FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Migración de lógica de comunicaciones a microservicio dedicado

AUTORES: Rohrer, Néstor Gustavo

DIRECTOR: Leonardo Corbalán

CODIRECTOR:

ASESOR PROFESIONAL: Fernando Lescano

CARRERA: Licenciatura en Sistemas

Resumen

En lo que respecta a soluciones de software, los últimos años han sido testigo de un incremento en la inclinación a los microservicios como elección frente a la necesidad de adaptación al creciente dinamismo de los avances tecnológicos. El enfoque de microservicios es una doctrina relativamente joven y en continuo crecimiento, con varios pendientes y desafíos, y con muchos conceptos que aún no han sido integrados de manera total por quienes se ven relacionados a dicho paradigma. El presente trabajo expone un ejemplo de aplicación de este enfoque, señalando aquellos detalles

Palabras Clave

Microservicios, monolítico, migración, sistemas distribuidos, computación en la nube, responsabilidad única, escalamiento, independencia, mantenibilidad, modularidad, reusabilidad, agilidad, descubrimiento de servicios, tolerancia a fallos, seguridad, alineamiento organizacional.

Conclusiones

La utilización de microservicios es un enfoque que, de estar aplicada adecuadamente y bajo las condiciones consideradas como apropiadas, otorga beneficios en cuanto a agilidad de desarrollo, seguridad y calidad en general. Tal paradigma ha mostrado un crecimiento importante en los últimos años y junto a su auge también han surgido nuevas herramientas y tecnologías como parte de esta evolución. Sin embargo, no tener el entendimiento suficiente de estos y todos los conceptos relativos a los microservicios puede conducir a efectos negativos y no deseados en aquellos sistemas en los que se aplique tal doctrina.

Trabajos Realizados

Investigación y análisis bibliográfico relacionado a principales aspectos de los microservicios.
Presentación en detalle del desarrollo desde cero de un microservicio, de su integración a una arquitectura e infraestructura existente, y de la planificación de migración de funcionalidad al microservicio creado.

Trabajos Futuros

Investigación y análisis sobre desafíos pendientes del enfoque de microservicios.
Investigación y análisis sobre avances de infraestructura en la nube y de su impacto en la creación de microservicios.
Examinación de otras herramientas que puedan simplificar el desarrollo de microservicios.
Estudio de implicaciones del desarrollo de los campos de inteligencia artificial y aprendizaje automático en el en los microservicios.

Fecha de la presentación: FEBRERO 2020

ÍNDICE GENERAL

1. INTRODUCCIÓN 3

1.1 MOTIVACIÓN 3
1.2 OBJETIVO DE LA TESIS 3
1.3 ESTRUCTURA DE LA TESIS 3

2. MICROSERVICIOS 5

2.1 TENDENCIA DEL SOFTWARE COMO SERVICIO 5
2.2 DEFINICIONES DE MICROSERVICIOS 7
2.3 CARACTERÍSTICAS DE LOS MICROSERVICIOS 8
2.3.1 Tamaño pequeño 8
2.3.2 Responsabilidad única 8
2.3.3 Autonomía 8
2.4 PRINCIPALES BENEFICIOS DE LOS MICROSERVICIOS 9
2.5 EVOLUCIÓN DE LOS MICROSERVICIOS 10
2.6 COMPARACIÓN CON SOA 13
2.7 DESAFÍOS Y DEUDAS DE LOS MICROSERVICIOS 13
2.8 REQUERIMIENTOS PARA LA MIGRACIÓN A MICROSERVICIOS 16
2.9 ¿CUÁNDO UTILIZAR MICROSERVICIOS? 17

3. DESARROLLO PROPUESTO 19

3.1 PROBLEMÁTICA A RESOLVER 19
3.2 DOMINIO DE LA APLICACIÓN PRINCIPAL Y FUNCIONALIDAD EXISTENTE 19
3.2.1 Funcionalidad relacionada al microservicio creado 20
3.3 ARQUITECTURA E INFRAESTRUCTURA DE LA APLICACIÓN PRINCIPAL Y MICROSERVICIOS 20
3.3.1 Arquitectura general de la aplicación principal 21
3.3.2 Infraestructura general de la aplicación principal 22
3.4 CREACIÓN DEL NUEVO MICROSERVICIO 24
3.4.1 Otras tecnologías utilizadas 25
3.4.2 Metodología de trabajo 33
3.4.3 Organización del equipo de desarrollo 34
3.4.4 Especificación de requerimientos 35
3.4.5 Desarrollo del nuevo microservicio 37
3.5 MIGRACIÓN DE FUNCIONALIDAD EXISTENTE AL MICROSERVICIO DEDICADO 64
3.5.1 Planificación de migración de funcionalidad 64

4. TRABAJOS FUTUROS 66

5. CONCLUSIONES 67

6. BIBLIOGRAFÍA 70

1. Introducción

1.1 Motivación

Con la tecnología como impulsora de gran cantidad de aspectos de nuestra vida cotidiana, los actores vinculados al desarrollo de software deben aprender a adaptarse y mantenerse ágiles para poder continuar aumentando o al menos manteniendo la velocidad de sus actividades.

La transición hacia una arquitectura de microservicios generalmente se considera un proceso impulsado por limitaciones técnicas de un sistema existente. Si bien eso es cierto en la mayoría de los casos, muchas de las otras razones para avanzar en esa dirección están dirigidas por requisitos de mayor nivel relacionados con la dinámica de los protagonistas vinculados a los avances tecnológicos en lo que al software respecta.

Cada vez son menos las aplicaciones de software que se construyen de manera monolítica, consistiendo probablemente de un servidor web y algún tipo de almacenamiento de datos, y en las que, frente a la necesidad de contar con mayor potencia para procesamiento de datos, habitualmente se intenta escalar verticalmente utilizando mejores servidores, mejores cables, etc.

Hoy en día, las arquitecturas escalables forman una parte clave de cualquier aplicación exitosa a gran escala. Pero las complejidades de administrar tales aplicaciones se intensifican cuando, por ejemplo, la aplicación se ejecuta en la infraestructura de la nube.

A pesar del gran impacto que están teniendo en la actualidad los sistemas distribuidos y, en especial, el enfoque de microservicios, aún existen muchas características que no han sido incorporadas y comprendidas de manera total por quienes se encuentran vinculados directa o indirectamente con la utilización de estos estilos de desarrollo de software.

Es esto último lo que motiva la presentación del presente trabajo. La existencia de una brecha de desconocimiento de las principales características del enfoque de microservicios es lo que impulsa por un lado a realizar un relevamiento de lo que se considera la información más destacada en este sentido, y por otro, a aportar un ejemplo práctico y concreto de la utilización de un enfoque cuyo auge aún continúa en crecimiento.

1.2 Objetivo de la tesis

El objetivo general que se pretende alcanzar con la exposición del presente análisis es el de brindar una descripción detallada de las principales características que se vinculan al término de microservicios, así como también aportar un ejemplo concreto y relacionado de implementación que soluciona una problemática específica, todo esto con la intención de aportar una porción más de documentación que pueda ser utilizada para futuras referencias o como base de apoyo ante la necesidad de resolución de problemas similares.

1.3 Estructura de la tesis

En esta sección se describe la forma en que está organizado el presente trabajo.

La actual tesis se encuentra compuesta por dos partes generales: una primera parte en la que se brinda información de fondo sobre el tópico de microservicios, y una segunda parte en la que se describen cada uno de los aspectos involucrados en el proceso que se llevó a cabo para realizar el desarrollo que aquí se expone.

En la primera parte, se comienza presentando una breve descripción sobre el estado actual de la utilización de microservicios como una solución tecnológica. Luego, se introducen una serie de definiciones y se analizan los principales puntos en común de las mismas con la finalidad de poder comprender mejor las distintas interpretaciones del concepto. Seguidamente, se presenta un sucinto análisis de las características esenciales de los microservicios.

La sección que sigue en el desarrollo de la primera parte detalla los que se consideran los principales beneficios de los microservicios y, a continuación, se ofrece una reseña acerca de cómo dicha práctica ha progresado desde que la idea principal se comenzó a gestar. Posteriormente, se ofrece una comparación entre el enfoque de microservicios con el patrón arquitectural SOA y se continúa con otra sección en la que se enumeran y describen una serie de aspectos que se consideran pendientes de mejoras por parte de los microservicios.

Ya finalizando esta primera parte de la exposición, se hace referencia por un lado a aquellos aspectos que se entienden como mandatorios para la tarea de mover funcionalidad hacia un microservicio y, por otro lado, a las situaciones interpretadas como oportunas para considerar la utilización de un enfoque de microservicios.

La segunda parte, se inicia realizando un detalle acerca de lo que representa el problema planteado que dio origen a las subsiguientes tareas de definición, análisis y desarrollo del microservicio y de la migración de funcionalidad existente a este último. Luego, se realiza una explicación acerca de la funcionalidad de la aplicación principal, sobre su campo de acción y sobre la funcionalidad existente que se encuentra vinculada a los requerimientos que llevaron a la creación del servicio.

Se continúa con un apartado destinado a presentar información contextual acerca de la arquitectura e infraestructura de la aplicación principal a la que se vinculará el microservicio creado, y seguidamente se da lugar a la pormenorización de todo lo implicado en las tareas de desarrollo de la nueva aplicación.

Casi finalizando el trabajo, y como última sección de la segunda parte, se relata brevemente el estado de las tareas de migración de funcionalidad ya existente al nuevo microservicio.

Finalmente, se presentan las conclusiones a las que se arribaron en relación al trabajo presentado y se mencionan las que se consideran posibles extensiones de funcionalidad a futuro.

2. Microservicios

En este capítulo se describen diferentes conceptos relacionados a la idea de microservicio, con el objetivo de brindar las definiciones y el marco conceptual necesario para comprender de manera más adecuada el trabajo presentado.

2.1 Tendencia del software como servicio

Los sistemas distribuidos han ido adoptando soluciones más cercanas al tipo de "grano fino" en las últimas décadas, cambiando de aplicaciones pesadas y monolíticas a microservicios pequeños y auto-contenidos (ver Figura 1).

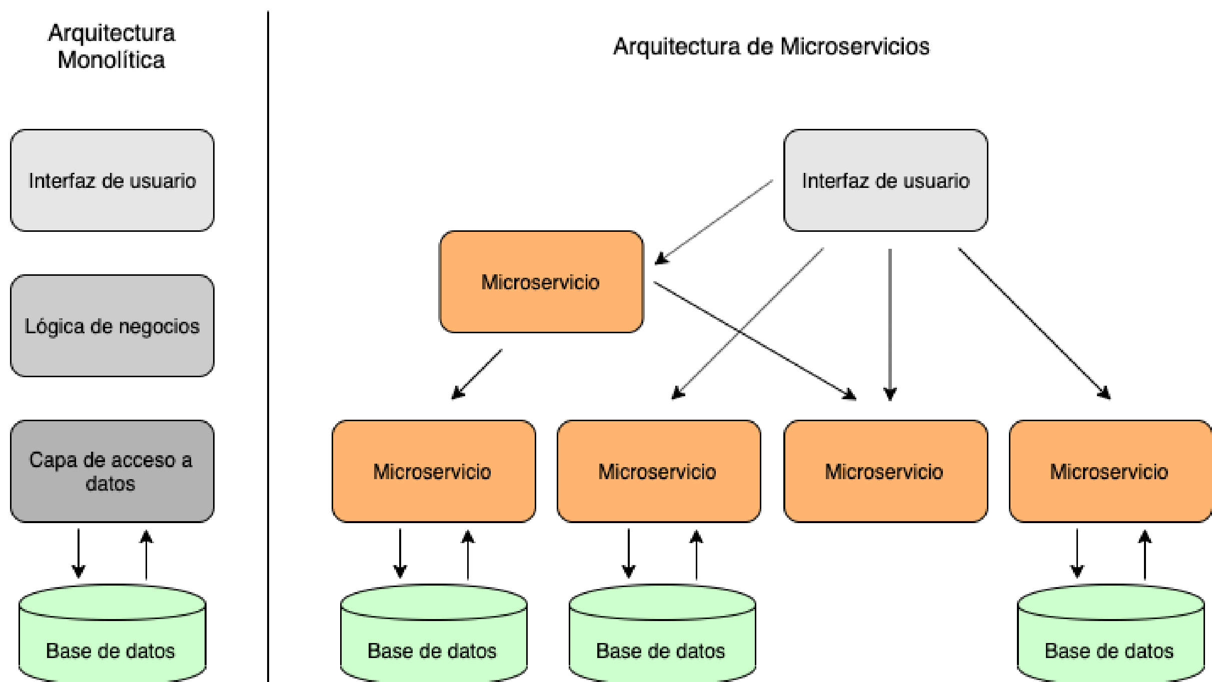

Figura 1. Arquitectura monolítica y arquitectura de microservicios [1]

El hecho de satisfacer la demanda por parte de la industria de la entrega de soluciones de software de manera rápida, ha traído consigo cambios en la automatización de la infraestructura, en la manera de realizar pruebas y en las diferentes técnicas de entrega continua de software.

Experimentar con arquitecturas de "grano fino" en este contexto, también acarreó otros resultados como mejoras en la manera de escalar los diseños (ver Figura 2), equipos de trabajo más autónomos, o en la adopción de nuevas tecnologías de forma más sencilla.

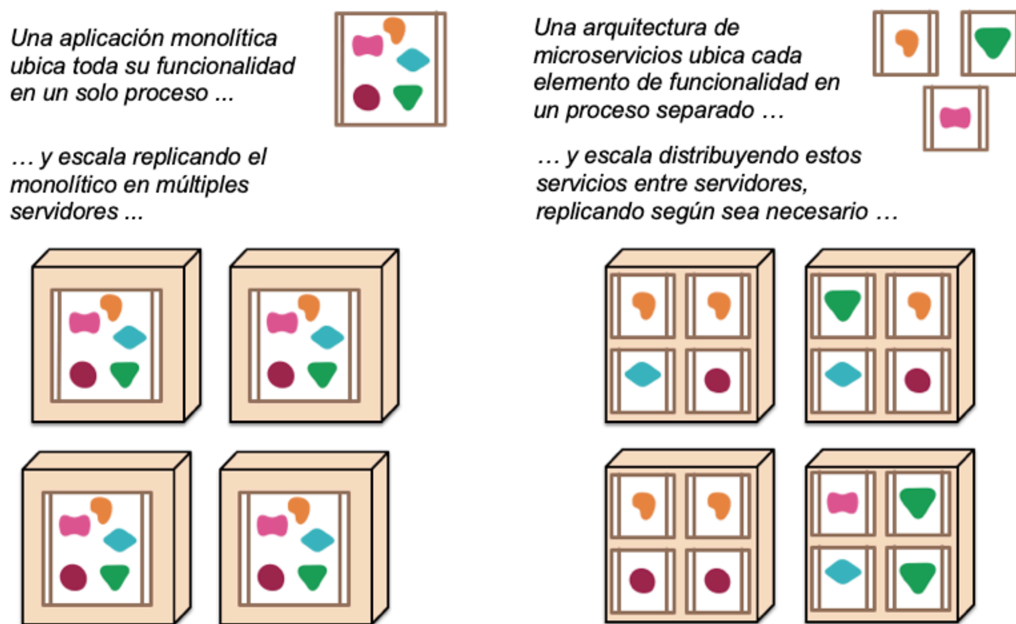

Figura 2. Aplicaciones monolíticas y microservicios [2]

Muchas organizaciones han encontrado que trabajando con soluciones arquitecturales de este tipo pueden entregar software de manera más eficiente e incorporar nuevas tecnologías a la par. Adicionalmente, el enfoque de microservicios como una forma predeterminada de crear productos de software es importante para reducir la erosión del software y la deuda técnica a largo plazo. Cada vez más equipos adoptan este enfoque para administrar y construir sus productos, y esta tendencia en el mundo del desarrollo de software en los últimos años queda en parte evidenciada por el incremento en el interés por el mismo (ver Figura 3).

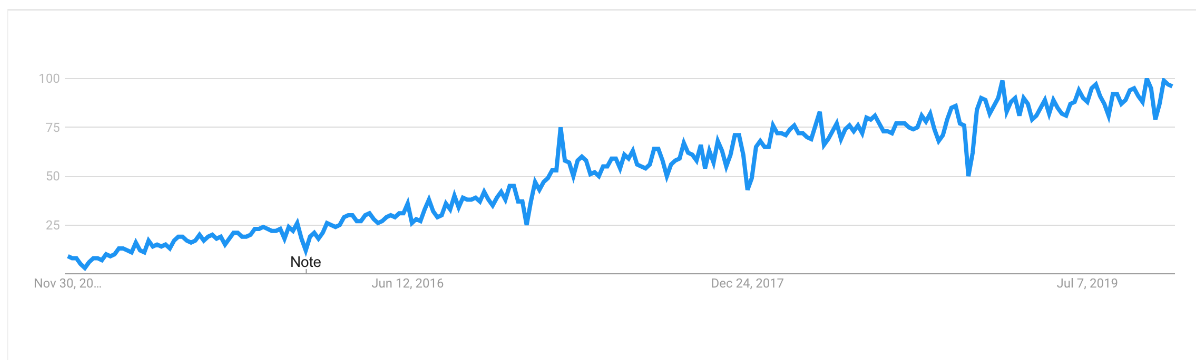

Figura 3. Aumento en el uso de palabra clave "microservicios" en los últimos 5 años, según informe de Google Trends [3]

Los microservicios brindan mayor libertad para reaccionar y tomar diferentes decisiones, permitiendo adaptarse rápidamente al cambio tecnológico continuo.

2.2 Definiciones de microservicios

A continuación, se listan algunas definiciones del término microservicio encontradas en la bibliografía consultada:

- En [4] se define a los microservicios como servicios pequeños y autónomos que trabajan juntos.
- En [5] se llama microservicio a un proceso cohesivo e independiente que interactúa a través de mensajes.
- En [6], Johannes Thönes describe a un microservicio como una aplicación pequeña que puede ser desplegada de manera independiente, que puede escalar de manera independiente, que puede ser probada de manera independiente, y que tiene una responsabilidad única. Con respecto a este último aspecto, se aclara que es una responsabilidad única en el sentido original de que tiene una única razón para cambiar y/o una única razón para ser reemplazado, pero, por otro lado, tiene una responsabilidad única en el sentido que hace una y solo una cosa y puede ser fácilmente entendida.
- En [7], se define a los microservicios como un estilo de arquitectura de software que pone énfasis en dividir el sistema en pequeños y livianos servicios que son creados deliberadamente para realizar una lógica de negocio estrechamente ligada, y que resulta de una evolución del estilo tradicional de la arquitectura orientada a servicios.

Si bien difieren las formas de especificar el concepto, la mayoría posee ideas claves como denominadores comunes. De las definiciones se pueden extraer características que prevalecen como tamaño pequeño, autonomía e independencia y código destinado a una única funcionalidad.

Tomando en cuenta la etimología de la palabra microservicio, es posible notar como el aspecto del tamaño es algo inherente al concepto. Si bien es tal vez la característica más subjetiva que aparece en la bibliografía debido a sus diversas interpretaciones, puede pensarse que para que un microservicio sea considerado como tal, al menos no deber dar la sensación de que el mismo es “grande”, tal y como se expresa en [4]. Aclarando lo anterior, y aportando una interpretación más, se podría definir la idea de tamaño pequeño de manera que si al tomar diferentes aspectos de nuestro sistema (como por ejemplo líneas de código, dependencias, recursos de infraestructura, etc.), se pueda afirmar que los mismos no pueden ser reducidos, divididos o acotados sin que el propósito principal del servicio se vea afectado.

En cuanto a la condición de tener una única responsabilidad, puede pensarse que la misma se desprende de la cualidad de un tamaño pequeño, o dependiendo del punto de vista, también es posible afirmar que realizar solo una tarea lleva a que la aplicación no sea “grande”. Es decir, para que un servicio sea pequeño, no debería hacer más de una cosa, y para hacer solo una cosa, no debería tener tanta complejidad. Más allá de estos razonamientos, es crucial que el sistema sea responsable de tan solo una función, a fin de evitar el acoplamiento entre funcionalidades y las desventajas que esto trae asociado, como por ejemplo tener que modificar varias partes del sistema al cambiar solo una funcionalidad.

Por otro lado, los microservicios deben ser sistemas autónomos e independientes. Es decir, deben poder ejecutar su lógica sin depender de algún factor exterior y con control propio de sus recursos.

Finalmente, y por fuera de las definiciones mencionadas anteriormente, es posible también definir al enfoque de microservicios como una forma de aplicación específica del paradigma de Arquitectura Orientada a Servicios (SOA, siglas del inglés *Service Oriented Architecture*). Los microservicios han surgido en parte como un uso de las principales

características del enfoque SOA en problemas reales de la industria y han evolucionado hasta llegar a formar el paradigma que hoy en día existe.

2.3 Características de los microservicios

Seguidamente, se brinda una descripción de las principales propiedades de los microservicios.

2.3.1 Tamaño pequeño

Un microservicio es una aplicación pequeña. Pero como se mencionó anteriormente, el aspecto del tamaño pequeño es quizás el más subjetivo en las diferentes definiciones.

Se podría evaluar el tamaño teniendo en cuenta las líneas de código, aunque esto resulta problemático debido a la capacidad de expresión de los diferentes lenguajes de programación, pudiendo algunos hacer más trabajo en menos líneas de código.

Otro factor que puede ayudar en la clarificación del término pequeño, es el cómo se alinea el servicio a los equipos de trabajo: puede decirse que cumple con tal característica si puede ser manejado por pocas personas. Aunque también esto es controversial debido a que con la frase “pocas personas” se introduce otra ambigüedad, requiriendo una definición concisa para este aspecto.

Ahora bien, quizá es más acertado pensar en la idea de pequeño a partir de otros principios que definen a los microservicios. Un microservicio puede ser tan complejo o “grande” como se desee mientras esté encargado de solo una tarea, mientras que un cambio en él no requiera cambios en otros servicios, y mientras todo el comportamiento y los datos necesarios estén encapsulados dentro de sus límites.

2.3.2 Responsabilidad única

Robert C. Martins describe el *Principio de Responsabilidad Única* estableciendo que "*Se debe reunir esas cosas que cambian por la misma razón y separar esas cosas que cambien por diferentes razones*" [8]. Siguiendo esta idea, un microservicio debería estar focalizado en realizar solo una tarea, y debería ser desarrollado de manera cohesiva, es decir, teniendo encapsulado todo el código necesario para cumplir con su propósito.

Sin un propósito único, un microservicio termina haciendo varias tareas, creciendo de cierta manera como múltiples servicios monolíticos. Si esto sucede, no es posible aprovechar los beneficios del enfoque de microservicio y se es víctima del costo operacional.

Al seguir el principio de responsabilidad única en un microservicio, se define de manera clara el lugar en el que una funcionalidad está implementada, se permite proteger la integridad de la información al permitir prácticas de ocultamiento de información que el microservicio requiere, se asegura que un cambio en un lugar no afectará otras funcionalidades, y se facilita el mantenimiento del código.

2.3.3 Autonomía

Un microservicio es también una entidad separada, que puede ser testeada, desplegada y mantenida de manera aislada. Esto está estrechamente relacionado con la idea de bajo acoplamiento. En una arquitectura de microservicios, los servicios deben conocer

poco y nada acerca de los demás, y todas las interacciones entre ellos deberían realizarse a través del intercambio de mensajes a través de la red.

Cada microservicio debe ser responsable de los datos necesarios para realizar su funcionalidad, y estos datos no deberían estar compartidos entre servicios. Compartir información con el resto del sistema expone los detalles de implementación y viola el principio de bajo acoplamiento. Además, si múltiples servicios usan los mismos datos, podría significar que también repliquen comportamientos, lo que va en contra de la alta cohesión y que a su vez genera que en caso de tener que modificar un comportamiento, debamos hacerlo en todos los servicios relacionados.

2.4 Principales beneficios de los microservicios

A continuación, se brinda una descripción de algunos de los beneficios más importantes de la utilización de microservicios:

- **Usos de tecnologías variadas**: utilizar una arquitectura de microservicios permite que cada servicio sea desarrollado utilizando el conjunto de tecnologías que se considera más adecuado para la función que debe cumplir. Lo mismo es válido para el caso en que el servicio deba adaptarse frente a nuevos requerimientos.

- **Manejo de fallos**: en un sistema monolítico, si alguna funcionalidad falla, es muy probable que todo deje de funcionar. Con una arquitectura de microservicios, es posible construir sistemas que manejen el fallo total de servicios y moderen la funcionalidad acordemente.

- **Escalabilidad**: con servicios más pequeños, es posible aumentar la capacidad de trabajo de aquellos que así lo requieren. Actualmente esto puede incluso hacerse bajo demanda, permitiendo controlar costos de utilización de recursos de manera más eficiente.

- **Facilidad de despliegue**: un cambio de solo una línea en una aplicación monolítica de un millón de líneas requiere que la aplicación entera sea desplegada a fin de liberar el cambio. Este tipo de despliegue termina ocurriendo con poca frecuencia debido a temores comprensibles por parte de quienes deben llevarlo a cabo. A su vez, esto provoca que los cambios se vayan acumulando entre lanzamientos de versiones del sistema, llegando a versiones productivas con cambios masivos. Y a mayor delta entre lanzamiento de versiones, mayor el riesgo de que algo suceda de manera incorrecta.

  Con microservicios, un cambio puede realizarse a un servicio y desplegar el mismo independientemente del resto del sistema. Esto es válido tanto para corrección de errores como para nueva funcionalidad.

- **Alineamiento organizacional**: los problemas asociados con grandes equipos de trabajo y con grandes bases de código pueden agravarse cuando los equipos están distribuidos.
  Los microservicios permiten un mejor alineamiento de la arquitectura con la organización de trabajo, ayudando a minimizar el número de personas trabajando en cualquier base de código hasta alcanzar el punto de equilibrio entre el tamaño del equipo y la productividad.

- **Composición**: una de las promesas claves de los sistemas distribuidos y de las arquitecturas orientadas a servicios es que se abren oportunidades para reutilizar

funcionalidad. Con microservicios, se permite que la funcionalidad sea consumida de diferentes maneras y con diferentes propósitos.

- **Optimización para el reemplazo**: con servicios independientes y pequeños en tamaño, el costo de reemplazarlos por una mejor implementación o incluso eliminarlos, es mucho más sencillo de administrar. Los equipos que utilizan microservicios se sienten cómodos con el hecho de reescribir completamente servicios cuando es requerido y con simplemente eliminarlos cuando ya no son necesarios.

Junto con estas utilidades devienen también ciertos desafíos como ser estrategias para descubrir servicios en la red, administración de la seguridad, optimización en comunicaciones, datos compartidos y la performance. Sin embargo, en cuanto estos últimos retos sean direccionados correctamente, es posible permitir al sistema sacar provecho de los beneficios primeramente mencionados.

2.5 Evolución de los microservicios

El término "microservicios" fue primeramente discutido en un taller de arquitectura de software en mayo de 2011, y se aludió a tal con el objetivo de denotar una aproximación arquitectural común que los participantes del taller habían estado explorando [9]. Previamente, expertos de la industria ya habían estado explorando algunas de las mismas ideas, y otros términos usados en la industria en aquel tiempo para expresar conceptos similares eran "arquitectura de grano fino orientada a servicios" (*fine-grained SOA*) y "arquitectura orientada a servicios hecha correctamente" (*SOA done right*). Todos esos términos relacionados con el paradigma SOA dejaban en evidencia que los microservicios estaban cobrando fuerza, y, por otro lado, que la industria no estaba del todo satisfecha con el paradigma de SOA, como quedó reflejado con el cambio masivo de SOAP a REST, un protocolo de invocación de servicios más ligero y simple.

Por otro lado, otros conceptos de desarrollo de software jugaron un rol clave para que los microservicios pudieran emerger. Esto es especialmente cierto para el diseño dirigido por dominio (DDD, *domain-driven design*), una aproximación de desarrollo basado en modelos guiada por principios tales como contextos de organización acotados e integración continua. También fueron grandes influencias el diseño para fallas, el aislamiento de datos, la automatización de infraestructura, la agilidad a escala, los equipos multifuncionales y la propiedad de productos de extremo a extremo. Todos estos enfoques resolvieron varios de los desafíos de aplicaciones distribuidas a gran escala, así como cuestiones organizacionales enfrentadas por grandes actores de la industria.

Desde un punto de vista tecnológico, las primeras aplicaciones relacionadas a microservicios fueron fuertemente influenciadas por una nueva generación de herramientas de desarrollo, despliegue, y gestión de software. A la vez que la arquitectura de microservicios se hizo más popular, aquellas herramientas continuaron evolucionando para admitir una base de usuarios más amplia y diversa, lo que llevó a la creación de tecnologías aún más avanzadas (ver Figura 4).

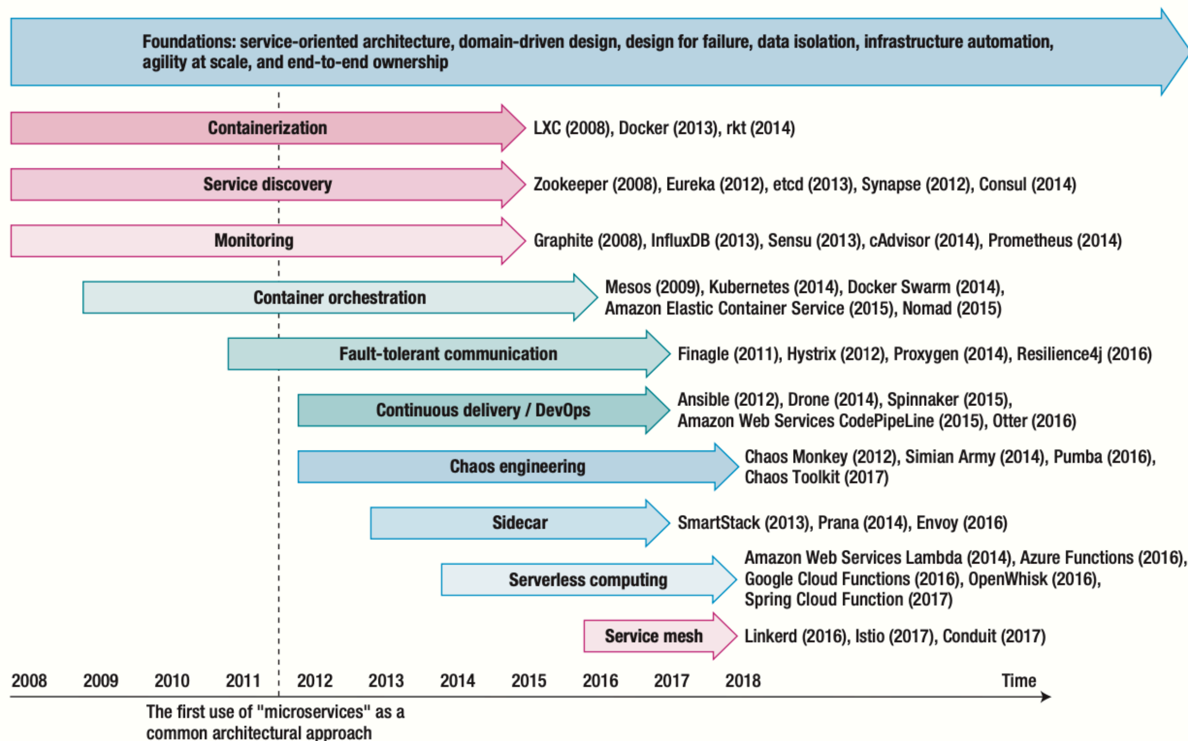

Figura 4. Línea de tiempo de tecnologías de microservicios [9]

La evolución tecnológica impactó sin dudas en el aspecto arquitectónico. La arquitectura de microservicios ha evolucionado de manera tal que actualmente es posible utilizar bibliotecas de descubrimiento de servicios y bibliotecas de comunicaciones tolerantes a fallas (ver Figura 5). Con esto, los servicios son capaces de utilizar un servicio de descubrimiento común para registrar las funcionalidades que otros proporcionaban, las pruebas e implementaciones son más simples, y es posible la reutilización de código en todos los servicios.

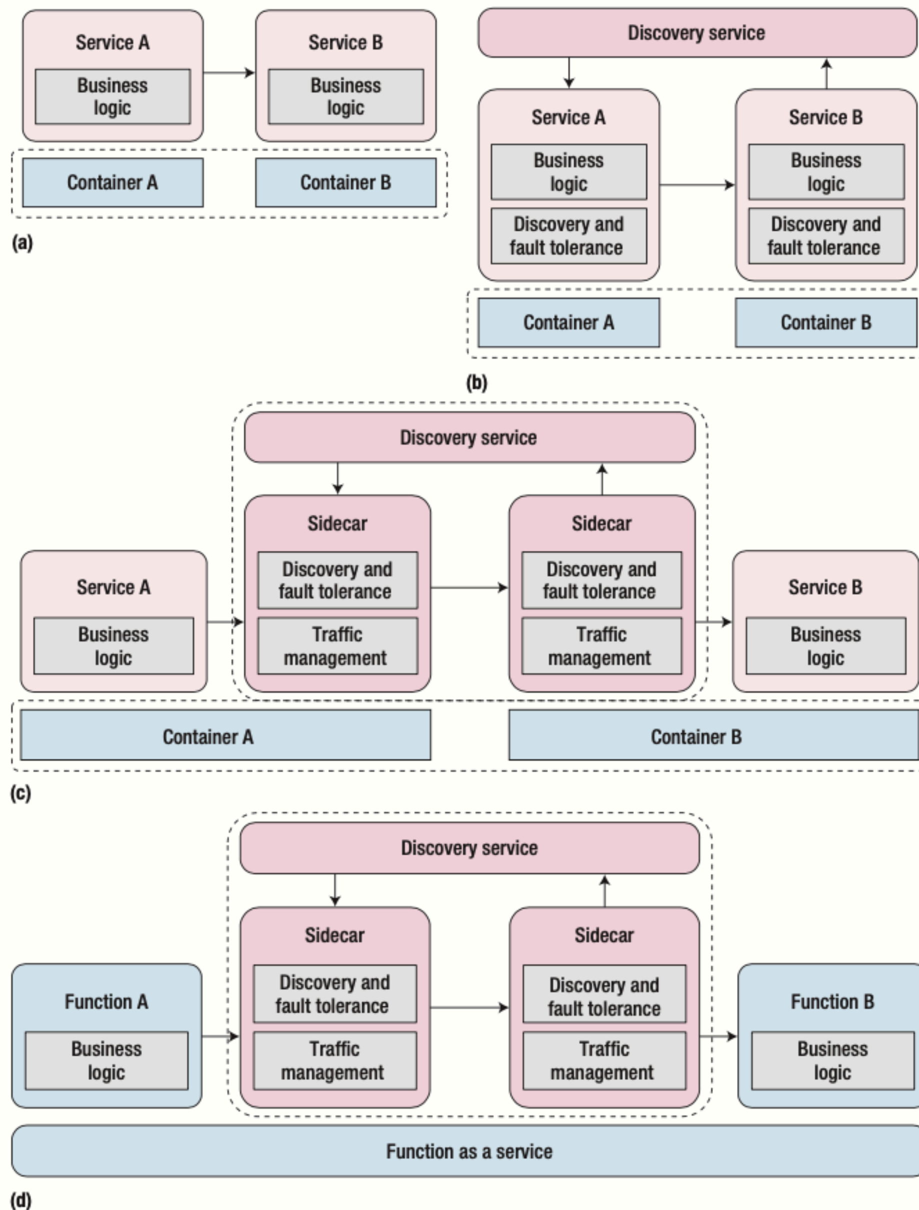

Figura 5. Cuatro generaciones de arquitectura de microservicios: (a) Orquestación de contenedores, (b) Descubrimiento de servicios y tolerancia a fallas, (c) Sidecar y malla de servicio, (d) Arquitectura sin servidor [9]

Otro punto no menor es la posibilidad de que los operadores de aplicaciones pueden monitorear y administrar dinámicamente el comportamiento de una variedad de características de comunicación de servicio a servicio, incluida la detección de servicios, el equilibrio de carga, la tolerancia a fallas, el enrutamiento de mensajes y la seguridad.

Finalmente, la más reciente generación de microservicios apunta a llevar los mismos a un nuevo plano, en el que las aplicaciones se convertirían esencialmente en colecciones de funciones "efímeras", cada una de las cuales podría crearse, actualizarse, reemplazarse y eliminarse tan rápida y arbitrariamente como sea necesario.

2.6 Comparación con SOA

SOA es un patrón arquitectural donde múltiples componentes (típicamente aplicaciones) se integran para proporcionar un conjunto final de capacidades. SOA surgió como un enfoque para combatir los desafíos de las grandes aplicaciones monolíticas, y tiene como objetivo promover la reutilización del software.

Si bien existen similitudes, SOA tiende a depender en gran medida de productos como buses de servicios empresariales u otros middlewares de peso similar, y se considera principalmente como una solución de integración, mientras que los microservicios solo se basan en tecnologías ligeras, haciendo foco en el nivel de aplicaciones.

Por otro lado, existe una falta de consenso sobre cómo utilizar de manera correcta el enfoque SOA. Debido a esto, se presentan problemas que son en realidad inconvenientes con protocolos de comunicación (por ejemplo, SOAP), falta de orientación sobre la granularidad del servicio u orientación incorrecta sobre cómo elegir los lugares para dividir el sistema. No se detalla lo suficiente sobre cuestiones concretas, las formas prácticas de garantizar que los servicios no se acoplen demasiado. La falta de especificación es donde se originan dificultades asociados con SOA.

El enfoque de microservicios surge en gran medida del uso de la mayoría de las ideas de SOA en el mundo real, lo que permite comprender mejor los sistemas y la arquitectura para emplear adecuadamente el enfoque SOA. En cierta forma, es una evolución o segunda iteración que sucede a partir de preponderar el pragmatismo sobre la teoría asociada a SOA, con el objetivo de eliminar los niveles innecesarios de complejidad para centrarse en la programación de servicios simples que implementen efectivamente una única funcionalidad.

En este punto de análisis, resulta totalmente acertado traer a colación y basarse en las tres características principales de los microservicios para marcar las diferencias más notorias con aplicaciones de una arquitectura SOA: un microservicio es relativamente más pequeño que un servicio típico, toda la funcionalidad relacionada está encapsulada dentro de él, y éste es autónomo e independiente de otros servicios.

2.7 Desafíos y deudas de los microservicios

A continuación, se describe una serie de aspectos que pueden ser considerados como puntos débiles del enfoque de microservicios, o acaso factores en los que son necesarios avances y ajustes con la finalidad de que el paradigma pueda seguir progresando.

- **Comunicación**: un mecanismo de comunicación correcto es vital en cualquier arquitectura de microservicios. Identificar el protocolo correcto, los tiempos de respuesta esperados, los tiempos de espera, y el diseño de API son claves en este sentido para construir un sistema confiable.

  Relacionado con las interfaces de comunicación, y dado que la autonomía de los microservicios permite utilizar la tecnología más adecuada para su desarrollo, una desventaja que se presenta es que las diferentes tecnologías típicamente tienen diferentes medios para especificar la forma en que se comunican los servicios. Esto hace que la comunicación entre un servicio y su cliente sea propensa a errores, debido a posibles ambigüedades. Es cierto que existen mecanismos para la especificación formal de los tipos de mensajes, los cuales pueden utilizarse para definir interfaces de servicio independientemente de tecnologías específicas. Sin embargo, aún no está claro cómo adaptarlos para implementar la comprobación en tiempo de ejecución de los mensajes para algunos estilos de arquitectura generalizados para microservicios, donde las interfaces están restringidas a un conjunto fijo de operaciones y las

acciones se expresan en rutas de recursos dinámicos. Otro problema similar es tratar de aplicar la comprobación de tipos estática a lenguajes dinámicos que se emplean en gran medida en el desarrollo de microservicios.

Otro aspecto relacionado a puntos frágiles en la comunicación es el orden en que se intercambian los mensajes. Falta de sincronización durante una sesión de comunicación entre servicios pueden traer problemas. Si bien existe trabajo en progreso para evitar esta clase de situaciones, como los *tipos de comportamientos* y la *coreografía de servicios*, ninguno de estos recursos es ampliamente adoptado en la práctica aún, debido a restricciones que se imponen en la forma de escribir los servicios y a la falta de especificaciones para utilizar estos recursos de manera eficiente.

- **Seguridad**: la seguridad en los microservicios se plantea como otro de los desafíos provenientes de los sistemas distribuidos. La forma de comunicación de los microservicios es más compleja que la de un sistema monolítico, y de esta complejidad surgen puntos a los que se debe prestar mayor atención si la seguridad del sistema importa.

  Por un lado, las engorrosas interacciones entre servicios hacen que las tareas de análisis y monitoreo de la aplicación posean un mayor grado de dificultad, punto débil que puede ser utilizado como eje en la planificación de ataques maliciosos. Además, en contraste con la forma de comunicación de una aplicación monolítica, donde las interacciones se dan en la misma aplicación, la comunicación de los microservicios a través de interfaces expuestas en la red incrementa las posibilidades de ataques.

  Desde otro punto de vista, están las relaciones de confianza que se generan en las múltiples y expuestas interacciones de los microservicios. Asumir que ninguno de los servicios de un sistema puede ser atacado representa un riesgo que puede llegar a comprometer la aplicación entera. En este sentido, son necesarias herramientas de seguridad que restrinjan la confianza que se tiene en microservicios individuales, aminorando el daño potencial si alguno de estos es atacado.

  Finalmente, se tiene la heterogeneidad relativa a los microservicios: la posibilidad de muchas tecnologías diferentes interactuando, fruto de la característica autónoma de los microservicios, hace que la seguridad y confianza sean aspectos aún más desafiantes al momento de construir los sistemas. Aún se requiere un mecanismo que permita garantizar un funcionamiento global seguro para cubrir este escenario.

- **Tolerancia a fallos**: la capacidad de un sistema de recuperarse frente al fallo de uno de sus componentes es sin duda algo que debe ser considerado al momento del desarrollo.
  Continuando con las atribuciones a las complejidades de los sistemas distribuidos, son varias las razones por las que un microservicio puede fallar. Problemas de comunicación en la red, inconvenientes a nivel de aplicación, fallas de hardware. Todas dependencias de los servicios que pueden tener impactos no deseados en el sistema.

  Si bien existen herramientas para controlar fallas y evitar su propagación, es necesario aún más trabajo para avanzar en la gestión automatizada de estos errores.

- **Descubrimiento de servicios**: el carácter autónomo de los microservicios permite que estos puedan ser empaquetados y ejecutados como un proceso, habilitando de esta manera, y entre otras cosas, a que los mismos puedan ser desplegados de manera dinámica. Es posible levantar y detener instancias según la demanda de tráfico, o ejecutarlas en distintas máquinas virtuales para optimizar su uso y ahorrar costos.

  La vida de la instancia de un servicio puede ser de solo unas horas, fundamentalmente en organizaciones sensibles a los costos que aprovechan esta

característica para distribuir las cargas de trabajo en las instancias de cómputo más económicas disponibles.

En este contexto, es necesario un mecanismo en el que el servicio cliente se comunique de manera consistente con una o más instancias de un servidor. Es imprescindible conocer la forma en que se dirigirá el tráfico a cada una de las máquinas disponibles, y la forma en que se pueden agregar o quitar recursos. Los servicios deben poder registrarse y anunciarse ellos mismos, y así facilitar la tarea de descubrir los puntos finales y las ubicaciones.

Si bien hoy en día existen herramientas reutilizables de descubrimiento de servicios, la elección de la mejor estrategia es aún un tema no menor y que debe ser tomado con especial atención.

- **Rendimiento**: utilizar un enfoque de microservicios generalmente implica un incremento en las actividades de comunicación entre elementos que ahora se encuentran distribuidos, lo cual también genera una dilación adicional en la experiencia del usuario.

  Es necesario utilizar las herramientas de sincronización y las estrategias para compartir datos más adecuadas a fin de evitar cualquier sobrecarga en la comunicación. Encontrar el diseño adecuado de cada microservicio, con el tamaño, los datos requeridos, la responsabilidad y las interfaces correctas, son los principales retos en este sentido, todo con el objetivo de evitar cualquier impacto negativo en el rendimiento del sistema.

- **Registro y seguimiento de actividad**: en sistemas basados en microservicios, el entendimiento del comportamiento del sistema como un todo es vital para mantener la salud del ecosistema de la aplicación y para poder realizar tareas de depuración relacionadas. El registro y rastreo de la actividad de los servicios son cruciales en virtud de ello. Identificar las interacciones libradas a partir de una acción de un usuario, o poder determinar la raíz de un error, son algunos ejemplos de tareas imprescindibles que deben poder realizarse. Determinar el proceso de gestión de registros más adecuado es clave en este aspecto.

- **Monitoreo de rendimiento y manejo de recursos**: otro desafío que se desprende de la particularidad distribuida de los microservicios es el control o supervisión de los diferentes recursos que componen la arquitectura del sistema: máquinas virtuales, contenedores, balanceadores, servicios, registros, etcétera. Actualmente existen tecnologías que permiten definir métricas y condiciones a partir de las cuales se pueden automatizar alertas que facilitan la gestión del sistema. Sin embargo, es vital recopilar datos actualizados sobre el estado del sistema sin caer en la situación de encontrarse en una tormenta de información irrelevante que impide y/o dificulta la tarea de tomar resoluciones pertinentes frente a situaciones que sí lo requieren.

- **Despliegue y escalabilidad**: tener una forma consistente de construir, testear, desplegar y gestionar los servicios es otro punto crucial. Si bien existen las herramientas que hacen que la implementación y las operaciones sean actividades relativamente sencillas, la selección de la solución correcta es fundamental ya que todas las herramientas poseen detalles particulares que deben entenderse para poder sacarles el mayor provecho.
  Además, la selección de la plataforma más apropiada influye significativamente en la arquitectura final del sistema, y de esto depende en gran medida la escalabilidad de los microservicios.

- **Granularidad de servicios**: como se dejó en claro, existe una falta de acuerdo sobre el tamaño correcto de los microservicios. Y si bien esta falta de consenso tal vez no

es tan importante, no deja de representar un desafío más la posibilidad de establecer un conjunto de patrones que funcionen como una guía frente a decisiones de diseño cuando se está dividiendo un dominio en microservicios y dimensionando cada servicio.

- **Cultura organizacional**: el alineamiento organizacional frente a los microservicios puede plantearse también como un problema si no es gestionado de manera correcta. Un escenario no deseado podría ser el tener varios equipos autónomos que desarrollan servicios desplegados de forma independiente y donde cada equipo toma decisiones locales sin tener en cuenta a los otros equipos. Algunas de estas determinaciones pueden ser la adopción de soluciones de infraestructura que son difíciles de comunicar y reutilizar a través de los servicios, o el hecho de tratar de solucionar problemas locales que son responsabilidad de otros equipos. Es decir, si los equipos de trabajo no son dirigidos de manera adecuada, teniendo en cuenta las necesidades y objetivos de todo el sistema, puede existir el riesgo de que al no ver el panorama completo se tomen decisiones que no son coherentes en el contexto de la arquitectura general.

Cabe destacar que es posible comprobar que a medida que la adopción de microservicios se va afianzando y su utilización es cada vez más común, también van surgiendo nuevas herramientas y mejoras como respuesta a todos los desafíos y problemas nombrados.

Por otro lado, es acertado igualmente recordar que adoptar un esquema de microservicios puede no ser la solución correcta en todos los casos, así como también, que hay situaciones en las que los microservicios serían una buena solución, pero no siempre se logra implementarlos de manera correcta.

2.8 Requerimientos para la migración a microservicios

Seguidamente se describe una serie de puntos claves e imprescindibles a tener en cuenta para los casos en los que se desea realizar un proceso de extracción de una funcionalidad a un microservicio.

- **Identificar la funcionalidad a separar**: es primordial detectar los límites del código base que componen la funcionalidad que se desea separar. También identificar la información relacionada en la base de datos y establecer mecanismos para separar la misma. Reconocer y determinar específicamente qué código junto con todas sus dependencias se extraerá permite planificar mejor la reforma del sistema, así como también disminuir los riesgos de fallas que pueden aparecer al remover código acoplado a otras funciones de nuestro sistema.

- **Contar con un aprovisionamiento automatizado de infraestructura**: es necesario contar con los recursos computacionales que permitan correr el código del microservicio que se va a desarrollar. También es vital que, ante un eventual cambio en la demanda de trabajo de nuestro futuro servicio, podamos permitir que la infraestructura se adapte vertical y/u horizontalmente y de manera rápida.

- **Incorporar o mejorar un sistema de monitoreo de recursos y de registro de actividad**: tal vez este no sea un requisito demasiado importante cuando se cuenta con solo unos pocos microservicios, pero a medida que comienza a incrementar la cantidad de servicios es crucial poder detectar problemas graves de manera rápida.

- **Creación de flujos de trabajo de entrega continua**: de igual manera, contar con esta posibilidad se hace imperante a medida que la cantidad de servicios de nuestro sistema aumenta. Es necesario poder desplegar el sistema de manera rápida y automatizada en los diferentes ambientes de desarrollo. Poder recuperarse de un fallo inesperado o poder entregar una nueva funcionalidad y siempre de manera ágil, son algunos ejemplos de por qué este punto se convierte en cierta medida en un prerrequisito en un enfoque de microservicios.

- **Bases y seguridad para la comunicación**: se vuelve indispensable contar con el soporte necesario para poder establecer la comunicación con el microservicio ya sea vía llamadas a procedimientos remotos (en inglés, *Remote Procedure Call*, RPC), llamadas del tipo REST, o vía mensajes y eventos. Se debe poseer además una estrategia para el caso en que el servicio deba ser expuesto a otros clientes, y considerar aspectos como seguridad, alertas, ruteo de peticiones, etcétera. El aspecto de la autenticación y autorización de las llamadas al servicio tampoco es menor. Se requiere un sistema de validación de las peticiones, y si es posible, uno que no sea redundante y agregue demasiada carga de procesamiento en las llamadas.

- **Conocimiento, cultura y organización de equipos de trabajo**: se debe contar con un equipo de trabajo y una estructura de comunicación en la organización que no solo sean compatibles con todas las complejidades que introduce la utilización de microservicios, sino que además permitan combatirlas y evolucionarlas a conceptos cada vez más sencillos de gestionar.

2.9 ¿Cuándo utilizar microservicios?

A continuación, se listan algunos de los patrones y condiciones considerados como más relevantes para identificar situaciones en las que sería conveniente utilizar un enfoque de microservicios:

- **Problemas de rendimiento:** existen situaciones en donde la aplicación monolítica se puede convertir en un cuello de botella en varios sentidos. Algunas tareas pueden volverse computacionalmente pesadas haciendo que una aplicación monolítica ya lenta, lo sea aún más.

- **Problemas de escalabilidad:** cuando se lidia con un esquema monolítico resulta difícil escalar el sistema para determinadas tareas, o incluso simplemente aislar recursos para diferentes tipos de tareas. Para estos casos generalmente se termina analizando escalar la aplicación entera, lo cual es un gasto innecesario de recursos, y no siempre se logra obtener el impacto deseado en la parte del sistema que se deseaba mejorar (ver Figura 2).

- **Problemas de agilidad**: puede suceder que haya varios desarrolladores que deban crear nuevas funcionalidades sobre una aplicación única, y que éstos se vean impedidos de realizar cambios ágiles o con desconfianza para efectuar cambios grandes ya que los mismos podrían afectar otras partes en las que se están trabajando. También, hay ocasiones en las que se debe esperar a que el módulo más lento finalice su despliegue para continuar con el despliegue del resto de la aplicación, problema que podría ser evitado extrayendo el módulo en cuestión a un microservicio e independizando su proceso de despliegue.

- **Estancamiento en una tecnología y/o necesidad de un gobierno descentralizado**: habitualmente se presentan casos de dependencia con la tecnología utilizada en algunas partes de una aplicación. Esto implica que, frente a un

cambio requerido de dicha tecnología, gran parte o todo el sistema debe reescribirse o adaptarse según la situación. Por otro lado, hay ocasiones en las que se presenta un problema determinado que no puede ser solucionado de manera efectiva con la plataforma tecnológica con la que está desarrollada la aplicación. Si estas circunstancias pueden ser detectadas y se desea evitarlas, con microservicios es posible desacoplar las áreas involucradas e implementarlas con la tecnología más conveniente, y cualquier decisión de realizar un nuevo cambio solo requerirá la adaptación de ese servicio.

- **Necesidad o planificación de crecimiento**: si la necesidad de hacer crecer la aplicación es un hecho concreto, utilizar un enfoque de microservicios permite que los equipos de desarrollo se habitúen a trabajar en pequeños servicios separados desde el principio, y contar con una estructura de trabajo organizada de esta manera facilita escalar los equipos cuando es necesario introducir grandes complejidades.

3. Desarrollo propuesto

En esta sección, se describe cada uno de los principales conceptos con los cuales se trabajó en el ámbito laboral y que condujeron al desarrollo del nuevo microservicio. Se especificará la cuestión primordial que dio comienzo al trabajo realizado, se brindará detalles sobre el proceso que se llevó a cabo para desarrollar el nuevo sistema, y se aportará información contextual de distintos factores relacionados con la intención de facilitar la comprensión de las tareas realizadas.

Cabe aclarar que el microservicio aquí desarrollado fue integrado tanto a una arquitectura como a una infraestructura ya existente, y que algunos detalles de dicha integración también serán detallados como parte de la presente exposición.

3.1 Problemática a resolver

La necesidad de resolver un problema vinculado a la incorporación de funcionalidad a una aplicación existente fue lo que motivó el desarrollo del software aquí presentado. La nueva funcionalidad solicitada estaba relacionada con el hecho de permitir el soporte de comunicaciones (llamadas telefónicas y mensajes de texto) para determinados actores del sistema y a partir de condiciones que se dieran entre las interacciones de éstos (más adelante, en la sección de “Especificación de requerimientos”, se aporta más información al respecto). El dilema en concreto consistió en que ya se poseía lógica relacionada en la aplicación productiva, e incorporar los nuevos requerimientos acoplándolos a este código podría representar la génesis de futuros problemas.

Teniendo en cuenta los beneficios que la utilización de microservicios otorga, se pensó que la mejor solución sería desarrollar un nuevo microservicio que encapsulara toda la lógica relacionada a las actividades de comunicación del sistema. Por esa razón, en lugar de hacer crecer en tamaño el código de la aplicación principal, se optó por organizar la nueva funcionalidad en una mini-aplicación” diferente y separada, es decir, en un microservicio.

Este nuevo microservicio gestionaría un dominio específico de negocio, siendo responsable de realizar las tareas relativas que ya se soportaban en la aplicación principal, así como también las funcionalidades nuevas que fueron requeridas.

3.2 Dominio de la aplicación principal y funcionalidad existente

A continuación, se brindan detalles acerca del propósito de la aplicación principal para la cual se desarrolló el nuevo microservicio, y se describirá una funcionalidad ya existente la cual se encuentra vinculada al ámbito de acción del sistema que fue creado.

El proyecto de software para el cual se realizó el desarrollo expuesto en el presente trabajo surgió originalmente como un servicio de aparcacoches en línea disponible en una de las ciudades con peor tráfico del mundo, Nueva York. Luego, el mismo fue mutando hasta tener como finalidad primordial la de brindar una serie de servicios orientados al estacionamiento, movilidad, mantenimiento y otras actividades de soporte para automóviles. Su objetivo principal es hacer que el estacionamiento y todo lo relacionado al mantenimiento de vehículos sea más fácil para sus propietarios, es decir, que éstos solo se preocupen por conducir sus automóviles y no deban pensar en conseguir lugar para estacionar, en cargar nafta, en lavarlo, cambiar un neumático o hacerle algún otro servicio de mantenimiento.

Los propietarios de automóviles de la ciudad neoyorquina son atendidos por un equipo de choferes pertenecientes a la empresa propietaria de la aplicación y listos para prestar

algún servicio al automóvil siempre que el propietario lo solicite a través de una aplicación móvil. Los choferes asimismo utilizan una versión de aplicación móvil específica para realizar sus operaciones.

Algunas de las operaciones que se gestionan a través de la aplicación son: pedidos de servicios para automóviles (estacionamiento, mantener automóvil en espera mientras el propietario realiza alguna actividad, lavado de automóvil, carga de combustibles, etc.), seguimiento de actividades y ubicaciones de choferes y automóviles, gestión de turnos de choferes, gestión de suscripciones y promociones, gestión de usuarios y permisos, gestión de áreas de trabajo, gestión de parámetros del sistema, etc.

Varios de los eventos lanzados a partir de las acciones gestionadas por la aplicación requieren de notificaciones a los usuarios (tanto para los propietarios de automóviles como para los choferes). Algunos ejemplos son: un servicio pedido es cancelado, se comienza a ejecutar un servicio, se concluye un servicio, un chofer arriba a una ubicación. No solo se envían notificaciones a los propietarios de automóviles, sino que también a los choferes como, por ejemplo, cuando no se presentan a un turno de trabajo, o cuando su cuenta resulta bloqueada por algún motivo.

3.2.1 Funcionalidad relacionada al microservicio creado

El código existente en la aplicación principal, y que se encuentra vinculado a la nueva funcionalidad que condujo a la creación del nuevo microservicio, es el responsable de brindar el soporte para las notificaciones mencionadas anteriormente. Estas acciones de notificación se realizan a través de mensajes de texto SMS (servicio de mensajes cortos, del inglés *Short Message Service*) y se ejecutan cuando en el sistema se cumplen determinadas condiciones que requieren que el usuario afectado sea informado de tales sucesos. Por ejemplo, un operador del sistema crea una petición de servicio, y luego un mensaje SMS es enviado al usuario que estará involucrado en la ejecución de las tareas relativas.

Esta funcionalidad existente se implementó a partir de la integración con la plataforma de comunicaciones Twilio [10], un conjunto de servicios web que brindan soporte para el desarrollo de software relacionado a llamadas telefónicas y envío de mensajes de texto. Para esto, se debió instalar kit de desarrollo de software correspondiente, incorporando a la aplicación principal las dependencias y las configuraciones relativas, y adicionalmente se incorporaron parámetros necesarios para poder autentificarse en la plataforma de Twilio a fin de poder utilizar su API.

Es importante destacar que la información requerida para la autentificación en la plataforma Twilio varía según los ambientes de desarrollo, requiriendo datos distintos para ambientes de pruebas y de producción. Esto en sí mismo genera mayor cantidad de información de configuración solo para la integración de una tecnología en particular dentro de la aplicación principal.

3.3 Arquitectura e infraestructura de la aplicación principal y microservicios

En esta sección se describe la arquitectura general y la infraestructura de la aplicación principal, haciendo especial foco en la configuración relativa a los microservicios existentes. El objetivo del siguiente detalle es brindar información contextual acerca de cómo se encuentra planteado el sistema al que se integró el nuevo microservicio desarrollado.

3.3.1 Arquitectura general de la aplicación principal

A continuación, en la Figura 6 se presenta un diagrama de la arquitectura general de la aplicación, y luego se describe brevemente la relación entre los distintos componentes.

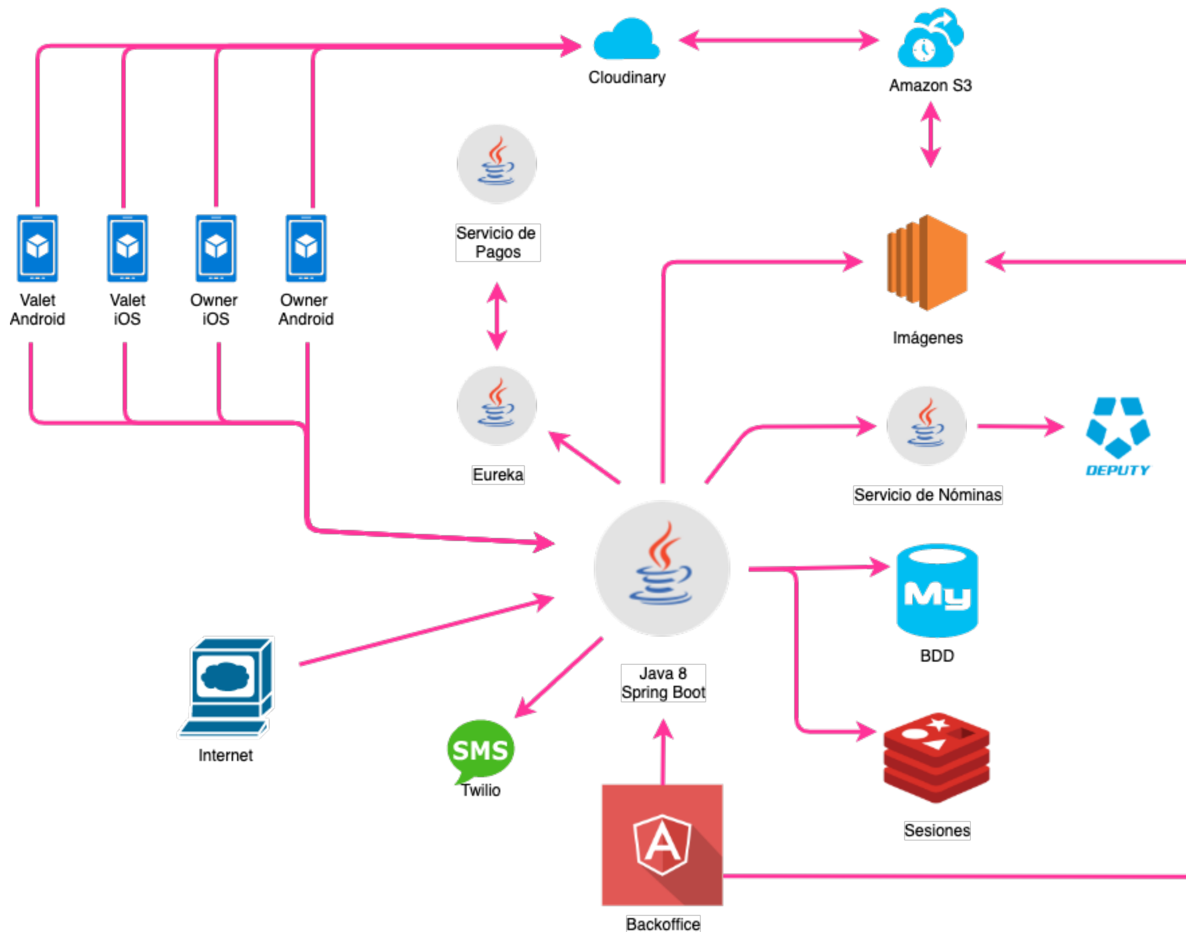

Figura 6. Modelo acotado de arquitectura de aplicación principal

Como se puede observar en el bosquejo de arquitectura, el sistema está compuesto por una aplicación Java como principal centro de las interacciones entre los distintos componentes. Esta aplicación, comúnmente denominada *backend*, representa la pieza encargada del procesamiento de la mayoría de las funciones del sistema. Con esta aplicación se comunican otros dos grandes tipos de sistemas que son las aplicaciones móviles (desarrolladas para las plataformas Android [11] y iOS [12]) y la aplicación *backoffice* (desarrollada con el framework de JavaScript AngulaJS [13]). Entre las aplicaciones móviles tenemos las utilizadas por los usuarios clientes del servicio (*Owner*) y las utilizadas por usuarios operadores (*Valet*). La aplicación *backoffice* también es utilizada por operadores, pero esta está más orientada a las tareas administrativas.

La aplicación *backend* se encarga de procesar todas las interacciones provenientes de los usuarios del sistema, para lo cual accede a distintos servicios y fuentes de datos, como ser: datos propios del modelo almacenados en la base de datos MySQL [14], datos de sesiones de usuarios en Redis [15], datos de turnos de operadores del sistema en Deputy (una plataforma de software basada en la nube para la gestión de turnos de trabajo de

empleados) [16], servicios de procesamientos de pagos y servicios de envío de mensajes de Twilio [10].

La gestión de imágenes que se utilizan tanto en el *backoffice* como en las aplicaciones móviles se realiza de manera compartida entre Cloudinary (servicio web de gestión de imágenes basado en la nube) [17] y un microservicio de imágenes que obtiene las mismas de Amazon S3, un servicio de almacenamiento de objetos en la nube [18].

Los servicios de Nóminas, Imágenes y Pagos constituyen la totalidad de microservicios presentes en el sistema. El de Pagos, es el microservicio desarrollado más recientemente. Este componente en particular fue creado siguiendo las premisas de un plan pensado para eliminar las dependencias entre los servicios existentes y aquellos que puedan crearse en un futuro, y así garantizar que las interacciones entre éstos sean más confiables y tolerantes a fallas. Entre otras características, y siguiendo el anterior propósito, este microservicio utiliza, a diferencia del resto, la herramienta de registro de servicios Eureka. Eureka es una aplicación que consta de un servicio que se utiliza principalmente en la nube de AWS [19] para localizar otros servicios y con el fin de equilibrar la carga y la conmutación por error de los servidores de nivel medio (Servidor Eureka). Eureka también presenta un componente cliente basado en Java, Cliente Eureka, que facilita mucho las interacciones con el servicio. El cliente también tiene un equilibrador de carga incorporado que realiza un equilibrio de carga básico por turnos [20].

Los lineamientos seguidos para con el microservicio de Pagos fueron los utilizados como guía en el proceso de desarrollo del trabajo aquí expuesto, y serán utilizados también en un futuro para llevar al mismo esquema a los servicios de Nóminas e Imágenes.

La herramienta Eureka y algunas de las tecnologías nombradas previamente son descritas con mayor detalle más adelante en la sección de “Principales tecnologías utilizadas”.

Por último, cada una de las interacciones del sistema se realiza a su vez a través de una infraestructura que se ilustra a continuación.

3.3.2 Infraestructura general de la aplicación principal

En este apartado se explica cómo está compuesta la infraestructura que soportará el desarrollo del nuevo microservicio.

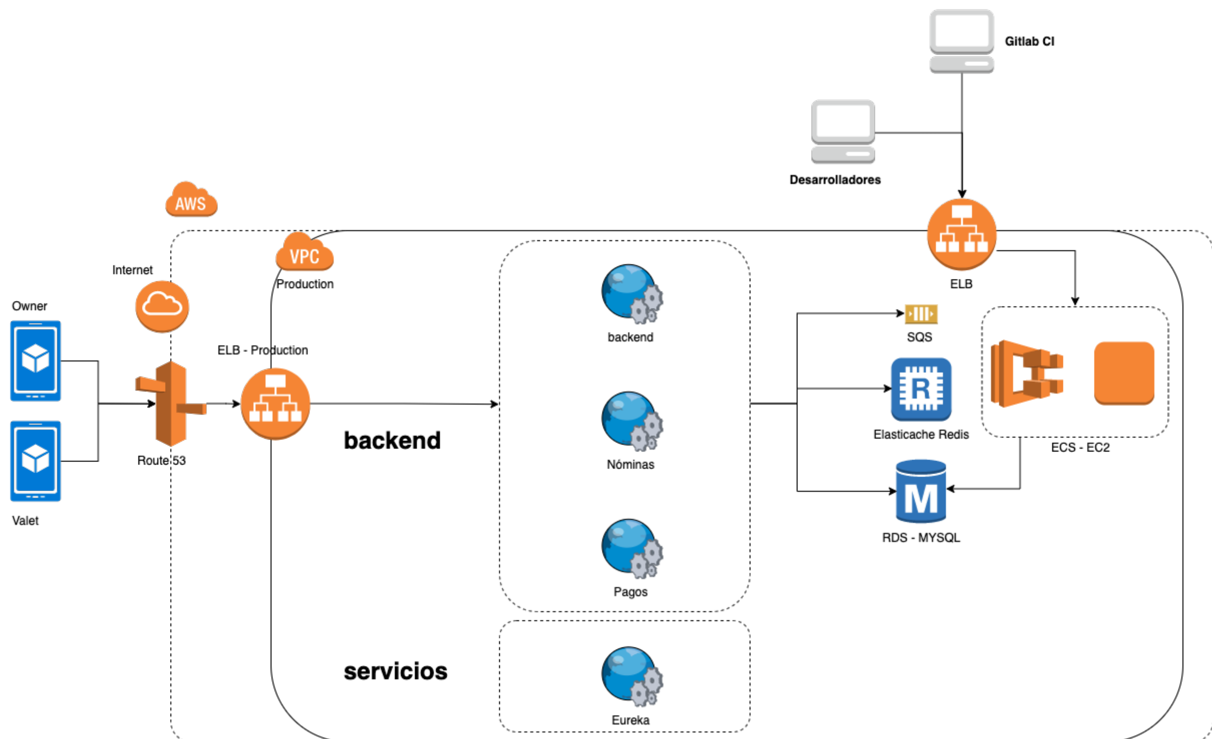

Figura 7. Modelo acotado de infraestructura de aplicación principal

La Figura 7 expone un modelo de infraestructura en el que se puede notar cómo la mayoría de los componentes de la aplicación principal hacen uso de distintos servicios de AWS (del inglés *Amazon Web Services*). AWS es una colección de servicios de computación en la nube pública (también llamados servicios web) que en conjunto forman una plataforma de computación en la nube, ofrecidas a través de internet por Amazon.com [19]. Productos de AWS son utilizados a lo largo de toda la infraestructura de la aplicación principal.

Por un lado, la comunicación hacia el *backend* y los distintos microservicios se realiza pasando primeramente por un Amazon Route 53, un servicio de DNS (sistema de nombres de dominio) web escalable y de alta disponibilidad en la nube [21]. Esta herramienta fue diseñada para ofrecer a los desarrolladores y las empresas un método fiable y rentable para redirigir a los usuarios finales a las aplicaciones en internet mediante la traducción de nombres legibles para las personas, como www.ejemplo.com, en direcciones IP numéricas, como 192.0.2.1, que utilizan los equipos para conectarse entre ellos. Junto a las características comunes de los servidores DNS tradicionales, AWS Route 53 provee además otras funciones como devolver al cliente la dirección IP del servidor que tiene la menor latencia. Es un servicio DNS con comprobaciones de estado configurables que solo devuelve la dirección IP de un nodo en buen estado.

Una vez que la comunicación es dirigida por el Route 53, la misma es gestionada por un Amazon ELB (*Elastic Load Balancing*, Equilibrio de carga elástica). Éste es un servicio de equilibrio de carga administrado que se escala automáticamente de manera vertical dependiendo del tráfico de internet [22]. Entre otras cosas, permite que las solicitudes sean dirigidas a zonas de disponibilidad en instancias que se consideran saludables. Es decir, si algunos de los servidores que se encuentran detrás del ELB deja de funcionar, el equilibrador de carga debería poder redirigir todas las solicitudes a alguno de los servidores restantes, lo que implica que no haya caída en la disponibilidad para los usuarios finales.

Asimismo, el ELB es el punto de entrada tanto para los desarrolladores como para el flujo de integración continua de GitLab. GitLab es una herramienta que entre otras utilidades brinda soporte para la configuración de flujos de trabajo de CI/CD (Integración Continua/Entrega Continua, del inglés *Continuos Integration/Continuos Delibery*) [23]. Cada vez que se concluye una funcionalidad o se arregla algún error en el código de la aplicación, se envía el código resultante al repositorio, registrando de esta manera los cambios y las versiones de la aplicación. Adicionalmente, se ejecutan una serie de pasos que integran el nuevo código e informan si dicha fusión está libre de errores o no. Detalles adicionales sobre la herramienta GitLab son presentados en la sección "Otras tecnologías utilizadas".

En el modelo de infraestructura expuesto se puede percibir además la presencia de un componente denominado VPC, el cual engloba al *backend*, los microservicios y el resto de los componentes del sistema. Amazon VPC (*Virtual Private Cloud*) es una red definida por software optimizada para el rendimiento en la transmisión de paquetes de red dentro, fuera y a través de las regiones de AWS. Es decir, permite aprovisionar una sección de la nube de AWS aislada de forma lógica, en la que se puede lanzar recursos de AWS en una red virtual definida por el usuario [24]. Dicha red virtual es prácticamente idéntica a las redes tradicionales que se utilizan en sus propios centros de datos, con los beneficios que supone utilizar la infraestructura escalable de AWS.

Por otro lado, tenemos los componentes Amazon EC2 (*Elastic Compute Cloud*, Nube de computación elástica) y Amazon ECS (*Elastic Container Service*, Servicio de contenedores elásticos). EC2 es un servicio web que proporciona capacidad computacional remota [25] y ECS es básicamente una agrupación lógica de máquinas/instancias EC2 [26]. ECS habilita una configuración que tiene por finalidad lograr un uso y gestión eficientes de los recursos de la(s) instancia(s) EC2, (almacenamiento, memoria, CPU, etc.). MySQL [14], el sistema de gestión de base de datos relacional utilizado a lo largo de la aplicación, se ejecuta haciendo uso de los servicios de EC2.

Otro de los componentes de la familia de Amazon es SQS. *Amazon Simple Queue Service* (SQS) es un servicio de colas de mensajes que permite desacoplar y ajustar la escala de microservicios, sistemas distribuidos y aplicaciones sin servidor. Con SQS se puede enviar, almacenar y recibir mensajes entre componentes de software de cualquier volumen, sin pérdida de mensajes ni la necesidad de que otros servicios estén disponibles [27]. Esto permite que las solicitudes de los diferentes componentes del sistema puedan procesarse de manera asíncrona (en lugar de una a la vez), mejorando la eficiencia en el procesamiento de los datos.

Finalmente, ElastiCache Redis (también de Amazon) es una utilidad creada sobre Redis [15] y ofrece un almacén de estructuras de datos en memoria que se utiliza como agente de base de datos, caché y mensaje [28]. En el caso de la aplicación principal, dicha herramienta es utilizada para almacenar y manejar la información de la sesión de los usuarios del sistema.

3.4 Creación del nuevo microservicio

Como se ha indicado anteriormente, la decisión de crear un nuevo microservicio surgió a partir del análisis detallado de tanto la funcionalidad que se estaba solicitando como de la funcionalidad relacionada que ya se poseía en la aplicación principal.

Una vez planteada la necesidad de la nueva funcionalidad por parte del dueño del producto de software, la misma pasó a través de un proceso de análisis, especificación y

planificación, con la finalidad de que el proceso de desarrollo se realice de la manera más eficiente posible.

A continuación, se describen diferentes aspectos y etapas que se atravesaron en el desarrollo del nuevo microservicio.

3.4.1 Otras tecnologías utilizadas

En esta sección se brinda un breve detalle sobre algunas de las demás herramientas utilizadas en las tareas de desarrollo de software. Es necesario aclarar que, si bien las tecnologías utilizadas en la aplicación principal conforman un conjunto mucho más amplio, aquí se describirán solo aquellas que están más relacionadas al proceso de creación del microservicio expuesto en la presente tesina.

IntelliJ IDEA: es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos [29]. Brinda soporte para una gran variedad de lenguajes, tecnologías y frameworks, y en conjunto provee un gran número de facilidades a los programadores para el desarrollo de software.

Jira: es una herramienta que permite gestionar el trabajo de un proyecto de desarrollo de software. Entre otras cosas, permite gestionar todo el ciclo de vida de una nueva funcionalidad o error, desde que es creado/detectado hasta que se finaliza/corrige [30]. Una de las características más utilizadas por parte de los desarrolladores es el tablero de administración (ver Figura 8). Un tablero muestra los problemas de uno o más proyectos, brindando una forma flexible de ver, administrar e informar sobre el trabajo en progreso.

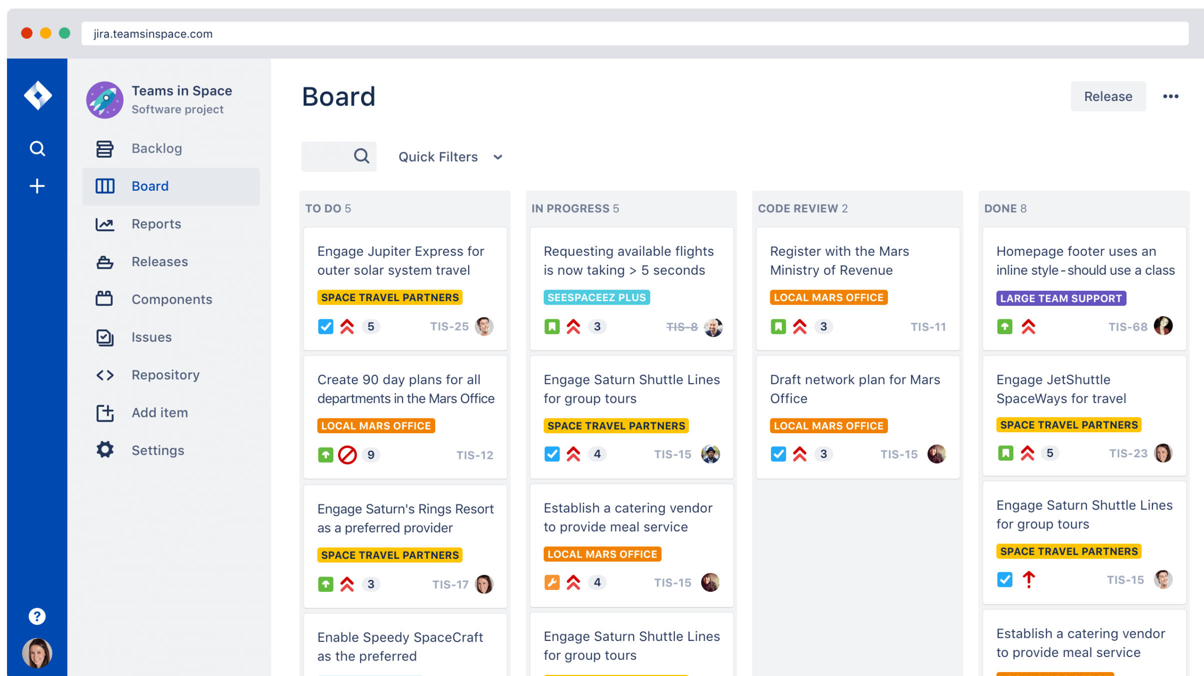

Figura 8. Ejemplo de tablero de administración Jira

Git: es un sistema de control de versiones distribuido para el seguimiento de cambios del código fuente durante el desarrollo de software. Es utilizado para coordinar el trabajo entre programadores, pero se puede usar para rastrear cambios en cualquier conjunto de archivos.

[31]. La característica de sistema distribuido significa que cada componente del mismo trabajará como un sistema de control de versiones y como una estación de trabajo a la vez (ver Figura 9). Esto sistemas no dependen de un servidor central para almacenar las versiones de un archivo del proyecto. Cada colaborador del proyecto tiene una copia local o "clon" del repositorio principal, es decir, cada uno mantiene un repositorio local por sí mismo, que en realidad es la copia o clon del repositorio central en su disco duro.

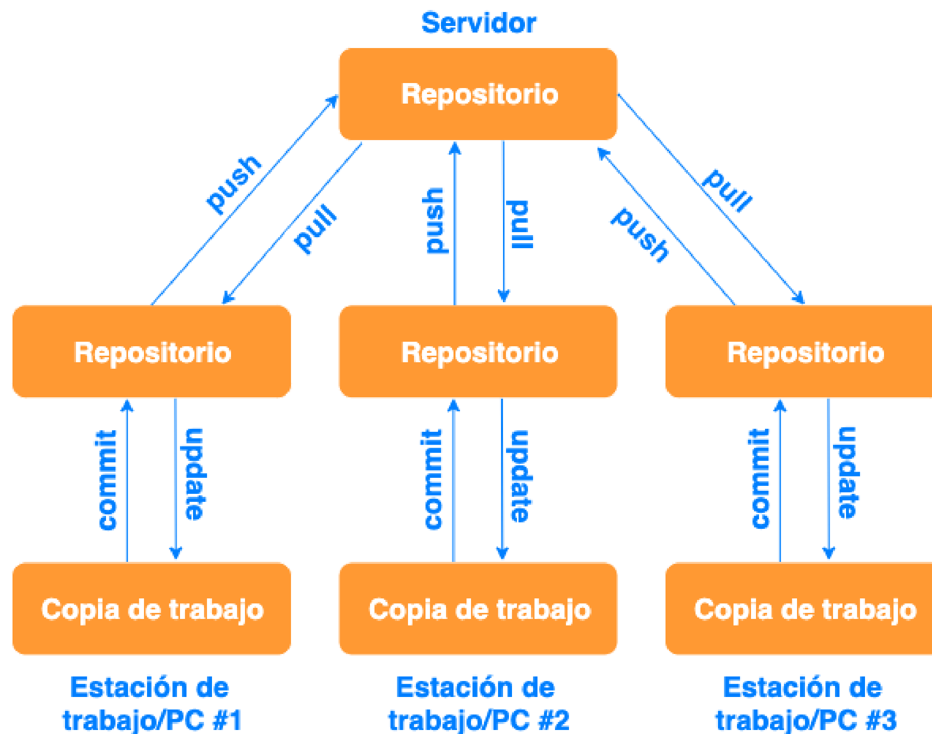

Figura 9. Representación de un sistema de control de versiones distribuido

Algunas de las ventajas de este tipo de sistema de control de versiones son:

- Todas las operaciones (excepto aquellas que interactúan con el servidor remoto) son muy rápidas, ya que la herramienta solo necesita acceder al disco duro. Por lo tanto, no siempre necesita una conexión a Internet.
- Es posible realizar la confirmación de nuevos cambios de manera local y sin manipular los datos en el repositorio principal. Una vez que se obtiene un grupo de conjuntos de cambios listos, el mismo puede ser enviado al repositorio principal subiendo todos los cambios a la vez.
- Si el servidor central se bloquea en algún momento, los datos perdidos se pueden recuperar fácilmente de cualquiera de los repositorios locales de los colaboradores.

La Figura 10 ilustra el funcionamiento básico de Git, describiendo el flujo de los cambios en las distintas áreas del sistema de control de versiones a partir de los comandos esenciales (representados por flechas). Más adelante, en la sección “Desarrollo del nuevo microservicio”, se otorgan más detalles acerca del control de versiones y el repositorio de código.

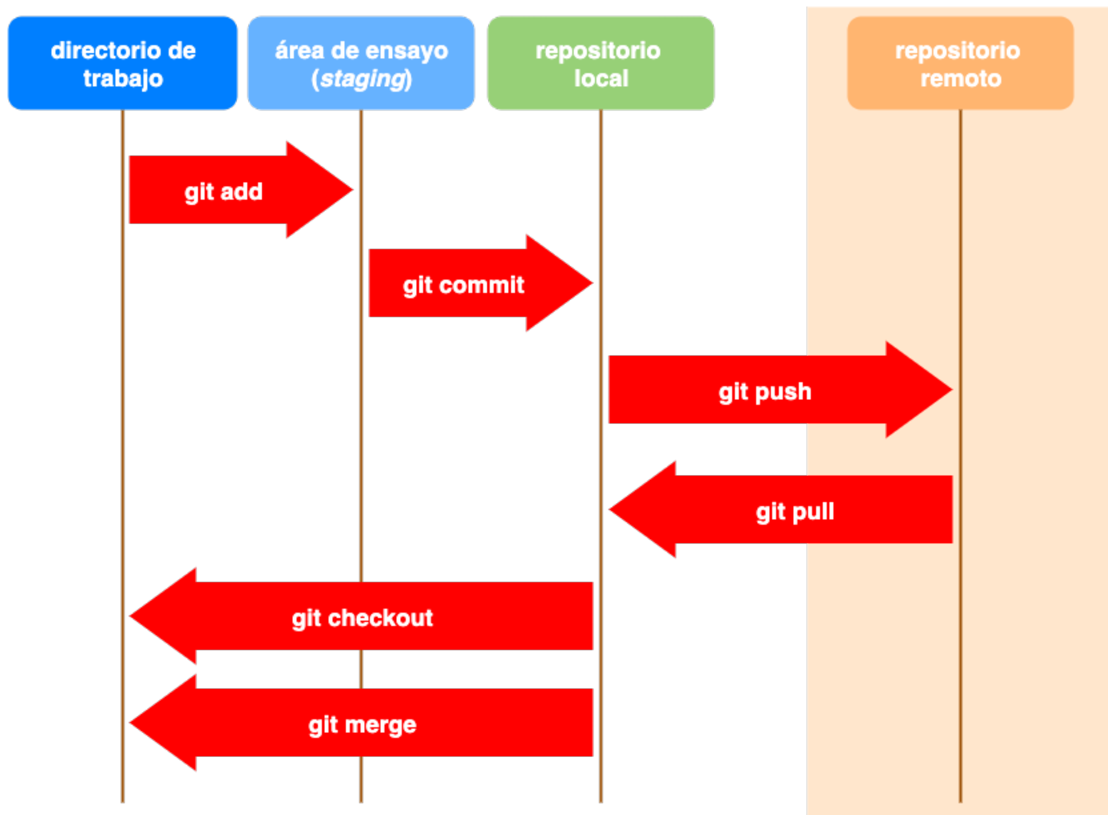

Figura 10. Representación esquemática del funcionamiento de Git

GitLab: es una herramienta que brinda soporte para el ciclo de vida de la práctica de DevOps [32], que está basada en la web y que provee un gestor de repositorio de código Git (ver Figura 11), brindando además herramientas de administración de bases de conocimiento (wikis), un sistema de seguimiento de incidentes o problemas de soporte, y soporte para la configuración de flujos de trabajo de CI/CD. De todas las herramientas que brinda GitLab, las más utilizadas en el desarrollo de los microservicios y demás componentes de la aplicación son aquellas relacionadas a la gestión del repositorio del código y las vinculadas al soporte para la integración continua.

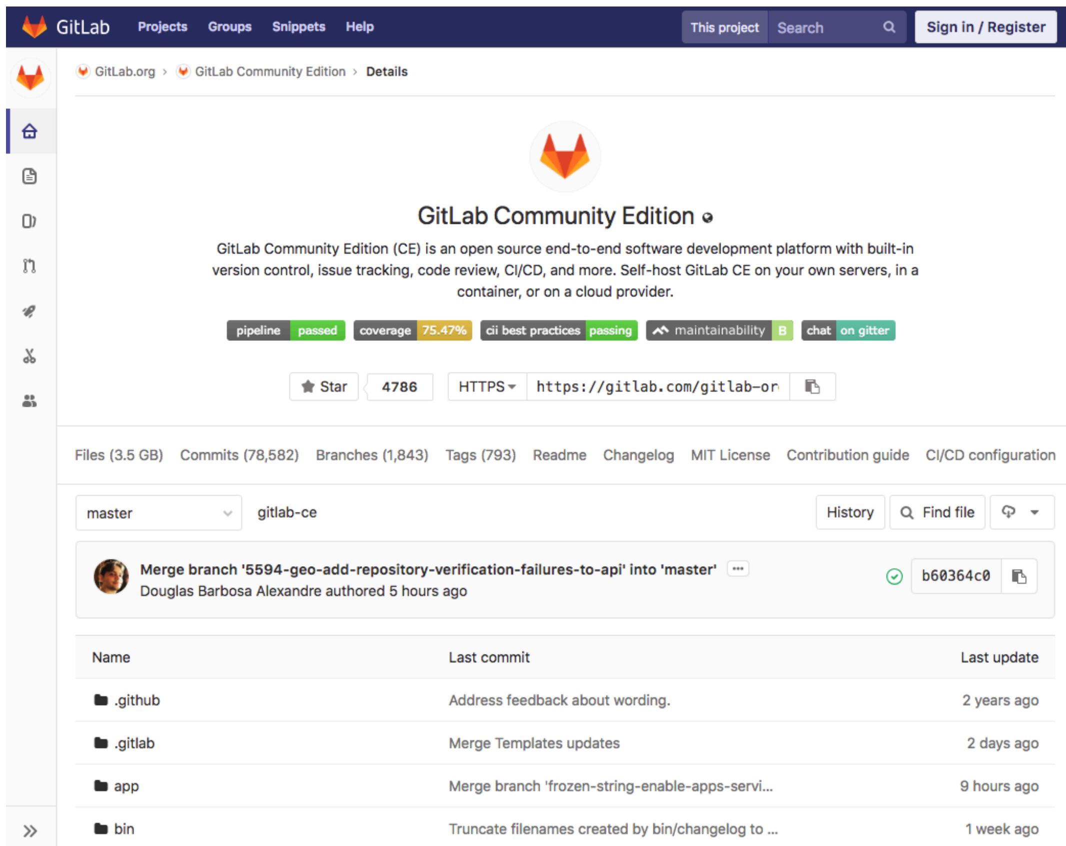

Figura 11. Sitio web GitLab: gestor de repositorio de código

Java: Java es un lenguaje de programación orientado a objetos, multihilos (puede correr varios procesos de manera simultánea) y multiplataforma. En particular, se utiliza la versión 1.8, y como aspectos destacados de esta versión se pueden mencionar el soporte para la programación funcional mediante expresiones Lambda y la mejora en la integración del lenguaje JavaScript. [33]

Gradle: es una herramienta que ayuda a automatizar el proceso de creación de una aplicación. Dicho proceso puede incluir tareas como descarga de dependencias de código, compilar el código fuente en código ejecutable, empaquetar el código ejecutable, correr diferentes pruebas, etc. En grandes proyectos, herramientas como estas son necesarias para poder llevar el control de qué es lo que se necesita construir, en qué secuencia y con qué dependencias, y para poder realizarlo de manera fácil y rápida. [34]

Spring Boot: es una herramienta que permite un desarrollo rápido de aplicaciones basadas en Spring [35], haciendo que el desarrollo sea más accesible. Spring por su parte, es un framework de aplicaciones para desarrollar en el lenguaje de programación Java. Spring es una herramienta muy poderosa y a la vez compleja, por lo que muchas veces resulta difícil de entender. Spring Boot sirve para facilitar el proceso de aprendizaje y utilización de Spring, resultando como una capa intermedia entre el usuario y el framework Spring, y que permite comenzar a utilizar esta última herramienta más compleja de manera más sencilla. No se requieren de habilidades avanzadas de programación en Java para crear y ejecutar rápidamente un microservicio Spring Boot. Es fácil de aprender y puede utilizarse para crear aplicaciones robustas y de orden de producción. Spring Boot y Spring Cloud proporcionan muchas herramientas que facilitan el desarrollo de un microservicio basado en

la nube. La mayoría de las funciones se pueden habilitar con solo unas pocas anotaciones (forma de añadir metadatos al código fuente), lo que agiliza el desarrollo.

Spring Cloud: es un conjunto de herramientas que permite a los desarrolladores construir rápidamente algunos de los patrones comunes en sistemas distribuidos (por ejemplo, gestión de configuración, descubrimiento de servicios, enrutamiento inteligente, micro-proxy, tokens únicos, sesiones distribuidas). [36]

AWS CLI: La interfaz de línea de comandos (CLI) es una herramienta unificada para administrar los productos de AWS. Permite controlar varios servicios de AWS desde la línea de comando y automatizarlos mediante secuencias de comandos [37].

Nginx: es un servidor proxy HTTP e inverso, un servidor proxy de correo y un servidor proxy TCP/UDP genérico. Nginx utiliza una arquitectura asíncrona controlada por eventos para manejar una cantidad masiva de conexiones (más de 10.000 conexiones simultáneas de clientes en un solo servidor) [38].

Softlayer (ahora IBM Cloud): Es un servicio que provee servidores dedicados, hosting gestionado y computación en la nube. Algunos de los componentes de la aplicación aún están desplegados en Softlayer [39].

Jenkins: es un servidor de automatización [40]. Es una herramienta que facilita el trabajo en el proceso de desarrollo de software automatizando aquellas tareas que pueden prescindir de la supervisión humana y facilitando aspectos técnicos para la entrega de software continua. Es posible integrar otras aplicaciones y configurar un flujo de trabajo de manera de que el software generado resulte de una serie de pasos según la configuración elegida. Generalmente el uso principal que se le asigna a esta herramienta es el de construir proyectos de software y someterlos a pruebas de manera continua, así como también monitorear ejecuciones de trabajos ejecutados externamente (ver Figura 12).

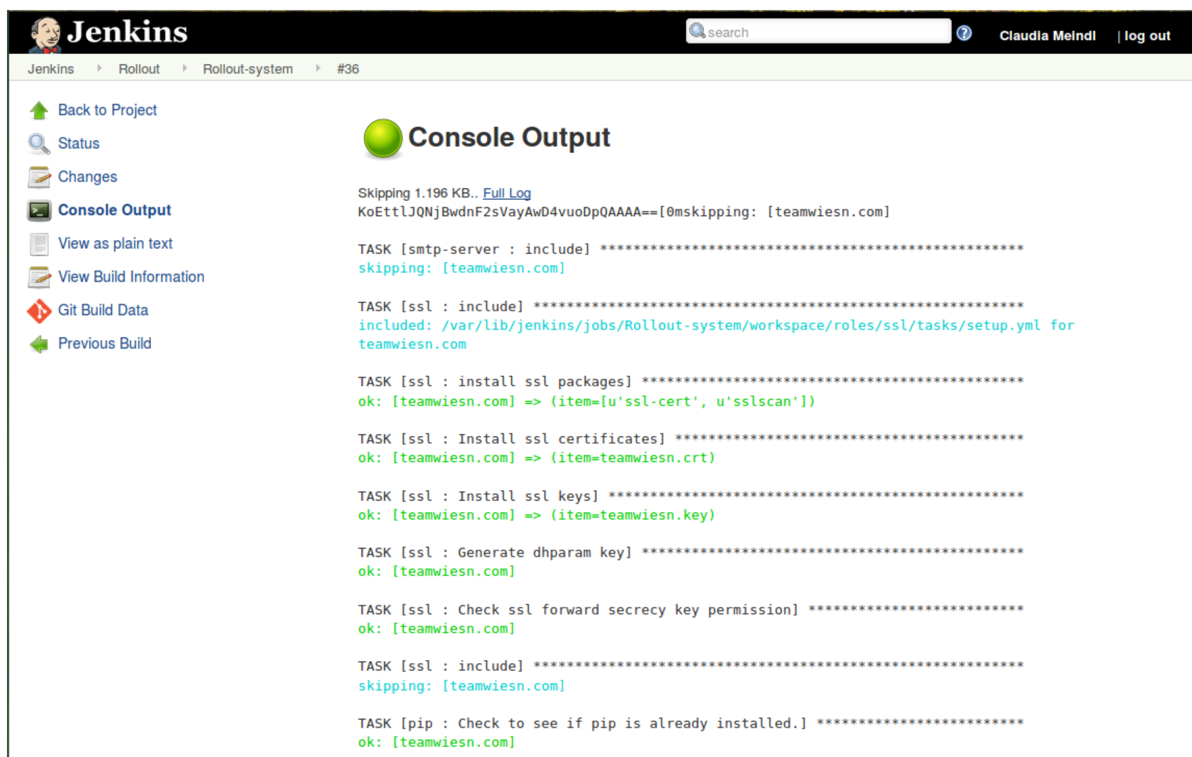

Figura 12. Ejemplo consola de salida de Jenkins

SonarQube: es una herramienta que permite inspeccionar el código fuente para obtener métricas y reportes de calidad [41]. Esto puede ser utilizado para la mejora continua de la calidad del software. Permite detectar problemas y vulnerabilidades, y puede ser integrado con el flujo de trabajo de desarrollo para habilitar una inspección de manera continua. A través de su sitio web, es posible visualizar la información de reportes de manera más sencilla (ver Figura 13).

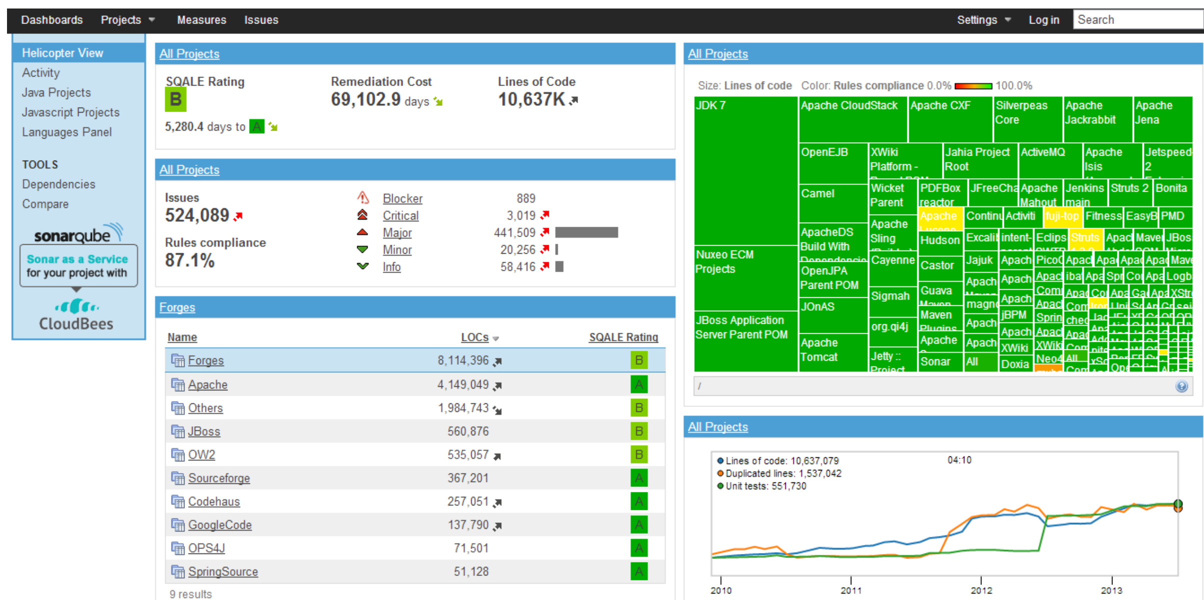

Figura 13. Tablero principal de SonarQube

Twilio: es una plataforma de comunicaciones en la nube. A través de sus servicios web permite desarrollar software para hacer y recibir llamadas telefónicas, enviar y recibir mensajes de texto, y desarrollar otras funciones de comunicación [10].

Twilio Proxy: es una API que permite la comunicación entre dos partes manteniendo privada la información personal de cada una de ellas [42]. Twilio Proxy permite enmascarar comunicaciones proporcionando automáticamente un número de teléfono al que asocia los números de los participantes de la comunicación para reenviar mensajes y llamadas de un lado al otro (ver Figura 14). La funcionalidad brindada por esta API, junto a la de otras herramientas de gestión de la plataforma Twilio, fueron las que se utilizaron para cubrir las necesidades básicas del requerimiento que llevó a la creación del microservicio del presente trabajo.

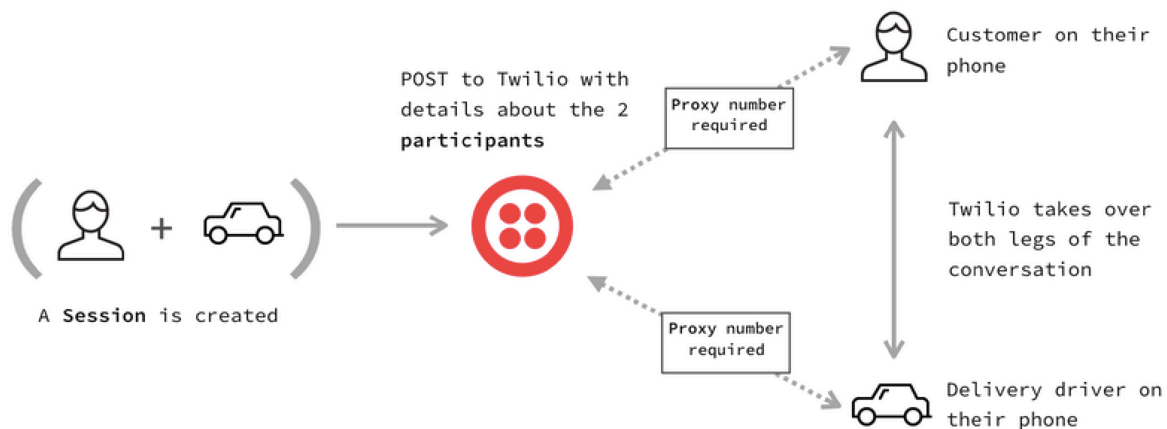

Figura 14. Flujo de interacción en sesión de comunicación de Twilio Proxy [42]

3.4.1.1 Componentes de software utilizados

De las tecnologías descritas anteriormente, aquellas que más se relacionan con la creación de microservicios son Spring Boot, el servidor y cliente Eureka, y las herramientas proporcionadas por AWS CLI. Adicionalmente, y dentro del espectro de librerías de software, se utiliza otro conjunto de herramientas, las cuales están agrupadas en un componente *Commons*, y que representan aquellas dependencias de software que son comunes a todos los microservicios que se desarrollan (ver Figura 15).

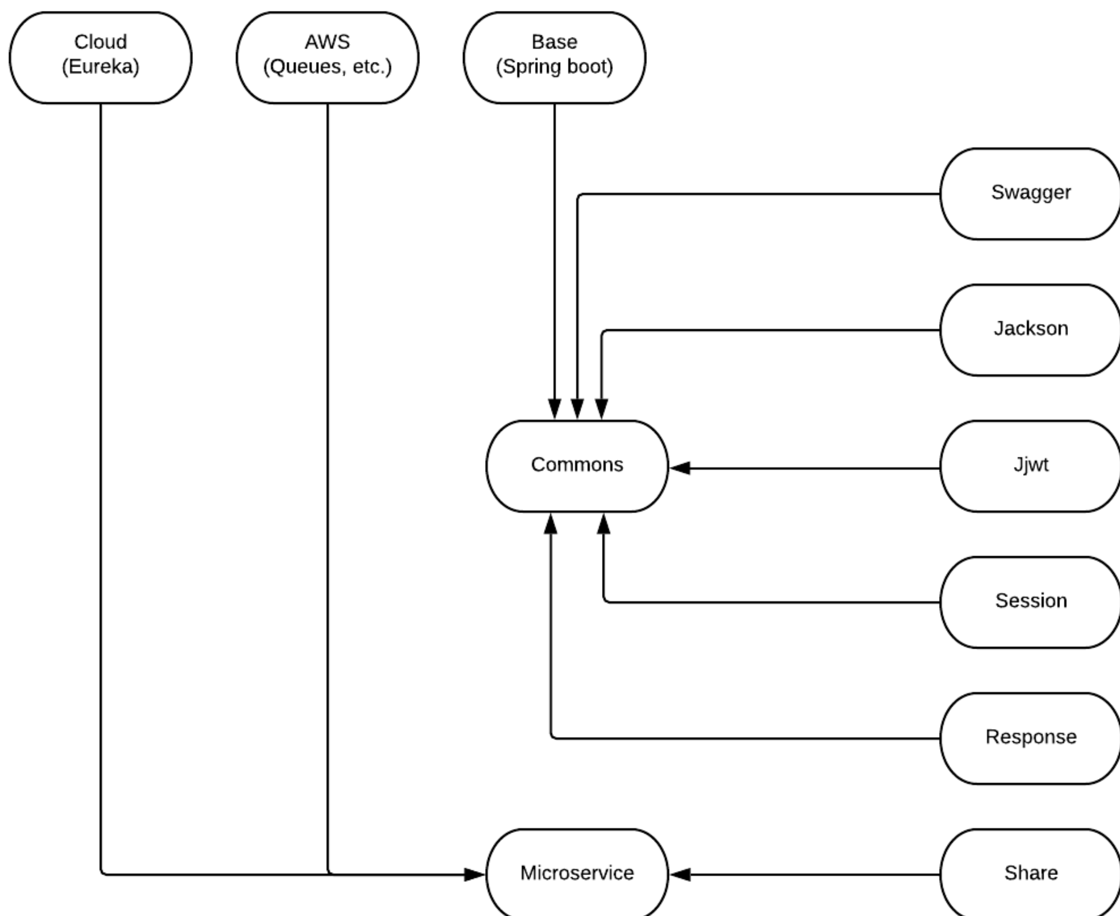

Figura 15. Estructura de componentes utilizados en microservicios

Cada uno de estos componentes está organizado de manera que puede ser importado como una dependencia en el proyecto que se está desarrollando. Cuando se crea un nuevo microservicio, se importan los componentes requeridos según la necesidad.

Una característica particular de estos componentes es que, como decisión de diseño de los microservicios, se decidió crear un mapa interno de algunos de ellos, con la finalidad de "envolver" los mismos a fin de mantener versiones aisladas y crear un comportamiento común para todos los microservicios:

- **Base** (com.leavecar.microservices.base): envuelve Spring Boot;
- **Cloud** (com.leavecar.microservices.cloud): envuelve Eureka;
- **AWS** (com.leavecar.microservices.aws): envuelve AWS Client;

- **Commons** (com.leavecar.microservices.commons): envuelve el conjunto de tecnologías encargado de servir información de configuración y comportamiento común a todos los microservicios.

Esta técnica se corresponde con el patrón de diseño *Adaptador* [43]. Un patrón de diseño es una solución reutilizable general para un problema que ocurre comúnmente en un contexto dado en el diseño de software. El patrón de diseño *Adaptador* es parte de los patrones estructurales y permite que la interfaz de un componente pueda ser utilizada como otra interfaz. La adaptación de la interfaz permite incorporar a los componentes en sistemas existentes que pueden esperar diferentes interfaces del mismo.

A continuación, se brinda un detalle de las tecnologías y los componentes relacionados a librerías de software no descriptos anteriormente:

- **Swagger**: es un conjunto de especificaciones y herramientas sobre cómo describir semánticamente las API. Tiene por objetivo ayudar a desarrolladores a diseñar, construir, documentar y consumir servicios web cubriendo los patrones de diseño más comunes del estilo REST [44] [45]

- **Jackson**: es una librería del lenguaje Java que permite convertir código Java (clases) a texto JSON [46] y viceversa [47].

- **Jjwt**: (Java JSON Web Token) es la implementación en el lenguaje Java del estándar Jwt [48], utilizado comúnmente para transmitir información de manera compacta y segura entre dos servicios. Esta información puede ser verificada y brinda confianza ya que la misma está firmada digitalmente. Debido al tamaño pequeño del token, el mismo puede enviarse a través de una URL, un parámetro POST, o dentro del encabezado http.
  La información útil del token tiene todo lo requerido para evitar llamar más de una vez a la base de datos. Generalmente, la información que se transmite es la referida a la autenticación de un usuario: una vez que el usuario accede al sistema, cada petición siguiente incluirá el token, permitiéndole al usuario acceder a recursos que son permitidos con ese token [49].
  Jjwt, junto con Redis, forma parte de la capa de autentificación del sistema. Esto permite proteger el acceso a las APIs, ya que las mismas solo pueden ser accedidas por usuarios que hayan iniciado sesión en el sistema, y los resultados retornados serán basados en el tipo de usuario que son. En la Figura 16 se puede observar que el flujo de autentificación básicamente es el siguiente:
    - El usuario envía una petición al servidor para obtener un token pasando la información de sus credenciales.
    - El servidor valida las credenciales y en caso de que estas sean correctas, envía en el mensaje de respuesta el token de autentificación.
    - Con cada petición al servidor, el usuario debe proveer el token, y el servidor ahora validará la información de autentificación relativa a este token.

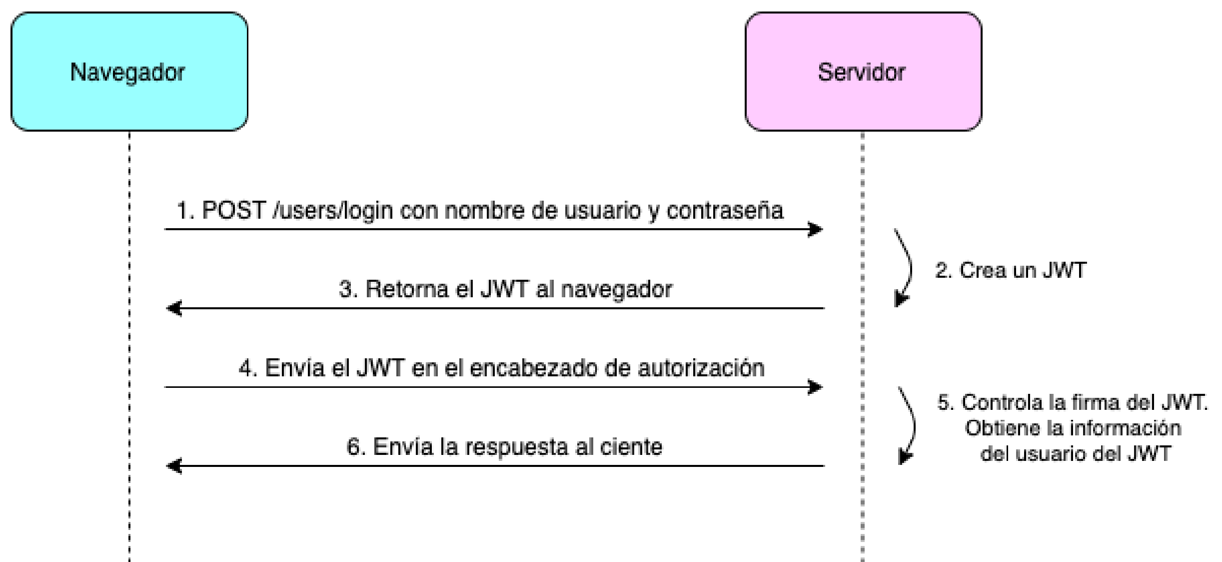

Figura 16. Ejemplo de proceso de autentificación utilizando JWT

- **Session**, **Response**, **Share**: son componentes en los que también se encapsulan las implementaciones necesarias para el manejo de sesiones de usuario, para la forma de las respuestas de los microservicios, y para estructuras de datos compartidas, respectivamente.

3.4.2 Metodología de trabajo

Se trabajó siguiendo la forma de trabajo ágil de Scrum [50]. El continuo cambio tecnológico y el dinamismo que este conlleva, junto con la siempre presente incertidumbre generada por la falta de predicción del comportamiento humano, hacen necesario la utilización de un marco de trabajo que permita adaptarse rápidamente a las demandas de la industria. Scrum es una de las formas ágiles de trabajo más populares y que mejor se adapta a la rápida e imparable transformación tecnológica.

La organización del trabajo con Scrum consiste en una forma de gestión de proyectos donde se establece un periodo de tiempo fijo y recurrente, denominado Sprint, y que generalmente dura entre una y cuatro semanas. Durante estos periodos de tiempo, la idea es desarrollar funcionalidades que aporten valor a los usuarios finales, haciendo especial énfasis en la revisión del trabajo, para ayudar al equipo de desarrollo a moverse de manera eficiente hacia la meta establecida. La Figura 17 muestra de manera simplificada la organización del trabajo siguiendo la metodología Scrum.

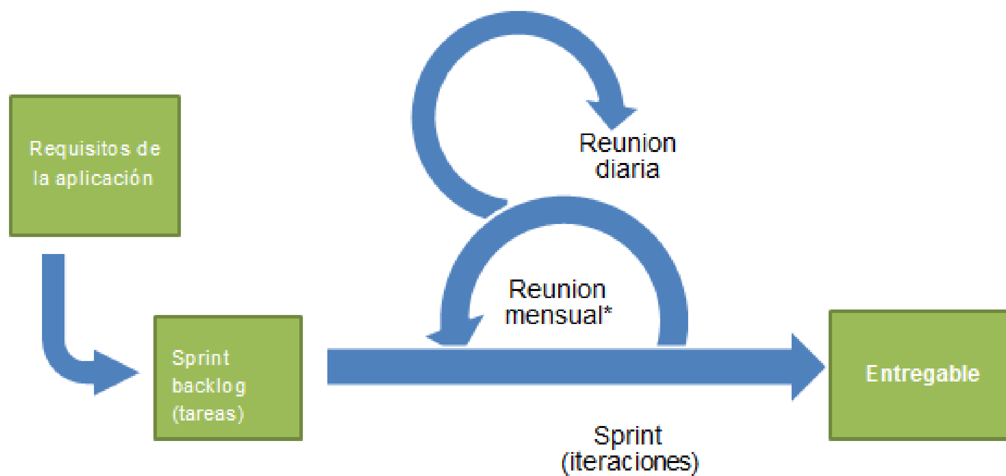

Figura 17. Marco de trabajo SCRUM [50]

Un típico equipo de Scrum consta de entre 3 y 9 personas, donde cada una desempeña un rol y sigue determinadas reglas.

Una de las claves para conseguir la agilidad en el trabajo es la buena comunicación entre los miembros del equipo, en donde es necesario informar sobre el progreso y los impedimentos que pudieran existir. El equipo trabaja de manera colaborativa con el objetivo de remover cualquier obstáculo que pueda impedir concluir con los trabajos comprometidos.

3.4.3 Organización del equipo de desarrollo

En cuanto al equipo de trabajo, ya se contaba con estructuras de pequeños equipos multifuncionales. Tal y como se viene recomendando por la práctica DevOps [51], el proyecto general es desarrollado por varios equipos de personas que desarrollan varias funciones (ver Figura 18), lo cual se adapta bien al enfoque de microservicios. Cada equipo es responsable de uno o varios servicios y contiene personas con diferentes habilidades, tales como habilidades de desarrollo y operaciones. Los miembros del equipo cooperan desde el inicio del proyecto para crear más valor para los usuarios finales.

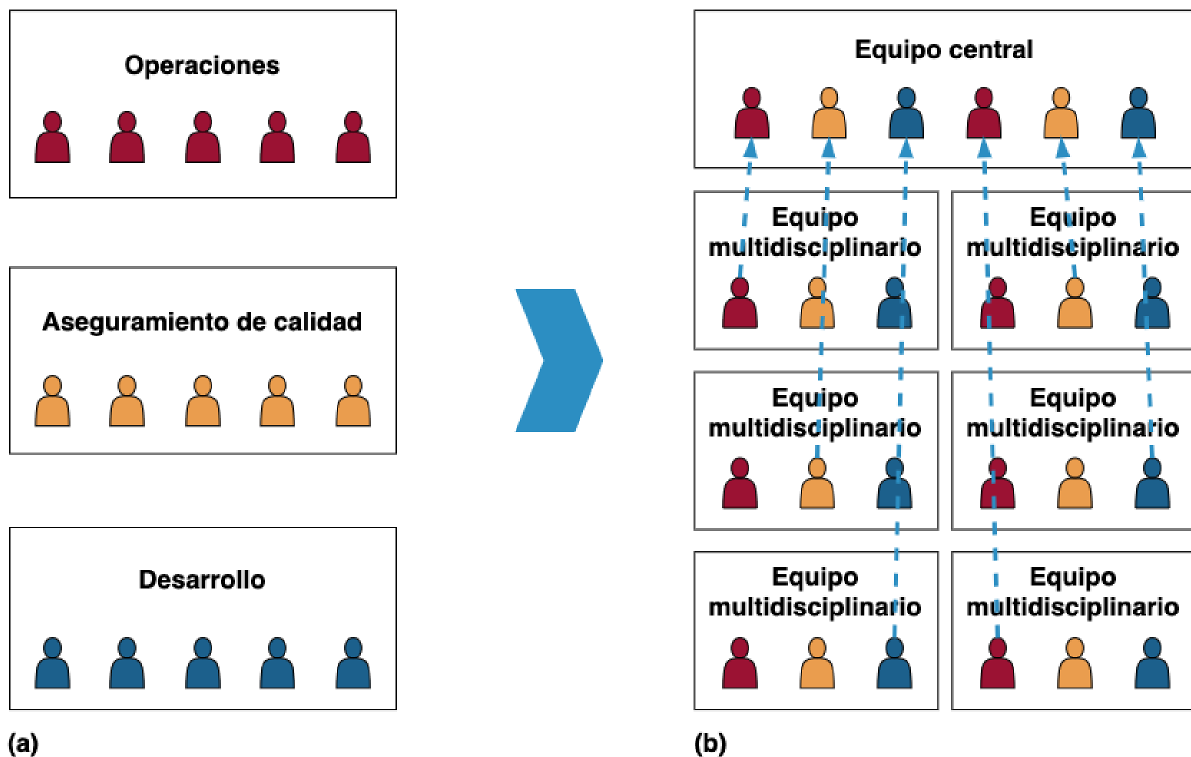

Figura 18. Formación de equipo de trabajo: (a) Equipos horizontales tradicionales, (b) Equipos verticales recomendados por DevOps [52]

3.4.4 Especificación de requerimientos

Generalmente, todo el equipo de trabajo participa en el proceso de especificación de requerimientos, así como también en la estimación de la complejidad de las tareas necesarias para el desarrollo de la funcionalidad. Esto permite, por un lado, lograr un mejor refinamiento de la descripción de las funciones a desarrollar, y, por otro lado, que se produzca una transferencia de conocimiento por parte de aquellos integrantes del proyecto que poseen mayor experiencia. Al discutir los diferentes detalles que el requerimiento involucrará, los miembros del equipo se nutren con conocimiento acerca del sistema y otras cuestiones tecnológicas que tal vez de manera individual no podrían haber surgido. Al participar todo el equipo, se obtienen mejores puntos de vista, se comparte conocimiento, y se cubren más detalles.

En cuanto al presente trabajo, los requerimientos consistían en permitir y registrar comunicaciones entre un usuario y un operador del sistema, y luego poder acceder a esos registros. Dicha funcionalidad era requerida tanto para llamadas telefónicas como para mensajes de texto SMS.

Debía permitirse la comunicación bidireccional y la misma debía realizarse enmascarando las líneas personales de quienes se estaban comunicando para proteger la privacidad de los participantes.

A partir de esta solicitud, se debió especificar de manera precisa las capacidades de negocio que se implementarían. Es decir, fue necesario describir de manera detallada las funcionalidades a implementar que finalmente concluirían aportando valor a los usuarios finales.

Con el caso de uso general definido (gestión de comunicación entre usuario y operador), se detallaron funciones específicas para los usuarios, y posteriormente se definieron un conjunto de sub-tareas que describían las labores de desarrollo de software que debían realizarse. Algunas de las tareas fueron:

- Definición de la arquitectura para la solución: si bien ya se contaba con una plantilla de una arquitectura de microservicios definida con antelación, debía revisarse si la misma serviría para implementar la funcionalidad solicitada y si sería necesario la utilización de dependencias de software adicionales para cubrir las necesidades de los requerimientos.
- Definiciones de implementación de los servicios o métodos necesarios en el microservicio: estas tareas requirieron de la especificación de implementación para las funciones encargadas del manejo de llamadas telefónicas, el manejo de mensajes de texto, la recuperación del historial de llamadas telefónicas, y la recuperación del historial de mensajes de texto.
- Especificación de la integración del microservicio con la aplicación principal ya existente: aquí debieron especificarse en qué puntos y bajo qué condiciones se terminaría consumiendo las funciones del microservicio. Se detallaron cuestiones relacionadas a lo que sería la sesión de comunicación de un usuario, como por ejemplo cuándo la misma sería creada, modificada o eliminada, qué tiempo de vida tendría, y otras reglas relacionadas al negocio y casos particulares que se deberían tener en cuenta.

3.4.4.1 Definición de casos de uso adicionales

Adicionalmente, se definieron otros escenarios de casos de uso para completar el refinamiento de requerimientos. Según Wikipedia [53], un caso de uso es la descripción de una acción o actividad. Describir todos los posibles escenarios que pudieran ocurrir a partir de la funcionalidad requerida es una actividad que permite descubrir aspectos que tal vez no fueron tenidos en cuenta al momento de especificar tareas concretas para desarrollar la funcionalidad.

Parte del proceso de descripción de la acción involucra tareas como: Identificar los actores que intervienen, sus roles, los permisos y condiciones con los que pueden realizar otras acciones relacionadas, especificar información de contexto del sistema para la acción particular que se está describiendo, identificar qué otros sucesos ocurren a partir de determinadas interacciones y cómo se responde o cómo se manejan estos eventos.

Por lo general, se define un escenario donde todo ocurre de la mejor manera esperada según lo requerido. Este escenario es conocido como “camino feliz”, y es la descripción que mayor valor aporta como documentación para el desarrollo y posteriores pruebas, ya que consta de todos los pasos y condiciones para que la funcionalidad pueda reproducirse de manera correcta.

A partir de este “camino feliz”, una práctica que ayuda a crear nuevos escenarios es introducir cambios de condiciones en cada paso de la descripción, para luego analizar cómo debe transcurrir la acción a partir de estos nuevos estados.

Si bien esta descripción puede ir acompañada por un diagrama, a fin de complementar la documentación y el entendimiento de los escenarios, tal práctica no fue realizada en esta instancia.

3.4.4.2 Investigación y pruebas de factibilidad de API de comunicaciones utilizada en el desarrollo

Esta etapa comprendió tareas de investigación de la tecnología base que se utilizaría para poder llevar adelante los requerimientos.

El equipo de desarrollo ya estaba familiarizado con las soluciones tecnológicas de la plataforma de comunicaciones Twilio, debido a que la misma ya había sido utilizada anteriormente en otra funcionalidad. Y, si bien se acudió directamente a las soluciones brindadas por la plataforma, se debió investigar qué tipo de soporte brindaba específicamente para la gestión de llamadas telefónicas y mensajes de texto.

Se indagó a partir de la documentación disponible en el sitio web de la plataforma cuáles serían las opciones de implementación más adecuadas, y luego, se consultaron dudas puntuales al equipo de soporte de la plataforma Twilio.

Luego del correspondiente análisis de la información recabada en la etapa de investigación, se realizaron algunas pequeñas pruebas técnicas para verificar que con la utilización de la tecnología no surgieran aspectos bloqueantes que impidieran el desarrollo de la funcionalidad. Estas pruebas básicamente consistieron en crear sesiones de comunicación de prueba en la plataforma Twilio, a través de su consola de administración y de manera manual (ver Figura 19), y luego realizar comunicaciones utilizando los datos creados en la sesión.

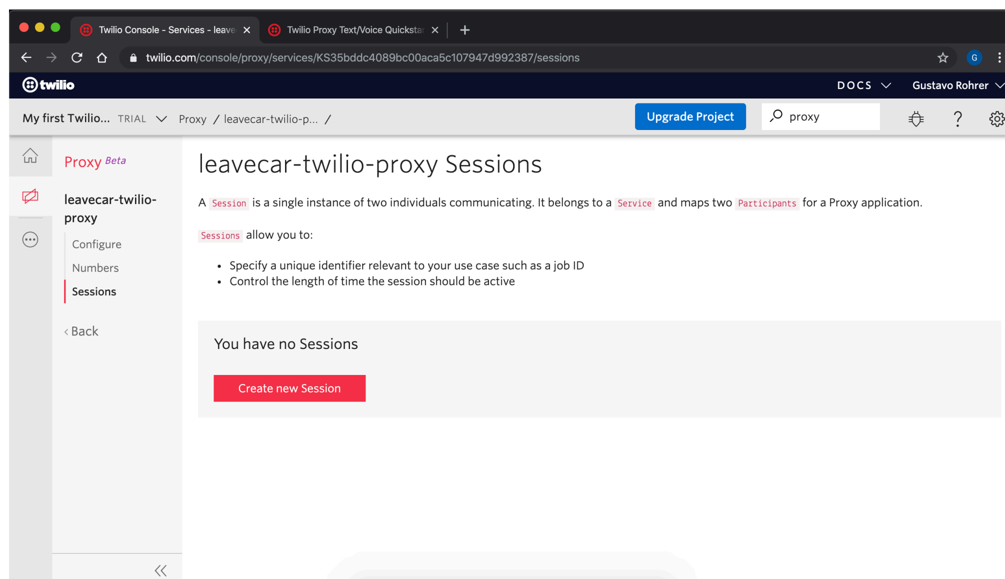

Figura 19. Consola de administración de plataforma Twilio: creación de sesión de comunicación de prueba

3.4.5 Desarrollo del nuevo microservicio

3.4.5.1 Elección de las tecnologías a utilizar

El microservicio fue desarrollado utilizando las tecnologías descritas anteriormente, principalmente por cuestiones de priorización de tiempos y recursos. Ya se habían creado otros microservicios anteriormente, con lo cual ya se contaba con la documentación, el

soporte y el marco de conocimiento necesarios como para poder realizar las tareas de desarrollo contando con una base sólida de técnicas y patrones que asegurarían un proceso de desarrollo ágil y menos propenso a errores.

En este punto, y tal cual se expresa en [52], es importante señalar la relevancia de la documentación de procesos y tecnologías, lo cual facilita una mejor accesibilidad a experiencias y buenas prácticas pasadas, y a su vez la posibilidad que las mismas sean reutilizadas de manera eficiente por otros desarrolladores.

3.4.5.2 Gestión del código fuente

A continuación, se describe de manera breve la forma en que se organiza y gestiona en general el código fuente. Este mismo esquema fue el que se utilizó para trabajar en la creación del nuevo microservicio.

3.4.5.2.1 Control de versiones y repositorio de código

Para poder gestionar el historial de cambios en el código fuente se utiliza la herramienta Git [31]. La mayoría de los proyectos de software de cierta envergadura involucran a varios programadores trabajando en paralelo, razón por la cual esta herramienta es imprescindible a fin de evitar conflictos. Además, los requerimientos del proyecto suelen cambiar, con lo cual a menudo también es necesario volver a versiones antiguas del código.

En parte, Git permite que los equipos de trabajo contribuyan de manera eficiente y efectiva, y de forma asíncrona, lo cual facilita la colaboración y permite resolver problemas más grandes y complejos.

3.4.5.2.2 Ramas de desarrollo de software (branches) y flujo de trabajo

De manera simple, una rama de desarrollo es una copia del código fuente de la versión productiva del sistema (o *rama principal*). Esto permite, entre otras cosas, organizar el trabajo de los desarrolladores con la finalidad de que éstos puedan desarrollar funcionalidades de manera paralela (en otras *ramas*) sin afectar el trabajo de otros integrantes del equipo. Luego, cuando se finaliza el desarrollo de cada nueva característica, se realiza un proceso de unificación (o *merge*) al código productivo o principal.

El modelo de *branching* (ramificación o bifurcación) del código utilizado es el siguiente:

- **Rama maestra (*master*)**: representa el código productivo, es decir, aquel código fuente de la aplicación que está siendo utilizada actualmente por los usuarios finales. Esta rama de código solo recibe *merges* (uniones o fusiones) de código solo desde una **rama de lanzamiento (*release*)**, o de una rama creada para hacer una corrección urgente en la rama productiva (**rama de revisión o *hotfix***).

- **Rama de desarrollo (*develop*)**: esta rama es una copia de la rama ***master***, y contiene además el código de aquellas nuevas funcionalidades que van a estar disponibles en la próxima entrega al cliente. Funciona como una rama de integración para las nuevas características desarrolladas. Esta rama solo recibe *merges* de aquellas ramas creadas a partir de ***develop*** y que contienen código relacionado a nuevas funcionalidades a entregar. Aunque también puede recibir código proveniente de un *hotfix* realizado en el ambiente de producción.

- **Rama para nueva característica (*feature branch*)**: cada nueva característica reside en su propia rama, pero, en lugar de ramificarse de la rama ***master***, se usa ***develop*** como

la rama principal. Cuando se completa una característica, se vuelve a fusionar a ***develop***. Las nuevas características nunca se fusionan directamente con *master*.

- **Rama de lanzamiento (*release*)**: una vez que la rama ***develop*** ha adquirido suficientes funciones para un lanzamiento a producción, se bifurca una rama ***release*** a partir de la rama ***develop***. La creación de esta rama inicia el próximo ciclo de lanzamiento a producción, por lo que no se pueden agregar nuevas características después de este punto, solo las correcciones de errores, la generación de documentación y otras tareas orientadas a la versión deben ir en esta rama . El uso de una rama dedicada para preparar versiones hace posible que un equipo pueda pulir la versión actual mientras que otro equipo continúa trabajando en las características para la próxima versión. También crea fases de desarrollo bien definidas.

- **Rama de revisión (o corrección, o mantenimiento en producción)**: las ramas de mantenimiento o "***hotfix***" se utilizan para corregir rápidamente las versiones de producción. Estas ramas se parecen mucho a las de ***release*** y ***feature***, excepto que están basadas en ***master*** en lugar de ***develop***. Esta es la única rama que debe bifurcarse directamente de *master*. Tan pronto como se complete la corrección, debe fusionarse tanto en *master* como en *develop* (y/o en la rama *de release* actual), y *master* debe etiquetarse con un número de versión actualizado.
  Tener una rama de desarrollo dedicada para la corrección de errores permite al equipo abordar problemas sin interrumpir el resto del flujo de trabajo o esperar el próximo ciclo de lanzamiento.

La anterior descripción puede verse ilustrada de manera sencilla en la Figura 20, y con mayor detalle en la Figura 21.

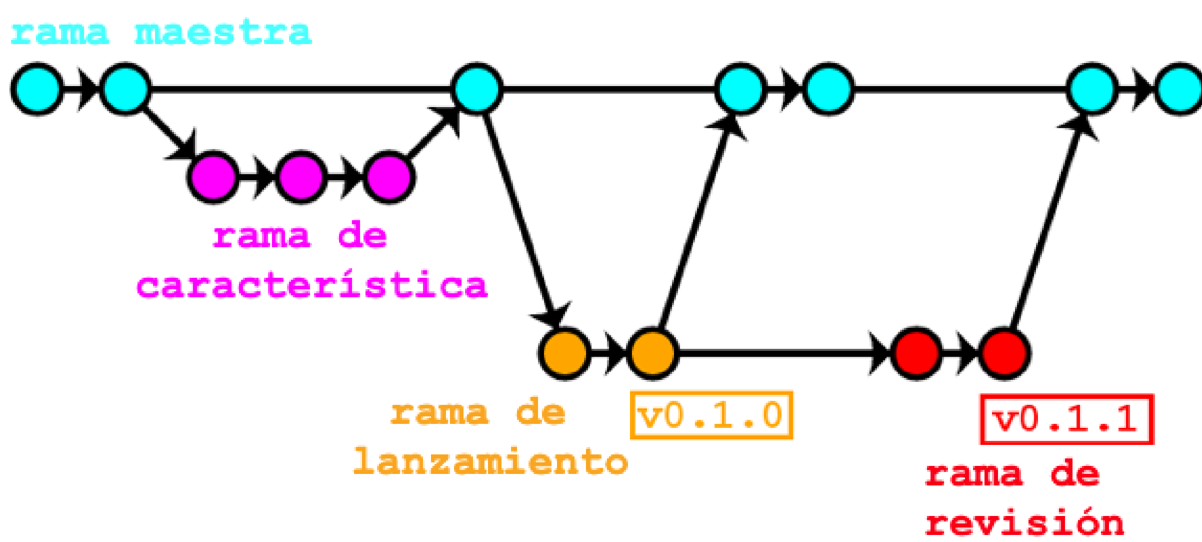

Figura 20. Representación simplificada de ramificación de código en un proyecto [54]

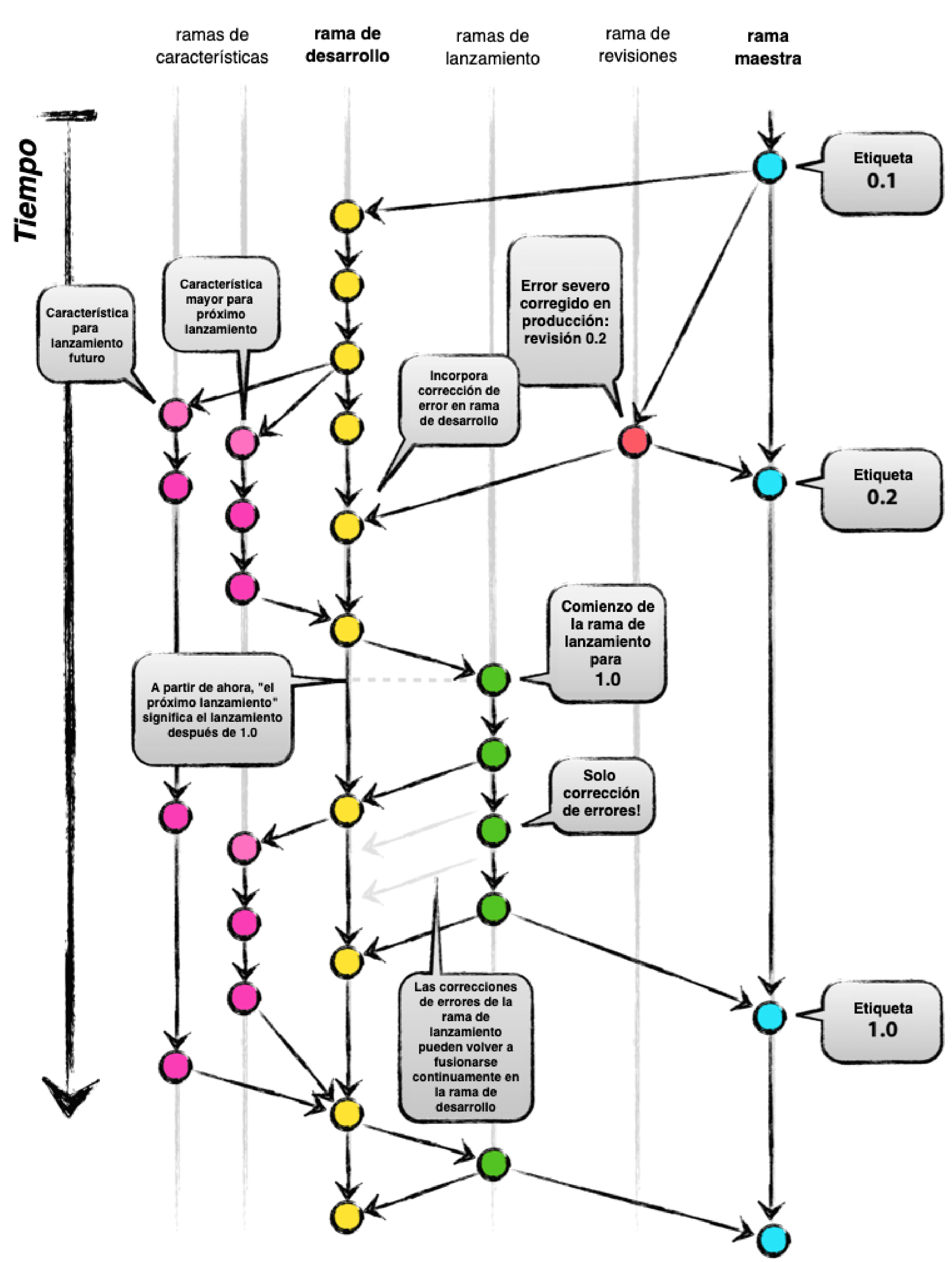

Figura 21. Modelo de estrategia de ramificación de código para el desarrollo de software [55]

A partir del esquema de *branching* utilizado, el flujo de desarrollo se realiza generalmente y a grandes rasgos de la siguiente manera:

1. Al comenzar a desarrollar una nueva característica se utilizan *feature branches* (no se unifica código fuente directamente a la rama *master*).
2. Todas las entregas de código deben ser sometidas a pruebas: se realizan pruebas manuales por parte del desarrollador en el momento del desarrollo (utilizando un ambiente de pruebas o el ambiente en el que se desarrolló la funcionalidad). Antes de enviar código al repositorio, deben correrse pruebas de código unitarias y pruebas de integración de funcionalidad (en el caso que las mismas existieran): las *pruebas unitarias* son rápidas y controlan el correcto funcionamiento del código de manera aislada, mientras que las *pruebas de integración* son más costosas en tiempo y permiten verificar cómo la nueva funcionalidad interactúa con el resto del sistema. Las pruebas de integración pueden detectar problemas que podrían haberse pasado por alto en las pruebas unitarias.
   Luego, un miembro del equipo de QA (del inglés *Quality Assurance*, Aseguramiento de calidad) asignado al equipo (y con ayuda de los desarrolladores) realiza otras pruebas manuales y da su aprobación para la entrega del código.
3. Antes que el nuevo código sea integrado, se realizan revisiones del mismo (*code reviews*) por parte del resto del equipo de desarrollo. Se realiza un examen sistemático del código con el objetivo de encontrar errores pasados por alto durante el desarrollo, y con la finalidad de mejorar la calidad general del software. Para esto se sigue un proceso conocido como *merge request* o *pull request* [56], que como su significado en inglés lo indica, representa una solicitud de "unificación" de una rama de desarrollo dentro de otra. Es durante este proceso en donde se obtiene retroalimentación (por parte del equipo de trabajo y por procesos automatizados de reportes de calidad) acerca de posibles optimizaciones y correcciones en el código, las cuales deben realizarse si se quiere que la solicitud de unificación sea aprobada.
4. En este punto, cuando el código pasa la revisión y es unificado a una rama integradora, se corren otras series de pruebas automatizadas. Entre estas pruebas, se encuentran las de regresión, las cuales sirven para confirmar que el nuevo código no tiene efectos negativos en las funcionalidades ya existentes.
5. Una vez que el código correspondiente a una funcionalidad cumple con los criterios de aprobación definidos y se obtiene la aceptación del equipo de QA, el mismo es integrado a la rama principal y es etiquetado con un número de versión.

Al contar con un flujo de trabajo bien definido y claro, es posible facilitar las tareas de entrega continua e integración continua, ya que partes del proceso pueden automatizarse de manera más sencilla.

3.4.5.3 Ambientes de desarrollo

Se cuenta con una serie de ambientes en AWS administrados por el equipo de desarrollo, los cuales están destinados para la integración y prueba del nuevo código. Cada ambiente se encuentra configurado con un conjunto de hardware, datos y configuraciones relativas a la finalidad de las pruebas que allí se realizan, y que generalmente son una réplica de la información del ambiente productivo. El objetivo de estos ambientes es el de habilitar un entorno en el cual se puedan probar las nuevas funcionalidades y cambios sin afectar la aplicación principal.

Ambiente DEV: es un ambiente de integración manual. Los desarrolladores unifican el código relativo al trabajo que están realizando. Las primeras pruebas manuales se realizan en este ambiente. Se prueban funcionalidades grandes a mediano plazo.

Ambiente CI: es el ambiente de integración continua. La herramienta Jenkins deja instalada la versión próxima a salir a producción (es decir aquella que se unificará en la rama principal de código). Aquí corren pruebas automatizadas y también se realizan pruebas manuales de funcionalidades a corto plazo.

Ambiente QA: es un ambiente destinado a las pruebas del equipo de QA de funcionalidades a corto plazo.

3.4.5.4 Desarrollo de funcionalidades

Como se mencionó, el microservicio fue creado utilizando una serie de tecnologías que ya habían sido empleadas en otros desarrollos. Se utilizó una plantilla o “esqueleto” base formado a partir de la creación de otros microservicios y que consiste en una aplicación Java que utiliza Gradle y está basada en Sprint Boot (componente Base), el framework de pruebas Mockito [57] (el mismo se describe brevemente más adelante al detallar las pruebas unitarias realizadas), y los componentes Commons y Cloud.

Se realizó una copia del código fuente del proyecto plantilla, y se inicializó un repositorio de código utilizando las herramientas Git y GitLab.

Luego, los primeros pasos de la etapa de desarrollo consistieron fundamentalmente en configuraciones del proyecto Spring y demás tecnologías empleadas.

Una de las formas de configurar aplicaciones Spring es utilizando archivos de configuración YAML, cuyo formato es un estándar de serialización de datos y resulta fácil de comprender. Una de las ventajas de contar con estos archivos es la posibilidad de crear diferentes perfiles en los cuales es posible definir diferentes propiedades según los ambientes en los que correrá la aplicación. La aplicación toma el primer perfil como el predeterminado, a menos que se declare lo contrario.
El archivo que se corresponde con esta configuración es llamado *application.yml* e inicialmente solo contiene datos sobre el nombre de la aplicación, configuraciones del servidor y la clave asignada a JWT.

```
spring:
  application.name: Communications

server:
  compression:
    enabled: true
    mime-types: application/json
  port: 8002


### TO BE OVERRIDE BY ENV CONFIG ##
jwt:
  secret: *******
```

Archivo *application.yml*

Por otro lado, en el archivo *gradle.properties* se establecen las configuraciones que utilizará la herramienta Gradle, tanto para gestionar las dependencias utilizadas en el proyecto como para brindarle al correspondiente proceso Java la información necesaria para construir la aplicación [58].

```
#################################################################
# project coordinates
#################################################################
group=com.leavecar.microservices
artifact=communications
version=0.1.0

#################################################################
# build dependencies versions
#################################################################

buildScanPluginVersion=1.9
dependencyMangamentVersion=1.0.3.RELEASE

#################################################################
# dependencies versions
#################################################################
springBootVersion=2.0.1.RELEASE
mockitoVersion=2.15.0
microserviceCommons=0.1.0
microserviceCloud=0.1.0
```

Archivo *gradle.properties*

También es necesario configurar el archivo *build.gradle*, el cual es una guía de configuración de compilación. Este archivo se utiliza para describir y manipular la lógica de compilación de la cual se encarga la máquina virtual de Java. El bloque de dependencias configura las dependencias que Gradle necesita usar para construir el proyecto.

```
....

dependencies {

    compile("com.leavecar.microservices:commons:${microserviceCommons}")
    compile("com.leavecar.microservices:cloud:${microserviceCloud}")

    compile('org.springframework.boot:spring-boot-configuration-processor')

    testCompile("org.springframework.boot:spring-boot-starter-test")
    testCompile("org.mockito:mockito-core:${mockitoVersion}")
}

....
```

Fragmento de archivo *build.gradle*

Pasando ya a archivos de código Java, un proyecto Spring Boot posee una clase con la anotación @*SpringBootApplication*. Este es el punto de partida de una aplicación Spring Boot.

```
package com.leavecar.microservices.communications;

import ...

@SpringBootApplication
@EnableDiscoveryClient
public class Main {

    @Value("${jwt.secret}")
    private String jwtSecret;

    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }

    @Bean
    public Jackson2ObjectMapperBuilder objectMapperBuilder() {
        return new DateMapper().getMapper();
    }

    @Bean
    public JWTUtil jwtUtil(){
        return new JWTUtil(jwtSecret);
    }

    @Bean
    public SessionHolder sessionHolder(){
          return new SessionHolder();
    }
}
```

Clase *Main.java*

La anotación *@SpringBootApplication*, es en realidad una combinación de otras anotaciones, *@SpringBootConfiguration*, *@EnableAutoConfiguration* y *@ComponentScan*. Respectivamente, estas anotaciones son utilizadas para: indicar al framework Spring que esta clase actuará como una clase de configuración, para establecer cómo se buscará esa información, y para especificar los lugares en los que buscar otros componentes anotados.

En la clase *Main.java* también se utiliza la anotación *@EnableDiscoveryClient*, la cual le dice a Spring que debe activar la aplicación cliente de la herramienta Eureka. El cliente Eureka, junto con el servidor Eureka, se encargan del registro de servicios, es decir, se ocupan de dar nombres a los servicios y registrar los mismos. Esto es de gran utilidad ya que por un lado no es necesario codificar las direcciones IP de los microservicios y, por otro lado, soluciona el problema de las direcciones IP dinámicas cuando existe escalado automático del servicio (cuando las nuevas instancias de un servicio se activan dinámicamente, los otros servicios que lo utilizan necesitan saber acerca de la disponibilidad adicional). Entonces, cada servicio se registra a sí mismo con Eureka, y se comunica con el servidor Eureka para notificar que está activo. Cuando un cliente requiere un servicio en particular, puede obtener un listado de las instancias del servicio a través de la herramienta de descubrimiento de Eureka, y luego consultar una instancia del servicio (ver Figura 22). Si el servidor Eureka no recibe ninguna notificación de un servicio, ese servicio es quitado del registro automáticamente.

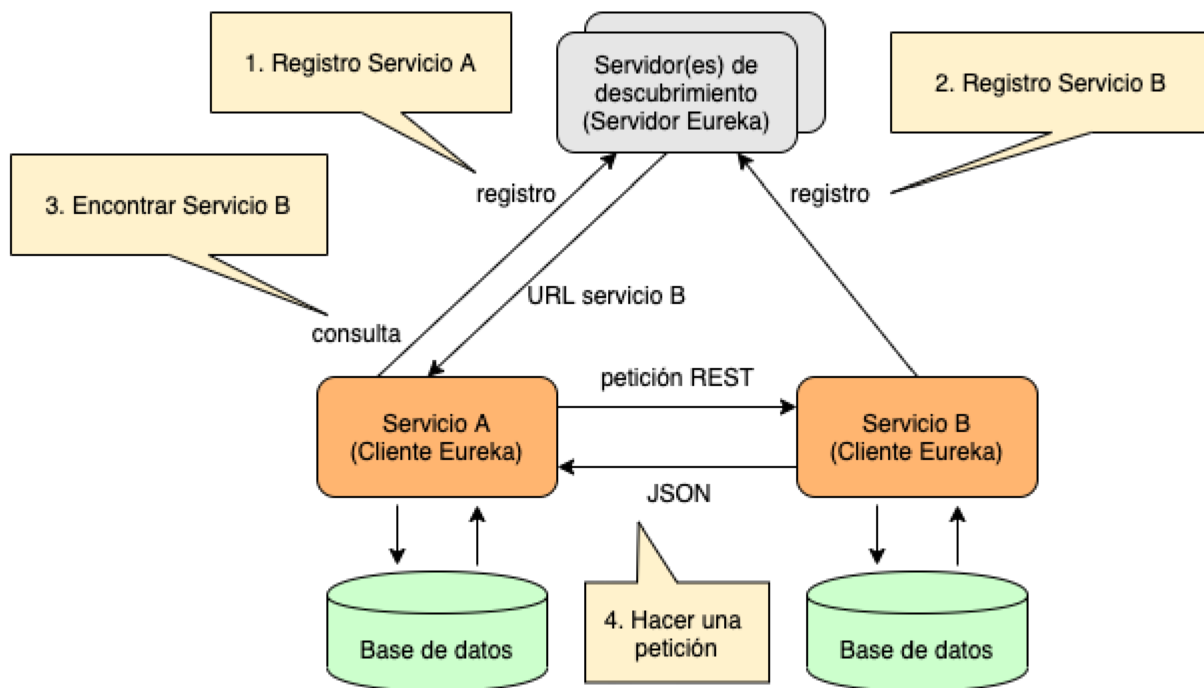

Figura 22. Ejemplo de descubrimiento de servicios

Otro aspecto que puede notarse en la clase *Main.java* es cómo se crean los métodos necesarios para utilizar las herramientas de JWT, Jackson, y para poder gestionar las sesiones de usuarios.

Luego, el resto del desarrollo continuó de la misma manera en la que se desarrollaría una aplicación Spring.

Primeramente, se definió la capa de controladores. La configuración en los archivos pertenecientes a la capa de controladores maneja todas las peticiones HTTP entrantes al servidor y devuelven una respuesta adecuada. Aquí cabe mencionar que la capa de controladores se corresponde con una de las tres partes del patrón arquitectónico de software Modelo-Vista-Controlador, o más conocido como MVC (ver Figura 23). El objetivo principal del patrón MVC es separar las representaciones internas de una aplicación de las formas en que estas son consumidas (generalmente por una interfaz de usuario). Originalmente, MVC fue creado para aplicaciones GUI (interfaz gráfica de usuario, del inglés *Graphical User Interface*), pero rápidamente se convirtió en un patrón popular para diseñar aplicaciones web. El patrón MVC tiene los siguientes componentes:

- Modelo: gestiona datos, lógica y reglas de la aplicación.
- Vista: se usa para presentar datos al usuario.
- Controlador: acepta la entrada del usuario y la convierte en comandos para el Modelo o la Vista.

Además, el patrón MVC también define las interacciones entre sus tres componentes: el Modelo recibe comandos y datos del Controlador. Almacena o procesa estos datos y actualiza la Vista. La Vista permite presentar los datos proporcionados por el Modelo al Usuario. El Controlador acepta entradas del Usuario y las convierte en comandos para el Modelo o la Vista.

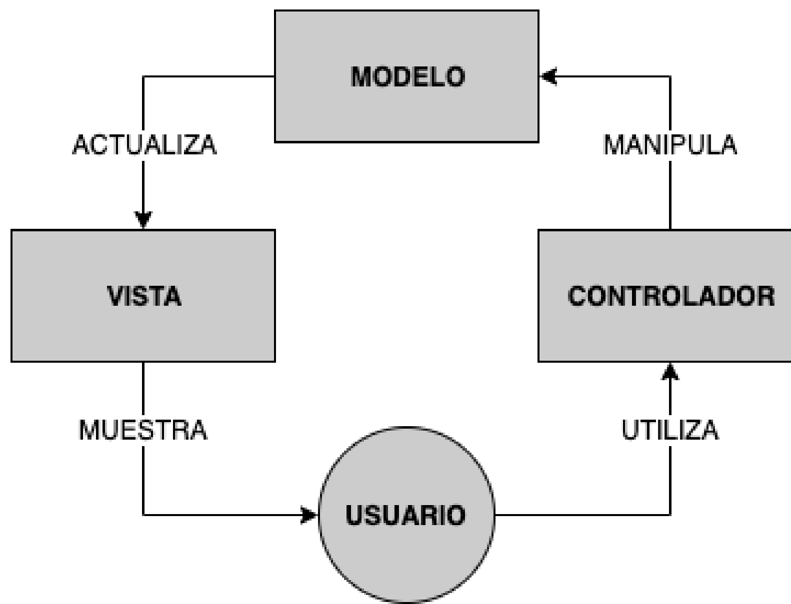

Figura 23. Patrón MVC

En una primera instancia, se crearon unos métodos de prueba para comprobar que todo funcione correctamente.

```
package com.leavecar.microservices.communications.controllers;

import ...

@RestController
public class CommunicationsController {

    @Autowired
    private SessionHolder holder;

    @GetMapping(value = "/test/{date}")
    @ResponseBody
    public LocalDateTime pathVariableExample(@PathVariable @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE_TIME) LocalDateTime date){
        return date;
    }

    @PostMapping(value = "/header/")
    @ResponseBody
    public String headerExample(@RequestBody CommunicationsDateTest Test){
        String system = holder.get().getSystem();
        String actor = holder.get().getActor();
        return system + " - " + actor;
    }

}
```

Clase *CommunicationsController.java* (versión de prueba del controlador principal)

La anotación *@RestController* le indica al framework Spring que esta clase es un controlador. En este caso, este controlador representa la puerta de entrada al microservicio y permite realizar el intercambio de información e instrucciones entre quien consume el servicio (la aplicación principal) y el modelo de datos que es manejado por el servicio.

La anotación @*Autowired* es utilizada para que el framework Spring cree la instancia del objeto anotado (más adelante se profundiza sobre este mecanismo).

Las anotaciones @*GetMapping* y @*PostMapping* significan que cualquier petición HTTP [59] (GET, POST, respectivamente) a las URLs indicadas en las anotaciones, será gestionada por el método relacionado a cada anotación. El método GET es utilizado para recuperar datos del servicio y no debe tener otro efecto. El método POST solicita al servidor que acepte los datos incluidos en el cuerpo del mensaje de la solicitud, generalmente para que sean almacenados.

La anotación @*RequestBody* indica que un parámetro del método debe estar vinculado al cuerpo de la petición al servidor. En el caso de la versión de prueba del controlador, se utiliza un objeto provisorio del tipo *CommunicationsDateTest* que por el momento no cumple ninguna función más que la de verificar que no existan errores al momento de realizar las llamadas.

Luego, en esta misma capa de controladores y a través del mecanismo de anotaciones de Spring, se inyectarán los servicios. De esta manera, las solicitudes que se realicen al servidor quedarán asignadas a los métodos de la capa de servicio. Esto funciona de tal manera ya que la aplicación Spring Boot utiliza las anotaciones del código para dirigir las solicitudes a los métodos correspondientes.

En este punto es conveniente abrir un paréntesis para mencionar brevemente la técnica de inyección de dependencias y el concepto de inversión de control.

“En informática, inyección de dependencias (en inglés *Dependency Injection*, DI) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.” Esta es la definición de Wikipedia [60], y generalmente es difícil de comprender. Cuando una clase A utiliza funcionalidad de otra clase B, entonces se dice que la clase A tiene una dependencia de la clase B. En Java, antes de que se puedan utilizar métodos de otras clases, primero se debe crear el objeto de esa clase. Con lo cual, transferir la tarea de crear el objeto a alguien más (contenedor de inyección de dependencias) y directamente utilizar la dependencia es lo que se llama inyección de dependencias (ver Figura 24).

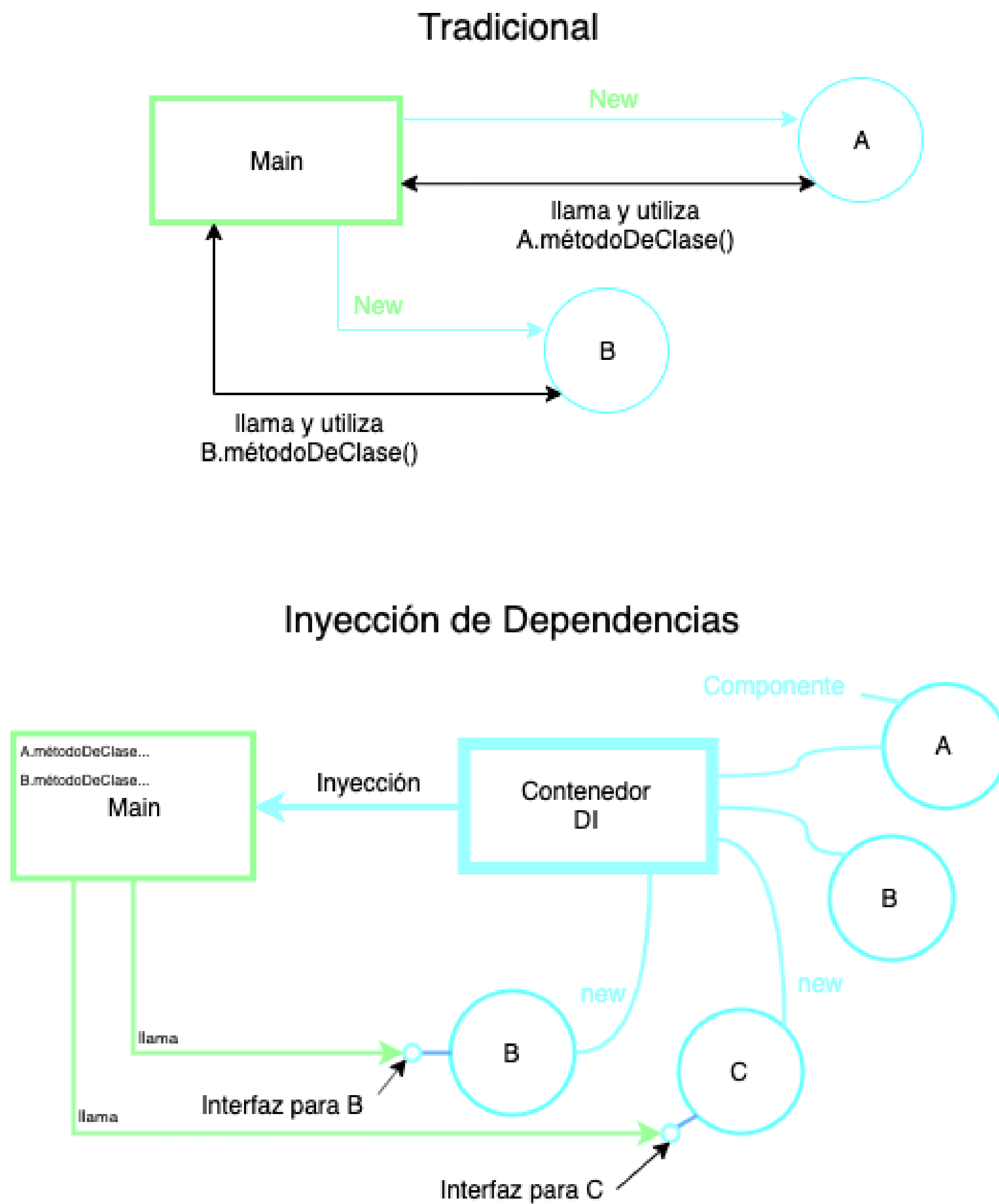

Figura 24. Representación esquemática de mecanismo tradicional de dependencias contra mecanismo de Inyección de Dependencias [61]

Con la inyección de dependencias es posible cambiar los objetos de los cuales las clases dependen en tiempo de ejecución, ya que las dependencias pueden ser inyectadas en tiempo de ejecución en lugar de tiempo de compilación. Esto permite, por ejemplo, que nuestros controladores no deban ser reescritos en el caso que se deseara cambiar alguno de los servicios de los cuales dependen. La inyección de dependencias es responsable de crear los objetos, conocer qué clases requieren esos objetos y proveer a estas clases con los objetos. Si hay algún cambio en los objetos, esto no debería afectar a la clase que los utiliza.

Por otro lado, la inversión de control es el concepto detrás de la inyección de dependencias. Este concepto establece que una clase no debe configurar sus dependencias estáticamente, sino que éstas deben ser configuradas por otra clase exterior. Es el quinto principio de SOLID (cinco principios básicos de programación y diseño orientados a objetos) [62] y establece que una clase debería depender de abstracciones y no de implementaciones (dicho de una manera muy simple). Es decir, de acuerdo a este principio, una clase debe

concentrarse en cumplir con sus responsabilidades y no en crear los objetos necesarios para este fin. Y aquí es donde entra en juego la inyección de dependencias: proporcionar a la clase con los objetos necesarios.

Una de las características más importantes de Spring (el framework utilizado en el desarrollo del microservicio), es la de implementar el mecanismo de inyección de dependencias, lo cual trae beneficios como facilitar las pruebas unitarias, reducir la cantidad de código duplicado (ya que la inicialización de dependencias es realizada por el componente inyector del framework), facilitar la extensión de la aplicación, y ayudar a escribir código más desacoplado, un aspecto no menor en el desarrollo de software.

Cerrando paréntesis y continuando con la descripción del desarrollo, una vez finalizadas las configuraciones iniciales relacionadas al proyecto Spring, se procedió a construir la aplicación (con la herramienta Gradle) y a desplegar la misma para realizar las primeras pruebas manuales y comprobar si el controlador de prueba funcionaba correctamente.

Una aplicación Spring Boot se puede implementar con un servidor embebido. Es posible generar un archivo JAR y ejecutarlo como cualquier otro archivo JAR. Esto evita tener que ejecutar y mantener una instancia de servidor independiente. Para facilitar aún más esto, Spring Boot proporciona soporte para distintos servidores integrados, que pueden configurarse con la herramienta Gradle.

Con la aplicación corriendo, se realizaron las primeras pruebas manuales de llamadas REST a nuestro microservicio. Para esto se utilizó la herramienta Postman [63], la cual es una plataforma para el desarrollo de APIs y que entre otras cosas permite fácil y rápidamente realizar peticiones REST y verificar el funcionamiento de las mismas.

Finalizada la verificación de la configuración inicial, se procedió con la integración de la API de Twilio y la creación de la capa de servicios.

Un paso inicial en esta etapa fue el de la obtención de los datos y credenciales necesarios para poder utilizar la API de la plataforma de Twilio. Para esto se debieron realizar una serie de configuraciones en la consola de administración de la plataforma (ver Figura 25) y así poder crear un servicio Twilio Proxy definitivo (servicio que se utilizará para crear las sesiones de comunicación necesarias para cumplir con los requisitos de la funcionalidad).

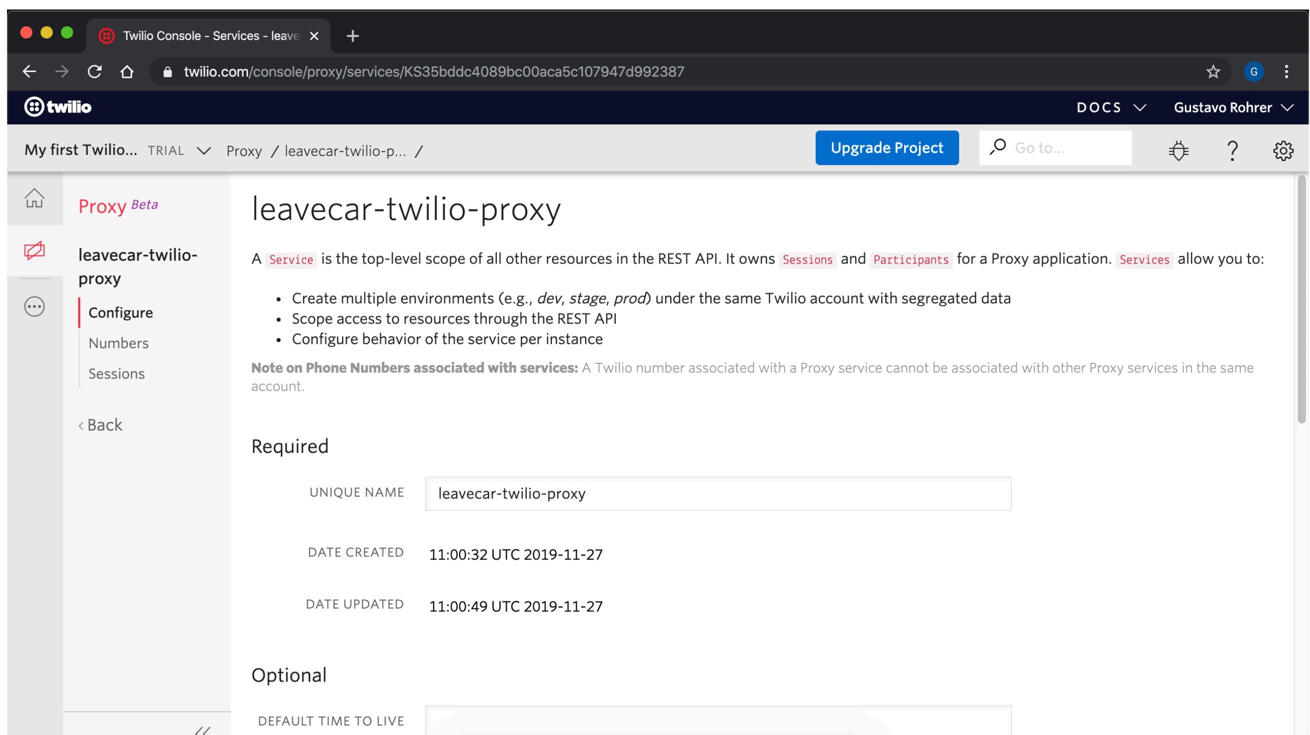

Figura 25. Consola de administración de plataforma Twilio: creación de servicio Twilio Proxy

Una vez obtenidos estos datos, se volcaron los mismos en la configuración del proyecto. Particularmente, se crearon las propiedades correspondientes en el archivo *application.yml*. Estas propiedades luego serán utilizadas por una clase de configuración que a su vez será inyectada en el servicio correspondiente y así se podrá establecer la comunicación con la API.

```
spring:
  application.name: Communications

server:
  compression:
    enabled: true
    mime-types: application/json
  port: 8002


### TO BE OVERRIDE BY ENV CONFIG ##
jwt:
  secret: *******

twilio:
  account-sid: **********************************
  auth-token: **********************************
  service-sid: **********************************
```

Datos de autentificación de la plataforma Twilio en archivo *application.yml*

También fue necesario realizar la correspondiente configuración de dependencias de la API en la herramienta Gradle. Se creó una entrada en la sección de versiones de dependencias del archivo *gradle.properties* y se adicionó la línea relativa en el archivo *build.gradle*.

```
....

#################################################################
# dependencies versions
#################################################################
springBootVersion=2.0.1.RELEASE
mockitoVersion=2.15.0
microserviceCommons=0.1.0
microserviceCloud=0.1.0
twilio=7.22.2
```

Fragmento de archivo *gradle.properties*: integración de SDK de Twilio (versión)

```
....

dependencies {

    compile("com.leavecar.microservices:commons:${microserviceCommons}")
    compile("com.leavecar.microservices:cloud:${microserviceCloud}")
    compile("com.twilio.sdk:twilio:${twilio}")

    //UNCOMMENT IF NEEDED
    //compile("org.springframework.boot:spring-boot-starter-data-jpa")

    testCompile("org.springframework.boot:spring-boot-starter-test")
    testCompile("org.mockito:mockito-core:${mockitoVersion}")
}

....
```

Fragmento de archivo *build.gradle*: integración de SDK de Twilio

Con la configuración de la integración de la API establecida, se corrió la tarea de Gradle para importar las dependencias y así verificar que todo funcione correctamente.

Luego, con la API de la plataforma Twilio integrada, se procedió a crear la capa de servicios. La capa de servicios proporciona modularidad del código relacionado a la lógica y reglas del negocio.

Generalmente la capa de servicio es la que interactúa con la capa de DAO (del inglés *Data Access Object*, Objeto de acceso a datos), la cual provee una interfaz común entre la aplicación y uno o más almacenamientos de datos [64].

La modularidad proporcionada por la capa de servicios también favorece el hecho de mantener un bajo acoplamiento en la aplicación. Por ejemplo, si se tuviese una capa de controladores con 50 métodos que llaman a 20 métodos de la capa de DAO, y luego se decide cambiar algunos de esos métodos de la capa de DAO, sería necesario cambiar los 50 métodos controladores. En cambio, si se cuenta con 20 métodos de la capa de servicios que utilizan los DAO, solo sería necesario modificar estos últimos métodos de servicios.

En principio, el microservicio no tendrá necesidad de persistir datos en una base de datos, con lo cual la capa de servicios solo se limitará a ejecutar la lógica relacionada a la funcionalidad requerida, pero sí creará y actualizará sesiones de comunicación en la plataforma de Twilio.

Finalmente, después de varias iteraciones y pruebas, el archivo Java correspondiente a la capa de servicios resultó en lo siguiente:

```
package com.leavecar.microservices.communications.services;

import ...

@Service
public class CommunicationsService {

    private static final Logger LOGGER =
LoggerFactory.getLogger(CommunicationsService.class);

    private final String serviceSid;

    public CommunicationsService(CommunicationsConfiguration config) {
        String accountSid = config.getAccountSid();
        String authToken = config.getAuthToken();
        Twilio.init(accountSid, authToken);
        serviceSid = config.getServiceSid();
    }

    public ResponseEntity createSession(CreateSessionRequest request) {
        if (request.getParticipantsToAdd().size() != 2) {
            LOGGER.error("Error trying to create communications session for jobId# {}. Number
of participants must be two", request.getJobId());
            return Response.error("Error trying to create communications session for jobId# "
+ request.getJobId() + ". Number of participants must be two");
        }
        AtomicReference<String> changeableParticipantSid = new AtomicReference<>();
        final String[] proxyIdentifier = new String[1];
        try {
            Session session = Session.creator(serviceSid)
                    .setUniqueName("sessionForJob#" + request.getJobId())
                    .setTtl(request.getSessionTtl())
                    .setMode(Session.Mode.VOICE_AND_MESSAGE)
                    .create();
            request.getParticipantsToAdd().forEach(p -> {
                Participant participant = Participant.creator(serviceSid, session.getSid(),
p.getPhone()).setFriendlyName(p.getName()).create();
                if (p.isChangeable()) {
                    changeableParticipantSid.set(participant.getSid());
                }
                proxyIdentifier[0] = participant.getProxyIdentifier();
            });
            CommunicationSessionDTO sessionResponse = new
CommunicationSessionDTO(session.getSid(), proxyIdentifier[0],
changeableParticipantSid.get());
            LOGGER.info("Communications Session created - jobId# {} - session# {}",
request.getJobId(), session.getSid());
            return Response.ok(sessionResponse);
        } catch (ApiException ex) {
            LOGGER.error("An exception occurred trying to create communications session,
service id: {}, code: {}, exception: {}", serviceSid, ex.getCode(), ex.getMessage());
            return Response.error(ex.getMessage(), ex.getCode().toString());
        }
    }

    public ResponseEntity updateSession(UpdateSessionRequest request) {
        try {
            Session session = Session.fetcher(serviceSid, request.getSessionId()).fetch();
            String newChangeableParticipantSid = null;
            if (request.getParticipantRemovedSid() != null) { // updates session participant
                if (request.getNewParticipant() == null) {
                    return Response.error("Error trying to update communications session
participant: " + session.getUniqueName() + ". Must specify data of new participant");
                }
                Participant.deleter(serviceSid, session.getSid(),
request.getParticipantRemovedSid())
                        .delete();
                ParticipantToAdd newParticipant = request.getNewParticipant();
                newChangeableParticipantSid = Participant.creator(serviceSid,
session.getSid(), newParticipant.getPhone())
                        .setFriendlyName(newParticipant.getName())
                        .create().getSid();
            }
            if (request.getSessionTtl() != null) { // updates session ttl
                Session.updater(serviceSid, session.getSid())
                        .setTtl(request.getSessionTtl())
                        .setStatus(Session.Status.IN_PROGRESS)
                        .update();
```

```
            }
            LOGGER.info("Communications Session updated - {}", session.getUniqueName());
            return Response.ok(newChangeableParticipantSid);
        } catch (ApiException ex) {
            LOGGER.error("An exception occurred trying to update communications session id:
{}, service id: {}, code: {}, exception: {}", request.getSessionId(), serviceSid,
ex.getCode(), ex.getMessage());
            return Response.error(ex.getMessage(), ex.getCode().toString());
        } catch (Exception ex) {
            LOGGER.error("An exception occurred trying to update communications session id:
{}, service id: {}, exception: {}", request.getSessionId(), serviceSid, ex.getMessage());
            return Response.error(ex.getMessage());
        }
    }

    public ResponseEntity deleteSession(String sessionSid) {
        try {
            Session.deleter(serviceSid, sessionSid).delete();
            LOGGER.info("Communications Session deleted - sessionId# {}", sessionSid);
            return Response.ok();
        } catch (ApiException ex) {
            LOGGER.error("An exception occurred trying to delete communications session id:
{}, code: {}, exception: {}", sessionSid, ex.getCode(), ex.getMessage());
            return Response.error(ex.getMessage(), ex.getCode().toString());
        } catch (Exception ex) {
            LOGGER.error("An exception occurred trying to delete communications session id:
{}, exception: {}", sessionSid, ex.getMessage());
            return Response.error(ex.getMessage());
        }
    }
}
```

Archivo *CommunicationsService.java*

La lógica final ejecutada en la capa de servicios surgió a partir del análisis de toda la información relacionada a los requerimientos de la funcionalidad solicitada. Se contemplaron los criterios de aceptación, los distintos casos de uso, y también aquella lógica ya existente en la aplicación que se vinculaba al desarrollo en cuestión.

A continuación, se brinda un detalle de los principales aspectos del archivo *CommunicationsService.java:*

- Anotación @*Service*: indica al framework Spring que la clase es un proveedor de servicios. Con esto, el contexto de Spring puede auto detectarla e inyectarla donde es necesario (en el archivo controlador, por ejemplo).

- Propiedad *LOGGER*: es un objeto perteneciente a una clase que brinda funcionalidad relacionada al registro de eventos de la aplicación. En este caso, el mismo se utiliza para registrar errores y éxitos en los procesos de comunicación con la API de la herramienta Twilio.

- Propiedad *serviceSid*: representa el identificador de servicio que se obtiene a partir de la inicialización del ambiente de Twilio. En este caso esta información es inyectada desde el archivo *application.yml*.

- Constructor de clase (*public CommunicationsService(CommunicationsConfiguration config) { … }*): este método es uno de los tres mecanismos a través de los cuales Spring inyecta dependencias (luego están los métodos *setter* y la inyección a través de interfaces). Aquí se obtienen los datos pertenecientes a la configuración de Twilio y se inicializa el ambiente de la plataforma, el cual queda disponible para ser utilizado a través de los distintos métodos de la clase *CommunicationsService.java*. Asimismo, es importante destacar que el constructor de clase recibe como parámetro un objeto del tipo

CommunicationsConfiguration. Es a través de esta clase mediante la cual se obtiene la configuración relativa del archivo *application.yml*:

```
package com.leavecar.microservices.communications.configuration;

import ...

@Component
@ConfigurationProperties("twilio-proxy")
public class CommunicationsConfiguration {

    private String accountSid;
    private String authToken;
    private String serviceSid;

    public String getAccountSid() {
        return accountSid;
    }

    public void setAccountSid(String accountSid) {
        this.accountSid = accountSid;
    }

    public String getAuthToken() {
        return authToken;
    }

    public void setAuthToken(String authToken) {
        this.authToken = authToken;
    }

    public String getServiceSid() {
        return serviceSid;
    }

    public void setServiceSid(String serviceSid) {
        this.serviceSid = serviceSid;
    }
}
```

Archivo *CommunicationsConfiguration.java*

En el archivo *CommunicationsConfiguration.java* se obtiene la información relacionada a las credenciales necesarias para acceder a las utilidades brindadas por la plataforma Twilio. Esto se hace a través de la anotación *@ConfigurationProperties("twilio-proxy")*. El framework Spring enlazará automáticamente cualquier propiedad definida en nuestro archivo de propiedades *application.yml* que tenga de prefijo "*twilio-proxy*" y el mismo nombre que uno de los campos en la clase *CommunicationsConfiguration.java*.

- Métodos *createSession*, *updateSession* y *deleteSession*: representan los métodos que gestionan cada faceta de la sesión de comunicación de la plataforma Twilio. Dentro de cada método se establece la lógica correspondiente y necesaria para la creación, actualización y borrado de una sesión de comunicación respectivamente. En general, cada método retorna una respuesta del tipo *ResponseEntity*, que es un objeto con la información relacionada al resultado del método llamado, y además posee información relativa al estado resultante de la operación HTTP adicionada por Spring. Los métodos también tienen como parámetro un objeto que contiene los datos necesarios para la operación que se realizará con la sesión de comunicación (según el método al que se esté invocando). Los tipos de los parámetros de los métodos *createSession* y *updateSession (CreateSessionRequest* y *UpdateSessionRequest)* son DTO (del inglés *Data Transfer Object*, objeto de transferencia de datos), es decir, un tipo de objeto cuya funcionalidad es la de solo transportar datos entre procesos [65]. Luego, cada método

desarrolla su lógica y anexa el resultado de ésta al objeto que representa la respuesta que se devuelve. A continuación, se detalla brevemente la lógica de cada método:

- *createSession*: este método se encarga de la creación de una sesión necesaria para poder establecer la comunicación entre dos participantes. Aquí primeramente se realiza un control acerca de los datos que se reciben en el servicio, verificando puntualmente si se cuenta con la información requerida de dos participantes para poder crear una sesión de comunicación en la plataforma Twilio. Pasado este control, se procede con la creación de la sesión y luego se crean los participantes de la misma. Finalmente se construye un objeto DTO con la información resultante de la sesión creada y se anexa en la respuesta del servicio.
- *updateSession*: encargado de la actualización de datos de la sesión de comunicación, este método primeramente recupera los datos de la sesión creada en la plataforma Twilio y luego controla qué datos se desean actualizar, procediendo con tal acción según corresponda. Por un lado, es posible actualizar uno de los participantes de la sesión de comunicación. Para esto, se controla que la información del nuevo participante haya sido enviada en la petición al servicio, y en caso afirmativo, se procede a eliminar de la sesión el participante reemplazado, y luego se crea y se incorpora el nuevo participante. Finalmente, también es posible actualizar el tiempo de vida de la sesión de comunicación. Este dato en particular es un dato de control que establece durante cuánto tiempo estará activa la sesión de comunicación. Este tiempo a su vez depende de diferentes reglas del negocio y es necesario controlarlo a fin de evitar costos innecesarios por la utilización de la plataforma Twilio.
- *deleteSession*: este método simplemente se encarga de la eliminación de una sesión de comunicación creada en la plataforma Twilio. A través del método específico de la API de Twilio, elimina aquella sesión indicada por el identificador correspondiente enviado en la llamada al método.

- Manejo de excepciones: cada interacción con la API de Twilio es realizada contemplando el manejo de excepciones (ver Figura 26). Una excepción en el desarrollo de software es un error inesperado que puede resultar en la disrupción del flujo normal de ejecución del sistema [66]. Básicamente, al ocurrir un error en la comunicación con la API de Twilio, o algún otro error correspondiente a la ejecución de la lógica del código, esta falla es interceptada, registrada y manejada, con el fin de enviar a quien esté consumiendo el servicio una respuesta conocida frente al error que se produjo.

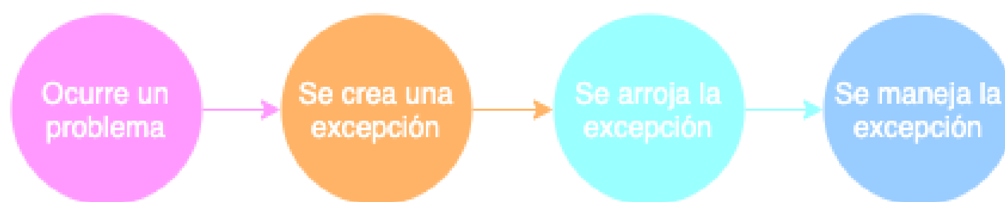

Figura 26. Ejemplo simplificado de flujo de manejo de excepciones en el lenguaje Java.

También fue necesaria la creación de otros archivos de configuración adicionales y relacionados a la seguridad de la aplicación. La clase *WebMvcConfiguration.java*, por ejemplo, que básicamente agrega un interceptor relacionado a la gestión de sesiones de usuarios. Esto permite que las peticiones que se realizan al servicio sean interceptadas para poder controlar la información de sesión de quién está realizando la petición, y de esta manera corroborar la seguridad de la petición entrante al servicio.

```
package com.leavecar.microservices.communications.configuration;

import ...

@Configuration
public class WebMvcConfiguration implements WebMvcConfigurer {

    @Autowired
    private SessionHolder sessionHolder;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new SessionInterceptor(sessionHolder));
    }
}
```

Archivo *WebMvcConfiguration.java*

El paso siguiente fue el desarrollo de las pruebas unitarias de código relacionadas a los métodos de la capa de servicios.

Una prueba unitaria es, como el nombre bien lo dice, una prueba de cada área, función (o unidad) del código. Aquí es importante destacar la importancia de esta práctica. Muchas veces las pruebas unitarias de las funcionalidades no son realizadas por falta de tiempo, falta de conocimiento o solamente porque no son requeridas. Pero en realidad, es una práctica que debería ser primordial ya que no solo permite asegurar que el código que se desarrolló funcione como se espera, sino que además brinda otras ventajas como:

- Permite realizar cambios en el código de manera segura y mejora la mantenibilidad del software: una vez que se ha probado la lógica de una función, esta puede ser cambiada siendo siempre conscientes de que las pruebas relacionadas deben ejecutarse exitosamente. El escenario más común y menos favorable cuando no se realizan pruebas es el de realizar una corrección de algún error y que la misma solucione el problema pero que termine “rompiendo” otra parte de la aplicación.
- Reduce la cantidad de errores: a medida que se prueban los fragmentos individuales de código, es posible descubrir y corregir de manera temprana los problemas en el código.

Cabe destacar que no solo se realizaron pruebas sobre el comportamiento esperado, sino que también se contemplaron otros casos de uso adversos y con información faltante para lograr una mayor cobertura de los escenarios a probar.

```
package com.leavecar.services;

import ...

@RunWith(SpringRunner.class)
public class CommunicationsServiceTest {

    @Rule
    public final OutputCapture outputCapture = new OutputCapture();

    private static String accountSid;
    private static String authToken;
    private static String serviceSid;

    @Before
    public void setUp() throws IOException {
        YamlPropertySourceLoader propertySourceLoader = new YamlPropertySourceLoader();
        PropertySource<?> propSource = propertySourceLoader.load("app", new
ClassPathResource("application.yml")).get(0);
        accountSid = (String) propSource.getProperty("twilio-proxy.account-sid");
        authToken = (String) propSource.getProperty("twilio-proxy.auth-token");
        serviceSid = (String) propSource.getProperty("twilio-proxy.service-sid");
    }
```

```
    @Test
    public void testCreateSessionHappyPath() {
        // given
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);

        // then
        CommunicationSessionDTO response = (CommunicationSessionDTO)
createSessionResponse.getBody();

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        Assertions.assertThat(response).isNotNull();
        Assertions.assertThat(response.getSessionId()).isNotNull();
        Assertions.assertThat(response.getProxyIdentifier()).isNotNull();
        Assertions.assertThat(response.getChangeableParticipantSid()).isNotNull();

        // removes session created
        buildCommunicationsService().deleteSession(response.getSessionId());
    }

    @Test
    public void testCreateSessionMissedParticipant() {
        // given
        CreateSessionRequest request = new CreateSessionRequest(Lists.newArrayList(new
ParticipantToAdd("+15005550006", "participantA", false)), 3600,1L);

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("Error trying to create
communications session for jobId# 1. Number of participants must be two");
    }

    @Test
    public void testCreateSessionInvalidAccountSid() {
        // given
        accountSid = "ACe94fc5217fd5b5d6cec7cb0912ec9999";
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to create communications session");
    }

    @Test
    public void testCreateSessionInvalidAuthToken() {
        // given
        authToken = "960c1cd519b898b78703c980e41c9999";
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to create communications session");
    }

    @Test
```

```
    public void testCreateSessionInvalidServiceSid() {
        // given
        serviceSid = "960c1cd519b898b78703c980e41c0000";
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to create communications session");
    }

    @Test
    public void testUpdateSessionHappyPath() {
        // given
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);
        CommunicationSessionDTO response = (CommunicationSessionDTO)
createSessionResponse.getBody();
        assert response != null;
        String sessionSid = response.getSessionId();
        String participantRemovedSid = response.getChangeableParticipantSid();
        ParticipantToAdd participantToAdd = new ParticipantToAdd("+15005550008",
"newParticipant", true);

        // when
        ResponseEntity updateSessionResponse = buildCommunicationsService().updateSession(new
UpdateSessionRequest(sessionSid, participantRemovedSid, participantToAdd, null));

        // then

Assertions.assertThat(updateSessionResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        String newChangeableParticipantSid = (String) updateSessionResponse.getBody();
        Assertions.assertThat(newChangeableParticipantSid).isNotNull();

Assertions.assertThat(newChangeableParticipantSid).isNotEqualTo(participantRemovedSid);

        // removes session created
        buildCommunicationsService().deleteSession(sessionSid);
    }

    @Test
    public void testUpdateSessionNewParticipantMissed() {
        // given
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);
        CommunicationSessionDTO response = (CommunicationSessionDTO)
createSessionResponse.getBody();
        assert response != null;
        String sessionSid = response.getSessionId();
        String participantRemovedSid = response.getChangeableParticipantSid();

        // when
        ResponseEntity updateSessionResponse = buildCommunicationsService().updateSession(new
UpdateSessionRequest(sessionSid, participantRemovedSid, null, null));

        // then

Assertions.assertThat(updateSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);

        // removes session created
        buildCommunicationsService().deleteSession(sessionSid);
    }

    @Test
    public void testUpdateSessionTtlHappyPath() {
        // given
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);
        CommunicationSessionDTO response = (CommunicationSessionDTO)
```

```
createSessionResponse.getBody();
        assert response != null;
        String sessionSid = response.getSessionId();

        // when
        ResponseEntity updateSessionResponse = buildCommunicationsService().updateSession(new
UpdateSessionRequest(sessionSid, null, null, 7200));

        // then
        Session session = Session.fetcher(serviceSid, sessionSid).fetch();

Assertions.assertThat(updateSessionResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        Assertions.assertThat(session.getTtl()).isEqualTo(7200);

        // removes session created
        buildCommunicationsService().deleteSession(sessionSid);
    }

    @Test
    public void testUpdateSessionInvalidAccountSid() {
        // given
        accountSid = "ACe94fc5217fd5b5d6cec7cb0912ec9999";

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().updateSession(Mockito.mock(UpdateSessionRequest.class));

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to update communications session");
    }

    @Test
    public void testUpdateSessionInvalidAuthToken() {
        // given
        authToken = "960c1cd519b898b78703c980e41c9999";

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().updateSession(Mockito.mock(UpdateSessionRequest.class));

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to update communications session");
    }

    @Test
    public void testUpdateSessionInvalidServiceSid() {
        // given
        serviceSid = "960c1cd519b898b78703c980e41c0000";

        // when
        ResponseEntity createSessionResponse =
buildCommunicationsService().updateSession(Mockito.mock(UpdateSessionRequest.class));

        // then

Assertions.assertThat(createSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("An exception occurred
trying to update communications session");
    }

    @Test
    public void testDeleteSessionHappyPath() {
        // given
        CreateSessionRequest request = TestUtils.createCreateSessionRequest();
        ResponseEntity createSessionResponse =
buildCommunicationsService().createSession(request);
        String sessionSid = ((CommunicationSessionDTO)
createSessionResponse.getBody()).getSessionId();

        // when
```

```
        ResponseEntity deleteSessionResponse =
buildCommunicationsService().deleteSession(sessionSid);

        // then

Assertions.assertThat(deleteSessionResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
    }

    @Test
    public void testDeleteSessionWithInvalidSessionSid() {
        // given
        String invalidSessionSid = "KS900eefe97e1386438444c673508ef000";

        // when
        ResponseEntity deleteSessionResponse =
buildCommunicationsService().deleteSession(invalidSessionSid);

        // then

Assertions.assertThat(deleteSessionResponse.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST
);
        Assertions.assertThat(outputCapture.toString()).contains("code: 20404");
    }

    private static CommunicationsService buildCommunicationsService() {
        CommunicationsConfiguration config = new CommunicationsConfiguration();
        config.setAccountSid(accountSid);
        config.setAuthToken(authToken);
        config.setServiceSid(serviceSid);
        return new CommunicationsService(config);
    }
}
```

Archivo *CommunicationsServiceTest.java*

Algunas de las librerías y herramientas utilizadas para facilitar el desarrollo de las pruebas unitarias fueron las siguientes:

- JUnit: es un framework para pruebas para el lenguaje Java. Los métodos de prueba deben ser anotados por la anotación @*Test*. También es posible definir un método para ejecutar antes o después de cada uno o de todos los métodos de prueba. Esto último se consigue con las anotaciones @*Before* (o @*After*) y @*BeforeClass* (o @*AfterClass*) [67].

- Spring test: es uno de los módulos del framework Spring que proporciona soporte específico para las pruebas de código, como ser el soporte para cargar contextos de aplicación, inyección de dependencias de instancias de prueba, ejecución de métodos de prueba transaccional, etc. [68].

- Mockito: es un framework de pruebas de código abierto para Java. Permite la creación de objetos duplicados que son simulados para las pruebas unitarias [57].

Concluido el desarrollo de la capa de servicios y las pruebas correspondientes, se crearon los puntos de entrada definitivos en la capa de controladores.

```java
package com.leavecar.microservices.communications.controllers;

import ...

@RestController
public class CommunicationsController {

    private final CommunicationsService communicationsService;

    public CommunicationsController(CommunicationsService communicationsService) {
        this.communicationsService = communicationsService;
    }

    @PostMapping(value = "/createSession")
    public ResponseEntity createSession(@RequestBody CreateSessionRequest request) {
        return communicationsService.createSession(request);
    }

    @PutMapping(value = "/updateSession")
    public ResponseEntity updateSession(@RequestBody UpdateSessionRequest request) {
        return communicationsService.updateSession(request);
    }

    @DeleteMapping("/deleteSession/{sessionSid}")
    public ResponseEntity deleteSession(@PathVariable("sessionSid") String sessionSid) {
        return communicationsService.deleteSession(sessionSid);
    }
}
```

Archivo *CommunicationsController.java*

Finalmente, se definieron solo tres puntos de entrada iniciales en el microservicio. Estos tres puntos de entrada se corresponden con los métodos de la capa de servicios creados.

Se agregaron dos nuevas anotaciones, *@PutMapping* y *@DeleteMapping*, y de igual manera, cualquier petición HTTP que se realice al servicio y que sea del tipo PUT o DELETE, será gestionada por el método vinculado a cada anotación y según corresponda. El método PUT se utiliza para cargar información al servidor generalmente con la intención de actualizar un recurso. Y el método DELETE, borra el recurso especificado.

La anotación *@PathVariable* sirve para obtener el valor del parámetro que recibe el método directamente de la URL asociada al punto de entrada del controlador.

Particularmente, en el microservicio creado no se cuenta con un modelo de datos definido específicamente, sino que éste está representado por las sesiones de comunicación creadas en la plataforma Twilio. Con lo cual, las sesiones de comunicación representan el componente Modelo en el patrón MVC, y es sobre éstas en dónde se realizan las operaciones aceptadas por el controlador del servicio.

Por último, es importante señalar que, en la etapa de migración de funcionalidad existente en la aplicación principal al microservicio creado, se deberán crear los métodos necesarios para poder brindar el soporte correspondiente. Con el fin de habilitar la operatoria de notificaciones SMS en el nuevo microservicio, se deberán crear nuevos puntos de entrada en la capa de controladores y la lógica respectiva en la capa de servicios.

3.4.5.5 Integración con flujo de trabajo de CI/CD

De manera paralela al desarrollo de la funcionalidad, se configuraron los flujos de trabajo de CI/CD (Continuos Integration / Continuos Delivery, Integración Continua / Entrega Continua).

Por un lado, CI es el proceso de unificar el código de todos los desarrolladores en una fuente principal, con el objetivo de encontrar y solucionar errores de manera temprana. Esto permite contar con una retroalimentación rápida y acelerar el proceso de desarrollo de software. Solucionar errores anticipadamente es menos costoso que posponerlos hasta que es demasiado tarde.

Un servidor de CI generalmente realiza las siguientes tareas:

- adquiere acceso al control del código fuente, por ejemplo, GitLab, y realiza comprobaciones del código en el servidor;
- ejecuta un análisis estático del código para identificar problemas relacionados con la sintaxis;
- construye y compila la aplicación usando el código fuente;
- ejecuta las diferentes pruebas del código;

El equipo de desarrollo corrige los problemas tan pronto como los encuentra el servidor. Las prácticas de CI incluyen la automatización de compilación, autocomprobación y compilaciones más rápidas.

Por otro lado, CD es una extensión del proceso de CI, donde el software se puede poner a disponibilidad a distintos tipos de usuarios en cualquier momento.

El proceso de CD agrega pasos adicionales al proceso de CI:

- empaquetamiento de la aplicación;
- firmado del código (si es necesario);
- implementación de la aplicación en ambientes específicos.

Con CD, los equipos de desarrollo pueden lograr versiones totalmente automatizadas, con pocos riesgos de errores y a un bajo costo. Este proceso permite que la implementación del sistema pueda realizarse de manera muy sencilla obteniendo una compilación del código liberable cuando se lo requiera. Sin embargo, siempre se requiere intervención manual para implementar el software en el ambiente productivo. Con otras prácticas como el *despliegue continuo*, no hay intervención manual, y el código comprometido llega a la producción de inmediato.

Para el caso del presente microservicio se utilizó un servidor de CI Jenkins [40], y se realizaron las configuraciones correspondientes para ejecutar casos de pruebas automatizados y para realizar la instalación del servicio en los diferentes ambientes de desarrollo.
Principalmente se configuró una tarea de Jenkins para que esté “alerta” a las unificaciones de código que se realizan en la rama de *develop*, y cuando esto sucede, automáticamente se instala esa versión del código en el ambiente de CI. Esta tarea de Jenkins en particular se encarga de ejecutar pruebas, notificar ante el fallo de la ejecución de las pruebas, disparar un proceso de la herramienta SonarQube [41] para generar informes de calidad del código, y además instalar la versión del código en el ambiente correspondiente (en caso de que las pruebas se hayan ejecutado correctamente). Esta disposición puede verse en parte representada en la Figura 27.

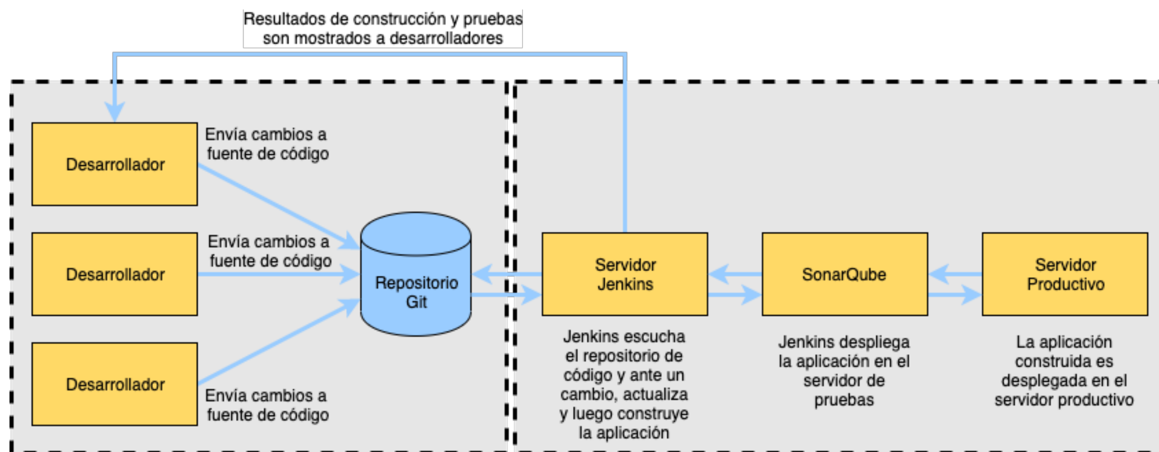

Figura 27. Ejemplo de modelo de arquitectura independiente de Jenkins

Para el flujo de trabajo de CD específicamente, se separó el código fuente, la configuración y la especificación del entorno para que puedan evolucionar de forma independiente. De esta manera, es posible cambiar la configuración sin volver a implementar el código fuente.

Luego se configuró en Jenkins otra tarea de despliegue a ambientes específicos, para contar con la posibilidad de instalar el servicio en un ambiente en particular según la necesidad. Esta tarea toma un número de versión del código fuente, ejecuta pruebas automatizadas y pone a disposición el software en el ambiente que se haya seleccionado.

3.4.5.6 Integración de la nueva funcionalidad a la aplicación principal

Ya con el microservicio creado y funcionando en un ambiente de desarrollo, el próximo paso consistió en el desarrollo de la lógica necesaria en el código de la aplicación principal para que, a partir de condiciones específicas, se haga uso del nuevo microservicio de comunicaciones y de esta manera se creen las sesiones de comunicación que permitirán cumplir con los requerimientos solicitados.

Primeramente, se realizó la integración del microservicio de comunicaciones creado en la aplicación principal. Al igual que con la integración de la API de Twilio en su momento, aquí se creó una entrada en el archivo *application.yml* de la aplicación Java principal, pero en este caso para almacenar la URL del microservicio.

Luego, se desarrolló una clase Java específica que contendría la lógica para comunicarse con el microservicio. Esta clase recibe la URL del microservicio a través de la anotación *@ConfigurationProperties("communications")* y define los métodos correspondientes para comunicarse a través de llamadas HTTP con el microservicio de comunicaciones.

Finalmente, se detectaron las condiciones en las que se debían crear sesiones de comunicación y se introdujo el flujo de acciones específicas para que la nueva funcionalidad pueda integrarse al curso normal de operación del sistema. Es decir, a partir de este punto, al ocurrir determinados eventos, la aplicación principal se comunicará con el nuevo microservicio de comunicaciones brindándole la información necesaria para que éste cree, actualice o elimine una sesión de comunicación según corresponda y registre los resultados del procesamiento de la sesión en la parte del modelo correspondiente para su posterior utilización.

3.4.5.7 Configuraciones finales y puesta en producción

Uno de los pasos culminantes antes de poder exponer la funcionalidad del microservicio dentro del ambiente productivo fue el chequeo de datos de configuración.

En primer lugar, se creó un archivo de configuración específico ubicado en lo que sería el ambiente productivo y en donde las propiedades definidas tienen precedencia sobre las propiedades del archivo *application.yml*. Cuando una aplicación solo se ejecuta en el ambiente de desarrollo entonces no hay ningún problema en mantener el archivo de configuración *application.yml*, pero se presentan problemas para otros ambientes como el de prueba o producción. En esos ambientes, puede ser necesario cambiar cualquier configuración o actualizar algunas variables en el archivo de configuración. Cada vez que se requiere cambiar alguna configuración, es necesario cambiar el archivo *application.yml* y volver a ejecutar todo el procedimiento de lanzamiento de la aplicación. Y este proceso puede que no sea el más eficiente para el resto de las partes vinculadas al servicio. A fin de resolver el problema, se debe usar una configuración distinta en cada entorno.

Luego, se realizaron las gestiones necesarias junto al equipo de infraestructura con el objetivo de contar con una URL provisoria del microservicio y así poder realizar una serie de pruebas finales.

También se verificaron archivos de configuración de la aplicación principal con el fin de poder determinar en qué lugares sería necesario adicionar la información requerida para que se pueda comenzar a consumir el nuevo microservicio. Se modificaron estos archivos adicionando la propiedad correspondiente a la URL del microservicio creado.

Finalmente, en el momento de la puesta en producción, se apagaron todas las aplicaciones y servicios, se levantó el nuevo microservicio productivo, se realizaron las últimas pruebas con una aplicación apuntando a la URL provisoria del microservicio, se modificaron reglas de ruteo para la nueva URL productiva, y se realizó un flujo completo de pruebas con usuarios de prueba del sistema. Concluidas las pruebas finales, se procedió a levantar el resto del sistema productivo.

3.5 Migración de funcionalidad existente al microservicio dedicado

Si bien el título de esta sección hace referencia a una de las cuestiones centrales marcadas en la propuesta del presente trabajo, aquellas tareas relacionadas a la actividad de mover la lógica que ya existía en la aplicación principal hacia el nuevo microservicio creado no llegaron a concretarse. Por cuestiones de prioridades en los requerimientos del cliente en el ámbito laboral, las tareas de migración y refactoring solo se limitaron a ser detalladas y planificadas para una futura realización.

Sin embargo, a pesar de que estas modificaciones no fueron ejecutadas, sí es posible brindar una descripción acerca de los principales detalles de su planificación, así como de los principales puntos que se deberán tener en cuenta a fin de trasladar funcionalidad productiva desde la aplicación principal al nuevo microservicio de comunicaciones.

A continuación, se brinda un breve detalle general acerca de la planificación de los principales puntos a considerar al momento de realizar la migración.

3.5.1 Planificación de migración de funcionalidad

La modificación de la aplicación principal no será un procedimiento de un solo paso; se deberá realizar incrementalmente sin afectar a los usuarios finales.

Con el nuevo microservicio funcionando con la nueva funcionalidad requerida originalmente, se deberá proceder a replicar (en el microservicio) la funcionalidad relacionada que se encuentra en la aplicación principal. Como se expresó previamente en la sección “Funcionalidad relacionada al microservicio creado”, la funcionalidad ya existente es la vinculada al envío de notificaciones a través de mensajes de texto SMS, funcionalidad que es realizada a través de la integración de la API de Twilio.

Una de las primeras tareas que deberá realizarse será la incorporación al nuevo microservicio de la lógica necesaria para el envío de notificaciones a través de SMS. En este sentido se tendrá que replicar en el microservicio la información relativa a la integración de la API de envío de SMS de Twilio. Aquí se podrá observar uno de los beneficios de agrupar lógica relacionada dentro de un único sistema, ya que parte de estos datos de configuración ya se encontrarán disponibles en el microservicio debido a que los datos de la plataforma Twilio son compartidos entre sus distintas herramientas.

Luego, será necesario reproducir los métodos de la capa de servicios junto con el código necesario para realizar la lógica de envío de SMS y, finalmente crear los nuevos puntos de entrada en el microservicio, los cuales serán consumidos por la aplicación principal una vez migrada la funcionalidad.

El paso culminante en el proceso de migración sería la modificación o *refactoring* de la lógica de la aplicación principal. Con la funcionalidad replicada y testeada en el nuevo microservicio, se tendrán que realizar las modificaciones en la aplicación principal para que se comience a utilizar el microservicio. En este punto, de manera preventiva siempre es aconsejable mantener el código muerto relativo a la funcionalidad migrada en la aplicación monolítica, creando tareas de deuda técnica para removerlo cuando se considere necesario.

Si bien los detalles sobre la planificación de las tareas de migración aquí expuestos podrían considerarse una actividad sencilla, modificar una aplicación, en este caso extrayendo funcionalidad a un microservicio y haciendo que el sistema principal consuma dicha funcionalidad de allí, no siempre es tarea fácil.

Las pruebas de integración son complejas. El despliegue en el entorno de desarrollo ya deja de ser una tarea relativamente trivial siendo que, aunque el código migrado de la aplicación ahora está en un servicio aislado, se debe tener todas las partes componentes de la aplicación corriendo en el ambiente en el que se está trabajando, a fin de probar que todo funcione correctamente.

4. Trabajos futuros

En esta sección se presentan aquellos trabajos futuros que se consideran que pueden desprenderse de la presente exposición:

- En general, cualquier tipo de profundización a detalle de algunos de los puntos marcados en la sección de “Desafíos y deudas de los microservicios”: realizar un trabajo relacionado a alguno de estos ítems sería considerado como una actividad de valor para poder dilucidar aún más sobre el tema relativo.
- Investigación sobre los avances de infraestructura en la nube y cómo esto impacta en el desarrollo de microservicios: cada vez parece más cercano el punto en el que los desarrolladores podrán crear servicios e implementarlos sin importar dónde estos residan, y sin duda esto tiene un gran impacto en las prácticas de desarrollo.
- Investigación de herramientas que facilitan el desarrollo, las pruebas y la puesta en ejecución de un sistema implementado con microservicios: si bien existen varias tecnologías populares, hay muchas otras que tal vez no tengan la notoriedad merecida y que podrían resultar de gran ayuda en las tareas de desarrollo.
- Investigación sobre las implicaciones de la inteligencia artificial y el aprendizaje automático: es difícil negar que estas disciplinas poseen un gran auge en la actualidad, y los proveedores de servicios de computación en la nube ya han comenzado a aumentar la variedad de herramientas e integraciones para eliminar la complejidad de dichas materias. Como resultado, las implementaciones no serán más que simplemente una cuestión de diseñar las llamadas de servicios adecuadas para una API en la nube y no pasarán por el hecho de la creación de algoritmos e infraestructura. Un estudio sobre tales cuestiones se cree más que interesante.

Si bien el listado previo de posibles trabajos a realizar a partir de la tesina que aquí se expone puede resultar un tanto escueto, es necesario prestar atención a la propuesta que hace referencia a la investigación de aquellos aspectos señalados en la sección de “Desafíos y deudas de los microservicios”, ya que de la misma pueden desprenderse una gran variedad de trabajos.

5. Conclusiones

En el presente trabajo se expuso primeramente una investigación bibliográfica describiendo los principales aspectos acerca de los microservicios. Luego se presentó un ejemplo de desarrollo de un microservicio desde cero, pasando por cada una de las principales etapas relativas a la creación del sistema e indicando los aspectos relacionados en cada parte.

A partir del desarrollo de esta tesina, es posible concluir que la arquitectura de microservicios permite lograr sistemas de manera más ágil, más segura y con mayor calidad. Los servicios desacoplados permiten a los equipos de desarrollo iterar rápidamente y con un impacto mínimo en el resto del sistema. Los microservicios proporcionan una manera fácil de prevenir o alejarse de la complejidad de los servicios monolíticos grandes a través de la utilización de servicios pequeños fácilmente mantenibles. También, queda evidenciado como a largo plazo resulta beneficioso poseer toda una determinada lógica de negocio agrupada bajo un microservicio específico.

Crear un microservicio utilizando los frameworks y las herramientas disponibles en la actualidad es relativamente sencillo. Sin embargo, si bien para el desarrollo aquí presentado se contaba en parte con documentación de experiencias similares anteriores que facilitaron aún más el desarrollo, tal y como fue mencionado, la utilización de microservicios introduce complejidades en el sistema que requieren de un esfuerzo adicional para resolverse. El desarrollo de sistemas distribuidos, como lo son los microservicios, requiere de conocimientos específicos y experiencia en el área. Adicionalmente, para que las arquitecturas involucradas sean completamente funcionales, se necesitan servicios de soporte como, por ejemplo, el descubrimiento de servicios y balanceadores de carga. Durante los pasos del desarrollo, se dedicó algún tiempo extra revisando estos conceptos y las herramientas y bibliotecas correspondientes. Aún así, a menudo se utilizó de manera incorrecta estas tecnologías. Por lo tanto, se puede afirmar que para aprovechar al máximo los microservicios, es necesario estar familiarizados con los conceptos relativos y sentirse cómodo con este tipo de programación.

Asimismo, cuando se modela un microservicio, es necesario ser disciplinado y respetar los tres principios de diseño relativos más importantes: propósito o responsabilidad única, bajo acoplamiento, alta cohesión. Esta es quizás la única manera de obtener todo el potencial de la arquitectura de microservicios.

Finalmente, se coincide y se ratifica lo declarado en la mayoría de la bibliografía consultada acerca de que la arquitectura de microservicios no es una bala de plata. El enfoque de microservicios no es una panacea: permite resolver algunos problemas, pero introduce otros. Algunos de los inconvenientes más comunes que se pueden mencionar son:

- En un entorno distribuido, entran en escena una gran variedad de nuevos componentes que pueden fallar. Cuando las posibles fallas en los microservicios no son correctamente gestionadas, éstas pueden ser catastróficas para el sistema general. En el caso del desarrollo aquí expuesto, un ejemplo son las distintas excepciones que pueden ocurrir ya sea en el microservicio de comunicaciones como en la comunicación con la API de Twilio. El tratamiento indebido de estas nuevas interacciones introducidas al sistema general puede concluir en una serie de eventos que afectaría seriamente el flujo normal de ejecución de la aplicación.

- El hecho que sea posible la utilización de demasiadas opciones diferentes de lenguajes de programación y tecnologías aumenta el costo operativo y fragmenta la organización del proceso de desarrollo de software. Una decisión apresurada en este sentido puede

conducir a problemas en las tareas de creación de nuevos servicios. Si bien no fue el caso de este trabajo, sí fue una situación potencial debido a que, por ejemplo, el proyecto cuenta con otros servicios desarrollados en diferentes lenguajes, y se podría haber optado por una nueva tecnología para el microservicio aquí presentado. Sin embargo, se decidió priorizar tiempos de desarrollo.

- Muchas veces, la falta de observabilidad dificulta la clasificación de problemas de rendimiento o fallas. La aplicación principal a la que se incorporó el nuevo microservicio ya contaba con otros servicios, y las solicitudes a estos componentes a menudo abarcan varias instancias de los mismos. Cada instancia de servicio genera un archivo de registro en un formato estandarizado.
  Con el microservicio de comunicaciones creado se generó también un nuevo archivo de registros. Teniendo en cuenta este contexto, se vuelve cada vez más compleja la tarea de examinar la información a la que los desarrolladores debemos acceder para buscar y analizar causas frente a la aparición de distintos inconvenientes. Con lo cual, comienza a ser imprescindible un servicio de registro centralizado que agrupe todos estos datos en solo un lugar, y de esta manera contar con otras ventajas como, por ejemplo, poder configurar alertas que se activen cuando ciertos mensajes aparecen en los registros.
  Por otro lado, y de manera similar, cuando la cartera de servicios aumenta, se vuelve crítico vigilar las transacciones para poder monitorear patrones y enviar alertas cuando ocurre un problema. Aquí también comienza a ser requerido un servicio de métricas para recopilar estadísticas sobre operaciones individuales.
  Un inconveniente más que se añade con una arquitectura de microservicios, y relacionado a la observabilidad, es el hecho de dificultar el rastreo distribuido de una solicitud. Ahora las solicitudes al sistema generalmente abarcarán múltiples componentes. Por ejemplo, una solicitud para crear una sesión de comunicaciones pasará por el *backend,* el microservicio de comunicaciones y el servicio de Twilio. En algunos casos, cada componente manejará una solicitud realizando una o más operaciones en múltiples servicios. En este sentido, y frente a la resolución de problemas, el rastreo de una solicitud de extremo a extremo comienza a complicarse, y origina la necesidad de disponer de un identificador único de transacción que prevalezca a lo largo de todos los mensajes en los que ésta se vea involucrada.
  Por último, cuando se implementa la arquitectura de microservicios, existe la posibilidad de que un servicio esté activo, pero no pueda manejar transacciones. Un aspecto no contemplado en el desarrollo realizado fue el de tener un punto final que se pueda utilizar para verificar el estado de la aplicación. Esta API debería poder verificar el estado del host, la conexión con otros servicios y/o infraestructura y cualquier lógica específica.

- A pesar de que los microservicios están poco vinculados, la falta de una imagen holística de todo el sistema podría ser problemática. Eventualmente, durante las tareas de desarrollo, y de manera regular, se tomaban decisiones desacertadas que luego debían ser revertidas o cambiadas en pos de adaptarse al funcionamiento esperado por la arquitectura definida (y del cual no poseía plena conciencia). Ocurrieron situaciones en las que se desconocían las interacciones y funcionamientos de determinados componentes del ecosistema de la aplicación general. Circunstancias como estas sucedieron en la mayoría de los casos por no poseer conocimiento de todos los componentes del sistema.

Si no se analizan de manera correcta cada uno de estos problemas desde el comienzo de la utilización de un enfoque de microservicios, lo mismo se puede traducir en problemas aún más graves.

Quizás una opción frente a este último planteo es comenzar de manera paulatina. Si la situación se corresponde con los inicios de la utilización de un enfoque de microservicios, tal

vez lo aconsejable es comenzar modestamente con solo uno o dos servicios, aprender de su utilización y con el tiempo y experiencia volcarse a la utilización de otros más.

6. Bibliografía

1. https://dzone.com/articles/what-are-microservices-actually
2. https://martinfowler.com/articles/microservices.html
3. https://trends.google.com/trends/explore?date=today%205-y&q=Microservices
4. NEWMAN, Sam. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015.
5. DRAGONI, Nicola, et al. Microservices: yesterday, today, and tomorrow. En Present and ulterior software engineering. Springer, Cham, 2017. p. 195-216.
6. THÖNES, Johannes. Microservices. IEEE software, 2015, vol. 32, no 1, p. 116-116.
7. FOWLER, Martin; LEWIS, James. Microservices. Viittattu, 2014, vol. 28, p. 2015.
8. ROBERT, C. Agile Software Development: Principles, Patterns, and Practices. 2003.
9. JAMSHIDI, Pooyan, et al. Microservices: The journey so far and challenges ahead. IEEE Software, 2018, vol. 35, no 3, p. 24-35.
10. https://www.twilio.com/
11. https://www.android.com/
12. www.apple.com/es/ios
13. https://angularjs.org/
14. https://www.mysql.com/
15. https://redis.io/
16. https://www.deputy.com/
17. https://cloudinary.com/
18. https://aws.amazon.com/es/s3/
19. https://aws.amazon.com/es/
20. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance
21. https://aws.amazon.com/es/route53/
22. https://aws.amazon.com/es/elasticloadbalancing/
23. https://about.gitlab.com/
24. https://aws.amazon.com/es/vpc/
25. https://aws.amazon.com/es/ec2/
26. https://aws.amazon.com/es/ecs/
27. https://aws.amazon.com/es/sqs/
28. https://aws.amazon.com/es/elasticache/redis/
29. https://www.jetbrains.com/idea/
30. https://www.atlassian.com/es/software/jira
31. https://git-scm.com/
32. https://es.wikipedia.org/wiki/DevOps
33. https://www.java.com/es/
34. https://docs.gradle.org/current/userguide/userguide.html
35. https://spring.io/projects/spring-boot
36. https://spring.io/projects/spring-cloud
37. https://aws.amazon.com/es/cli/
38. https://nginx.org/en/
39. https://www.ibm.com/cloud
40. https://jenkins.io/
41. https://docs.sonarqube.org/latest/
42. https://www.twilio.com/proxy
43. https://es.wikipedia.org/wiki/Adaptador_(patr%C3%B3n_de_dise%C3%B1o)
44. https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional
45. https://swagger.io/
46. https://www.json.org/json-en.html
47. https://github.com/FasterXML/jackson
48. https://tools.ietf.org/html/rfc7519
49. https://github.com/jwtk/jjwt

50. https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)
51. https://es.wikipedia.org/wiki/DevOps
52. BALALAIE, Armin; HEYDARNOORI, Abbas; JAMSHIDI, Pooyan. Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software, 2016, vol. 33, no 3, p. 42-52.
53. https://es.wikipedia.org/wiki/Caso_de_uso
54. https://es.wikipedia.org/wiki/Git
55. https://nvie.com/posts/a-successful-git-branching-model/
56. https://docs.gitlab.com/ee/user/project/merge_requests/
57. https://site.mockito.org/
58. https://docs.gradle.org/current/userguide/build_environment.html
59. https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto
60. https://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_dependencias
61. https://itblogsogeti.com/2015/10/29/inyeccion-de-dependencias-vs-inversion-de-control-eduard-moret-sogeti/
62. https://es.wikipedia.org/wiki/SOLID
63. https://www.getpostman.com/
64. https://es.wikipedia.org/wiki/Objeto_de_acceso_a_datos
65. https://es.wikipedia.org/wiki/Objeto_de_transferencia_de_datos
66. https://es.wikipedia.org/wiki/Manejo_de_excepciones
67. https://junit.org/junit5/
68. https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html