UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Tecnologías de la Información

TRABAJO FIN DE GRADO

Sistema de Notificación Automatizada de Peticiones de Servicio

Roque Rojo Bacete

Febrero, 2022

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

Tecnologías de la Información

TRABAJO FIN DE GRADO

Sistema de Notificación Automatizada de Peticiones de Servicio

Autor: Roque Rojo Bacete

Tutor: David Vallejo Fernández

Co-tutor: Samuel Campos Herros

Febrero, 2022

Sistema de Notificación Automatizada de Peticiones de Servicio
© Roque Rojo Bacete, 2022

Esta obra ha sido preparada con la plantilla LaTeX de TFG para la UCLM publicada por Jesús Salido en GitHub[1] y Overleaf [2] como parte del curso *«LaTeX esencial para preparación de TFG, Tesis y otros documentos académicos»* impartido en la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

La copia y distribución de esta obra no está permitida sin la previa autorización del propietario de los derechos de explotación de la misma, siendo ésta Avanttic Consultoría Tecnológica S.L

avanttic

[1] https://github.com/JesusSalido/TFG_ESI_UCLM, DOI: 10.5281/zenodo.4574562
[2] https://www.overleaf.com/latex/templates/plantilla-de-tfg-escuela-superior-de-informatica-uclm/phjgscmfqtsw

Tribunal:

Presidente: _____

Vocal: _____

Secretario(a): _____

Fecha de defensa: _____

Calificación: _____

Presidente	Vocal	Secretario(a)
Fdo.:	Fdo.:	Fdo.:

A mi abuelo Francisco,
por creer en mí antes de que yo mismo lo hiciera.

TRABAJO FIN DE GRADO - ESCUELA SUP. DE INFORMÁTICA (UCLM)

Sistema de Notificación Automatizada de Peticiones de Servicio

Roque Rojo Bacete
Ciudad Real, Febrero 2022

Resumen

Este Trabajo de Fin de Grado se enmarca dentro de la empresa *Avanttic* Consultoría Tecnológica S.L, en la que debido a la creciente demanda de los servicios de soporte dentro de una empresa comprometida con la atención al cliente y a los procesos de negocio del mismo, la capacidad de prestar una atención rápida y eficiente en los esquemas de producción avanzadas es esencial.

En la actualidad, existen varias herramientas de gestión de peticiones en función de cada cliente (como *Jira* o *ServiceNow*). Tradicionalmente, la notificación de peticiones de servicio ha supuesto un punto de ralentización, debido a la existencia de un administrador que gestiona las peticiones de forma manual. Es decir, el administrador se comunica con consultores hasta encontrar uno que este disponible y cualificado para resolver la petición o la incidencia. Por lo que, para mitigar los efectos causados por el cuello de botella que surge a la hora de buscar un consultor, es necesario definir un sistema capaz de notificar las de peticiones de servicio de manera automatizada en función de las aptitudes y la disponibilidad de los consultores. La automatización de este proceso supone que las peticiones e incidencias se resuelvan antes, lo que aumenta la efectividad de la empresa, la satisfacción de los clientes y que se cumpla el tiempo pactado entre *Avanttic* y los clientes.

Así, en este *TFG* se propone el diseño, desarrollo y despliegue de un sistema que notifique automáticamente las peticiones de servicio a los consultores que estén disponibles en función de sus preferencias, mediante comunicaciones explícitas, como por ejemplo el correo electrónico. Con esto, se pueden mitigar los potenciales *cuellos de botella* y los problemas que se pueden producir a la hora de buscar un consultor libre. El sistema esta compuesto por 2 módulos:

1. **StreamConnect**. Gestiona las peticiones de servicio e incidencias y lleva a cabo la notificación vía correo electrónico, diferenciando entre peticiones urgentes y estándar.
2. **StreamSub**. Que permite personalizar las notificaciones que se reciben de *StreamConnect* en función de sus aptitudes, gestionando para ello los consultores, su disponibilidad y sus suscripciones a clientes y áreas. Y por otra parte también permite gestionar las empresas y las áreas evitando que se tenga que acceder a *Streaming Service*, actuando como una fuente de gestión única.

El empleo de este tipo de arquitectura permite escalar tanto vertical como horizontalmente el sistema, posibilitando la evolución del mismo.

BACHELOR DISSERTATION - ESCUELA SUP. DE INFORMÁTICA (UCLM)

Automated Service Request Notification System

Roque Rojo Bacete
Ciudad Real, February 2022

Abstract

This Bachelor Dissertation is framed within *Avanttic* Consultoría Tecnológica S.L, in which due to the growing demand for support services within a companny committed to customer service and customer business processes, the ability to provide fast and efficient attention in advanced production schemes is essential.

Currently, the are several tools for managing different requests depending on each customer (such as *Jira or ServiceNow*). Traditionally, service request notification has been a point of slowdown, due to the existence of an administrator who manages the requests manually. That is, the administrator communicates with consultants until he finds one who is available and qualified to resolve the request or incident. Therefore, to mitigate the effects caused by the bottleneck that arises when looking for a consultant, it is necessary to define a system capable of notifying service requests in an automated way based on the skills and availability of consultants. The automation of this process means that requests and incidents are resolved sooner, which increases the effectiveness of the company, customer satisfaction and that the agreed time between *Avanttic* and customers is met.

Thus, this bachelor dissertation proposes the design, development and deployment of a system that automatically notifies service requests to available consultants according to their preferences, through explicit communications, such as email. This can mitigate potential bottlenecks and problems that can occur when looking for a free consultant. The system is composed of 2 modules:

1. **StreamConnect**. Manages service requests and incidents and performs notification via e-mail, differentiating between urgent and standard requests.
2. **StreamSub**. It allows you to customize the notifications you receive from *StreamConnect* according to your capabilities, managing consultants, their availability and their subscriptions to clients and areas. On the other hand, it also allows you to manage companies and areas without having to access Streaming Service, acting as a single management source.

The use of this type of architecture allows the system to scale both vertically and horizontally, enabling its evolution.

Agradecimientos

En primer lugar, darle las gracias a ***Pepi***, por enseñarme que siempre hay varias formas de ver y afrontar los problemas.
A mi ***Padre***, por enseñarme a trabajar duro y a ser constante, y junto con mi madre, darme la oportunidad.
A la ***chica más "trabajadora" que conozco***, por comprenderme, por escucharme y concederme todos mis caprichos **e.é** . Y en definitiva, ser mi amasijo de huesos.
A mi ***tía Maribel***, por su apoyo incondicional, sentirse orgullosa de mi y ser el corazón de la familia.
A ***Loloro***, ***Walls*** y al ***: Niñito***, por ser los mejores compis de piso que he podido tener.
A ***El de los Audios***, por escucharme, comprenderme y llorar juntos siempre que lo he necesitado. Y hacerme oír los *podcast* más largos que he escuchado nunca.
A ***mi sol y mis estrellas***, y a ***la luna de mi vida***, por alegrarme vida.
A ***todos mis amigos*** que he hecho durante esta etapa. Por todas las noches que nos hemos pasado en la biblioteca, por las fiestas, por las borracheras y lo más importante, por como nos hemos ayudado entre todos en los peores momentos.
A ***las chicas más pillonas que existen***, por contarme los peores chistes que he escuchado en mi vida, por animarme en los malos momentos y sacarme a beber **e.é** .
A ***L.A***, por ayudarme siempre que se lo he pedido.
A ***Mercedes***, por su apoyo y ayudarme a enfrentar mis demonios.
A mi ***tutor Samuel***, por ser un gran profesional, por ayudarme a encontrar la solución a los problemas que me han surgido y aguantar lo pesado que puedo llegar ser.
A mi ***tutor David***, por acompañarme en este proceso y estar dispuesto siempre a ayudarme.
A ***Avanttic*** y a ***los compañeros***, por darme la oportunidad, y hacerme sentir parte esta pequeña *familia.*
A ***Sonso,*** por ayudarme a conservar el mental a lo largo de todo este proceso.
A mi ***compi Laura***, por todo lo que nos hemos apoyado y ayudado durante esta etapa.
A ***Brandon Sanderson***, por hacerme ver que mientras *lo entienda*, no puedo fracasar.
A la ***gente que ya no forma parte de mi vida***, porque ahora comprendo la suerte que tuvimos de apoyarnos mientras aún estábamos juntos.

Viaje antes que Destino.

Roque Rojo Bacete
Ciudad Real, 2022

Acrónimos

LISTA DE ACRÓNIMOS

API	:	Application Programming Interface
AWS	:	Amazon Web Service
DDoS	:	Distributed Denial of Service
DNS	:	Domain Name System
CI/CD	:	Continuous Integration/Continuous Delivery
CSS	:	Cascading Style Sheets
E/S	:	Entrea y Salida
ESB	:	Enterprise Service Bus
HTML	:	HyperText Markup Language
HTTP	:	Hypertext Transfer Protocol
IaaS	:	Infraestructure as a Service
IDE	:	Integrated Development Environment
ITAML	:	Information Technology Active Management
ITIL	:	Information Technology Infrastructure Library
ITSM	:	Information Technology Service Management
JVM	:	Java Virtual Machine
JSON	:	JavaScript Object Notation
MySQL	:	My Structured Query Language
NPM	:	Node Package Manager
OCI	:	Oracle Cloud Infraestructure
OKE	:	Container Engine for Kubernetes
PaaS	:	Plataform as a Service
SaaS	:	Software as a Service
SOA	:	Service-Oriented Architectures
SQL	:	Structured Query Language
SSL	:	Secure Sockets Layer
TI	:	Tecnologías de la Información
TFG	:	Trabajo Fin de Grado
UI	:	User Interface
URL	:	Uniform Resource Locator
VCN	:	Virtual Cloud Network

Índice general

Índice de figuras

Índice de tablas

Índice de listados

CAPÍTULO 1

Introducción

Este Trabajo Fin de Grado, se enmarca en el contexto de la empresa *Avanttic* Consultoría Tecnológica S.L., que ante el aumento del número de peticiones de servicios e incidencias por parte los clientes y utilizando un sistema de *ITSM* como es *Jira* ha detectado un cuello de botella que ralentiza la búsqueda de un consultor disponible para resolver la incidencia. Este *TFG* tiene como propósito solucionar este cuello de botella.

1.1. MARCO DE TRABAJO Y PROBLEMÁTICA

La realización de este *TFG* ha sido posible gracias al convenio *FORTE* existente entre la Escuela Superior de Informática de la Universidad de Castilla-La Macha y la empresa *Avanttic* Consultoría Tecnológica S.L, *partner* 100 % de tecnología *Oracle*. Que permite a alumno realizar las prácticas en una empresa al mismo tiempo que asigna un proyecto real propuesto por la empresa, que posteriormente podrá utilizar como *TFG*.

Este *TFG* busca solventar el cuello de botella que se produce a la hora de notificar y buscar un consultor disponible. Este cuello de botella queda expuesto en la figura 1.1. Como podemos ver en la figura mencionada anteriormente, cuando un cliente crea una incidencia en *Jira* existe una una persona, a la cual llamaremos administrador que es notificada por *Jira*. Una vez el administrador recibe la notificación empieza a entablar comunicación vía correo electrónico uno a uno consultores que podrían resolver la incidencia hasta encontrar a uno disponible. Una vez encuentra a un consultor disponible le asigna la petición de servicio o incidencia y este se pone a trabajar en ella hasta resolverla.

Esto deriva en problemas de rendimiento y eficacia. Además, *Avanttic* ofrece a algunos de sus clientes un servicio *premium* que estipula que se debe empezar con la resolución del incidente en menos de 1 hora, por lo cual encontrar a un consultor disponible rápido es esencial.

1.2. PROPUESTA

Para solucionar esta problemática, se ha propuesto un sistema web compuesto por 2 módulos. Los cuales se detallan a continuación:

- **StreamSub**. Este módulo permite gestionar los consultores, las suscripciones, las áreas, las empresas (son representadas por *Streams*) y las asignaciones (que representa las áreas que tienen contratada cada empresa).

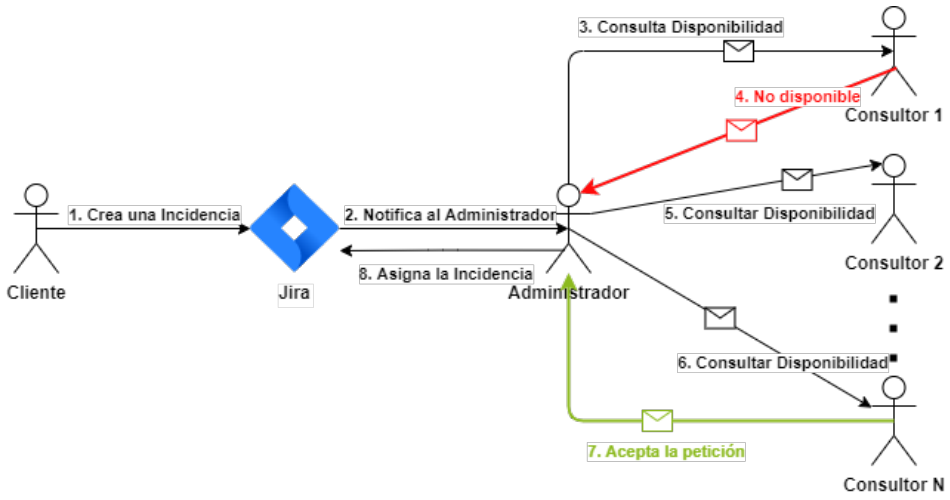


Figura 1.1: Representación del Cuello de Botella

- **StreamConnect**. Este módulo es el encargado de recibir las incidencias de *Jira* y realizar el envío de correos a los consultores suscritos y disponibles solicitando para ello información a el módulo *StreamSub*.

A continuación, explicaremos el flujo de trabajo de la solución que hemos propuesto, el cual queda representado en la figura 1.2.

1. **Creación de una Incidencia**. El cliente crea una incidencia *Jira* rellenando los campos pertinentes.
2. **El Webhook envía información**. Se ha configurado un Webhook que reacciona a la creación de incidencias. Enviando información de las mismas en formato *JSON* mediante una petición *POST* a *StreamConnect*, que obtendrá los datos más importantes de la misma.
3. **Producir Incidencias**. Con los datos de la incidencia y solicitando datos a *StreamSub* sobre las áreas que tiene contratadas las empresa que origina la incidencia, *StreamConnect* publica un mensaje en *Streaming Service* con los datos de la incidencia en la partición designada para esa área.
4. **Consumir Incidencias**. Cuando se detecta que se ha producido un nuevo mensaje *StreamConnect* obtiene los datos del área de la que procede y de la empresa y solicita a *StreamSub* los *emails* de los consultores están disponibles y están suscritos al área de la empresa de las que procede esa incidencia.
5. **Notificar la Incidencia**. Cuando *StreamConnect* recibe la lista de los *emails* de los consultores de *StreamSub*, en función de si la incidencia es urgente o no, seleccionará una plantilla de las 2 que se han creado. Y la rellenará con los datos de la incidencia, finamente realizará el envío de los correos a las direcciones obtenidas de *StreamSub* (en la figura 6.3 puede verse un ejemplo incidencia con ambas plantillas).
6. **Aceptar la Incidencia**. Una vez que los consultores reciben el correo para aceptar la incidencia simplemente tienen que acceder al enlace que aparece en correo, que los redirigirá a la incidencia en *Jira* y podrán aceptarla fácilmente.

Esta solución permite eliminar el cuello de botella descrito en la sección anterior, puesto que pasamos de tener un sistema de notificación de 1:1 (figura 1.1), a un sistema de notificación de 1:N (figura 1.2). Permitiendo personalizar el tipo incidencias que recibe cada consultor en función de sus aptitudes.

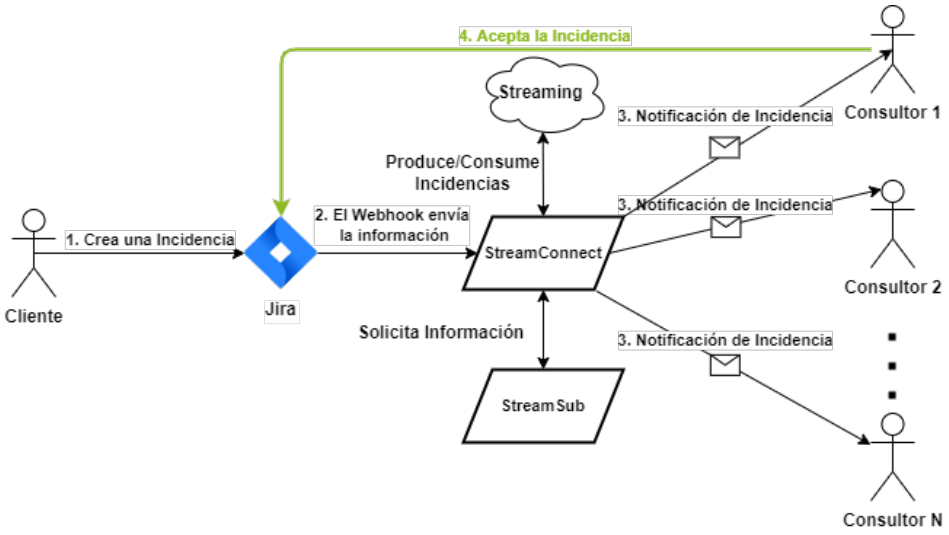


Figura 1.2: Solución Propuesta

1.3. ESTRUCTURA DEL DOCUMENTO

A continuación, se expone la estructura que tendrá este *TFG*, el cual constituido por 7 capítulos y 2 anexos.

- **Capítulo 1: *Introducción.*** En este capítulo se realiza una breve introducción del problema que va a resolver este *TFG*, así como el marco de trabajo, la solución propuesta y la estructura de este documente. En líneas generales se pretende ofrecer al lector una visión general del trabajo realizado.
- **Capítulo 2: *Objetivos.*** Este otro capítulo detalla el objetivo general y los objetivos específicos que deben de cumplirse para garantizar la finalización de este *TFG*. Junto a las competencias que se adquieren.
- **Capítulo 3: *Estado del Arte.*** Este capítulo tiene como objetivo poner en contexto al lector sobre *ITSM*, el desarrollo web y la computación en la nube.
- **Capítulo 4: *Metodología.*** En este capítulo se expondrá la metodología de Gestión y desarrollo utilizada a lo largo del ciclo de vida del proyecto, así como la planificación y el marco tecnológico empleado.
- **Capítulo 5: *Arquitectura.*** En este otro capítulo se explicará de manera detallada la arquitectura general que tiene el proyecto. Detallando los problemas que han surgido, la manera de afrontarlos y las ventajas de a ver elegido esa solución y no otra. Finalmente, cuenta con una serie de diagramas que facilitan al lector la compresión del proceso.
- **Capítulo 6: *Resultados.*** En este capítulo se explicarán las diferentes iteraciones que se han seguido durante el desarrollo del proyecto.

- **Capítulo 7: *Conclusiones.*** En este último capítulo se analizará y justificará la consecución del objetivo principal, los objetivos específicos y las competencias adquiridas. Finalizando con una reflexión sobre las posibles mejoras del proyecto y una valoración personal del mismo.
- **Anexo A: *Listados Complementarios.*** En este anexo se encontraran listados de código que ilustrarán de forma complementaria lo que se explica a lo largo de todo el documento.
- **Anexo B: *Figuras Complementarias.*** En este anexo se encontraran figuras ilustrarán de forma complementaria lo que se explica a lo largo de todo el documento.
- **Anexo C: *Visión General del Funcionamiento de la Aplicación.*** En este anexo se encontrará una explicación adicional junto a unos diagramas de secuencia para ayudar al lector a comprender el funcionamiento de la aplicación.

CAPÍTULO 2

Objetivos

En este capítulo se van a describir todos los objetivos del Trabajo de Fin de Grado, incluyendo tanto el objetivo general del proyecto, como los objetivos específicos que derivan de él.

2.1. OBJETIVO GENERAL

El objetivo principal de este *TFG* es el diseño y desarrollo de un sistema que notifique automáticamente las peticiones de servicio a los consultores que se encuentren disponibles, filtrando entre los que están suscritos a un buzón, que representa e identifica al cliente dentro de *Avanttic*.

2.2. OBJETIVOS ESPECÍFICOS

En esta sección se van a describir los sub-objetivos en los que se puede dividir el objetivo principal del proyecto.

1. **Estudiar e instalar las tecnologías y herramientas relativas al desarrollo, despliegue e integración del sistema en el contexto de producción.** Este objetivo específico engloba todo el proceso de aprendizaje tanto teórico como práctico que el alumno ha necesitado para abordar este *TFG* con éxito.
2. **Crear, desarrollar e integrar el módulo de notificación automatiza de peticiones de servicio.** Este objetivo específico abarca el diseño, la elección de tecnologías, el desarrollo, el despliegue en *Oracle Cloud*, la integración con *StreamSub* y las pruebas del módulo *StreamConnect*. Además, incluye también la conexión con el *Webhook* de *Jira* y la conexión con *Oracle Streaming Service*, la gestión de las incidencias, el renderizado de plantillas *HTML*.
3. **Crear, desarrollar e integrar el módulo para gestionar los consultores, suscripciones, *Streams*, áreas y asignaciones**. Este objetivo específico comprende el diseño, la elección de tecnologías, el desarrollo, el despliegue en *Oracle Cloud*, la integración con *StreamConnect* y las pruebas del módulo *StreamSub*. Además del diseño, creación, despliegue y conexión de la base de datos utilizada para gestionar los datos de los consultores y de los clientes.
4. **Creación de la documentación necesaria que detalle el trabajo realizado a lo largo de este proyecto.** Este objetivo específico abarca toda la documentación creada para durante este *TFG*, incluyendo los esquemas, bocetos, figuras, diagramas. Y en definitiva, la elaboración del presente documento.

2.3. COMPETENCIAS ACADÉMICAS

Aparte de los intereses de la empresa, con el desarrollo de este *TFG* se pretenden adquirir las siguientes competencias de la rama de *Tecnologías de la Información*:

- **IT1:** Capacidad para comprender el entorno de una organización y sus necesidades en el ámbito de las tecnologías de la información y las comunicaciones.
- **IT3:** Capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas.
- **IT5:** Capacidad para seleccionar, desplegar, integrar y gestionar sistemas de información que satisfagan las necesidades de la organización, con los criterios de coste y calidad identificados.

CAPÍTULO 3

Estado del Arte

En este capítulo se hablará de cómo se lleva a cabo la gestión de servicios de tecnologías de la información y el soporte técnico. Además, se pondrá en contexto al lector sobre los diferentes procesos de *ITSM* en la sección 3.1, sobre el de desarrollo, las arquitecturas y tecnologías web más utilizadas en actualidad en la sección 3.2 y sobre la computación en la nube, los modelos de negocio y las diferentes ventajas que ofrece en la sección 3.3,

3.1. SISTEMAS DE ATENCIÓN AL CLIENTE Y GESTIÓN DE PETICIONES

En esta sección se pone en contexto al lector sobre la forma en la que las organizaciones llevan acabo el soporte técnico y la atención al cliente en la actualidad.

3.1.1. Visión General

ITSM (Information Technology Service Management) o gestión de servicios de tecnología de la información es un término que describe la forma en la que los equipos de *TI* gestionan la prestación integral de servicios *TI* a los clientes. Esto incluye todos los procesos y actividades para diseñar, crear y entregar los servicios de *TI*, y para ofrecerles soporte [15].

Según el enfoque más conocido de *ITSM*, *ITIL* (*Information Technology Infrastructure Library*), consta de varios procesos básicos:

- **Gestión de Activos de *TI***.
- **Gestión de Problemas**.
- **Gestión del Conocimiento.**
- **Gestión de los Cambios**.
- **Gestión de Solicitudes de Asistencia.**
- **Gestión de las Incidencias**.

Los 3 primeros están fuera del ámbito clásico del soporte de *TI*, esto es porque *ITSM* abarca todas las actividades relacionadas con la entrega de un servicio *TI* a la empresa.

Antes de hablar sobre los diferentes procesos de *ITIL* definiremos algunos conceptos importantes:

- **Incidencia.** Es cualquier evento que altere, reduzca o amenace con reducir la calidad del servicio y necesita una respuesta cuanto antes.
- **Problema.** Causa subyacente no identificada de uno o varios incidentes que se puede prevenir.
- **Solicitud de Asistencia.** Es una solicitud formal de un cliente para que se le proporcione algo.

- **Cambio.** Consiste en añadir, modificar o eliminar algo que pueda afectar a los servicios de *TI*, normalmente suele estar vinculado a una solicitud de asistencia.

Gestión de Activos de *TI*

La gestión de activos de *TI (ITAM)* sirve para garantizar que los activos de una organización se puedan contabilizar, desplegar, mantener, actualizar y eliminar en el momento oportuno, es decir, garantiza que los elementos de valor, tanto los tangibles como los intangibles, de una organización tengan seguimiento y uso [15].

Un activo de *TI* es *hardware*, sistemas de *software* o información que tienen valor para una organización, por ejemplo, los ordenadores y las licencias de *software* que se utilizan para proporcionar servicios de *TI*. Los activos tienen un periodo de uso finito, por lo que, para maximizar el valor que una organización puede generar de ellos, el ciclo de vida de los activos se puede gestionar de forma proactiva.
Las organizaciones pueden definir distintas etapas para el ciclo de vida de sus activos, aunque las más comunes suelen ser: la planificación, la adquisición, la implementación, el mantenimiento y la retirada. El factor más importante de la gestión de activos de *TI* que debe realizar una organización es aplicar procesos durante todas las etapas del ciclo de vida de un activo para conocer el coste total de propiedad y optimizar el uso de sus activos.
Algunas de las ventajas que se obtienen con la gestión de activos de *TI* son:

- **Fuente de información única.** Proporciona una única fuente de información fiable en la que se refleja el estado, uso y precio de la organización.
- **Mejora el aprovechamiento y reduce el desperdicio de activos.** Al mantener una información actualiza evita compras innecesarias, reduciendo los costes asociados a licencias y soporte técnico.
- **Mejora la productividad.** Evita que haya personas que tengan que estar encargadas de realizar un seguimiento de los activos, permitiendo que se centren en otras tareas que proporcionen valor a la empresa.

Gestión de Problemas

La gestión de problemas consiste en identificar y gestionar las causas de los incidentes en un servicio *TI* [15].
Una gestión de problemas no es solo detectar y corregir incidentes, sino también en identificar y comprender las causas subyacentes de un incidente y determinar el mejor modo para eliminar el origen del problema.
Un problema se diferencia de un incidente en que, los incidentes finalizan cuando el servicio se vuelve a restablecer y a seguir en funcionamiento. Mientras que no se hayan resuelto las causas subyacentes y los factores que han contribuido en dar al incidente, el problema persistirá.
Los pasos principales pasos que contribuyen al proceso de gestión de problemas son:

1. **Detección de problemas**. Buscar problemas de forma proactiva para poder corregirlos o identificar soluciones alternativas antes de que se produzcan incidentes.

2. **Categorización y priorización**. Supervisar y evaluar los problemas conocidos para organizar los equipos, para que se encarguen de los más relevantes e importantes.
3. **Investigación y diagnóstico**. Identifica las causas subyacentes que contribuyen al problema y las mejores medidas para solucionarlo.
4. **Registro de errores conocidos**. Un *error conocido*, es un problema que tiene un origen ya documentado y una solución alternativa. Registrar esta información supone menos tiempo de inactividad si el problema desemboca en un incidente.
5. **Creación de una solución alternativa**. Una *solución alternativa* es una solución temporal para reducir el impacto de los problemas, evitando que se conviertan en incidentes. Este paso solo se lleva acabo si es necesario debido al que el problema no se pueda identificar y eliminar fácilmente.
6. **Resolución y cierre del problema**. Un problema se cierra cuando este ha sido eliminado y ya no puede causar incidentes.

Algunas de las ventajas que derivan del uso de una gestión de problemas son:

- **Reduce el tiempo de resolución.** Al no enfocarse únicamente en el incidente actual y centrarse en las causas, los equipos conocerán mejor el sistema, por lo que aprenderán de errores pasados, mejorando el tiempo de resolución.
- **Aumenta la productividad.** Al solucionar las causas de los incidentes, aparecen con menor frecuencia. Por lo que el equipo puede centrarse en otras tareas.
- **Incrementa la satisfacción del cliente.** Como he dicho anteriormente una buena gestión de problemas conduce a menos incidentes, por lo que los clientes tendrán menos incidentes y menos repetitivos, por lo que estarán mas satisfechos que si los incidentes fueran recurrentes.

Gestión de los Conocimientos

La gestión de conocimientos es el proceso de crear, seleccionar, compartir, utilizar y gestionar los conocimientos de toda una organización. La gestión de conocimientos pretende garantizar que las partes interesadas obtengan la información correcta, en el formato adecuado, en el nivel correspondiente y en el momento adecuando, en función de su nivel de acceso y otras políticas [15].

Las *bases de conocimientos* son el pilar básico de la gestión de conocimientos, y podemos definirlas como bibliotecas de autoservicio en línea con información sobre un producto, servicio, departamento o tema. La información puede proceder de cualquier parte, aunque en su mayoría procede de colaborados con amplia experiencia en el tema en cuestión. Las bases de conocimiento también pueden incluir preguntas frecuentes y guías para la solución de problemas entre otras cosas. Podemos diferenciar entre 3 tipos de conocimiento:

- **Conocimiento tácito**. Es el tipo de conocimiento que tiene su origen en el contexto, la práctica o la experiencia personal. Se encuentra en el cerebro de la gente, por lo cual es difícil de transmitir a los demás. Un ejemplo de conocimiento tácito es hablar un idioma.
- **Conocimiento explícito**. Es codificado, es decir, que se ha documentado y es fácilmente accesible, es fácil de almacenar y recuperar. Su principal reto es asegurarse que se revisa y actualiza.

- **Conocimiento implícito**. Esta integrado en los procesos, rutinas o la cultura de la organización. Puede tener un formato formalizado, como un manual, pero el conocimiento en sí no es explícito, si no que reside en la forma en la que funciona una organización.

La gestión de conocimiento surge por la necesidad que tienen los equipos de *TI* de mantenerse al día con un gran numero de tecnologías y procedimientos diferentes, necesarios para dar un soporte eficaz a los clientes. Una buena gestión de conocimientos debe aglutinar el conocimiento de todas las personas de la organización y permitir compartirlo fácilmente. Las ventajas que derivan de su uso son:

- Pone al alcance de las personas de la organización el conocimiento necesario para diseñar, desarrollar y proporcionar productos y servicios.
- Acorta los ciclos de desarrollos de nuevas iniciativas.
- Permite una gestión más eficaz de los entornos empresariales.
- Aprovechar mejor los activos y el capital intelectual de la plantilla, haciendo que este siempre disponible.

Gestión de Cambios

La gestión de cambios es una práctica de *TI* diseñada para minimizar las interrupciones en los servicios de *TI* mientras se realizan cambios en los sistemas y servicios críticos [15].

Un cambio es agregar, modificar o eliminar cualquier cosa que pueda repercutir de manera directa o indirecta en los servicios que se ofrecen. *ITIL* define 3 tipos de cambios:

- **Cambio estándar**. Este tipo de cambios son habituales, suponen un riesgo bajo y están preaprobados, siguen un proceso documentado y aprobado. Muchas empresas automatizan la gestión de estos cambios, permitiendo que los equipos se centren en los cambios normales y urgentes. Un ejemplo de cambio estándar puede ser: aumentar el almacenamiento de un dispositivo.
- **Cambio normal**. Son cambios que no son urgentes, no tienen un proceso definido y no están previamente aprobados. Un ejemplo podría ser una mejora del rendimiento.
- **Cambio urgente**. Son cambios que surgen de un error o amenaza imprevistos. Este tipo de cambios deben abordarse de forma inmediata para que los clientes o empleados puedan seguir utilizando el servicio o proteger el sistemas de una amenaza. Un ejemplo sería un parche de seguridad o ocuparse de la interrupción del servicio.

En la figura 3.1 se pueden apreciar los pasos que se siguen a la hora de procesar una solicitud de cambio:

1. **Solicitud de Cambio.** Alguien solicita un cambio, añadiendo notas sobre los posibles riesgos, implementación esperada y los sistemas a los que afecta.
2. **Revisión.** Un gestor revisa la solicitud de cambio, evaluando los riesgos y costes.
3. **Plan de Cambio.** El equipo crea un plan estratégico para el cambio, en el que se incluye documentación sobre los resultados esperados, los recursos necesarios, el cronograma, los requisitos de prueba y las forma de revertir el cambio si fuera necesario.

4. **Aprobación de los Cambios.** El gestor revisa el plan y aprueban el cambio.
5. **Implementación de los Cambios.** El equipo integra el cambio en el sistema al mismo tiempo que documenta los procesos y los resultados.
6. **Cierre de los Cambios.** El gestor revisa y cierra el cambio registrando en un informe si el cambio ha sido efectivo y puntual, si la estación que se hizo fue precisa y si se ajusto al presupuesto.

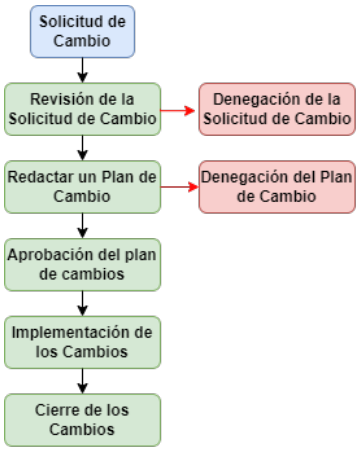


Figura 3.1: Pasos seguidos para gestionar un Cambio

Gestión de Solicitudes de Asistencia

La gestión de solicitudes de asistencia trata de procesar las solicitudes de asistencia los usuarios de una manera eficiente. Esta relacionada con otros procesos, como el de la gestión de incidentes, problemas y cambios. Suelen ser de bajo riesgo y pueden automatizarse, reduciendo la carga del equipo de *TI* [15].
Las organizaciones reciben un gran número de solicitudes de asistencia por lo que *ITIL* especifica que las solicitudes deben gestionarse en un proceso de resolución diferente además de los centros de asistencia (o *Service Desk*).
Esta resolución de solicitudes puede definirse como el proceso por el que se satisfacen las solicitudes de asistencia de los cliente e implica la gestión del ciclo de vida completo de todas las solicitudes de asistencia. El equipo del *Service Desk* se dedica a responder a las solicitudes y resolverlas, ofreciendo un servicio de soporte de calidad al cliente.
El proceso de resolución de solicitudes de asistencia consta de los siguientes pasos:

1. Un cliente solicita ayuda en un portal de asistencia, que llega en forma de *ticket*.
2. El equipo del *Service Desk* evalúa la solicitud de acuerdo con los procesos de aprobación y clasificación que define la empresa.
3. Un agente del *Service Desk* trabaja para resolver la solicitud de asistencia o deriva la solicitud en alguien que pueda hacerlo.
4. Una vez se resuelve la solicitud, el agente del *Service Desk* cierra el *ticket* y se pone contacto con el cliente para asegurarse de que esta satisfecho.

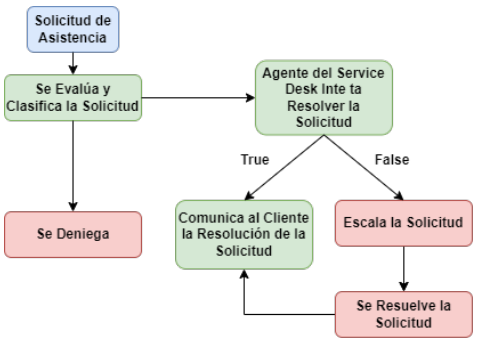


Figura 3.2: Proceso Gestión Solicitudes de Asistencia

Gestión de las Incidencias

La gestión de incidencias es el proceso seguido para responder a eventos no planificados o interrupciones de servicio, con el fin de recuperar el estado operativo, garantizando que el funcionamiento normal del servicio se recupere lo antes posible y se minimice el impacto en el negocio [15].
La gestión de incidentes sigue el siguiente proceso *(Véase la figura 3.3)*:

1. Identificar el incidente (*ticket*) y registrarlo. Los campos más habituales a la hora de registrar un incidente son:
   - Nombre de la persona que lo notifica.
   - Fecha y hora a la que se notifica el incidente.
   - Descripción del incidente, haciendo hincapié en las funciones afectadas.
   - Numero de identificación único asignado al incidente, que se utilizará para realizar el seguimiento.
2. Categorizar el incidente. Asignar a cada incidente una categoría lógica e intuitivo dentro del *Service Desk*.
3. Asignar una prioridad al incidente basándose en el impacto al negocio, numero de personas afectadas y la seguridad. En caso de duda establecer siempre la prioridad más alta.
4. A la hora de dar una respuesta se dan los siguientes pasos:
   4.1 El agente del *Service Desk* realiza una primera suposición sobre la causa del error para empezar empezar a resolverlo basándose en las bases de conocimiento y manuales. Se consigue solucionar el flujo termina aquí pasa directamente a cerrar el incidente, si no, el incidente se escalará.
   4.2 Si el agente del *Service Desk* no ha sido capaz de solucionarlo, se recopilará toda la información que se disponga y se pasará a otro miembro de la organización con conocimientos más técnicos, para que se pueda diagnosticar correctamente el problema.
   4.3 Una vez se llegue a un diagnóstico correcto se darán los pasos necesarios para resolver el incidente.
   4.4 Si el incidente se había escalado se vuelve al *Service Desk* para cerrar la incidencia, ya que solo los miembros del equipo del *Service Desk* pueden cerrar las incidencias.

Algunas de las ventajas que tiene la gestión de incidentes para una organización son:

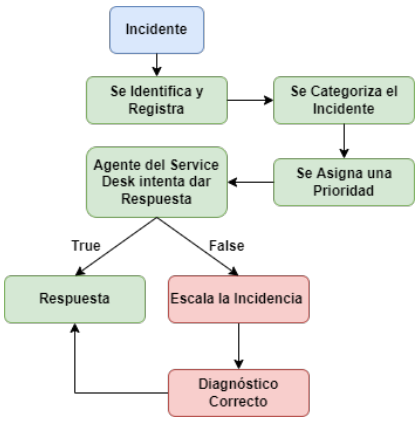


Figura 3.3: Proceso Gestión de Incidencias

- Resolución de incidentes más rápida.
- Menor pérdida de ingresos, lo cual es importante por que en la actualidad como demuestran algunos estudios [5] se pueden perder hasta 300.000$ por hora que el servicio este interrumpido.
- Mejora de la comunicación interna y externa durante los incidentes.
- Mejora y aprendizaje continuo.

3.1.2. Principales Herramientas

En esta subsección hablaremos sobre las principales herramientas que utilizan los sistemas de *ITSM* para encargarse de las solicitudes de asistencia y de las incidencias.

Centro de Ayuda

Un centro de ayuda, también conocido como *Help Desk*, es un equipo centralizado dentro de una empresa que atiende a empleados o clientes en masa, utilizando un producto de software para organizar las peticiones de servicio o *tickets*. Representa el soporte técnico nivel 1, etapa inicial de servicio, encargada de resolver problemas más simples y comunes.

Podemos diferenciar entre 3 tipos de *Help Desk* diferentes:

- ***Help Desk* para el soporte *TI***. Destinado a ayudar al personal interno de la empresa a resolver incidentes, abarcando desde cambiar una contraseña hasta restaurar una interrupción del servicio.
- ***Help Desk* dedicado a la atención al cliente**. Dedicado a los clientes externos de la organización. Responde preguntas sobre los productos y servicios que han adquirido los clientes y ayuda a resolver los problemas que se puedan tener con ellos.
- ***Help Desk* para empresas**. Son utilizados por los departamentos de Recursos Humanos, Finanzas y Jurídicos. Los *Help Desk* gestionados por personal de Recursos Humanos pueden ayudar a los empleados a cambiar opciones de su nómina y obtener informes de historial laboral, los del Departamento Jurídico pueden responder a solicitudes de revisión de contratos y los de Finanzas puede responder a preguntas sobre cuentas por pagar y gastos derivados.

Las finalidades que tiene la utilización de un *Help Desk* son las siguientes:

- **Punto de contacto único**. Permite que los clientes, tanto los internos como los externos saber donde acudir en caso de necesitar ayuda.
- **Responder preguntas**. Los clientes pueden ponerse en contacto con un agente del *Help Desk* cuando necesitan respuestas o instrucciones paso a paso.
- **Ahorrar tiempo**. Cuando un *Help Desk* esta bien gestionado aglutina el conocimiento y proporciona orientación con flujos de trabajo para agilizar y facilitar la solución de los problemas de los clientes.
- **Medir la satisfacción del cliente.** Después de cada petición realizada, los clientes pueden proporcionar un *feedback*, que permite conocer el grado de satisfacción del cliente y mejorar los procesos, bases de conocimientos y las soluciones que se ofrecen.

Las funcionalidades que permite utilizar un software de *Help Desk* son:

- **Soporte de correo electrónico**. Permite enviar notificaciones de las incidencias (*tickets*) a una dirección de correo específica.
- **Portal de autoservicio**. Es un espacio intuitivo en el que los clientes pueden enviar preguntas en forma de *tickets* al *Help Desk* u obtener respuestas inmediatas gracias a una base de conocimientos.
- **Informes y análisis**. Permite generar informes sobre unas métricas claves, como la productividad de los agentes, la satisfacción del cliente y los costes de soporte. Estos análisis detallados son utilizados para mejorar continuamente la calidad y la eficiencia del servicio.
- **Automatización.** El *software* del centro de ayuda puede automatizar tareas recurrentes para ahorrar tiempo a los agentes del *Help Desk*. También se puede automatizar el cierre de *tickets* inactivos o avisar a un gestor cuando surja una incidencia de máxima prioridad.
- **Gestión de *SLA* y *SLO***. Los acuerdos del nivel de servicio (*SLA*) son un contrato entre una organización y un cliente, mientras que los objetivos de nivel de servicio (*SLO*) son los objetivos descritos en el *SLA*, que definen los niveles de servicio que tienen que satisfacer los agentes del *Help Desk*, como el tiempo hasta la primera respuesta o el tiempo de resolución.
- **Personalización**. Permite que las empresas personalicen su *Help Desk* para que lleve los colores y el logotipo de la empresa.

Un ejemplo de *Help Desk* que podemos encontrar en nuestra vida cotidiana podrían ser las páginas de *FAQ (Frequently Asked Questions)* y los *chatbots*.

Centro de Asistencia

Según *ITIL* un centro de asistencia, también conocido como *Service Desk*, es el punto de contacto único entre el proveedor de servicios *TI* y los clientes. Gestiona los incidentes, las solicitudes de servicio y la comunicación con los clientes. Se puede entender como una evolución del *Help Desk*, ya que incluye todas las funciones del *Help Desk*. También incluye otros módulos que hacen que forme parte de la gestión de: problemas, cambios, conocimiento y activos. Están diseñados para mejorar las operaciones y gestiones internas de la organización, facilitando el crecimiento de la empresa.

3.1.3. Comparativa

Aunque los *Service Desk* y los *Help Desk* tienen algunas características similares, podemos discernir una serie de diferencias claras [16].

Help Desk	Service Desk
Incluye un sistema de gestión de incidencias y capacidades de autoservicio.	Incluye módulos para la gestión de incidentes, problemas, cambios, conocimientos y activos.
Es táctico y reactivo.	Estratégico y proactivo
Se centra en resolver los problemas de los usuarios finales.	Se centra en la estrategia de servicio a largo plazo.
Enfocado en la de reparación de fallas.	Tiene un enfoque holístico que está alineado con los objetivos comerciales.
No suele necesitar mucho personal	Requiere más personal

Tabla 3.1: *Help Desk* vs *Service Desk*

La mayor diferencia entre *Help Desk* y *Service Desk* es que un *Help Desk* es literalmente un subconjunto de un *Service Desk*. *Service Desk* ofrece una gama más amplia y servicios más complejos. Un *Service Desk* puede ser utilizado como un *Help Desk*, pero no al revés.

Help Desk es una solución independiente que realiza tareas relacionadas con la gestión de *tickets* y ofrece funcionalidad de autoservicio. Los *Service Desk* son un sistema mucho más complejo y esta integrado junto a otros procesos de *ITSM* como la gestión de cambios, de problemas, de activos y del conocimiento.

Los *Help Desk* reaccionan ante las incidencias cuando un usuario registra un *ticket* para solucionar el problema, esto quiere decir que son de naturaleza reactiva. Por otra parte los *Service Desk* se encargan de garantizar de manera proactiva que las operaciones de *TI* se ejecuten sin problema.

3.2. DESARROLLO WEB

En esta sección se comentará los principales aspectos del desarrollo web, los diferentes tipos de arquitectura utilizadas en la actualidad y las tecnologías web utilizadas para desarrollar el *TFG*.

3.2.1. Visión General

Podríamos definir brevemente el desarrollo web como el proceso de creación de sitios web para internet o una intranet. Utilizando tecnologías del lado del servidor, que sirven para crear servicios y dar funcionalidad de valor a la web y tecnologías del lado del cliente, que sirven para dar una buena apariencia y conseguir que el usuario tenga una experiencia agradable.

Podemos distinguir 3 tipos de desarrolladores web:

1. **Desarrollador *Front-End***. Son los encargados crear y diseñar la web, incluyendo la maquetación, comportamiento e interacción con el usuario. Este tipo de desarrolladores están especializados en lenguajes como *HTML*, *CSS* y *JavaScript*.
2. **Desarrollador *Back-End***. Son los encargados de crear las funcionalidades de la página web, y de que el almacenamiento y uso de los datos tenga un desempeño correcto, es decir, de todo lo que el usuario no ve.

Están especializados en bases de datos, servidores y lenguajes de programación como *NodeJS, Python, Ruby, PHP, Visual Basic .Net* y *Java*. Y en lenguajes de bases de datos como *MySQL, SQL Server, MsSQL* o *PLSql*.

3. **Desarrollador *Full-Stack***. Son los desarrolladores que se encargan de ambas partes, tanto del Front-End como del Back-End.

En este *TFG* el equipo de desarrollo esta conformado por desarrolladores *Full-Stack*.

3.2.2. Arquitecturas Web

Podemos definir la arquitectura de una aplicación web como el esquema de interacciones simultáneas entre los componentes, las bases de datos, los sistemas *middleware*, las interfaces de usuario y los servicios de una aplicación. Tradicionalmente, en una arquitectura web se pueden distinguir los siguientes componentes:

- **Client-Side**. Es conocido como *Front-End*, está escrito en lenguajes como *HTML, CSS* o *JavaScript* y se almacena dentro del navegador. Es donde da lugar la interacción con el usuario.
- **Server-Side**. También conocido como *Back-End* controla toda la lógica de negocio y responde las peticiones *HTTP*. Suele estar escrito en lenguajes como *NodeJS, Java, PHP, Ruby* y *Python*.
- **Base de Datos**. Contiene los datos que gestiona el *Back-End*.

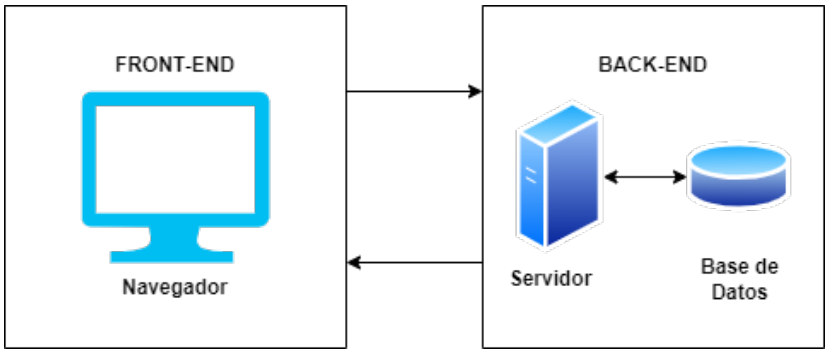


Figura 3.4: Componentes Principales Arquitectura Web

Una vez conocido los componentes básicos de la arquitectura de una aplicación web analizaremos los tipos de arquitecturas más utilizadas actualmente.

Arquitectura en capas

Es un tipo de arquitectura tradicional que suele utilizarse para diseñar aplicaciones empresariales, que por lo general están ligadas a aplicaciones heredadas.

En esta arquitectura, hay varias capas o niveles (normalmente 3) que componen la aplicación, y cada una cumple una función particular. Estás capas ayudan a gestionar las dependencias y a ejecutar funciones lógicas. Las capas se organizan de forma horizontal, lo que quiere decir que una capa solo puede acceder a los recursos de la que está inmediatamente debajo de ella, o de cualquiera de las inferiores.

Arquitectura Monolítica

Este tipo de arquitectura también suele estar ligada a los sistemas heredados, son pilas de aplicaciones únicas que contienen todas las funciones dentro de cada aplicación.

Tienen un fuerte nivel de acoplamiento por lo que al actualizar o ajustar un solo aspecto en este tipo de aplicación tendrá repercusiones en toda la infraestructura subyacente. Un solo cambio en el código implica volver a desplegarla por completo.

Arquitectura de Microservicios

No es solo un tipo de arquitectura web, puesto que también se puede utilizar para abordar la forma en que se escribe el software. Consiste en dividir las funcionalidades de las aplicaciones en elementos más pequeños que son independientes entre sí. A cada uno de estos elementos lo llamaremos microservicio [4].

Estos microservicios se encuentran distribuidos y tienen un nivel bajo de acoplamiento, es decir, no influyen en los demás.

El objetivo de esta arquitectura es proporcionar un software de calidad y con rapidez, puesto que es posible que los desarrolladores puedan trabajar en varios microservicios de manera individual, sin tener la necesidad de estar actualizando toda la aplicación después de realizar los cambios.

Al comparar la arquitectura de una aplicación monolítica con una arquitectura de microservicios *(Véase la figura 3.5)* podemos discernir las siguientes ventajas:

- **Agilidad**. El poder trabajar en varios microservicios de forma paralela permite desarrollar aplicaciones de calidad rápidamente, mientras que en un arquitectura monolítica cada vez que se desarrolla una nueva función habría que actualizar toda la infraestructura, dificultando el trabajo en paralelo.
- **Recuperación**. Al ser servicios independientes, no afectan a los demás. Por lo que si un servicio falla, este no afectará a toda la aplicación, mientras que las aplicaciones monolíticas el fallo sería catastrófico.
- **Escalabilidad**. A medida que una aplicación demanda más servicios, dividir las implementaciones en distintos servidores o infraestructuras para satisfacer las necesidades. Mientras que en una arquitectura monolítica no es posible dividir el software implementado.
- **Accesibilidad**. Puesto que las aplicaciones se desglosan en servicios más pequeños, los desarrolladores pueden comprender, actualizar y mejorar estos servicios más fácilmente, lo que también aumenta la rapidez de los ciclos de desarrollo. En una arquitectura monolítica, habría que comprender el funcionamiento de la toda la aplicación en conjunto, perjudicando a los desarrolladores que lleven poco tiempo en el proyecto.
- **Compra**. Es posible comprar o vender un microservicio a terceros, en el caso de comprar un servicio, se ahorraría el tiempo de crear un servicio que ya existe y se sabe que funciona. Mientras que si lo vendemos aumentaría la reputación de nuestra empresa aparte que el beneficio económico que nos llevamos. En una arquitectura monolítica esto no sería posible, puesto que tienen un nivel de solapamiento demasiado alto.

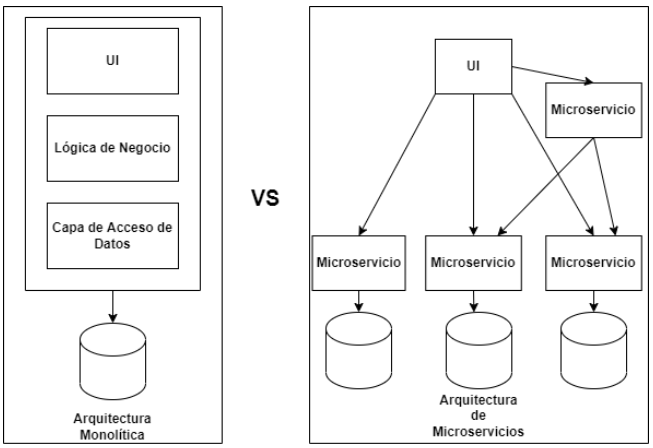


Figura 3.5: Arquitectura Monolítica VS Arquitectura de Microservicios

Arquitectura Orientada a Servicios *(SOA)*

Es un tipo de arquitectura que se asemeja bastante a la arquitectura de microservicios. La *SOA* estructura las aplicaciones en servicios independientes y reutilizables, que se comunican a través de un bus de servicios empresariales *(ESB)*.

Las aplicaciones que utilizan este tipo de arquitectura, basan su funcionamiento en que una aplicación *Front-End*, es decir una Interfaz de usuario *(UI)* que accede a un conjunto de servicios integrados a través de un *ESB* proporcionando valor a una empresa o cliente.

Realizando una comparación entre la arquitectura de microservicios y *SOA (Véase la figura 3.6)* llegamos a la conclusión de que la principal diferencia entre ambas arquitecturas es que, en la arquitectura de microservicios, los microservicios pueden comunicarse entre sí, generalmente sin estado, para que las aplicaciones diseñadas de esta manera sean más tolerantes a los errores y menos dependientes del *ESB*. Por lo demás, *SOA* mantiene las mismas ventajas que la arquitectura de microservicios sobre la arquitectura monolítica.

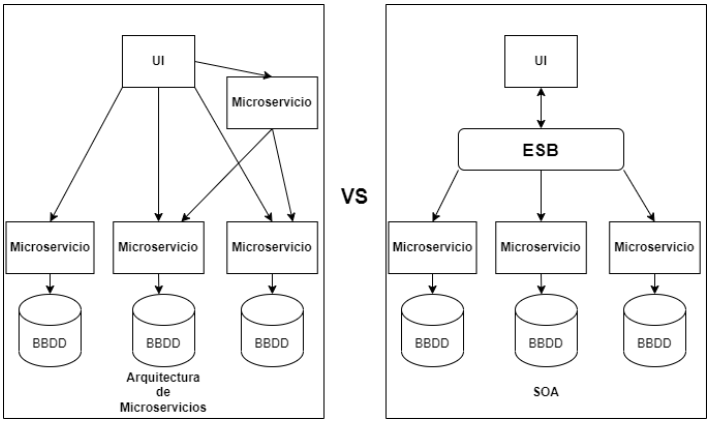


Figura 3.6: Arquitectura de Microservicios VS *SOA*

3.2.3. Tecnologías Web

A continuación, se expondrán algunas de las tecnologías utilizadas para el desarrollo web más frecuentes en la actualidad.

SpringBoot

SpringBoot es un módulo gratuito del *framework Spring*, fue creado para simplificar el desarrollo de aplicaciones basadas en *Spring Framework* bajo licencia *Apache* 2.0. El proyecto de *Spring* sale a la luz por primera vez en 2003, aunque no fue hasta 2014 donde apareció la versión 1.0 del modulo *SpringBoot*, actualmente tenemos la versión 2.4.0 que da soporte a *Java* 16 [3].

***SpringBoot* funciona sobre la *JVM* (Java Virtual Machine)**, lo que permite desarrollar nuevas aplicaciones, migrar y reutilizar código de cual proyecto que ya tengamos desarrollado en *Java*, *Kotlin* o *Grovy* sobre uno de los entornos mas extendido y potente en la actualidad como es *JVM*. Que garantiza que toda aplicación escrita en *Spring*, y *Java* en general, pueda ser ejecutada en los principales sistemas operativos, tanto de servidores, como máquinas virtuales o en computadoras personales.

***SpringBoot* permite hacer aplicaciones web y microservicios**, actualmente, la mayoría de las empresas están migrando todas sus aplicaciones a una arquitectura web basada en microservicios, o *Startups* que utilizan directamente está arquitectura, gracias a la gran flexibilidad que presenta la separación de responsabilidades en microservicios independientes permitiendo crear una aplicación especializada, escalable y fácilmente reutilizable.
Por otra parte también puede utilizarse *SpringBoot* para realizar aplicaciones web tradicionales o multipágina en donde el *HTML* es generado dinámicamente en el lado del servidor, antes de ser entregado al navegador del usuario.
Esta gran versatilidad permite que podamos afrontar prácticamente cualquier reto en el ámbito profesional de la programación.

En *SpringBoot* es rápido a la hora de desarrollar, puesto nos ahorra el tener que configurar contenedores y servidores web de *Apache Tomcat*. Además, tenemos toda la estructura de nuestra aplicación autoconfigurada en nuestro *framework*. Todas las dependencias estarán alojadas en un formato legible en un archivo *POM* para su gestión en *Maven*.

***SpringBoot* es rápido a la hora de ejecutar**, puesto que al estar basado en el *framework Spring*, todas las aplicaciones que creemos estarán optimizadas para realizar altas cargas de trabajo consumiendo una cantidad mínima de memoria *RAM*. Gracias a que la mayoría de entidades que utiliza están basadas en el patrón *Singleton*, por lo que estos objetos serán reutilizados, evitando tener que crear y mantener objetos en memoria.

NodeJS

NodeJS es un entorno de ejecución de un solo hilo, de código abierto y multiplataforma utilizada para crear el *Back-End* de aplicaciones web rápidas y escalables. Se ejecuta en el motor de ejecución de *JavaScript V8*, utiliza una arquitectura de *E/S* basada en eventos sin bloqueo, lo que lo hace perfecto para desarrollar aplicaciones en tiempo real. Esta escrito en *C, C++ y JavaScript*.

Utiliza la arquitectura *Single Threaded Event Loop* para manejar las peticiones de múltiples clientes al mismo tiempo [7].

En la que una petición cualquiera seguiría los siguientes pasos para ser procesada *(Véase la figura 3.7)*:

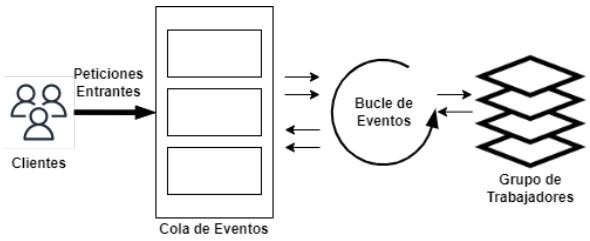


Figura 3.7: Arquitectura de *NodeJS*

1. Cuando llega una solicitud, *NodeJS* la coloca en la Cola de eventos.
2. El Bucle de Eventos, de un solo hilo queda a la espera de peticiones de forma indefinida.
3. Cuando llega una solicitud a la Cola de Eventos, el Bucle de Eventos la recoge y comprueba si es necesario una operación de *E/S* de bloqueo. En este punto las peticiones pueden seguir 2 caminos posibles:
    - **No requiere bloqueo**. Si la petición no requiere realizar un acción bloqueante, el Bucle de Eventos procesa la solicitud y envía una respuesta.
    - **Requiere bloqueo**. Si la petición requiere una acción bloqueante el Bucle de Eventos asigna uno de los hilos internos limitados para que procese la solicitud. Este grupo de hilos internos es conocido como Grupo de Trabajadores.
4. El Bucle de eventos registra las solicitudes que se bloquean y las coloca en la Cola de Eventos en lugar de permitir que la tarea se bloqueé. Así es como mantiene su naturaleza no bloqueante.

Algunas de las características que ofrece son las siguientes:

- **Facilidad de Uso**. Es muy fácil iniciarse en su uso, por lo que es una buena opción para desarrolladores novicios. Cuenta con una gran comunidad y muchos tutoriales que ayudan en el proceso.
- **Escalable**. Al trabajar con un solo hilo de ejecución *NodeJS* es capaz de manejar un gran número de conexiones simultáneas, manteniendo un alto rendimiento.
- **Velocidad**. La ejecución de hilos sin bloqueo hace que *NodeJS* sea rápido y eficiente.
- **Paquetes**. Existen una gran cantidad de paquetes de código abierto que simplifican el trabajo de los desarrolladores. Además la instalación de estos paquetes es sencilla mediante el instalador de paquetes *NPM*.
- **Multiplataforma**. *NodeJS* permite crear sitios web *SaaS*, aplicaciones de escritorio, microservicios e incluso aplicaciones móviles.
- **Mantenible**. Es una opción fácil, puesto que el *Front-End* y el *Back-End* pueden gestionarse con *JavaScript* como único lenguaje.

Express

Express es un *framework* gratuito y de código abierto más popular de *NodeJS*. Fue lanzado inicialmente en Noviembre de 2010, en la actualidad se encuentra en la versión 4.17.1 de la *API*. Proporciona mecanismos para:

- Escribir manejadores de peticiones con los diferentes verbos *HTML* en diferentes *URL*.
- Integración con motores de renderización de vistas para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web, como que puerto usar, donde se localizan las plantillas que se van a utilizar para renderizar la respuesta.
- Permite procesar peticiones *middleware* adicional en cualquier punto dentro del manejo de la petición.

Aunque *Express* es un *framework* minimalista, los desarrolladores han creado paquetes de *middleware* compatibles para abordar casi cualquier problema de desarrollo web.

Existen muchos tipos de librerías, que permiten trabajar con *cookies*, sesiones, inicios de sesión del usuario, parámetros *URL*, datos de una petición *POST*, cabeceras de seguridad o configuración de variables de entorno.
Esta considerado un *framework* no dogmático, es decir, es transigente, puesto que permite insertar casi cualquier *middleware* compatible que necesitemos utilizar a la hora de manejar una petición, en el orden que nos resulte más cómodo. También permite estructurar la aplicación en cualquier tipo de estructura de directorios [7].

3.3. COMPUTACIÓN EN LA NUBE

En esta sección, hablaremos sobre Computación en la nube, también conocida como *Cloud Computing*, así como de sus diferentes tipos, herramientas, modelos de negocio, ventajas y finalmente realizaremos un análisis de sus principales plataformas.

3.3.1. Conceptos Fundamentales

La computación en la nube es una tecnología que permite acceder desde cualquier lugar del mundo a una serie de recursos informáticos, entre los que se encuentran el almacenamiento, la potencia de procesamiento, las bases de datos, las redes, herramientas de análisis, inteligencia artificial y aplicaciones software a través de Internet [2].

Características

- **Autoservicio Bajo Demanda.** Los usuarios pueden hacer que se generen recursos informáticos para casi cualquier tipo de carga de trabajo bajo demanda. Lo que ahorra a las organizaciones tener un administrador *TI* que administren los recursos informáticos.
- **Elasticidad.** Las organizaciones pueden escalar los recursos informáticos tanto como necesiten para realizar sus funciones.
- **Pago por Uso.** Permite a las organizaciones pagar por los recursos y carga de trabajo de utilizan.
- **Capacidad de Carga de Trabajo.** Los proveedores de computación en la nube suelen implementar sus recursos de forma redundante, lo que asegura un almacenamiento consistente y que las cargas de trabajo importantes de los usuarios se distribuyan a través de múltiples regiones globales.

- **Flexibilidad de Migración.** Las organizaciones pueden movilizar cargas de trabajo desde o hacia la nube para cubrir los nuevos servicios que les puedan surgir.

3.3.2. Funcionamiento y Clasificación

La computación en la nube utiliza un servidor remoto que conecta los dispositivos de los usuarios a los recursos informáticos centralizados. Un servidor remoto mantiene el registro de los datos y programas que necesita el usuario. Permitiendo acceder a los recursos a través de Internet, aunque estén almacenados en la otra punta del mundo.
Según el tipo de implementación podemos discernir la siguiente clasificación [1]:

- **Nube Pública.** En este tipo de nube los recursos informáticos los proporciona un proveedor de servicios en la nube como *OCI o AWS* a cualquier persona que desee contratarlos. El cliente se hace responsable de lo que envíe a la nube y el proveedor se encarga de la mantenibilidad, gestión y seguridad de todos los recursos.
- **Nube Privada.** A diferencia de la anterior, mantiene la infraestructura de la nube en su dominio interno restringiendo el acceso a usuarios autorizados, lo que aumenta la seguridad. Además puesto que esta nube se diseña exclusivamente para una organización puede personalizar ciertas funciones y soporte para sus necesidades.
- **Nube Híbrida.** Es la combinación de la nube privada y pública permitiendo que se compartan datos y aplicaciones entre ellas. Los servicios y aplicaciones confidenciales se mantienen más seguros en la nube privada, mientras que los servidores web de acceso público y los *endpoints* orientados a los clientes residen en la nube pública.
- **Nube Comunitaria.** Consiste en que varias empresas u organizaciones reúnen un pool sus recursos en la nube para resolver un problema común.

3.3.3. Modelos de Negocio

Actualmente la computación en la nube ofrece una amplia variedad de servicios, herramientas y funcionalidades para atender las necesidades de las empresas y organizaciones. Podemos distinguir los siguientes modelos:

- **Infraestructura como Servicio *(IaaS).*** Es una capa básica de servicio en la nube que permite que las organizaciones alquilen la infraestructura *TI* (servidores, redes, sistemas operativos, almacenamiento) de un proveedor de servicios en la nube. Permite incluso reservar máquinas preconfiguradas para tareas especializadas como equilibradores de carga, bases de datos y servidores de correo. Destinado normalmente a administradores de sistemas.
- **Plataforma como Servicio *(PaaS)***. Es una extensión de *IaaS*, que además de la infraestructura *TI* incluye varias herramientas y software que los desarrolladores necesitan para construir, probar y ejecutar sus aplicaciones sin tener que preocuparse por gestionar sistemas operativos, bases de datos y otras herramientas de desarrollo. Destinado normalmente a desarrolladores.
- **Software como Servicio *(SaaS)***. Proporciona un producto software completo, el cual es ofrecido por el proveedor. En este modelo tampoco hay que preocuparse por la infraestructura,

nos centramos en consumir el servicio a través de internet o desde cualquier dispositivo capaz de utilizar un navegador Web. Destinado normalmente a usuarios finales.

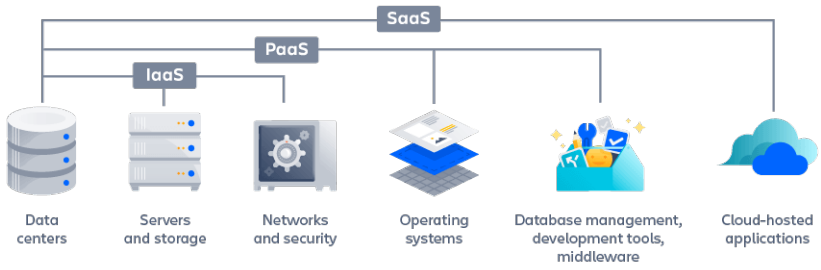


Figura 3.8: Modelos de Negocio (Imagen obtenida del libro [2])

3.3.4. Ventajas de la Computación en la Nube

Algunas de las ventajas técnicas y empresariales que ofrece a las empresas y organizaciones son [10]:

- **Costes.** Evita a sus usuarios el gasto de invertir masivamente en el hardware y software necesarios, permitiendo escalar automáticamente tus servicios en función de la demanda gastando únicamente por el uso que se haga. También permite probar nuevas tecnologías sin el coste y riesgo que suele suponer a las organizaciones.
- **Accesibilidad.** Permite que sus usuarios puedan trabajar desde cualquier lugar y en cualquier dispositivo mientras tenga una conexión a internet, esta ventaja nunca ha sido tan importante en un contexto actual como es el del *Covid-19*.
- **Seguridad.** Al utilizar modelos como *IaaS* y *SaaS* el software que utilizamos será actualizado automáticamente por el proveedor de servicio, mejorando la seguridad interna de nuestros sistemas. También se disponen de una serie de herramientas de vanguardia, tales como:
  - **Protección automática frente ataques *DDoS***.
  - **Cortafuegos de última generación**.
  - **Consola de Gestión de Acceso a la Identidad**. Ofrecen una consola a través de la cual se pueden gestionar y controlar el acceso a qué y que puede acceder cada identidad de nuestro servicio en la nube.
  - **Cifrado Automático**. La nube cifra por defecto todos los datos que gestiona, normalmente utilizan *AES de 256 bits*.
- **Velocidad.** Los recursos informáticos que podemos acceder en la nube son casi ilimitados, dedicar estos recursos en el procesamiento de cargas de trabajo complejas, en vez de tardar semanas pueden pasar a minutos.

  Gracias a la internacionalización de los centros de datos de los proveedores, ofrecen a sus usuarios una baja latencia en el uso de sus servicios, proporcionando una experiencia de usuario consistente.
- **Recuperación.** Los datos de la organización dejan de almacenarse en los dispositivos locales, se encuentran en la nube, con una serie de copias de seguridad que actualizan su contenido en

tiempo real. Lo que ayuda a la recuperación frente a ataques *malware*, desastres naturales y errores del software, hardware o humanos.

- **Sostenibilidad.** Con el estado actual del medioambiente los proveedores de servicio (como *Oracle, Amazon, Google* y *Microsoft*) han empezado a invertir en el uso de energía limpia previendo que la energía utilizada para mantener sus instalaciones procederán al 100 % de fuentes verdes en 2025, por lo que utilizar la nube reduce el impacto medioambiental y la huella de carbono de nuestra organización.
- **Integración Continua y Entrega Continua.** La integración continua y entrega continua (*CI/CD*) es una de las prácticas mas recomendadas para los profesionales de *DevOps* incrementan la velocidad de los equipos y a agilizar el lanzamiento del producto *software* al mercado. Esta compuesto por 2 conceptos diferentes:
  - **Integración Continua (*CI*)**. Consiste en la automatización de la compilación, pruebas y empaquetación de sus aplicaciones de una manera sencilla y recurrente. Permitiendo que los cambios de código se fusionen de forma fiable en un repositorio central. También ayuda a optimizar el tiempo que se tarda en realizar cambios en el código, permitiendo que los desarrolladores se centren en mejorar el *software*
  - **Entrega Continua (*CD*)**. Es la entrega automatizada con los códigos completos del *software* a entornos de pruebas y desarrollo. Dado que cada paso esta automatizado en el proceso de compilación, la liberación del código se realiza de forma segura. También existe el concepto de implementación continua, que es el siguiente paso de la entrega continua, que provoca que cuando un cambio pase las pruebas automatizas se coloco automáticamente en producción.

  En definitiva podemos ver en la figura 3.9, el objetivo de *CI/CD* es crear una cadena que involucre la menor cantidad de trabajo humano posible, evitando el trabajo manual innecesario y el error.

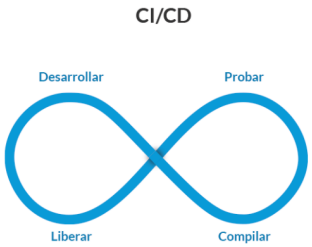


Figura 3.9: Cadena *CI/CD* (Imagen obtenida de tecnova [17])

3.3.5. Plataformas de Computación en la Nube

Existen varias plataformas y proveedores de computación en la nube. Las más conocidas actualmente son: *Microsoft Azure, Google Cloud, Amazon Web Service, y Oracle Cloud*. En este apartado hablaremos sobre la plataforma *Oracle Cloud*, ya que ha sido la plataforma utilizada para desarrollar este *TFG*.

Oracle Cloud

Es un servicio de computación en la nube ofrecido por *Oracle Corporation*, que proporciona servidores, almacenamiento, redes, aplicaciones y servicios a sus usuarios a través de internet. *Oracle Cloud* puede desplegarse tanto en la nube pública, como la privada y la híbrida.

Oracle Cloud proporciona infraestructura como servicio *IaaS*, plataforma como servicio *PaaS* y software como servicio *SaaS*. Los cuales se pueden utilizar para crear, implementar, integrar y extender aplicaciones en la nube.

Admite numerosos estándares abiertos (como *SQL, HTML5 o REST*), soluciones de código abierto (como *Kubernetes, Apache Hadoop o Apache Kafka*) y una gran variedad de lenguajes de programación, bases de datos, herramientas y *frameworks*, además de los específicos de *Oracle*. [6]

Oracle Cloud puede dividirse en 3 en función del tipo de servicio que proporciona a los usuarios:

1. **Oracle Cloud Infraestructure**. Es el nombre que ha dado *Oracle* a su *IaaS* entre la que se cuentan servicios de:
    - **Compute.** Proporciona instancias de diferentes tipos de máquinas virtuales, adaptándose a las diferentes de cargas de trabajo y características de rendimiento. Entre los que se incluyen servicios como *Container Engine for Kubernetes*, *Container Registry* y *Cloud Shell*.
    - **Almacenamiento.** Proporciona volúmenes de bloques, almacenamiento de objetos y archivos con la capacidad para habilitar bases de datos, análisis y contenido.
    - **Redes.** Proporciona una red con dirección *IP*, subredes, enrutamiento y *firewalls* completamente configurables.
    - **Base de Datos**. Permite que las bases de datos de *Oracle* se implementen bajo demanda en un entorno de nube con *Real Application Clusters*, seguridad de datos y controles granulares.
    - **Balanceadores de Carga**. Permite equilibrar la carga para enrutar automáticamente el tráfico a través de los dominios existentes, garantizando una alta disponibilidad y una gran tolerancia a fallos en las aplicaciones que aloja.
    - **Servicios *Edge***. Permite monitorear la ruta entre los usuarios y los recursos, lo que permite adaptarse a los cambios e interrupciones utilizando una infraestructura *DNS* segura.
    - ***FastConnect***. Garantiza una conexión privada a través de redes locales y en la nube. A través de proveedores como: *Equinix, AT&T* y *Colt*.
2. **Oracle Cloud Plataform**. Es el nombre que ha recibido el *PaaS* de *Oracle*. Entre los que se incluyen los siguientes servicios para:
    - **Gestión de Datos**. Ofrece una plataforma de gestión de datos para cargas de trabajos de gran escala como *Big Data*, *Machine Learning* y análisis de imágenes. La plataforma permite desplegar bases de datos *Oracle, MySQL* y *NoSQL* bajo demanda como servicios gestionados en la nube.
    - **Desarrollo de Aplicaciones**. La nube de *Oracle* ofrece plataformas de desarrollo de aplicaciones abierta basada en estándares y compatible con contenedores en nube para

construir, implementar, y administrar las aplicaciones en la nube. Entre sus servicios se incluyen *Java*, asistentes digitales, *Application Container Cloud*, *Developer Cloud, Visual Builder Studio, API Catalog y blockchain.*

- **Integración**. Ofrece plataformas que ayudan con la integración y replicación de datos, administración de *API* y para la integración y migración de datos. Alguno de los servicios más conocidos para la integración son: *data integration platform cloud, data integrator cloud service, GoldenGate cloud service, integration cloud, API platform cloud service, apiary cloud service* y *SOA cloud service.*
- **Análisis de Negocio**. Proporciona servicios capaces de analizar y generar información a partir de los datos de diferentes orígenes. Los servicios más reconocidos son: *analytics cloud, business intelligence, big data discovery, big data preparation, data visualization* y *essbase.*
- **Seguridad**. Proporciona servicios de identidad y seguridad que garantizan un monitoreo y acceso seguro en la nube híbrida a la par que cumple los requisitos establecidos por el gobierno para las *TI*. Todo esto gracias a los servicios de *Identity Cloud Service* y *Cloud Access Security Broker Cloud Service.*
- **Administración**. Proporciona una plataforma integrada para la supervisión, gestión y análisis. Utilizada para mejorar la estabilidad de *TI*, evitar interrupciones del servicio, mejorar *DevOps* e incrementar la seguridad. Algunos de los servicios más reconocidos son: Application *Performance Monitoring, Log Analytics, Orchestration, IT Analytics, Configuration and Compliance* y *Security Monitoring.*
- **Gestión de Contenido**. Es una plataforma para la gestión de contenidos, sitios web y flujos de trabajo. Lo ofrece mediante los servicios de *Content Management, WebCenter Portal Cloud Service* y *Oracle DIVA Cloud.*

3. **Oracle Cloud Applications**. Es el nombre que ha recibido el *SaaS* de *Oracle*. Ofrece una gran variedad de productos en sectores industriales con varias opciones de despliegue cumpliendo con los estándares. Algunos de sus servicios más conocidos son: *Oracle Customer Experience, Oracle Human Capital Management,Oracle Supply Chain Management, Oracle Internet of Things Applications, Block-Chain Cloud Service.*

A continuación nos detendremos brevemente en explicar los servicios de *Oracle Cloud* utilizados a lo largo del desarrollo de este *TFG*:

- **Streaming Service**. Es un servicio totalmente gestionado y escalable que permite procesar flujo de datos de gran volumen en tiempo real. Es el equivalente *Oracle* de *Apache Kafka*, de hecho la *API* de *Kafka* y la de *Streaming* son compatibles, es decir, podemos utilizar aplicaciones escritas en *Kafka* con *Streaming* sin tener que modificar el código.
  Una ventaja que tiene *Streaming* sobre *Kafka* es que al estar ofrecido como un *PaaS* no hace falta preocuparse sobre el mantenimiento.

  En la figura 3.10 podemos ver como está estructurado *Streaming*, a continuación explicaremos los principales elementos que la componen y su flujo de funcionamiento.

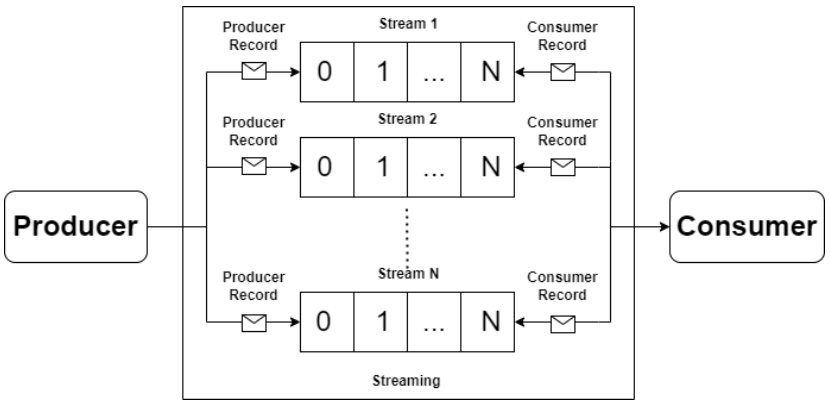


Figura 3.10: Estructura *Streaming*

- **Stream**. Es el equivalente a un *topic* en *Kafka*. Es una cola de mensajes, compuesta de 1 o N particiones, en la que los productores pueden depositar sus mensajes. *(Véase la figura 3.11)*

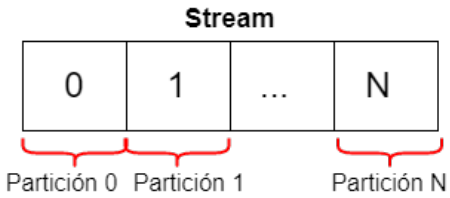


Figura 3.11: Estructura de un *Stream*

- **Stream Pool**. Es una agrupaciones lógicas para los *Streams*. Todos los *Streams* pertenecen a una única *Stream pool*, si no se crea un *Streaming* utiliza una por defecto para contener los *Streams*.
- **Partición**. Son las divisiones lógicas que tiene cada cola de mensajes. Cada *Stream* esta compuesto por al menos 1 partición, y el número máximo está limitados por los recursos que se tengan contratados. Los productores pueden utilizar dos políticas para realizar los envíos de los mensajes. La primera, y la utilizada en este *TFG*, es el de programar una lógica y en función a está, enviar el mensaje a una partición. Mientras que la segunda política consiste en dejar que el productor coloque el mensaje en una partición basándose en la *Key* del mensaje.
- **Productor**. Los productores son los encargados de enviar los mensajes a las particiones de una cola de mensajes, a estos mensajes se le llaman *Producer Records*. Estos *Producer Records* están compuestos por los siguientes parámetros:
  - **Stream**. Parámetro obligatorio que indica a que cola de mensajes se va a enviar.
  - **Value**. Parámetro obligatorio, es el contenido del mensaje se va enviar, normalmente un *JSON*.
  - **Key**. Parámetro opcional, que puede ser utilizado por los productores para enviar el mensaje a una partición.
  - **Partición**. Parámetro opcional, que puede ser utilizado para que el productor envíe

el mensaje a la partición de un *Stream* específico.

- **Consumidor**. Los consumidores pueden suscribirse a los *Streams* y obtener los mensajes que lleguen, llamados *Consumer Records*. Los consumidores almacenan la información del último mensaje que han consumido en la partición de cada *Stream* al que están suscritos, nos referimos a esta posición como *offset*. Los *Consumer Records* citados anteriormente, están compuestos por los siguientes parámetros:
  - **Stream**. Parámetro obligatorio, que indica de que cola de mensajes procede.
  - **Value**. Parámetro obligatorio,que contiene el mensaje enviado, normalmente un *JSON*.
  - **Partición**. Parámetro obligatorio, que indica la partición de la que procede el mensaje.
  - **Offset**. Parámetro obligatorio, que indica la posición en la que se encontraba el mensaje consumido.
  - **Key**. Parámetro opcional, que indica la *Key* que contenía el *Producer Record* original.

- **Container Registry**. Es un servicio de registro *Docker* gestionado por *Oracle* que permite a los desarrolladores almacenar, compartir y gestionar las imágenes *Docker*. Normalmente se emplea en conjunto con servicios como *OKE y Visual Builder Studio*.
- **Container Engine for Kubernetes (OKE)**. Es un servicio totalmente gestionado, ajustable y de alta disponibilidad que permite desplegar y gestionar aplicaciones basadas en contenedores en la nube. Permite crear y gestionar *Clusters* de *Kubernetes*, permitiendo escalar nuestros recursos de nuestro sistema de forma sencilla. A continuación explicaré los conceptos más importantes de *Kubernetes*:
  - **Nodo**. Es una máquina física o virtual que esta ejecutando programas. Los recursos asignados a cada nodo, es decir, el número de *CPUs* y la memoria disponible son definidos en el momento en que los nodos son creados.
  - **Kubernetes Cluster**. Es un conjunto de nodos que ejecutan aplicaciones *Docker*. Esta formado por dos tipos de nodos *Nodos Maestros* y *Nodos Trabajadores*.
  - **Nodo Maestro**. Estos tipos de nodos son encargados de mantener el estado deseado del Cluster y asignar las tareas a los Nodos Trabajadores.
  - **Nodo Trabajador**. Están organizados en *pools* de nodos y son los que realizan las tareas asignadas por los Nodos Maestros. Los Nodos Trabajadores que se encuentran en la misma *pool* están comunicados por una subred de nuestra *VCN*.
  - **Pool de Nodos**. Por lo general todos los nodos trabajadores tienen la misma configuración en el cluster de kubernetes, pero las *pool* nos permite crear grupos de máquinas que tengan configuraciones diferentes a las establecidas como base en el cluster.
  - **Pod**. Un *pod* es un grupo de 1 o N contenedores, con recursos de almacenamiento y de red compartidos y la especificación de como ejecutar esos contenedores. Es la unidad más pequeña que se puede despegar en *Kubernetes*.
  - **Deployment**. Es un controlador que permite definir y mantener las características de los *pods* que desplegamos, como puede ser el numero de replicas, imagen *Docker* de la

aplicación que ejecuta o la etiqueta que lo identifica dentro del nodo.

- **Service**. Un servicio es una abstracción que define un conjunto lógico de *pods* y una política para acceder a ellos. Los *pods* que forman esté grupo son seleccionados gracias a la etiqueta especificada en su creación. Permite exponer un *IP* única y puerto de un *pod*, lo que habilita la comunicación entre los participes del grupo dentro de nuestro nodo. Además también nos permite exponer nuestro *pod* como un servicio de red, accesible a través de una *IP* externa.
- **Ingress**. Es un objeto de *Kubernetes* que nos permite exponer las rutas *HTTP y HTTPS* de nuestras aplicaciones fuera de nuestro cluster. Establece una serie de reglas, que permiten elegir que tipo de *pod* debería responder en función de la ruta de la petición y filtrar el acceso a nuestros servicios basándonos en parámetros como la *URL* a la que accedemos o la *IP* desde la que se realiza petición, lo que proporciona seguridad adicional a nuestros servicios.
- **Secrets**. Son objetos que contienen información sensible como contraseñas, *tokens*, o certificados *SSL*, permitiéndonos que esa información no se encuentre en el código de nuestra aplicación.

- **Visual Builder Studio**. Es una plataforma ágil de desarrollo, colaborativa, con gestión de código y automatización de *CI/CD*. Esta automatización es posible debido a que *Visual Builder Studio* permite consumir datos de fuentes *REST*, lo que le permite comunicarse con los servicios Oracle, accediendo *Container Registry* (donde podrá consumir y publicar imágenes de *Docker*) y *Container Engine for Kubernetes* donde podrá desplegar todo tipo de objetos de *Kubernetes*. Todo esto es automatizable gracias a las *builds, jobs y pipeline*.
  - **Builds**. Son procesos que se ejecutan cuando se quiere desplegar una aplicación, normalmente estos procesos se definen en los *jobs* y se ejecutan al realizar un *commit* en el *git* que contiene el código fuente de la aplicación que queremos desplegar.
  - **Jobs**. Secuencia de pasos que sigue nuestro sistema con la finalidad de desplegar una aplicación. Puede solicitar datos e información y realizar cambios tanto en *OKE* como en *Container Registry*.
  - **Pipeline**. Define el orden de ejecución de una serie de *jobs*.
- **Cloud Shell**. Es un terminal web basado en el explorador al que se puede acceder desde la consola de Oracle Cloud. Proporciona acceso a un shell de *Linux* con una *CLI* de *Oracle Cloud Infraestructure*. Tiene varias herramientas útiles para los desarrolladores instaladas ya como: *Docker* y la *kubectl*.

Oracle Cloud esta compuesto por una red global de centros de datos gestionados por *Oracle Corporation*, que distribuye su nube en varias regiones. En cada región, hay por lo menos 3 dominios de disponibilidad. Y dentro de cada dominio de falla contiene un centro de datos independiente con aislamiento de energético, térmico y de red.

CAPÍTULO 4

Metodología

En este capítulo se van a explicar tanto la metodología de gestión como la de desarrollo empleadas para desarrollar este proyecto. Seguido de la planificación seguida en este proyecto. Y finalmente el marco tecnológico utilizado en este *TFG*.

4.1. METODOLOGÍA DE GESTIÓN

La metodología de gestión que se ha elegido para este *TFG* ha sido la ***Metodología Avanttic FORTE***. Propuesta por Macario Polo Usaola, profesor de la Escuela Superior de Informática y director del *Aula SMACT* creada por *Avanttic*. La metodología *Avanttic FORTE* define una serie de pautas y buenas prácticas. Divide el proyecto en pequeñas interacciones que a su vez se dividen en tareas. Esta división permite que los fallos que puedan acontecer no provoquen grandes retrasos en la planificación del proyecto. Lo que la hace idónea para desarrollar el proyecto en los 6 meses que estipula el convenio y facilitar la comunicación entre el alumno y los tutores.

A continuación, se hablará de los roles principales que se deben identificar y de las distintas etapas que se han de cumplir en el uso de esta metodología.

4.1.1. Identificación de Roles

Se han de identificar los roles que garanticen el seguimiento de las técnicas definidas por la metodología.

- **Equipo de Desarrollo**. El equipo que se va a encargar de planificar, diseñar y implementar el proyecto. En este *TFG* el equipo de desarrollo es el alumno y autor del mismo Roque Rojo Bacete.
- **Cliente**. Empresa que esta interesada en la realización del proyecto, para poder explotar el producto. En este caso es la empresa *Avanttic*, representada por Samuel Campos Herros.
- **Director Académico**. Es el responsable por parte de la Escuela Superior de Informática, que garantiza que el alumno aplique las buenas prácticas de la metodología, proporcionando apoyo a la hora de redactar la documentación. En este *TFG* el director académico ha sido David Vallejo Fernández.

4.1.2. Etapas de la Metodología de Gestión

Según la metodología que hemos escogido, a la hora de elaborar un diseño del proyecto se tienen que seguir una serie pasos, realizados en el orden que aparece en la figura 4.1.

1. **Identificación de los Roles**. Consiste en definir el equipo de desarrollo, el cliente y el director académico.
2. **Definición del Alcance**. Se acuerda entre todos los participantes del proyecto lo que se va a realizar y lo que no.
3. **Identificación y Estimación de los Riesgos**. En esta etapa definen los riesgos que pueden hacer peligrar el desarrollo del proyecto y se establecen planes de contingencia para mitigar el daño de dichos riesgos.
4. **Elaborar el Plan de Proyecto**. Se realizará una estimación del tiempo que se tardara en realizar las tareas del proyecto.
5. **Toma de Contacto con las Tecnologías**. En esta etapa se formará al equipo de desarrollo con las tecnologías que se han de utilizar.
6. **Priorización de requisitos**. Se asignará una prioridad a los requisitos en función del valor que aporten a la organización.

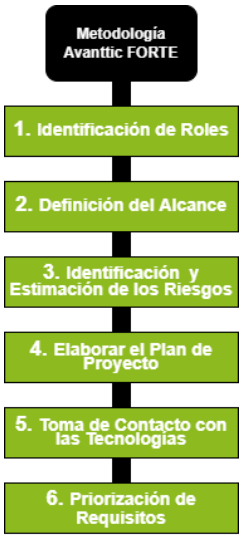


Figura 4.1: Metodología de Gestión

4.2. METODOLOGÍA DE DESARROLLO

Al igual que en la metodología de gestión, en la metodología de Desarrollo hemos elegido también la metodología *Avanttic FORTE*. Al basada en incrementes y un desglose en interacciones se asemeja bastante a una metodología iterativa e incremental.

Para mostrar el funcionamiento de la metodología se ha desarrollado un diagrama de flujo en la que muestran todas las etapas que se siguen en esta metodología, en la figura 4.2.

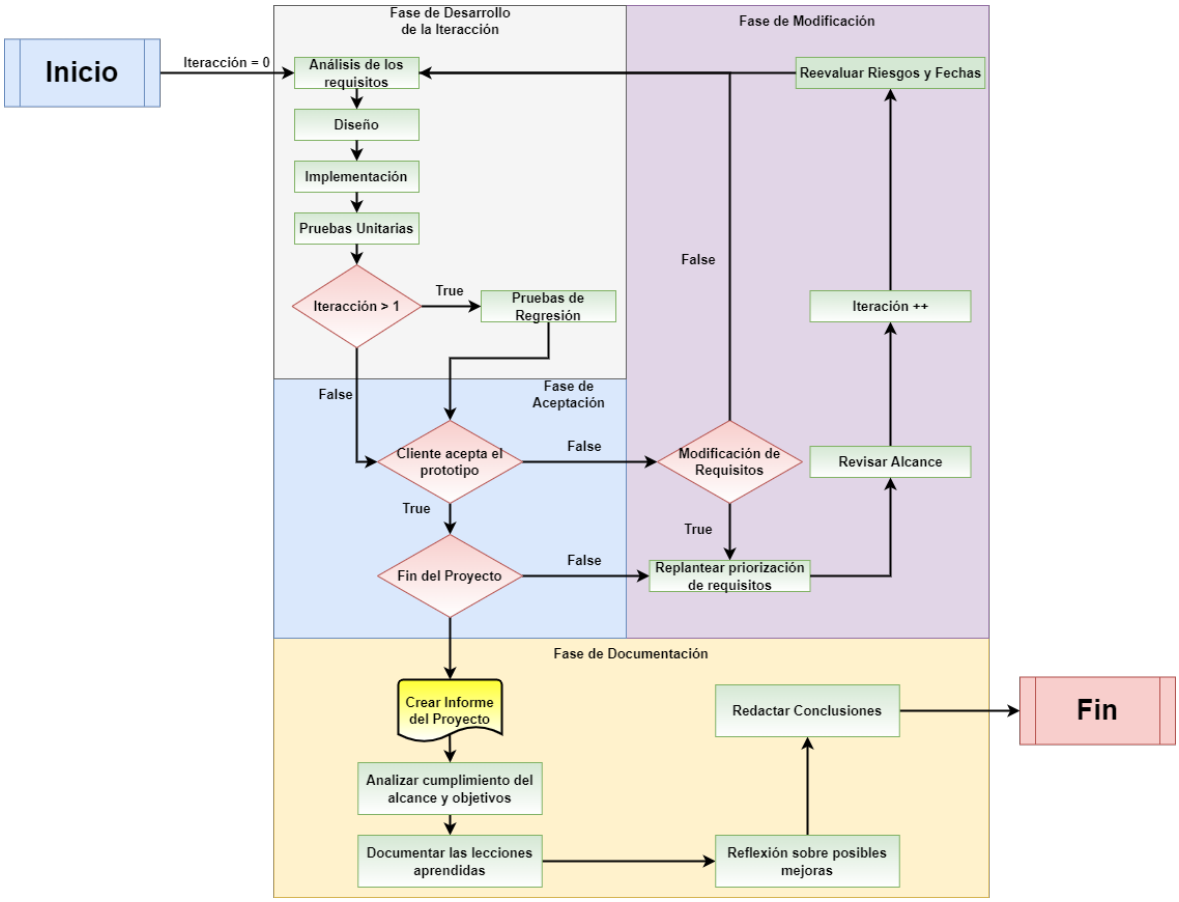


Figura 4.2: Metodología de Desarrollo

4.3. PLANIFICACIÓN

Esta sección tiene como objetivo mostrar la planificación que se hizo en los primeros días de desarrollo. Para desarrollar la planificación se han tenido en cuenta todos los pasos que requiere la metodología, iteraciones, tareas.

La fecha de inicio del proyecto quedo establecida el 6 de septiembre de 2021 y se estimó que la fecha de finalización sería el 10 de enero de 2022. Como la Escuela Superior de Informática fija la fecha límite para esta convocatoria el 9 de febrero de 2022, el resto del mes de enero se destinará revisar la documentación y realizar correcciones.

La jornada laboral esta compuesta por 8 horas de trabajo por día y se trabajará 5 días por semana. La planificación de las tareas e iteraciones quedan reflejadas en la figura 4.3.

Diagrama de *Gantt*

Se ha desarrollado el desglose de las iteraciones y tareas que conforman el proyecto en un diagrama de *Gantt*, que permite tener una visión de la duración de las iteraciones, las tareas y de las dependencias entre ellas. Esto queda ilustrado en la figura 4.4.

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	**Trabajo Fin de Grado**	**90 días**	**lun 06/09/21**	**lun 10/01/22**	
2	Incorporación a la empresa	1 día	lun 06/09/21	lun 06/09/21	
3	Propuesta TFG	5 días	mar 07/09/21	lun 13/09/21	2
4	**Iteración 0: Toma de Contacto**	**29 días**	**mar 14/09/21**	**vie 22/10/21**	**3**
5	Estudio Previo de las Tecnologías	14 días	mar 14/09/21	vie 01/10/21	
6	Instalación de las tecnologías necesarias	5 días	lun 04/10/21	vie 08/10/21	5
7	Curso SpringBoot, JavaScript y Express	10 días	lun 11/10/21	vie 22/10/21	6
8	Definir Alcance y los Requisitos del Proyecto	3 días	lun 04/10/21	mié 06/10/21	5
9	Identificar y Estimar los Riesgos	2 días	jue 07/10/21	vie 08/10/21	8
10	**Iteración 1: StreamConnect**	**17 días**	**lun 25/10/21**	**mar 16/11/21**	**4**
11	Comparar y elegir la librería de Apache Kafka	3 días	lun 25/10/21	mié 27/10/21	
12	Crear Aplicación Base: StreamConnect	2 días	jue 28/10/21	vie 29/10/21	11
13	Conexión a Streaming	2 días	lun 01/11/21	mar 02/11/21	12
14	Configurar Recursos de Streaming	2 días	mié 03/11/21	jue 04/11/21	13
15	Envío de Correos	1,5 días	lun 01/11/21	mar 02/11/21	12
16	Creación Imagen Docker: StreamConnect	2 días	lun 01/11/21	mar 02/11/21	12
17	Desplegar aplicación en Kubernetes	7 días	lun 01/11/21	mar 09/11/21	12;2
18	Conexión Webhook de Jira	3 días	mié 10/11/21	vie 12/11/21	17
19	Pruebas	2 días	lun 15/11/21	mar 16/11/21	11;12;13;14;16;17;18;15
20	**Iteración 2: StreamSub**	**25 días**	**mié 17/11/21**	**mar 21/12/21**	**10**
21	Bocetos y Esquema de la Base de Datos	2 días	mié 17/11/21	jue 18/11/21	
22	Crear Aplicación Base de StreamSub	3 días	mié 17/11/21	vie 19/11/21	
23	Crear Base de Datos local y Kubernetes	2 días	vie 19/11/21	lun 22/11/21	21
24	Desarrollo Back-End	10 días	mar 23/11/21	lun 06/12/21	23
25	Envío de Correos	1 día	lun 22/11/21	lun 22/11/21	22
26	Desarrollo Front-End	6 días	lun 22/11/21	lun 29/11/21	22
27	Crear Imagen Docker: StreamSub	2 días	mar 07/12/21	mié 08/12/21	24;26;25
28	Desplegar aplicación en Kubernetes	7 días	jue 09/12/21	vie 17/12/21	27
29	Pruebas	2 días	lun 20/12/21	mar 21/12/21	21;22;23;24;26;27;28
30	**Iteración 3: Conexión StreamSub - StreamConne**	**9 días**	**mié 22/12/21**	**lun 03/01/22**	**20;10**
31	Microservicios de StreamConnect	3 días	mié 22/12/21	vie 24/12/21	
32	Microservicios de StreamSub	4 días	lun 27/12/21	jue 30/12/21	31
33	Pruebas	2 días	vie 31/12/21	lun 03/01/22	32
34	Documentar TFG	83 días	mar 14/09/21	jue 06/01/22	3
35	**Finalización TFG**	0 días	lun 10/01/22	lun 10/01/22	4;10;20;30;34

Figura 4.3: Planificación de tareas realizada en *Microsoft Project*

4.4. MARCO TECNOLÓGICO

Lenguajes de Programación

- **Java:** es un lenguaje de programación multiplataforma orientado a objetos, que ofrece una gran fiabilidad, rapidez y seguridad. Permite a los desarrolladores escriban el código una vez y lo ejecuten en cualquier plataforma capaz de soportar Java sin necesidad de recompilar el código. En este proyecto ha sido utilizado para desarrollar la aplicación StreamConnect junto con el *framework SpringBoot*.

- **JavaScript:** es un lenguaje de programación web multiplataforma y orientado a objetos. Soporta los navegadores más utilizados: *Chrome*, *FireFox*, *Opera* y *Safari*. Es un lenguaje interpretado, lo que significa que no es necesario compilar un programa para poder ejecutarlo. [9]
  Ha sido utilizado para desarrollar la aplicación StreamSub, tanto en el lado del cliente, como en el lado del servidor.

- **HTML5 (*HyperText Markup Language*):** es la quinta versión del lenguaje básico que utiliza la *World Wide Web* para definir el significado y la estructura del contenido de la web. Además de *HTML* es común que se utilicen otras tecnologías para dar cierta apariencia al contenido (*CSS*) o cierto comportamiento (*JavaScript*). [9]
  Se ha utilizado para crear la estructura y contenido de la aplicación de StreamSub.

- **CSS3 (*Cascade Style Sheet*):** es la última versión del lenguaje de estilos *CSS*, que permite modificar la apariencia en un documento desarrollado por un lenguaje de marcas (en este caso *HTML5*), estilizando las páginas web en las que se utiliza.
  Ha sido empleado junto al framework Bootstrap para dar un toque moderno y estilizado a la aplicación StreamSub.
- **SQL (*Structured Query Language*):** es un lenguaje de dominio especifico, diseñado para administrar, gestionar y recuperar información de las bases de datos relacionales.
  Ha utilizado en este proyecto para definir las tablas de las base de datos, y las consultas en el lado del servidor de la aplicación *StreamSub*.

Entornos de Desarrollo

- **Visual Studio Code:** Es un entorno de desarrollo multiplataforma, desarrollado por *Microsoft* para *Windows*, *Linux* y *macOS*. Es compatible con varios lenguajes de programación, como: *JavaScript, HTML, Java, CSS o TypeScript*. Además, es posible agregar extensiones para hacerlo compatible con más lenguajes de programación.
  Todo el código de la aplicación *StreamSub* ha sido desarrollado con este entorno de desarrollo.
- **Spring Tool Suite 4:** es un entorno de desarrollo basado en la versión *Java EE* de Eclipse, pero altamente personalizado para trabajar con el *framework* de *Spring*.
  Toda la aplicación de StreamConnect ha sido desarrollada con *Spring Tool Suite 4*.

Entornos de Ejecución

- **NodeJS:** es un entorno de ejecución de *JavaScript* orientado a eventos asíncronos, diseñado para aplicaciones web escalables. Cuenta con un sistema de gestión de paquetes llamado *NPM*.
  Ha sido utilizado a la hora de desarrollar la aplicación de *StreamSub*.

Frameworks

- **Express:** Es un *framework* de desarrollo de aplicaciones web minimalista y flexible para *NodeJS*, inspirado en el *framework de Ruby*: Sinatra.
  Ha sido empleado en su totalidad para la creación del lado del servidor de la aplicación *StreamSub*.
- **SpringBoot:** es una especialización del *framework Spring*, que permite ahorrarse gran parte de la configuración necesaria de las aplicaciones basadas en *Spring*.
  Ha sido utilizado para construir la aplicación de *StreamConnect*, y elegido por la buena conectividad que ofrece con *Oracle Streaming Service* gracias a su *API* de *Apache Kafka*, que ahora es compatible con la mayoría de funciones de *Streaming*.
- **Bootstrap 5:** es la quinta versión de un *framework CSS* multiplaforma de código abierto utilizado para el desarrollo del *Front-End* de sitios y aplicaciones web. Contiene plantillas de diseño para todos los elementos *HTML*, permitiendo crear aplicaciones *responsive*.
  Ha sido utilizado para crear el *Front-End* la aplicación *StreamSub*.

Plataformas:

- **Jira:** es una herramienta en línea utilizada para la administración de tareas de un proyecto, el seguimiento de errores, incidencias y peticiones de servicio.
  En este proyecto nos centraremos en las incidencias y peticiones de servicio. Utilizando el servicio de *Webhooks* que ofrece la plataforma, para automatizar el envío de la información a *StreamConnect*.
- **Kubernetes:** es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Facilitando la automatización, despliegue y escalado de aplicaciones, además de ofrecer a una alta disponibilidad y recuperación de errores
- **Docker:** es una plataforma para el desarrollo, envío y ejecución de aplicaciones. Permite separar las aplicaciones de las infraestructuras mediante la creación de imágenes que envuelven a tu aplicación en un entorno virtual que dispone de todo lo necesario para su funcionamiento.
- **Apache Kafka:** es una plataforma de *event streaming* distribuida por la comunidad que es capaz de manejar billones de eventos al día. De forma inicial, *Apache Kafka* fue concebida como una cola de mensajería (*messaging queue*) y tiene su base en una representación de *commits log* distribuidos. [11]
- **Oracle Cloud Infraestructure:** es una plataforma que ofrece un servicio de computación en nube que proporciona servidores, almacenamiento, redes, aplicaciones y servicios a través de una red global de centros de datos administrados por *Oracle Corporation*. [6]
  Entre todos los servicios que ofrece los utilizados han sido: *Streaming, Container Engine for Kubernetes, Container Registry* y *Cloud Shell*.
- **MySQL:** es un sistema de gestión de bases de datos relacional desarrollado por *Oracle Corporation* y está considerada como la base de datos de código abierto más popular del mundo, pensada para utilizarla en entornos de desarrollo web.

Pruebas:

- **Postman:** es un cliente *HTTP* que nos da la posibilidad de probar los diferentes tipos de peticiones *HTTP* a través de una interfaz gráfica de usuario, por la cual podremos analizar las respuestas que posteriormente deberán ser validados.
  En este proyecto ha sido utilizado para realizar pruebas a los diferentes servicios creados, tanto para StreamSub como para StreamConnect.

Control de Versiones y Automatización de Despliegue:

- **Git:** es una herramienta de código abierto que realiza una función del control de versiones del código de forma distribuida. Mantiene un historial completo de todas las versiones del código, por lo que en caso de error se puede volver a una versión anterior sin problema. También facilita la colaboración de varias personas dentro de un mismo proyecto.
  En este proyecto ha sido utilizado la versión que tiene integrada Visual Builder Studio, tanto para el código relativo a *StreamSub*, como el de *StreamConnect*.

- **Visual Builder Studio:** es una herramienta de última generación para el desarrollo de aplicaciones web y móviles que permite a los desarrolladores manejar el ciclo de vida del software y automatizar el entorno de desarrollo. Permitiendo desplegar aplicaciones en *OKE* con el simple hecho de hacer un *commit* en el repositorio *git*, el cual se encuentra integrado dentro de la herramienta. [6]
  En este proyecto se ha utilizado tanto como repositorio del código de *StreamSub* y *StreamConnect*, como para automatizar el despliegue de las aplicaciones en *OKE*.

Edición de Texto:

- **LaTeX:** LaTeX es un sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas. Su principal intención es facilitar el uso del lenguaje de composición tipográfica Tex.
  Ha sido utilizado para la realizar el presente documento.
- **Overleaf:** es un editor colaborativo de *LaTeX* basado en la nube que se utiliza para escribir, editar y publicar documentos científicos. Se asocia con una amplia gama de editoriales científicas para proporcionar plantillas oficiales de *LaTeX* para revistas y enlaces de envío directo.
  Ha sido utilizado para crear este documento junto a la plantilla del curso de Jesús Salido [13] profesor de la *UCLM*.

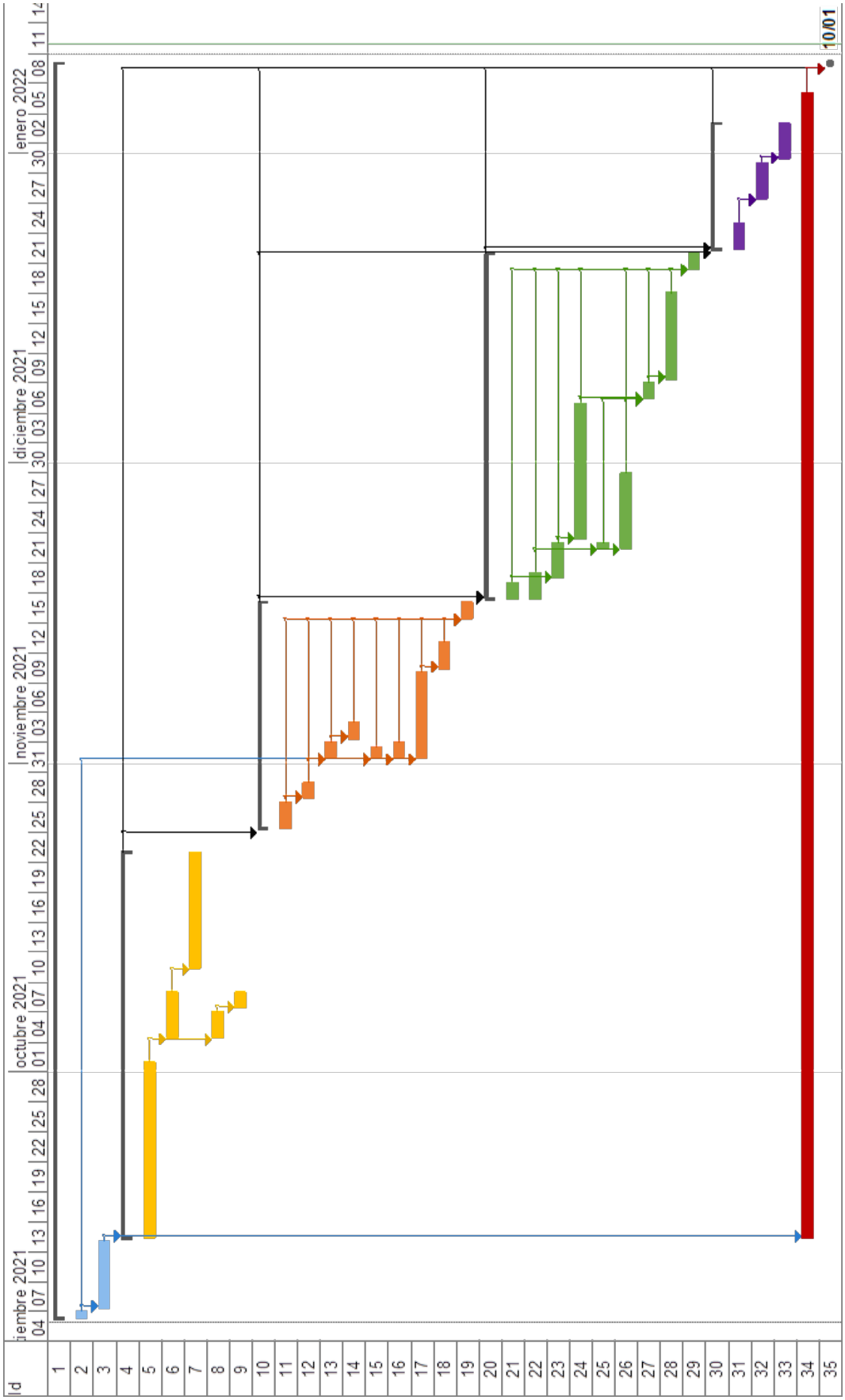


Figura 4.4: Diagrama de *Gantt*

CAPÍTULO 5

Arquitectura

En este capítulo se procederá a analizar las distintas partes que componen la arquitectura del proyecto, explicando en detalle los diferentes módulos que componen la solución. También se explicarán los problemas que han surgido a lo largo del desarrollo junto a las soluciones que se han planteado, y las ventajas haber escogido solución.

A la hora de diseñar la arquitectura del proyecto se han tenido en cuenta los requisitos funcionales y no funcionales que se establecieron en la iteración 0 del capítulo 6.

5.1. ORGANIZACIÓN DEL PROYECTO

En esta sección mostraremos una serie de diagramas y figuras que permitirán tener una idea general sobre la arquitectura de la solución que se ha planteado y de los distintos módulos que la componen.

5.1.1. Visión General de la Arquitectura

En la figura 5.1 podemos ver que la solución que se plantea sigue una arquitectura web basada en microservicios, que permite que los módulos que la componen tengan el mínimo acoplamiento posible, que puedan evolucionar de manera independiente y hacer que los servicios que puede ofrecer la solución escalen fácilmente. Esta compuesto por 2 módulos claramente diferenciados *StreamSub* y *StreamConnect*.

StreamSub es una aplicación web que utiliza *HTML*, *Bootstrap* y *ejs* para crear un *Front-End* comunicado con un *Back-End* de *NodeJS*. Es el único capaz de seleccionar, insertar, editar y eliminar los datos de nuestra Base de Datos *MySQL*. Es el único *Front-End* que tendrá nuestro sistema, y podemos acceder a el mediante un navegador.

StreamConnect, a diferencia de *StreamSub* está compuesto únicamente por *Back-End*, es decir, no tiene ninguna *UI* (Interfaz de Usuario). Esta desarrollada en el *framework* de *Java, SpringBoot*. Es la encargada de comunicarse con *Streaming Service* y *Jira*.

Tanto *StreamSub* como *StreamConnect* comparten las siguientes características:

1. Se encuentran desplegados en *Oracle Cloud* y gestionados por *Oracle Kubernetes Engine*, que es un *PaaS*.
2. Tienen la capacidad de acceder y enviar correos a través de *SMTP* de *Google* con las cuentas de *streamsubb@gmail.com* y *streamconnectt@gmail.com* respectivamente.

3. Tienen comunicación entre sí a través de una *API REST*.

También contamos con una serie de módulos que no son propios, cuya función es:

- **Streaming Service**. Es una plataforma de transmisión de eventos en tiempo real. Utilizamos una librería de *SpringBoot* de *Apache Kafka*, que permite producir y consumir las incidencias.
- **SMTP de Google**. Permite enviar correos electrónicos utilizando una cuenta de *Gmail* y los servidores de *Google*. *StreamSub* lo utilizará para notificar cuando las suscripciones de un consultor cambien, mientras que *StreamConnect* lo utilizará para notificar a los consultores sobre las nuevas incidencias.
- **Jira**. Enviará los datos de las incidencias que se creen en tiempo real a *StreamConnect* mediante su *Webhook*.

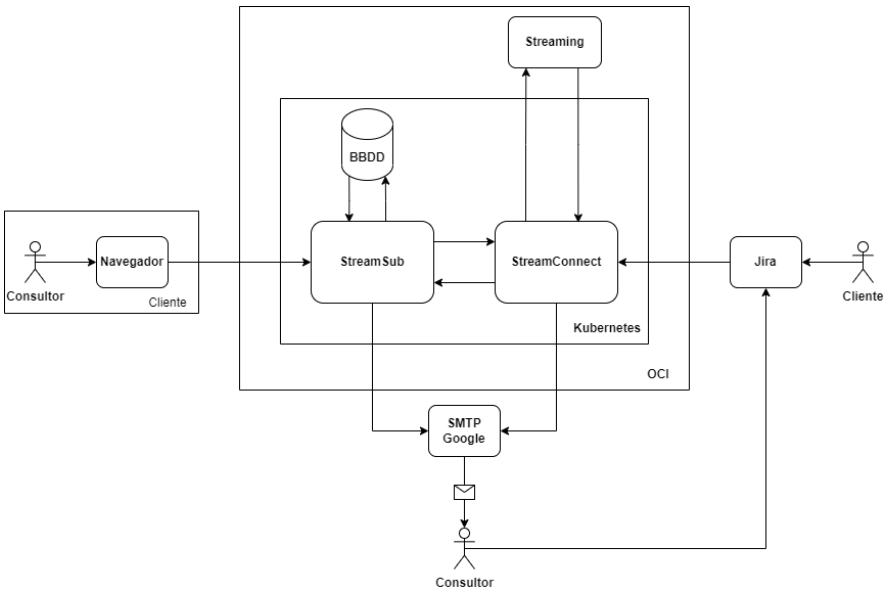


Figura 5.1: Arquitectura Global de la Solución

5.1.2. Arquitectura *Front-End* y *Back-End* de la Solución

StreamSub

Aunque el *Front-End* y *Back-End* de la aplicación se encuentran en el mismo proyecto, sus clases y métodos están aislados. Realizar esta separación conlleva una serie de ventajas entre las que podemos destacar:

- **Mayor Escalabilidad.** Al estar separadas, si algunas de las partes necesita una mayor cantidad de recursos es más fácil y eficaz aumentar los recursos de esa única parte.
- **Distintos Equipos de Desarrollo.** Permite tener diferentes equipos de desarrollo trabajando de forma independiente en el *Back-End* y en *Front-End*.
- **Actualizaciones más Sencillas.** Cuando se completa el *Back-End* la parte que más suele cambiar es el *Front-End* de la aplicación. El *Back-End* seguirá funcionando de la misma forma, por lo que no hace falta modificarlo.

- **Ampliar el Numero de Plataformas.** Con el *Back-End* separado del *Front-End* si se decidiera ampliar el servicio a una aplicación nativa en dispositivos móviles, solo sería necesario desarrollar la parte *Front-End* de la aplicación móvil, puesto que internamente podría seguir utilizando el mismo *Back-End*.

La comunicación entre ambos extremos se hace mediante el envío y la recepción de peticiones a los *endpoints* de la *API REST* habilitados para ello.

El uso de una *API REST* permite el intercambio de información entre el *Front-End* y *Back-End* de una aplicación. Proporcionando aislamiento entre la lógica del *Back-End* y el *Front-End*, por lo que el cliente no se preocupará de como está implementando el servidor, y a su vez el servidor no le importará como son usados los datos que envía al cliente.
Los métodos principales que utiliza *API REST* son:

- **GET.** Solicitar información de un recurso.
- **POST.** Creación de un nuevo recurso.
- **PUT.** Actualizar valores de un recursos existente.
- **PATCH.** Actualizar un único valor de un recurso existente.
- **DELETE.** Eliminar un recurso determinado.

Los recursos que gestiona una *API REST* poseen una *URI* (*Uniform Resource Identifier*), que los identifica y diferencia de los demás recursos, es decir, no puede ser compartida por más de un recurso. Tiene la siguiente estructura:

protocolo://host:puerto/ruta del recurso

En la figura B.1 se puede ver un ejemplo de petición al *endpoint* de la *API REST* que realiza el *Front-End* de *StreamSub* al *Back-End* de que solicita acceso al recurso *infoAreas*.

A continuación, vamos a hablar sobre la organización de la aplicación *StreamSub* basándonos en la figura 5.2. Podemos distinguir 3 zonas:

- **Front-End**. Compuesto por 2 carpetas principales la primera es *views* que contiene los archivos *ejs* que componen la *UI* de nuestra aplicación, y la carpeta *templates*, que contiene las plantillas utilizadas en la aplicación como los correos, *navbar*, *footer*.

  Y segundo la carpeta *public* que a su vez contiene la carpeta *css* y *js*. La primera alberga los estilos de *Bootstrap* y mientras que la segunda contiene archivos *js* que dar funcionalidad a elementos de la *UI*.

- **Back-End**. Compuesto por 2 carpetas, 2 archivos js y las variables de entorno. La primera de la carpetas se llama *database* y como su propio nombre indica contiene los archivos *js* relacionados con la Base de Datos, el primero de ellos se llama *conexión.js* que realiza la conexión a la Base de Datos y el segundo se llama *DBUtil.js* que contienen todas las consultas realizadas a la Base de Datos.

  La carpeta *router* contiene el archivo *rutas.js* que gestiona y define todos los *endpoints* de la aplicación.

  Los 3 archivos restantes son: *index.js* que define las características base del servidor de *NodeJS* y lo ejecuta. *sendEmail.js* que implementa la funcionalidad de enviar mensajes cuando se

actualizan las suscripciones de un consultor y finalmente *.env* que contiene los datos sensibles de nuestra aplicación como contraseñas o tokens.

- **Despliegue**. En esta sección se encuentran los algunos de los ficheros más importantes como son el *Dockerfile*, en el que se especifican todos los pasos llevados a cabo para generar una imagen de *Docker*.
  En *streamSub-deployment.yaml* y *streamSub-service.yaml* se especifican las características que va a tener tanto el *deployment* como el *service* de esta aplicación en *Kubernetes*.

  El archivo *package.json* contiene las dependencias y scripts utilizados durante el desarrollo y finalmente *.gitignore* recoge los archivos que no queremos que se actualicen en el *git* donde almacenamos las aplicaciones.

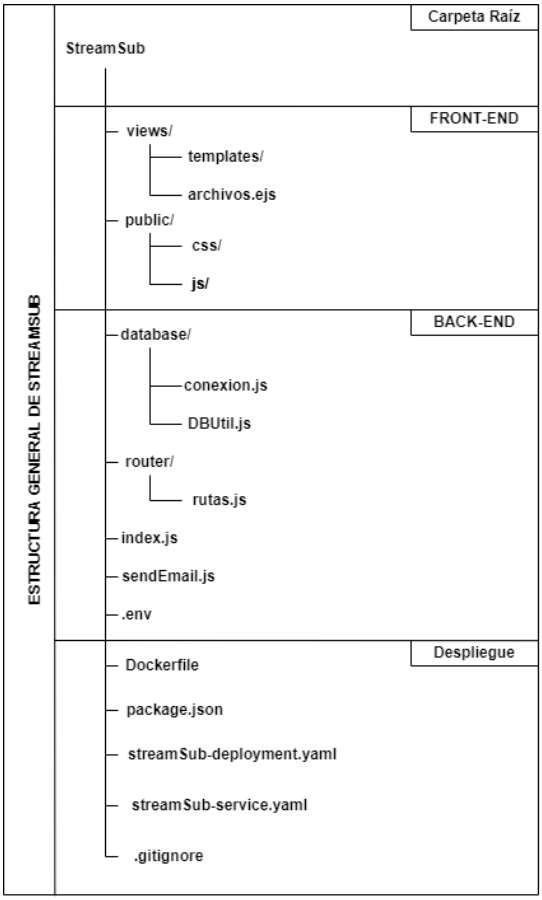


Figura 5.2: Estructura del Sistema *StreamSub*

StreamConnect

A diferencia de *StreamSub*, *StreamConnect* esta compuesto únicamente por código que se ejecuta en el *Back-End*. El *Back-End* de StreamConnect se comunica con: *StreamSub, Streaming Service, Jira* y el *SMTP* de *Google* a través de *API REST* mediante los métodos citados anteriormente.

- **StreamSub**. Mediante los endpoints que tenemos definidos solicitamos información a la Base de Datos, como las asignaciones o la lista de consultores suscritos al área de un *Stream*.
- **Streaming Service**. La comunicación con este servicio la llevamos acabo con la *API* compatible con *Streaming* de *Apache Kafka*, que nos utilizamos para crear y borrar *Streams*, enviar mensajes,

consumir los mensajes y obtener información de los *Streams* como las particiones que tiene cada uno.

- **Jira**. Para recibir la información de las peticiones de servicio creadas por el cliente se ha habilitado un *endpoint* que permite recibir peticiones *POST* de un *Webhook* de *Jira* que hemos configurado para que cada vez que se cree una petición de servicio envíe un *POST* al *endpoint* de *StreamConnect*.

- **SMTP de Google**. Utilizamos la *API* de Google para enviar correos electrónicos a los consultores que estén suscritos a las áreas que originarias de la petición de servicio.

En la figura 5.3 podemos apreciar que en su totalidad esta formado por *Back-End*. Al figura anteriormente citada podemos distinguir dos carpetas principales ubicadas dentro de *src/main/* las cuales son:

- **java**. Que alberga 3 paquetes, el primero de ellos es *com.StreamConnect* que contiene el archivo *StreamConnectApplication.java* que lanza la aplicación.
  El paquete *com.StreamConnect.config* contiene el archivo *KafkaConfiguration.java*, que definen las características y propiedades de los Consumidores, Productores y Administradores de *Streaming Service*.
  Finalmente, el paquete *com.StreamConnect.issue* en la que se encuentra toda la lógica de la aplicación. En *IssueController.java* se definen los *endpoints* de entrada, es decir, los que tienen que ver con realizar peticiones de información a *StreamConnect* o enviar. Mientras que *SuscriptoresController* contiene las peticiones de salida que realiza *StreamConnect* a *StreamSub*. *MessageService.java* es el encargado de realizar el envío de los mensajes a los consultores, gracias a la *API* de *Google*. Por otra parte, *IssueService* es el encargado de enviar las peticiones de servicio de los clientes colocándolas en su zona correspondiente y consumirlas. También dispone de un método que analiza las peticiones antes de su envió, obteniendo la información considerada relevante.

- **resources**. En este directorio encontramos recursos que utiliza la aplicación para funcionar, en la carpeta *templates* sen encuentran las plantillas utilizadas para los correos, que permite diferenciar las peticiones urgentes de las estándar. El archivo*application.yml* contiene la información sobre la configuración de el *SMTP* de Google, los *customfield* de *Jira* e información sensible de la aplicación como contraseñas y tokens.

- **pom.xml** Archivo *XML* que define la estructura del proyecto Maven, además de definir las dependencias utilizadas.

Al igual que en *StreamSub* contamos con una serie de archivos que nos permiten desplegar la aplicación en *Kubernetes*, el primero de ellos es el *Dockefile* en el que especificamos los pasos necesarios para crear la imagen *Docker* de *StreamConnect*. Seguidos de los archivos *streamConnect-deployment, streamConnect-service, streamConnect-ingress* que definen las características que tendrán el *Deployment, Service e Ingress* de nuestra aplicación.

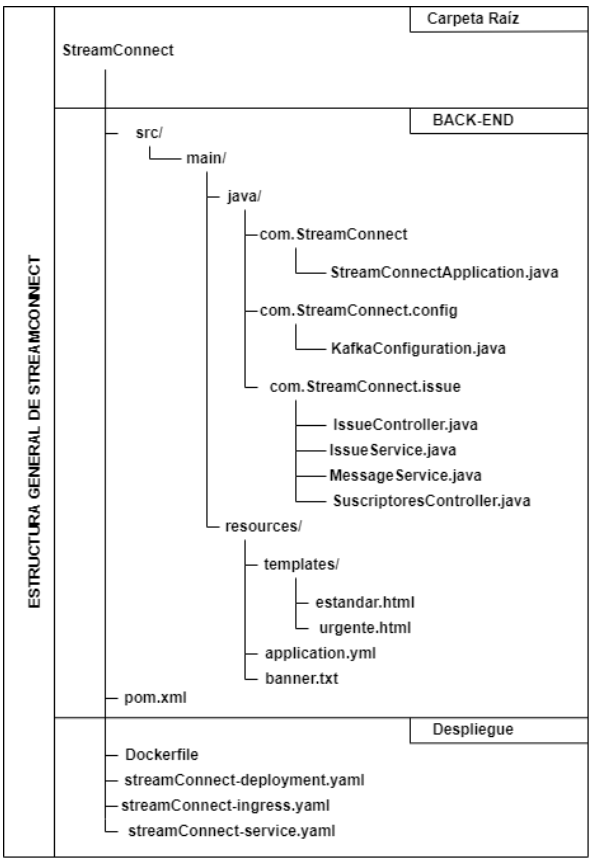


Figura 5.3: Estructura Simplificada del Sistema *StreamConnect*

5.2. DISEÑO DE LA BASE DE DATOS

5.2.1. Elección de la Base de Datos

A la hora de elegir la base de datos con la que se va a trabajar se han barajado 2 tipos:

- **Base de Datos Relacional**. Este tipo de base de datos almacena y proporciona acceso a puntos de datos relacionados entre sí. Representa los datos en tablas, formadas por filas y columnas. Una fila es un registro con un ID único, al que llamamos clave primaria, mientras que las columnas de la tabla forman los atributos de los datos, cada registro suele valores para cada atributo, lo que facilita la creación de relaciones entre los puntos de datos. Estas relaciones, se dan gracias a lo que llamamos clave foránea, que normalmente es la clave primaria de otra tabla que aparece en otra tabla como columna y relaciona ambas tablas [12].
- **Base de Datos No Relacional**. Estas Bases de Datos los datos ya no se almacenan en tablas, si no en documentos, que pueden representarse como un *JSON*. Son más flexibles que las relacionales, ya que permite decidir sobre la marcha que campos queremos guardar en cada registro, al igual que en un objeto *JSON*. Los datos dejan de estar relacionados por lo que se recurren a otras estrategias para simular las relaciones como duplicar información.

Finalmente, se decidió utilizar una base datos relacional para este proyecto, puesto que los datos con los que vamos a trabajar están bien definidos, guardan relaciones importantes entre sí y será necesario definir restricciones. Mientras que las bases de datos no relaciones delegan la responsabilidad de controlar y establecer las relaciones y restricciones al programador, las base de datos relacionales

facilitan esta tarea. Concretamente, la base de datos elegida ha sido *MySQL*, puesto que el alumno tenía experiencia previa utilizando esta base de datos y al ser *Avanttic* un *partner* de *Oracle* dispone de extensa documentación y acceso a foros con resolución de problemas y dudas. Otras de las ventajas que ofrece el uso de *MySQL*:

- **Gratuitidad**. El uso de *MySQL* es libre y gratuito.
- **Velocidad**. Realiza las operaciones rápidamente y con un gran rendimiento.
- **Seguridad**. Las contraseñas son encriptadas, tienen varios niveles de privilegios y acceso para los usuarios.
- **Multiplataforma**. Compatible con *Linux* y *Windows*.
- **Fácil de Usar**. No es necesario un gran conocimiento para empezar a utilizarla.
- **Comunidad**. Al ser una de las más populares, cuenta con una gran comunidad, tutoriales y foros activos que ayudan a la hora de resolver problemas.

5.2.2. Modelo Entidad-Relación

El modelo entidad-relación es un paradigma de diseño que permite representar de manera simplificada los componentes que participan en un proceso de negocio y el modo en el que estos se relacionan entre sí [14]. Como su propio nombre indica para crear un diagrama de Entidad-Relación es necesario definir las entidades y relaciones que tiene nuestro sistema.

Entidades

Una entidad son los objetos de la base de datos de los que queremos almacenar información, cada entidad tiene una serie de atributos, que coinciden con las propiedades que queremos almacenar. Estos atributos pueden servir como relación con otras entidades de la base de datos. Existen varios tipos de entidades, la primera son las *entidades fuertes* que son aquellas capaces de existir por si mismas. La segunda son las *entidades débiles*, cuya existencia depende de una *entidad fuerte*.

Para este proyecto se han definido las siguientes entidades:

- **Consultor**. Un consultor esta compuesto por los atributos de *id_consultor*, que será la clave primaria, *email* que representa la información de contacto del consultor, por lo cual sera único y finalmente *disponibilidad*, que define si el consultor esta disponible y por tanto puede recibir notificaciones. La sentencia *DDL* utilizada para la creación de la tabla de consultores puede verse en el listado 5.1.

```
1 CREATE TABLE consultores (
2     id_consultor INT NOT NULL AUTO_INCREMENT,
3     email CHAR(120) NOT NULL,
4     disponibilidad CHAR(45) NOT NULL DEFAULT 'true'
5     PRIMARY KEY (id_consultor),
6     UNIQUE(email));
```

Listado 5.1: Sentencia *DDL* crear Tabla consultores

- **Area**. Un área esta compuesta por los atributos de *id_area*, que será la clave primaria y *area* que representa el nombre del área a la que ofrecemos servicio, por lo que este nombre será único. La sentencia *DDL* utilizada para la creación de la tabla de áreas puede verse en el listado 5.2 .

```
1 CREATE TABLE areas (
2   id_area INT NOT NULL AUTO_INCREMENT,
3   area VARCHAR(120) NOT NULL,
4   PRIMARY KEY (id_area),
5   UNIQUE (area));
```

Listado 5.2: Sentencia *DDL* crear Tabla áreas

- **Suscripciones**. Un suscripción esta compuesta por los atributos de *id_consultor* que es clave foránea de la tabla de consultores, *topic* y *particion*. La tupla de *id_consultor* y *topic* forma la clave primaria de esta tabla. Podemos ver la sentencia *DDL* utilizada para crear la tabla de suscripciones en el listado 5.3.

```
1 CREATE TABLE suscripciones (
2   id_consultor INT,
3   topic CHAR(120) NOT NULL,
4   PRIMARY KEY(id_consultor,topic)
5   FOREIGN KEY (id_consultor) REFERENCES ↩
      ↪ consultores(id_consultor) ON DELETE CASCADE);
```

Listado 5.3: Sentencia *DDL* crear Tabla suscripciones

- **Asignaciones**. Un asignación esta compuesta por los atributos de *id_area* que es clave foránea de la tabla de áreas, *topic* y *particion* el conjunto de las 3 forma la clave primaria de esta tabla. Podemos ver la sentencia *DDL* utilizada para crear la tabla de asignaciones en el listado 5.4.

```
1 CREATE TABLE asignaciones (
2   id_area INT NOT NULL,
3   particion INT NOT NULL,
4   topic VARCHAR(120) NOT NULL,
5   PRIMARY KEY (id_area,particion,topic),
6   FOREIGN KEY (id_area) REFERENCES areas(id_area) ON DELETE ↩
      ↪ CASCADE);
```

Listado 5.4: Sentencia *DDL* crear Tabla asignaciones

Relaciones

Las relaciones son asociaciones entre entidades que permite utilizar sentencias para recuperar datos usando como base los de otra tabla. Existen varios 3 tipos de relaciones: de uno a uno *(1:1)*, de uno a muchos *(1:N)* y de muchos a muchos *(N:M)*.

En la figura 5.4 podemos ver el diagrama Entidad-Relación de nuestra Base de Datos, en el que podemos apreciar 2 relaciones de uno a muchos. A continuación procederé a explicar el motivo de cada relación.

- **Relación entre áreas y asignaciones**. Es una relación de uno a muchos, puesto que un área puede estar asignada a 0 o varias de las diferentes particiones de los *Streams*.

- **Relación entre consultores y suscripciones**. Es una relación de uno a muchos, puesto que un consultor puede estar suscrito a 0 o a varias particiones de un *Stream*.

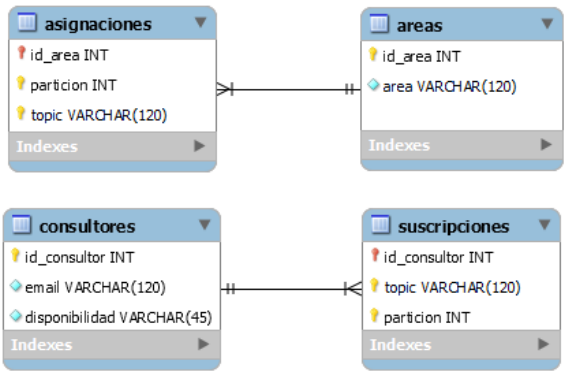


Figura 5.4: Diagrama entidad relación de la Base de Datos

5.3. MÓDULOS

En está sección explicaremos los dos módulos fundamentales que se han desarrollado para esta solución: *StreamSub* y *StreamConnect*. Además también explicaremos los problemas encontrados, la solución que se ha escogido junto a las ventajas de haberla escogido y no otra posible implementación en cada funcionalidad implementada.

5.3.1. StreamConnect

Este módulo es el núcleo principal de nuestra solución, puesto que entre sus funcionalidades se encuentran: recibir las peticiones de servicio de Jira, conexión con *Streaming*, realizar cambios en los *Streams* y solicitar información, enviar el mensaje a la partición y *Stream* correcto, mantenerse a la escucha de de los mensajes entrantes, enviar correos electrónicos a los destinatarios correctos, solicitar y proveer microservicios a *StreamSub*.

En las siguientes subsecciones se detallará el funcionamiento de las funcionalidades de *StreamConnect* y los problemas, si los hubo, durante su implementación.

Compatibilidad de *APIs*

Originalmente la solución estaba planteada para realizar una aplicación única en *NodeJS*, pero surgió un problema de compatibilidad entre la *API* KafkaJS y *Streaming Service*.

Esta incompatibilidad impedía personalizar el criterio que tienen los productores a la hora de realizar el envío de mensajes a particiones de un *Stream*. Este fallo crítico llevo a investigar otras *APIs* que fueran completamente compatibles con *Streaming*. Llegando así a la *API* de *spring-kafka*, que como su propio nombre indica esta disponible para el *framework* de *Java Spring*, con el que se ha desarrollado este módulo. Entre las ventajas de esta solución podemos destacar:

- Permite programar una lógica para nuestro programa que permite elegir a que partición enviamos los mensajes.
- Al realizar una separación entre *StreamSub* y *StreamConnect* nuestra solución pasa a ser una aplicación web basada en microservicios, lo que reduce el acoplamiento entre los dos sistemas.

- Permite escalar fácilmente el servicio de *StreamConnect* de ser necesario en caso de que aumenten sus solicitudes en exceso.
- Al desarrollar en 2 tecnologías diferentes, se han aprendido más conceptos que si se hubiera trabajado únicamente en una.

Recibir Peticiones de Servicio de Jira

Para recibir las nuevas peticiones de servicio creadas en Jira se ha habilitado el *endpoint: /streamConnect/issues* que recibe peticiones *POST* del *Webhook* de Jira.

Un *Webhook* es un mecanismo que permite enviar información mediante una petición *POST* de *HTTP* que se desencadena ante un determinado evento de nuestro sistema. Tanto la información que se envía, el evento y la *URL* a la que se envía la información queda definida en el momento en que creamos el *Webhook*. Como podemos ver en la figura 5.5 lo hemos configurado de la siguiente forma:

1. **URL.** La *URL* que recibirá las peticiones *POST* y posteriormente las procesará es: *https://streamconnect.avanttic.com/streamConnect/issues*.
2. **Información.** Se ha seleccionado que se envíe información sobre las incidencias.
3. **Eventos.** Se ha especificado que el desencadenante de la petición *POST* se ha la creación de una incidencia.

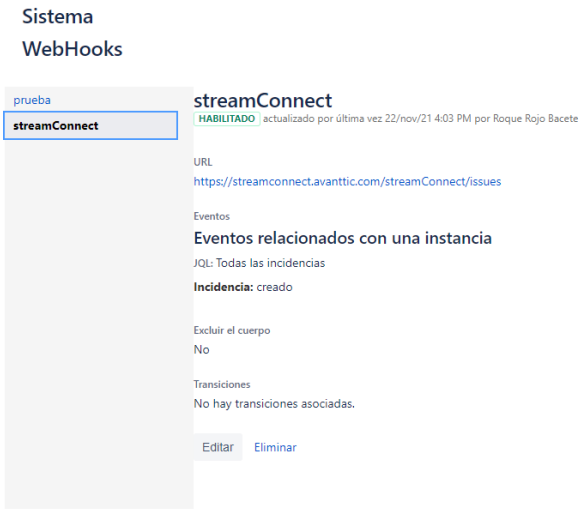


Figura 5.5: Configuración del *Webhook* de *Jira*

Una vez configurado, *StreamConnect* recibirá mediante una petición *POST* la información de cada petición de servicio que se cree en formato *JSON*, después analizará el *JSON* y obtendrá los atributos más importantes de cada petición y finalmente un productor los enviará a su *Stream* y partición correspondiente.

El problema surge cuando el 21 de Julio de 2018 por razones de seguridad Jira restringió la posibilidad de registrar *Webhooks* en direcciones *URLs* no seguras. Obligando a utilizar certificados *SSL* válidos y no autofirmados. Para intentar solucionarlo se solicitó al área de sistemas de Avanttic que nos creará un subdominio en su *DNS* el cual fue *streamconnect.avanttic.com* y el certificado *SSL* que utiliza Avanttic en su propia web.

El certificado que utiliza *Avanttic* es de tipo *Wildcard*, también conocido como certificado comodín, que puede ser utilizado con varios subdominios diferentes dentro de un subdominio, y da cobertura a los subdominios que sigan la estructura ***.avanttic.com** .

Para esto se creó en *Kubernetes* un *tls-secret* al que llamamos avanttic.com y que contiene el certificado de Avanttic. Posteriormente se vinculó al *Ingress* de nuestra aplicación especificándolo en el *Ingress* como se puede ver en el listado A.4.

Pero por motivos de la configuración inicial del *Ingress*, se utilizaba un certificado por defecto autofirmado y caducado, y no detectaba el nuestro. Por lo que hubo que desinstalar el *Ingress* e instalarlo de nuevo modificando en su *Deployment* la opción que seleccionaba el certificado por defecto y sustituyéndolo por nuestro *tls-secret* avanttic.com

Una vez hecho esto el navegador ya nos detectaba como una página segura, pero empezamos a recibir el error ***Unable to verify the first certificate***, esto sucedía debido a que habíamos especificado únicamente el certificado de Avanttic, en otras palabras, faltaban los certificados intermedios que enlazaban con el certificado original expedido por la autoridad de certificación *GoDaddy*. Para obtener estos certificados intermedios hay que buscarlo en el repositorio de certificados de *GoDaddy*. Finalmente, para solucionar el problema solo hay que crear un *tls-secret* que contenga la cadena de certificados correcta.

Otra posible forma de abordar este problema era usar una herramienta llamada hookdeck. Esta herramienta genera un *URL* que puede como destino de *Webhooks* y analizar las peticiones recibidas. Además cuenta con una funcionalidad que le permite reenviar la petición *POST* a cualquier tipo de *URL* sea segura o no. Aunque esta opción fue descartada debido a que logramos solucionar la problemática del certificado *SSL* y por no poner en riesgo la privacidad de los datos de nuestros clientes al confiar en aplicaciones de terceros, si que fue utilizada para trabajar mientras se resolvía la problemática del certificado.

Las ventajas de utilizar la solución escogida son:

- Garantizamos la integridad, cifrado y correcta recepción de los datos de nuestros clientes por la entidad correcta.
- No dependemos de la disponibilidad de servicios de terceros.
- Como podemos ver en la figura 5.6 al utilizar la herramienta de *ssllabs*, la cual permite realizar un test a nuestro certificado *SSL* y en función de su grado de seguridad calificarnos. Basándose en criterios como calidad de la cadena de certificados, cifrado, intercambio de claves y el protocolo de ayuda. Se ha obtenido la nota más alta un ***A+***.

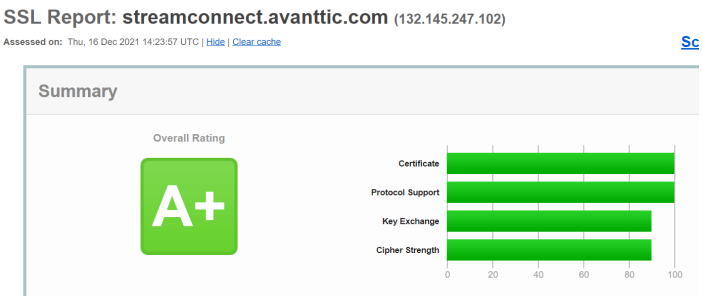


Figura 5.6: Calificación del certificado *SSL*

Conexión con *Streaming Service*

Como hemos mencionado anteriormente la comunicación con *Streaming Service* la realizaremos por medio de la *API* de *Spring*. Para ello debemos configurar todos los elementos de *Streaming* que utilizamos, en este caso son 3: los productores, los consumidores y los administradores. Para lo cual se ha creado un paquete en el proyecto llamado *com.streamConnect.config* que contiene la clase *KafkaConfiguration.java* que contiene la configuración necesaria para su funcionamiento. En el listado A.1 podemos ver un ejemplo de como se han establecido las opciones de la configuración de un administrador.

Los valores son obtenidos desde el archivo *application.yml*. Las ventajas de haber elegido esta aproximación son:

- **Mayor seguridad.** Puesto que evitamos mostrar información sensible en el código de nuestra aplicación.
- **Facilidad de Modificación.** En caso de necesitar un cambio es más fácil realizarlo y evita tener que recompilar todo el código de la aplicación con cada cambio.

Gestionar Recursos de Streaming

StreamConnect se encarga lanzar una serie peticiones contra el servicio de *Streaming*, como son la creación, eliminación y recopilación de información de los *Streams*. Esto se ha conseguido mediante la creación de unos métodos en la clase *IssueService* del paquete *com.streamConnect.issue*. Mientras se llevaba acabo esta funcionalidad ha surgido varios problemas

El primero y más importante de ellos es que *Streaming Service* no permite la edición del numero de particiones de un *Stream* por lo que ha sido imposible crear un método que permita está función.

A la hora de obtener la información de los *Streams* existentes no había ningún método directo de obtenerlo. La solución encontrada para esta problemática fue la creación de un método, llamado *getTopics* que combina dos elementos de *Streaming*: un Productor y un Administrador. Que devuelve un *JSON* cuyo atributos son los *Streams* y el valor el numero total de particiones.

```
1 public JSONObject getTopics() throws InterruptedException, ↩
    ↪ ExecutionException {
2   JSONObject topicInfo = new JSONObject();
3   ArrayList<String> topicNames = new ↩
      ↪ ArrayList<String>(admin.listTopics().names().get());
4   for (String topicName : topicNames) {
5     topicInfo.put(topicName, ↩
        ↪ kafkaTemplate.partitionsFor(topicName).size());
6   }
7   return topicInfo;
8 }
```

Listado 5.5: Método *getTopics* de *StreamConnect*

Como podemos ver en el listado 5.5 el administrador obtiene una lista con los nombres de todos los *Streams*, después se recorre la lista y durante este proceso vamos añadiendo como atributos los nombres de los *Streams* y como valor el resultado de obtener la longitud de la lista que obtiene un

productor cuando llama a su método *partitionsFor* y le pasamos como dato el nombre del *Stream*. Alguna de las ventajas que derivan de esta solución son:

- Tenemos un método sencillo que permite obtener en formato *JSON* los *Streams* con sus respectivas particiones en tiempo real.
- Al obtener los datos en tiempo real, no es necesario almacenar información sobre los *Streams* en la base de datos, ni tener que actualizar información de los mismos cada vez que se realice una modificación.

Lógica para Enviar los Mensajes a las Particiones

Para poder establecer una lógica para el envío de los mensajes, en la base de datos a la cual solo tiene acceso *StreamSub* se ha creado la tabla *areas* y *asignaciones*. En la que las áreas representan los servicios que ofrece Avanttic a las empresas, mientras que la tabla asignaciones almacena la relación de los datos con la tupla *(area, Stream, particion)*, es decir, a una partición de un determinado *Stream* le asignamos un área.

Por otra parte, *Jira* permite la creación de campos personalizados, para este caso hemos habilitado 2. El primero es Empresa, que permite mediante un *Select* elegir la empresa de la que procede la petición, el segundo en cambio, permite seleccionar el área a la que pertenece la petición de servicio. Estos datos son enviados en la petición *POST* que recibirá *StreamConnect* una vez que se cree la petición.

Cuando *StreamConnect* detecta que se ha recibido una petición de servicio, lo primero que hace es filtrar los datos considerados de más importancia para la empresa y los guarda en formato *JSON*. Una vez hecho esto llama al método *sendMessage*. Que analiza los datos de la petición, cuando llega al área realiza una petición *GET* con los parámetros *nombreTopic* (que es el equivalente al nombre del *Stream*) y *area* a *StreamSub* que le proporciona las particiones que tienen asignada ese área en concreto. Una vez hecho esto, recorre la lista de particiones asignadas y envía el mensaje a su lugar correspondiente. Lo citado anteriormente queda representado en el listado 6.2.

La otra alternativa, hubiera sido mantener un listado estático de particiones que tuvieran un área asignada, es decir, la partición N de cada bloque siempre tiene asignada la misma área. Esto supone un problema grave de rendimiento y optimización de recursos puesto que si una empresa contratará un servicio asignado a la partición K, tendríamos que crear un *Stream* con K particiones, a pesar de que solo utilizaríamos una. Mientras que si utilizamos la solución planteada anteriormente optimizaremos al máximo los recursos y el coste total.

Consumir Mensajes

La *API* de *Apache Kafka* proporciona un tipo especial de consumidor llamado *KafkaListener* que permite mantenerse a la escucha constante de un *Stream*, pudiendo incluso especificar las particiones. Pero no tiene una forma directa de escuchar a todos los *Streams*. Sin embargo existe una opción que permite escuchar *Streams* identificando un determinado patrón en el nombre de los *Streams*.
Por lo que la solución encontrada ha sido establecer una regla para que los nombres de los *Streams* sigan siempre el formato: **AVT-*(Nombre de la Empresa)***. Todo esto queda reflejado en el listado 6.1.

Obtener Destinatarios de los Correos Electrónicos

Para obtener los destinatarios de los correos se plantearon dos posibles alternativas.

La primera consistía en que *StreamConnect* debía almacenar la información de las suscripciones en una lista de objetos *Stream*, que cuyos atributos serían el nombre del *Stream*, y una lista de suscripciones, cada elemento de la lista a su vez sería una lista con las direcciones de correo electrónico de los consultores suscritos y la posición de cada vector identificaría la partición a la que están suscritos. Está lista se iría actualizando cada vez que se recibiera un *POST* desde *StreamSub*, que se lanzaría cada vez que las suscripciones de algún consultor se modificaran. Después, al llegar una petición se recorrería la lista de con los *Streams* hasta llegar al mismo que el del origen de la petición, una vez encontrado el *Stream* buscado recorrería la lista de las particiones hasta llegar a la indicada y finalmente recorrería a su vez la lista de los consultores suscritos a la partición y por cada suscriptor se llamaría al método *enviarMail*.

La segunda en cambio consiste en que cada vez que llegue una petición, se comprueba el *Stream* y la partición de la que procede. Con estos datos se realiza una petición *GET* a *StreamSub* que consulta la Base de Datos y devuelve un vector con los consultores suscritos. Una vez se tiene este vector se llama al método *enviarMail* una única vez pasándole como parámetro el vector con todas las suscripciones. Finalmente la solución escogida fue la segunda por los siguientes motivos:

- La primera opción requiere un coste computacional demasiado alto, llegando a una complejidad cúbica.
- Si por motivos de demanda hubiera sido necesario escalar el servicio de *StreamConnect* las listas con la información de las suscripciones podría quedarse desfasada. Puesto que en *Kubernetes* si existen más instancias de una aplicación balancea la carga, lo que implica que no siempre recibirá o responderá la misma instancia del programa. Provocando que siempre haya instancias desactualizadas.

Enviar Correos Electrónicos

Es la principal funcionalidad de *StreamConnect* es que permite informar mediante un correo electrónico a los consultores de las peticiones de servicio de las áreas y *Streams* a los que están suscritos. Para diferenciar las peticiones que son urgentes de las que no lo son, nos basamos en el campo *prioridad* del formulario de *Jira* al crear la incidencia. Si se seleccionan los valores de *High* o *Highest* se enviará un correo urgente, en cambio, si se eligen los valores de *Medium, Low* o *Lowest* se enviará un correo estándar.

Uno de los problemas que surgió es el personalizar el contenido de cada mensaje. Esto se ha solucionado mediante el uso del motor de plantillas *thymeleaf*, que permite renderizar una plantilla *HTML* modificando el contenido que se deseé. La ventaja de utilizar el motor de plantillas *thymeleaf* es que no hace falta almacenar el contenido *HTML* del mensaje en una variable del programa e ir concatenando los datos que se quieren, lo que proporciona una mayor limpieza de código y facilidad de modificación para el desarrollador.

Otra problema que surgió es que los *pods* de *Kubernetes* no tenían acceso a Internet, por lo que les era imposible conectar con el *SMTP* de Google y enviar los correos pertinentes. Esto se debía a

que el nodo maestro de *Kubernetes* estaba en una versión superior a la de los nodos trabajadores, esto generaba un problema de incompatibilidad ya que los *pods* que se encuentran en los nodos trabajadores. Y un nodo trabajador accede a internet utilizando como *proxy* el nodo maestro por lo que al encontrarse en diferentes versiones no podían establecer la conexión. Esto se soluciono actualizando la versión de los nodos trabajadores a la misma versión que el nodo maestro.

Comunicación con *StreamSub*

Para que *StreamSub* pueda recibir datos de *StreamConnect* se ha habilitado los siguientes *endpoints*:

- ***/streamConnect/topics*** envía la información de los *Streams* y el número de particiones que tiene cada uno
- ***/streamConnect/crearTopic*** Permite crear un *Stream* indicando el nombre y el número de particiones que se deseadas en el cuerpo de la petición.
- ***/streamConnect/borrarTopic*** Permite borrar un *Stream* indicando el nombre en el cuerpo de la petición.

5.3.2. StreamSub

Este módulo contiene el único *Front-End* de nuestra solución aunque también tiene su propio *Back-End* independientemente de *StreamConnect*. Entre sus funcionalidades se encuentran: gestionar los recursos de *Streaming Service*, los consultores, las suscripciones, las áreas y las asignaciones a través de un *Front-End*, finalmente también proporciona información sobre las asignaciones y suscripciones a *StreamConnect*.

En las siguientes subsecciones se detallará el funcionamiento de las funcionalidades de *StreamSub* y los problemas, si los hubo, durante su implementación.

Conexión a la Base de Datos

Para establecer la conexión a la Base de Datos se ha utilizado la librería de *NodeJS* para *MySQL*, debido a que es fácil de configurar, es ligera, tiene licencia *MIT* de libre uso, es ampliamente utilizada por lo tanto hay mucha información sobre uso y es soportado nativamente por *NPM*. Inicialmente para entablar comunicación con la Base de Datos se utilizaba el método *createConnection*, aunque durante las pruebas no dio problemas. Cuando desplegamos la aplicación en *Kubernetes* surgieron 2 principalmente:

- ***Timeout***. Las conexiones por defecto, tienen un tiempo máximo de vida de 8 horas si no es utilizada y una vez cerrada no vuelve a crearse por sí misma, lo que provocaba que la aplicación una vez desplegada dejase de funcionar una vez pasadas las 8 horas. Se puede aumentar el tiempo máximo de vida, pero no alargarlo de manera indefinida. Una posible solución era utilizar el *listener* que viene por defecto con la librería y cuando detectará un error realizar un *callback* al método que crea la conexión.
- **Conexión bloqueante**. Las conexión creada es bloqueante, es decir, mientras que se esta ejecutando una consulta a la base de datos no se puede realizar otra. Esto supone un grave problema de rendimiento, puesto que puede haber varios usuarios utilizando *StreamSub*

simultáneamente.

La solución encontrada a esta problemática es utilizar una *pool* de conexiones. Una *pool* de conexiones gestiona varias conexiones simultáneas de manera *"perezosa"*, esto quiere decir que aunque al crearla se defina un número máximo de N conexiones solo mantendrá vivas las conexiones que utilizan e irá creando las demás conexiones sólo si las necesita, permitiendo que se ejecuten varias consultas de forma paralela. Podemos ver como se ha configurado el *pool* de conexiones en el listado A.16.

Algunas de las ventajas de utilizar un *pool* de conexiones frente a utilizar una única conexión e ir renovándola cuando exceda su tiempo máximo de vida son:

- Al tener un mecanismo que permite realizar varias consultas simultáneamente aumenta la eficiencia y rendimiento de la aplicación puesto que permite que varios usuarios la utilicen al mismo tiempo.
- Permite a programador no tener que preocuparse por el tiempo de vida de la conexiones, puesto que las crea cuando las necesita.
- Evita que el programador tenga que estar atento a liberar las conexiones, puesto que las pool poseen un método al que se accede con *pool.query*, que es una abreviatura del flujo de código:

pool.getConnection() ->connection.query() ->connection.release()

Gestionar Recursos de *Streaming Service*

Puesto que *StreamConnect* carece de *Front-End* se ha creado una pestaña llamada *Streams* en *StreamSub* que permite la creación y eliminación de *Streams*. Para lo cual, *StreamSub* realiza una petición *POST* a los *endpoints* de *StreamConnect* habilitados para ello.

Para ***crear Streams*** la petición *POST* se tiene que lanzar contra el *endpoint /streamConnect/-crearTopic* para lo cual son necesarios 2 parámetros. El primero de ellos es nombre del *Stream* que queremos crear, teniendo en cuenta que siempre debe empezar por el patrón *AVT-** y por otra parte, también es necesario especificar el número de particiones, por defecto y como mínimo 1, mientras que el límite viene dado por la cantidad de recursos que tenemos contratados en *Oracle Cloud*.
A la hora de crear un *Stream* es necesario tener en cuenta que no puede existir un *Stream* con el mismo nombre y que debe seguir la estructura definida por *AVT-**, para ello se ha implementado un método que comprueba si el *Stream* que se desea crear ya existe y que añade automáticamente *AVT-** al nombre en caso de que el usuario no lo haya hecho.

Para ***eliminar Streams*** la petición *POST* que generamos se tiene que lanzar contra el *endpoint /streamConnect/borrarTopic* y únicamente es necesario que utilicemos como parámetro el nombre de *Stream* que queremos eliminar. Es necesario asegurarse de que el *Stream* que queremos eliminar existe de lo cual se encarga la propia *UI* de *StreamSub*.

Algunas de las ventajas que conlleva realizar la gestión de los recursos de *Streaming Service* en *StreamSub* son:

- Reduce la carga de trabajo de *StreamConnect*.
- Centraliza en un mismo lugar todos los recursos necesarios facilitando la tarea de los usuarios.

Enviar Correos Electrónicos

Al igual que *StreamConnect*, *StreamSub* tiene la capacidad de enviar correos a los consultores, informándoles de cuando sus suscripciones han sido alteradas. En la figura 6.6 podemos ver un ejemplo de correo que se le enviaría a los consultores, en el que se puede ver un listado de los *Streams* y áreas a la que se encuentra suscrito el consultor, además de disponer de un enlace directo a *StreamSub* que le permite modificar sus suscripciones.

Los problemas y soluciones que han surgido en el desarrollo de esta funcionalidad han sido los mimos que encontramos en durante el desarrollo de *StreamConnect*. Con la única salvedad de que el motor de plantillas utilizado ha sido *EJS* puesto que ya lo utilizamos en nuestro *Front-End*.

Comunicación con *StreamConnect*

Para que *StreamConnect* pueda recibir datos de *StreamSub* se ha habilitado los siguientes *endpoints*:

- ***/streamSub/suscriptores/:{topic}/:{particion}*** que permite recibir un vector con todos los suscriptores a una determinada partición de un *Stream*, especificando el *Stream* y la partición que se desea consultar.
- ***/streamSub/asignacion/:{topic}/:{area}*** que permite recibir el número de la partición a la que esta asignada un área en un *Stream*, especificando el nombre del *Stream* y al área que se desea consultar.

CAPÍTULO 6

Resultados

Mientras que en el capítulo 5 se han explicado los problemas que han surgido durante el desarrollo, cómo se han solucionado y las ventajas de haber elegido esa solución, en este capítulo expondremos los resultados obtenidos durante la elaboración del proyecto. Para esto, se ha dividido el capítulo en 4 secciones que corresponde a cada una de las iteraciones definidas en la figura 4.3. Para facilitar la compresión, cada sección cuenta con un apartado de planificación (en la que se muestra las horas estimadas de cada tarea), resultados (en la se explica el objetivo de cada tarea, como se ha completado y el resultado) y revisión (en la que se realiza una comparación de las horas estimadas respecto a las horas reales).

El número total de horas asciende hasta las 555 horas [1]. Esto es debido a que la planificación ha sido realizada en rangos de fecha, y como se explicó en el capítulo 1, el proyecto ha sido elaborado a la par que se hacían las prácticas y se han tenido en cuenta las interrupciones causadas al tener que realizar otro tipo de tareas relativas a las prácticas en la empresa.

6.1. ITERACIÓN 0

Esta iteración se denomina toma de contacto y tiene como objetivo que el alumno establezca una base estable y a partir de ella, empezar a desarrollar el proyecto. En esta interacción estudiarán, instalará y formará sobre las tecnologías necesarias para desarrollar el proyecto. También se define el alcance, los objetivos y requisitos funcionales y no funcionales del proyecto, finalmente se identificará y estimarán los riesgos que pueden suceder durante el desarrollo.

6.1.1. Planificación

Esta iteración esta planificada para que comience el 14 de septiembre de 2021 y se finalice el 22 de octubre de 2021. Como se puede apreciar en la figura 4.3 se divide en 5 tareas. En la tabla 6.1 se puede ver la duración que se ha estimado para cada tarea en horas.

[1] El número de horas requeridas para alcanzar los 12 créditos de un TFG (300) ha sido superado debido a que este proyecto se esta integrando como caso real en una empresa.

Iteración	Tareas	Horas Estimadas
0	Estudio e Instalación de las Tecnologías	100
	Definir Alcance y Requistos del Proyecto	40
	Identificar y Estimar los Riesgos	16
Horas Totales		156

Tabla 6.1: Planificación Iteración 0

6.1.2. Resultados

Estudio Previo e Instalación de las Tecnologías

En esta tarea el alumno ha realizado un estudio sobre los temas y tecnologías necesarios para abordar el desarrollo del *TFG*. Puesto que con la información que tenía le sería imposible desarrollas las siguientes tareas.
Para cumplir esta tarea, el alumno también realizó varios cursos de *SpringBoot, JavaScript, Express* y *Apache Kafka* alojados en *Udemy*, una plataforma de aprendizaje en línea. Puesto que el proyecto ha sido desarrollado en un ordenador nuevo proporcionado por *Avanttic*, el alumno tuvo que instalar las tecnologías y herramientas necesarias. Entre la que podemos destacar: *Java, Docker Desktop, Visual Studio Code, Spring Tool Suite 4, NodeJS, MySQL, Postman y Apache Kafka.*

Definir el Alcance y los Requisitos del Proyecto

Durante las primeras semanas, siempre que la disponibilidad de los tutores lo permitieran se llevaron a cabo reuniones semanales, que ayudaron a definir el alcance y los requisitos del proyecto.

El **Alcance del Proyecto** ha sido definido como "*Crear un sistema que permita que los consultores se suscriban a las áreas de soporte sobre las que Avanttic ofrece servicio a sus clientes y que todos los consultores suscritos y disponibles reciban una notificación cuando se declare una incidencia*".

En cuanto a los **Requisitos**, son las necesidades sobre el contenido, la forma o la funcionalidad que debe cumplir nuestro sistema. Estos requisitos han sido divididos entre funcionales y no funcionales.

Los *Requisitos Funcionales* definen las funcionalidades que debe presentar nuestra solución. Los requisitos funcionales propuestos para este proyecto son:

- Gestionar los consultores, los *Streams*, las áreas, suscripciones de los consultores y las asignaciones de las áreas a las particiones de los *Streams*.
- Notificar al consultor cuando sus suscripciones sean modificadas.
- Obtener las incidencia desde *Jira* en tiempo real.
- Procesar las incidencias según la empresa y área de la procedan.
- Notificar a todos los consultores que estén disponibles y suscritos a un área de un *Stream* cuando se registre una incidencia que proceda de ese área y de ese *Stream*.
- Las incidencias urgentes y no urgentes deben notificarse de forma diferente.

Los *Requisitos no Funcionales*, también conocidos como atributos de calidad, son los criterios que se utilizan para juzgar los comportamientos. Los requisitos no funcionales propuestos para este proyecto son:

- Tanto *StreamSub* como *StreamConnect* deben estar alojados en *Oracle Cloud*.
- El despliegue en *Oracle Cloud* debe hacerse con las tecnologías *Docker* y *Kubernetes*.
- La solución debe ser escalable y su despliegue debe estar automatizado, garantizando los principios fundamentales de *CI/CD*.

Identificar y Estimar los Riesgos

Esta tarea tiene como objetivo identificar los riesgos y elaborar un plan de respuesta para mitigar el efecto de los riesgos que pueden suceder durante el desarrollo del proyecto. Para esto el alumno se apoyó en la lista de comprobación de riesgos en proyectos *software*, formada por 111 riesgos que pueden dar lugar a la hora de desarrollar un proyecto *software* [8]. Estos riesgos están agrupados en 12 categorías definidas por una letra, estás categorías son:

A Elaboración de la Planificación.
B Organización y Gestión.
C Ambiente / Infraestructura de Desarrollo.
D Usuarios finales.
E Cliente.
F Personal Contratado.
G Requisitos.
H Producto.
I Fuerzas Mayores.
J Personal.
K Diseño e Implementación.
L Proceso.

Los riesgos que tenían la probabilidad de ocurrir han sido recogidos en el siguiente listado.

- **A.2** Calendario demasiado optimista.
- **A.5** No se puede construir un proyecto de tal envergadura en el tiempo asignado.
- **A.12** Las áreas desconocidas del producto llevan más tiempo del esperado en el diseño y/o la implementación.
- **B.10** La planificación del proyecto es demasiado mala para ajustarse a la velocidad de desarrollo deseada.
- **C.7** La curva de aprendizaje para las nuevas herramientas de desarrollo es más larga de lo esperado.
- **E.1** El cliente insiste en nuevos requisitos.
- **J.7** El personal necesita un tiempo extra para acostumbrarse a trabajar con herramientas o entornos nuevos.

En la tabla 6.2 se describen las posibles respuestas a los riesgos que han sido identificados en el listado anterior, también conocido como plan de contingencia. Nuestro plan consiste en asignar a cada riesgo una acción (aceptar, mitigar, transferir, evitar) y una respuesta.

6.1.3. Revisión

En este apartado se van a comparar los resultados obtenidos con la planificación inicial de la iteración. Para lo cual en la tabla 6.1 se ha realizado una comparación de las horas estimadas frente a las horas reales.

Riesgo	Acción	Respuesta
A.2	Mitigar	Como se ha elaborado un calendario con holgura suficiente, este riesgo se mitigará reajustando el horario.
A.5	Aceptar	Se eliminarán algunas tareas que no sean necesarias para el correcto funcionamiento de las aplicaciones.
A.12	Aceptar	Este riesgo se aceptará, puesto que durante las primeras semanas, en la planificación se ha planteado una etapa de investigación y aprendizaje para poner en contexto al alumno.
B.10	Aceptar	Si se fuera por detrás de los tiempos de establecidos se llevaría a cabo una replanificación del proyecto.
C.7	Aceptar	Se aceptará, por que se tuvo en cuenta en la planificación y se dedicó una tarea para evitarlo.
E.1	Mitigar	Solo se llevarán a cabo si sobra tiempo y todos los los requisitos originales hayan sido completados.
J.7	Aceptar	Se aceptará, por que al igual que en los riesgos A.12 y B.10 hay una etapa especifica en la planificación para apaliar este riesgo.

Tabla 6.2: Plan de Contingencia

Iteración	Tareas	Horas Estimadas	Horas Reales
3	Estudio e Instalación de las Tecnologías	100	108
	Definir Alcance y Requisitos de Proyecto	40	36
	Identificar y Estimar Riesgos	16	17
Horas Totales		156	161

Tabla 6.3: Revisión Iteración 0

6.2. ITERACIÓN 1

En la iteración anterior se consiguió realizar un marco de investigación, aprendizaje e instalación de las herramientas que han sido necesarias para la ejecución de este proyecto. Sin embargo, esta iteración da pie al comienzo del desarrollo del proyecto. La iteración recibe el nombre de *StreamConnect* y abarca el desarrollo completo de la aplicación.

6.2.1. Planificación

La planificación de esta iteración establece que comience el 25 de octubre de 2021 y termine el 16 de noviembre de 2021. En la tabla 6.4 podemos ver los requisitos de esta iteración y las tareas en las que se desglosa cada requisito junto a las horas que se ha estimado que dure cada tarea.

6.2.2. Resultados

Comparar y Elegir Librería para Trabajar con *Apache Kafka*

Como *Streaming Service* es un servicio relativamente nuevo, no existe demasiada documentación, sobre el uso de la tecnología. Pero, gracias a que internamente *Streaming* funciona como *Apache Kafka* se lanzó una actualización de *Streaming* que hace compatibles las librerías más famosas de *Apache Kafka*, como lo son *KafkaJS, Confluent-Kafka-Python* y *Spring-Kafka*.

Las librerías que se propusieron para desarrollar el *TFG* fueron la de *KafkaJS*, que permite trabajar desde un entorno con *NodeJS* y *Spring-Kafka*, que como su propio nombre indica permite trabajar

Iteración	Requisito	Tareas	Horas Estimadas
1	Desarrollar la Aplicación	Elegir librería Apache Kafka	16
		Crear Aplicación Base	10
		Conexión a Streaming	10
		Configurar Recursos Streaming Service	14
		Envío de Correos	12
	Automatizar el Despliegue	Crear Imagen Docker	16
		Desplegar Aplicación en Kubernetes	56
	Recibir Peticiones de Jira	Conexión Webhook Jira	24
Horas Totales			158

Tabla 6.4: Planificación Iteración 1

con el *Framework* de *Java SpringBoot*.

Prueba	Spring-Kafka	KafkaJS
Conexión con Streaming	Si	Si
Enviar Mensaje a un Stream	Si	Si
Enviar Mensaje a la partición concreta de un Stream	Si	No
Listar los Streams	Si	Si
Consumir mensajes de un Stream	Si	Si
Consumir mensajes de todos los Streams	Si	No
Obtener número de particiones de un Stream	Si	No
Crear Stream con N particiones	Si	Si
Editar Streams	No	No
Eliminar Streams	Si	Si

Tabla 6.5: Pruebas Realizada para Elegir la Librería

En la tabla 6.5 podemos ver las pruebas que se han realizado. Se puede apreciar que la librería de *KafkaJS* no es compatible del todo con *Streaming* puesto que no se puede enviar un mensaje a una partición concreta, tampoco se puede consumir de todos los *Streams* simultáneamente, ni obtener el numero de particiones de un *Stream*.

Aunque en *Apache Kafka* si que se puede modificar el número de particiones de un *Topic* siempre aumentar la cantidad, nunca disminuir. En *Oracle Cloud* por motivos de seguridad no se permite editar el número de particiones de los *Streams*, por lo que aunque las librerías dispongan de esa funcionalidad *Streaming* no lo permite.

Una vez realizado este estudio, se concluyó que la librería que se va a utilizar será *Spring-Kafka*.

Crear Aplicación Base de *StreamConnect*

En la tarea anterior, se decidió que se utilizaría la librería de *Spring-Kafka*, por lo que la aplicación de *StreamConnect* la construiremos utilizando el lenguaje de programación *Java*, concretamente la versión 16.0.2 y *Framework SpringBoot* en su versión 2.6.2.

Para desarrollar la aplicación se ha utilizado el *IDE SpringToolSuite* diseñado específicamente

para trabajar con aplicaciones *Spring* y *SpringBoot*. Para crear el proyecto, se ha utilizado el asistente de creación de proyectos *SpringBoot* de *SpringToolSuite*. Que permite seleccionar el tipo de proyecto que queremos utilizar, el tipo de empaquetamiento, el lenguaje, la versión de *Java*, la versión de *SpringBoot* y las dependencias. Las opciones seleccionadas para la aplicación base están representadas en la figura B.2.

Una vez se generó el proyecto *StreamConnect* se creo la clase *IssueController.java*, la cual actuará como *REST Controller*. En ella se configuraron dos *endpoints*, utilizaba el método *GET* y se llamaba *hola* y en caso de tener éxito respondía un "Hola Mundo!". El segundo en cambio, utiliza el método *POST* y devuelve al usuario “El sistema ha recibido tu petición POST". El código empleado puede verse en el listado A.2.

Conexión a *Streaming Service*

Para realizar la conexión a *Streaming* es necesario crear un compartimento en la nube de *Oracle*. Para ello hay que acceder a *Oracle Cloud*, concretamente al apartado de *Identity & Security* y al área de *Compartments*. Donde mostrará un listado con todos los compartimentos existentes de nuestra tenencia y un botón que abrirá un formulario para crear compartimentos. Cuando nos encontremos en el formulario debemos establecer el nombre del compartimento, una breve descripción de su cometido y el compartimento padre, en nuestro caso será el raíz. Todo esto queda reflejado en la figura B.4.

Acto seguido, hay que acceder al servicio de *Streaming* desde *Oracle Cloud* y seleccionar el compartimento que acabamos de crear. Para poder realizar la conexión a *Streaming* debemos acceder las *pools* y crear uno nuevo o utilizar la que viene por defecto, en este caso he utilizado el que viene por defecto con unas modificaciones. Para realizar las modificaciones hay que acceder a los detalles de la *DefaultPool*, en la que desactivaremos la opción *Auto create topics* que inhabilitara la creación automática de *Streams* y pulsaremos el botón de *Create Kafka Connection Settings* lo que nos proporcionará los datos necesarios para conectarnos a *Streaming* utilizando una librería de *Apache Kafka*, las configuraciones que nos proporciona *Streaming* pueden verse en la figura B.5.

Una vez se obtuvo la configuración necesaria para conectarse con *Streaming* se creo un objeto *AdminClient* para corroborar si la conexión a *Streaming* había funcionado, todo esto queda ilustrado en el listado A.1.

Para comprobar si el estado de la conexión había sido un éxito se invocó al método *listTopics().names()* de *admin* que como se puede ver en la figura 6.1, se han obtenido todos los *Streams* existentes en ese momento, por lo que podemos concluir que la conexión con *Streaming* ha sido un éxito.

```
2022-01-07 11:39:00.236  INFO 20872 --- [           main]
Los Streams existentes son:
[AVT-Stream, AVT-Affinity, AVT-WiZink, AVT-Global]
2022-01-07 11:39:01.317  INFO 20872 --- [| adminclient-1]
2022-01-07 11:39:01.328  INFO 20872 --- [| adminclient-1]
2022-01-07 11:39:01.329  INFO 20872 --- [| adminclient-1]
```

Figura 6.1: Comprobación de la Conexión a *Streaming*

Configurar los Recursos de *Streaming*

Cuando hablamos de recursos de *Streaming* nos referimos a configurar el administrado, los productores y los consumidores. Para llevar acabo esta tarea se realizaron dos cambios importantes en el proyecto,

El primero de ellos consiste en cambiar el archivo de *application.properties* por uno en formato *yaml*, puesto que *SpringBoot* admite ambos formatos. En este fichero se externalizará los datos sensibles de nuestro programa, como es el caso de la configuración de *Streaming*.

El segundo de ellos consiste en reorganizar la estructura del proyecto. Se crearon 2 paquetes:

- **config**. Contiene la clase *KafkaConfiguration.java* en la que se encontrará la configuración de los recursos de *Streaming*.
- **issue**. en la que se alojará la lógica del programa, la clase *issueController.java* ha sido reubicada aquí.

Se ha configurado de tal forma que obtiene los datos de la configuración del archivo *application.yml*. Podemos destacar que se ha desactivado la opción *ALLOW_AUTO_CREATE_TOPICS*, del consumidor, evitando así que cree *topics* que no deberían existir. Finalmente, en la misma clase se han creado unas *beans*, para generar los recursos con la configuración deseada. Dicha configuración puede verse en el listado A.1.
Por ultimo, se ha creado una clase llamada *IssueService.java* en la que se crearon los siguientes métodos:

- **listen**. Es un consumidor de tipo *KafkaListener*, consume los mensaje de todos los *Streams* que comiencen por *AVT-*. Cada vez que consume un mensaje, basándose en la partición y *Stream* solicitará a *StreamSub* los emails de los consultores que se encuentran suscritos, y una vez los obtenga, realizará el envío de correos. El código de este método puede verse en el listado 6.1.

```
1  @KafkaListener(topicPattern = "AVT-.*", groupId = "consumer")
2  public void listen(ConsumerRecord<String, String> message) ↩
       ↪ throws ParseException, URISyntaxException {
3    String destinatarios[] = ↩
       ↪ sc.getSuscriptores(message.topic(), ↩
       ↪ message.partition());
4    log.info("Topic: {}, Particion: {}, Key: {}, Mensaje: ↩
       ↪ {}", message.topic(), message.partition(), ↩
       ↪ message.key(),
5        message.value());
6    if (destinatarios.length > 0) {
7      for (int i = 0; i < destinatarios.length; i++) {
8        log.info("Destinatario: {}", destinatarios[i]);
9      }
10     messageService.enviarMail(message, destinatarios);
11   } else {
12     log.warn("No hay nadie suscrito a la particion {} del ↩
         ↪ topic {}", message.partition(), message.topic());
13   }
14 }
```

Listado 6.1: Método *listen* de *StreamConnect*

- **sendMessage**. Es un productor, que envía los datos de la incidencia recibidas de *Jira* en formato *JSON* a la partición de un *Stream*, basándose en el área que se ha seleccionado a la hora de crear la incidencia. Esta partición se obtendrá al hacer una petición *StreamSub*. El código de este método se encuentra en el listado 6.2.

```
1  public void sendMessage(JSONObject mensaje) throws ↩
       ↪ InterruptedException, URISyntaxException {
2    ProducerRecord<String, String> pr;
3    String nombreTopic = mensaje.get("empresa").toString();
4    String area = mensaje.get("area").toString();
5    int particiones[] = sc.getAsignaciones(nombreTopic, area);
6    if (particiones.length > 0) {
7      for (int particion : particiones) {
8        log.info("Produciendo peticion a {} area {} particion ↩
           ↪ {}", nombreTopic.substring(4), area, particion );
9        pr = new ProducerRecord<String, String>(nombreTopic, ↩
           ↪ particion, nombreTopic+" "+area,
10           mensaje.toJSONString());
11       kafkaTemplate.send(pr).get();
12     }
13   }else {
14     log.warn("El area {} no esta asignada al topic {}", ↩
         ↪ area, nombreTopic);
15   }
16 }
```

Listado 6.2: Método *sendMessage* de *StreamConnect*

Hasta que no se completo la iteración 3, tanto el envío de correos al consumir una petición, como el envío de mensajes a *Streaming* se realizaba con valores estáticos.

Creación Imagen Docker de la Aplicación

Para crear una imagen *Docker* de nuestra aplicación, debemos crear un archivo en el directorio raíz de nuestro proyecto llamado *Dockerfile*, en el que se definen una serie de pasos a seguir para crear la imagen de la aplicación. El contenido de archivo *Dockerfile* puede verse en el listado 6.3, en el cual cabe destacar que la imagen base sobre la que se trabajará tendrá instalado el *Java 16* y que se expone el puerto *8080*, haciéndolo coincidir con el que utiliza *SpringBoot*.

```
1 FROM openjdk:16-jdk-alpine
2 EXPOSE 8080
3 ARG JAR_FILE=target/streamConnect-0.0.1-SNAPSHOT.jar
4 ADD ${JAR_FILE} app.jar
5 ENTRYPOINT ["java","-Djava.security- ↩
    ↪ egd=file:/dev/./urandom","-jar","/app.jar"]
```

Listado 6.3: *DockerFile StreamConnect*

Desplegar la Aplicación en *Kubernetes*

Para poder desplegar la aplicación en *Kubernetes* es necesario crear un *cluster* de *Kubernetes*, para ello hay que acceder a *Oracle Cloud* concretamente a la zona de *Containers & Artifacts* y rellenar

el formulario con las características deseadas para el *Cluster*. Dicho formulario puede verse en la figura B.3.

Una vez tengamos el *cluster creado* necesario crear 3 archivos necesarios para el despliegue de la aplicación. El primero de ellos es el *Deployment*, en el cual se definen parámetros importantes, como una label para referirse a él, el numero de réplicas que pueden existir, la dirección e imagen que se ha de utilizar y el puerto que se expone, todo esto queda reflejado en el listado 6.4.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stream-connect-deployment
spec:
  selector:
    matchLabels:
      app: stream-connect
  replicas: 1
  template:
    metadata:
      labels:
        app: stream-connect
    spec:
      containers:
      - name: stream-connect
        image: fra.ocir.io/fr0k8vgjqhib/streamconnect:latest
        ports:
        - containerPort: 8080
      imagePullSecrets:
      - name: regcred
```

Listado 6.4: *Deployment StreamConnect*

El segundo de ellos es el *Service*, que establece la forma de comunicarse internamente entre los *pods*. En el listado 6.5, se puede ver que se ha creado un *Service* de tipo *ClusterIP*. Afecta a los *pods* que se llaman *stream-connect*, utiliza protocolo *TCP* y vincula el puerto 80 del *pod* con el 8080 de la aplicación.

```
apiVersion: v1
kind: Service
metadata:
  name: stream-connect-service
spec:
  selector:
    app: stream-connect
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Listado 6.5: *Service StreamConnect*

Por último se define el *Ingress*, que entre sus funciones esta controlar el acceso a los *pods*, crear una *IP* pública, alojar varias aplicaciones diferenciando por la ruta y asociar certificados *SSL*. En el

listado A.8 podemos ver que no se ha definido ninguna regla en particular, únicamente se vincula al *Service* creando anteriormente.

Una vez definido los archivos necesarios para el despliegue hay que crear un proyecto en la plataforma *Visual Builder* y crear un repositorio de *git* dentro de nuestro proyecto en el que subiremos el código.

Ahora debemos configurar la *build* de nuestra aplicación. Como *Java* es un lenguaje compilado, será necesario crear una *pipeline* que encadene 2 *jobs*. Para estos *jobs* se ha configurado un *Build Executor Template* con las tecnologías necesarias, las cuales pueden verse en la figura B.6.

1. El primero de estos *jobs* se llama **Build**, y tiene como objetivo compilar el programa y obtener el archivo *jar* que se genera. Para ello se han realizado las siguientes configuraciones en las diferentes ramas *(Véase la figura B.7)*:
    - **Git**. Se selecciona el repositorio de código que se debe utilizar y se establece que cada vez que se haga un *commit*, se inicie este *job*.
    - **Steps**. Se ha creado una orden de *Maven* que ejecuta el comando *clean install* basándose en el fichero *pom.xml* de nuestro proyecto.
    - **After Build**. Guardan temporalmente los archivos almacenados en la carpeta *target*.
2. El segundo *job* se llama **Docker and Deploy**, que necesita el archivo generado por el *job* anterior para crear una imagen *Docker*, almacenarla en *Container Registry* y finalmente desplegar nuestra aplicación en *Kubernetes*. Las configuraciones realizadas han sido *(Véase la figura B.9 y B.10)*:
    - **Git**. Es igual que el utilizado en el paso anterior, con la diferencia de que esta vez no se ejecuta automáticamente después de un *commit*.
    - **Before Build**. Copia los archivos generados por la última *build* exitosa del *job Build*.
    - **Steps**. Se han configurado 3 tipos de acciones diferentes, las cuales son:
        - **Docker**. Se realizan 3 acciones *Docker* diferentes, la primera realiza el *login* en el *Container Registry*. La segunda construye la imagen *Docker* y la etiqueta como *latest*. Y la tercera finalmente publica la imagen generada en el *Container Registry*.
        - **OCIcli**. Permite al *CLI* de *Oracle Cloud Infraestructure* y por ende a nuestro *Cluster* de *Kubernetes*.
        - **Unix Shell**. Permite definir una serie de comandos para que se ejecuten. En nuestro caso, los comandos que hemos definido descargar la configuración de *kubernetes*, borran el *Ingress, Deployment* y el *Service* en caso de que existan. Después espera un minuto a que terminen de borrarse y despliega los nuevos. Los comandos ejecutados pueden verse en el listado A.3.

Una vez se tienen los dos *jobs* se procede a crear una *pipeline*, estableciendo que cuando termine de ejecutarse el *job Build*, se ejecute el *job Docker and Deploy (Véase la figura B.8)* . Una vez creado, simplemente debemos ejecutar el *pipeline*, o el *job Build* o hacer un *commit*.

Cuando termine de ejecutarse esta *pipeline*, en la figura 6.2 podemos ver que nuestro aplicación se ha desplegado correctamente en *Kubernetes*.

```
> kubectl get pods --sort-by=.metadata.creationTimestamp | tail -1
stream-connect-deployment-59fc7bdd7b-hqhhl   1/1     Running   0          12d
> kubectl get svc --sort-by=.metadata.creationTimestamp | tail -1
stream-connect-service     ClusterIP      10.96.51.184    <none>           80/TCP          12d
> kubectl get ingress
NAME                     CLASS    HOSTS                        ADDRESS           PORTS     AGE
stream-connect-ingress   <none>   streamconnect.avanttic.com   132.145.247.102   80, 443   12d
>
```

Figura 6.2: Despliegue *StreamConnect*

Conexión *Webhook* de *Jira*

El primero de los requisitos para conseguir conectarse al *Webhook* es que nuestra aplicación sea accesible desde internet, lo cual fue conseguido en la tarea anterior.

El segundo requisito es que nuestra aplicación cuente con un certificado *SSL*. Para lo cual se solicitó al departamento de infraestructura y sistemas de *Avanttic* para que registre en su *DNS* el subdominio *streamconnect.avanttic.com* y hacer que apunte a los *nameservers* de *Oracle Cloud*, vinculando la *IP 132.145.247.102* que proporciona el *Ingress* con dicho subdominio. También se solicito el certificado *Wildcard* que utiliza *Avanttic* por lo que se nos proporciono dos archivos, *avanttic.com.key* que contiene clave privada necesaria para utilizar el certificado y el archivo *avanttic.com.pem* que contiene el certificado.
Una vez tenemos todo esto, debemos crear un *tls-secret* que va a contener el certificado *SSL*. Para ello debemos ejecutar el comando que se ve en el listado 6.6.

```
1 kubectl create secret tls avanttic.com --cert avanttic.com.pem ↩
    ↪ --key avanttic.com.key
```

Listado 6.6: Comando para crear *Secret-TLS*

Una vez configuramos el *secret* debemos modificar el *Ingress*, especificando el nombre del *tls-secret* que se ha creado anteriormente una regla para que solo acepte peticiones que utilicen el *host streamconnect.avanttic.com*. Todos estos cambios pueden verse en el listado A.4.

A continuación, hay que configurar el *Webhook* de *Jira*, para ello debemos establecer el *endpoint* al que enviará las peticiones *POST* y el tipo de evento que queremos al que reaccione, en este caso, la creación de incidencias. Todo esto puede verse en la figura 5.5. Por último, se ha habilitado un *endpoint* simple llamado *issues*, que recibe la información en formato *JSON* y la imprime. Un ejemplo del *JSON* que se recibe puede verse en listado A.5.

Envío de Correos

Para realizar esta tarea ha sido necesario utilizar dos librerías más, *spring-boot-starter-thymeleaf* y *spring-boot-starter-mail* junto al *SMTP* de *Google*, cuya configuración ha sido externalizada y puede verse en el listado A.6.

Se han creado 2 plantillas *HTML*, una para peticiones urgentes y otra para peticiones estándar ubicadas en el directorio *resources/templates*. Ambas plantillas son rellenadas con datos relevantes obtenidos *Webhook*, estos datos son: resumen de la incidencia, descripción, empresa, prioridad, área y archivos adjuntos.

En el paquete *issue* se ha creado la clase *MessageService* que contiene el método *enviarMail*, cuyo código es visible en el listado A.7. Este método, recibe el mensaje del *KafkaListener* y un *Array* con los

destinatarios, y en función de la prioridad procesará la plantilla de correos urgente o la no urgente. Y enviará el mensaje a todos los destinatarios los cuales serán obtenidos al realizar una petición *GET* a *StreamSub*. Los correos son enviados por la cuenta de *gmail: streamconnectt@gmail.com*. El resultado puede verse en la figura 6.3.

(a) Correo Estándar **(b)** Correo Urgente

Figura 6.3: Ejemplos de los *Email* enviados

6.2.3. Revisión

En la tabla 6.6 se ha realizado una comparación de las horas que se habían estimado en la planificación de esta iteración 6.4 con las horas reales que ha tomado cada tarea. Podemos apreciar que algunas tareas han tomado más tiempo y otras menos. Destacando la tarea de despliegue de la aplicación en *Kubernetes* que ha sido más complicada de lo esperado y la conexión con el *Webhook* de *Jira*, puesto que tardaron más tiempo del esperado en proporcionarnos el certificado y fue más difícil de lo pensando activarlo.

Iteración	Requisito	Tareas	Horas Estimadas	Horas Reales
1	Desarrollar la Aplicación	Elegir librería Apache Kafka	16	16
		Crear Aplicación Base	10	10
		Conexión a Streaming	10	12
		Configurar Recursos Streaming Service	14	18
		Envío de Correos	12	10
	Automatizar el Despliegue	Crear Imagen Docker	16	8
		Desplegar Aplicación en Kubernetes	56	70
	Recibir Peticiones de Jira	Conexión Webhook Jira	24	38
Horas Totales			158	182

Tabla 6.6: Revisión Iteración 1

6.3. ITERACIÓN 2

A esta iteración ha sido denominada como *StreamSub*, y su propósito es desarrollar y desplegar una aplicación web, que permita gestionar los consultores, las suscripciones de los consultores, los *Streams*, las áreas y las asignaciones de las áreas a las particiones de los *Streams*.

6.3.1. Planificación

Esta iteración esta planificada para que comience el 17 de noviembre de 2021 y termine 21 de diciembre de 2021, aunque debido al retraso en alguna de las tareas de la iteración anterior comienza y termina más tarde. Para esta iteración, igual que en las anteriores se ha realiza una tabla 6.7 en la que se muestran los requisitos, las tareas en las que se desglosa cada requisito y las horas estimadas para la realización de cada tarea.

Iteración	Requisito	Tarea	Horas Estimadas
2	Diseñar la Aplicación y Crear la Base de Datos	Bocetos y Esquema de la Base de Datos	16
		Crear Aplicación Base	12
		Crear Base de Datos local y en Kubernetes	10
	Desarrollar la Aplicación	Desarrollar Back-End	50
		Desarrollar Front-End	28
		Envío de Correos	10
	Automatizar el Despliegue (CI/CD)	Crear Imagen Docker	16
		Desplegar la Aplicación en Kubernetes	56
Horas Totales			208

Tabla 6.7: Planificación Iteración 2

6.3.2. Resultados

Bocetos y Esquema de la Base de Datos

Esta tarea tiene la finalidad de facilitar al alumno el desarrollo de la aplicación. Para esto, se han realizado varios bocetos en la plataforma online *moqups*, que estiman el resultado final de la aplicación y sirven como base al alumno para el desarrollo. Los bocetos finales, que han resultado teniendo en cuentas las correcciones impuestas por el tutor pueden verse en las figuras: B.11, B.12.
En cuanto al esquema de la base de datos, podemos verlo en la figura 5.4. Muestra 4 tablas diferentes:

- **Consultores**. Compuesta por un *id_consultor* que actúa como clave primaria ,el *email* del consultor, el cual es único y la disponibilidad, que por defecto será *true*. Cuenta con una relación de uno a muchos con la tabla de suscriptores, permitiendo que un consultor tenga de 0 a N suscripciones.
- **Suscripciones**. Compuesta por 3 columnas que forman la clave primaria. *id_consultor*, que proviene de la tabla consultores. *topic* y *particion* que representan el *Stream* y la partición a la que se suscribe respectivamente.

- **Áreas**. Compuesta por el *id_area* que la identifica que a su vez hace de clave primaria y por el nombre del área, el cual es único. Tiene una relación de uno a muchos con la tabla de asignaciones, permitiendo que el área pueda estar asignada a 0 o N particiones de un *Stream*.
- **Asignaciones**. Compuesta 3 columnas que forman su clave primaria. La primera se llama *id_area* y proviene de la tabla *areas* e identifica el área. Las otras dos *particion* y *topic* definen la partición y el *Stream* a la que se le asigna el área.

No hay ninguna tabla que almacene la información de los *Streams*, puesto que se obtendrán desde *StreamConnect* en tiempo real.

Crear Aplicación Base

El objetivo de esta tarea es crear una aplicación base desde la cual iniciar el desarrollo de la aplicación final. A diferencia de la iteración anterior, para desarrollar esta aplicación utilizaremos *JavaScript*, *NodeJS* y el *Framework Express* en su versión 4.17.1. Los comandos utilizados para instalar los paquetes utilizados en *StreamSub* pueden verse en el listado A.9.

Para esta tarea se ha creado un servidor simple de *Express* que escucha en el puerto 3000, el cual se lanza desde el archivo *index.js*. También se ha creado la carpeta *router*, en la que se encuentra el archivo *rutas.js* que contiene todas las rutas de nuestra aplicación web. Se ha habilitado un *endpoint* llamado hola, que devuelve el mensaje "Hola desde StreamSub!", el resultado puede verse en la figura B.13 .

Crear Base de Datos Local y en *Kubernetes*

La finalidad de esta tarea es crear la base de datos que hemos especificado en la primera tarea de esta iteración. Se creará tanto en el *cluster* de *Kubernetes* como en local, permitiendo trabajar al alumno sin necesidad de publicar la aplicación.

Para el despliegue en local se instaló *MySQLServer*, se creo la base de datos llamada *streamconnect*. Después en la consola de *MySQL* se ejecutaron los comandos necesarios para crear las tablas, los cuales pueden verse en en los listados 5.1, 5.2, 5.3, 5.4.

Para el despliegue en *Kubernetes* se ha creado el archivo *mysql-stateful-v2.yaml*, que al igual que un *Deployment*, permite desplegar *pods* en *Kubernetes*, con la diferencia de que un *StatefulSet* permite hacer a un *pod* persistente asociándolo a un volumen de datos, el código puede verse en el listado A.10. También fue necesario crear un *service*, que permitirá la comunicación con el resto de *pods*, para esto nos proporcionará una *IP* interna del *cluster*, en este caso ha sido *10.96.49.81*. El código de *mysql-service.yaml* puede encontrarse en el listado A.11. Finalmente también ha sido necesario crear un *secret opaque*, que almacena la contraseña de la base de datos. El código puede encontrarse en el listado A.12. Para desplegar ambos objetos se ejecutaron los comandos del listado 6.7 en el *Cloud Shell* de *Oracle*.

```
1 kubectl  apply -f mysql-stateful-v2.yaml
2 kubectl  apply -f mysql-service.yaml
3 kubectl  apply -f mysql-secrets.yaml
```

Listado 6.7: Despliegue *MySQL Kubernetes*

Una vez los objetos se desplieguen desde *Cloud Shell* deberemos ejecutar el comando 6.8. Que permite ejecutar una terminar interactiva en el pod de *MySQL* y acceder a él. En el que ejecutaremos los comandos 5.1, 5.2, 5.3, 5.4. El resultado de esta tarea puede verse en la figura B.16.

```
1 kubectl exec -it mysql-0 -n mysql -- /bin/bash
```

Listado 6.8: Abrir terminal *MySQL*

Desarrollar el *Back-End*

El objetivo de esta tarea, como su nombre indica es crear el *Back-End* de *StreamSub*. Entro los hitos más relevantes podemos destacar los siguientes:

- **Base de Datos**. Se ha creado un directorio llamado *database* que contiene dos archivos, el primero de ellos es *conexion.js* que utiliza un pool de conexiones para conectarse a nuestra aplicación, el cual puede verse en el listado A.16. Los datos sensibles han sido externalizados en el archivo *.env*, cuyo contenido es visible en listado A.13. El segundo de ellos se llama *DBUtil.js* que condensa todos los métodos que realizan consultas a la base de datos. Un ejemplo de estos métodos es *getConsultores* A.14, que devuelve un array con los datos de los consultores.
- **Rutas**. Se han creado una serie de controladores para los métodos *GET, POST* y *DELETE*. Puesto que esta tarea como la siguiente se han realizado de forma paralela quedaron definidos todos los *endpoints* necesarios en el *Front-End*. Un ejemplo de estos métodos puede verse en el listado A.17.

Desarrollar el *Front-End*

Esta tarea se realizo en paralelo con la tarea anterior, y tiene como objetivo, utilizando bocetos generados anteriormente crear una interfaz sencilla y minimalista que permita: gestionar los consultores, las suscripciones, los *Stream* las áreas y las asignaciones. Para ello nos hemos valido de *Bootstrap 5* y el motor de plantillas *ejs*, que permite reutilizar y renderizar código *HTML*. Para gestionar la vistas, se ha creado la carpeta *views*, que contiene un archivo *ejs* por cada una de las páginas que se han diseñado y el directorio *templates*, que contiene los archivos *ejs* que se reutilizarán, es decir, el de la cabecera, el *footer* y la *navbar*.

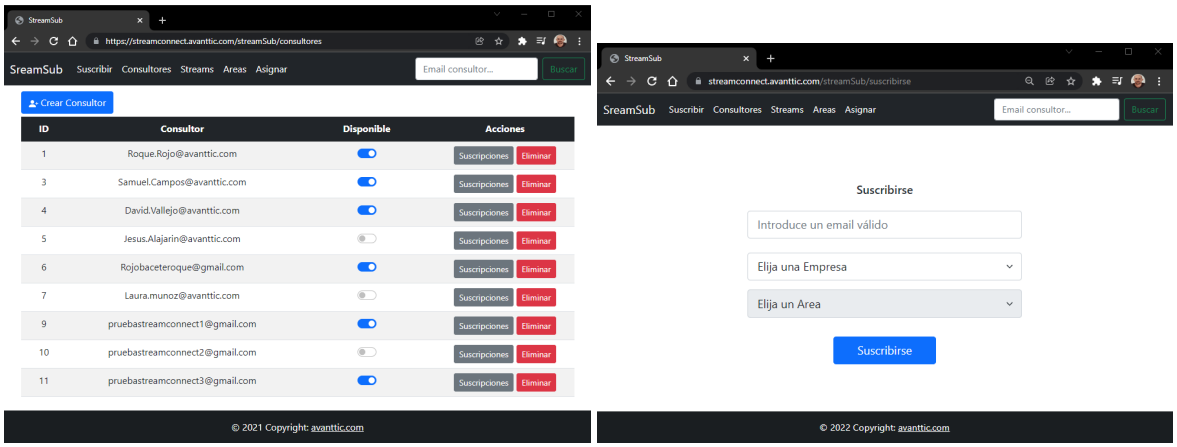
A su vez, todas las vistas poseen un archivo *JavaScript* que marca el comportamiento elementos de la web. Estos archivos se encuentran alojados *js* que se encuentra en la carpeta *public*, que contiene todos los archivos *CSS* y *js* necesarios para el funcionamiento de *Bootstrap*. En el listado A.18 podemos ver un ejemplo del contenido de estos archivos.

Todas las vistas siguen la siguiente estructura 6.9, en donde en la primera línea se llama al contenido de la cabecera, que a su vez incluye la *navbar*, posteriormente se incluye el código *HTML, Bootstrap* y *ejs (véase el listado A.19)* y se ejecuta el código *JavaScript* correspondiente a la vista renderizada. Finalmente se llama al *footer*, lo que completará la vista. En la figura 6.4, podemos ver un ejemplo de los resultados obtenidos.

```
1 <%- include('template/cabecera') %>
2   <!-- código HTML \& Bootstrap \& ejs-->
3   <script type="text/javascript" src="js/vista.js"></script>
```

```
4 <%- include('template/footer') %>
```

Listado 6.9: Estructura vistas



(a) Vista Consultores **(b)** Vista Suscribirse

Figura 6.4: Ejemplo Vistas *StreamSub*

Crear Imagen *Docker* de la Aplicación

Al igual que la iteración anterior, para contenerizar nuestra aplicación ha sido necesario crear un archivo *Dockerfile* en nuestro proyecto, el cual puede consultarse en el listado 6.10. En el cabe destacar que se utilizará una máquina base con *NodeJS* instalado, se expondrá el puerto 3000 y se ejecuta el comando *chmod* para obtener todos los permisos sobre los archivos, puesto que en el despliegue daba un error sobre que carecía de algunos permisos necesarios.

```
1 FROM node
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
7 RUN chmod -R a+rwx /app
8 CMD ["npm","start"]
```

Listado 6.10: *Dockerfile StreamSub*

Desplegar aplicación en *Kubernetes*

En esta iteración ha sido necesario definir el *Deployment* y el *Service* y modificar levemente el *Ingress*, puesto que lo compartirán *StreamConnect* y *StreamSub*. El contenido de el *Deployment* y el *Service* es similar al de *StreamConnect*, los cuales pueden consultarse en los listados A.20 y A.21. En cuanto al *Ingress* ha sido modificado para permitir alojar 2 *Back-End* simultáneamente. Basándose en la ruta de la *URL* introducida. Las que comiencen con */StreamConnect*, las redirigirá al *Service* de *StreamConnect*. Mientras que las que comiencen por */StreamSub* las redirigirá al *Service* de *StreamSub*. Las modificaciones del *Ingress* pueden verse en el listado 6.11.

```
1 apiVersion: extensions/v1beta1
2 kind: Ingress
```

```yaml
metadata:
  name: stream-connect-ingress
spec:
  tls:
  - secretName: avanttic.com
    hosts:
    - streamconnect.avanttic.com
  rules:
  - host: streamconnect.avanttic.com
    http:
      paths:
      - path: /streamConnect
        backend:
          serviceName: stream-connect-service
          servicePort: 80
      - path: /streamSub
        backend:
          serviceName: stream-sub-service
          servicePort: 80
```

Listado 6.11: *Ingress*

A continuación, se ha de crear otro repositorio *Git* en el proyecto de *Visual Builder* para subir el código de la aplicación. Una vez subido es necesario configurar el despliegue de la aplicación. A diferencia de *StreamConnect*, *StreamSub* utiliza el lenguaje *JavaScript*, el cual es un lenguaje de programación interpretado por lo que no será necesario crear un *job* para su compilación. El *job* creado ha recibido el nombre de *streamSub_deployToKubernetes* y han sido necesarias 2 configuraciones *(Véase la figura B.14 y B.15)*:

- **Git**. Se establece que el repositorio que ha de utilizar es el de *StreamSub* y que se ejecute cada vez que se realice un *commit*.
- **Steps**. Se han configurado 3 acciones diferentes las cuales son:
  - **Docker**. Divididos en 3, el primero realiza el *login* al *ContainerRegistry*, el segundo construye la imagen *Docker* basándose en el *Dockerfile* hay definido y el último publica la imagen generada en el *Container Registry*.
  - **OCIcli**. Permite acceder a *OCI* ya ejecutar comandos en nuestro cluster.
  - **Unix Shell**. A diferencia de la iteración anterior, no actualizaremos el *Ingress*, puesto que lo comparte con *StreamConnect*. Estos comandos pueden verse en el listado A.15

Una vez configurado el *job* para desplegar la aplicación en *Kubernetes* simplemente hay que darle a *run job* o realizar un *commit* en nuestra aplicación para que se despliegue. El resultado de este despliegue puede verse en la figura 6.5.

```
> kubectl get pods | tail -2
stream-connect-deployment-59fc7bdd7b-9spml   1/1     Running   0          43h
stream-sub-deployment-749fbfb675-zjw99       1/1     Running   0          14d
> kubectl get svc | tail -2
stream-connect-service     ClusterIP      10.96.160.93    <none>           80/TCP          43h
stream-sub-service         ClusterIP      10.96.18.19     <none>           80/TCP          14d
> kubectl get ingress
NAME                     CLASS    HOSTS                        ADDRESS           PORTS     AGE
stream-connect-ingress   <none>   streamconnect.avanttic.com   132.145.247.102   80, 443   43h
```

Figura 6.5: Resultado Despliegue *StreamSub*

Envío de Correos

Esta tarea tiene como objetivo notificar por correo electrónico a los consultores cuando sus suscripciones han sido modificadas, mostrando en el correo los *Streams* y las áreas a las que se encuentra suscrito. Para ello se ha hecho uso del paquete *nodemailer* y se ha creado una clase llamada *sendMail.js*, cuyo contenido puede verse en el listado A.22. Esta clase contiene la función *enviarCorreo* que toma como parámetro el email del consultor al que se le modifican las suscripciones y las suscripciones actuales. Y con el paquete *ejs* se renderiza una plantilla similar a la utilizada en *StreamConnect*, la cual se ha ubicado en el directorio */templates* con los datos de las suscripciones actuales. Los datos de la configuración del *SMTP* de *Google* se encuentran en el archivo *.env*, cuyo contenido se refleja en el listado A.13. El remitente de los correos será *streamsubb@gmail.com* y se enviarán cada vez que se modifiquen las suscripciones de un consultor. En la figura 6.6, se puede ver un ejemplo de los correos que enviará *StreamSub*.

Figura 6.6: Ejemplo Correo enviado al modificar las suscripciones

6.3.3. Revisión

La tabla 6.8, muestra una comparación entre las horas que se estimaron para cada tarea en la planificación. La mayoría de las tareas han llevado menos tiempo del pensado, puesto que el alumno se ha familiarizado más con la tecnologías, sobre todo con el despliegue de aplicaciones en *Kubernetes*, por lo que se recupera el tiempo que se perdió en la iteración anterior y se gana algo de tiempo *extra*.

6.4. ITERACIÓN 3

Esta iteración recibe el nombre de Comunicación *StreamSub-StreamConnect*, y tiene como objetivo crear los microservicios necesarios para que *StreamSub* y *StreamConnect* puedan intercambiar información y gestionar los recursos de *Streaming*.

Iteración	Requisito	Tarea	Horas Estimadas	Horas Reales
2	Diseñar la Aplicación y Crear la Base de Datos	Bocetos y Esquema de la Base de Datos	16	16
		Crear Aplicación Base	12	8
		Crear Base de Datos local y en Kubernetes	10	14
	Desarrollar la Aplicación	Desarrollar Back-End	50	50
		Desarrollar Front-End	28	28
		Envío de Correos	10	8
	Automatizar el Despliegue (CI/CD)	Crear Imagen Docker	16	14
		Desplegar la Aplicación en Kubernetes	56	50
Horas Totales			208	188

Tabla 6.8: Revisión Iteración 2

6.4.1. Planificación

Esta iteración esta planificada para que comience el día 22 de diciembre de 2021 y que termine el 3 de enero de 2022. Y esta constituido por 2 tareas, las cuales quedan reflejadas en la tabla 6.9 junto al tiempo estimado en horas para cada tarea.

Iteración	Requisito	Tarea	Horas Estimadas
3	Desarrollar y Consumir Microservicios	Microservicios StreamConnect	12
		Microservicios StreamSub	12
Horas Totales			24

Tabla 6.9: Planificación Iteración 3

6.4.2. Resultados

Microservicios StreamConnect

Esta tarea tiene como objetivo definir los microservicios necesarios para que *StreamSub* pueda crear, eliminar y solicitar información de los *Streams*. Para esto se han definido 3 *endpoints* en *IssueControlller.java*. Los cuales son: */crearTopic, /borrarTopic* y *topics*. Cada uno llama a su método correspondiente de la clase *IssueService.java*.

- **crearTopic**. Este método recibe como parámetros el nombre del *Stream* y el numero de particiones. Con estos datos se creará un *NewTopic*, el cual sera utilizado por el método *createTopics* del administrador,que creará el *Stream*.
- **borrarTopic**. Recibe como parámetro el nombre del *Stream* que se desea borrar. El nombre será utilizado por el método *deleteTopics* del administrador, que procederá a borrar *Stream*.
- **getTopics**. Este método devuelve un *JSON* con el nombre de cada uno de los *Streams* y el número de particiones. Para ello hace uso del método *listTopics* del administrador, que devuelve una lista con el nombre los *Streams* y un productor, que puede obtener el número de particiones de un *Stream*.

El código de sendos métodos puede verse en el listado A.23.

Por otra parte, en *StreamSub*, se han definido una serie métodos que permiten utilizar los microservicios de *StreamConnect*. En el listado A.26 se puede ver un ejemplo de un método que realiza una petición *GET* al *endpoint* llamado *topics* de *StreamConnect* y obtiene los datos deseados en formato *JSON*.

En el listado A.25 podemos ver otro ejemplo, en este caso, es de un método definido para realizar una petición *POST* al *endpoint /borrarTopic* de *StreamConnect*. El formulario controla que los *Streams* que se desean borrar existan, al igual que el formulario de crear un *Stream* se asegura de que el *Stream* que se desea crear no exista con anterioridad antes de realizar la petición.

Microservicios StreamSub

Esta tarea tiene como objetivo definir los microservicios necesarios para que *StreamConnect* pueda obtener información sobre los suscriptores y las asignaciones. Para esto han habilitado en el archivo *rutas.js* 2 *endpoint* que atienden peticiones *GET* y realizan consultas a la base de datos, los cuales son:

- **/asignacion/*:{topic}*/*:{area}***. Proporciona una array de enteros con las particiones del *Stream* que tiene asignada ese área.
- **/suscriptores/*:{topic}*/*:{particion}***. Obtiene un array de *String* con las direcciones de correo electrónico de los consultores suscritos a esa partición.

En el listado A.27 podemos ver el código de los 2 *endpoints*, que realizan consultas a la Base de Datos y después envía los datos obtenidos.

Para poder realizar peticiones a *StreamSub*, en *StreamConnect* se ha habilitado una clase llamada *SuscriptoresController.java* que contiene los métodos para que realizar las peticiones *GET* a *StreamSub*, estos métodos pueden verse en el listado A.24.

El primero de estos métodos se utiliza para saber a que partición hay que mandar el mensaje, lo cual puede verse en el listado 6.2. El segundo en cambio, es utilizado para obtener los destinatarios de los emails en el método *KafkaListener*, el cual puede verse en el listado 6.1.

6.4.3. Revisión

En la tabla 6.10 se realiza una comparación de las horas estimadas con las horas reales que ha llevado la realización de las tareas. Como se puede ver aunque se ha recortado tiempo en la primera, la creación de los servicios de la segunda ha llevado más tiempo del esperado.

Iteración	Requisito	Tareas	Horas Estimadas	Horas Reales
3	Desarrollar y Consumir Microservicios	Microservicios StreamConnect	12	10
		Microservicios StreamSub	12	14
Horas Totales			24	24

Tabla 6.10: Revisión Iteración 3

CAPÍTULO 7

Conclusiones

En este último capítulo se realizará una valoración final de los resultados alcanzados durante el desarrollo de este *TFG*. Para esto, revisaremos los objetivos que se expusieron en el capítulo 2. Y se finalizará el capítulo con una reflexión sobre posibles actualizaciones o mejores que se podrían incorporar al proyecto en un futuro.

7.1. OBJETIVOS ALCANZADOS

En el capítulo 2 de este documento, se estableció que el objetivo principal de este *TFG* era el diseño y desarrollo de un sistema que notifique automáticamente las peticiones de servicio a los consultores que se encuentren disponibles, filtrando entre los que están suscritos a un buzón, que representa e identifica al cliente dentro de *Avanttic*.

Este objetivo principal, se divide a su vez en objetivos más pequeños y específicos. La consecución de estos objetivos deriva en el cumplimiento del objetivo general. En esta sección se volverán a presentar los objetivos específicos junto a la justificación de su consecución.

- **Estudiar e instalar las tecnologías y herramientas relativas al desarrollo, despliegue e integración del sistema en contexto de producción.** Las principales tecnologías con las que ha tenido que familiarizarse el alumno han sido *Kubernetes*, *Docker*, *Streaming*, *SpringBoot* concretamente con la librería de *Apache Kafka*, *NodeJS* y su *framework Express*. Este objetivo ha sido abordado en la iteración 0, en la que se realizo un estudio previo de todas las tecnologías que se iban a utilizar, mediante los cursos y a lo largo de todo el proyecto.
- **Crear, desarrollar, desplegar e integrar el módulo de notificación automatizada de peticiones de servicio.** El módulo que se ha desarrollado recibe el nombre de *StreamConnect* el cual notifica mediante un correo electrónico a los consultores disponibles y suscritos al buzón en el cual se genera la incidencia, notificando de forma diferente los urgentes de los no urgentes. Se encuentra desplegado en *OKE* siguiendo los principios de *CI/CD* utilizando para ello *Visual Builder Studio*. Tanto la creación, como el desarrollo y el despliegue ha sido abordado en la iteración 1, en cambio, la integración con *StreamSub* abordo en la iteración 3.
- **Crear, desarrollar, desplegar e integrar el módulo para gestionar los consultores, suscripciones, *Streams*, áreas y asignaciones.** El módulo que se ha desarrollado recibe el nombre de *StreamSub*. Permite gestionar los consultores, suscripciones, *Streams*, áreas y asignaciones mediante interfaz sencilla y minimalista elaborada con *Express* y *Bootstrap*. Se

encuentra desplegado en *OKE* siguiendo los principios de *CI/CD*, utilizando para ello *Visual Builder Studio*. La creación, desarrollo y despliegue se ha abordado a lo largo de la iteración 2, mientras que la integración con *StreamConnect* se ha llevado a cabo en la iteración 3.

- **Creación de la documentación necesaria que detalle el trabajo realizado a lo largo de este proyecto**. El propio documento se encarga de cumplir este objetivo. Los 6 capítulos y sus complementos constituyen la documentación que se ha llevado acabo a lo largo del proyecto. Su realización abarca los 5 meses que ha durado el programa *FORTE*.

Una vez conseguidos estos objetivos específicos, podemos decir sin miedo a equivocarnos que el objetivo principal ha sido alcanzado con éxito.

7.2. JUSTIFICACIÓN DE COMPETENCIAS ADQUIRIDAS

En esta sección se justifican las competencias académicas trabajadas durante el desarrollo de este proyecto.

TI1: ***Capacidad para comprender el entorno de una organización y sus necesidades en el ámbito de las tecnologías de la información y las comunicaciones***. Para la realización de este proyecto ha sido necesario comprender las herramientas y procesos que utiliza la empresa a la hora de tratar con las incidencias, la forma en la que se estructuran las incidencias en *Jira* y el flujo que siguen hasta que finalmente llegan a un consultor. Esto me ha permitido identificar el problema del cuello de botella, y elaborar una solución en función a las necesidades de la empresa. El problema y la solución propuesta pueden verse en el capítulo 1.

TI3: ***Capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas.***. Durante el desarrollo del proyecto se ha seguido la metodología *Avanttic FORTE*, expuesta en el capítulo 4. Puesto que aunque se plantearon otras, como *Agile*, utilizando el marco de trabajo *SCRUM* se descartaron debido a que *Avanttic FORTE* es una metodología que ha sido diseñada específicamente para cumplir con las necesidades de la empresa y a diferencia de *SCRUM* se adapta perfectamente al entorno de trabajo sin necesidad de realizar cambios para adaptarla.

TI5: ***Capacidad para seleccionar, desplegar, integrar y gestionar sistemas de información que satisfagan las necesidades de la organización, con los criterios de coste y calidad identificados***. Esta competencia ha sido adquirida a la hora de seleccionar las tecnologías y herramientas más adecuadas para desarrollar los módulos de *StreamConnect* y *StreamSub*. Además sendos módulos han sido desplegados en *Oracle Cloud* siguiendo los principios de *CI/CD* puesto que la aplicación es susceptible a ser mejorada a posteriori y se ha conseguido integrar ambos módulos permitiendo intercambiar información. Todo esto dentro del presupuesto que se le había asignado a la solución del problema que se planteaba en capítulo 1.

7.3. TRABAJO FUTURO

Como se planteó en el capítulo 1, este *TFG* se centra el resolver el caso de uso de eliminar el cuello de botella existente en la notificación a los consultores disponibles. No obstante, en esta sección se plantearán algunas posibles implementaciones o mejoras futuras.

1. **Obtención de Estadísticas.** Al identificar la empresa como un *Stream* y las áreas a las *Avanttic* ofrece servicio como las particiones. Se puede habilitar un módulo que utilizando *Confluent Metrics Reporter* permita obtener métricas sobre las incidencias que generan las empresas, una sola empresa, las áreas en general o las áreas especificas en un intervalo X de tiempo. Permitiendo realizar informes y gráficos de valor para la empresa, utilizando por ejemplo *Grafana*.
2. **Ampliación a otras Plataformas.** Este *TFG* se centra la plataforma de *Jira*, no obstante algunos clientes de *Avanttic* utilizan otras plataformas para gestionar los incidentes y las peticiones de servicio, esto obliga a que una persona tenga que trasladar manualmente la incidencia a *Jira*. Para solucionar esto se podría habilitar en *StreamConnect* la posibilidad de recibir las peticiones de esas plataformas. Y utilizando la *API* de *Jira* crear de nuevo esa incidencia, una vez creada se iniciaría el flujo que puede verse en la figura C.1
3. **Machine Learning.** Una posible mejora sería desarrollar un modelo de *Machine Learning* con la tecnología *Oracle Machine Learing* que, utilizando como datos la empresa y el área de la incidencia que ha sido asignada a un consultor en *Jira*, poder entrenarlo para obtener una lista con los consultores más afines a aceptar las futuras incidencias. Este módulo permitiría sustituir gradualmente a *StreamSub*, automatizando aun más el proceso.

7.4. VALORACIÓN PERSONAL

Este proyecto realizado bajo la dirección *Avanttic* Consultoría Tecnológica S.L ha sido mi primer proyecto en un entorno laboral real. Gracias a este proyecto he podido aprender aprender nuevas tecnologías, herramientas y adquirir experiencia laboral, complementando los conocimientos adquiridos durante los años de mi carrera como estudiante de Ingeniería Informática.

Me parece que el convenio *FORTE* es lo mejor que le puede pasar a un estudiante que se encuentra terminando la carrera. Gracias a esta beca he podido formarme junto a grandes profesionales que me han ayudado a prepararme para el mundo laboral. También me gustaría destacar y agradecer el trato que he recibido de la empresa, puesto que desde el primer minuto me he sentido integrado pasando a formar parte de una especie de pequeña *"familia"*. Y en definitiva, decir que me llevo una experiencia muy positiva del programa *FORTE*.

Y en cuanto al *TFG*, me ha fascinado todo el tema de *Kubernetes*, *Docker*, el despliegue automatizado de aplicaciones y la *CI/CD*. Creo que este tipo de tecnología y aproximación tiene mucho futuro y me ha encantado aprenderla.

Bibliografía

[1] Luis Joyanes Aguilar. Computación en la nube e innovaciones tecnológicas. *El nuevo paradigma de la Sociedad del Co*, 2011.

[2] Luis Joyanes Aguilar. *Computación en la nube: estrategias de Cloud Computing en las empresas*. Marcombo, May 2012. ISBN: 8426718930.

[3] Josh Long & Kenny Bastani. *Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry*. O'Reilly Media, June 2016. ISBN: 9781449374648.

[4] Red Hat. *¿Qué son los Microservicios?* Artículo disponible en URL: https://www.redhat.com/es/topics/microservices/what-are-microservices, 2021. Ultimo acceso: Noviembre 2021.

[5] ITIC. *Cost of Hourly Downtime*. Estudio disponible en URL: https://itic-corp.com/blog/2016/08/cost-of-hourly-downtime-soars-81-of-enterprises-say-it-exceeds-300k-on-average/, 2021. Ultimo acceso: noviembre 2021.

[6] Michał Tomasz Jakóbczyk. *Practical Oracle Cloud Infrastructure*. Springer, 2020.

[7] Greg Lim. *Beginning MERN Stack: Build and Deploy a Full Stack MongoDB, Express, React, Node.js App*. The Anima Group, June 2021. ISBN: 979-8523625503.

[8] Steve Mcconnell. *Desarrollo y Gestión de Proyectos Informáticos*. McGraw-Hill Spanish, May 1997. ISBN: 8448112296.

[9] Julie Meloni. *HTML, CSS, and JavaScript All in One: Covering HTML5, CSS3, and ES6*. McGraw-Hill Spanish, June 2019. ISBN: 0672338084.

[10] Andrés Leonardo Oviedo Briones. *Estudio de las ventajas del manejo de cloud computing (computación en la nube)*. Estudio disponible en URL: http://repositorio.puce.edu.ec/handle/22000/3372, 2018. Ultimo acceso: noviembre 2021.

[11] Neha Narkhede&Gwen Shapira&Todd Palino. *Kafka - The Definitive Guide: Real-time data and stream processing at scale*. O'Reilly Media, May 2017. ISBN: 9781491936160.

[12] Blas León Romero. *El libro práctico de bases de datos*. Apress, February 2021. ISBN: 8409334690.

[13] Jesús Salido. *Curso: LaTeX esencial para preparación de TFG, Tesis y otros documentos académicos*. URL: http://visilab.etsii.uclm.es/?page_id=1468, 2010. Último acceso: sep. 2021.

[14] ESIC Business&Marketing School. *Modelo Entidad-Relación: Descripción y Aplicaciones*. Artículo disponible en URL: https://www.esic.edu/rethink/tecnologia/modelo-entidad-relacion-descripcion-aplicaciones, 2022. Ultimo acceso: Noviembre 2021.

[15] Kevin J Smith. *The Practical Guide To World-Class IT Service Management*. The Anima Group, March 2017. ISBN: 8426718930.

[16] Hatma Suryotrisongko and Meli Dyah Qoiru Mucharomah. Ideal help desk/service desk in e-government and service quality: A literature review. In *2017 11th International Conference on Information Communication Technology and System (ICTS)*, pages 203–208, 2017.

[17] Tecnova. *¿Qué es CI/CD?* Artículo disponible en URL: https://www.tecnova.cl/2021/05/26/que-es-ci-cd/, 2021. Ultimo acceso: Diciembre 2021.

ANEXOS

ANEXO A

Listados Complementarios

En este anexo se mostrarán los listados demasiado voluminosos para estar contenidos en el texto principal. Estará divido en las diferentes zonas que tiene la memoria

A.1. ITERACIÓN 1

```
@Configuration
public class KafkaConfiguration {
  // Valores definidos en application.yml
  @Value("${my.configuracionKafka.credenciales}")
  private String credenciales;
  @Value("${my.configuracionKafka.bootstrapServer}")
  private String bootstrapServer;
  @Value("${my.configuracionKafka.protocol}")
  private String protocol;
  @Value("${my.configuracionKafka.mechanism}")
  private String mechanism;
  @Value("${my.configuracionKafka.concurrency}")
  private int concurrency;
  // Properties
  public Map<String, Object> producerProperties() {
    Map<String, Object> producerProperties = new HashMap<>();
    producerProperties.put(ProducerConfig.
        ↪ BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
    producerProperties.put(ProducerConfig.
        ↪ KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    producerProperties.put(ProducerConfig.
        ↪ VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    producerProperties.put("security.protocol", protocol);
    producerProperties.put("sasl.mechanism", mechanism);
    producerProperties.put("sasl.jaas.config", credenciales);
    return producerProperties;
  }
  public Map<String, Object> consumerProperties() {
    Map<String, Object> consumerProperties = new HashMap<>();
    consumerProperties.put(ConsumerConfig.
        ↪ BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
    consumerProperties.put(ConsumerConfig.GROUP_ID_CONFIG,
        ↪ "consumer");
```

```
29      consumerProperties.put(ConsumerConfig. ↩
            ↪ KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
30      consumerProperties.put(ConsumerConfig. ↩
            ↪ VALUE_DESERIALIZER_CLASS_CONFIG, ↩
            ↪ StringDeserializer.class);
31      consumerProperties.put(ConsumerConfig.
32      ALLOW_AUTO_CREATE_TOPICS_CONFIG, "false");
33      consumerProperties.put("security.protocol", protocol);
34      consumerProperties.put("sasl.mechanism", mechanism);
35      consumerProperties.put("sasl.jaas.config", credenciales);
36      return consumerProperties;
37    }
38    public Map<String, Object> adminProperties() {
39      Map<String, Object> adminProperties = new HashMap<>();
40      adminProperties.put(ConsumerConfig. ↩
            ↪ BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
41      adminProperties.put("security.protocol", protocol);
42      adminProperties.put("sasl.mechanism", mechanism);
43      adminProperties.put("sasl.jaas.config", credenciales);
44      return adminProperties;
45    }
46    // Consumer
47    @Bean
48    public ConsumerFactory<String, String> consumerFactory() {
49      return new DefaultKafkaConsumerFactory<>(consumerProperties());
50    }
51    @Bean
52    public ConcurrentKafkaListenerContainerFactory<String, ↩
        ↪ String> kafkaListenerContainerFactory() {
53      ConcurrentKafkaListenerContainerFactory<String, String> ↩
            ↪ factory = new ↩
            ↪ ConcurrentKafkaListenerContainerFactory<>();
54      factory.setConsumerFactory(consumerFactory());
55      factory.setConcurrency(concurrency);
56      return factory;
57    }
58    // Producer
59    @Bean
60    public KafkaTemplate<String, String> createTemplate() {
61      DefaultKafkaProducerFactory<String, String> producerFactory ↩
            ↪ = new DefaultKafkaProducerFactory<String, String>(
62          producerProperties());
63      KafkaTemplate<String, String> template = new ↩
            ↪ KafkaTemplate<>(producerFactory);
64      return template;
65    }
66    // Admin
67    @Bean
68    public AdminClient admin() {
69      AdminClient admin = AdminClient.create(adminProperties());
70      return admin;
71    }
```

```
}
```

Listado A.1: Configuración Recursos de *Streaming*

```
@RestController
@CrossOrigin(origins = "*", methods = { RequestMethod.GET, RequestMethod.POST })
@RequestMapping("/streamConnect")
public class IssueController {
  private static final Logger log = LoggerFactory.getLogger(IssueController.class);
  @GetMapping("/hola")
  public String getHola() {
    return "Hola Mundo!";
  }
  @PostMapping("/issues")
  public String escucharIssues(@RequestBody String payload)
      throws InterruptedException, ExecutionException, ParseException, URISyntaxException {
      log.info("Se ha recibido un peticion de servicio.");
    return "Se ha recibido una petición de servicio";
  }
}
```

Listado A.2: *Issue Controller* Iteración 0

```
if mkdir $HOME/.kube ; then
  echo "Downloading kubeconfig" fin
fi
oci ce cluster create-kubeconfig --cluster-id ocid1.cluster.oc1.eu-frankfurt-1.aaaaaaaaafrdozjqme 4dmojsgftgmztdga2wenzsmmzgiyzxgc4tqnbwhfrt --file $HOME/.kube/config --region eu-frankfurt-1
export KUBECONFIG=$HOME/.kube/config
kubectl delete -f streamConnect-ingress.yaml
kubectl delete -f streamConnect-service.yaml
kubectl delete -f streamConnect-deployment.yaml
sleep 60
kubectl apply -f streamConnect-deployment.yaml
kubectl apply -f streamConnect-service.yaml
kubectl apply -f streamConnect-ingress.yaml
sleep 20
```

Listado A.3: *Unix Shell StreamConnect*

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: stream-connect-ingress
spec:
  tls:
  - secretName: avanttic.com
    hosts:
```

```
    - streamconnect.avanttic.com
  rules:
  - host: streamconnect.avanttic.com
    http:
      paths:
      - path: /streamConnect
        backend:
          serviceName: stream-connect-service
          servicePort: 80
```

Listado A.4: Cambios en el *Ingress* para utilizar el *TLS-Secret*

```
{
   "timestamp":1639053724362,
   "webhookEvent":"jira:issue_created",
   "issue_event_type_name":"issue_created",
   "user":{
      "self":"https://testavt.atlassian.net/rest/api/2/user? ↩
         ↪ accountId=6138a222f6070d006b6c8cb5",
      "accountId":"6138a222f6070d006b6c8cb5",
      "avatarUrls":{
         "48x48":"https://secure.gravatar.com/avatar/ ↩
            ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A% ↩
            ↪ 2F%2Favatar-management--avatars.us-west-2.prod. ↩
            ↪ public.atl-paas.net%2Finitials%2FRB-1.png",
         "24x24":"https://secure.gravatar.com/avatar/ ↩
            ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A% ↩
            ↪ 2F%2Favatar-management--avatars.us-west-2.prod. ↩
            ↪ public.atl-paas.net%2Finitials%2FRB-1.png",
         "16x16":"https://secure.gravatar.com/avatar/ ↩
            ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A% ↩
            ↪ 2F%2Favatar-management--avatars.us-west-2.prod ↩
            ↪ .public.atl-paas.net%2Finitials%2FRB-1.png",
         "32x32":"https://secure.gravatar.com/avatar/ ↩
            ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A% ↩
            ↪ 2F%2Favatar-management--avatars.us-west-2.prod ↩
            ↪ .public.atl-paas.net%2Finitials%2FRB-1.png"
      },
      "displayName":"Roque Rojo Bacete",
      "active":true,
      "timeZone":"Etc/GMT",
      "accountType":"atlassian"
   },
   "issue":{
      "id":"10045",
      "self":"https://testavt.atlassian.net/rest/api/2/10045",
      "key":"TP-35",
      "fields":{
         "statuscategorychangedate":"2021-12-09T12:42:04.271+0000",
         "issuetype":{
            "self":"https://testavt.atlassian.net/rest/api/2/ ↩
               ↪ issuetype/10001",
            "id":"10001",
```

```
            "description":"Tasks track small, distinct pieces of work.",
            "iconUrl":"https://testavt.atlassian.net/secure/viewavatar?size=medium&avatarId=10318&avatarType=issuetype",
            "name":"Task",
            "subtask":false,
            "avatarId":10318,
            "entityId":"e0a06e34-3c8a-43e3-9dcb-68d9b0a46f7e",
            "hierarchyLevel":0
         },
         "timespent":null,
         "project":{
            "self":"https://testavt.atlassian.net/rest/api/2/project/10000",
            "id":"10000",
            "key":"TP",
            "name":"Test Project",
            "projectTypeKey":"software",
            "simplified":true,
            "avatarUrls":{
               "48x48":"https://testavt.atlassian.net/secure/projectavatar?pid=10000&avatarId=10425",
               "24x24":"https://testavt.atlassian.net/secure/projectavatar?size=small&s=small&pid=10000&avatarId=10425",
               "16x16":"https://testavt.atlassian.net/secure/projectavatar?size=small&s=small&pid=10000&avatarId=10425",
               "32x32":"https://testavt.atlassian.net/secure/projectavatar?size=small&s=small&pid=10000&avatarId=10425"
            }
         },
         "customfield_10032":null,
         "fixVersions":[],
         "customfield_10033":null,
         "customfield_10034":null,
         "aggregatetimespent":null,
         "resolution":null,
         "customfield_10035":null,
         "customfield_10036":null,
         "customfield_10027":null,
         "customfield_10028":null,
         "customfield_10029":null,
         "resolutiondate":null,
         "workratio":-1,
         "issuerestriction":{
            "issuerestrictions":{},
            "shouldDisplay":false
         },
         "lastViewed":"2021-12-09T12:42:04.517+0000",
```

```
69          "watches":{
70             "self":"https://testavt.atlassian.net/rest/api/2/ ↩
                  ↪ issue/TP-35/watchers",
71             "watchCount":0,
72             "isWatching":true
73          },
74          "created":"2021-12-09T12:42:03.973+0000",
75          "customfield_10020":null,
76          "customfield_10021":null,
77          "customfield_10022":null,
78          "priority":{
79             "self":"https://testavt.atlassian.net/rest/api/2/ ↩
                  ↪ priority/3",
80             "iconUrl":"https://testavt.atlassian.net/images/ ↩
                  ↪ icons/priorities/medium.svg",
81             "name":"Medium",
82             "id":"3"
83          },
84          "customfield_10023":null,
85          "customfield_10024":null,
86          "customfield_10025":null,
87          "labels":[],
88          "customfield_10026":[],
89          "customfield_10016":null,
90          "customfield_10017":null,
91          "customfield_10018":{
92             "hasEpicLinkFieldDependency":false,
93             "showField":false,
94             "nonEditableReason":{
95                "reason":"PLUGIN_LICENSE_ERROR",
96                "message":"El enlace principal solo está ↩
                     ↪ disponible para los usuarios de Jira Premium."
97             }
98          },
99          "customfield_10019":"0|i0009z:",
100         "aggregatetimeoriginalestimate":null,
101         "timeestimate":null,
102         "versions":[],
103         "issuelinks":[],
104         "assignee":null,
105         "updated":"2021-12-09T12:42:03.973+0000",
106         "status":{
107            "self":"https://testavt.atlassian.net/rest/api/2/ ↩
                  ↪ status/10000",
108            "description":"",
109            "iconUrl":"https://testavt.atlassian.net/",
110            "name":"To Do",
111            "id":"10000",
112            "statusCategory":{
113               "self":"https://testavt.atlassian.net/rest/api ↩
                     ↪ /2/statuscategory/2",
114               "id":2,
```

```
                "key":"new",
                "colorName":"blue-gray",
                "name":"New"
            }
        },
        "components":[],
        "timeoriginalestimate":null,
        "description":"dfasfsdfsadf",
        "customfield_10010":null,
        "customfield_10014":null,
        "timetracking":{},
        "customfield_10015":null,
        "customfield_10005":null,
        "customfield_10006":null,
        "customfield_10007":null,
        "security":null,
        "customfield_10008":null,
        "attachment":[],
        "customfield_10009":null,
        "aggregatetimeestimate":null,
        "summary":"dfsadfsdfsdfsadf",
        "creator":{
            "self":"https://testavt.atlassian.net/rest/api/2 ↩
                ↪ /user?accountId=6138a222f6070d006b6c8cb5",
            "accountId":"6138a222f6070d006b6c8cb5",
            "avatarUrls":{
                "48x48":"https://secure.gravatar.com/avatar/ ↩
                    ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A ↩
                    ↪ %2F%2Favatar-management--avatars.us-west-2. ↩
                    ↪ prod.public.atl-paas.net%2Finitials ↩
                    ↪ %2FRB-1.png",
                "24x24":"https://secure.gravatar.com/avatar/ ↩
                    ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A ↩
                    ↪ %2F%2Favatar-management--avatars.us-west-2. ↩
                    ↪ prod.public.atl-paas.net%2Finitials ↩
                    ↪ %2FRB-1.png",
                "16x16":"https://secure.gravatar.com/avatar/ ↩
                    ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A ↩
                    ↪ %2F%2Favatar-management--avatars.us-west-2. ↩
                    ↪ prod.public.atl-paas.net%2Finitials ↩
                    ↪ %2FRB-1.png",
                "32x32":"https://secure.gravatar.com/avatar/ ↩
                    ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https%3A ↩
                    ↪ %2F%2Favatar-management--avatars.us-west-2. ↩
                    ↪ prod.public.atl-paas.net%2Finitials ↩
                    ↪ %2FRB-1.png"
            },
            "displayName":"Roque Rojo Bacete",
            "active":true,
            "timeZone":"Etc/GMT",
            "accountType":"atlassian"
        },
```

```
150             "subtasks":[],
151             "customfield_10041":null,
152             "reporter":{
153                "self":"https://testavt.atlassian.net/rest/api/2/
                      ↪ user?accountId=6138a222f6070d006b6c8cb5",
154                "accountId":"6138a222f6070d006b6c8cb5",
155                "avatarUrls":{
156                   "48x48":"https://secure.gravatar.com/avatar/
                         ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https
                         ↪ %3A%2F%2Favatar-management--avatars.us-
                         ↪ west-2.prod.public.atl-paas.net%2Finitials
                         ↪ %2FRB-1.png",
157                   "24x24":"https://secure.gravatar.com/avatar/
                         ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https
                         ↪ %3A%2F%2Favatar-management--avatars.us-
                         ↪ west-2.prod.public.atl-paas.net%2Finitials
                         ↪ %2FRB-1.png",
158                   "16x16":"https://secure.gravatar.com/avatar/
                         ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https
                         ↪ %3A%2F%2Favatar-management--avatars.us-
                         ↪ west-2.prod.public.atl-paas.net%2Finitials
                         ↪ %2FRB-1.png",
159                   "32x32":"https://secure.gravatar.com/avatar/
                         ↪ ac95b8c9a77ecf3fb4cf36bc5e8fbffc?d=https
                         ↪ %3A%2F%2Favatar-management--avatars.us-
                         ↪ west-2.prod.public.atl-paas.net%2Finitials
                         ↪ %2FRB-1.png"
160                },
161                "displayName":"Roque Rojo Bacete",
162                "active":true,
163                "timeZone":"Etc/GMT",
164                "accountType":"atlassian"
165             },
166             "customfield_10043":null,
167             "aggregateprogress":{
168                "progress":0,
169                "total":0
170             },
171             "customfield_10000":"{}",
172             "customfield_10044":null,
173             "customfield_10001":null,
174             "customfield_10045":null,
175             "customfield_10046":null,
176             "customfield_10002":null,
177             "customfield_10047":null,
178             "customfield_10003":null,
179             "customfield_10004":null,
180             "customfield_10038":[
181             ],
182             "customfield_10039":null,
183             "environment":null,
184             "duedate":null,
```

```
            "progress":{
               "progress":0,
               "total":0
            },
            "votes":{
               "self":"https://testavt.atlassian.net/rest/ ↩
                  ↪ api/2/issue/TP-35/votes",
               "votes":0,
               "hasVoted":false
            }
         }
      },
      "changelog":{
         "id":"10070",
         "items":[
            {
               "field":"description",
               "fieldtype":"jira",
               "fieldId":"description",
               "from":null,
               "fromString":null,
               "to":null,
               "toString":"dfasfsdfsadf"
            },
            {
               "field":"priority",
               "fieldtype":"jira",
               "fieldId":"priority",
               "from":null,
               "fromString":null,
               "to":"3",
               "toString":"Medium"
            },
            {
               "field":"reporter",
               "fieldtype":"jira",
               "fieldId":"reporter",
               "from":null,
               "fromString":null,
               "to":"6138a222f6070d006b6c8cb5",
               "toString":"Roque Rojo Bacete",
               "tmpFromAccountId":null,
               "tmpToAccountId":"6138a222f6070d006b6c8cb5"
            },
            {
               "field":"Estado",
               "fieldtype":"jira",
               "fieldId":"status",
               "from":null,
               "fromString":null,
               "to":"10000",
               "toString":"To Do"
```

```
            },
            {
                "field":"summary",
                "fieldtype":"jira",
                "fieldId":"summary",
                "from":null,
                "fromString":null,
                "to":null,
                "toString":"dfsadfsdfsdfsadf"
            }
        ]
    }
}
```

Listado A.5: *JSON Webhook Jira*

```
spring:
  mail:
    default-encoding: UTF-8
    host: smtp.gmail.com
    username: streamconnectt@gmail.com
    password: otroScreto
    port: 587
    properties:
      mail:
        smtp:
          auth: true
          starttls:
            enable: true
    protocol: smtp
    test-connection: false
my:
    configuracionKafka:
        credenciales: ↩
            ↪ "org.apache.kafka.common.security.plain.Plain ↩
            ↪ LoginModule required ↩
            ↪ username=\"avtcloud/oracleidentitycloudservice/ ↩
            ↪ roque.rojo@avanttic.com/ocid1.streampool.oc1.eu ↩
            ↪ -frankfurt-1.amaaaaaa5yv2zpiaf5wp36kjvwatg3j5uvr ↩
            ↪ 3grbpaaudtznkqcyfoy7f6npa\" password=\"secreto-e.e";
        bootstrapServer: ↩
            ↪ "cell-1.streaming.eu-frankfurt-1.oci.oraclecloud ↩
            ↪ .com:9092"
        protocol: "SASL_SSL"
        mechanism: "PLAIN"
        concurrency: 3
    jira:
        empresa: "customfield_10044"
        area: "customfield_10039"
```

Listado A.6: *Application.yml*

```
public String enviarMail(ConsumerRecord<String, String> message, String[] destinatarios) throws ParseException {
  JSONObject texto = (JSONObject) parser.parse(message.value());
  Context context = new Context();
  String process;
  String prioridad = texto.get("prioridad").toString();
  context.setVariable("origen", texto.get("empresa").toString() + " " + texto.get("area").toString());
  context.setVariable("url", texto.get("url"));
  context.setVariable("cuerpo", texto.get("description").toString());
  String attachments = "";
  if (!getAttachments((JSONArray) texto.get("attachments")).isEmpty()) {
    attachments = "Tiene los siguientes archivos adjuntos:\n"
        + getAttachments((JSONArray) texto.get("attachments"));
  }
  context.setVariable("attachment", getAttachments((JSONArray) texto.get("attachments")));

  MimeMessage msg = javaMailSender.createMimeMessage();

  try {
    MimeMessageHelper helper = new MimeMessageHelper(msg, true);
    helper.setTo(destinatarios);
    helper.setFrom("streamconnectt@gmail.com");
    if (prioridad.equals("High") || prioridad.equals("Highest")) {
      process = templateEngine.process("urgente", context);
      helper.setSubject(
          "[Urgente - " + texto.get("empresa").toString() + "] " + texto.get("summary").toString());
    } else {
      process = templateEngine.process("estandar", context);
      helper.setSubject(
          "[StreamConnect - " + texto.get("empresa").toString() + "] " + texto.get("summary").toString());
    }
    helper.setText(process, true);

    javaMailSender.send(msg);

  } catch (Exception e) {
    return "No se pudo enviar el correo" + e.getMessage();
  }
  return "Correo Enviado";
}
```

Listado A.7: Método *enviarMail*

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
```

```
  name: stream-connect-ingress
  rules:
    http:
      paths:
      - backend:
          serviceName: stream-connect-service
          servicePort: 80
```

Listado A.8: *Ingress* Iteración 1

A.2. ITERACIÓN 2

```
npm install mysql
npm install express
npm install ejs
npm install dotenv
npm install axios
npm install body-parser
npm install cowsay
npm install nodemailer
```

Listado A.9: Comandos Instalar Paquetes *StreamSub*

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  namespace: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 1
  template:
    metadata:
      labels:
        app: mysql
    spec:
      securityContext:
        fsGroup: 1000
      containers:
      - name: mysql
        image: mysql:5.6
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: db
          mountPath: /var/lib/mysql
        securityContext:
          privileged: true
        env:
```

```
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-secret
                key: password
      initContainers:
        - name: mount-fix
          image: busybox
          command: ["/bin/chmod"]
          args:
            - "-R"
            - "777"
            - "/var/lib/mysql"
          volumeMounts:
          - name: db
            mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: db
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Listado A.10: *Satefullset MySQL*

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
  namespace: mysql
spec:
  selector:
    app: mysql
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

Listado A.11: *Service MySQL*

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
  namespace: mysql
type: Opaque
stringData:
  password: 'secreto e.e!'
```

Listado A.12: *Secret MySQL*

```
PORT = 3000
HOST = 10.96.49.81
DATABASE = streamconnect
USER = streamconnect
PASSWORD = secreto
LIMIT = 10

HOSTSMTP = smtp.gmail.com
PORTSMTP = 465
SECURE = true
USERSMTP = streamsubb@gmail.com
PASSWORDSMTP = secretoSecretoso
```

Listado A.13: Contenido archivo *.env*

```
var arrayConsultores = []
    return new Promise((resolve, reject) => {
        const sqlGetUsuarios = "SELECT * FROM consultores ORDER BY id_consultor";
        pool.query(sqlGetUsuarios, (error, result) => {
            if (error) throw error;
            result.forEach(row => {
                var consultor = {
                    id_consultor: row.id_consultor,
                    email: row.email,
                    disponibilidad: row.disponibilidad
                };
                arrayConsultores.push(consultor);
            });
            resolve(arrayConsultores);
        });
    });
```

Listado A.14: *getConsultores*

```
if mkdir $HOME/.kube ; then
  echo "Downloading kubeconfig"
fi
oci ce cluster create-kubeconfig --cluster-id ocid1.cluster.oc1.eu-frankfurt-1.aaaaaaaaafrdozjqme4dmojsgftgmztdga2wenzsmmzgiyzxgc4tqnbwhfrt --file $HOME/.kube/config --region eu-frankfurt-1
export KUBECONFIG=$HOME/.kube/config
kubectl delete -f streamSub-service.yaml
kubectl delete -f streamSub-deployment.yaml
sleep 60
kubectl apply -f streamSub-deployment.yaml
kubectl apply -f streamSub-service.yaml
sleep 20
```

Listado A.15: *Unix Shell StreamSub*

```
const mysql = require('mysql');
```

```
require('dotenv').config();
var dbConfig = {
    connectionLimit: process.env.LIMIT,
    host: process.env.HOST,
    database: process.env.DATABASE,
    user: process.env.USER,
    password: process.env.PASSWORD,
}
var pool = mysql.createPool(dbConfig);
module.exports = pool;
```

Listado A.16: *Pool* de Conexiones a la Base de Datos

```
router.get('/consultores', async (req, res) => {
    var consultores;
    try {
        consultores = await DButil.getConsultores();
        res.render('consultores', {
            consultores: consultores});
    } catch (error) {
        console.error('Se ha producido un error a la hora de cargar los consultores: ' + error.code);
        res.render('404', { mensaje: 'No se han podido cargar los consultores' });
    }
});
```

Listado A.17: *Endpoint getConsultores*

```
// Constantes document
const formAddUser = document.forms['formAddUser'];
const btnsVerSuscripciones = document.getElementsByName('btnVerSuscripciones');
const deleteConfirm = document.querySelector('#btnDelete');
const btnsEliminar = document.getElementsByName('btnEliminar');
const addUserConfirm = document.getElementById('addUserConfirm');
const modalConfirmacionConsultor = new bootstrap.Modal(document.getElementById('confirmacionConsultor'), {});
const bodyConfirmacionConsultor = document.getElementById('bodyConfirmacionConsultor');
const modalConsultorInvalido = new bootstrap.Modal(document.getElementById('consultorInvalido'), {});
const bodyConsultorInvalido = document.getElementById('bodyConsultorInvalido');
const modalAddConsultor = new bootstrap.Modal(document.getElementById('addConsultor'), {});
const checksDisponibilidad = document.getElementsByName('checkDisponibilidad');
formAddUser.addEventListener('submit', async function (e) {
    addConsultor(e);
});
```

```
16 checksDisponibilidad.forEach(check => ↩
       ↪ (check.addEventListener('change', async () => {
17     if(check.dataset.disponibilidad=='true'){
18         check.dataset.disponibilidad='false';
19         await actualizarDisponibilidad(check.dataset. ↩
               ↪ id_consultor,"false");
20     }else{
21         check.dataset.disponibilidad='true';
22         await actualizarDisponibilidad(check.dataset. ↩
               ↪ id_consultor,"true");
23     }
24 })));
25 btnsEliminar.forEach(btn => (btn.addEventListener('click', ↩
       ↪ async () => {
26     var id_consultor = btn.dataset.id_consultor;
27     deleteConfirm.addEventListener('click', () => {
28         borrarConsultor(id_consultor);
29     });
30 })));
31 btnsVerSuscripciones.forEach(btn => ↩
       ↪ (btn.addEventListener('click', async () => {
32     var email = btn.dataset.email;
33     verSuscripciones(email);
34 })));
35 async function verSuscripciones(email) {
36     try {
37         const data = await fetch('/streamSub/suscripciones?' + ↩
               ↪ new URLSearchParams({
38             buscarEmail: email,
39         }));
40         if (data.status == 200) {
41             window.location.href = data.url
42         } else {
43             throw error;
44         }
45     } catch (error) {
46         console.error('Fue mal el Fetch');
47     }
48 }
49 function reload() {
50     window.location.reload();
51 }
52 async function borrarConsultor(id, topic) {
53     try {
54         const data = await fetch(`/streamSub/delete/${id}`, { ↩
               ↪ method: 'DELETE' });
55         const res = await data.json();
56         if (res.estado == true) {
57             console.log("Borrado Con exito");
58             reload();
59         } else { throw error; }
60     } catch (error) {
```

```
61          console.error("Fue mal el fetch");
62      }
63  }
64  async function addConsultor(e) {
65      e.preventDefault();
66      const data = await fetch('/streamSub/addConsultor', {
67          method: 'POST',
68          body: new URLSearchParams(new FormData(e.target))
69      });
70      var res = await data.json();
71      if (await res.estado == true) {
72          modalAddConsultor.hide();
73          bodyConfirmacionConsultor.textContent = 'El consultor ' ↩
                ↪ + res.email + ' se ha añadido con éxito con el ↩
                ↪ ID: ' + res.id;
74          modalConfirmacionConsultor.show();
75      } else {
76          modalAddConsultor.hide();
77          bodyConsultorInvalido.textContent = 'El consultor ' + ↩
                ↪ res.email + ' ya existe';
78          modalConsultorInvalido.show();
79      }
80  }
81  async function ↩
        ↪ actualizarDisponibilidad(id_consultor,disponibilidad) {
82      const updateData = new URLSearchParams();
83      updateData.append('id_consultor',id_consultor);
84      updateData.append('disponibilidad',disponibilidad);
85      const data = await fetch('/streamSub/updateDisponibilidad', {
86          method: 'POST',
87          body: updateData
88      });
89      await data.json();
```

Listado A.18: *JavaScript* Consultores

```
1   <%- include('template/cabecera') %>
2     <!-- Modal Suscripcion exitosa-->
3     <div class="modal fade" id="confirmacionSuscripcion" ↩
          ↪ tabindex="-1" aria-labelledby="confirmarSuscripcion"
4       aria-hidden="true">
5       <div class="modal-dialog modal-dialog-centered">
6         <div class="modal-content">
7           <div class="modal-header">
8             <h5 class="modal-title" ↩
                  ↪ id="confirmarSuscripcion">Suscripcion Exitosa</h5>
9             <button type="button" class="btn-close" ↩
                  ↪ data-bs-dismiss="modal" ↩
                  ↪ aria-label="Close"></button>
10          </div>
11          <div class="modal-body" id="bodyConfirmarSuscripcion">
12
13          </div>
```

```
14          <div class="modal-footer modal-footer-centered">
15            <button type="button" class="btn btn-success" ↩
                  ↪ data-bs-dismiss="modal">Aceptar</button>
16          </div>
17        </div>
18      </div>
19    </div>
20    <!-- Modal Suscripción Invalida-->
21    <div class="modal fade" id="suscripcionInvalida" ↩
        ↪ tabindex="-1" aria-labelledby="suscripcionInvalida"
22      aria-hidden="true">
23      <div class="modal-dialog modal-dialog-centered">
24        <div class="modal-content">
25          <div class="modal-header">
26            <h5 class="modal-title" ↩
                  ↪ id="suscripcionInvalida">Suscripcion Invalida</h5>
27            <button type="button" class="btn-close" ↩
                  ↪ data-bs-dismiss="modal" ↩
                  ↪ aria-label="Close"></button>
28          </div>
29          <div class="modal-body" id="bodySuscripcionInvalida">
30            Suscripcion Invalida
31          </div>
32          <div class="modal-footer modal-footer-centered">
33            <button type="button" class="btn btn-danger" ↩
                  ↪ data-bs-dismiss="modal">Aceptar</button>
34          </div>
35        </div>
36      </div>
37    </div>
38    <!-- Modal pregunta addConsultor-->
39    <div class="modal fade" id="addConsultor" tabindex="-1" ↩
        ↪ aria-labelledby="addConsultorLabel" aria-hidden="true">
40      <div class="modal-dialog">
41        <div class="modal-content">
42          <div class="modal-header">
43            <h5 class="modal-title" id="addConsultorLabel">¿Desea ↩
                  ↪ Añadir Consultor?</h5>
44            <button type="button" class="btn-close" ↩
                  ↪ data-bs-dismiss="modal" ↩
                  ↪ aria-label="Close"></button>
45          </div>
46          <div class="modal-body" id="bodyAddConsultor">
47
48          </div>
49          <div class="modal-footer">
50            <button type="button" class="btn btn-secondary" ↩
                  ↪ data-bs-dismiss="modal">Cancelar</button>
51            <button id="btnCrearConsultor" type="button" ↩
                  ↪ class="btn btn-warning" ↩
                  ↪ data-bs-dismiss="modal">Crear</button>
52          </div>
```

```
53        </div>
54      </div>
55    </div>
56    <section style="padding-top: 50px; padding-bottom: 5px;">
57      <div class="container-fluid h-custom">
58        <div class="row d-flex justify-content-center ↩
              ↪ align-items-center h-100">
59          <div class="col-md-8 col-lg-6 col-xl-4 offset-xl-1">
60            <form id="formSuscripcion" class="needs-validation ↩
                ↪ text-center">
61              <div class="divider d-flex align-items-center ↩
                  ↪ my-4"></div>
62              <div>
63                <h5 class=" form-outline mb-4 ↩
                    ↪ card-title">Suscribirse</h5>
64              </div>
65              <div class="form-outline mb-4">
66                <input type="email" id="email" name="email" ↩
                    ↪ class="form-control form-control-lg"
67                  placeholder="Introduce un email válido" ↩
                      ↪ aria-describedby="emailHelp" required />
68              </div>
69              <div class="form-outline mb-3">
70                <select class="form-select form-select-lg mb-3" ↩
                    ↪ id="selectTopic" name="topic" required>
71                  <option hidden selected="selected" ↩
                      ↪ value="">Elija una Empresa</option>
72                </select>
73                <select class="form-select form-select-lg mb-3" ↩
                    ↪ id="selectPartition" name="particion" ↩
                    ↪ required disabled>
74                  <option hidden selected="selected" ↩
                      ↪ value="">Elija un Area</option>
75                </select>
76              </div>
77              <div class="text-center  mt-4 pt-2">
78                <button type="submit" class="btn btn-primary btn-lg"
79                  style="padding-left: 2.5rem; padding-right: ↩
                      ↪ 2.5rem;">Suscribirse</button>
80              </div>
81            </form>
82          </div>
83        </div>
84      </div>
85    </section>
86    <script type="text/javascript" src="js/suscribirse.js"></script>
87    <%- include('template/footer') %>
```

Listado A.19: Contenido del archivo *suscribirse.ejs*

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
```

```
  name: stream-sub-deployment
spec:
  selector:
    matchLabels:
      app: stream-sub
  replicas: 1
  template:
    metadata:
      labels:
        app: stream-sub
    spec:
      containers:
      - name: stream-sub
        image: fra.ocir.io/fr0k8vgjqhib/streamsub:latest
        ports:
        - containerPort: 3000
      imagePullSecrets:
      - name: regcred
```

Listado A.20: *Deployment StreamSub*

```
apiVersion: v1
kind: Service
metadata:
  name: stream-sub-service
spec:
  selector:
    app: stream-sub
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
```

Listado A.21: *Service StreamSub*

```
var nodemailer = require("nodemailer");
var ejs = require("ejs");
var transporter = nodemailer.createTransport({
    host: process.env.HOSTSMTP,
    port: process.env.PORTSMTP,
    secure: process.env.SECURE,
    auth: {
        user: process.env.USERSMTP,
        pass: process.env.PASSWORDSMTP
    }
});

function enviarCorreo(destinatario,suscripciones) {
    ejs.renderFile(__dirname + '/views/template/correo.ejs', ↩
        ↪ {destinatario, suscripciones}, function (err, data) {
        if (err) {
            console.error(err);
        } else {
```

```
            var mainOptions = {
                from: '"StreamSub" streamsubb@gmail.com',
                to: destinatario,
                subject: "Suscripciones Actualizadas",
                html: data
            };
            transporter.sendMail(mainOptions, (err, info) => {
                if (err) {
                    console.error(err)
                } else {
                    console.log("Mensaje enviado con éxito");
                }
            });
        }
    });
}

module.exports={
    enviarCorreo:enviarCorreo
}
```

Listado A.22: *sendEmail.js*

A.3. ITERACIÓN 3

```
public void crearTopic(String topic, int particion) {
  log.info("Creando topic {}... ", topic);
  NewTopic nt = new NewTopic(topic, particion, (short) 1);
  admin.createTopics(Arrays.asList(nt));
}
public void borrarTopic(String topic) {
  log.info("Borrando topic {}... ", topic);
  admin.deleteTopics(Arrays.asList(topic));
}
public JSONObject getTopics() throws InterruptedException, ↩
    ↪ ExecutionException {
  JSONObject topicInfo = new JSONObject();
  ArrayList<String> topicNames = new ↩
      ↪ ArrayList<String>(admin.listTopics().names().get());
  for (String topicName : topicNames) {
    topicInfo.put(topicName, ↩
        ↪ kafkaTemplate.partitionsFor(topicName).size());
  }
  return topicInfo;
}
```

Listado A.23: Métodos *crearTopic*, *borrarTopic* y *getTopics*

```
public String[] getSuscriptores(String topic, int particion) ↩
    ↪ throws URISyntaxException {
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.getMessageConverters().add(new ↩
        ↪ StringHttpMessageConverter());
```

```
4     log.info("Obtiendo suscripciones de la particion {} de {}", ↩
          ↪ particion, topic);
5     String[] result = restTemplate.getForObject(
6     "https://streamconnect.avanttic.com/streamSub/suscriptores/ ↩
          ↪ {topic}/{particion}", String[].class, topic,particion);
7     if (result.length == 0) {
8       log.warn("No hay nadie suscrito a la particion {} del ↩
            ↪ topic {}", particion, topic);
9     }
10    return result;
11  }
12 public int[] getAsignaciones(String topic, String area) throws ↩
      ↪ URISyntaxException {
13   RestTemplate restTemplate = new RestTemplate();
14   restTemplate.getMessageConverters().add(new ↩
         ↪ StringHttpMessageConverter());
15   log.info("Obtiendo particiones que tienen asignada el area: ↩
         ↪ {} del topic {}", area, topic);
16   int[] result = restTemplate.getForObject(
17     "https://streamconnect.avanttic.com/streamSub/asignacion/ ↩
           ↪ {topic}/{area}", int[].class, topic, area);
18   if (result.length == 0) {
19     log.warn("El topic {} no tiene asignada en ninguna ↩
           ↪ particion el area {}", topic, area);
20   } else {
21     for (int i : result) {
22       log.info("Las particion {} tiene asignada el area {} en ↩
             ↪ el topic {}", i, area, topic);
23     }
24   }
25   return result;
26 }
```

Listado A.24: *SuscriptoresController.java*

```
1  formAddTopic.addEventListener('submit',
2      async function (e) {
3          e.preventDefault();
4              if (await checkTopic(inputNombre)) {
5                  const data = await ↩
                       ↪ fetch('https://streamconnect.avanttic.com/ ↩
                       ↪ streamConnect/crearTopic', {
6                      method: 'POST',
7                      body: new URLSearchParams(new ↩
                           ↪ FormData(e.target))
8                  });
9                  modalAddTopic.hide();
10                 modalConfirmarCreacion.show();
11             } else {
12                 modalAddTopic.hide();
13                 modalCreacionInvalida.show();
14             }
15     }
```

```
);
```

Listado A.25: Petición *POST* a */borrarTopic*

```
async function getTopicInfo() {
    const data = await
        fetch('https://streamconnect.avanttic.com/
        streamConnect/topics', {
         method: 'GET',
    }).catch(error => { console.error("Se ha producido un error
        al conseguir los topics") });
    return await data.json();}
```

Listado A.26: Petición *GET* al *endpoint /topics* de *StreamConnect*

```
router.get('/asignacion/:topic/:area', async (req, res) => {
    var topic = req.params.topic;
    var area = req.params.area;
    var id_area = await DButil.buscarIdArea(area);
    var resultado = await
        DButil.buscarParticionAsignada(id_area, topic)
    res.send(resultado.arrayParticiones);
});
router.get('/suscriptores/:topic/:particion', async (req, res)
    => {
    var topic = req.params.topic;
    var particion = req.params.particion;
    var suscriptores = await DButil.getIdSuscriptores(topic,
        particion);
    res.json(suscriptores);
});
```

Listado A.27: *Endpoints* microservicios *StreamSub*

ANEXO B

Figuras Complementarias

En este anexo, se encuentran una serie de figuras complementarias que ayudan a ilustrar las explicaciones del documento principal.

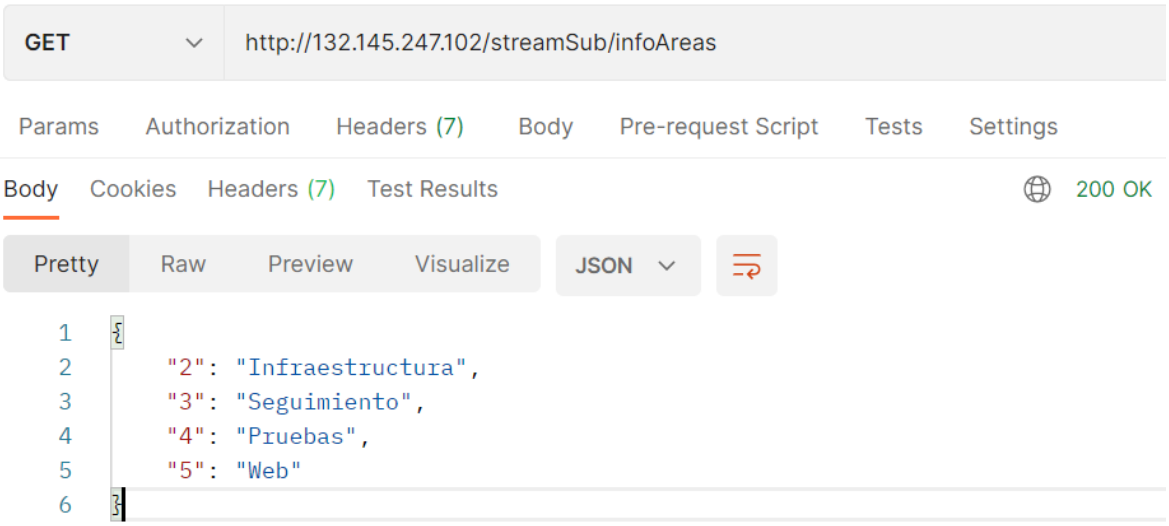
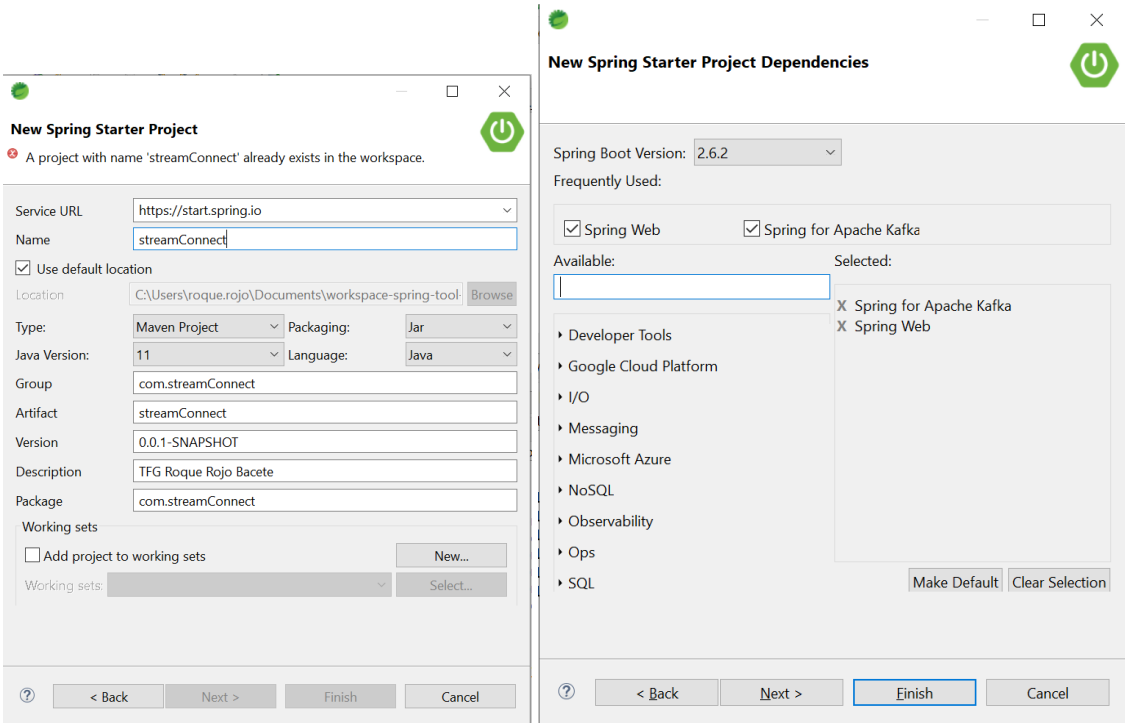


Figura B.1: Ejemplo de petición a *StreamConnect*



(a) Paso 1 **(b)** Paso 2

Figura B.2: *Iniciar Proyecto SpringBoot*

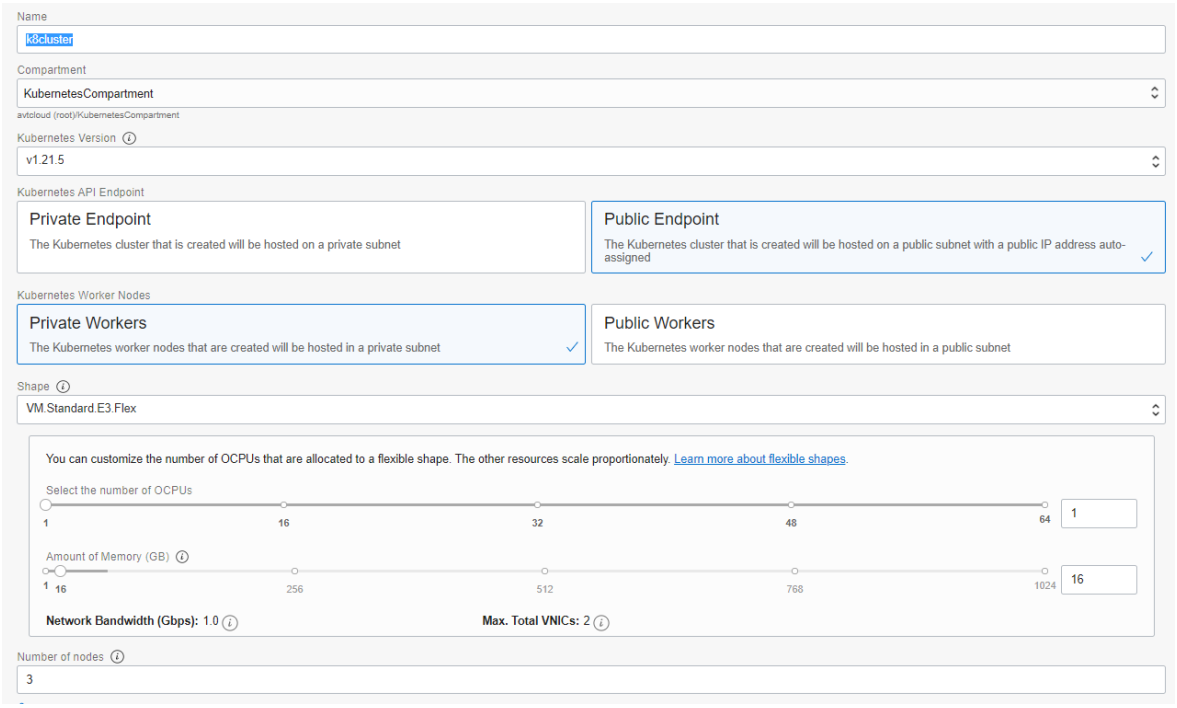


Figura B.3: Formulario para crear un *Cluster* de *Kubernetes*

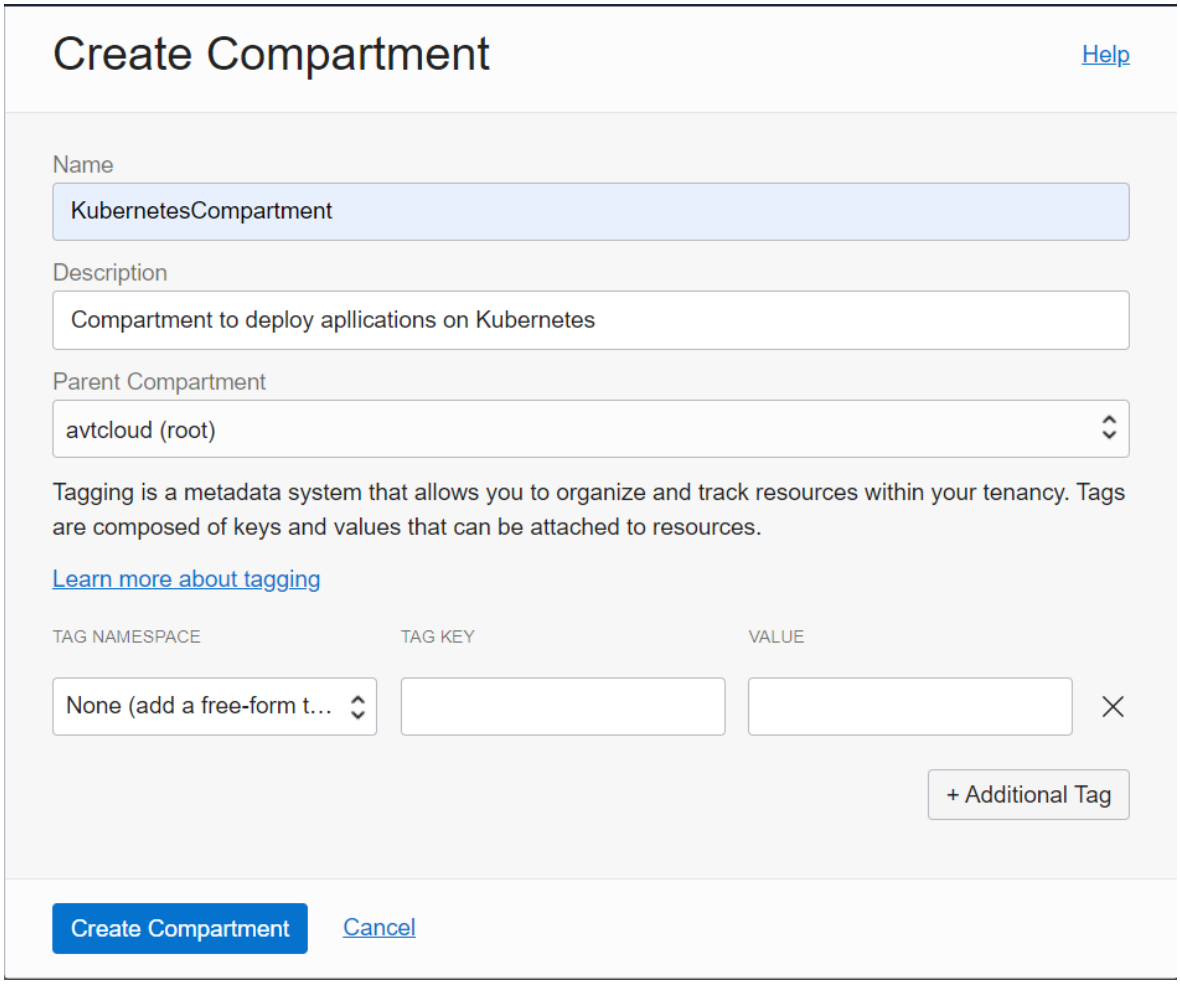


Figura B.4: Formulario para crear un Compartimento en *Oracle Cloud*

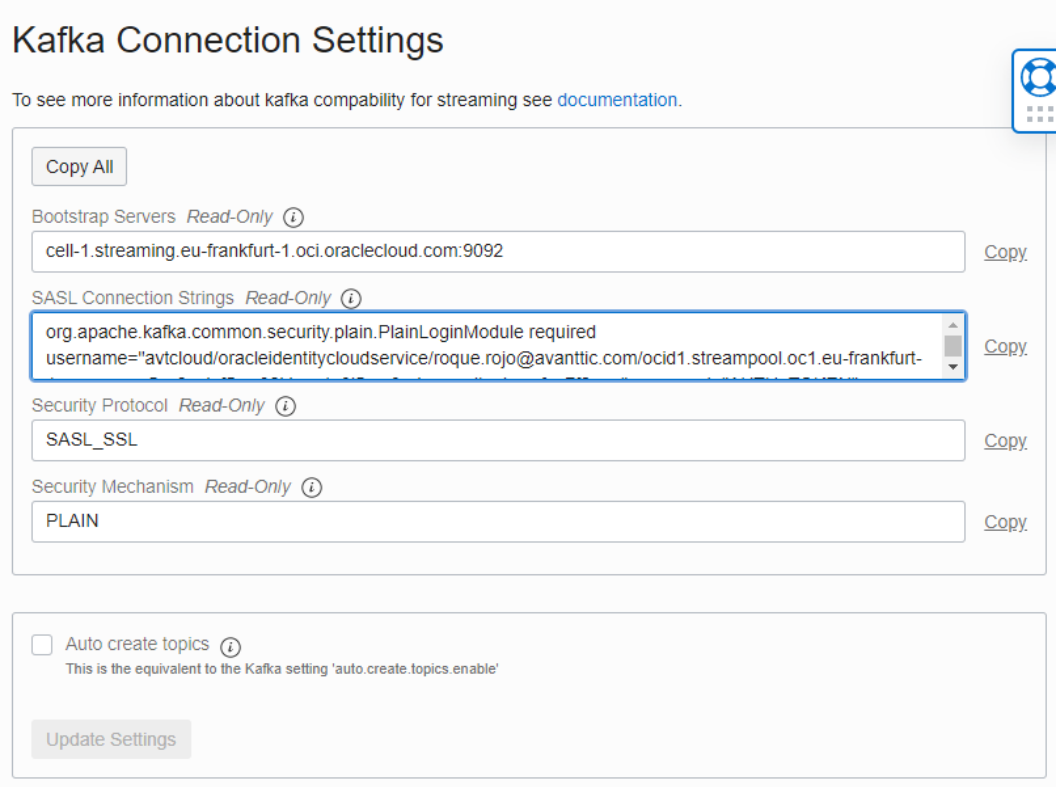


Figura B.5: Configuración necesaria para conectarse a *Streaming* con *Apache Kafka*

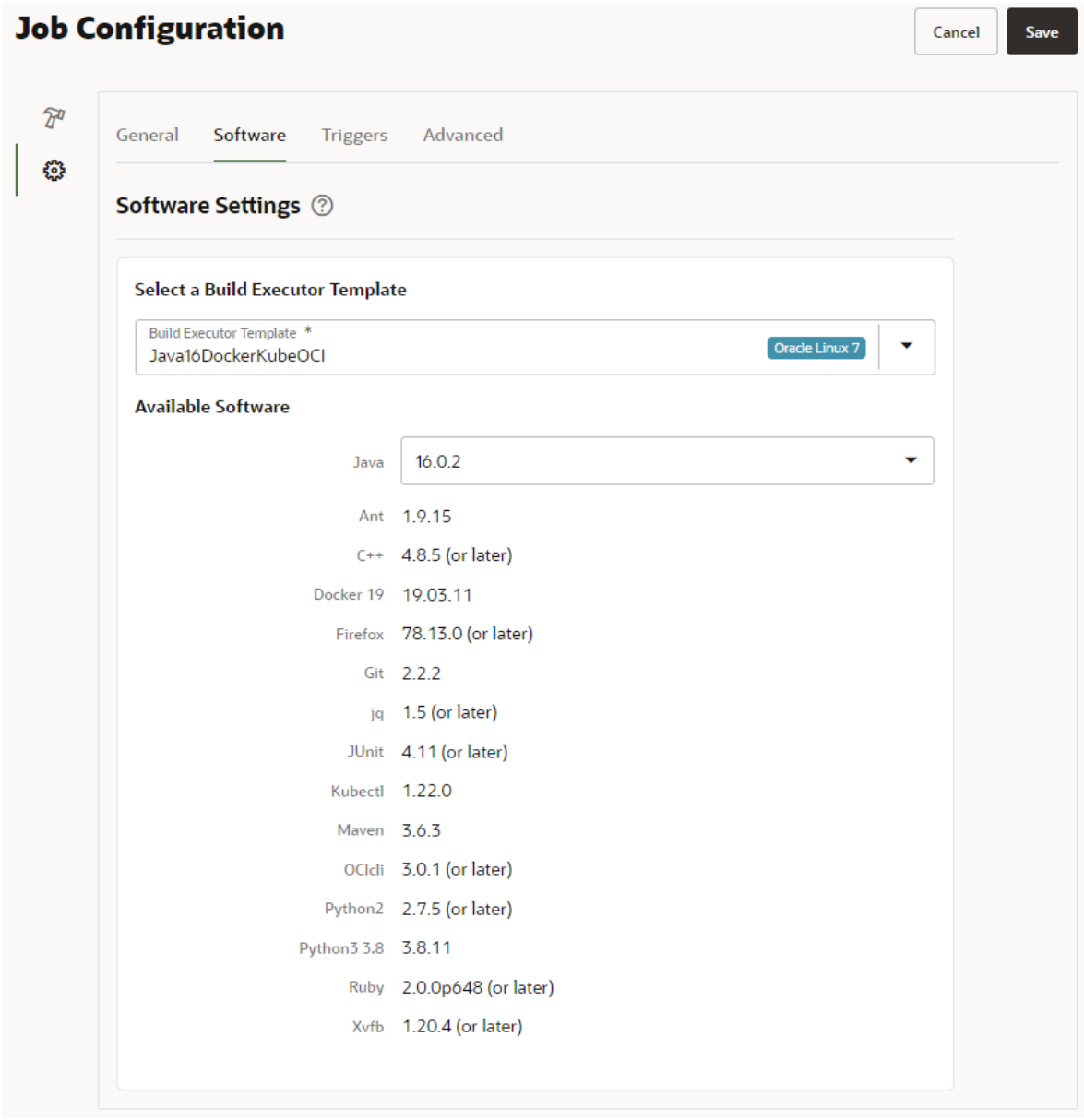
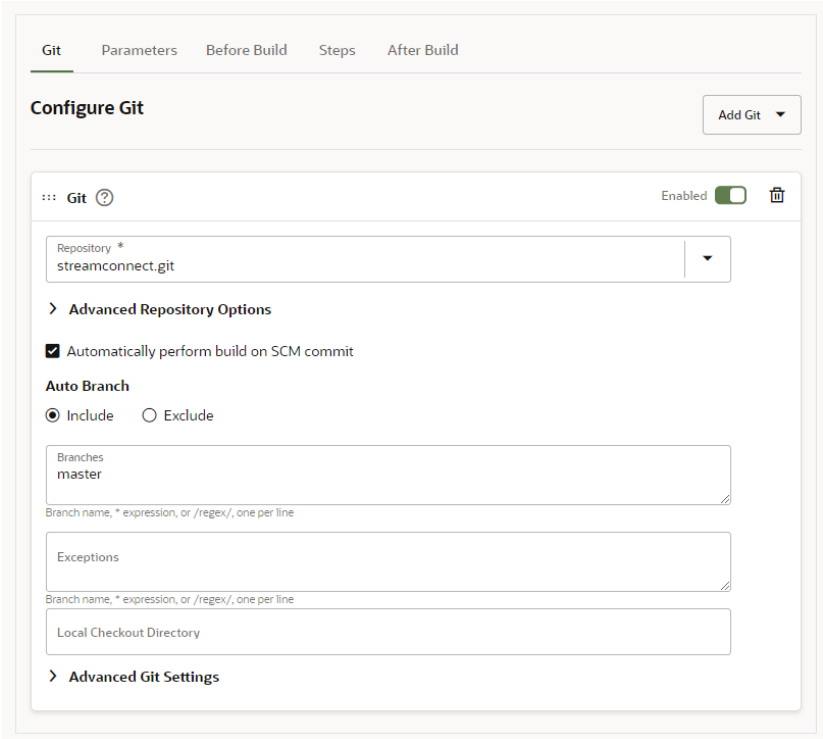
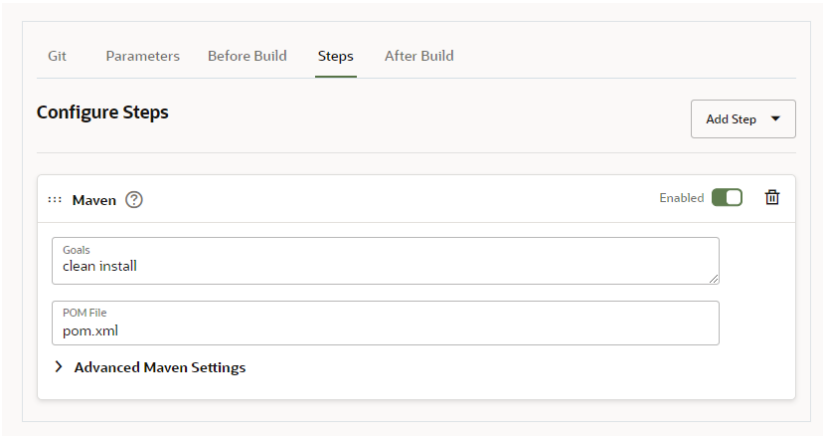


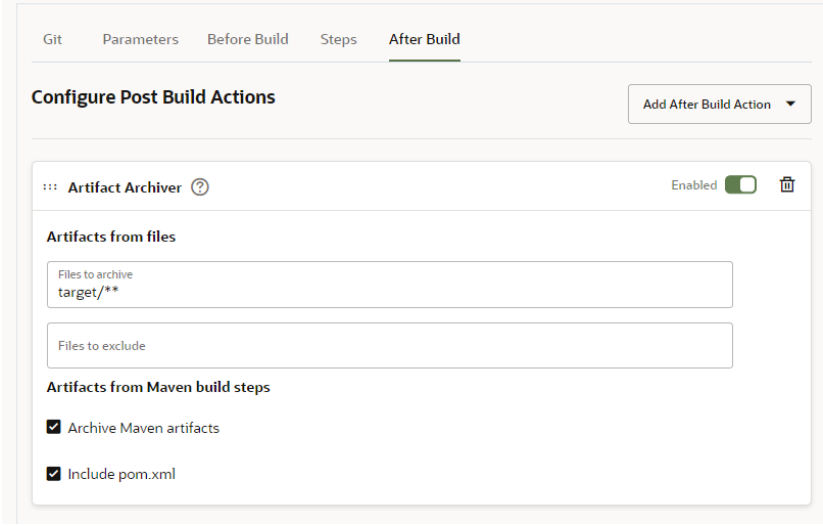
Figura B.6: *Build Executor*



(a) Git



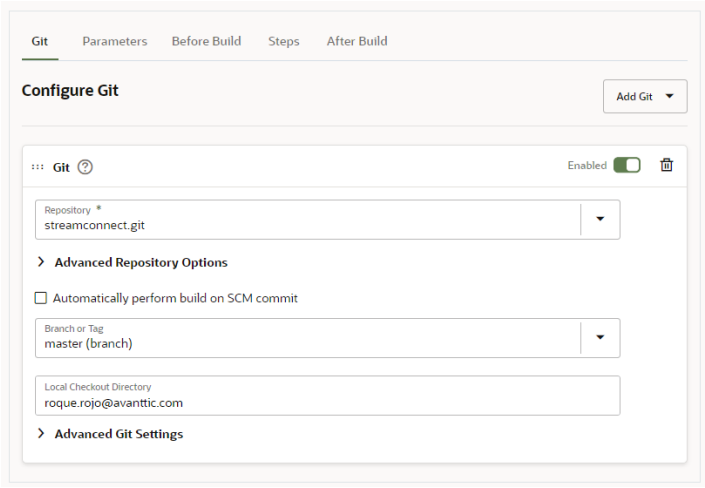
(b) Steps



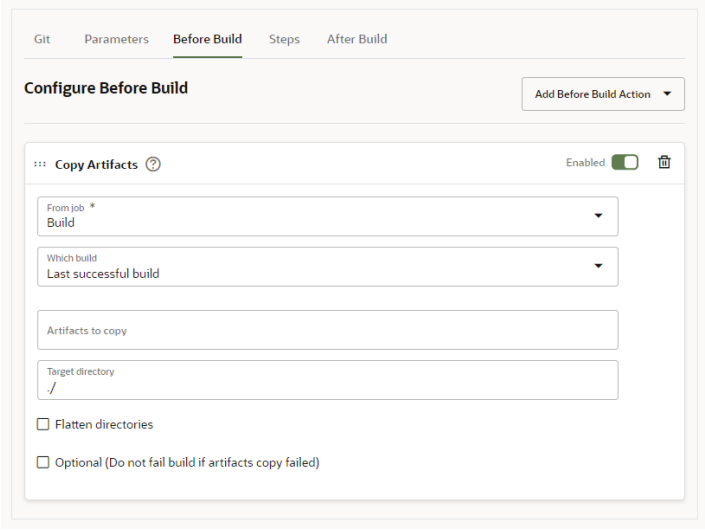
(c) After Build

Figura B.7: *Job: Build*

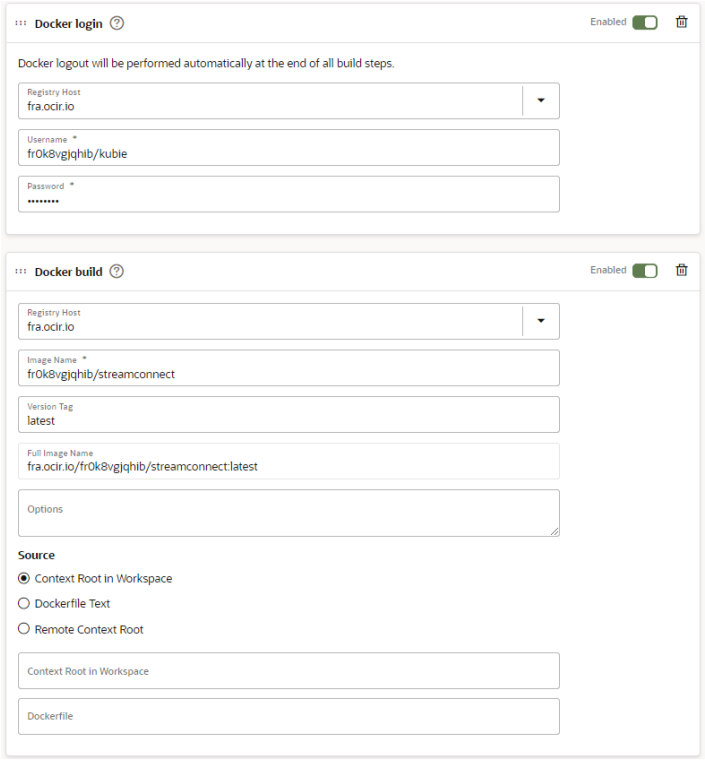
Figura B.8: *Pipeline Build and Deploy*



(a) Git

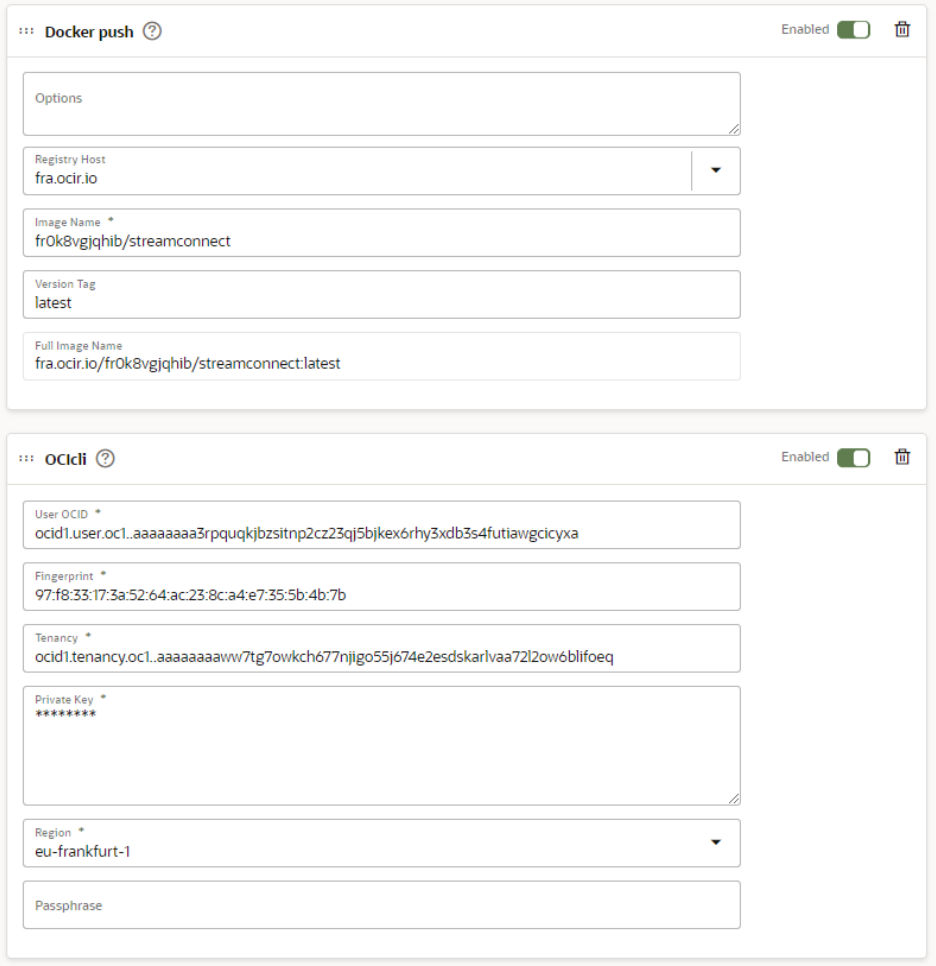


(b) Before Build

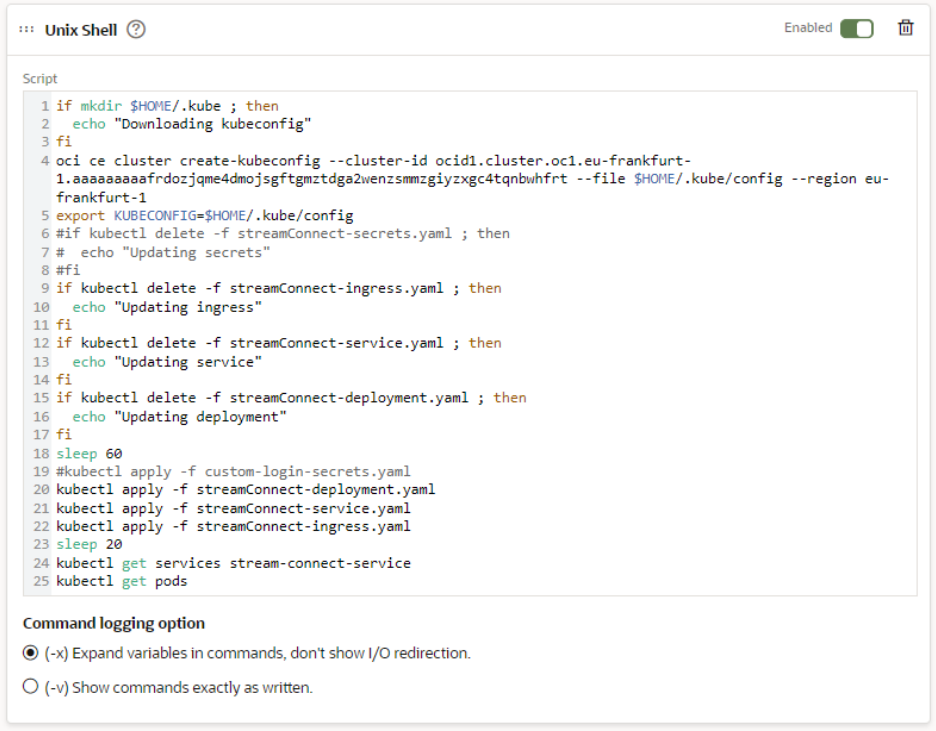


(c) Steps 1

Figura B.9: *Job: Docker and Deploy*

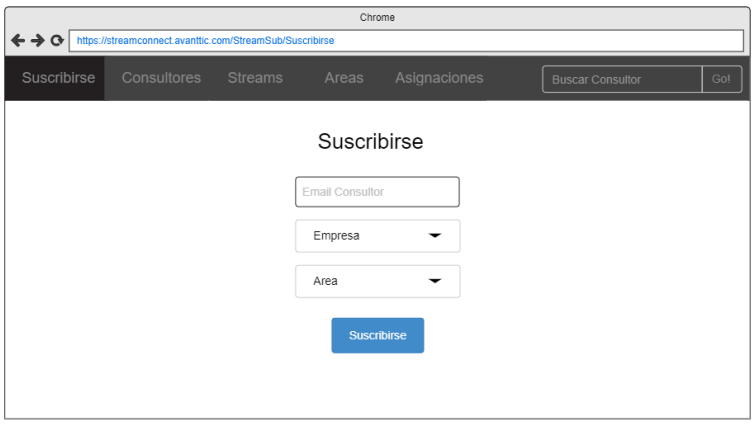


(a) Steps 2

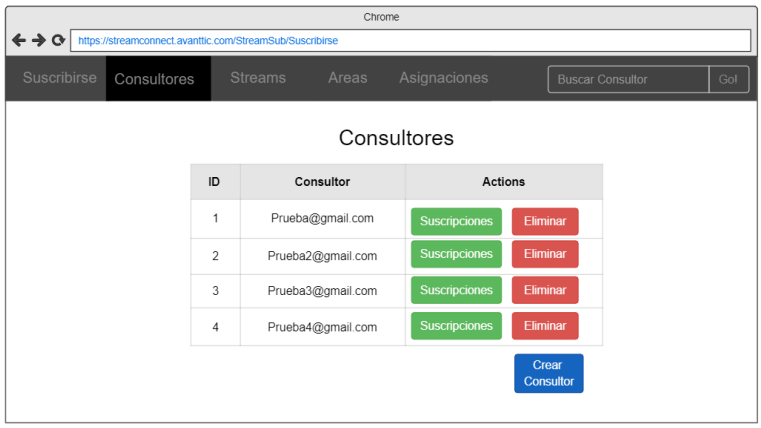


(b) Steps 3

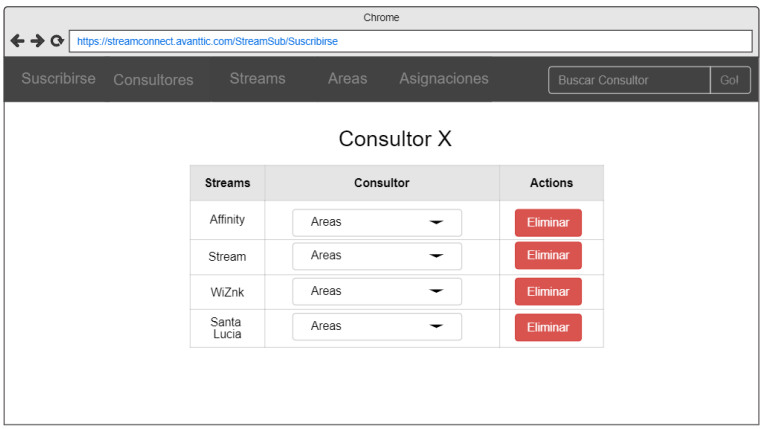
Figura B.10: *Job: Docker and Deploy* (2)



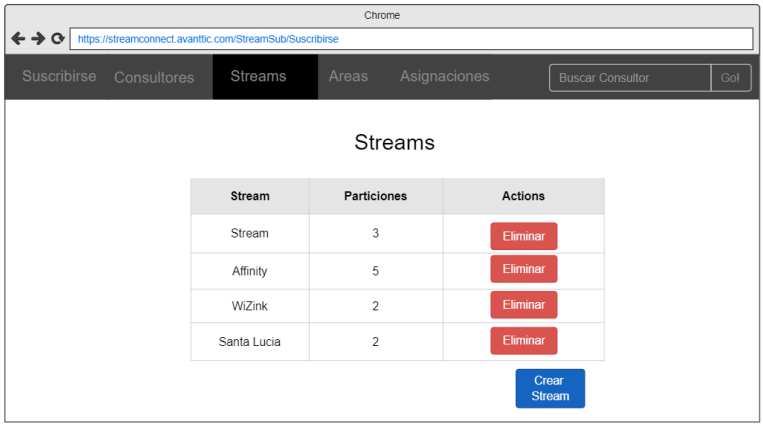
(a) Boceto Suscribirse



(b) Boceto Consultores

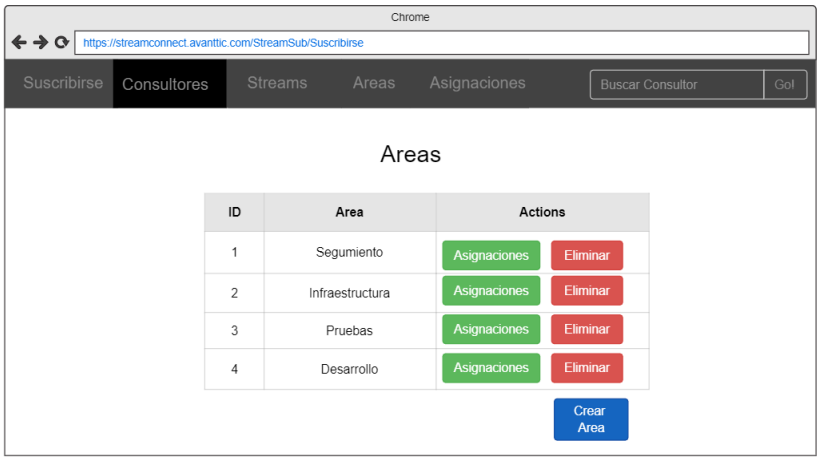


(c) Boceto Suscripciones

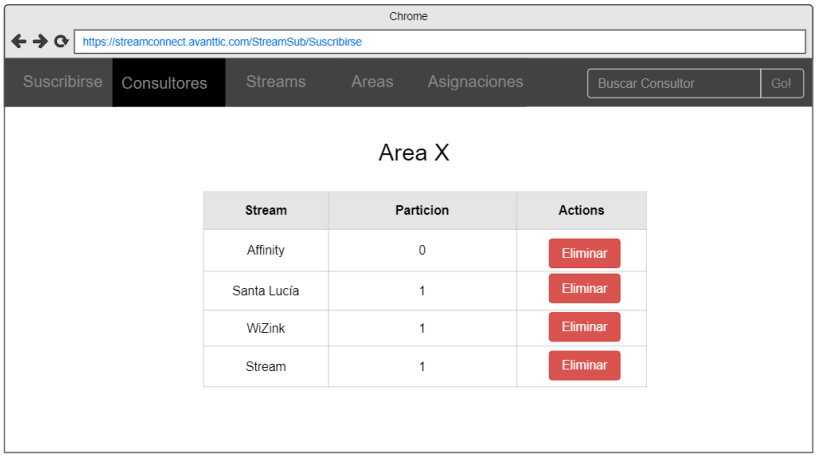


(d) Boceto Stream

Figura B.11: Bocetos 1



(a) Boceto Área



(b) Boceto Asignaciones

(c) Boceto Asignar

Figura B.12: Bocetos 2

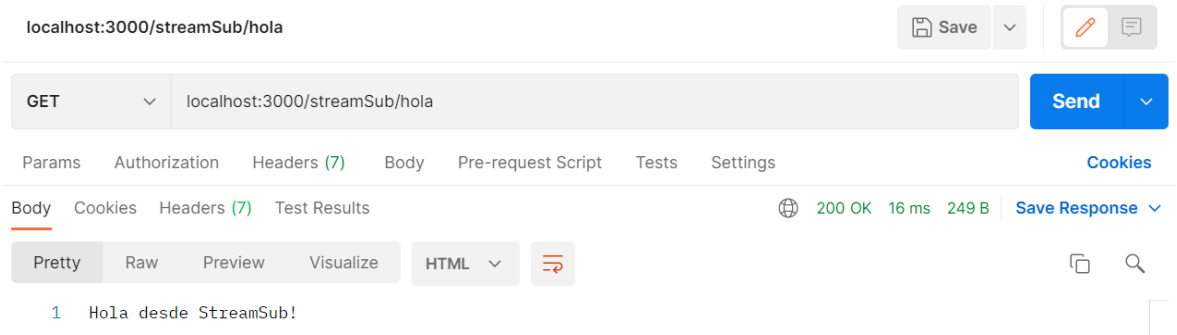
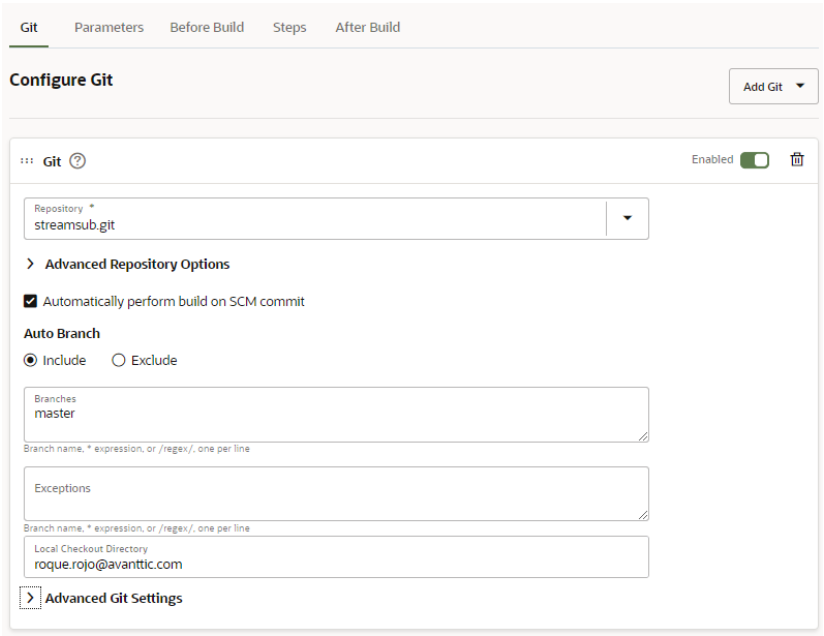
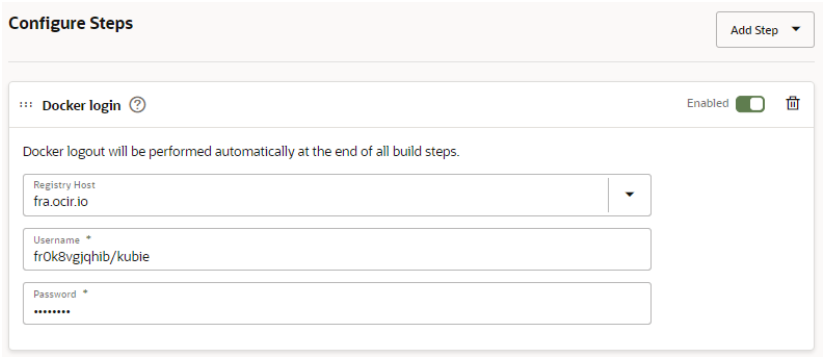


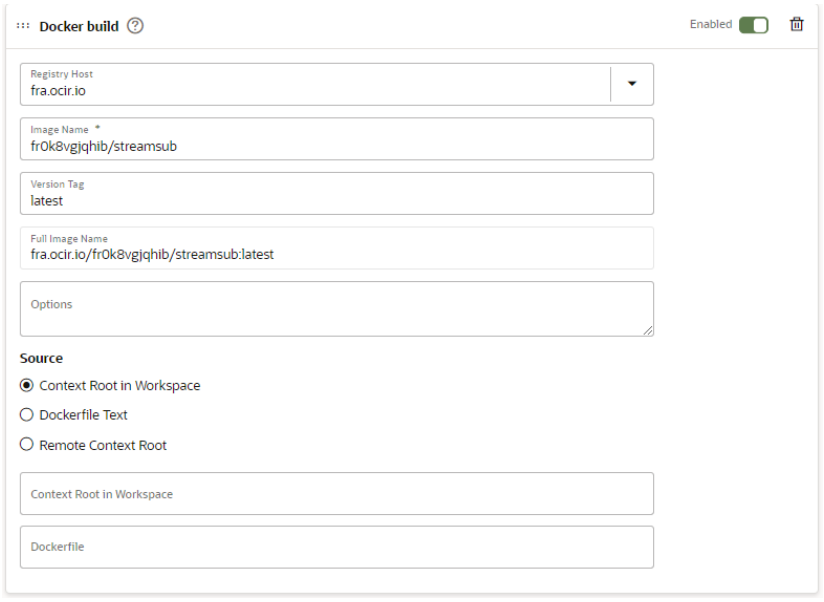
Figura B.13: Hola *StreamSub*



(a) Git

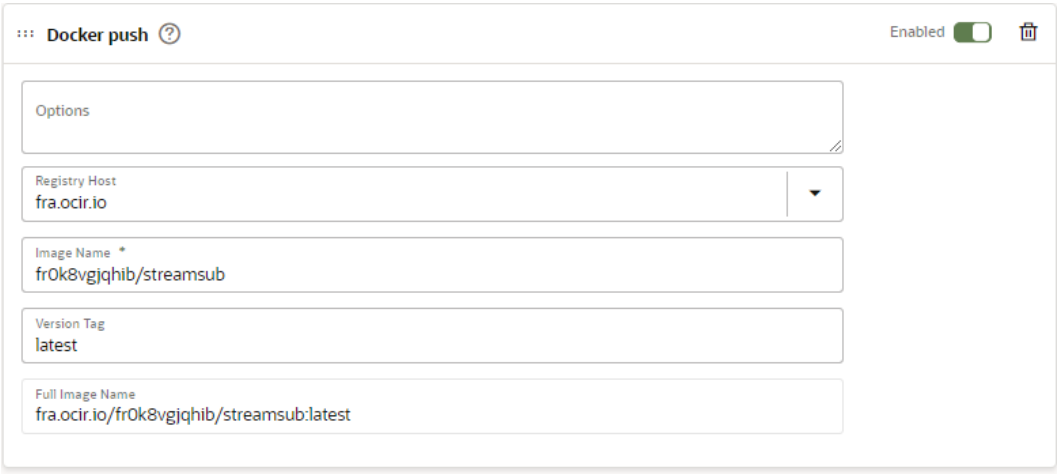


(b) Before Build

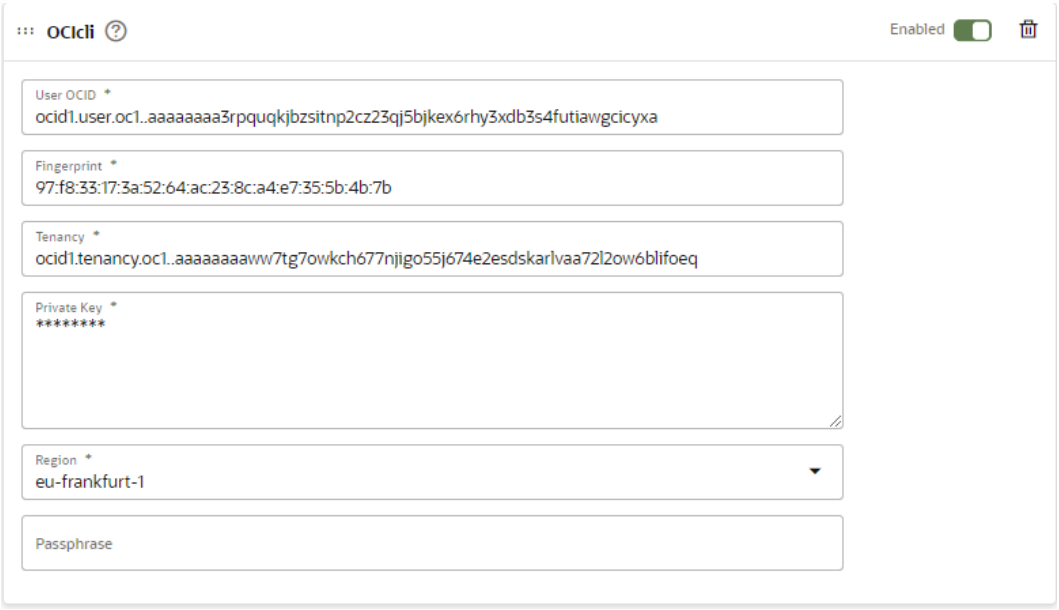


(c) Step 2

Figura B.14: *Job: streamSub_deployToKubernetes*



(a) Step 3



(b) Step 4

(c) Step 5

Figura B.15: *Job: streamSub_deployToKubernetes* (2)

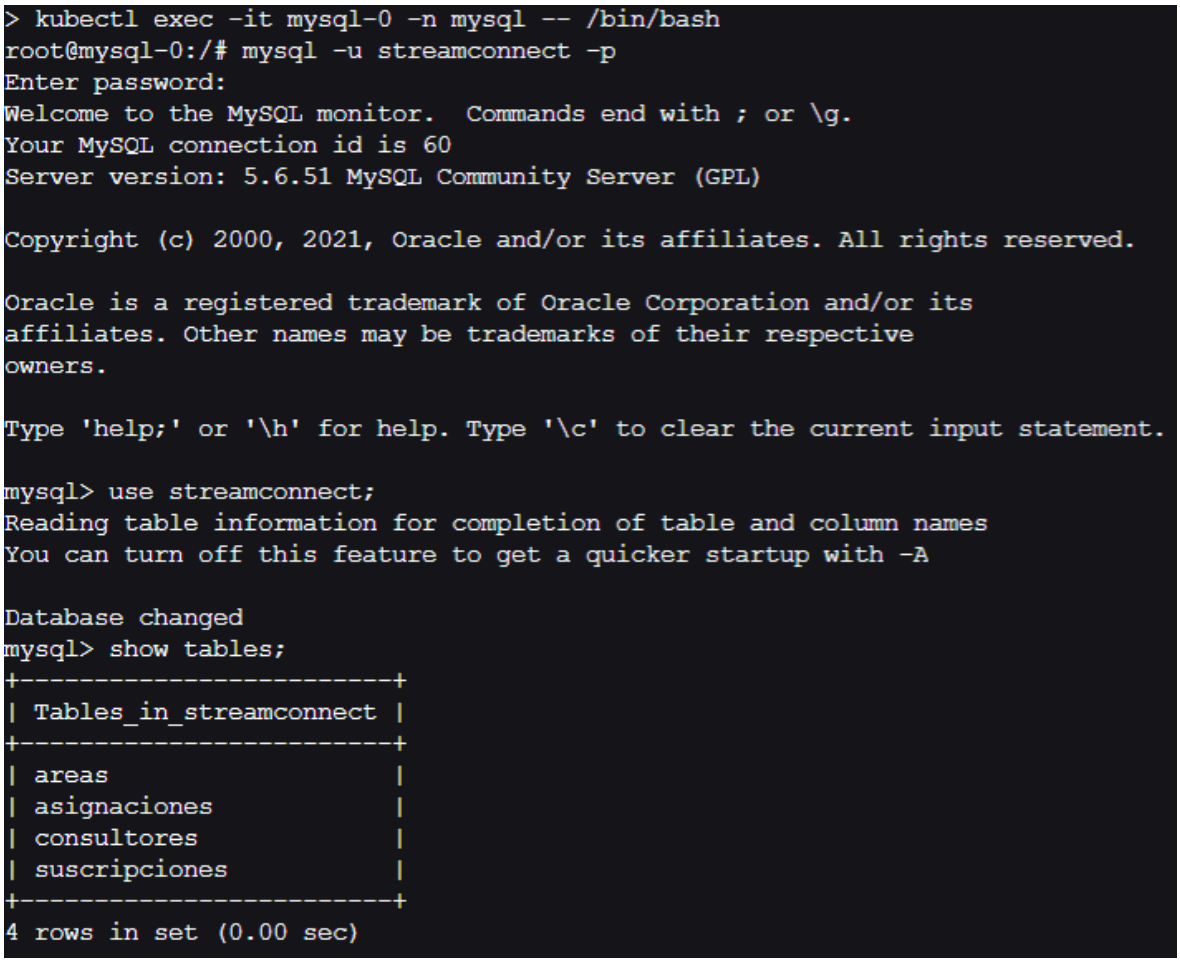


Figura B.16: *MySQL* desplegado en *Kubernetes*

ANEXO C

Visión General del Funcionamiento de la Aplicación

C.1. VISIÓN GENERAL DEL FUNCIONAMIENTO DE LA APLICACIÓN

Este anexo tiene como objetivo ayudar al lector a comprender el funcionamiento de la aplicación de una manera más clara y concisa. Para ello en las siguientes subsecciones mostraremos y explicaremos mediante diagramas de secuencia el funcionamiento de los aspectos más importante de nuestra solución.

C.1.1. Enviar Correo a los Consultores

En esta subsección se va a explicar el flujo de trabajo que se lleva a cabo desde que el cliente crea una petición de servicio hasta que la reciben los consultores suscritos, basándonos para ello en la C.1.

El proceso se inicia cuando un cliente accede a su cuenta *Jira* y crea una incidencia rellenando los campos pertinentes del formulario. Una vez se crea la incidencia el *Webhook* de *Jira* se activa y manda la información de la incidencia al *endpoint issues* habilitado para recibir peticiones *POST* de *StreamConnect*, que responderá con un *STATUS 200 OK*.

Una vez llega la petición *StreamConnect* la procesa, obteniendo una serie de valores importantes, entre los que destacamos el *Stream* de procedencia y el área. Una vez los tiene, realiza una petición *GET* al *endpoint asignacion* de *StreamSub*, utilizando como parámetros de consulta el nombre del *Stream* y área.
Por su parte, *StreamSub* realizará una consulta a la Base de Datos, buscando las particiones a las que tiene asignada ese área en el *Stream*, está consulta devolverá un vector de enteros. *StreamSub* responderá a la petición procedente de *StreamConnect* con un *STATUS 200 OK* y el vector de las particiones.

Una vez que *StreamConnect* conoce las particiones y el *Stream* al que debe enviar el mensaje, manda en formato *JSON* la información relevante de la incidencia a su destino.

Por otra parte, el consumidor de *StreamConnect* que se encuentra suscrito a todas las particiones de los *Streams* que siguen el patrón *AVT-** detectará el mensaje nuevo y lo consumirá.
Cuando esto sucede, *StreamConnect* obtiene los datos necesarios del *Consumer Record* que ha consumido y realiza una petición *GET* al *endpoint suscriptores* de *StreamSub* utilizando como parámetros

de consulta el *Stream* y las partición de la que procede.
Cuando recibe la consulta *StreamSub* hace una consulta a la Base de Datos, obteniendo los email de todos los consultores que se han suscrito a la partición que tiene asignada el área del *Stream* al que ha sido enviada la incidencia, esta consulta retornará un vector con todos los email de los consultores suscritos. Una vez que *StreamSub* tiene los datos que necesita *StreamConnect*, responde con un *STATUS 200 OK* y el vector con los email de los consultores.

Una vez tiene a lista de destinatarios, *StreamConnect* procesa el *Consumer Record*, y en función de si la incidencia es urgente o no, selecciona una plantilla de correo urgente o estándar. Una vez selecciona la plantilla la renderiza, rellenándola con los datos de la incidencia. Se comunica con el *SMTP* de Google y envía a los consultores el correo electrónico.

C.1.2. Crear un *Stream*

En esta subsección se va a explicar el flujo de trabajo que se sigue para crear un nuevo *Stream* a través de *StreamSub* para ello debemos fijarnos en la figura C.2.

El flujo se inicia cuando un consultor quiere crear un *Stream*, para ello debe rellenar los datos del formulario, es decir, el nombre del *Stream* y la cantidad de particiones que debe tener. Una vez esta relleno, solicita la información de los *Streams* a *StreamConnect*.

Para obtener la información de los *Streams*, *StreamConnect* solicita la información a *Streaming Service* mediante el método *getTopics*, el cual puede verse en la figura 5.5. Una vez obtenida, comprueba que el nombre introducido exista ya, si es así, *StreamSub* realizará una petición *POST* contra el *endpoint crearTopic* de *StreamConnect* utilizando como parámetros el *nombre* y las particiones deseadas.

StreamConnect contactará con *Streaming Service* mediante el método *crearTopic*, utilizando como parámetros el nombre y el número de particiones. Una vez echo esto, *Streaming Service* responderá con un *STATUS 200 OK*, que es lo mismo que responderá *StreamConnect* a *StreamSub*. Una vez echo esto, se notificará al consultor que el *Stream* ha sido creado con éxito.

C.1.3. Eliminar un *Stream*

En esta subsección se va a explicar el flujo de trabajo que se sigue a la hora de eliminar *Stream* a través de *StreamSub*. Para ello debemos fijarnos en la figura C.3.

El flujo se inicia cuando un consultor desea borrar un *Stream* existente. Una vez confirme que desea borrarlo, se lanzará una petición *POST* contra el *endpoint borrarTopic* de *StreamConnect*. Este *endpoint* realizará una petición a streamConnect con el método *borrarTopic*, que borrará el *Stream* y responderá con un *STATUS 200 OK*, igual que responderá *StreamConnect* a *StreamSub*.

Una vez tenga la respuesta de *StreamConnect* procederá a realizar dos consultas contra la Base de Datos, las cuales eliminarán todas las suscripciones y asignaciones que tenga ese *Stream*. Después de esto, se le notificará al usuario de que el *Stream* ha sido eliminado con éxito.

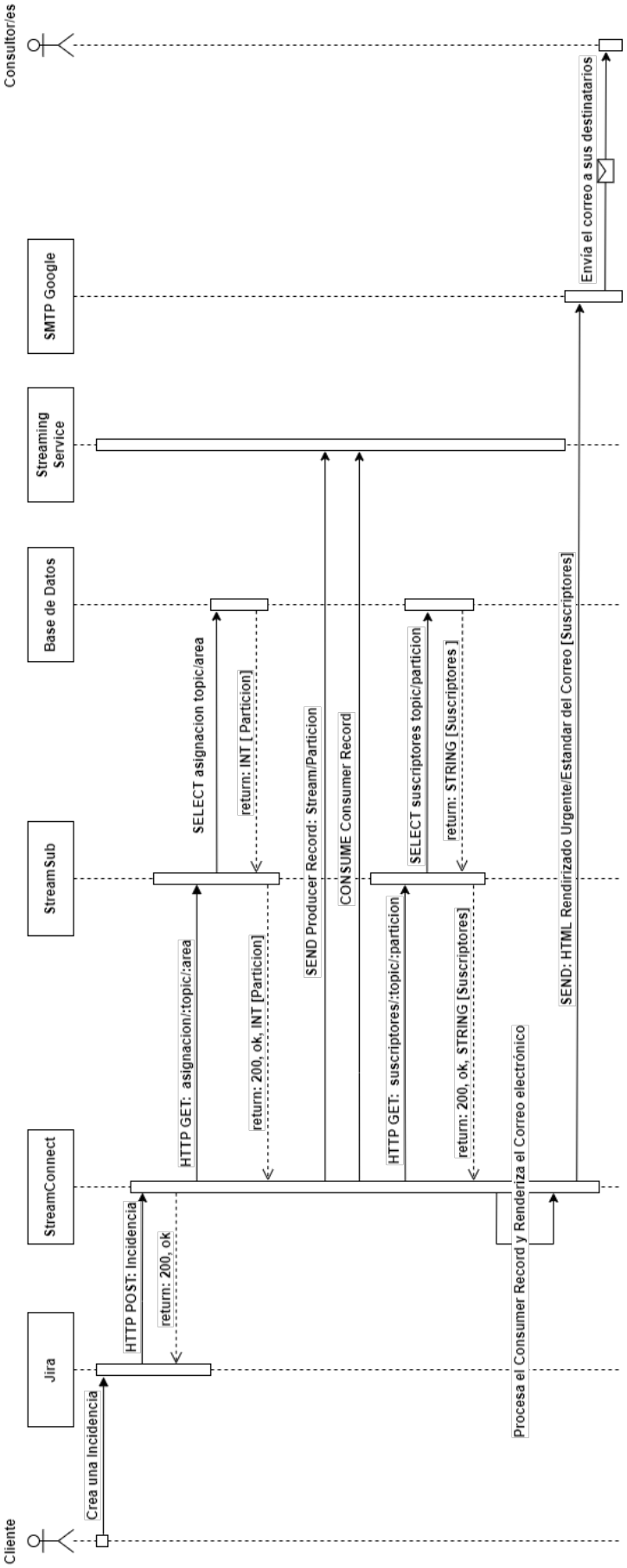


Figura C.1: Diagrama de Secuencia

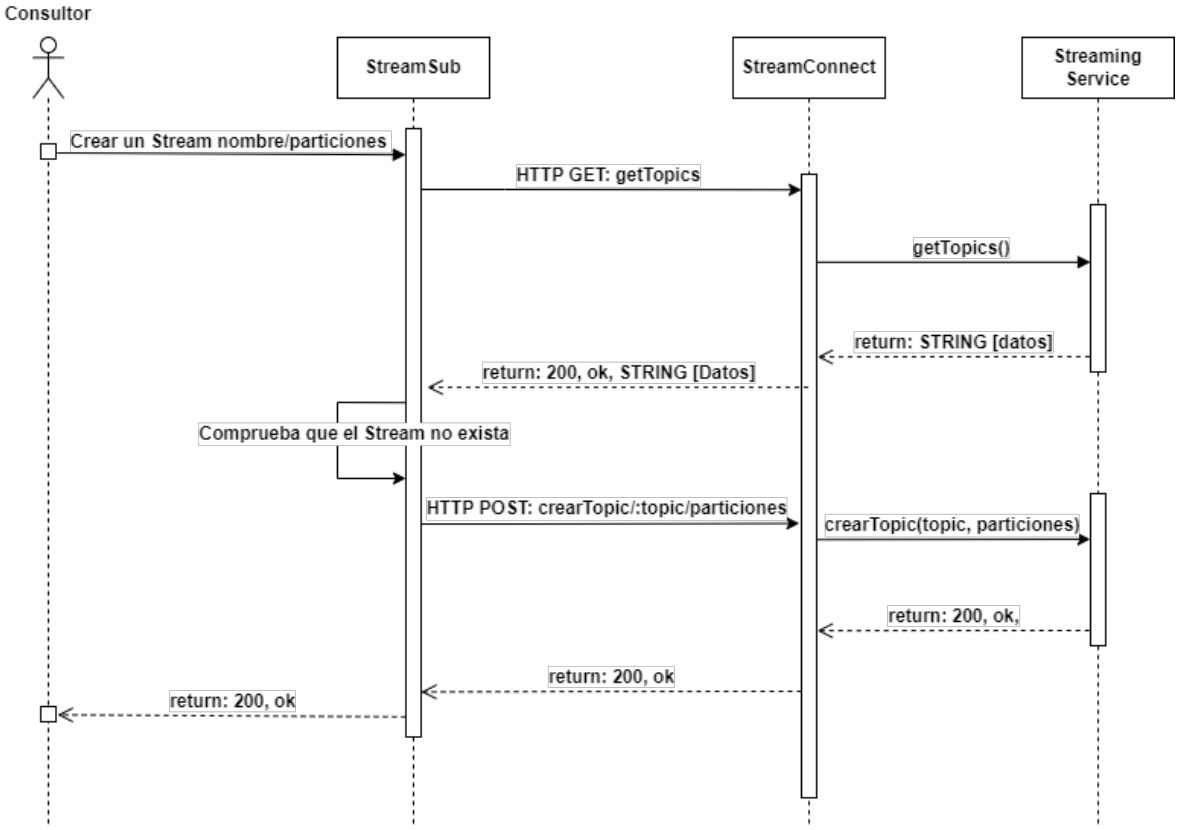


Figura C.2: Diagrama de Secuencia Crear *Topic*

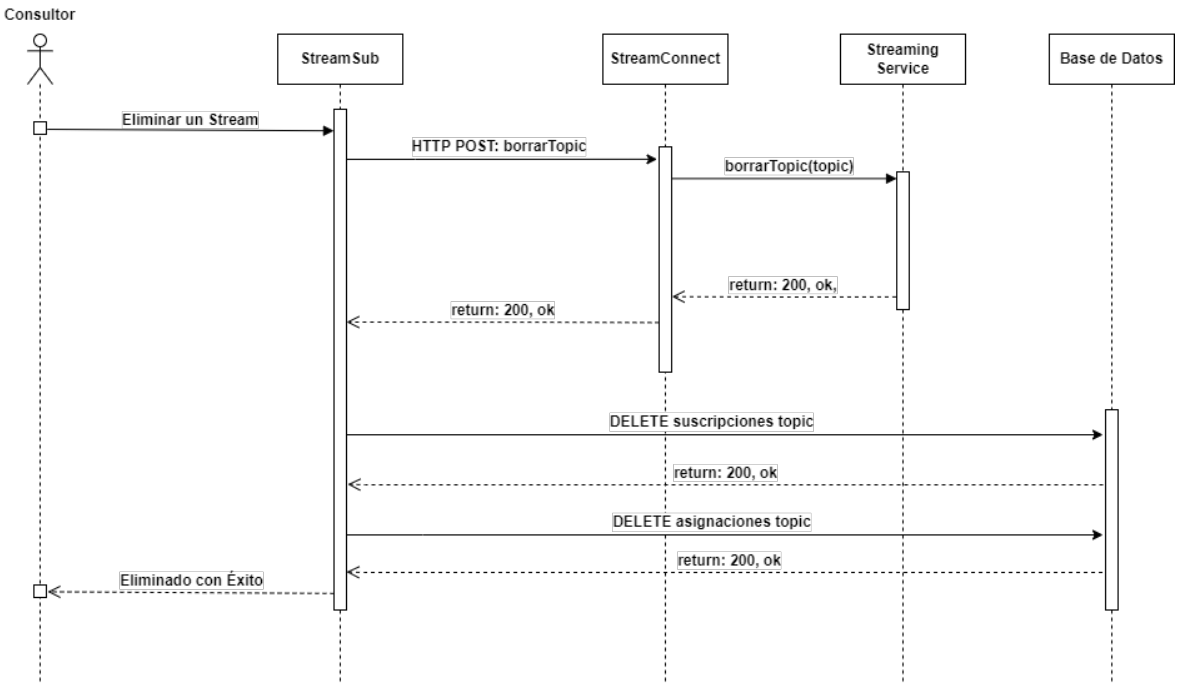


Figura C.3: Diagrama de Secuencia Borrar *Topic*