echo "which simulates a test that runs 20 seconds longer than test-job1"
       - sleep 20

deploy-prod:
  stage: deploy
  script:
       - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
```



Creación y registro del docker en gitlab

Para registrar contenedores en gitlab se agrega la siguiente sección al archivo .gitlab-ci.yml. Es importante indicar que los registros de contenedor solo se ejecuten bajo ciertas condiciones determinadas, en este caso cuando existan cambios en la rama main.

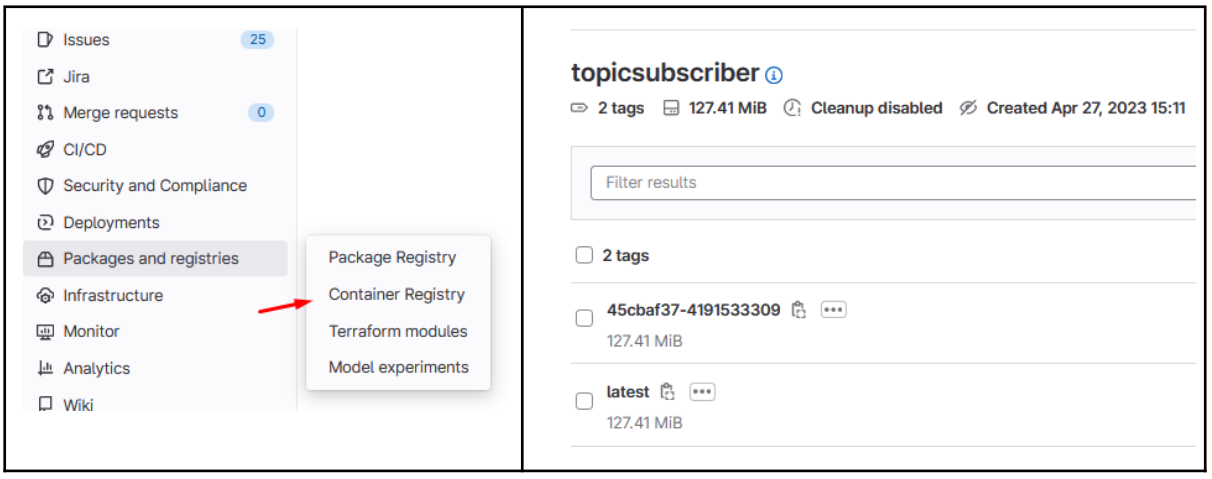
Primero en el stage

```
stages:
  - build
  - test
  - docker-register
  - deploy
….
```

Y a continuación el *stage* dentro del *job.* En este proceso se realizan dos *push* de la imagen, de forma que mantenemos un histórico a la vez que se indica con el tag latest cuál es la última versión actualizada

```
docker-register-job:
  image: docker:latest
  stage: docker-register
  services:
        - docker:dind

  before_script:
        # Create a tag using the commit and the job ID
        - export TAG=${CI_COMMIT_SHORT_SHA}-${CI_JOB_ID}

        # Login into the containers registry
        - docker login $CI_REGISTRY -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"

  script:
        # Build and push the real image
        - docker build -t $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG .
        - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG

        # Retag and push the image to have a latest tag
        - docker tag $CI_REGISTRY/${CI_PROJECT_PATH}:$TAG
$CI_REGISTRY/${CI_PROJECT_PATH}:latest
        - docker push $CI_REGISTRY/${CI_PROJECT_PATH}:latest
  only:
        - main
```

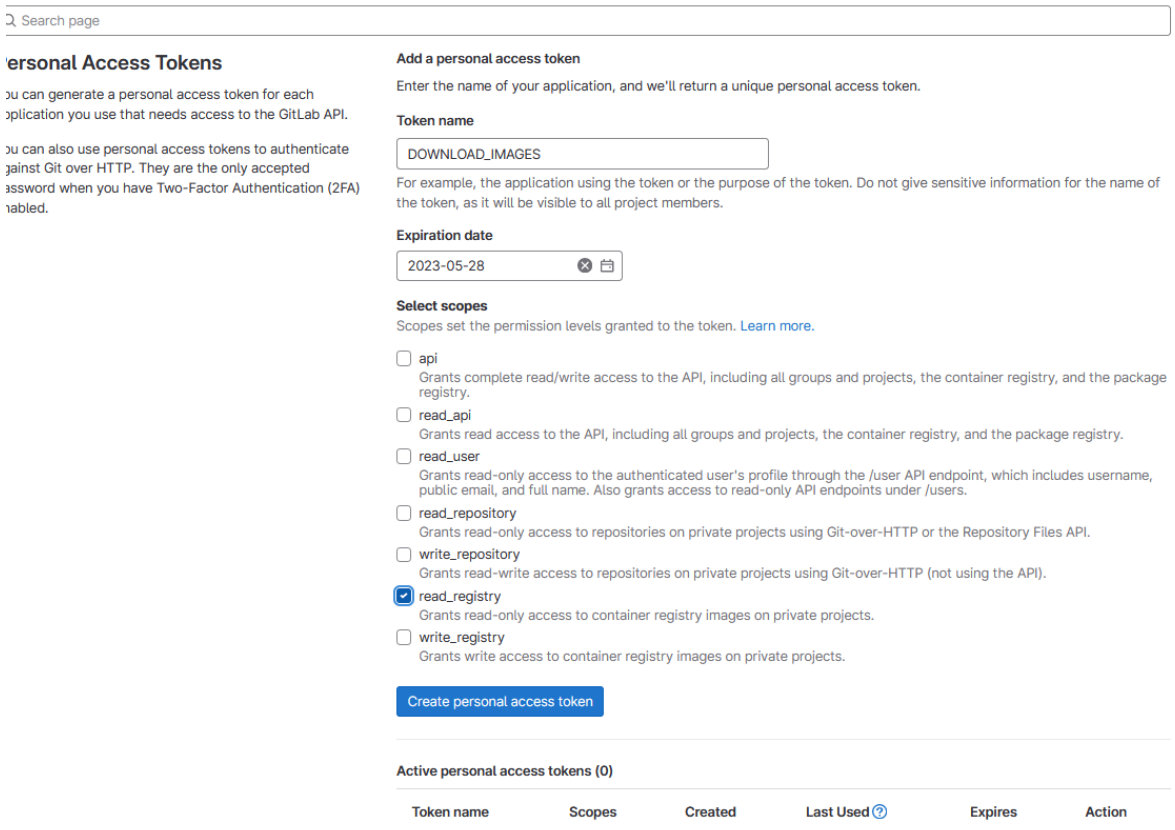
Si todo va correctamente se puede ver la imagen creada en nuestro registro de containers



Probar la imagen registrada en Gitlab

Para descargar una imagen Docker desde el registro de GitLab, se deben seguir los siguientes pasos:

En primer lugar, se crea un personal access token de solo lectura para el proyecto y de esta manera poder leer las imágenes del registro de gitlab.

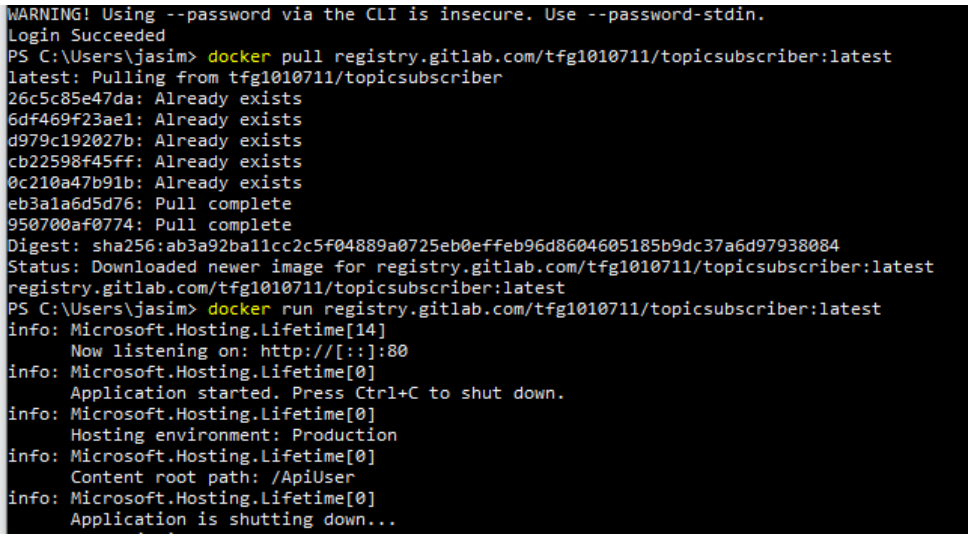


Se debe copiar el token y guardarse en un sitio seguro, ya que no se podrá volver a recuperar.

A continuación, ya se puede hacer login desde el sistema.

```
$ docker login registry.gitlab.com -u <username> -p <token>
```

Si se procede a realizar un pull ya se tiene disponible la imagen en una máquina local



A partir de este punto ya se puede hacer uso local del contenedor.

```
docker run --name topicsubscriber -d -p 8080:80
registry.gitlab.com/tfg1010711/topicsubscriber:latest
```

Anexo VI Instalación y configuración de RabbitMQ

Por qué RabbitMQ

RabbitMQ es una buena opción para proyectos *de microservicios* y proporciona clientes para múltiples lenguajes, incluidos Java, .NET y Python. Es flexible para adaptarse a diferentes patrones de mensajería y utiliza el protocolo AMQP (Advanced Message Queuing Protocol), que es un protocolo de mensajería estándar y de plataforma cruzada que permite la interoperabilidad entre diferentes lenguajes de programación y sistemas de mensajería.

Agregar Rabbit a Spring Boot

Para agregar RabbitMQ a Spring Boot se agrega al archivo POM la siguiente dependencia.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Ahora se deben agregar las siguientes propiedades para RabbitMQ en el archivo properties del proyecto java que se vaya a utilizar.

```
spring:
        rabbitmq:
        host: localhost
        port: 5672
        username: guest
        password: guest
        exchange-name: dailynews.send.notification
        queue-mail: dailynews.send.notification.queue.mail
        queue-hist: dailynews.send.notification.queue.hist
        routing-hist: dailynews.send.notification.routing.hist
        routing-mail: dailynews.send.notification.routing.mail
```

Se puede consultar más información específica sobre la configuración de RabbitMQ en las secciones correspondientes de los anexos Configuración de RabbitMQ para C# y Configuración de RabbitMQ para Java de la creación de dichos servicios.

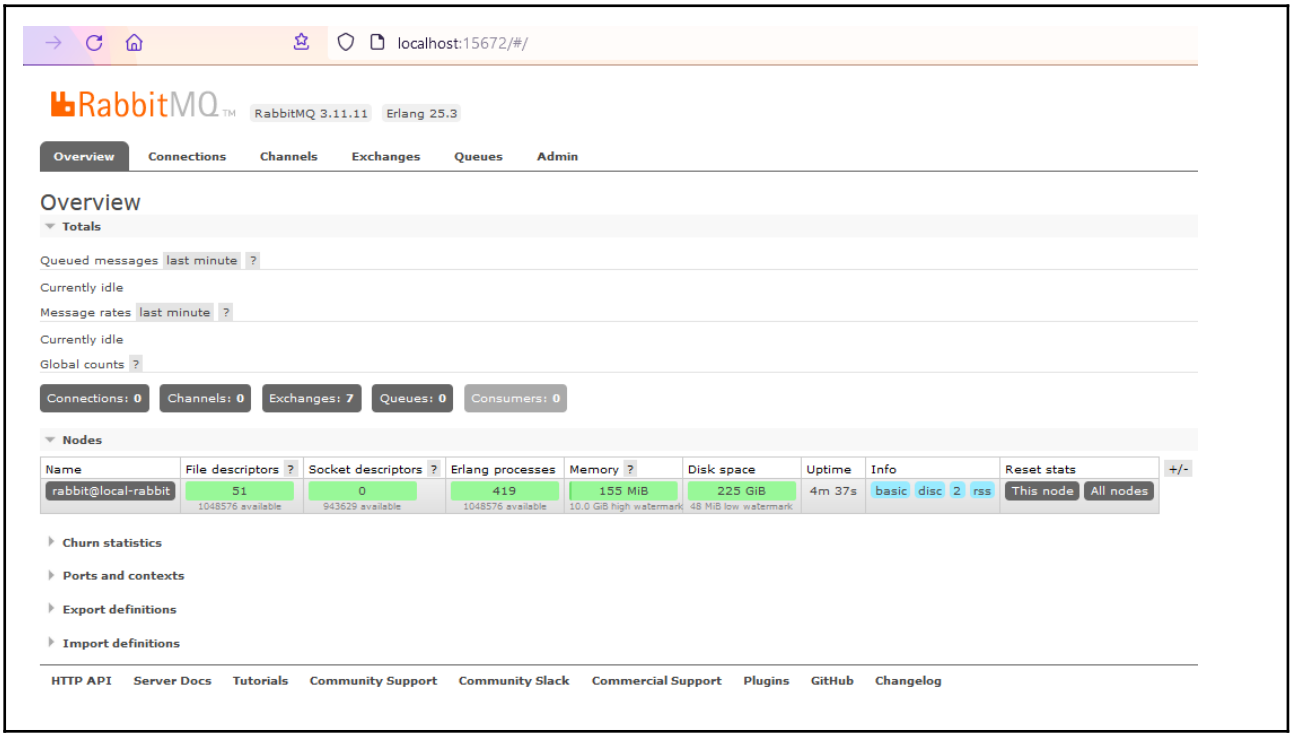
Creación de Docker y Kubernetes

Preparación del Dockerfile

Para instalar el docker de RabbitMQ con el *plugin* de gestión, ejecutamos el siguiente comando desde consola.

```
docker run -d --hostname local-rabbit --name local-rabbit -p 15672:15672 -p 5672:5672
rabbitmq:3-management
```

Si todo va bien, se puede acceder a la consola de gestión de RabbitMQ conectando a la dirección: http://localhost:15672 e introduciendo las credenciales de administrador (por defecto: usuario: "guest" y contraseña: "guest").



Preparación del Pod de kubernetes

Este manifiesto YAML describe un StatefulSet en Kubernetes que tiene como propósito desplegar y gestionar un clúster de RabbitMQ. Utiliza la imagen rabbitmq:3-management, que incluye la interfaz de administración de RabbitMQ.

Un StatefulSet es un controlador en Kubernetes que permite manejar aplicaciones con estado. En este caso, el StatefulSet asegura que se ejecute una réplica de RabbitMQ, con la posibilidad de escalar a más réplicas si es necesario.

La imagen rabbitmq:3-management proporciona una versión específica de RabbitMQ con soporte para la interfaz de administración. La interfaz de administración permite monitorizar y administrar el clúster de RabbitMQ de manera más sencilla, ofreciendo una interfaz web para visualizar el estado de las colas, conexiones y otros aspectos relacionados con RabbitMQ.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rabbitmq
spec:
  serviceName: rabbitmq
  replicas: 1
```

```
  selector:
       matchLabels:
       app: rabbitmq
  template:
       metadata:
       labels:
       app: rabbitmq
       spec:
       containers:
       - name: rabbitmq
       image: rabbitmq:3-management
       ports:
              - name: rabbitmq
              containerPort: 5672
              protocol: TCP
              - name: rabbitmq-mgmt
              containerPort: 15672
              protocol: TCP
       env:
              - name: RABBITMQ_DEFAULT_USER
              valueFrom:
              secretKeyRef:
              name: rabbitmq-credentials
              key: RABBITMQ_DEFAULT_USER
              - name: RABBITMQ_DEFAULT_PASS
              valueFrom:
              secretKeyRef:
              name: rabbitmq-credentials
              key: RABBITMQ_DEFAULT_PASS
       volumeMounts:
              - name: rabbitmq-data
              mountPath: /var/lib/rabbitmq/mnesia
  volumeClaimTemplates:
       - metadata:
       name: rabbitmq-data
       spec:
       accessModes: [ "ReadWriteOnce" ]
       resources:
       requests:
              storage: 10Gi
```

A continuación, se indica este nuevo deploy a Kubernetes mediante la línea de comandos con la siguiente instrucción.

```
kubectl apply -f rabbitmq-deployment.yaml
```

Para exponer el servicio, se necesita un archivo adicional de tipo Service, ya que en este punto el pod solo es visible desde dentro del clúster. El fichero service expone el POD al exterior, publicando los puertos internos mediante los puertos 5762 y el 15672, dando acceso exterior a rabbit y al panel de control. El campo selector especifica la etiqueta de los pods con las que está relacionada este servicio para que se enrute el tráfico hacia ellos. En un proveedor de la nube, el valor de type se puede establecer como LoadBalancer, lo que significa que el Servicio se expone externamente utilizando el balanceador de carga del proveedor. Como el campo type está ausente en el manifiesto, el valor predeterminado para type es de ClusterIP.

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: rabbitmq
spec:
  ports:
  - name: rabbitmq
       port: 5672
       protocol: TCP
       targetPort: 5672
  - name: rabbitmq-mgmt
       port: 15672
       protocol: TCP
       targetPort: 15672
  selector:
       app: rabbitmq
```

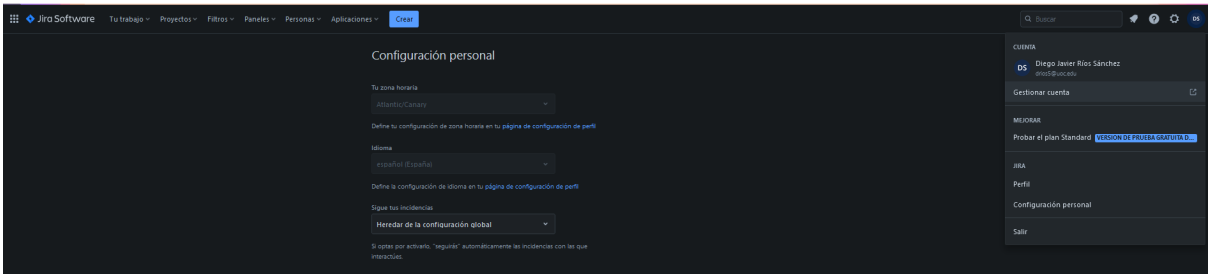
Por último, se activa el servicio en el clúster.

```
kubectl apply -f rabbitmq-service.yaml
```

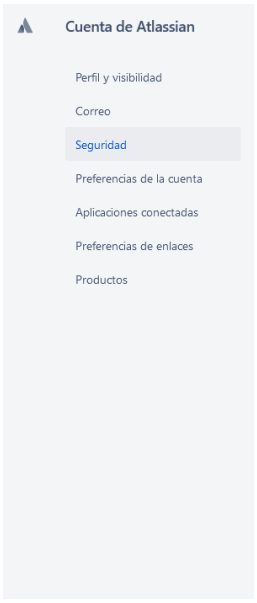
Anexo VII Configuración de la integración entre JIRA y GITLAB

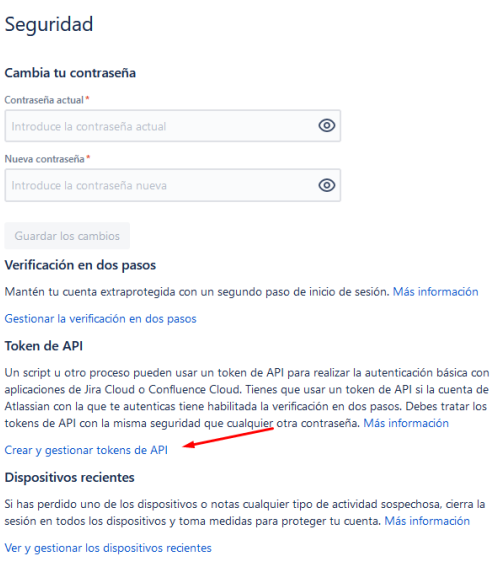
Creación de API token

Para relacionar las cuentas de JIRA y GITLAB se debe crear un API TOKEN desde JIRA que a continuación se agregará desde GITLAB Para ello se debe entrar en la opción de menú gestionar dentro de la cuenta de JIRA.

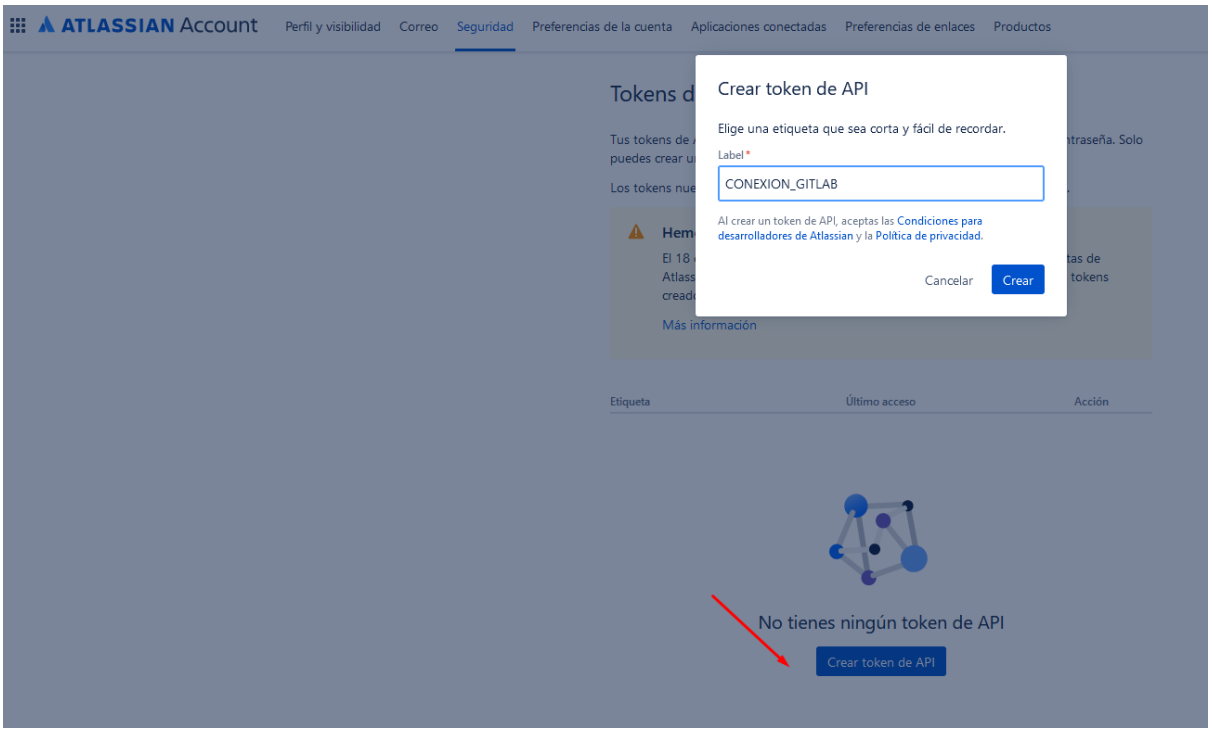


Ir a seguridad y a la opción Token de API. A continuación se debe acceder a la sección Crear y gestionar tokens de API



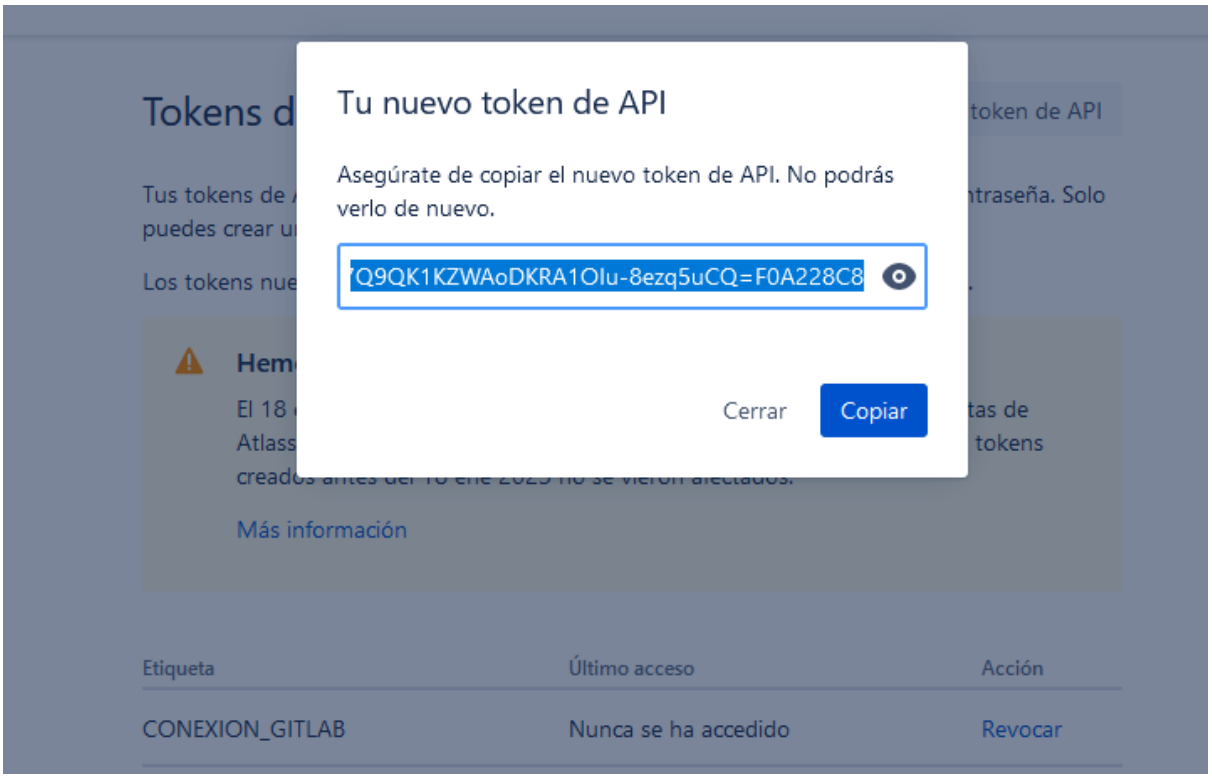


Se selecciona la creación de token y se le da una etiqueta.

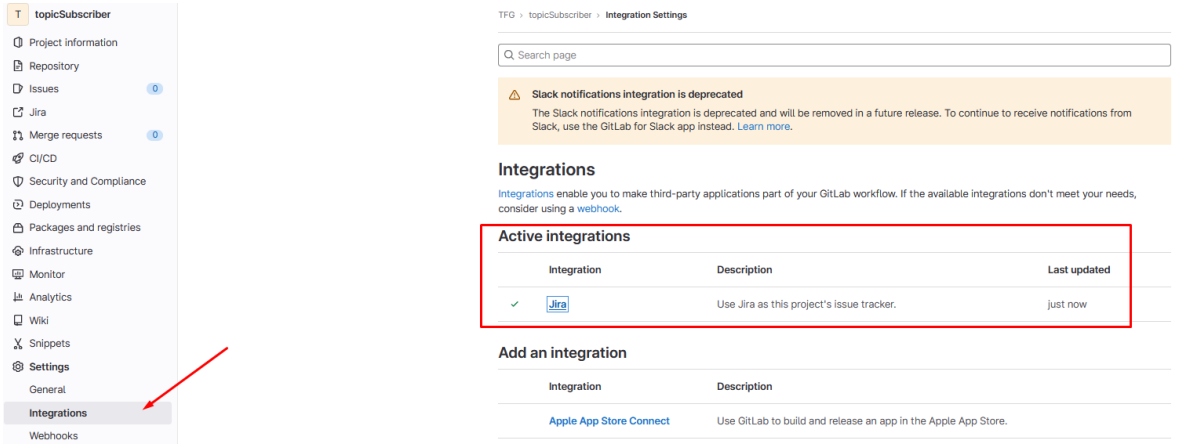


Se copia el token generado, ya que no será posible verlo en el futuro.

```
ATATT3xFfGF04rb5l5l4DHrHxQxRFRNKonUCLugdNpkMf2xvc4ER4zmsaguvd5Zu4QRJ7ZqD3UfhlgI0FP-PIM-
TXfWc-gUoC5IqWG8e9IherPxAZmkpRrKZENKbZxnZL1SEn_D1fsvrB2U5uR7mwL8kK447Q9QK1KZWAoDKRA1OIu
-8ezq5uCQ=F0A228C8
```

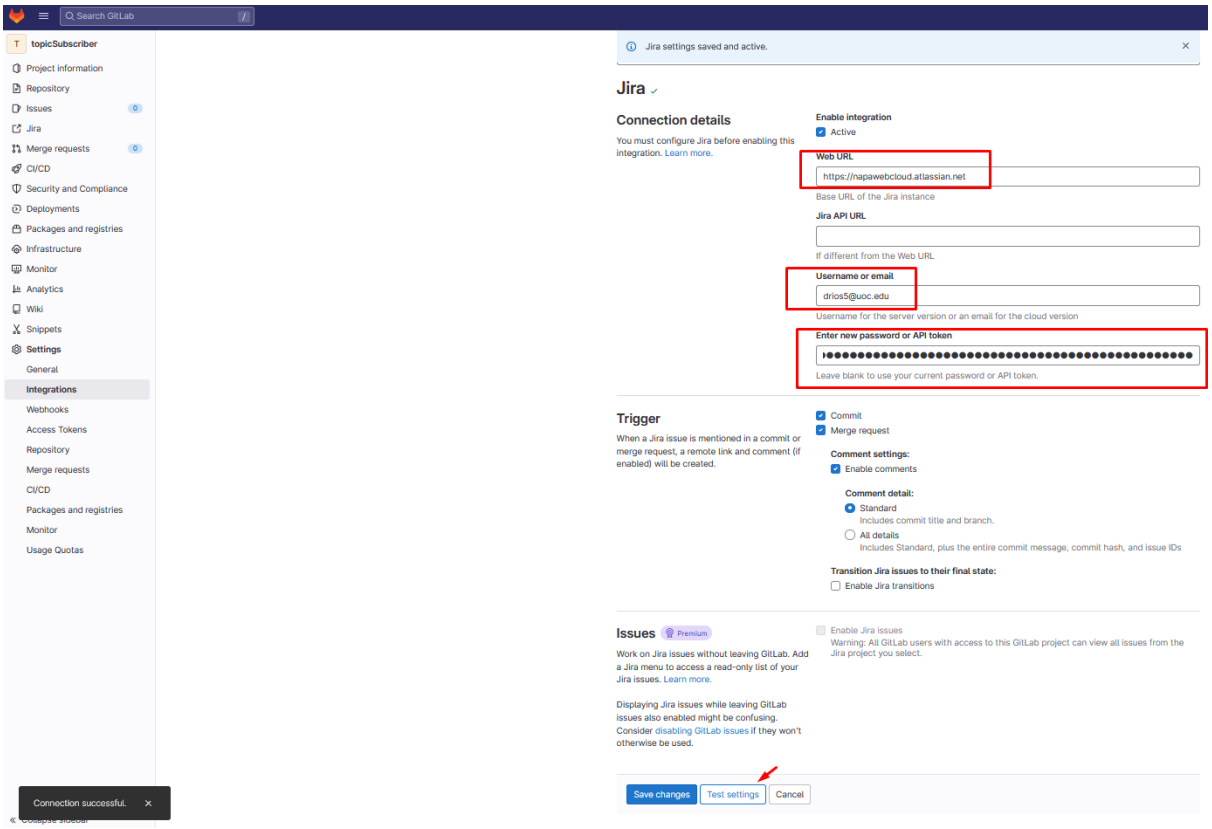


Ahora se activa la integración en GITLAB. Para ello vamos al proyecto que queremos enlazar y se selecciona en el menú la opción integrations.

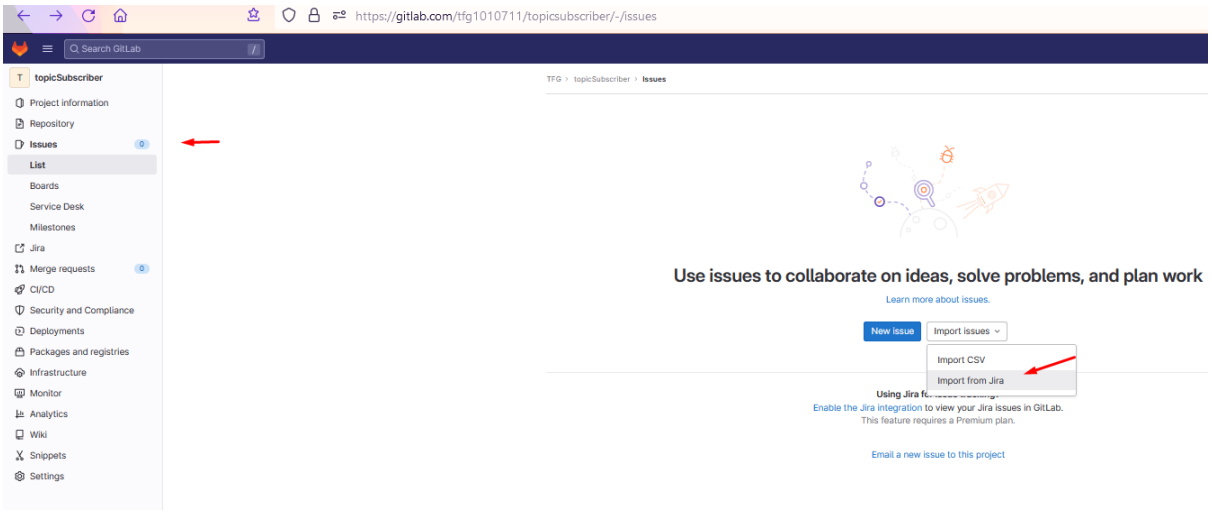


Por último, se introduce la dirección web del proyecto Jira, el usuario de Jira y el token.

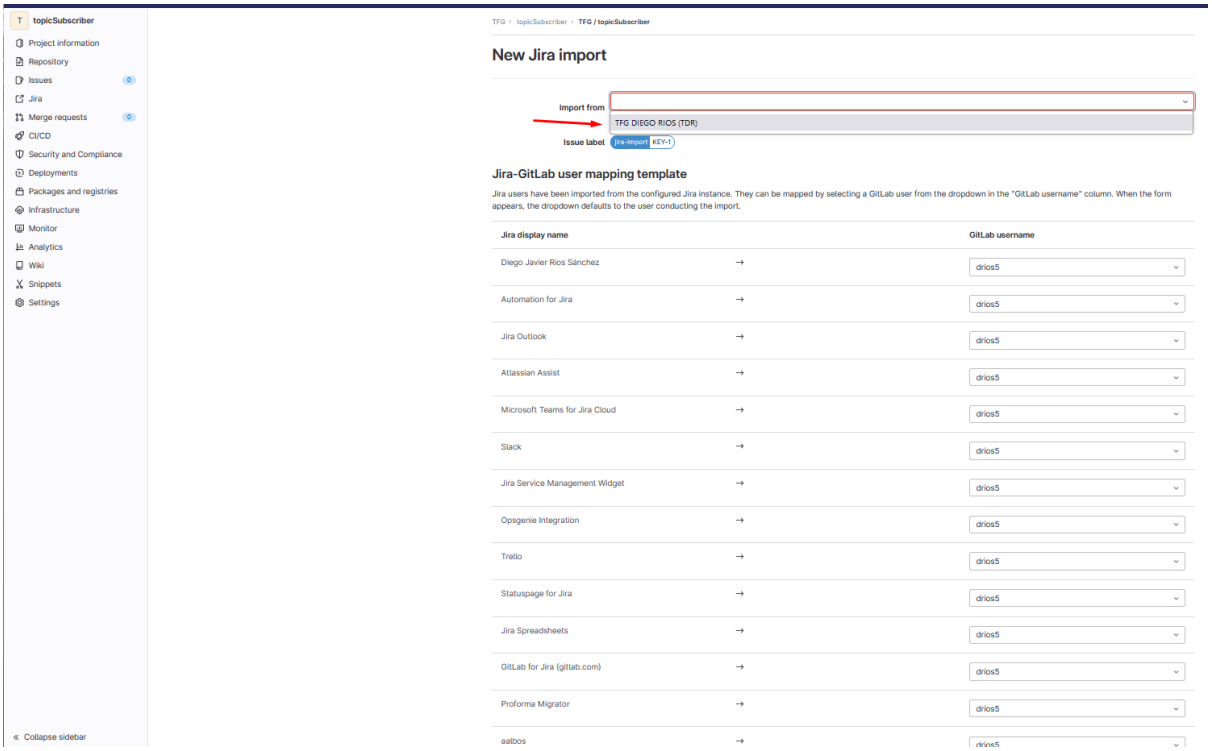
Se prueba la configuración con test settings y se guarda.



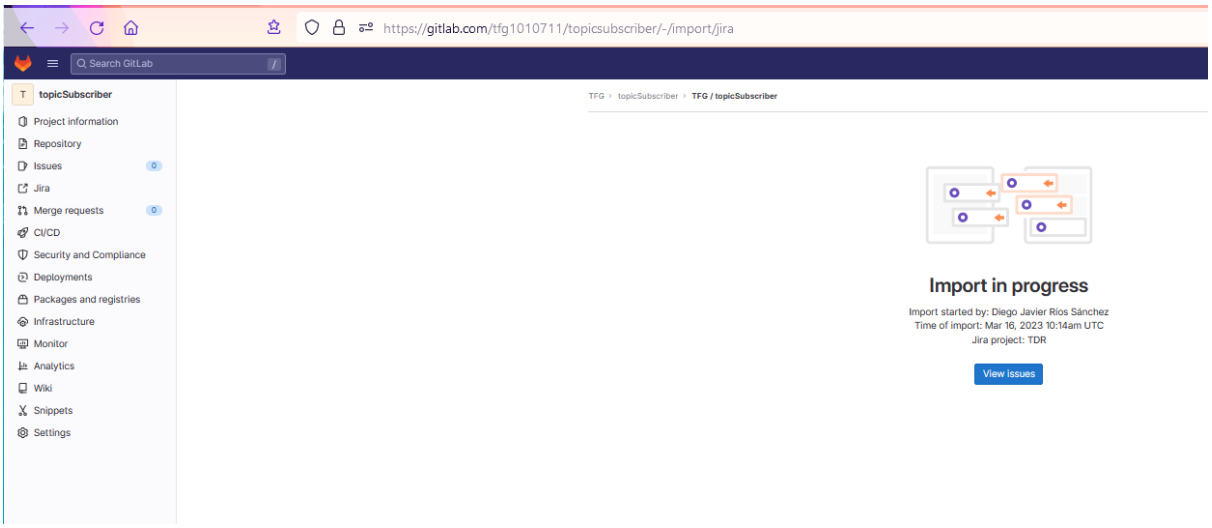
Ahora desde la opción de menú de issues se selecciona la opción importar issues from Jira



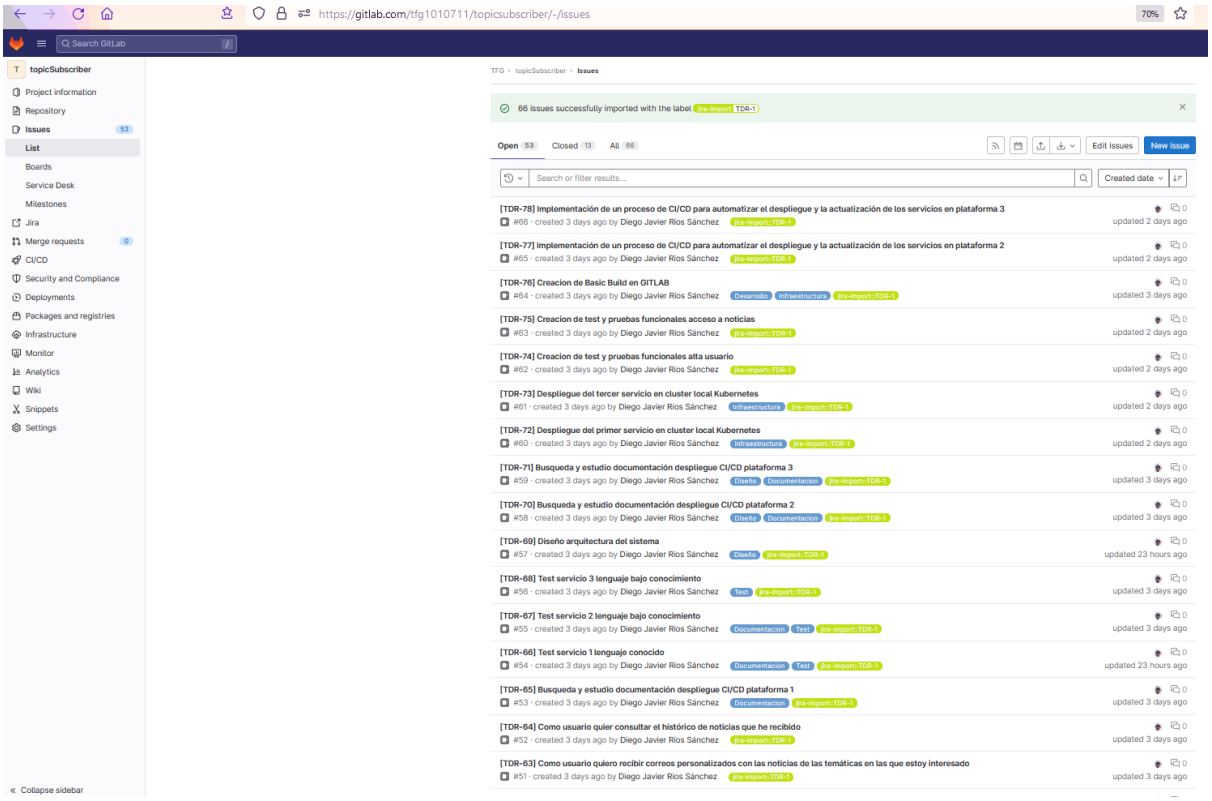
Se selecciona el proyecto.



A continuación las incidencias comenzarán a importarse.

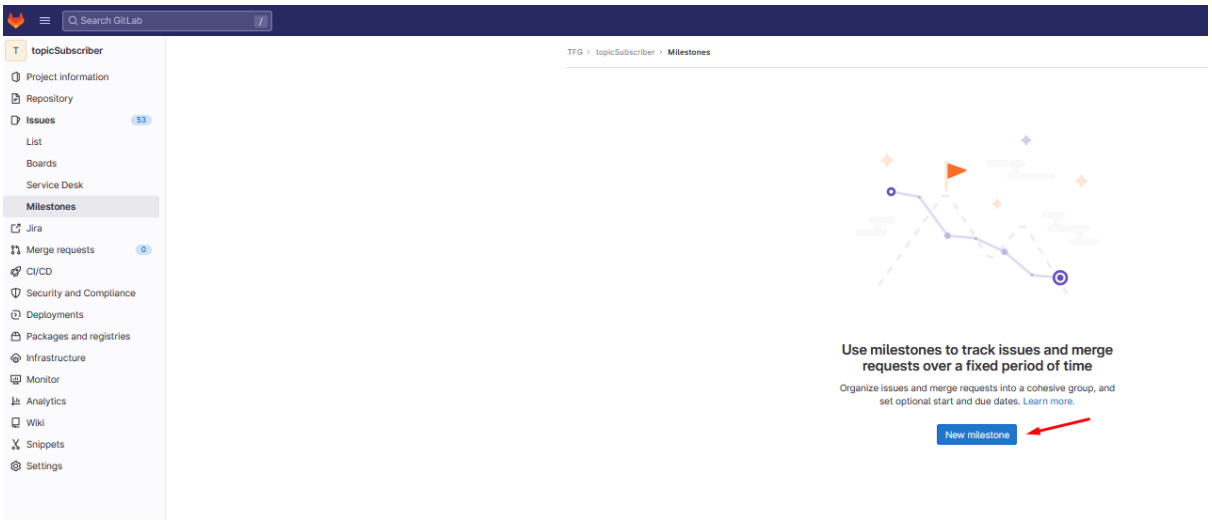


Finalmente, se puede comprobar que las incidencias han sido importadas

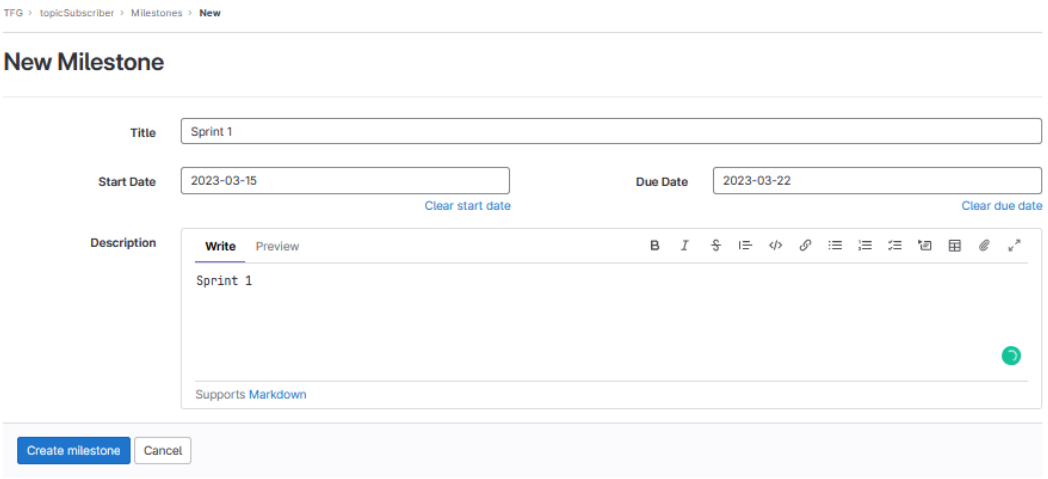


Creación de Milestone para agrupar los Sprints del proyecto

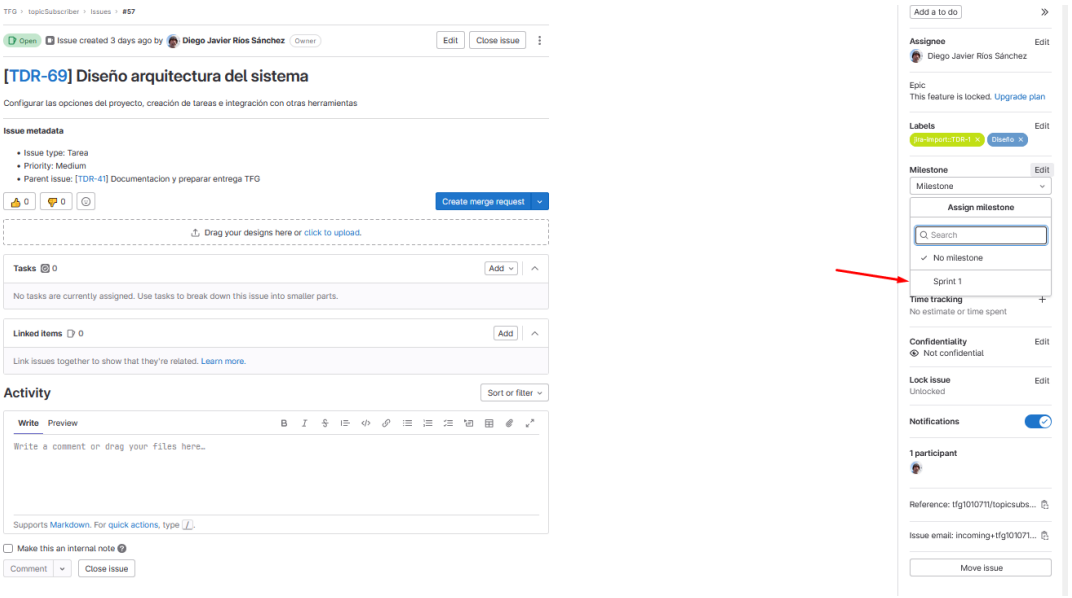
El siguiente paso consiste en crear un milestone para agrupar las incidencias y poder realizar seguimiento del proyecto.



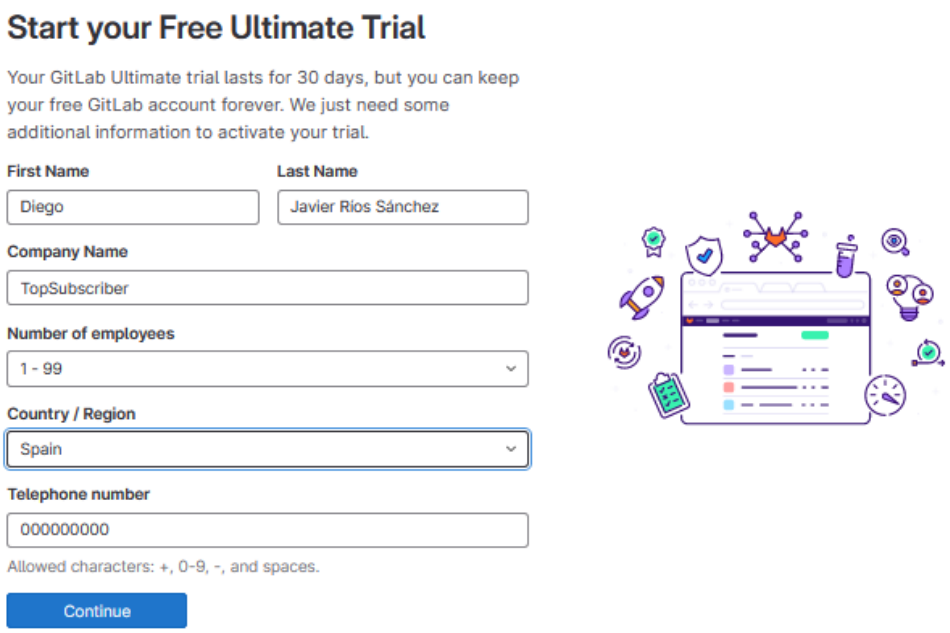
Para ello, se crea el milestone indicando el Sprint al que pertenece y el intervalo temporal del Sprint



Por último, se asignan las incidencias al milestone creado.



Para acceder a la funcionalidad de Burndown Charts se puede activar el Free Ultimate trial.



Ahora ya está activado el BurnDown Chart.

Anexo VIII Kubernetes

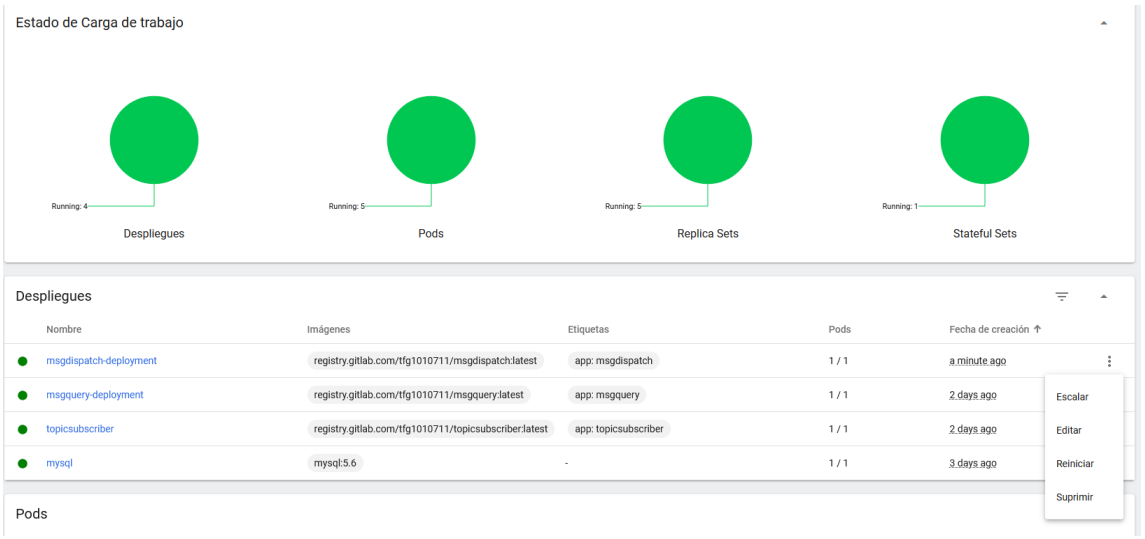
La creación del clúster de Kubernetes se realiza originalmente mediante la activación de esta característica en Docker Desktop para Windows. Sin embargo, en este caso, se opta por crear un clúster utilizando Kind (Kubernetes in Docker), ya que Docker Desktop presentó algunos problemas de estabilidad.

Kind permite tener un clúster de Kubernetes localizado en nuestra máquina para realizar pruebas y desarrollo. Esta decisión se toma con el objetivo de simplificar y agilizar el proceso de configuración del clúster.

Kubernetes utiliza archivos declarativos para definir y configurar el estado deseado de los recursos, la infraestructura y las aplicaciones en el clúster, en este caso en formato YAML.

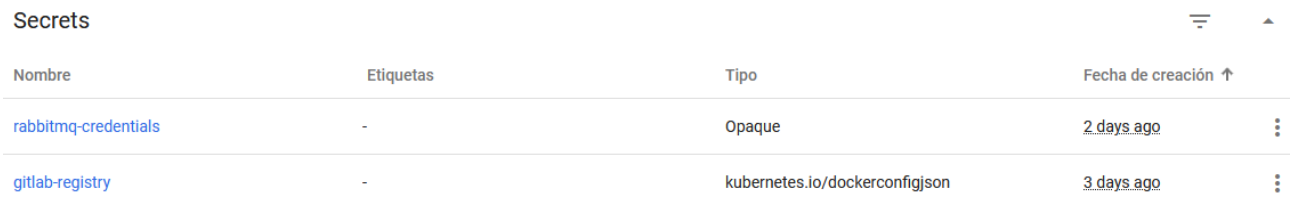
Pods

Un Pod en Kubernetes es la unidad más básica de implementación y consta de uno o más contenedores que se ejecutan juntos y comparten el mismo contexto y recursos. Los contenedores dentro de un Pod comparten la misma dirección IP, los mismos puertos de red y pueden comunicarse entre sí a través de la interfaz de loopback. En la definición de este sistema se desea que cada Pod funcione de forma operativa independiente.



Secrets

Se utilizan para guardar las credenciales de gitlab y de rabbitmq



Para crear el secret de GitLab se usa el access token con permiso de lectura sobre el registro de contenedores que se creó en GitLab.

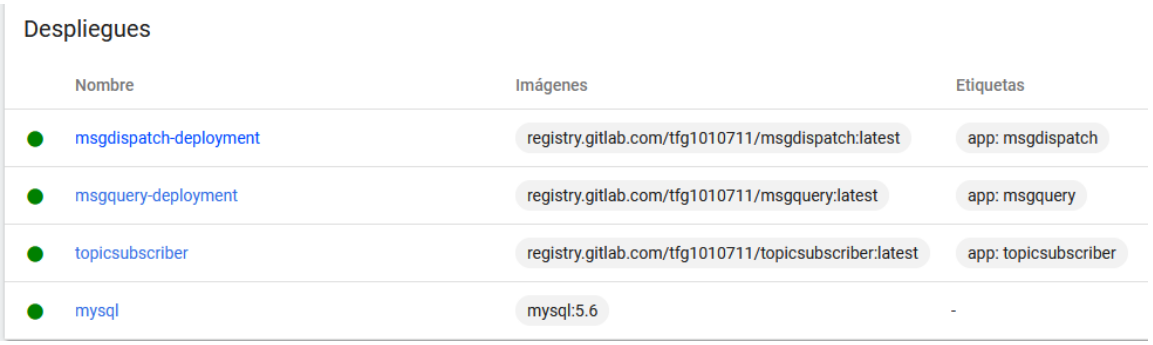
```
kubectl create secret docker-registry gitlab-registry
--docker-server=registry.gitlab.com --docker-username=@drios5
--docker-password=<TOKEN> --docker-email=drios5@uoc.edu
```

También se puede aplicar un fichero secret en formato declarativo. Se puede crear los secretos de esta forma codificándolos en base64.

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-credentials
type: Opaque
data:
  RABBITMQ_DEFAULT_USER: dXNlcm5hbWU=
  RABBITMQ_DEFAULT_PASS: cGFzc3dvcmQ=
```

Deployment

Un archivo típico de deployment presenta el siguiente formato indicando básicamente su nombre, sus puertos internos, selectores para su identificación, la imagen con la que trabaja y variables de entorno, en este caso asociadas a los secretos que permiten su conexión con Gitlab y RabbitMQ.



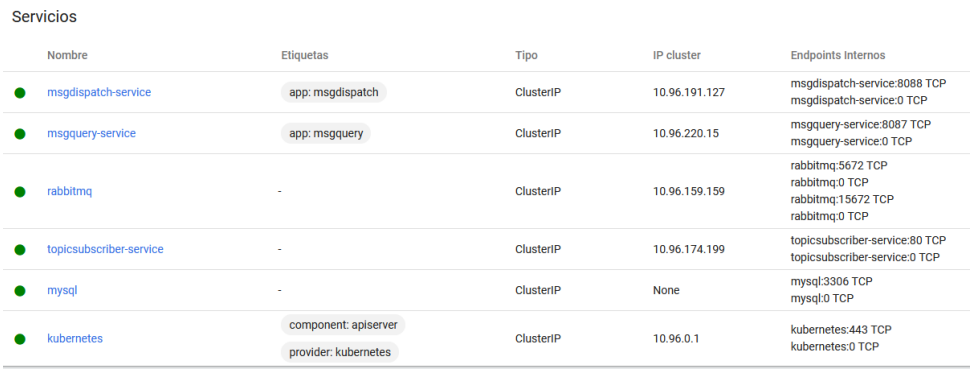
El formato de un archivo de deployment se podría declarar de la siguiente manera

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: msgdispatch-deployment
  labels:
      app: msgdispatch
```

```
spec:
  replicas: 1
  selector:
       matchLabels:
       app: msgdispatch
  template:
       metadata:
       labels:
       app: msgdispatch
       spec:
       containers:
       - name: msgdispatch
       image: registry.gitlab.com/tfg1010711/msgdispatch:latest
       imagePullPolicy: Always
       ports:
       - containerPort: 8088
       env:
       - name: RABBIT_HOST
              value: rabbitmq.default.svc.cluster.local
       - name: RABBIT_PORT
              value: "5672"
       - name: RABBITMQ_DEFAULT_USER
              valueFrom:
              secretKeyRef:
              name: rabbitmq-credentials
              key: RABBITMQ_DEFAULT_USER
       - name: RABBITMQ_DEFAULT_PASS
              valueFrom:
              secretKeyRef:
              name: rabbitmq-credentials
              key: RABBITMQ_DEFAULT_PASS
       imagePullSecrets:
       - name: gitlab-registry
```

Services

En este proyecto se utilizan los services para descubrimiento automático, permitiendo el escalado o el remplazo mientras se mantiene un nombre y una IP constante, para la intercomunicación de los componentes y para exponer los servicios fuera del clúster. En la configuración local se usa un tipo ClusterIP para a continuación exponerlos mediante *portforwarding*, aunque una configuración típica en la nube sería de tipo Loadbalancer.



Un archivo service muestra la siguiente estructura.

```
apiVersion: v1
kind: Service
metadata:
  name: msgdispatch-service
  labels:
```

```
      app: msgdispatch
spec:
  selector:
      app: msgdispatch
  ports:
      - name: http
      port: 8088
      protocol: TCP
      targetPort: 8088
  type: ClusterIP
```

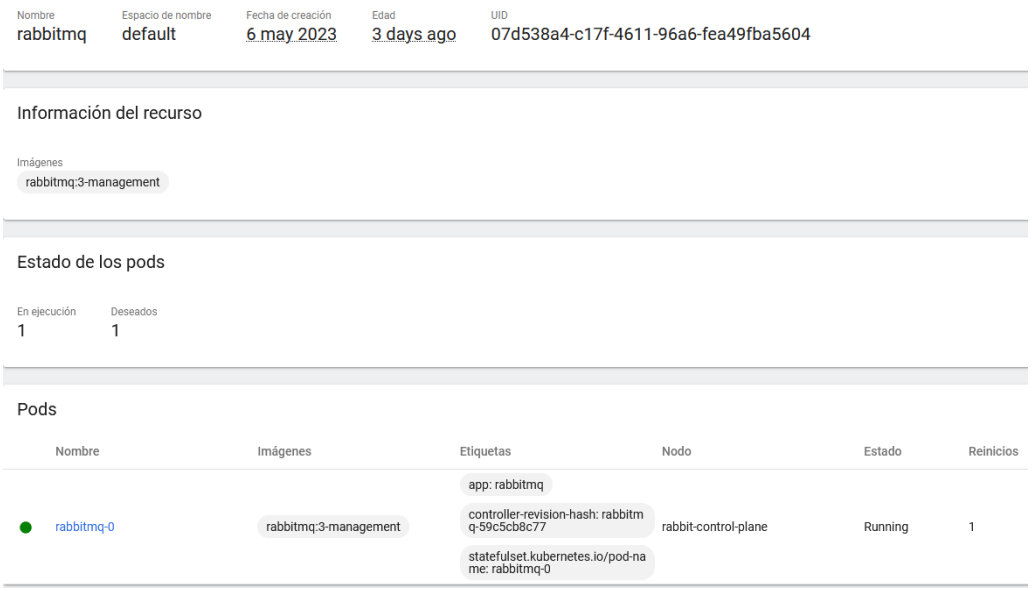
StatefulSet y LocalStorage

Para montar RabbitMQ se establece una configuración de tipo StatefulSet. Al utilizar esta configuración, se garantiza la alta disponibilidad y escalabilidad del servicio. Un StatefulSet es un controlador de Kubernetes que se utiliza para gestionar aplicaciones con estado, como RabbitMQ.

Proporciona una identidad única y persistente para cada instancia del servicio, lo que permite el almacenamiento persistente de los datos y facilita la replicación y escalado de las instancias. Para más información, consultar el anexo asociado de RabbitMQ

Además, se utiliza el proveedor local storage, que asegura que los datos de RabbitMQ se almacenen de forma persistente en el entorno local de Kubernetes. Esto es importante para garantizar la integridad y disponibilidad de los datos, incluso en caso de fallos o reinicios de los nodos del clúster.

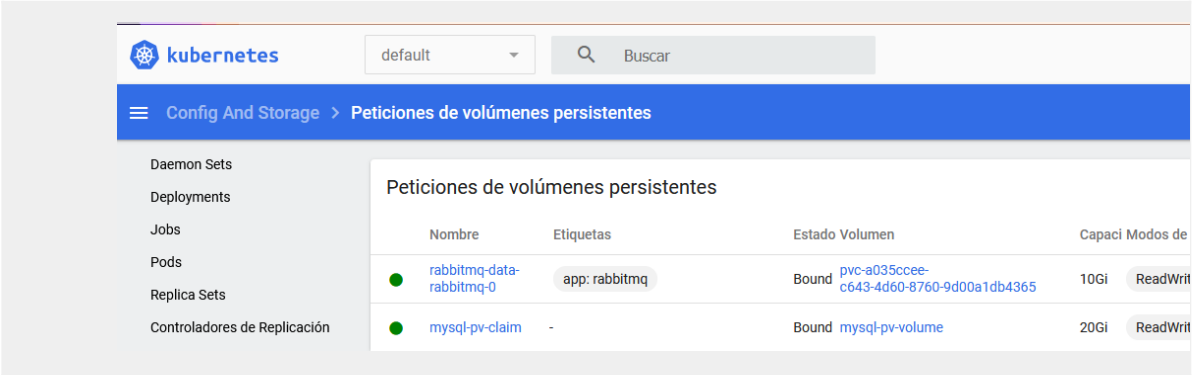
Estos archivos de configuración básicos nos permiten desplegar una aplicación y exponer nuestros pods a través de servicios en el clúster de Kubernetes.



Persistent Volume y Persistent Volume Claims

En este proyecto, se utiliza Persistent Volume y Persistent Volume Claims para brindar persistencia a RabbitMQ. De esta manera, el Persistent Volume Claim actúa como una solicitud de

almacenamiento en nombre de un Pod y proporciona una abstracción de alto nivel para solicitar y asignar recursos de almacenamiento persistentes dentro del clúster de Kubernetes. Cuando un Pod requiere almacenamiento persistente, se crea un PVC que describe los requisitos del almacenamiento, como el tamaño, el modo de acceso y otras opciones específicas. A continuación, Kubernetes se encarga de encontrar un Persistent Volume (PV) que cumpla con esos requisitos.



Desplegar la interfaz de usuario del panel de control de Kubernetes localmente

Para proteger los datos del clúster, el panel de control de Kubernetes se despliega con una configuración mínima de RBAC de forma predeterminada. Actualmente, el panel de control solo admite iniciar sesión con un Bearer Token. Para acceder al panel se siguen las instrucciones de la documentación de Kubernetes. Consistente en la creación de un usuario y un token que permite el acceso al panel

Una vez credo el usuario, se puede crear el token que se debe renovar cada vez que se acaba la sesión. Para crear el token se utiliza la siguiente instrucción con powershell de forma que el token queda guardado en el buffer.

```
kubectl -n kubernetes-dashboard create token admin-user | Set-Clipboard
```

Como resultado se obtiene un token de acceso que concede autorización para acceder al panel con un formato similar al siguiente.

```
eyJhbGciOiJSUzI1NiIsImtpZCI6Ijlvc21oWVN5bk1nd0dyd3drdzhicHBPNXNVTVVkVXg4VHVqNk5oWDhsTG8ifQ.eyJ
hdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjgzMzc0NzMwL
CJpYXQiOjE2ODMzNzExMzAsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiw
ia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsInNlcnZpY2VhY2NvdW50Ijp7I
m5hbWUiOiJhZG1pbi11c2VyIiwidWlkIjoiMWQzNDkyMWItZDdmYS00MGFjLWFkMjAtNTg0ZmEwYzE4OWM5In19LCJuYmY
iOjE2ODMzNzExMzAsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlcm5ldGVzLWRhc2hib2FyZDphZG1pbi11c
2VyIn0.XBv_ja-5HHcftlAQJn0K4PkO3EVGrsoI96TYGUFWy6SU07Pqyf9JDT6nzuRGX3sCtZ2N136DR3zry6UYwZjtD8a
GkYMoP14YlAmJe9Dv1GWwog7K9ojuvLPP_rl22n6vj8dcYxcehbba2xiH8w5QdgBgWlSdR1N6ayQiEoziua0njQ-wyfp-2
Gmuuc4I3IxhxGdZqJfBKtcvhl-vaL7U9xppNupi9IGbSvWJnIZ-Ow2neR7YMXr1OzOj4yZ8MqupgYng1c5KOCCjcHkKRgq
vAnlSA4yj-UTKPe2VCHDnitGUdY0HgN2Aw3ckBdkOuF_zAPX4r60HYJQiXwEFw077nA
```

Por último el acceso al panel de control se habilita utilizando la herramienta de línea de comandos kubectl, ejecutando el siguiente comando:

```
kubectl proxy
```

De esta forma se puede acceder al panel desde la siguiente dirección:

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/.

Crear un clúster GKE de Google

Para activar el clúster de kubernetes se puede seguir la guía que proporciona gitlab y que se puede leer aquí. En el presente proyecto para la creación del clúster y siguiendo la guía se utiliza Terraform.

Se solicitan dos prerrequisitos

- Una cuenta de servicio con Google Google Cloud Platform (GCP) service account.
- Y un runner you para poder ejecutar el pipeline de GitLab CI/CD.

En primer lugar y como prerrequisito se crea una cuenta. Se utilará uno de los runners de Gitlab. Siguiendo la con la documentación de GitLab, se ejecutan los siguientes pasos

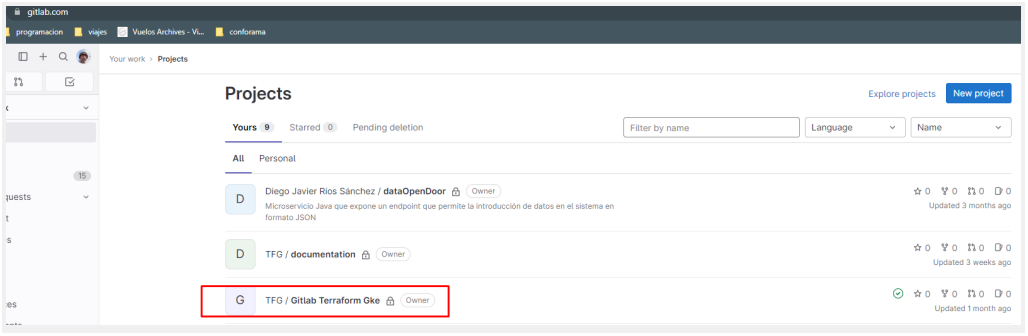
Pasos:

1. Importar el proyecto de ejemplo
2. Registrar el agente para Kubernetes
3. Crear las credenciales para GCP
4. Configurar el proyecto
5. Provisionar el clúster

Paso 1. Importar el proyecto Terraform de ejemplo

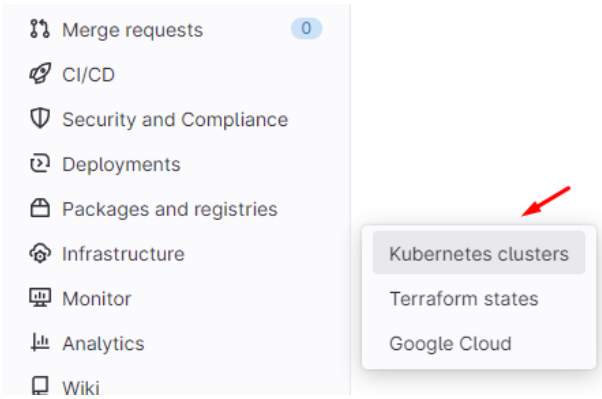
Se efectúa la creación de un proyecto nuevo y como opción importamos el ejemplo que proporciona GitLab. Después, mediante variables se pueden configurar los requisitos necesarios para que el proyecto quede operativo según nuestros requisitos.

```
https://gitlab.com/gitlab-org/configure/examples/gitlab-terraform-gke.git
```

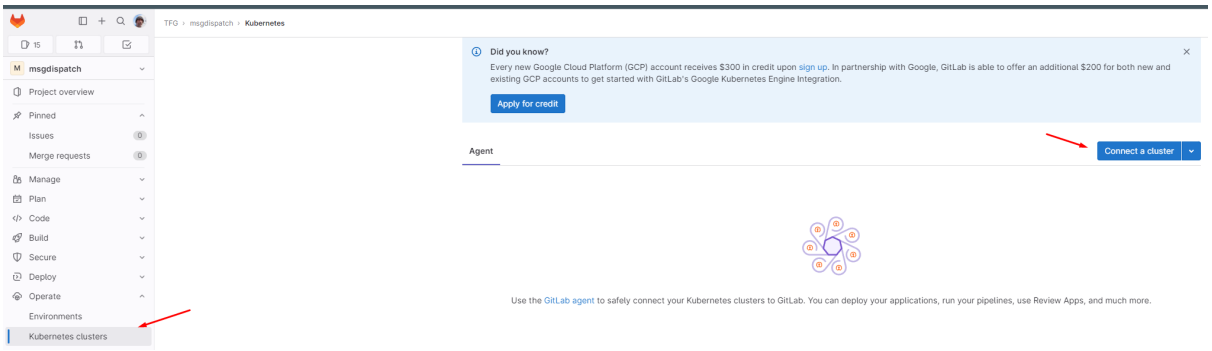


Paso 2. Creación del agente para Kubernetes

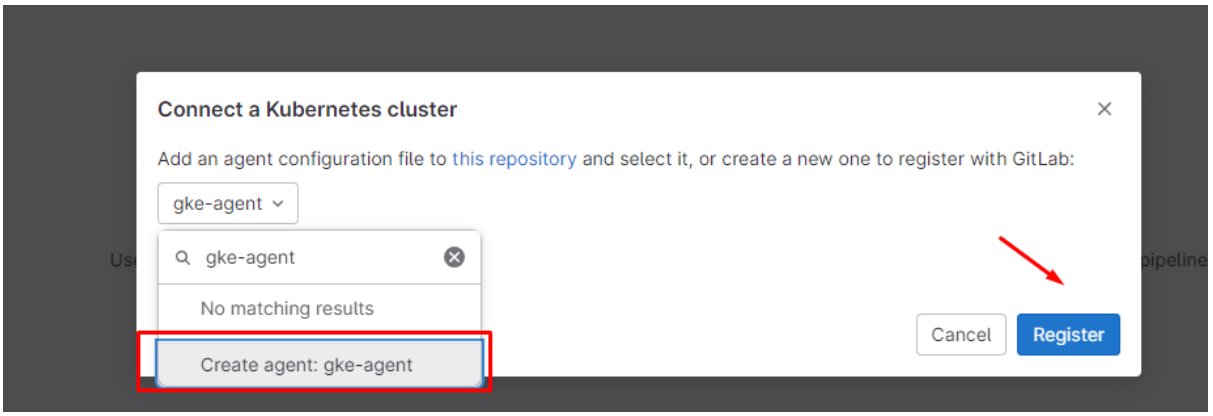
En primer lugar, se abre el proyecto que se quiere enlazar con Kubernetes en Google Cloud y en la barra izquierda, se selecciona **Infrastructure > Kubernetes clusters**.



En la nueva versión de GitLab la opción de menú es **Operate> Kubernetes clusters**.



A continuación, se pulsa el botón ***Connect a cluster*** y en el desplegable de selección se pone como nombre `gke-agent` lo que mostrará en la parte inferior el texto ***Create agent gke-agent***. Finalmente se pulsa la opción de ***Register***.



GitLab genera un token de registro para el agente. Se debe almacenar de forma segura este token secreto, ya que se necesitará más adelante. En pantalla el sistema muestra los datos que se deben guardar

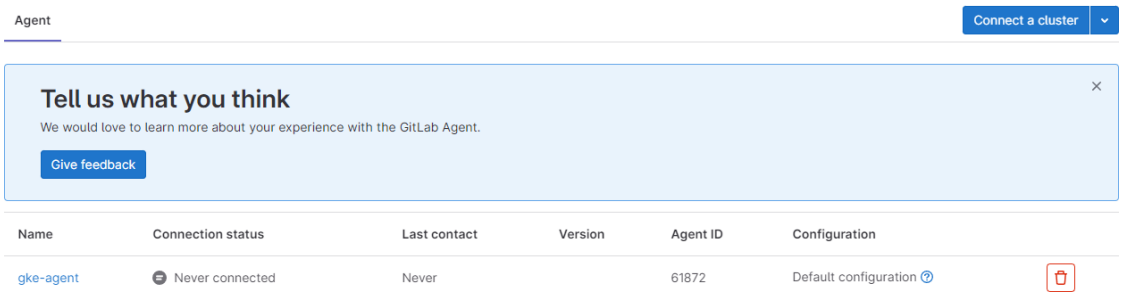
Agent access token

```
Ush8V6Jxu8Zr36x2Q8kWTnBnW6LqvTX3RMxZxxyPWo_MMYWsuw
```

Instalación recomendada usando Helm

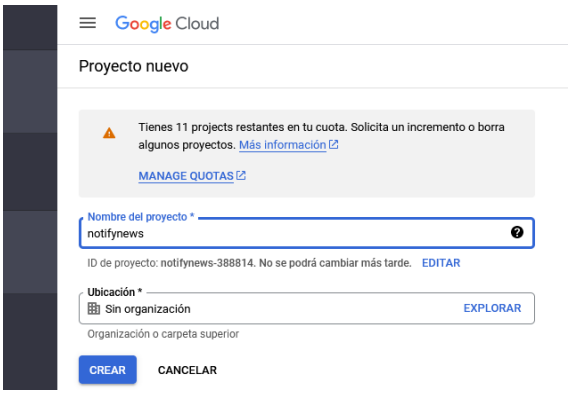
Este paso no es necesario pues la automatización se encarga de preparar todo el contexto necesario para la ejecución efectiva del cluster.

Se toma nota de los datos. Al cerrar la pantalla se muestra el agente creado.



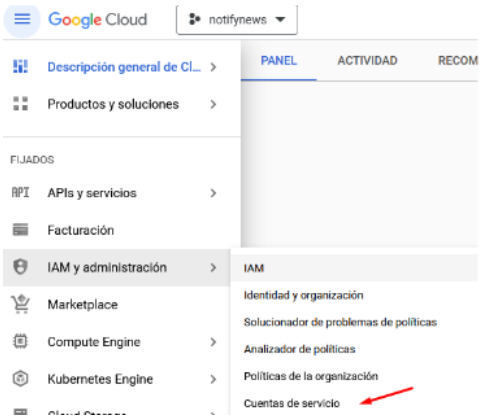
Paso 3. Crear las credenciales para GCP

Como requisito previo, se debe tener una cuenta creada en Google Cloud Platform (GCP), en la cual se haya creado un proyecto.

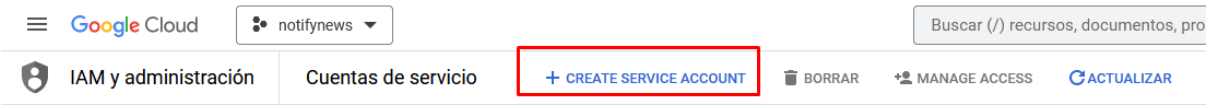


Una vez creado el proyecto se activa una cuenta de servicio siguiendo los siguientes pasos.

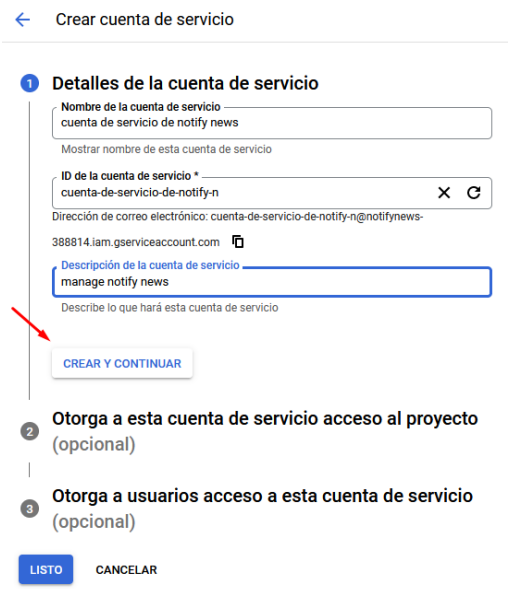
En el menú de IAM y administración se selecciona cuentas de usuario.



En la pantalla siguiente se selecciona crear cuenta de servicio.



Ahora se siguen los pasos del asistente. Se da un nombre al servicio que asignará de forma automática un id al servicio. Finalmente, se agrega una descripción y se pasa al siguiente paso del asistente pulsando en “crear y continuar”.



Ahora, Para autenticar GCP con GitLab, se deben activar los siguientes roles:

- Visualizador de la red de compute, *Compute Network Viewer*
- Administrador de Kubernetes Engine, *Kubernetes Engine Admin*
- Usuario de cuenta de servicio, *Service Account User*

Cuando se está realizando despliegues utilizando Terraform para crear recursos, se pueden agregar los siguientes roles para facilitar esta tarea

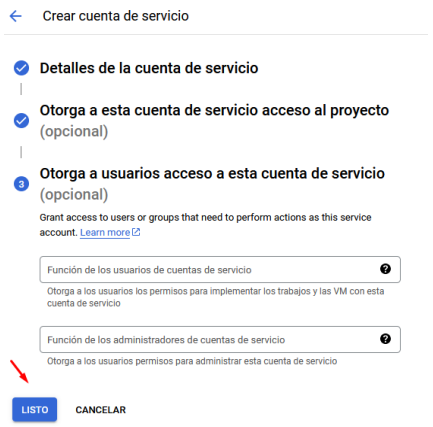
- Administrador de red de Compute Engine *Compute network admin*
- Administrador de cuenta de servicios *Service Account Admin*
- Administrador de IAM de proyecto *Project IAM Admin*

De esta manera, utilizando Infrastructure as Code (IAC) con Terraform, se pueden agregar los recursos necesarios de manera más eficiente y controlada.

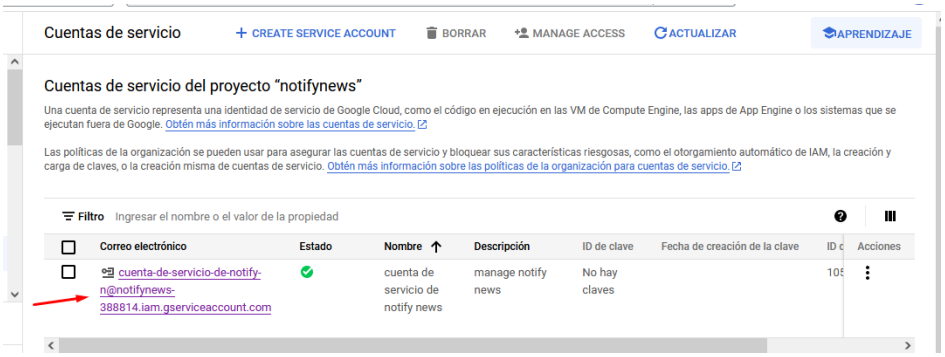
Se requieren tanto cuentas de servicio de usuario como de administrador. El rol de usuario reemplaza a la cuenta de servicio predeterminada al crear el grupo de nodos. Por otro lado, el rol de administrador crea una cuenta de servicio en el espacio de nombres kube-system.

La cuenta de servicio de usuario se utiliza para otorgar permisos específicos a un usuario o conjunto de usuarios, permitiéndoles acceder y gestionar ciertos recursos dentro del entorno. La cuenta de servicio del administrador, por otro lado, se utiliza para tareas administrativas más amplias, como la creación de cuentas de servicio adicionales y la configuración del entorno en el espacio de nombres kube-system

Finalizado este proceso damos a continuar. En el último paso, simplemente se pulsa el botón listo.

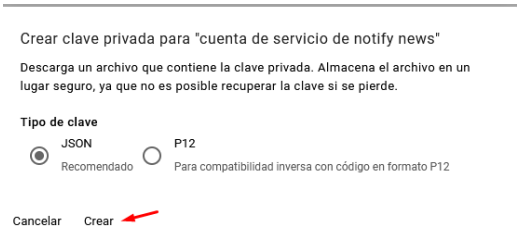


Como resultado se ha creado la cuenta de servicio. Ahora se debe crear una clave.

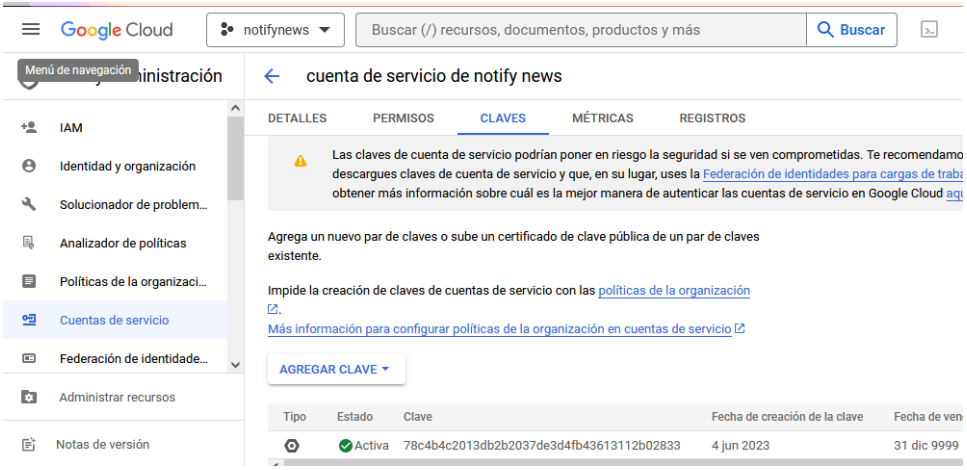


Se pulsa sobre la cuenta. Se selecciona la opción de claves y se procede a agregar una clave nueva.

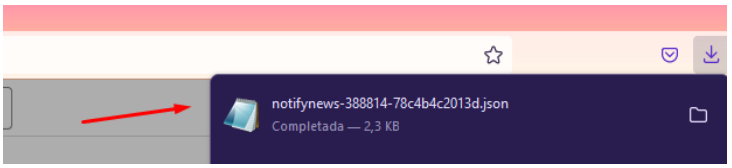
Se crea la clave recomendada tipo JSON.



Como resultado, el sistema crea una clave



y procede a descargarla.



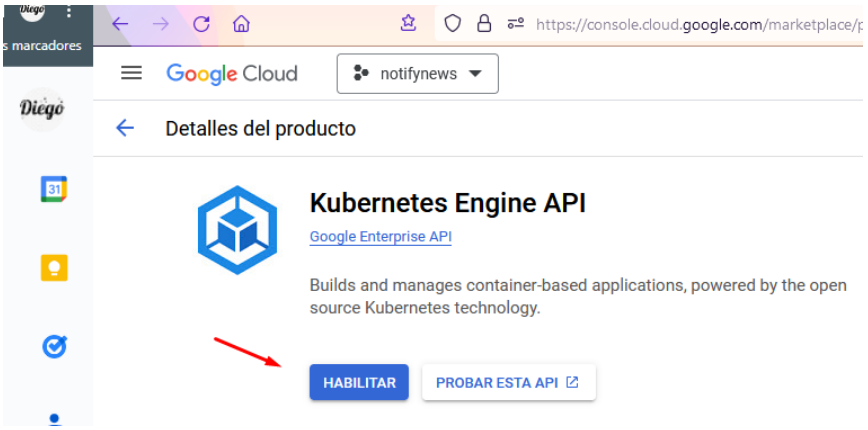
Ahora, se procede a codificar el archivo JSON a base64 redireccionando la salida a un archivo. Se renombra el archivo original para que sea más manejable.

En este caso se usa Git CMD para realizar la conversión.

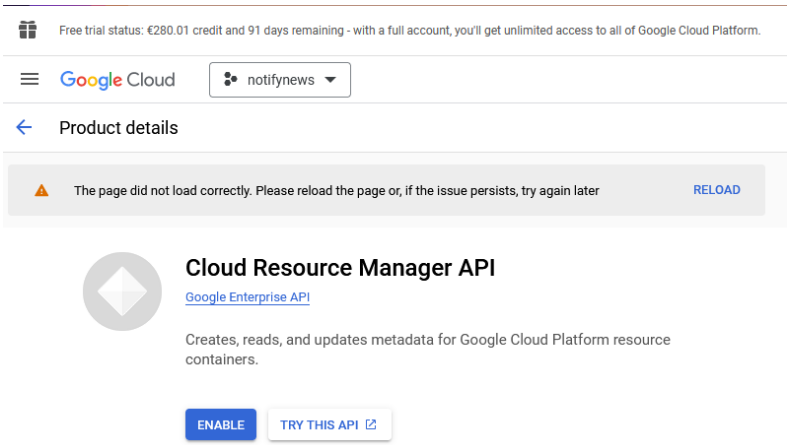
```
"C:\Program Files\Git\bin\bash.exe" --login -i

base64 sa-key.json | tr -d '\r\n' > sa-key.b64
```

La salida de este comando se utilizará como la variable de entorno en GitLab **BASE64_GOOGLE_CREDENTIALS** en el siguiente paso.

Por último se activa el motor **API de Kubernetes**.



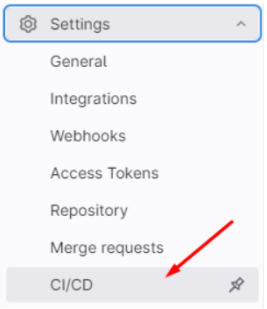
Para poder desplegar con Terraform desde GitLab también se debe activar el recurso **Cloud Resource Manager API**



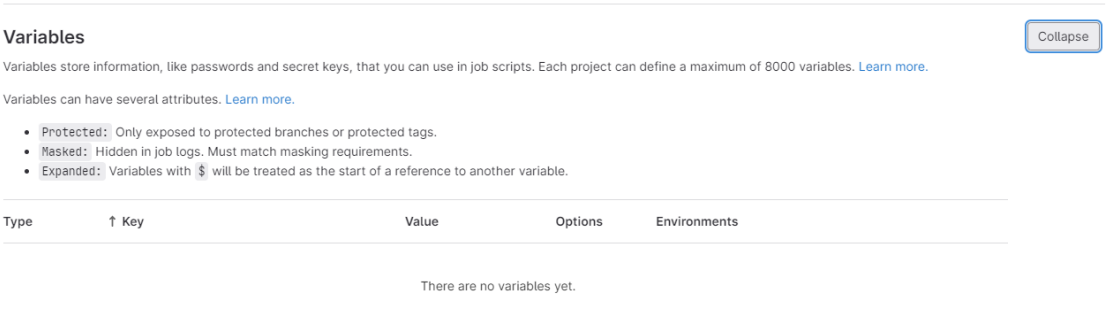
Paso 4. Configuración del proyecto

Para configurar el proyecto con Terraform, se utilizan las variables de entorno de CI/CD.

En la barra de la izquierda se selecciona ***Settings > CI/CD***.



Se abre el menú de **Variables** y se procede a agregar la configuración



1. Se crea la variable **BASE64_GOOGLE_CREDENTIALS** con la salida del archivo JSON codificado que se ha creado.
2. Se crea la variable **TF_VAR_gcp_project** con el nombre del proyecto en.
3. Se establece la variable **TF_VAR_agent_token** con el valor del token generado por GitLab en el paso 2.
4. Se establece el valor de la variable **TF_VAR_kas_address** al valor del agente que se generó en el paso 2.

```
helm repo add gitlab https://charts.gitlab.io
helm repo update
helm upgrade --install gke-agent gitlab/gitlab-agent \
    --namespace gitlab-agent-gke-agent \
    --create-namespace \
    --set image.tag=v16.1.1 \
    --set config.token=tfKpnQT29zxQqyodyxJJ3GpxET-aUbrVhzj5YZZjtJGauYxGSA \
    --set config.kasAddress=wss://kas.gitlab.com
```

Configuración Opcional:

El fichero variables.tf contiene otras variables que también pueden ser modificadas desde la configuración de variables:

- **TF_VAR_gcp_region**: Se establece a una región cercana, europe-west1.
- **TF_VAR_cluster_name**: El nombre de clúster: clusternotifynews
- **TF_VAR_cluster_description**: Se establece como valor el recomendado **$CI_PROJECT_URL** para crear una referencia entre el proyecto GitLab y la GCP. De esta forma se sabe que proyecto es el responsable de provisionar el clúster que se ve en el panel de control de GCP
- **TF_VAR_machine_type**: Se usa una máquina n1-standard-1.
- **TF_VAR_node_count**: Se estable el número de nodos a 1.
- **TF_VAR_agent_namespace**: Establece el *namespace* de Kubernetes namespace para el agente the GitLab. No se pone ningún valor.

```
BASE64_GOOGLE_CREDENTIALS                ewogICJ0eXBlIjogInNlcnZpY2VfYWNjb3VudCI
                                         sCiAgInByb2plY3RfaWQiOiAibm90aWZ5bmV3cy
                                         0zODg4MTQiLAogICJwcml2YXRlX2tleV9pZCI6I
                                         CI3OGM0YjRjMjAxM2RiMmIyMDM3ZGUzZDRmYjQz
                                         NjEzMTEyYjAyODMzIiwKICAicHJpdmF0ZV9rZXk
                                         iOiAiLS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS
                                         0tXG5NSUlFdkFJQkFEQU5CZ2txaGtpRzl3MEJBU
                                         UVGQUFTQ0JLWXdnZ1NpQWdFQUFvSUJBUUN0Sk56
                                         TDkwOFFFMnRsXG45S095WUE1WjFsdDNrZ1dWTmF
                                         RM0dsT3IxNys3U0FNVnEzS3MyMmZKQVV5UUxFVD
                                         Z5K0JxYUFLejhWOGZiaWVjXG5FWXdzWVBIdUFZS
                                         3RqdE9Id3F3Zk1DVkZiUUx0bTVTcHhLa1d0aWhj
                                         Mk55b0FIK2VaV1RmY1JvSU56VWxGUE9zXG51MzB
                                         TUFZVUGdBUFRsVmpFdjl4V290Y3EwcGVzU3lIVV
                                         BycXcwa3k3ZnFuUndpcVFUS2ZjbzArTkxTcTQxd
                                         kFzXG5NclN2aXgxeTVUbVFqK1JKR1BjS2ZqMDRJ
                                         RDVHQnJZeU5LOC9CMms1UjhjTzZJUU40Skg2UGF
                                         BV3E5VHl4RFQ2XG5xTEZVUUdvOVJtMksyMDJUK1
                                         RucWJma1NlT3poWGFIWUltZUQxV1Bpd2ZLSi9DK
                                         zMyTHNVTG5HMThpcnBkK3pjXG52aURabmxFUEFn
                                         TUJBQUVDZ2dFQUZtQTBycUZUMU9LRmx5UUNBYnZ
                                         iTHUzK1hSb0FZYk9RZU5TTXFxRVR0S3JPXG5Sej
                                         hrYjBWZGlNTUR0NEFaSEFZOEJ0UlJRUnI0N1pGc
                                         FNnaVVnbW0yNmtqRXNCUk1ydmRRRWluQkwxTXl0
                                         UzFOXG5qWkVPUHhLa2JiZFM4TjA3b09jK25ZZTY
                                         wd1A3ZU1MWDZPTkQreGIxQnE5SEQxRjJLdHNXcV
                                         JXYlUvdnFRUWd3XG5MOHVLRnNaS3B0ZzZ6OURtN
                                         01OOGpnU2h4SU5ZZmpIMGd1U29aQ05aYWNKeTk3
                                         MDRSNFg3MjlaMlAzZ3lwSVQxXG4rOFhqT2dGOSt
                                         PbS9Mb0x5MTJESkpXR3g1Rjlvc0J6K1NaTFFiV2
```

	NOcWFOLzBnenFFU1hIYnR0Uktjc0svakhPXG5aO Hd4dkxlK1JvVFYrU3lnUStGeXhaMWdSRWkzeWRC a1BaZERWVUhUTFFLQmdRRGFSSHd6dmt1bjB6V3g 2WTdvXG5tNmtpVUJQMjRjaTZCTmhkNmJlYkhUT3 pPQnlzOS9RZ3RjVktQT2haTnErOUFGcGFZNWFJU EdCcGZ6TmxEWkZyXG5zSFd2Y1ZIa3ViQ1I4Rzcw YkJ1VjM2Z3N4cUhZZnBSeHlmNWhFK0szRUZXQ0x 6ZmxkNkp0Y2d4Y1NZakNBT1B6XG5tRDFwK1Nna2 VKZFIrVEd4TzMyVVBta3Rnd0tCZ1FETEUycis3M zNwT3JjWHJGeDJzZlp2OTZ6MWxYMTNuNmRnXG52 U0llNmFoTWFJVUlScEd5dDVNNm5UNkkzR0pIdXU 2c3MwQi9MdUd5M1dDSTNrSkdHaXo3S2dBa0wyL1 AwUmJXXG5tT0t0WlVrbXliNmphMEZXdDNpVnpye nk3dnZmVk16V2ZUb1FFNlVQa2Y3NWJ0cjd4Wm9l UjBsSlRqRDNpcUkxXG5sYys3aS92a2hRS0JnQ3d WeXhSVWFrbUNjb1QxektTVTFDdVhDd0ZCaDE1ek hoU1I0VURXUEJ3RnlGenpVLzdMXG5XQWJ2bUdwM GpqaFpldVpvMlAyNVhDdlN3bG0xek5wNGNMeTM0 YmVWL2VEc05DendMUDR2aDNOTHgyTEg1a201XG5 aOFRKMUcvdlA2WWt6V0J1c29MeWx5YUl2N09YcE hNT0c5RnN2cnZuRTRFK0dOaFlVTHdCZ3hMckFvR 0FQUkNpXG52UFJSUFYvZGpLM1V4QkJ4bFd1NXc4 Tm5EaUNoR1RMbVhydE9VMk9lbFVWN0xHMk5SVFB oaVJBVU1aT2hqNmI3XG5Qd05lOVFVaXNTVjhjaz RPdHB3RG5OM3kxbC8rYVJ0aUZsZjlyajgrSnkvc Vh1UWV5NVYyTmVydExGUDRvSXJaXG5lZG9lYVZj eWY5ajQybnFxQkc3Vk44amVhcUZ5YzRDUURjcGp QVzBDZ1lBM1Fla3NMeVVnSXIrVVphMUR1ZDd4XG 5ITDg1dG5FZ3hkR00wYkpUNDE4ZjlpeXJ3MHVJd W53M0VsOFg0Lzd2TjhHUlRTVGREVmJPMndRclBk T1pTMUE0XG5PekpwdHpsQ3R0bWpLVE1ISVFCUm9 jOGxHSXA2T2dQdmJ5YkhiTHNQb3hRaTJPcTVncD ZvTnQ4bS8vaXhLb0VRXG4xZlRrNW5Jci9iVUFwV k5vZU81cDZRPT1cbi0tLS0tRU5EIFBSSVZBVEUg S0VZLS0tLS1cbiIsCiAgImNsaWVudF9lbWFpbCI 6ICJjdWVudGEtZGUtc2VydmljaW8tZGUtbm90aW Z5LW5Abm90aWZ5bmV3cy0zODg4MTQuaWFtLmdzZ XJ2aWNlYWNjb3VudC5jb20iLAogICJjbGllbnRf aWQiOiAiMTA1NTg5ODcyMDU0Nzc2ODAyODU3Iiw KICAiYXV0aF91cmkiOiAiaHR0cHM6Ly9hY2NvdW 50cy5nb29nbGUuY29tL28vb2F1dGgyL2F1dGgiL AogICJ0b2tlbl91cmkiOiAiaHR0cHM6Ly9vYXV0 aDIuZ29vZ2xlYXBpcy5jb20vdG9rZW4iLAogICJ hdXRoX3Byb3ZpZGVyX3g1MDlfY2VydF91cmwiOi AiaHR0cHM6Ly93d3cuZ29vZ2xlYXBpcy5jb20vb 2F1dGgyL3YxL2NlcnRzIiwKICAiY2xpZW50X3g1 MDlfY2VydF91cmwiOiAiaHR0cHM6Ly93d3cuZ29 vZ2xlYXBpcy5jb20vcm9ib3QvdjEvbWV0YWRhdG EveDUwOS9jdWVudGEtZGUtc2VydmljaW8tZGUtb m90aWZ5LW4lNDBub3RpZnluZXdzLTM4ODgxNC5p YW0uZ3NlcnZpY2VhY2NvdW50LmNvbSIsCiAgInV uaXZlcnNlX2RvbWFpbiI6ICJnb29nbGVhcGlzLm NvbSIKfQo=
TF_VAR_agent_token	Ush8V6Jxu8Zr36x2Q8kWTnBnW6LqvTX3RMxZxxy PWo_MMYWsuw
TF_VAR_cluster_description	$CI_PROJECT_URL
TF_VAR_cluster_name	clusternotifynews

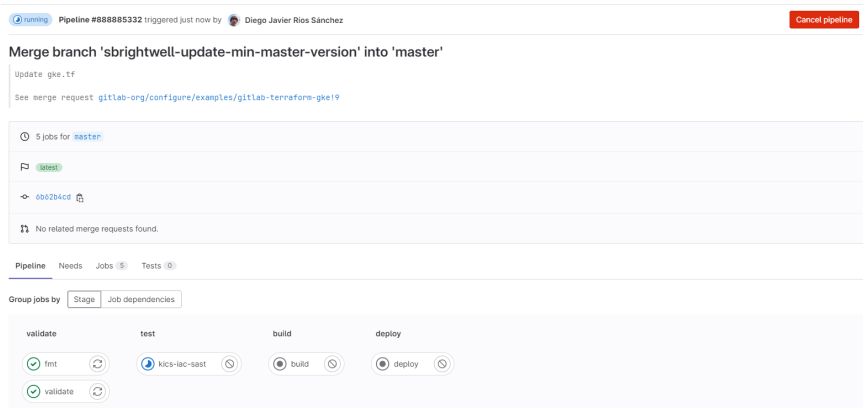
TF_VAR_gcp_project	notifynews-388814
TF_VAR_kas_address	wss://kas.gitlab.com
TF_VAR_machine_type	n1-standard-1
TF_VAR_gcp_region	europe-west1
TF_VAR_node_count	1

Paso 5. Provisionar el clúster

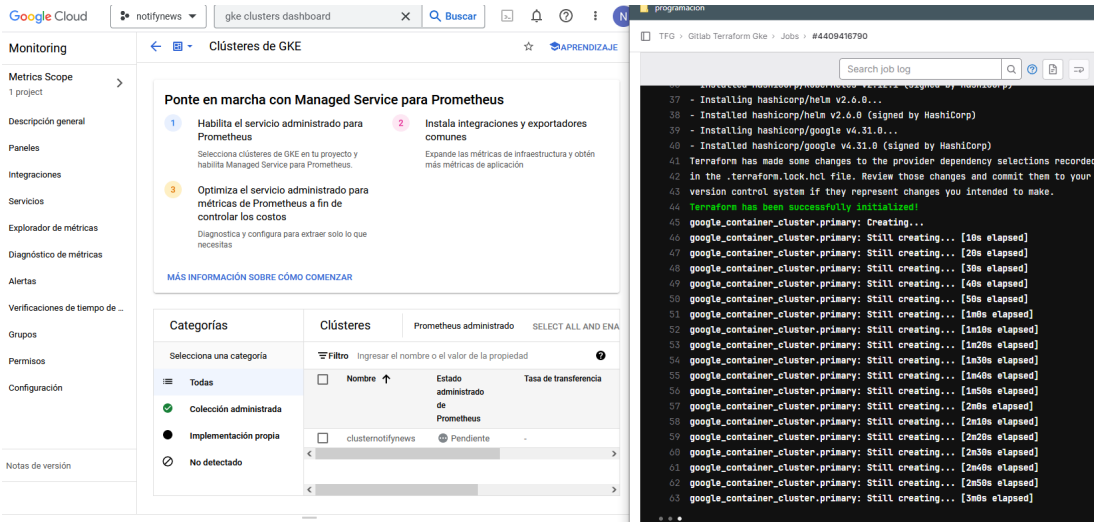
Después de configurar el proyecto, se debe ejecutar de forma manual la provisión del clúster.

1. En la barra izquierda, desde CI/CD > Pipelines. Para la nueva versión de Gitlab Build> Pipeline

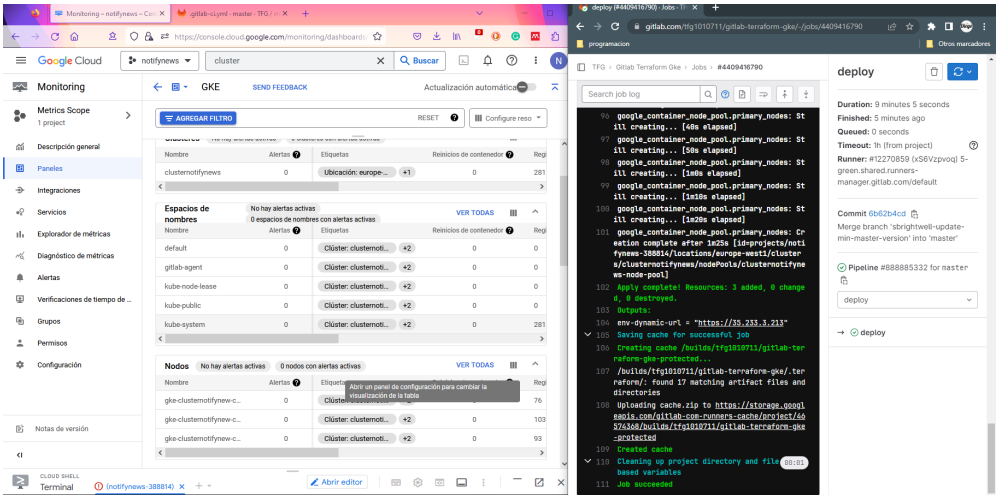
2. Ahora se ejecuta el pipeline manualmente.



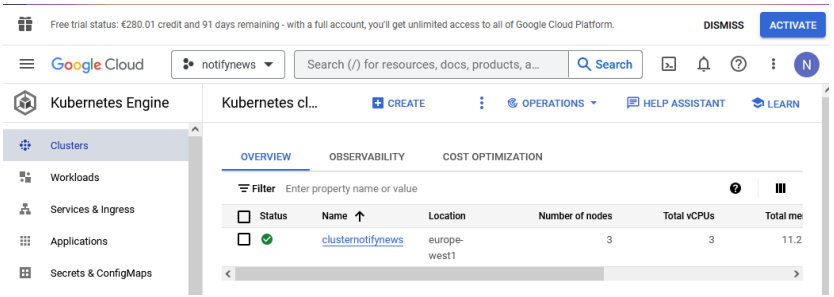
El último paso del pipeline también se debe ejecutar manualmente. El proceso tarda algo de tiempo.



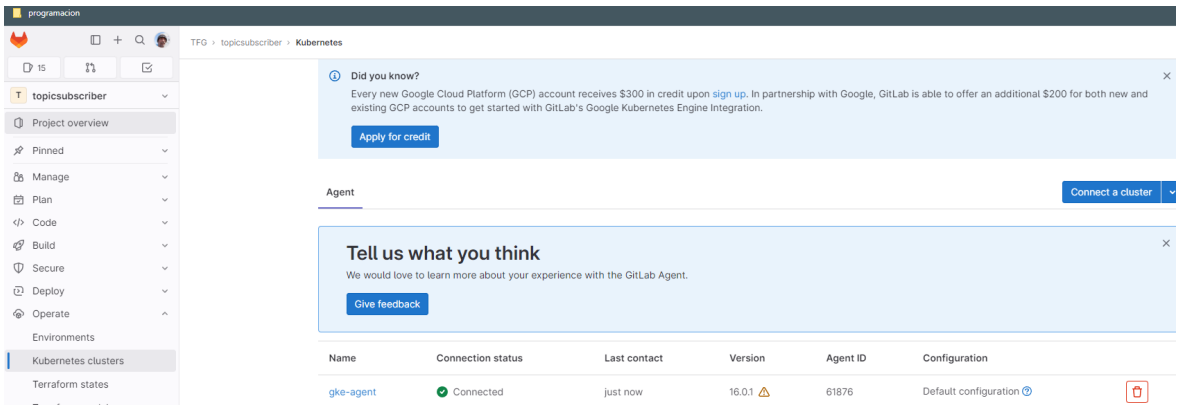
Cuando el pipeline finalice de forma satisfactoria se podrá ver el nuevo clúster desde la cuenta de Google.



1. En GCP: en GCP console's Kubernetes list.



2. En GitLab: en la barra lateral de proyecto,Infrastructure > Kubernetes clusters o bien Operate > Kubernetes clusters



Paso 6. Usar el clúster

Después de provisionar el clúster, este se conecta a GitLab y está preparado para los deployments. Para comprobar la conexión:

1. En el menú de la izquierda, seleccionar Infrastructure > Kubernetes clusters o bien Operate > Kubernetes clusters
2. En el listado, comprobar el valor de la columna “estado de la conexión”.

Creación Básica de objetos en Google Cloud

El proceso de CI/CD se utiliza para automatizar los despliegues de los microservicios de la aplicación. Sin embargo, algunos componentes, como los secrets de RabbitMQ o las configuraciones específicas del entorno, no se ajustan al flujo típico de CI/CD. Estos elementos no se integran directamente desde un proceso de CI/CD, por lo que requieren una ejecución manual. Para llevar a cabo esta tarea, se puede instalar el SDK de Google Cloud desde PowerShell y ejecutar el despliegue de estos objetos básicos. Otra opción posible es utilizar Cloud Shell, aunque esto implica cargar previamente los objetos locales en el servidor remoto

```
(New-Object
Net.WebClient).DownloadFile("https://dl.google.com/dl/cloudsdk/channels/rapid/GoogleClo
udSDKInstaller.exe", "$env:Temp\GoogleCloudSDKInstaller.exe")

& $env:Temp\GoogleCloudSDKInstaller.exe
```

El sistema solicita autenticación mediante el navegador. Se seleccionan las opciones por defecto porque se va a trabajar en la misma zona. Ahora se intenta conectar con el clúster.

En primer lugar se comprueba el listado de clústers disponibles

```
gcloud container clusters list
```

```
C:\Program Files (x86)\Google\Cloud SDK>gcloud container clusters list
NAME              LOCATION      MASTER_VERSION    MASTER_IP     MACHINE_TYPE   NODE_VERSION      NUM_NODES  STATUS
clusternotifynews  europe-west1  1.22.17-gke.8000  35.233.3.213  n1-standard-1  1.22.17-gke.8000  3          RECONCILING
C:\Program Files (x86)\Google\Cloud SDK>
```

Se instala el plugin de autenticación

```
gcloud components install gke-gcloud-auth-plugin
```

```
+-----------------------+---------+---------+
| gke-gcloud-auth-plugin |   0.5.3 | 7.8 MiB |
+-----------------------+---------+---------+

For the latest full release notes, please visit:
  https://cloud.google.com/sdk/release_notes

Do you want to continue (Y/n)?  Y

#============================================================#
#= Creating update staging area                             =#
#============================================================#
#= Installing: gke-gcloud-auth-plugin                       =#
#============================================================#
#= Installing: gke-gcloud-auth-plugin                       =#
#============================================================#
#= Creating backup and activating new installation          =#
#============================================================#

Performing post processing steps...done.

Update done!

Presione una tecla para continuar . . .
```

```
gcloud container clusters get-credentials clusternotifynews
```

Se crean los secrets manualmente. El primero, que guarda los datos de conexión con GitLab, se introduce desde línea de comandos.

```
kubectl create secret docker-registry gitlab-registry
--docker-server=registry.gitlab.com  --docker-username=@drios5
--docker-password=glpat-Tyhex3xJun8SMxSsmExj   --docker-email=drios5@uoc.edu
```

El segundo Secret, que contiene los datos de autenticación de RabbitMQ, se aplica mediante un apply de un archivo local.

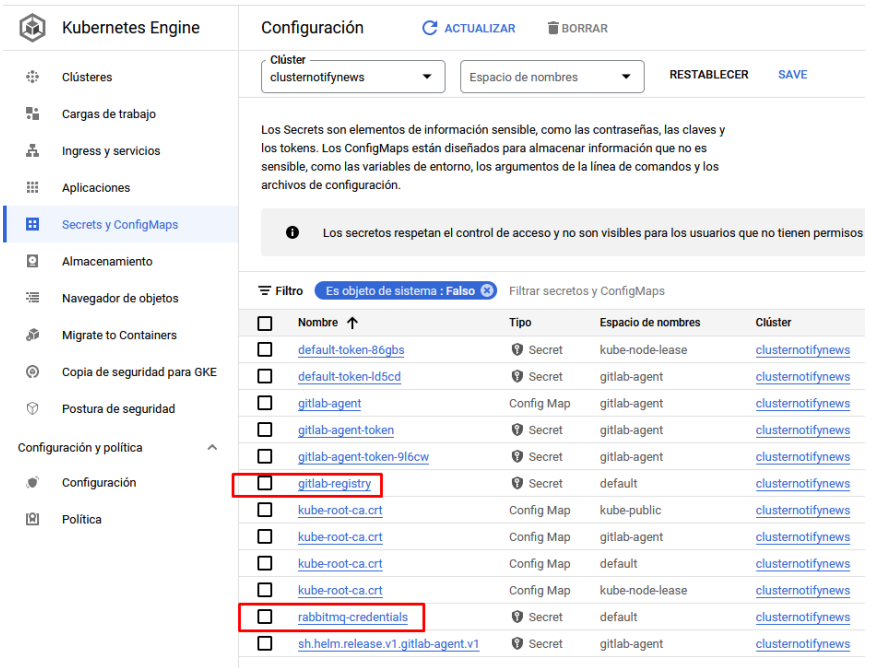
```
kubectl apply -f rabbit-secret.yaml
```

El contenido del archivo es el siguiente

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-credentials
type: Opaque
data:
  RABBITMQ_DEFAULT_USER: dXNlcm5hbWU=
  RABBITMQ_DEFAULT_PASS: cGFzc3dvcmQ=
```

```
C:\tfg\microservices\RabbitMQ>kubectl apply -f rabbit-secret.yaml
secret/rabbitmq-credentials created
```

Para comprobar que los Secrets se han creado de forma correcta se realiza una consulta desde el panel de Google.



A continuación se crea el recurso de rabbitMQ

```
C:\tfg\microservices\RabbitMQ>kubectl apply -f rabbitmq-deployment.yaml
statefulset.apps/rabbitmq created

C:\tfg\microservices\RabbitMQ>kubectl apply -f rabbitmq-service.yaml
```

Hay que tener en cuenta que para el despliegue operativo de los microservicios, se deberá configurar las variables de conexión correspondientes en cada proyecto.

Anexo IX Retrospectivas

Sprint 1

En este *Sprint*, se han logrado importantes avances en el proyecto, aunque se han presentado algunas dificultades. En primer lugar, se ha realizado una descripción de la arquitectura del sistema, aunque se ha necesitado más tiempo del previsto para ello.

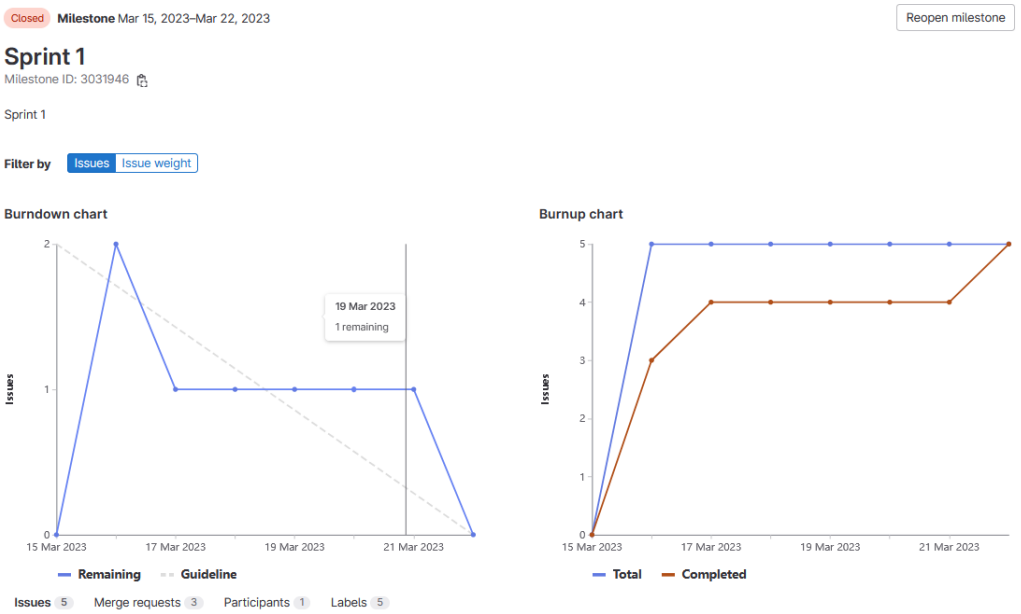
Por una parte, se ha implementado la estructura básica de los microservicios especificados para este *Sprint*, aunque se ha presentado la dificultad del desconocimiento de algunas herramientas para utilizar con el segundo lenguaje.

Otro aspecto destacable es que se ha adelantado la documentación del trabajo, lo que muestra un buen trabajo de investigación y planificación. Además, se ha dedicado un tiempo importante a estudiar posibles soluciones técnicas viables para el proyecto, aunque se recomienda centrarse más específicamente en las tareas de cada *Sprint* en vez de dedicar tanto tiempo a la investigación.

También se destaca que se ha sufrido una enfermedad en casa, lo que ha dificultado el trabajo y ha requerido recuperar tiempo fuera de la planificación, lo que muestra que siempre deben tenerse en cuenta posibles imprevistos.

Entre las lecciones aprendidas, se identifica la importancia de identificar correctamente el tipo de herramientas a utilizar, lo que puede ahorrar tiempo en el desarrollo del proyecto. También se sugiere subdividir las tareas concretas para no desviarse de las acciones que se toman sobre las incidencias.

Finalmente, reseñar que se ha conseguido terminar el trabajo a tiempo, lo que es un aspecto muy positivo.



Sprint 2

Durante este *Sprint* se ha tomado la decisión de usar Python como lenguaje para el tercer servicio.

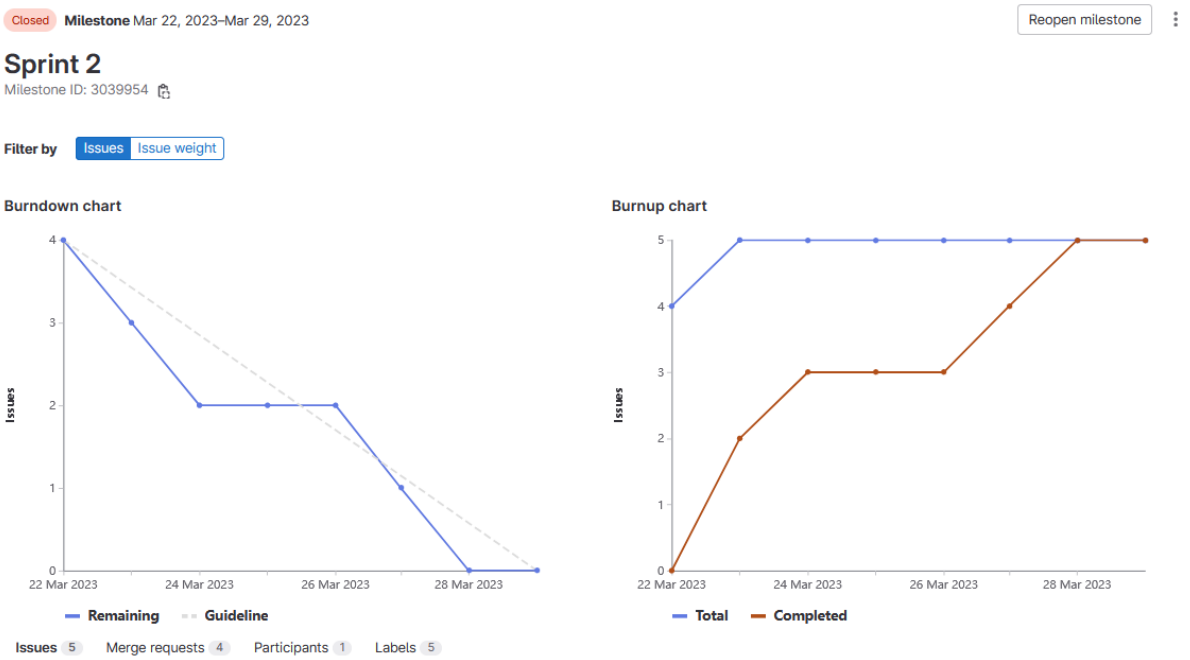
Una vez terminado el desarrollo del esqueleto, se ha decidido continuar la implementación y dockerizarlo para aprovechar el impulso actual. Se estima que continuar en este punto optimiza los tiempos de entrega al evitar cambios de tareas. Esta tarea también ha permitido tener una visión general del funcionamiento de Python.

Como parte positiva de la tarea anterior, cabe destacar el refuerzo del conocimiento sobre docker, incluyendo la asignación de volúmenes, la creación de redes y la descarga directa de imágenes de Docker Hub.

A su vez, se ha refactorizado el servicio en C#, se han utilizado los patrones de DTO y de inyección de dependencias mediante la creación de interfaces y se han implementado tests que cubren todos los métodos del controlador. Se ha consumido más tiempo para configurar el proyecto y el repositorio, de forma que los test estén incluidos en el mismo repositorio, debido a que ha sido necesario cambiar la estructura de la solución y los proyectos. Esto ha permitido validar la documentación de creación del servicio.

Este Sprint ha mostrado la necesidad de tener cierto tiempo dimensionado para la selección, instalación y prueba de algunas herramientas, lo cual es importante tener en cuenta para planificaciones futuras que involucren elementos desconocidos.

En cuanto a la documentación, se ha decidido crear unos anexos que permitan realizar las tareas básicas del proyecto. La información específica sobre los anexos se incluye en el documento principal. Toda la documentación de los anexos está actualizada en este punto.



Sprint 3

Durante este *Sprint* se han abordado tres tipos de tareas:

1. Dockerización de servicios: esta tarea se ha completado sin problemas.

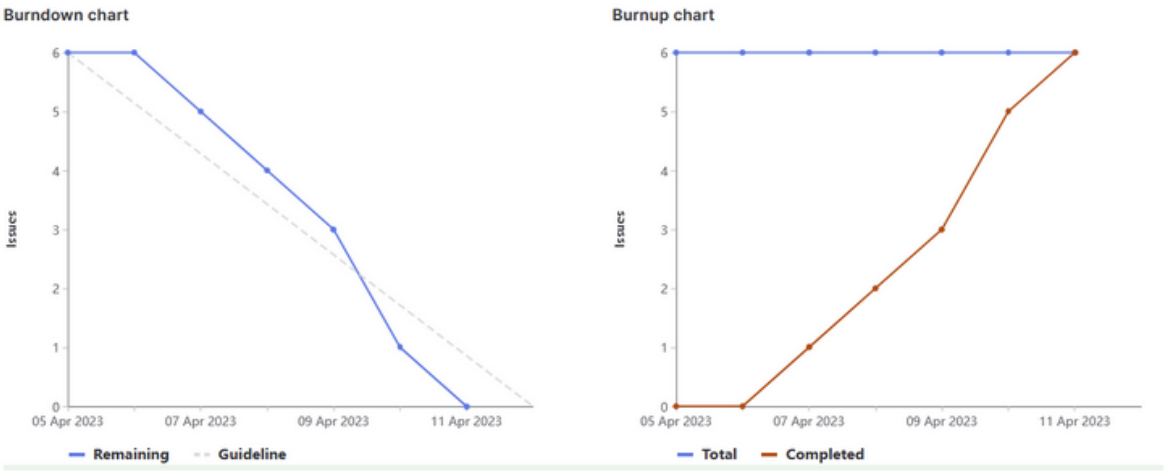
2. Creación de un pipeline básico en GitLab: se ha estudiado la creación de pipelines y se ha creado uno para cada uno de los tres servicios, incluyendo dos etapas (build y test). Sin embargo, el test del servicio Python ha presentado problemas, por lo que se considera una deuda técnica que se abordará en el futuro con el desarrollo final del pipeline.

3. Estudio del despliegue en plataformas en la nube: se ha investigado el proceso de despliegue en Azure y AWS, lo que ha llevado a identificar varias posibles estrategias para el despliegue final. Se podría optar por usar los registros de contenedores de los proveedores, o bien una solución intermedia como Docker Hub. Por el momento, se ha decidido utilizar el registro de contenedores de GitLab, ya que es privado y no implica un coste adicional.

Sprint 4

Este *Sprint* ha demostrado que el aumento de complejidad del sistema puede hacer que tareas que inicialmente parezcan sencillas tiendan a tardar más tiempo debido a las implicaciones que puede tener cualquier pequeño cambio en el sistema. Es positivo trabajar con microservicios porque de esta forma los problemas se aíslan, pero, por otro lado, la cantidad de tareas que se debe realizar en cada capa exige de máxima atención y coordinación. Debe funcionar el aplicativo con su parametrización, la dockerización, incluyendo la imagen y el contenedor, para ver que todo

está correcto y finalmente el paso a kubernetes, con la creación del deployment y el servicio que exponga la API al exterior. En este punto, cualquier problema de configuración puede crear importantes retrasos inevitables, pues la parametrización debe probarse necesariamente para comprobar que el sistema funciona en armonía.



Sprint 5

En este *Sprint*, se ha comenzado a trabajar en el desarrollo de las funcionalidades de los microservicios. En la fase de creación del esqueleto, se progresó bastante con la implementación de la funcionalidad de alta de usuario, preparando los endpoints para poder gestionar todas las peticiones principales (POST, PUT, DELETE y GET) por lo que la carga futura de este desarrollo se ve reducida. También se ha avanzado en la programación de tareas de creación de infraestructura, empezando por la creación de las bases de datos y del servidor de RabbitMQ en el clúster local de Kubernetes, ya que esta parte es necesaria para implementar la lógica de los servicios.

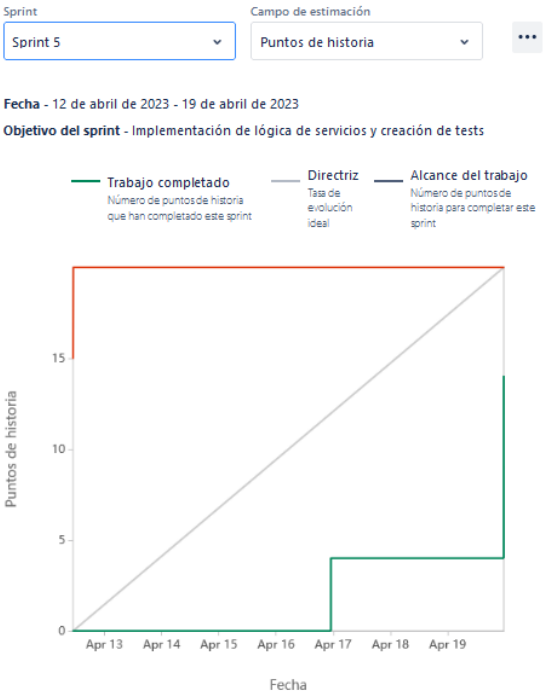
Sin embargo, el desarrollo simultáneo de la lógica de varios servicios a la vez está siendo más costoso de lo esperado, y como resultado, se han producido algunos retrasos en este *Sprint*. Por una parte, se ha identificado que este retraso se debe al trabajo simultáneo en varios servicios. Por otro lado, se debe mejorar las *Definition of Ready* (DoR) y *Definition of Done* (DoD), ya que muchas tareas tienen prerrequisitos o no están correctamente definidas, lo que dificulta saber cuándo se pueden dar por terminadas de forma efectiva.

Por lo tanto, se propone trabajar en un microservicio cada vez, en todas las fases que sean necesarias, aunque ello implique replanificación. También, se considera necesario aplicar unos elementos mínimos de calidad antes de considerar una tarea como entregada, tales como la realización de algunos tests unitarios y funcionales básicos.

Finalmente, se ha decidido enfocar todo el esfuerzo del *Sprint* en msgquery, el componente que se considera el más complejo. Se han conseguido resultados positivos con este desarrollo, implementando toda la lógica necesaria, que incluye la extracción de datos de la BBDD, la consulta externa a la API de noticias, la serialización y la comunicación con RabbitMQ, a lo que se acompaña la realización de nuevos tests. Por otro lado, no se ha conseguido automatizar los tests

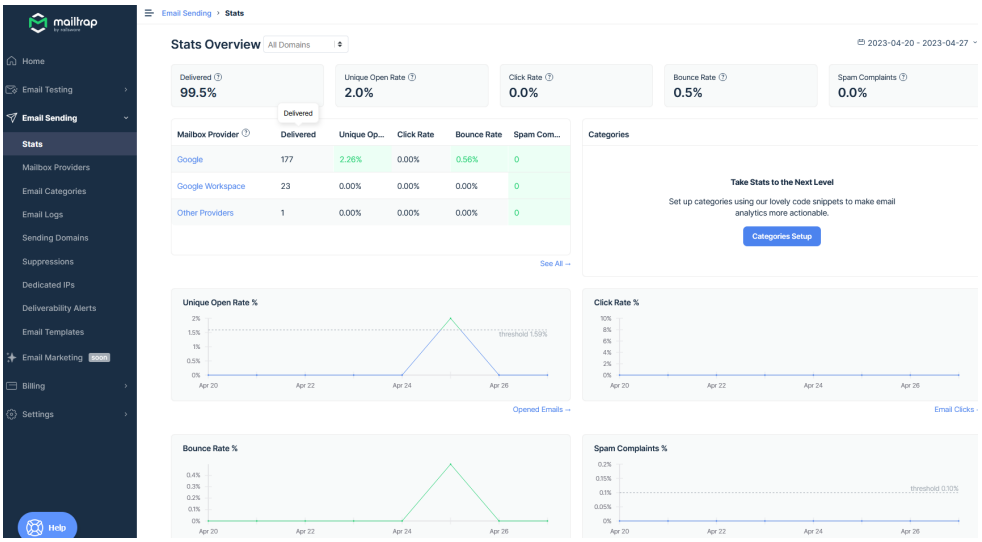
funcionales debido al desconocimiento del uso de Rabbit. Este punto lleva un tiempo considerable, ya que, además de realizar las pruebas locales, hay que configurar los pipelines para su ejecución en GitLab.

Otro imprevisto en el *Sprint* ha sido la imposibilidad de conectar con el servidor de correo de Gmail, que se proponía inicialmente debido a un cambio en la política de conexión de google. Por este motivo se han tomado dos decisiones. Por una parte, se ha decidido en este punto separar el componente de envío de correo del componente de consulta de las APIS, lo que implica una mayor complejidad del proyecto, pero a cambio reporta una mayor modularidad. Por otro lado, queda pendiente para próximos *Sprints* la investigación de un nuevo acercamiento al envío de correo. Se propone investigar una opción gratuita y limitada que ofrece Amazon o bien intentar realizar la configuración en Google según los nuevos requisitos de la plataforma. Ambos casos requieren estudio para saber cuál es la decisión más adecuada.

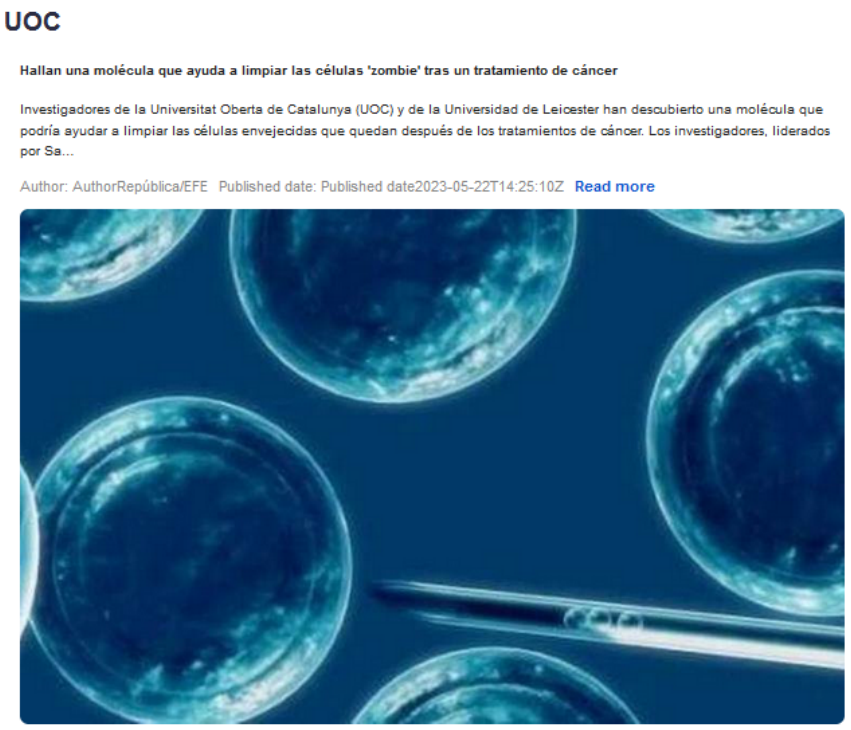


Sprint 6

Durante este *Sprint* el esfuerzo se ha centrado en el desarrollo del componente de envío de correo. Se ha creado un nuevo componente msgdispatch que se encarga de recibir mediante colas asíncronas el contenido de correo que se debe enviar a los usuarios. Para ello se realiza la conexión mediante un servidor externo llama mailtrap que es el encargado de gestionar el envío a la vez que permite el control de estadísticas. Una vez confirmado el envío, el componente envía un mensaje informativo de vuelta al componente principal.



El desarrollo de esta funcionalidad no ha permitido el desarrollo del componente de consulta de histórico en Python. La mayor complejidad del *Sprint* ha sido identificar la forma de enviar el correo, comprobar el envío correcto de las estructuras de los mensajes y el aprendizaje del uso de las colas en RabbitMQ.



Al haber alcanzado el desarrollo de 3 componentes en diferentes lenguajes se propone continuar en el próximo *Sprint* con la dockerización y la inclusión en el clúster de este componente así como centrar los esfuerzos en la implementación de los procesos del *pipeline* para el despliegue del aplicativo en las plataformas de los proveedores de la nube.

Sprint 7

Este *Sprint* ha sido especialmente duro y se ha decidido alargarlo una semana extra porque se han encontrado muchos problemas a la hora de poder comunicar el Pod de rabbit con los otros servicios y se consideraba fundamental tener una entrega mínimamente operativa para la PEC 3. A nivel de gestión, el *Sprint* ha sido caótico y la imposibilidad de conseguir la comunicación interna de los pods ha hecho que se retrasara todo el trabajo del despliegue.

El trabajo ha sido muy intenso y exasperante hasta el punto de afectar profundamente a la productividad. En este punto se tomó la decisión de reducir el número de horas de trabajo para poder despejar la mente, pues la actividad fue continua, con jornadas diarias de trabajo muy largas y sin días de descanso. El hacer otro tipo de actividades que no fuera la dedicación total al proyecto fue totalmente necesaria, pues se había llegado a un momento de *burnout*.

Por otra parte, ha habido algunos avances positivos. Se ha descubierto terraform que ha permitido crear una configuración de un clúster declarativo en Google desde gitlab. El proceso es sencillo y parece que se puede aplicar también con sencillez en Amazon.

Además, se han mejorado los pipelines para que registren las imágenes de los contenedores y de esta forma el clúster pueda tomar siempre las últimas compilaciones.

Creo que parte del tiempo perdido se debe a la pérdida de enfoque del objetivo principal del proyecto, que era el de la creación y despliegue de clústers y esto ya es suficientemente costoso debido a que en ese sentido el conocimiento previo era nulo, y se centró demasiado tiempo en

realizar un diseño óptimo de los servicios y sus funcionalidades, lo cual era un objetivo secundario. Por otra parte, el haber conseguido un producto de calidad importante reporta una gran satisfacción, con unas desviaciones tolerables y todavía se considera que se puede realizar una entrega de prototipo mínimo desplegado en condiciones pero a costa de reducir el alcance.

Un problema fundamental también ha sido el desconocimiento de .NET y sus configuraciones, lo que ha retrasado durante varios días el *Sprint.*

En este momento se procede a cerrar la mayoría de las historias de usuario, pues ya han sido implementadas, se prioriza centrar los esfuerzos en la orquestación del clúster y como objetivo secundario el despliegue en las plataformas en las que de tiempo, empezando con google pues el clúster ya está creado. Cualquier otro objetivo secundario del proyecto se abandona, puesto que se considera que no aportan tanto valor.

Sprint 8

En este punto se consideran conseguidos los objetivos proyecto. Se ha conseguido montar el clúster local de forma satisfactoria y la orquestación adecuada de los servicios. El tiempo restante del trabajo se centrará en cuatro aspectos fundamentales:

1. Pruebas y optimización del producto
2. Despliegue en las plataformas de proveedores de servicio, al menos una
3. Revisión de la documentación
4. Creación de vídeo de demostración
5. Iteraciones desde el punto 2 si queda tiempo restante

Sprint 9 y 10

Se consiguen los objetivos indicados en el Sprint anterior.

- Se optimiza el código de los servicios, se realiza una nueva implementación de clase de servicio principal de envío de msgdispatch y se agrega una plantilla html para el envío de correo mediante Thymeleaf.

- Se hacen pruebas en las plataformas, se realizan despliegues y recreaciones del entorno.

- Se revisa la documentación y se crean los vídeos entregables de explicación de proyecto y de demostración de producto.

- Se comprueba la recepción diaria de los correos con la temática solicitada por las personas usuarias.