UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Migración de un módulo software a un microservicio en un contexto industrial

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: García Gonzálvez, Francisco

Tutor/a: Abrahao Gonzales, Silvia Mara

Cotutor/a externo: BELLOSTA CARRERAS, SANTIAGO

CURSO ACADÉMICO: 2022/2023

Resumen

Los microservicios se han convertido en los últimos años en una importante herramienta estratégica que posibilita a las empresas de software acelerar el desarrollo de aplicaciones y aumentar su agilidad. Se trata de sistemas autónomos desarrollados y desplegados de forma independiente con un objetivo concreto. Los microservicios, entre otras ventajas, permiten escalar de forma independiente de otros servicios, tienen un mantenimiento sencillo, pueden mejorar el rendimiento y permiten desarrollar sistemas tolerantes a fallos ya que el fallo en un servicio no afecta al resto del sistema. Dadas las ventajas que supone para empresas tecnológicas el uso de microservicios frente a aplicaciones más monolíticas, en este TFG se propone realizar y caracterizar el proceso de migración de un módulo funcional dentro de un sistema software empleado en un contexto industrial a un microservicio desplegado de forma independiente. Concretamente, se espera que, a partir de un programa complejo empleado en la empresa EDICOM, se genere un microservicio que contenga y mejore el servicio de uno de los módulos funcionales del sistema. Además, paralelamente al proceso de migración, el programa cliente actual del microservicio será modificado al desarrollar una nueva interfaz gráfica de usuario que posibilite el uso de los servicios migrados. Finalmente, una vez migrado el módulo funcional, se pretende realizar el despliegue del microservicio haciendo uso de la tecnología que ofrece Kubernetes. Para llevar a cabo el proyecto se empleará Spring, un framework para el desarrollo de aplicaciones escritas en Java que facilita el desarrollo de microservicios basados en comunicaciones HTTP y se utilizará Maven como herramienta para la gestión y construcción del microservicio. Para el desarrollo de la parte del programa cliente se usará JavaScript como lenguaje de programación y la biblioteca DHTMLX para la implantación de componentes de interfaz de usuario. Además, se empleará Kubernetes, una plataforma de código abierto que permitirá la automatización del despliegue, mantenimiento y gestión del microservicio. Como resultado del TFG se ha desarrollado y desplegado el microservicio en producción en EDICOM. Se espera que este proyecto genere un impacto positivo en el mantenimiento, eficiencia, escalabilidad y tolerancia a fallos del sistema aumentando, de este modo, la calidad del servicio que ofrece la empresa.

Palabras clave: desarrollo web, DevOps, despliegue, intercambio electrónico de datos, microservicio, migración

Abstract

Microservices have become an important strategic tool in enabling software companies to accelerate application development and increase agility in recent years. These are loose coupling systems developed and deployed independently with a specific objective. Among other advantages, microservices allow scaling independently of other services – they are easy to maintain, increase performance, and allow fault-tolerant systems to be developed since a failure in one service does not affect the rest of the system. Given the advantages that microservices bring for technological companies (as opposed to monolithic applications), in this TFG, we characterize and describe the migration process of a functional module within a software system used in an industrial context to a microservice deployed independently. Specifically, for a complex system used in EDICOM, we created a microservice to improve the service of one of the functional modules of the system. Moreover, parallel to the migration process, the current client program of the microservice needs to be modified by developing a new graphical user interface that enables the use of the migrated services. Finally, once the functional module has been migrated, we deployed the microservice using the technology offered by Kubernetes. To carry out this project, we used the Spring framework in order to facilitate the development of microservices based on HTTP communications written in Java. In addition, Maven has been adopted as a tool for the construction and management of the microservice. On the one hand, for the development of the part of the client program, we used JavaScript as the programming language, and the DHTMLX library to implement the user interface components. On the other hand, the open-source platform Kubernetes has been employed to allow the automation of the deployment, maintenance, and management of the microservice. As a result of this TFG, the microservice has been developed and deployed in a production environment at EDICOM. This project is expected to positively impact the system maintenance as well as its efficiency, scalability, and fault tolerance, thus increasing the quality of the service offered by the company.

Keywords: deployment, DevOps, electronic data interchange, microservice, migration, Web development

Tabla de contenidos

1. Introducción

Desde comienzos de los años 2000, con la aparición de frameworks como J2EE [1], también conocido como Java empresarial, las aplicaciones se han construido de forma más o menos monolítica independientemente de la aproximación arquitectónica seguida para la construcción del sistema (cliente-servidor, orientados a servicios, etc.). Sin embargo, con el surgimiento de sistemas basados en la nube y aplicaciones alojadas de forma remota, aparecen dificultades para el uso y desarrollo continuo de estas aplicaciones. Un problema eminente del tipo de arquitectura monolítica se manifiesta cuando un simple cambio en cualquiera de los módulos de los que consta la aplicación supone el despliegue de toda la aplicación [2], [3]. Estos inconvenientes quedan en gran parte resueltos con las arquitecturas basadas en microservicios [4]. Con estas nuevas tendencias en ingeniería del software el desarrollo de arquitecturas monolíticas tradicionales se está viendo comprometido por no ajustarse a las necesidades actuales de las empresas de tecnologías de la información.

Las arquitecturas basadas en microservicios han evolucionado enormemente estos últimos años y permiten desarrollar aplicaciones modulares físicamente separadas posibilitando la división del servicio que ofrece una organización en unidades autónomas con alto grado de cohesión y bajo acoplamiento [5]. Existen múltiples factores que han propiciado el surgimiento de estas arquitecturas basadas en microservicios a nivel industrial. Además de la aparición de nuevas tecnologías, la necesidad creciente de automatizar procesos de desarrollo o el aumento de la demanda en la agilidad y velocidad de entrega en empresas de tecnologías de la información han sido factores decisivos en el éxito de los microservicios [4]. Actualmente organizaciones como Netflix, Amazon o eBay utilizan la arquitectura de microservicios para particionar sus aplicaciones monolíticas en unidades funcionales más pequeñas [6]. Como consecuencia del éxito de estas grandes corporaciones, otras empresas han adoptado este patrón arquitectónico para refactorizar sus aplicaciones monolíticas. De hecho, la *International Data Corporation (IDC)* estimaba que antes de 2022 el 80% de las aplicaciones basadas en la nube estarían desarrolladas usando una arquitectura de microservicios [7].

Los microservicios son servicios de pequeño tamaño que se despliegan de forma independiente [8]. Pueden ser escritos en diferentes lenguajes de programación y escalar de forma independiente de otros servicios. Debido a su tamaño son más sencillos de mantener y más tolerantes a fallos puesto que el fallo en uno de los servicios no afecta a todo el sistema como ocurriría en un sistema monolítico [9]. Además, por la propia naturaleza de los microservicios parece el tipo de arquitectura más adecuada para la adopción de prácticas DevOps en aquellas organizaciones que promueven un desarrollo de software ágil. Estas prácticas combinan estrategias de desarrollo de software, mantenimiento de calidad, y operaciones llevadas a cabo por los equipos de desarrollo y calidad [10]. Tienen como objetivo reducir el tiempo de entrega de un producto y entre versiones garantizando la calidad de éste.

La arquitectura de microservicios y los procesos DevOps han estado evolucionando constantemente durante los últimos años en la industria de las tecnologías de la información. Sin embargo, la mayoría de los estudios realizados sobre la interacción entre microservicios y estrategias DevOps se han llevado a cabo a nivel académico y pocos a nivel industrial [11]. Existe una carencia de trabajos que exploren la refactorización, evolución o mantenimiento de los microservicios, así como estudios empíricos que ayuden a comprender y analizar las herramientas de microservicios en DevOps. Este trabajo pretende aportar a la literatura académica evidencias, dentro de un contexto industrial, del proceso de migración de un módulo software de una aplicación monolítica a un microservicio siguiendo un paradigma DevOps.

El trabajo se ha llevado a cabo como parte del proyecto *Proposals* en del departamento de I+D dentro de EDICOM, empresa dedicada al intercambio electrónico de datos. *Ebimap* es un servicio clave para el trabajo de los consultores dentro de la organización y, durante muchos años, ha satisfecho las necesidades de negocio con éxito. Sin embargo, en la actualidad, el volumen empresarial ha crecido enormemente. El número de consultores que hacen uso del servicio ha aumentado significativamente y, como resultado, la carga a la que está sometida la aplicación. Además, se trata de un servicio en continua evolución y expansión de manera que nuevas versiones y funcionalidades se desarrollan para satisfacer las demandas empresariales. Toda esta situación hace que el coste de la gestión y mantenimiento del servicio se vea incrementado de forma considerable.

Para mejorar la monitorización del sistema se ha propuesto hacer una revisión de la arquitectura del servicio. Concretamente, se ha planteado la migración de uno de sus módulos, el módulo de propuestas, a un microservicio independiente. Además, se propone que este nuevo microservicio incorpore nuevos requisitos solicitados por los consultores de manera que incremente la calidad del servicio. Este módulo, a nivel general, se encarga de generar una descripción detallada de un modelo de datos que los consultores de la empresa proporcionan a un determinado cliente o que el cliente proporciona y sirve de base para mapear diferentes estructuras de datos. Como resultado de la migración y desarrollo del nuevo servicio de propuestas se pretende avanzar en la transformación de la cultura organizacional hacia DevOps aumentando la calidad de los servicios que la empresa ofrece a sus clientes.

1.1. Estructura

En este trabajo de fin de grado se detalla el proceso de migración de un módulo software de una aplicación monolítica a microservicio, así como las implicaciones que tiene para las estrategias DevOps desde un punto de vista industrial. La memoria del trabajo está estructurada como sigue.

En el presente capítulo introductorio se recogen diversos aspectos que pretenden enmarcar el contexto en el que el proyecto se ha llevado a cabo, la motivación y objetivos que han impulsado el desarrollo del proyecto, el impacto esperado que supondrá la realización de éste y la metodología que se ha seguido durante todo el proceso. El segundo capítulo consiste fundamentalmente en una revisión bibliográfica donde se abordará el estado del arte en el desarrollo y migración de microservicios, así como su interacción con las estrategias DevOps. En el tercer capítulo tratamos de describir el marco industrial donde se ha llevado a cabo el proyecto profundizando, además, en los aspectos funcionales del servicio. En el cuarto capítulo abordamos el análisis de los problemas y requisitos que nos encontramos a la hora de enfrentar un proceso de migración y desarrollo del microservicio. Se identificarán estos problemas y requisitos planteados para finalmente seleccionar la mejor solución. El quinto capítulo especifica varios aspectos del diseño de la solución como la tecnología empleada durante la realización del proyecto, la arquitectura propuesta para el desarrollo del servicio y el diseño detallado del sistema. A continuación, en el sexto capítulo, se describe el desarrollo de la solución propuesta ahondando en aquellas partes del sistema que resultan más relevantes a nivel de microservicios. En el séptimo capítulo nos centramos en la implantación del sistema y en las tecnologías usadas principalmente para el despliegue del microservicio. En el octavo capítulo se presentan los tipos de pruebas realizadas en la verificación del nuevo microservicio y las herramientas utilizadas para su implementación. El noveno capítulo concluye la memoria del TFG discutiendo sobre la aportación del presente trabajo tanto a nivel industrial como académico. En el décimo capítulo se presentan algunas ideas sobre trabajos futuros que podrían ampliar el conocimiento del área de estudio. En el decimoprimer capítulo se detallan todas las referencias bibliográficas empleadas en la elaboración de la memoria y, finalmente, en el duodécimo capítulo se recogen los anexos del TFG, incluyendo los objetivos de desarrollo sostenible.

1.2. Motivación

En la actualidad las arquitecturas basadas en microservicios han incrementado enormemente su popularidad y están siendo ampliamente adoptadas tanto por pequeñas como grandes organizaciones. Entre los beneficios que las empresas obtienen de este tipo de arquitecturas destacan la mejora en el mantenimiento, escalabilidad y el soporte de los microservicios en las estrategias DevOps [12].

Sin embargo, pese al incremento en la popularidad de los microservicios y las estrategias DevOps en la industria, Taibi et al. [13] señalan la escasez de estudios empíricos en un contexto industrial en el desarrollo de sistemas basados en microservicios. De la misma manera, Waseem et al. [11] encontraron que solo el 21,27% de los estudios que analizaron tenían un autor de la industria. Además, señalan la ausencia en la literatura científica de actividades de ingeniería de requisitos relacionada con los microservicios, así como actividades de arquitectura de microservicios en un contexto DevOps. También

evidencian desde el punto de vista de las estrategias DevOps una ausencia de trabajos que exploren la refactorización, evolución o mantenimiento de los microservicios. Finalmente, los mismos autores apuntan que también es necesario estudios empíricos que ayuden a entender y evaluar las herramientas de microservicios en DevOps. El campo de la migración de los microservicios y refactorizaciones arquitectónicas está aún por investigar [14].

Con este proyecto fin de grado se pretende documentar el proceso de migración y desarrollo dentro de un contexto industrial intentando cubrir algunas carencias observadas en la literatura científica. Con el fin de incorporar a estudios sobre microservicios trabajadores de ámbitos empresariales que reflejen sus experiencias en el desarrollo de estos servicios, nos hemos fundamentado en la propuesta realizada por el departamento de I+D de la empresa EDICOM a sus equipos de desarrollo para la migración a microservicios de algunos de los servicios integrados en sus aplicaciones monolíticas. Dada la relevancia a nivel organizacional que suponen las refactorizaciones de los servicios que ofrece la empresa, se ha decidido realizar el proceso de migración de uno de estos microservicios.

1.3. Objetivos

Este trabajo se centra en dos objetivos principales clasificables en función del punto de vista en el que se ha abordado el proyecto:

- Desde un punto de vista práctico y empresarial este proyecto pretende la migración de un módulo software de una aplicación monolítica a un microservicio. Este nuevo microservicio, además, incorporará nuevas funcionalidades demandadas por los usuarios del servicio. Tras la implementación de este microservicio se espera conseguir una mejora en la calidad del servicio y un mayor grado de implantación de estrategias DevOps en los procesos que ofrece la empresa.

- Desde un punto de vista académico, este proyecto persigue contribuir a la literatura científica con información empírica sobre el desarrollo y migración de microservicios en contextos industriales. También intenta explorar herramientas y procesos que ayuden a evolucionar hacia estrategias DevOps a nivel organizacional.

1.4. Impacto esperado

La migración del módulo software a microservicio se espera que genere un impacto sobre diversos aspectos:

- Independencia del servicio de Propuestas: El microservicio, al ser una entidad autónoma, permitirá ser ejecutado de forma independiente al resto del sistema. De esta manera, ante cualquier cambio realizado en el servicio de propuestas, ya no será necesario desplegar todo el proyecto monolítico.

- Experimentación e innovación: El microservicio, al ser una entidad más simple y de menor tamaño, posibilitará experimentar e innovar nuevas funcionalidades en el microservicio con una inversión mucho menor evitando efectos colaterales en la aplicación monolítica.

- Escalabilidad: El microservicio podrá escalar de forma independiente al resto de la aplicación.

- Crecimiento continuo: El desarrollo del microservicio permitirá un crecimiento paulatino en las funciones del servicio. Mientras que inicialmente se migrarían aquellas funciones básicas, posteriormente se podrían desarrollar funciones adicionales demandadas por los consultores de la empresa.

- Reduce la deuda técnica*: Como consecuencia del menor tamaño del microservicio, será más sencillo actualizar o modificar la tecnología usada sin tener que hacerlo en toda la aplicación monolítica. Además, una arquitectura escalable y flexible contribuye a la sostenibilidad del servicio en el tiempo evitando así la deuda técnica.

- Posibilita procesos DevOps: La empresa tiene establecida la filosofía de las estrategias DevOps en sus productos porque posibilitan una mayor agilidad en el desarrollo, velocidad de entrega del producto, testeo automático, suministro automático de una infraestructura y despliegue automático. Sin embargo, la transición hacia esta estrategia es paulatina. La migración del módulo software a microservicio independiente supone un paso más en el objetivo de alcanzar la completitud en la adquisición de estas estrategias.

* Deuda técnica: El coste de corrección de sistemas de software que a largo plazo resultan problemáticos debido a la baja calidad del código

- Documentar el proceso de migración: El presente trabajo supone un aporte a la literatura científica sobre el proceso de migración de aplicaciones a microservicios en contextos industriales. Al documentar el proceso es posible replicarlo en futuras migraciones empresariales. Además, se describen algunas lecciones aprendidas que pueden ser de interés para investigadores y profesionales del área de Ingeniería del Software.

1.5. Metodología

La metodología que se ha seguido en el desarrollo del proyecto es una metodología meramente ágil. A diferencia de metodologías más tradicionales, las metodologías ágiles consideran que los requisitos pueden cambiar en cualquier fase del ciclo de desarrollo de un producto de manera que dicho producto se encuentra en constante evolución. Estas metodologías contemplan todo el ciclo de desarrollo de un producto identificando los posibles problemas en las primeras etapas y minimizando el impacto negativo que dichos problemas puedan tener en la totalidad del proyecto. Las metodologías ágiles se basan en los cuatro principios del manifiesto ágil propuestos en 2001 [15]:

1. Individuos e interacciones sobre procesos y herramientas
2. Software funcional sobre documentación extensiva
3. Colaboración con el cliente sobre negociación contractual
4. Respuesta ante el cambio sobre seguir un plan

Aunque todas las metodologías ágiles siguen estos principios, podemos encontrar diferentes técnicas. Una de las técnicas más extendidas es el proceso Scrum que hace uso de buenas prácticas para trabajar colaborativamente en equipo para obtener el mejor resultado posible de un proyecto [16]. Scrum es la principal técnica que se ha empleado en la realización de este proyecto y estos son algunos de los principios básicos se han seguido:

- Desarrollo dividido en Sprints
- Backlog del producto y backlog del sprint
- Reuniones de planificación
- Reuniones diarias
- Reuniones de retrospectiva
- Estimación de tareas
- Análisis de la gráfica Burndown

- Presencia de Scrum Master
- Grupo multifuncional

En EDICOM, el departamento de I+D está dividido en equipos de pocos individuos y cada equipo está liderado por un Scrum Master. Este proyecto ha sido desarrollado dentro del equipo de Web Service (WS) dirigido por Oscar Albert Arcas. Se trata de un grupo multifuncional compuesto por seis desarrolladores full stack.

De acuerdo con la técnica del Scrum, el trabajo que se debe realizar sobre el proyecto está divido en iteraciones temporales denominadas sprints y que, en el caso de EDICOM, tienen una duración de dos semanas. La dinámica de cada sprint dentro de la empresa se realiza de la siguiente manera. Al comienzo del sprint se realiza una reunión de planificación en la que se decide qué tarea se mueve del backlog a dicho sprint. Además, se analiza la tarea y se estima el tiempo que llevará realizarla. Una vez finalizada la reunión de planificación el objetivo del sprint queda definido para todos los miembros del equipo. Cada día, al comienzo de la jornada, se realiza una reunión diaria moderada por el Scrum Master que no suele durar más de 15 minutos. En esta reunión todos los miembros del equipo exponen de manera breve y concisa el trabajo realizado el día anterior y algunos de los problemas encontrados en la realización de las tareas. Además, para visualizar el estado del sprint se recurre a la gráfica burndown donde se muestra el trabajo pendiente dentro del sprint. Al final del sprint el Scrum Master del equipo realiza una reunión de retrospectiva donde se discute cómo ha ido el sprint, los problemas encontrados y los logros conseguidos.

Para dejar constancia del trabajo realizado y llevar un seguimiento del estado del sprint y las tareas individuales se emplea una herramienta interna de la empresa con una funcionalidad similar a la de un tablero Scrum. En esta herramienta es posible, entre otras muchas acciones, imputar el tiempo dedicado a cada tarea, así como indicar si la tarea está en progreso, se ha enviado al departamento de calidad para realizar testeo de esta o simplemente la tarea se ha completado sin necesidad de testeo.

2. Estado del arte

En la actualidad existe un incremento en el desarrollo de aplicaciones basadas en un conjunto de servicios independientes, autónomos y escalables. Cada uno de estos servicios, denominados microservicios, son desarrollados y desplegados de forma independiente [8]. Esta aproximación en la construcción de aplicaciones se contrapone al desarrollo tradicional de sistemas con arquitecturas monolíticas. Las aplicaciones monolíticas, al contrario que los microservicios, son sistemas construidos como una única unidad e indivisible. En la figura 1 se muestra la estructura de una aplicación monolítica típicamente organizada en una capa de presentación, una capa de negocio y una capa de datos. Cada uno de los módulos A, B y C, a su vez, presentan una funcionalidad diferente

para la aplicación. En esta arquitectura, cada una de las capas en las que se estructura la aplicación hay representación de los tres módulos. En este tipo de aplicaciones las capas normalmente están físicamente separadas mientras que los diferentes módulos son desarrollados dentro de cada capa.

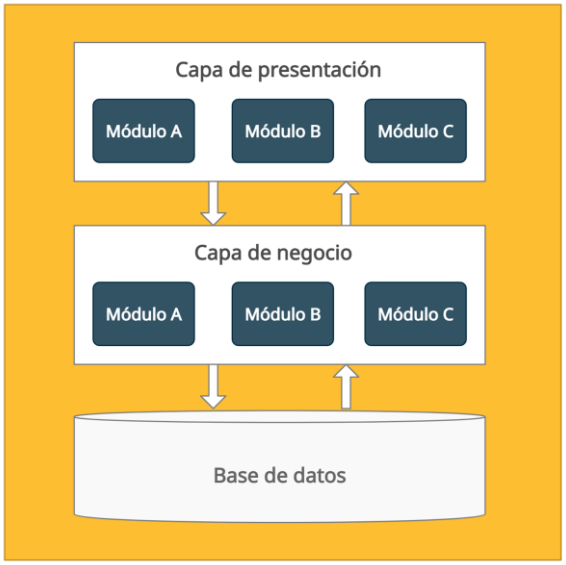

Figura 1. Estructura típica de una arquitectura monolítica

En el caso de una arquitectura basada en microservicios los límites se invierten. Mientras que una aplicación monolítica se implementa como una sola entidad unificada, una arquitectura basada en microservicios descompone el sistema en una colección de unidades independientes de menor tamaño (ver Figura 2). En esta arquitectura cada uno de los microservicios presenta su propia capa de presentación, capa de negocio y capa de datos de manera que los cambios que se realizan en un microservicio no afectan a otro. Cada uno de los microservicios lleva a cabo un determinado proceso como un servicio independiente de manera que tienen su propia lógica y bases de datos. La comunicación entre microservicios se lleva a cabo mediante protocolos ligeros como HTTP y REST o protocolos de mensajería como JMS o AMQP.

Figura 2. Estructura típica de una arquitectura basada en microservicios

Netflix, Uber, Ebay, Airbnb, Amazon, LinkedIn, Twitter, Nike, son algunas de las empresas que han transformado sus aplicaciones tradicionales monolíticas en una arquitectura basada en servicios [6], [12]. Esta transformación viene motivada con el propósito de superar las limitaciones del desarrollo tradicional de aplicaciones monolíticas. Algunas de estas limitaciones son:

- Elevada complejidad en el código base que hace el mantenimiento cada vez más complicado, así como un despliegue más costoso. Desde el punto de vista industrial, las aplicaciones monolíticas suelen formar parte de los activos de las empresas requiriendo mantenimiento durante muchos años. Estas aplicaciones, a lo largo de los años, han crecido aumentando su complejidad y han acumulado deuda técnica que hacen al sistema difícil de mantener con un esfuerzo razonable.

- Las aplicaciones monolíticas escalan clonando toda la aplicación como un conjunto sin tener la posibilidad de que sólo escalen los módulos de la aplicación que presentan cuello de botella. Este tipo de aplicaciones escalan duplicando instancias de toda la aplicación.

- Las bases de datos suelen ser grandes en aplicaciones monolíticas y si no es replicada puede comprometer a la disponibilidad de la aplicación.

Actualmente la arquitectura de microservicios ha incrementado enormemente su popularidad ya que permite un diseño de aplicaciones donde cada módulo funcional puede ser desarrollado y desplegado de forma independiente haciendo el mantenimiento mucho más sencillo que en aplicaciones monolíticas [17]–[20]. Esta arquitectura permite el escalado independiente de diferentes partes de la aplicación [19]. Asimismo, cada uno de los microservicios puede ser desarrollado con diferentes lenguajes de programación o utilizar diferentes tecnologías, se logra reducir el tiempo para su salida al mercado y una

mejor compresibilidad del código base. Los microservicios, además, son integrados de forma sencilla en el desarrollo de aplicaciones a través de mecanismos de comunicación ligeros, normalmente RESTFUL. Una ventaja adicional sería la independencia de los microservicios, entendida como la posibilidad de establecer para cada servicio su propio entorno de ejecución (arquitectura, plataforma, etc.), ciclo de desarrollo y que su despliegue y operación no dependa de los demás microservicios que componen el sistema.

Aunque se considera que las arquitecturas basadas en microservicios son altamente beneficiosas, particularmente en el caso de grandes compañías, también presentan algunas desventajas. Entre los mayores inconvenientes la mayoría de los estudios destacan la complejidad intrínseca que se alcanza dentro de este tipo de arquitecturas, lo que conlleva una mayor dificultad para monitorizar todo el sistema [21]. Además, la construcción de microservicios desde cero es una tarea muy costosa económicamente que requiere una elevada inversión de tiempo. Este impacto económico también es atribuible a las migraciones de una aplicación monolítica a microservicios [22]. Por tanto, la decisión de generar microservicios a partir de aplicaciones monolíticas debe estar fundamentada para lograr unos beneficios esperados. El proceso de migración no solo supone una inversión de tiempo y dinero, sino que, para los ingenieros de software, una descomposición de un sistema en servicios con la adecuada granularidad es un gran desafío para garantizar el éxito de la migración. Esta descomposición del sistema en pequeños servicios y su monitorización son, por tanto, algunos factores para tener en cuenta a la hora de la migración [20].

Para evitar fracasar en este proceso de migración de un sistema a microservicios es necesaria la puesta en marcha de una metodología. Debido a la diversidad de escenarios en la industria de las tecnologías de la información, esta metodología no es estricta y debe adaptarse a las necesidades de la empresa. Con el fin de planificar adecuadamente el proceso de migración, Balaie et al. [20] definen diferentes patrones de migración dependiendo del problema a resolver con la construcción del microservicio y los posibles desafíos a los que se enfrenta este patrón. Zdum et al. [23] sugieren que es una buena práctica iniciar un proyecto con una aplicación monolítica y, cuando el sistema lo requiera, empezar al individualizar servicios a partir de la aplicación monolítica. Este planteamiento presenta la contraparte de que muchas aplicaciones monolíticas han sido desarrolladas sin planificar una futura migración y, en estos casos, las aplicaciones pueden necesitar un rediseño sustancial de su arquitectura.

Otra aproximación para el proceso de migración está propuesta por Richardson [24] y consiste en seleccionar qué módulos del sistema monolítico se van a transformar en microservicios y gradualmente aplicar dicha transformación. El código de la aplicación monolítica es adaptado a realizar llamadas al microservicio a través de un servicio REST. Normalmente este tipo de migración supone cambios significativos en el código de la aplicación monolítica. Eisele [25] propone un proceso de migración donde el microservicio es desplegado de forma paralela a la aplicación monolítica. A diferencia de la aproximación anterior, en ésta no hay modificaciones en el código de la aplicación

monolítica, pero existe un proxy o balanceador de carga que decide si la petición va hacia el módulo de la aplicación monolítica o es redireccionado al microservicio creado.

En definitiva, las migraciones no son tareas sencillas especialmente si la aplicación ya está siendo usada en producción en las actividades diarias de la empresa. La mayoría de estas migraciones requieren cambios de código invasivos en las aplicaciones monolíticas, así como interrupciones planeadas cuando se liberan nuevas versiones [24], [25]. Además, normalmente aparecen problemas en las nuevas implementaciones, por lo que es necesario esfuerzo para alcanzar una versión previa estable.

En cuanto a los diferentes aspectos de un proyecto que se debe tener en cuenta a la hora de realizar una migración, Freire et al. [26] destacan los siguientes:

1. La granularidad de la refactorización. Esto es, el nivel de necesidad de refactorización del código de la aplicación monolítica que va a ser migrada a microservicio. Se debe plantear si la unidad de código que se migra será parcial o completamente refactorizada.

2. La estrategia de migración de código. Tras seleccionar la unidad de código que será migrada es necesario evaluar las diferentes estrategias que se van a llevar a cabo para la modificación del código. Esta alteración del código se puede llevar de una manera más o menos invasiva dependiendo de cuánto código es necesario modificar en la aplicación monolítica.

3. La migración de las dependencias. Se trata de evaluar las dependencias que existían antes y después de la migración del código.

4. La inactividad de la aplicación monolítica. Se debe valorar si, durante el proceso de migración, es posible o no la interrupción de la aplicación monolítica para llevar a cabo la migración.

5.La migración de los datos. La migración de los datos se convierte en otro requisito ya que el microservicio podría ser responsable de su propia base de datos.

6. Capacidad para revertir el proceso de migración. Cabe la posibilidad que la migración presente problemas, aunque sean temporales, por lo que los desarrolladores deben contemplar la posibilidad de que se tenga que volver a la aplicación monolítica. La habilidad para realizar este retorno es un requisito deseable en cualquier migración.

2.1. Microservicios y DevOps

El paradigma DevOps comprende una serie de principios y prácticas que integran el desarrollo de software y operaciones de las tecnologías de la información que permiten mantener la calidad del producto al tiempo que aumenta la frecuencia de entrega de éste

[27]. Se trata de una serie de prácticas para el desarrollo, testing y despliegue rápido del software al promover la colaboración entre desarrolladores, testers y operadores [28].

Muchos profesionales e investigadores afirman que la arquitectura de microservicios acoge de forma natural las estrategias DevOps [29]. Por ejemplo, las prácticas DevOps y la arquitectura en microservicios promueven la idea de la descomposición de grandes componentes en pequeñas piezas para poder gestionarlas mejor a través de equipos funcionales [30]. La adopción de estrategias DevOps basadas en microservicios genera una productividad adicional al posibilitar la integración continua, la automatización de tests, un rápido despliegue y un ambiente sincronizado y flexible. Chen et al [31] propone que las estrategias DevOps deben asegurar cuatro atributos de calidad: disponibilidad, modificabilidad, rendimiento y testeabilidad. Por ejemplo, sugiere que la disponibilidad sea revisada monitorizando el sistema detectando excepciones, reconfigurando clusters de forma automática en caso de fallos o haciendo roll back en servicios desplegados cuando sea necesario. La arquitectura de microservicios usada con DevOps puede, además, traer otros beneficios como fiabilidad y escalabilidad del sistema, realizar frecuentes versiones, gestión de equipos descentralizados para controlar el desarrollo de la aplicación [32]. Además, proporciona herramientas que ayudan al proceso de codificación continua, testing, empaquetamiento y monitorización de sistemas basados en microservicios.

En última instancia, los microservicios y los procesos DevOps permiten a las empresas ganar en agilidad y eficiencia operacional [10]. Las organizaciones deben, además, adaptarse a trabajar bajo el paradigma DevOps y su combinación con los microservicios. Algunos estudios [33] sugieren que las organizaciones deben transformar su estructura creando equipos multifuncionales, enseñando a sus trabajadores nuevas habilidades relacionadas con los procesos DevOps, incentivando el trabajo en equipo, etc.

Uno de los ejes fundamentales en la implementación de DevOps es la automatización, por esta razón en el mercado se encuentran diversas prácticas y herramientas que apoyan las diferentes fases del proceso de ingeniería de software continua (ver Figura 3). Por ejemplo, la integración continua (CI) y la entrega continua (CD) son prácticas frecuentemente usadas en la creación de software bajo el paradigma DevOps. La integración continua permite a los desarrolladores incorporar los cambios realizados al código base tan pronto como sea posible construyéndose el proyecto y validando la calidad del código a través de los test de integración [34]. La entrega continua es la fase siguiente a la integración continua y asegura que el proyecto esté preparado para ser usado en un entorno de desarrollo, testing o producción [34].

La entrega continua, por tanto, está estrechamente ligada al concepto de despliegue continuo. Todo este proceso de desarrollo, testeo y despliegue de nuevas versiones del producto software se puede automatizar a través de lo que, en ingeniería del software, se denomina pipeline de CI/CD. El uso de este pipeline supone numerosas ventajas en el desarrollo de software tales como un incremento en la productividad de los desarrolladores, aumento de la entrega de productos o detección temprana de errores en

el producto [35]. Las estrategias DevOps hacen uso de herramientas como GitLab (ver Figura 3) que ayudan a los desarrolladores a configurar estos pipelines. De manera que, cuando un desarrollador, realiza un cambio en el código del repositorio, Gitlab ejecuta el pipeline CI/CD que consiste en una serie de trabajos. Por término trabajo se entiende a la actividad que se lleva a cabo en el pipeline, por ejemplo, la ejecución de tests unitarios. Si todos los trabajos del pipeline se llevan a cabo con éxito, entonces el proyecto es desplegado. Por tanto, a medida que pasamos de proyectos monolíticos a aplicaciones basadas en microservicios la necesidad de un pipeline CI/CD se convierte en un requisito de gran importancia a la hora de gestionar el despliegue individual de cada uno de los microservicios.

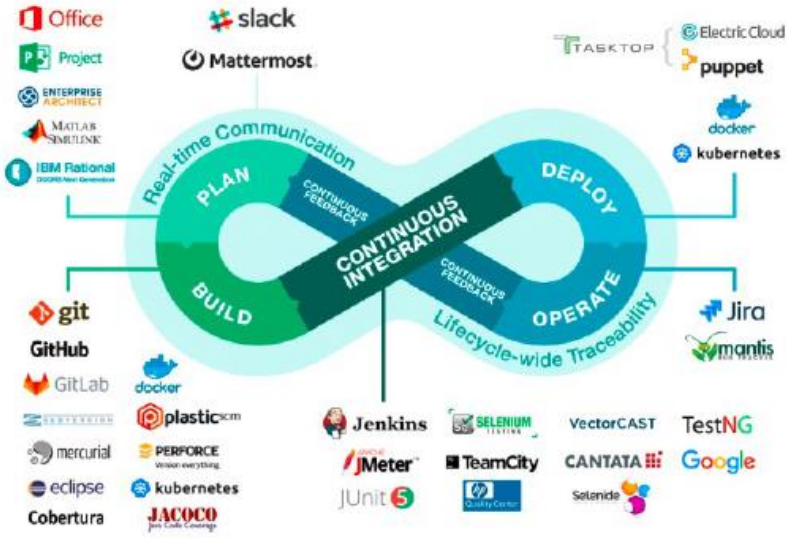

Figura 3. Proceso de ingeniería de software continua y herramientas DevOps (Fuente: [36])

2.2. Crítica al estado del arte

A pesar de la gran popularidad de la que gozan hoy en día los microservicios y la gran cantidad de literatura científica que se genera actualmente en torno a estos, la mayoría de los trabajos basados en microservicios están fundamentados en sus aspectos teóricos, centrándose en el estudio de sus ventajas y problemas potenciales. Sin embargo, desde un punto de vista práctico, es posible apreciar una ausencia de evidencia empírica en muchos de estos trabajos. Considerando contextos industriales, existe una carencia aparente de publicaciones que impliquen actividades de ingeniería del software para la creación y desarrollo de sistemas basados en microservicios. Se hallan muchos menos trabajos si consideramos el terreno de las refactorizaciones de aplicaciones monolíticas a microservicios. De hecho, Pahl y Jamshidi [14] consideran el estudio de las migraciones como parte de futuras tendencias dentro del área de los microservicios. Cuando

examinamos los microservicios desde una perspectiva DevOps se manifiesta que, a nivel industrial, es un dominio moderno y en continua expansión. En este sentido, se detecta que son necesarios trabajos que analicen y evalúen las herramientas y actividades implicadas en la refactorización y mantenimiento de los microservicios en contextos empresariales.

2.3. Propuesta de trabajo

La generación de un servicio de Propuestas independiente se ha abordado como una refactorización incremental de la aplicación monolítica. Esto es, se ha planteado un proceso de migración en el que se identifican y se seleccionan los límites del módulo software que es candidato a microservicio para, gradualmente, ir aplicando su transformación. Esta aproximación en el proceso de migración es similar a la propuesta por Richardson [24] y es esquematizada en la Figura 4.

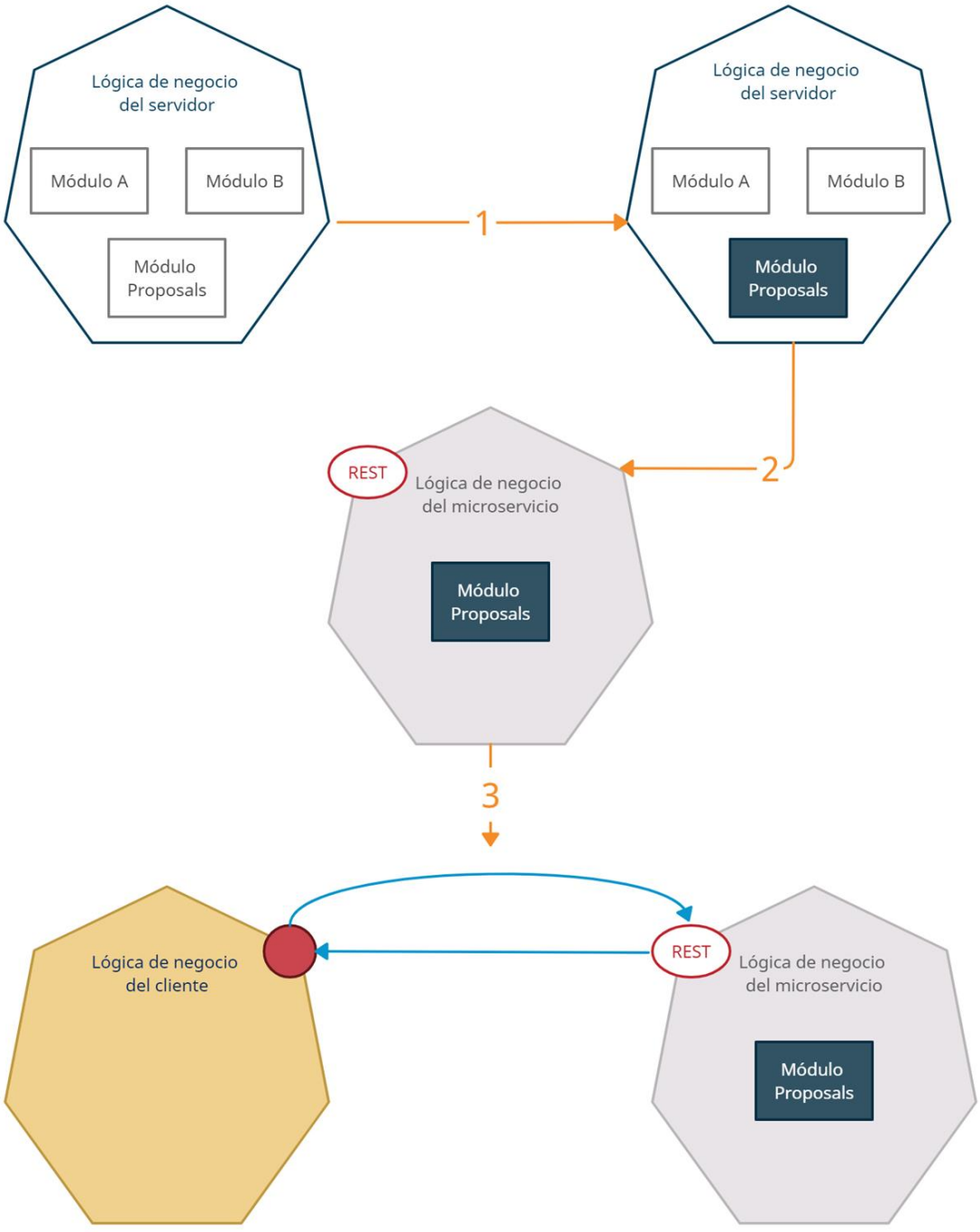

Figura 4. Proceso de migración a microservicio planteado para el sistema de Propuestas. En una primera fase se identifican los requisitos que el microservicio debe incluir. En la segunda fase el módulo funcional se aísla y se estructura conformando el microservicio. Finalmente, en una tercera fase la lógica de negocio del cliente se direcciona al microservicio para hacer uso de los nuevos servicios.

En una primera fase se determinaron y sintetizaron todos los requisitos que el microservicio debía cumplir. Dentro de estos requisitos se consideró la funcionalidad básica que el servicio ya incluye dentro de la aplicación monolítica, así como nuevos requisitos propuestos por primera vez para el microservicio. Una vez definida la

funcionalidad del módulo software, se analizó la forma de aislar el módulo software y como estaría estructurado en términos de API. Finalmente, tras el proceso de migración la funcionalidad del módulo fue replicada y ampliada en el nuevo microservicio aplicando, en un determinado punto de su desarrollo, una redirección a dicho microservicio.

3. Contexto del proyecto

Hoy en día la comunicación entre individuos, empresas y entidades gubernamentales se realiza mayoritariamente a través de sistemas informáticos. El intercambio de información entre diferentes organismos relacionados con los negocios se realiza a través de intercambio electrónico de datos o EDI (*Electronic Data Interchange*) y consiste en el intercambio de datos entre computadores con un patrón estandarizado [37]. EDI, por tanto, permite a las empresas intercambiar documentos comerciales en un contexto electrónico sin necesidad de participación humana durante el proceso. Además, el intercambio de datos es una forma más fácil, rápida y segura de intercambio de información entre empresas con un coste más bajo posibilitando un mayor desarrollo y crecimiento económico de las empresas implicadas.

Los principales componentes del intercambio electrónico de datos son [38]:

1. Los estándares del intercambio electrónico de datos

   Los estándares permiten codificar los datos para simplificar la trasferencia electrónica. En este sentido los interlocutores que participan en el intercambio electrónico de datos deben acordar la estructura de la información que se va a intercambiar y, además, esta estructura también debe ser formulada en términos de los estándares a nivel internacional. Los estándares usados en el comercio electrónico son numerosos y variados y normalmente cada país tiene su propio estándar. Entre los principales estándares podemos resaltar el ASC X12, que es el principal estándar para el intercambio electrónico de datos usado en Estados Unidos y EDIFACT, que es el acrónimo en inglés de Intercambio Electrónico de Datos para la Administración, Comercio y Transporte y es usado principalmente en Europa y Asia.

2. El software para el intercambio electrónico de datos

   El software para el intercambio electrónico de datos permite la transformación de datos con una estructura específica de un interlocutor emisor a datos con la estructura de datos del interlocutor receptor. El principal papel que juega el software en el intercambio de datos es la conversión del formato de datos y el envío de los mensajes.

3. Las redes de comunicación para el intercambio electrónico de datos

   La red de valor añadido o VAN se trata de una red de telecomunicaciones que está diseñada para la transmisión de los datos que permite reducir costes y un el desarrollo de metas estratégicas.

EDICOM es una de las principales empresas multinacionales especializadas en el intercambio electrónico de datos. Una pieza clave en el software usado por EDICOM para el intercambio electrónico de datos es la herramienta *Ebimap* que posibilita el mapeado de datos entre dos modelos de datos. Esto es, permite establecer correspondencias entre un modelo de datos origen y un modelo de datos destino. En el proceso de mapeado los datos del modelo origen sufren un proceso de transformación para adaptarlos al modelo de datos destino. Para realizar el mapeado, la herramienta *Ebimap* puede acceder a ficheros de texto para transformar los datos desde una aplicación origen y almacenarlos en una aplicación destino. Además, proporciona una representación gráfica de los modelos de datos y su mapeado facilitando la integración de datos entre diferentes sistemas. El objetivo de esta herramienta es que se garantice una transformación de datos de forma automática, estandarizada y de calidad.

Un modelo de datos en el servicio *Ebimap* se conoce como interfaz. Una interfaz consta de una agrupación jerárquica de distintos registros donde cada registro, a su vez, está formado por una serie de campos. Esto es, cada una de las unidades lógicas del modelo de datos es un registro y cada registro contiene una serie de campos secuenciales. Estos campos constituyen la unidad mínima de información donde se incluye el valor de un dato para el que se definen ciertas propiedades que debe cumplir. La pantalla de creación y edición de una interfaz en la aplicación puede visualizarse en el Anexo 11.1 *Pantalla de edición de una interfaz*

Un concepto importante es el de IEAD, que define el modo de acceso a los datos, esto es, el modelo de sintaxis que tendrá la interfaz. A partir de esta sintaxis la aplicación es capaz de implementar las interfaces. Entre todos los modelos de sintaxis que *Ebimap* es capaz de reconocer podemos destacar el IEAD LOT, IEAD TXT y el IEAD XML. Estos IEADs determinan el tipo de archivo que la herramienta reconoce como origen de los datos. El modelo tipo texto indica que el origen de los datos es un archivo de texto plano. LOT es un formato propio de la aplicación donde el usuario no especifica un símbolo separador. Finalmente, en el caso de XML, el origen de los datos viene en un archivo de tipo XML.

Ebimap, además de su principal función principal como mapeador, presenta otras funciones que hacen uso de los modelos de datos o interfaces. Una de las funciones importantes de la aplicación es el sistema de Propuestas que se encarga de generar y gestionar propuestas. Una propuesta es una descripción detallada de la información que proporcionaremos a un determinado cliente o que el cliente nos proporcionará. Cabe resaltar que, actualmente, a nivel estructural, una propuesta es un objeto interfaz

modificado para la generación de información apropiada para el cliente. El servicio de Propuestas permite de esta manera establecer con un nuevo interlocutor cómo se debe generar la información en un modelo de datos. Por ejemplo, imaginemos que un nuevo cliente X quiere establecer una comunicación mediante facturas con un determinado interlocutor. Como este interlocutor tiene su propia guía para las facturas debo generar una propuesta para el nuevo cliente que exprese como se debería realizar el mapeado de los datos con el interlocutor. A modo general, la aplicación permite que, a partir de un modelo genérico de documento, generar una propuesta de dicho documento para un determinado interlocutor que será entregado a otro interlocutor para definir cómo será el mapeado de los datos entre ambos interlocutores. Con el servicio de Propuestas se consigue una mayor agilidad para la generación de propuestas automatizando el marcado de propuestas para un determinado interlocutor. Este servicio, además, permite estandarizar y uniformizar el proceso de generación de propuestas por parte de los consultores con la consiguiente mejora en la calidad de las propuestas ofrecidas a los clientes.

El sistema de Propuestas tiene diversas funcionalidades. La principal es la generación del documento de propuesta en un formato odt (ver anexo 11.2 Documento odt de las propuestas). Otras funcionalidades destacables son la generación de ejemplos de datos con la estructura de la propuesta, la validación de ficheros de datos según la estructura de la propuesta, la exportación/importación de una propuesta a partir de un fichero csv, la comparación de propuestas y la fusión de éstas.

A nivel arquitectónico el servicio Ebimap está constituido por dos entidades diferenciadas, *Ebimap Server* y *Ebimap Web*. *Ebimap Server* es un proyecto puramente de backend de más de 250.000 líneas de código. Es una aplicación escrita en Java que ofrece acceso a un repositorio y contiene la mayor parte de la lógica de la herramienta *Ebimap*. *Ebimap Web*, por otro lado, es una aplicación web que contiene más de 100.000 líneas de código y está compuesta por un backend y un frontend. El backend está escrito en Java y hace uso del framework Spring para facilitar el desarrollo de la aplicación y Maven para la gestión y construcción del proyecto. El backend del *Ebimap Web* invoca, a su vez, los servicios del *Ebimap Server* a través de SOAP. El frontend de *Ebimap Web* está desarrollado en JavaScript puro y hace uso de la librería DHTMLX para la construcción de la interfaz gráfica de usuario de la aplicación. El módulo que procesa toda la lógica asociada propuestas se localiza dentro del proyecto *Ebimap Server*. *Ebimap Web* es el proyecto encargado de implementar la interfaz gráfica de usuario de las propuestas y gestionar las llamadas que se realizan al *Ebimap Server*.

Con el objetivo de mejorar el servicio de Propuestas se ha planteado la migración del módulo funcional de propuestas desde *Ebimap Server* a un microservicio independiente. Además de los beneficios intrínsecos que obtendremos del propio microservicio al facilitar los procesos DevOps, también se favorece la innovación de nuevas funcionalidades en el servicio de Propuestas evitando posibles efectos secundarios sobre el resto de los servicios del *Ebimap*.

Teniendo en cuenta esta situación el departamento de I+D ha trabajado juntamente con el departamento de consultoría para poder, no solo mejorar las funcionalidades ya existentes en el servicio de Propuestas, sino que también incorporar nuevos requisitos funcionales que aumenten la calidad del servicio. Entre los requisitos más destacables que se proponen para el nuevo servicio de Propuestas es su independencia con respecto a las interfaces. El nuevo objeto propuesta ya no se trata de una extensión de las interfaces, sino que es un nuevo objeto denominado *Proposal Generic Specification* o PGS. En los PGS, a diferencia de las propuestas basadas en interfaces donde se definía el modo de acceso a los datos, se produce una independencia del formato. Se abstrae la propuesta del modelo de sintaxis, esto es, si el origen de los datos es un IEAD LOT, IEAD TXT o IEAD XML.

Otro requisito importante es la compatibilidad del microservicio de Propuestas con *MyEdicom*. *MyEdicom* es un área privada para clientes y usuarios donde pueden monitorizar sus proyectos. Con el nuevo servicio de Propuestas se pretende que el propio cliente pueda, de forma sencilla, acceder a la propuesta y actualizar algunas propiedades de los campos como “Situación en” o “Condición en” con información perteneciente al cliente. Para llevar a cabo este punto se ha propuesto desarrollar, además, un nuevo tipo de objeto llamado *Proposal Client Specification* o PCS, que facilitará la integración del servicio de Propuestas con *MyEdicom*.

Otro requisito relevante es la adaptación del sistema de internacionalización para las nuevas propuestas. Las propuestas tienen como objetivo definir cómo será el mapeado de los datos entre interlocutores y cada interlocutor posee su propio idioma. Por tanto, independientemente de los datos que se almacenen en cada uno de los PGS, el documento de la propuesta debe poder ser traducido al idioma que se desee para facilitar al cliente la comprensión del informe.

Además, cabe señalar que, simultáneamente al proceso de migración y mejora del microservicio *Proposals*, el proyecto *Ebimap Web* debe ser adaptado para trabajar con el nuevo servicio de Propuestas. En este sentido, se propone crear una nueva interfaz gráfica de usuario tanto para los objetos PGS como PCS de manera que el consultor pueda gestionar la información relativa a las propuestas de forma sencilla y familiar. Por último, cabe mencionar que, aunque en nuestro equipo de trabajo se han desarrollado las llamadas al microservicio necesarias para gestionar los PCS desde *MyEdicom*, este proyecto es monitorizado por otro equipo de desarrollo dentro de la empresa. De este modo, el proceso de integración del microservicio de Propuestas con el proyecto *MyEdicom* no se incluirá en esta memoria.

4. Análisis del problema

4.1. Identificación y análisis de las soluciones posibles

Para migrar un módulo software a un nuevo microservicio nos enfrentamos a diversas decisiones de diseño. Seguramente la cuestión más relevante sea el establecimiento de los límites del microservicio, esto es, determinar qué tamaño debe tener. Para resolver esta pregunta vamos a hacer referencia al diseño guiado por el dominio, que es un enfoque para el desarrollo de software que centra el desarrollo en base a un modelo de dominio que tiene una gran comprensión de los procesos y las reglas del dominio [39]. Uno de los conceptos de esta aproximación se conoce como *Bounded Context* y hace referencia a un subdominio o subsistema de un dominio o sistema mayor y se encarga de realizar una función concreta. Este concepto se puede extrapolar al contexto de los microservicios para determinar los límites de un microservicio de manera que, cada *Bounded Context*, podría hacer referencia a un microservicio. Los microservicios creados bajo esta asunción deberían ser servicios normalmente independientes y vagamente acoplados.

No existe una receta exacta para establecer los límites del microservicio y, en ocasiones, es complicado. No obstante, cuando se trata de realizar una migración desde una aplicación monolítica a microservicio, los límites resultan más sencillos de establecer por el conocimiento previo que se tiene de las dependencias entre los módulos del sistema. En nuestra aplicación monolítica los límites del microservicio vienen determinados por diversos factores. Se ha tenido en cuenta que el microservicio cumpla con el principio de responsabilidad única de manera que su lógica de negocio no sea compartida por otros módulos de la aplicación. El microservicio también debe mantener una alta cohesión evitando el acoplamiento con otros módulos. Además, se ha considerado que el tamaño del módulo migrado debe ser adecuado para permitir un desarrollo ágil dentro de un equipo de pequeño tamaño. Sin embargo, el principal factor que determina el tamaño de nuestro microservicio se basa en la selección de un subdominio del servicio ofrecido por la aplicación monolítica, concretamente el módulo funcional encargado de las propuestas dentro del *Ebimap Server*.

Para conocer los límites del sistema y establecer el *boundex context* para nuestro microservicio se desarrolló un modelo conceptual, concretamente un modelo de dominio que permite la resolución de este tipo de problemas. Este modelo se puede entender como una representación del vocabulario, conceptos importantes y sus relaciones abarcando el dominio del problema. El modelo de dominio ayuda a comprender los requisitos y las decisiones de diseño. Al llevar a cabo el proyecto bajo un contexto ágil nos sirve, además de apoyo al backlog, análisis de las tareas y como guía para llevar a cabo todo el proceso de migración y desarrollo del microservicio. Para la construcción del modelo no sólo se han empleado aquellos requisitos que el servicio de Propuestas ofrecía en el sistema original, también se han incluido aquellos que se han propuesto durante el inicio de la

migración para su implementación como nuevas funcionalidades en el nuevo microservicio. Por supuesto, este modelo es dinámico y, a medida que se proponen nuevos requisitos, el modelo de dominio puede cambiar.

En la Figura 5 se representa un modelo de dominio establecido para el servicio de Propuestas y del que se pueden extraer los principales conceptos relacionados con dicho servicio. Específicamente, a partir de este modelo se puede advertir que una propuesta está constituida por registros que, a su vez, están compuestos por campos. Cada uno de estos campos posee una serie de propiedades que lo caracterizan.

La propuesta, a su vez, tiene asociado un modelo de acceso a datos, el Iead, que puede ser de tres tipos diferentes, XML, LOT o TXT. Este iead se tiene en cuenta a la hora de realizar ciertos procedimientos como la generación de ejemplos, generación del documento odt y para la validación de las propuestas haciendo uso de las reglas de validación. Además, estos procedimientos son específicos para cada uno de los interlocutores de la propuesta. El documento odt debe generarse en el lenguaje especificado por lo que debe hacerse uso de diccionarios para traduzcan la cadena correspondiente.

A la propuesta también se le pueden asociar una serie de metadatos y es posible exportarla e importarla desde o hacia el sistema para trabajar con ella. Además, las propuestas se pueden comparar entre ellas e incluso fusionarlas. Dentro de la aplicación, se puede acceder a las propuestas desde el repositorio principal, además, están relacionadas con otro tipo de objetos, los PCS que serán empleados desde la aplicación *MyEdicom*.

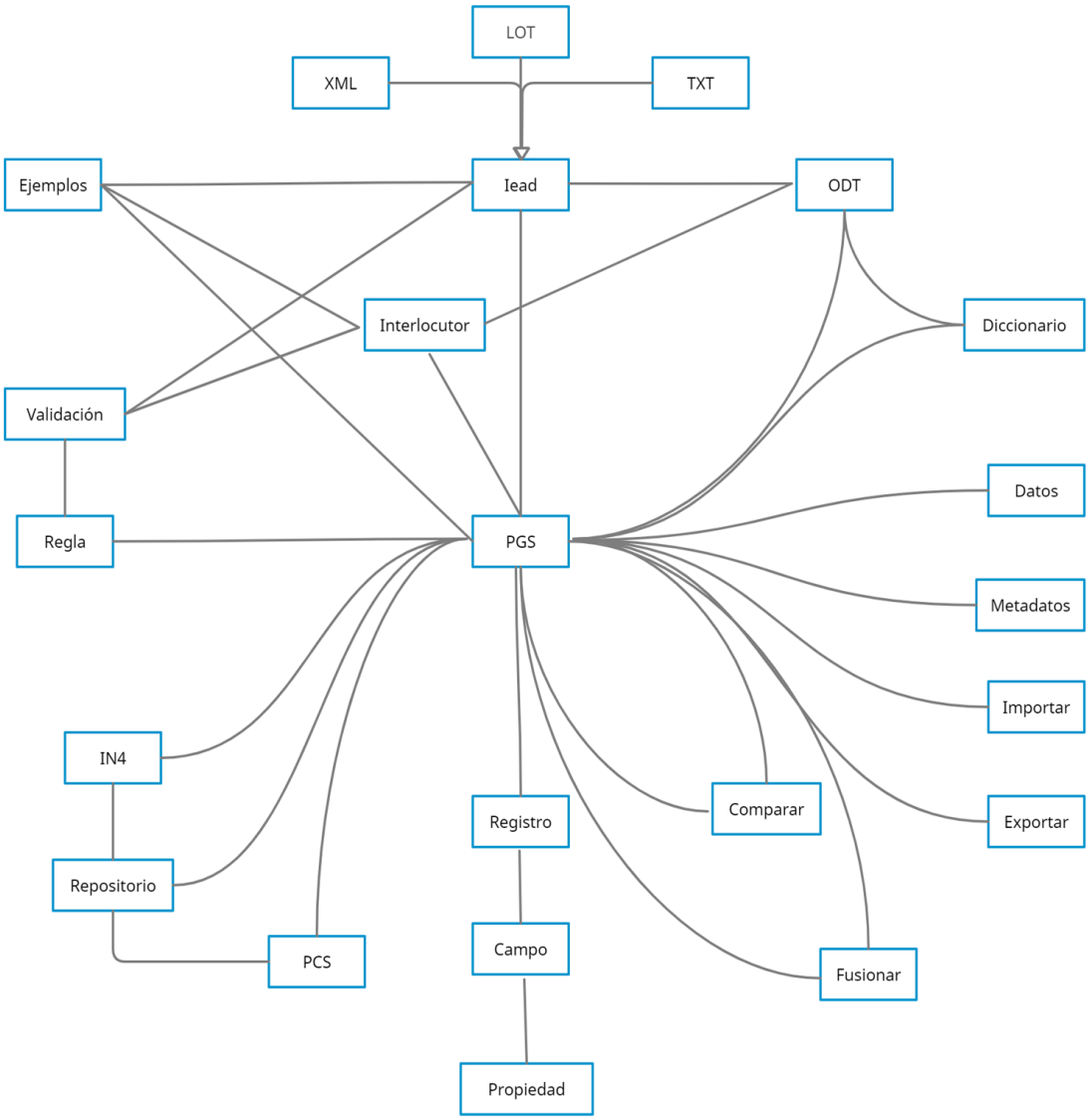

Figura 5. Modelo de dominio para el sistema de Propuestas

Por otro lado, a la hora de crear o migrar el microservicio es importante decidir si tendrá un estilo de comunicación síncrona o asíncrona con el resto de los proyectos [4]. El estilo de comunicación síncrono es una comunicación sin estado en el que un cliente lanza la petición al microservicio pasando la información necesaria y queda a la espera de la respuesta del servicio [40]. Al ser una comunicación sin estado el servicio puede tener una gran disponibilidad al poder generar múltiples instancias activas del microservicio que acepten tráfico [41]. En el caso de existir un error el sistema queda en un estado consistente sin comprometer la integridad de los datos. La mayor desventaja de este sistema es que el usuario del microservicio tiene que esperar hasta que la petición se resuelva, lo que limita la escalabilidad del sistema. La comunicación síncrona, además, aumenta la probabilidad de fallo ya que si en la cadena de peticiones falla un servicio entonces toda la cadena fallará.

En la comunicación asíncrona el microservicio es totalmente independiente al estar diseñado para recibir de forma asíncrona mensajes como entrada al sistema [17]. Esto supone que haya una mayor capacidad de escalabilidad y menos acoplamiento ya que cada microservicio, dependiendo de la carga, puede variar en hilos de ejecución para soportar la carga [41]. Si hay una sobrecarga de mensajes éstos quedarán encolados, pero a diferencia del diseño síncrono, no afectará a toda la cadena. El punto débil de este sistema de comunicación es que existe una dependencia con un servidor de mensajes y aspectos como la tolerancia a fallos puede ser complicado de gestionar.

En nuestro proyecto, debido a la mayor simplicidad de implementación, hemos optado por una comunicación síncrona con el microservicio. Además, el servicio de Propuestas se trata de un servicio dentro de *Ebimap* que no recibirá gran cantidad de peticiones por lo que, actualmente, tampoco resulta esencial una comunicación asíncrona con el microservicio.

Además, a la hora de diseñar el acceso al microservicio es importante tener en cuenta qué tipo de aplicaciones harán uso del servicio. Actualmente en nuestro caso son dos aplicaciones independientes, *Ebimap Web* y *MyEdicom*, las que harán uso del microservicio de Propuestas. Para implementar de forma sencilla y uniforme el acceso de estas aplicaciones al microservicio se propone el uso del patrón arquitectónico denominado API Gateway. Este tipo de patrón permite que clientes de diferentes naturalezas y con sus propias necesidades puedan tener de forma centralizada los endpoints del servicio [42] posibilitando que los microservicios puedan desarrollarse sin la influencia de sus clientes [13]. Además, se trata de un tipo de patrón arquitectónico muy usado cuando una empresa migra sus aplicaciones a microservicios de forma gradual de manera que es posible remplazar paulatinamente sus componentes con el tiempo [43]. También es un patrón frecuentemente usado cuando se quiere proporcionar a los clientes acceso a los endpoints de microservicios estrechamente relacionados, de manera que el Gateway, dependiendo de la petición de los clientes, enruta al microservicio correspondiente [43].

En nuestro caso, por el momento, solo implementará acceso al microservicio de Propuestas. No obstante, desarrollar un patrón API Gateway podría considerarse una buena práctica de programación al facilitar el mantenimiento del servicio. Por ejemplo, se podría querer en un futuro incluir en el API Gateway el acceso a otros microservicios estrechamente relacionados con el microservicio de Propuestas de forma sencilla sin necesidad de hacer grandes modificaciones en el código de los clientes o incluso podría particionarse el microservicio de Propuestas de manera que los cambios solo se reflejarían en el API Gateway sin necesidad de que afecten a los clientes.

La identificación y especificación de requisitos para el proceso de migración y desarrollo del microservicio de Propuestas se ha llevado a cabo, como en la totalidad del proyecto, siguiendo una aproximación ágil. Como se mencionaba en el subapartado de metodologías de la introducción, el servicio de Propuestas se ha desarrollado de forma iterativa e incremental a través de sprints. En cada uno de los sprints, el equipo de

desarrollo ha implementado nuevas características, enriquecido las ya existentes o ha resuelto defectos encontrados en la implementación.

Como parte de la filosofía ágil, los stakeholders del servicio de Propuestas, que son fundamentalmente los consultores de la empresa, han estado involucrados durante todo proceso de desarrollo del servicio. A lo largo de los sprints, y particularmente en las iteraciones iniciales, los consultores han colaborado con el equipo de desarrollo para identificar y priorizar las tareas que servirán como hoja de ruta durante el progreso del proyecto. Estas tareas podrían entenderse como historias de usuario en el contexto empresarial donde se ha llevado cabo el desarrollo del sistema. En la tarea se recoge la descripción del objetivo de forma clara y concisa ayudando al programador a entender el requisito.

Las tareas se han reunido en el backlog del equipo de desarrollo ordenándolas en función de la prioridad del requisito. Aquellos requisitos más prioritarios se incorporaron en las próximas iteraciones mientras que los menos prioritarios permanecieron en el backlog hasta el sprint oportuno. Las tareas se han definido de manera que permiten declarar las necesidades del usuario y sirven como punto de partida para el desarrollador, debiendo tener el tamaño adecuado para ser implementadas en una sola iteración.

Si una historia era demasiado grande o era considerada una épica entonces se subdividían en múltiples tareas que pudieran ser estimadas, implementadas y probadas en una sola iteración. En la Figura 6 se muestra un detalle parcial de la descripción de una tarea ya implementada. En la sección *Estado* podemos observar que se trata de una tarea que ya ha sido realizada por la persona que responde a la tarea. En este apartado es importante destacar el campo *Sugiere* que indica la procedencia del interesado que ha propuesto la tarea. En la sección tratamiento es donde se otorga un título a la tarea en el campo *Asunto* y se describe de forma clara y breve el requisito que se pide implementar. Además de la información mostrada en la figura, el desarrollador tiene acceso a más información de la tarea como la prioridad de esta o si se trata de un requisito especificado de forma completa, parcial o incompleta.

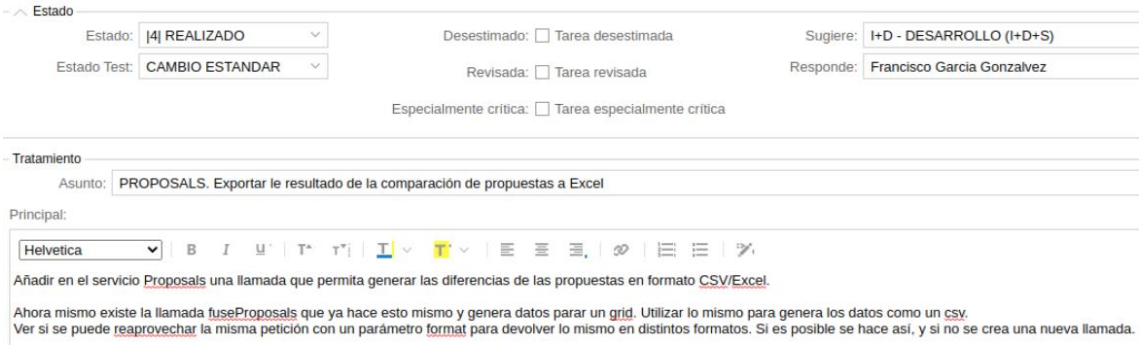

Figura 6. Detalle de una tarea del sistema de Propuestas

4.2. Solución propuesta

Para llevar a cabo la migración y desarrollo del servicio de Propuestas se planteó determinar los límites del servicio a partir de los requisitos que se esperan del mismo. Estos requisitos incluyen aquellas funcionalidades ya establecidas en el sistema original de Propuestas, así como los nuevos requisitos que se han sido sugeridos para mejorar el sistema existente. En este sentido, a partir del modelo de dominio se pudo extraer aquellos subdominios funcionales que forman parte del microservicio.

Una vez establecidas las fronteras del servicio, se procedió a determinar el proceso de aislamiento del módulo software. Para la implementación del microservicio se propuso emplear HTTP/REST debido a que se trata de una comunicación simple y familiar ya que se basa en una comunicación de petición y respuesta. El protocolo HTTP, además, favorece la interoperabilidad, el enrutado del tráfico, balanceos de carga, tratamiento del protocolo y la seguridad de sistemas [40]. En definitiva, la pila de protocolos HTTP/REST hace que el desarrollo de microservicios sea más fácil. Por ejemplo, su testeo es mucho más sencillo ya que con el uso de herramientas como Postman es posible hacer peticiones a un determinado endpoint del servicio para comprobar su correcto funcionamiento.

Esta comunicación no requiere un broker intermediario lo que simplifica mucho la arquitectura del sistema. Este tipo de comunicación también tiene sus desventajas. Una de ellas es que se reduce la disponibilidad del servicio con respecto a otros estilos de comunicación ya que el cliente y el servicio se comunican directamente sin intermediarios que puedan almacenar los mensajes. No obstante, el servicio de Propuestas no debería presentar problemas al respecto al no presentar una alta tasa de peticiones.

Por otro lado, teniendo en cuenta los subdominios que integrarán el servicio de Propuestas y la arquitectura del microservicio se procedió a plantear la estructura de la API y la implementación de los endpoints. Como se ha comentado en el punto anterior, entre las aplicaciones cliente del microservicio de Propuestas y el propio microservicio se desarrolló un pequeño proyecto que actúa como un API Gateway. Este proyecto llamado *ProposalsRestClient* permite, de forma uniforme, el enrutado de los clientes *Ebimap Web* y *MyEdicom*, hacia el microservicio de Propuestas.

Simultáneamente al trabajo en el microservicio también se ejecutó la refactorización del proyecto *Ebimap Web* para la comunicación con el microservicio, así como el desarrollo de la nueva interfaz gráfica de usuario del sistema de Propuestas. Es importante resaltar que, al trabajar bajo un paradigma ágil, los requisitos del sistema de Propuestas podrían verse alterados por parte de los stakeholders pudiendo afectar en menor o mayor medida al dominio de negocio. No obstante, trabajar en sprints de corta duración permite abordar estos cambios de forma eficiente.

5. Diseño de la solución

5.1. Tecnología empleada

En esta sección se describe la tecnología usada a lo largo del ciclo de vida de nuestra aplicación, a excepción de las herramientas de testeo que se detallarán en la octava seccción.

Durante el desarrollo del todo el proyecto se empleó GitLab como sistema de control de versiones [44]. GitLab es una plataforma de gestión de proyectos basada en la nube que permite a los desarrolladores gestionar el código de los proyectos de manera colaborativa. Ayuda a los desarrolladores a monitorizar todo el ciclo de vida de la aplicación, cubriendo todo el ciclo DevOps desde las etapas más tempranas hasta el despliegue del código en producción. El software, además, proporciona integración continua, entrega continua y testing.

Como entorno de desarrollo integrado (IDE) para el desarrollo del microservicio *Proposals* y backend de *Ebimap Web* se ha empleado Eclipse [45]. Este IDE está escrito mayoritariamente en Java y es usado especialmente para el desarrollo de aplicaciones Java. Se trata de un IDE que permite fácilmente extender sus funcionalidades a través de la instalación de herramientas de desarrollo. Para la implementación del frontent en el proyecto *Ebimap Web*, escrito en JavaScript, se usó el editor de código fuente Visual Studio Code [46]. De forma similar a Eclipse, Visual Studio Code puede extenderse mediante el uso de complementos.

Microservicio *Proposals* y backend de *EbimapWeb*

La construcción del microservicio *Proposals* se realizó haciendo uso de Spring, un framework de código abierto para el desarrollo de aplicaciones Java [47]. Además, se empleó específicamente Spring Boot, una extensión del framework que simplifica la configuración requerida para desarrollar una aplicación. Algunas de las ventajas de este framework son:

1. Facilita la creación de aplicaciones autónomas e independientes
2. Presenta un tomcat embebido de manera que no hay necesidad de desplegar archivos WAR
3. Proporciona dependencias iniciales simplificando la configuración para la compilación
4. Configura automáticamente librerías siempre que es posible

5. Provee algunas características para los entornos de producción como métricas o verificación de la salud del sistema

Para la gestión y construcción del microservicio se utilizó la herramienta de software Maven que facilita la compilación, el testeo y el empaquetamiento de proyectos [48]. Se trata de uno de los programas de código abierto más usados en la industria a nivel mundial. Dentro de las características de esta herramienta podemos resaltar:

1. Gestión declarativa de las dependencias. Los proyectos Java normalmente tienen dependencias de otros proyectos. En lugar de gestionar cada una de las dependencias de forma manual Maven permite declarar las dependencias del proyecto en un archivo pom.xml de manera que dichas dependencias se descargan automáticamente en el proyecto.
2. La funcionalidad de Maven es fácilmente personalizable mediante cientos de plugins disponibles.
3. Maven poseen una interfaz uniforme y estándar para la compilación de proyectos usando sencillos comandos.
4. Todos los principales IDEs tienen herramientas de soporte para Maven
5. Presenta arquetipos, que son de plantillas predefinidas de proyecto que son usadas para crear nuevos proyectos.
6. Es de código abierto con abundante documentación y apoyo de la comunidad.

Un aspecto importante cuando se define un microservicio es la construcción de la API que permita a los desarrolladores conocer la funcionalidad que ofrece el servicio. Un API de calidad no sólo debe tener un diseño sencillo, sino que también debe presentar una documentación de calidad para los usuarios. En nuestro proyecto se ha elegido Swagger [49], una herramienta de código abierto que sirve para diseñar, construir y documentar servicios REST así como interaccionar con la API.

Frontend de *Ebimap Web*

La parte frontend del proyecto *Ebimap Web* escrita en JavaScript. Dentro de las dependencias del proyecto cabe destacar la librería DHTMLX [50] que ofrece un conjunto de componentes de interfaz de usuario para la construcción de aplicaciones web. Esta librería es ampliamente usada en la aplicación *Ebimap Web* y, por extensión, se ha empleado en la creación de la interfaz de usuario para las propuestas.

También cabe destacar la dependencia con webpack [51]. Webpack es un empaquetador de módulos, esto es, genera un solo archivo de todos los módulos que la aplicación

requiere para funcionar. Escribir el código en módulos ayuda con la organización, el mantenimiento, testeo y más importante, la gestión de dependencias.

Despliegue del microservicio

El despliegue del microservicio se realiza mediante el uso de contenedores. Docker es la plataforma que usamos en el proyecto y que permite el empaquetado y ejecución de aplicaciones en contenedores [52]. Los contenedores son similares a las máquinas virtuales, pero a diferencia de éstas, sólo virtualizan las capas de software por encima del sistema operativo. Contienen su propio sistema de archivos, CPU compartida, memoria, espacio de procesos entre otras cosas. Al estar desacoplados de la infraestructura son portables lo que pueden ser considerados como unidad para la distribución y el testeo de la aplicación. Los contenedores suponen grandes beneficios tales como:

- Despliegue y desarrollo ágil de aplicaciones. La creación de contenedores es más sencilla que el uso de máquinas virtuales.
- Desarrollo, integración y despliegue continuo. Permite la creación frecuente de imágenes de contenedores y, por tanto, un despliegue eficiente.
- Proporciona información y métricas tanto del sistema operativo como de la salud de las aplicaciones.
- Permite que la aplicación sea consistente en entornos de desarrollo, producción y pruebas.
- Son portables ya que hacen uso del sistema operativo anfitrión y no requieren un sistema operativo invitado en cada instancia como en el caso de las máquinas virtuales.
- Posibilita el desarrollo de microservicios que pueden ser desplegados de forma dinámica y no como una aplicación monolítica.
- Consigue un aislamiento de los recursos haciendo más previsible el rendimiento de la aplicación y una mejora en la eficiencia de utilización de los recursos.

La gestión de los contenedores se realiza mediante Kubernetes, una plataforma de código abierto que proporciona un contexto para la monitorización de contenedores facilitando la configuración y automatización [53]. Kubernetes proporciona:

- Descubrimiento de servicios exponiendo un contenedor haciendo uso del nombre DNS o su propia dirección IP.
- Balanceo de la carga distribuyendo el tráfico hacia el contenedor manteniendo el despliegue estable.
- Orquestación del almacenamiento al permitir montar automáticamente un sistema de almacenamiento.

- Rollouts y rollbacks automáticos. Permite fijar el estado deseado para el despliegue de los contenedores.
- Empaquetado automático de contenedores. Kubernetes proporciona un clúster de nodos que puede ser usado para ejecutar tareas en los contenedores. Se puede especificar cuantos recursos, CPU y RAM, necesita cada contenedor de manare que kubernetes puede adecuar los contenedores a los nodos haciendo el mejor uso de los recursos.
- Autocorreción. Kubernetes reinicia los contenedores que fallan, sustituye los contenedores y elimina aquellos que no responden a la verificación de integridad definida por el usuario.
- Gestión de la configuración y claves. Kubernetes permite almacenar y gestionar informaciones confidenciales como claves o tokens.

5.2. Arquitectura del Sistema

En este apartado se describe la arquitectura general del microservicio y su interacción con el resto de las aplicaciones que integran del servicio de Propuestas. En la sección posterior se profundizará en cómo se ha estructurado en detalle el microservicio *Proposals*. En la figura 7 podemos observar un esquema general del microservicio y su interacción con otros proyectos. La lógica de negocio del microservicio tiene un puerto de entrada que define un conjunto de operaciones y cómo interacciona con el exterior. Este puerto es una API expuesta por el microservicio y permite ser invocada por aplicaciones externas. Concretamente esta API será invocada por la aplicación *Ebimap Web* y *MyEdicom* a través del API Gateway cuando se haga uso del servicio de Propuestas. La interacción con el servicio de Propuestas vía API posibilita la encapsulación de los detalles de su implementación.

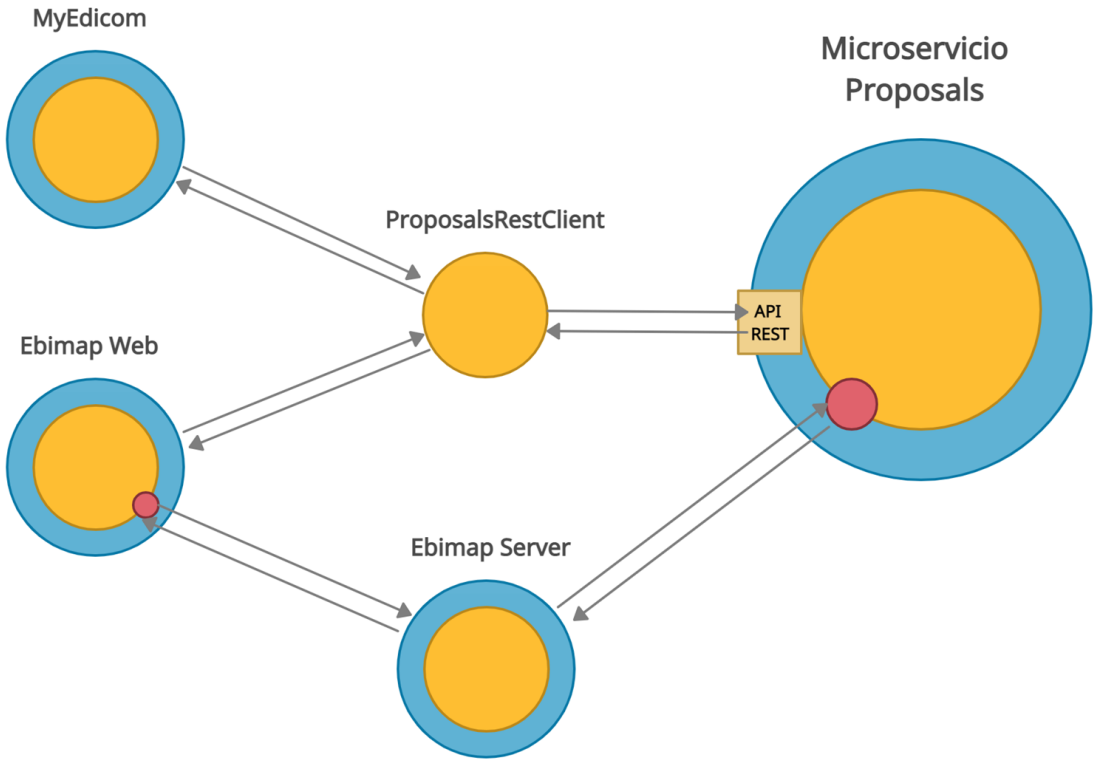

Figura 7. Arquitectura general del servicio de Propuestas

Alrededor de la capa de negocio del microservicio se encuentran los adaptadores. El adaptador de entrada son todas aquellas clases de la aplicación que manejan las peticiones desde el mundo exterior. Estas clases son, en nuestro caso, los controladores de Spring que implementa el conjunto de endpoints REST. Los controladores permiten interceptar la petición realizada al microservicio y devolver la respuesta de éste una vez que está preparada. Los controladores son, a su vez, los encargados de invocar la lógica de negocio del microservicio con los datos de la petición y recoger la información de la lógica de negocio una vez finalizado su ejecución. La lógica de negocio, implementada mayoritariamente en forma de clases de servicios dentro del proyecto, se encarga del procesamiento de los datos capturados en los controladores. Esta lógica, además, implementa solicitudes desde la capa de negocio al invocar servicios externos. Concretamente el microservicio tiene un puerto de salida que permite invocar a la lógica del *Ebimap Server*, proyecto encargado del acceso a un repositorio.

Como parte de la estructura del servicio de Propuestas se ha implementado el patrón arquitectónico API Gateway. *ProposalRestClient* es el proyecto que actúa como API Gateway de manera que los clientes del microservicio, *Ebimap Web* y *MyEdicom*, puedan tener de forma centralizada a los endpoints de la API del microservicio de Propuestas.

Por otro lado, la aplicación *Ebimap Web* está formada por un backend y un frontend. La migración del módulo de propuestas a un microservicio independiente ha supuesto una refactorización de ambas capas. A nivel de frontend se ha desarrollado una nueva interfaz gráfica de usuario para los objetos PGS y PCS. El frontend interacciona con el backend

del *Ebimap Web* a través peticiones interceptadas por controladores. Los controladores son los encargados de invocar a la lógica de negocio con la información procedente del frontend. La lógica de negocio se encarga de transformar la información y realizar la petición adecuada al microservicio de Propuestas. Una vez procesada la información, la información será recibida por la lógica del proyecto *Ebimap Web*, que, a su vez, se encargará de pasar la información a su frontend a través del controlador.

5.3. Diseño detallado

En este punto se profundiza en la estructura del microservicio *Proposals* detallando aquellos paquetes más relevantes de su implementación. En la siguiente figura se muestra la estructura la paquetería del microservicio de Propuestas.

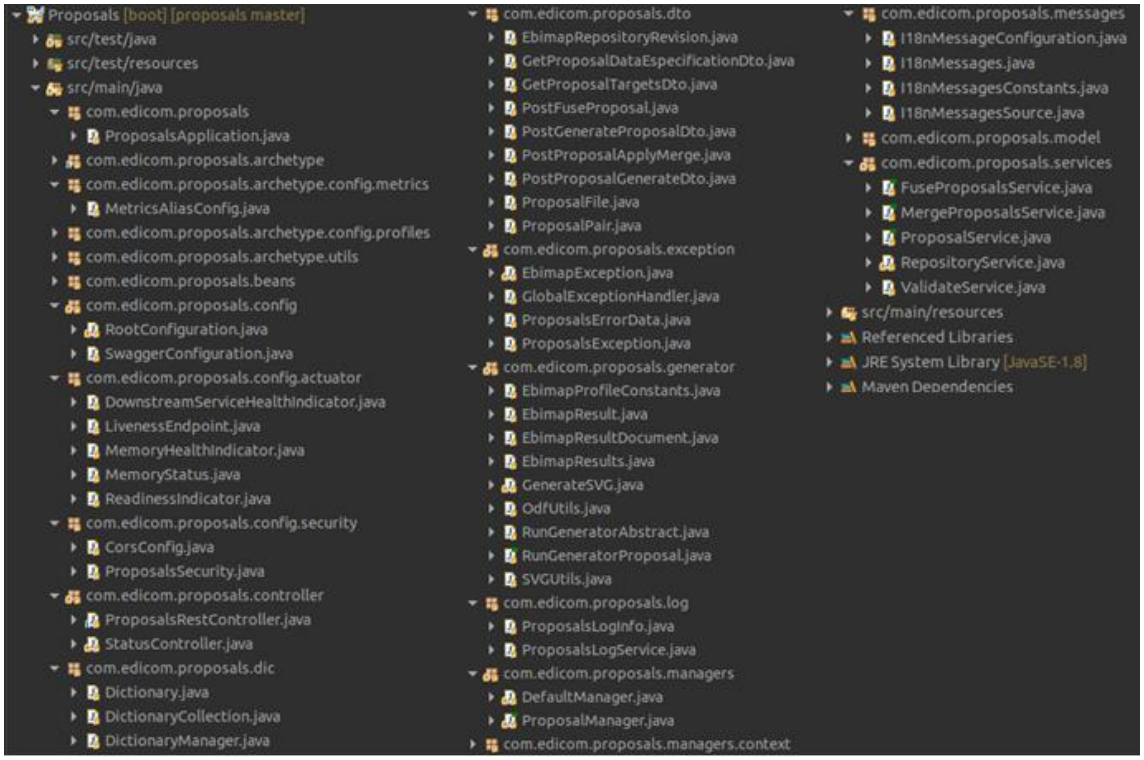

Figura 8. Paquetería del microservicio de Propuestas

Paquete *com.edicom.proposals.dto*. En este paquete se concretan los DTOs que definen los objetos que son manejados por el API del microservicio. Los DTOs permiten transferir la información en objetos desde *Ebimap Web* hasta el microservicio *Proposals*. Por ejemplo, el DTO PostProposalGenerateDto es utilizado en el endpoint que genera el documento odt de propuestas y contiene tanto la estructura base de la propuesta en base 64 como todos los parámetros especificados por el usuario en la interfaz gráfica y que son requeridos para generar el documento de la propuesta.

Paquete *com.edicom.proposals.config*. Dentro de este paquete se da apoyo a las configuraciones del proyecto. La clase RootConfigurationProperties proporciona un soporte efectivo para aquellos componentes que están anotados con @ConfigurationProperties. También podemos encontrar dentro de este paquete la clase que da soporte a la documentación de la API del proyecto. Concretamente la clase SwaggerConfiguration que, como su nombre indica, ajusta la configuración de Swagger.

Paquete *com.edicom.proposals.config.actuator*. Dentro de este paquete se establece la configuración del subproyecto Actuator de Spring Boot. Actuator ayuda a monitorizar y gestionar la aplicación.

Paquete *com.edicom.proposals.config.security*. Los ajustes de seguridad están presentes en este paquete donde se ubica la clase ProposalsSecurity, que presenta la anotación @EnableWebSecurity y que habilita el soporte de la seguridad web de spring y proporciona integración con Spring MVC. Además, extiende la clase WebSecurityConfigureAdapter que permite sobreescribir algunos de sus métodos para especificar la configuración de la seguridad web. Uno de estos métodos es el método *configure(HttpSecurity)* que define que URLs deben estar aseguradas y cuáles no. Por ejemplo, para el caso del path /actuator/health no se requiere ningún tipo de autenticación. Sin embargo, para el caso del path /rest/** se requiere el role de administrador.

Paquete *com.edicom.proposals.messages*. El servicio de Propuestas debe poder ofrecer respuestas en múltiples lenguas. Dentro de este paquete se configura la internacionalización de la aplicación.

Paquete *com.edicom.proposals.generator*. Dentro de este paquete se definen una serie de clases de utilidad que son usadas en el proyecto. Se definen, por ejemplo, constantes, métodos para la manipulación de documentos OpenOffice o manejo de gráficos vectoriales escalables (SVG).

Paquete *com.edicom.proposals.controller*. En este paquete es donde se define la capa controladora. Esta capa permite interceptar la petición realizada al microservicio y devolver la respuesta de este una vez que está preparada. El controlador principal de este paquete es el que contiene los endpoints funcionales del microservicio como el que genera el documento odt de la propuesta, compara propuestas o fusiona las mismas. En este paquete encontramos también un controlador secundario y contiene los endpoints que posibilitan comprobar la salud del sistema.

Paquete *com.edicom.proposals.dic.* Toda la gestión de los diccionarios para la traducción de las propuestas está contenida en este paquete. En él se define un manager que permite recuperar las cadenas actualizadas de los diccionarios cada seis horas.

Paquete *com.edicom.proposals.exception*. Dentro de este paquete se define la excepción ProposalsException para el servicio de Propuestas, así como el manejador de excepciones. Este último, especificado como GlobalExceptionHandler usa la anotación de spring @RestControllerAdvice que permite el manejo de excepción en Api REST.

Concretamente esta anotación permite al manejador actuar como una especie de interceptor de manera que envuelve la lógica en nuestros controladores permitiendo aplicar una lógica común en ellos. Además, los métodos de la clase controladora, anotados con @ExceptionHandler son compartidos globalmente por todos los controladores para capturar excepciones y trasladarlas a respuestas HTTP. La anotación @ExceptionHanlder permite indicar que tipo de excepción queremos manejar.

Paquete *com.edicom.proposals.managers*. En este paquete y, particularmente en la clase ProposalManager, se implementan métodos de acceso a los servicios.

Paquete *com.edicom.proposals.services*. En este paquete se engloban las clases que recogen la mayor parte de la lógica de negocio. Los servicios se dividen en ProposalService, que implementa la lógica para la generación de propuestas, FuseProposalService, que permite la comparación de propuestas, MergeProposalService, donde se define la lógica para la unificación de dos propuestas y ValidateService que posibilita la validación de las propuestas.

En el directorio src/main/resources podemos encontrar los archivos de configuración y otros recursos. En nuestro proyectos encontramos el archivo application.properties donde se definen, en forma de clave-valor, algunas propiedades de configuración del proyecto. Además, también encontramos un documento de extensión .odt que sirve como plantilla para la generación del documento de la propuesta.

El directorio src/test es el lugar donde residen los tests de cada componente de la aplicación. Dentro del subdirectorio src/test/java se implementa el código para los tests. Este aspecto se verá en mayor profundidad en el apartado de Pruebas de la memoria. Al directorio de test se añade, además, el subdirectorio src/test/resources donde se especifican archivos de configuración de los testeos y otros archivos empleados en los test para simular propuestas.

Dentro del proyecto también encontramos los archivos jar que permiten ejecutar correctamente la aplicación. Dentro de Maven dependencias encontramos aquellas dependencias que son añadidas en el archivo pom. JRE System Libraries es aquella colección de librerías de Java SE que se usan para crear proyectos Java. En Referenced Libraries se encuentran las librerías third-party que se usan en el proyecto.

En la aplicación *Ebimap Web* también se realizaron cambios para adaptar el sistema al nuevo microservicio. A nivel arquitectónico podemos localizar, en el código del proyecto escrito en JavaScript, la implementación del frontend del servicio de Propuestas. Como se muestra en la figura 9, dentro del directorio *objects*, donde se encuentran otros elementos importantes del *Ebimap Web*, tales como interfaces o mapas, se define el comportamiento y la apariencia de la interfaz de las propuestas.

En la figura 9 se puede observar que, dentro de un objeto propuesta, se especifica un directorio *actions* que contiene los scripts que establecen acciones y eventos sobre elementos de la interfaz de usuario de la propuesta. Dentro del directorio *ui* se definen

cada uno de los componentes que forman parte de la interfaz gráfica de usuario tales como los menús o grids. Finalmente, en el último subdirectorio dentro de los objetos de propuestas se detallan las ventanas que forman parte de la interfaz gráfica de las propuestas. Además de la descripción del comportamiento e interfaz de un objeto propuesta, se establecen una serie de servicios que serán los responsables de realizar las llamadas a la parte Java del proyecto *Ebimap Web*, que, a su vez, se encarga de llamar al microservicio *Proposals*. Finalmente resaltar que la parte de java del *Ebimap Web* asociada al servicio de Propuestas tiene una estructura sencilla basada en controladores y servicios de Spring muy similar a lo descrito para el caso del microservicio.

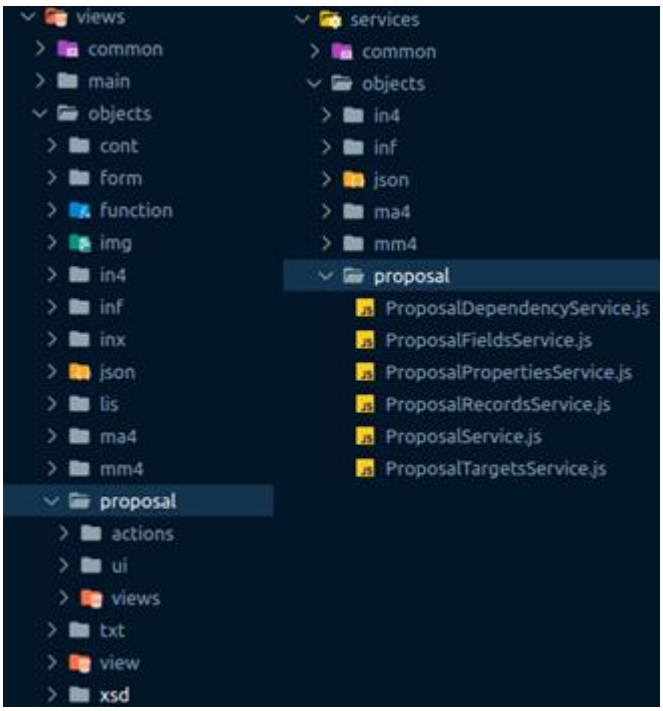

Figura 9. Árbol de carpetas del frontend de la aplicación *Ebimap Web*

6. Desarrollo de la solución

El proceso de migración del módulo de propuestas de la aplicación monolítica a microservicio se ha percibido como una forma de modernización del servicio de Propuestas, de manera que, en lugar de comenzar el microservicio desde cero, se ha producido una refactorización incremental de todo el servicio. El primer paso en el proceso de migración ha consistido en determinar y sintetizar todos los requisitos que debe cumplir el microservicio en términos de funcionalidad. Muchos de los requisitos están impuestos por el módulo software de propuestas ya existente en la aplicación monolítica. Estos requisitos suponen la funcionalidad básica que debe tener el microservicio. Algunas de estas funcionalidades son la de generación del documento odt

de la propuesta, la generación de ejemplos a partir de la propuesta, la comparación de propuestas o la fusión de éstas.

Otros requisitos, no obstante, no estaban en el módulo funcional original y se han implementado por primera vez en el microservicio por petición de los consultores de la empresa que trabajan con el servicio *Ebimap*. Por ejemplo, el acceso a las propuestas a través de la aplicación *MyEdicom* o la incorporación de los modelos de acceso a datos o IEADS en la creación del documento odt la propuesta, en la generación de los ejemplos o validación de la propuesta. En cualquier caso, en esta primera etapa del proceso de migración, se definió el comportamiento y las operaciones que debía tener el microservicio.

Para ayudar en la definición de los límites del servicio se creó un modelo de dominio de alto nivel para proporcionar un vocabulario que describiese los elementos y operaciones básicas del sistema. Este modelo de dominio define el *bounded context* según DDD e incluye todos los artefactos del módulo funcional. El *bounded context* por tanto se generó analizando el módulo funcional de la aplicación origen e incluyendo nuevos requisitos del servicio que se derivan a partir de las historias de usuario o tareas. Es importante resaltar que este modelo se trata de un punto de partida para definir el escenario arquitectónico del microservicio teniendo en cuenta que durante el proceso de migración podrían surgir otros nuevos requisitos o incluso descartar alguno estipulado inicialmente.

A continuación, se trató de determinar la forma en la que se iba a aislar el módulo software en el microservicio *Proposals*. En esta fase fue necesario analizar y definir todas las competencias del módulo funcional de propuestas que se veían afectadas en el proceso de migración. Para identificar estas competencias se analizaron todos los procesos y componentes relacionados con la gestión de propuestas y, dentro de este proceso de gestión de propuestas, se trató de descomponer otras subcompetencias (generación, comparación, fusión de propuestas, gestión de los diccionarios, etc) que permitirían estructurar el microservicio adecuadamente (ver figura 10). El resultado de este proceso, además, permitió tener identificadas aquellas partes del proyecto que iban a ser susceptibles de refactorización.

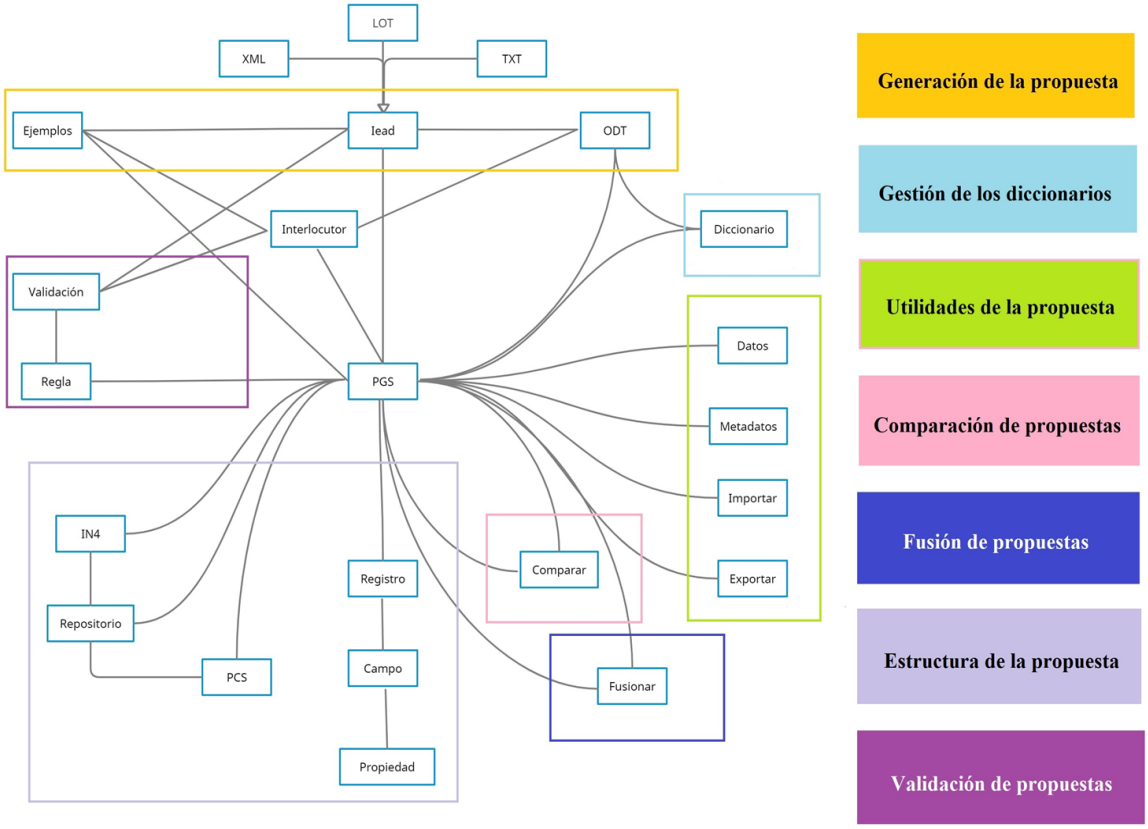

Figura 10. Extracción de subdominios a partir del modelo de dominio

En un paso posterior se definió cómo iba a estar estructurado el microservicio en términos de API. Para el microservicio *Proposal* se definieron un conjunto de operaciones que el módulo cliente de *Ebimap Web* pudiese invocar. Para la definición del API usamos la herramienta Swagger [49], que se trata de una herramienta que permite el desarrollo de documentación de APIs.

En la Figura 11 podemos observar los endpoints funcionales que actualmente soporta el microservicio de Propuestas. Además de estos endpoints se ha implementado una API que comprueba la salud del microservicio. Se trata de una solución del servicio para decir a la infraestructura de despliegue si es capaz de gestionar peticiones y son relevantes a la hora de desplegar el microservicio. Spring Boot Actuator es una librería de Java que implementa el endpoint /actuator/health y que devuelve un 200 como respuesta siempre que el servicio está en buen estado y un 503 en otro caso.

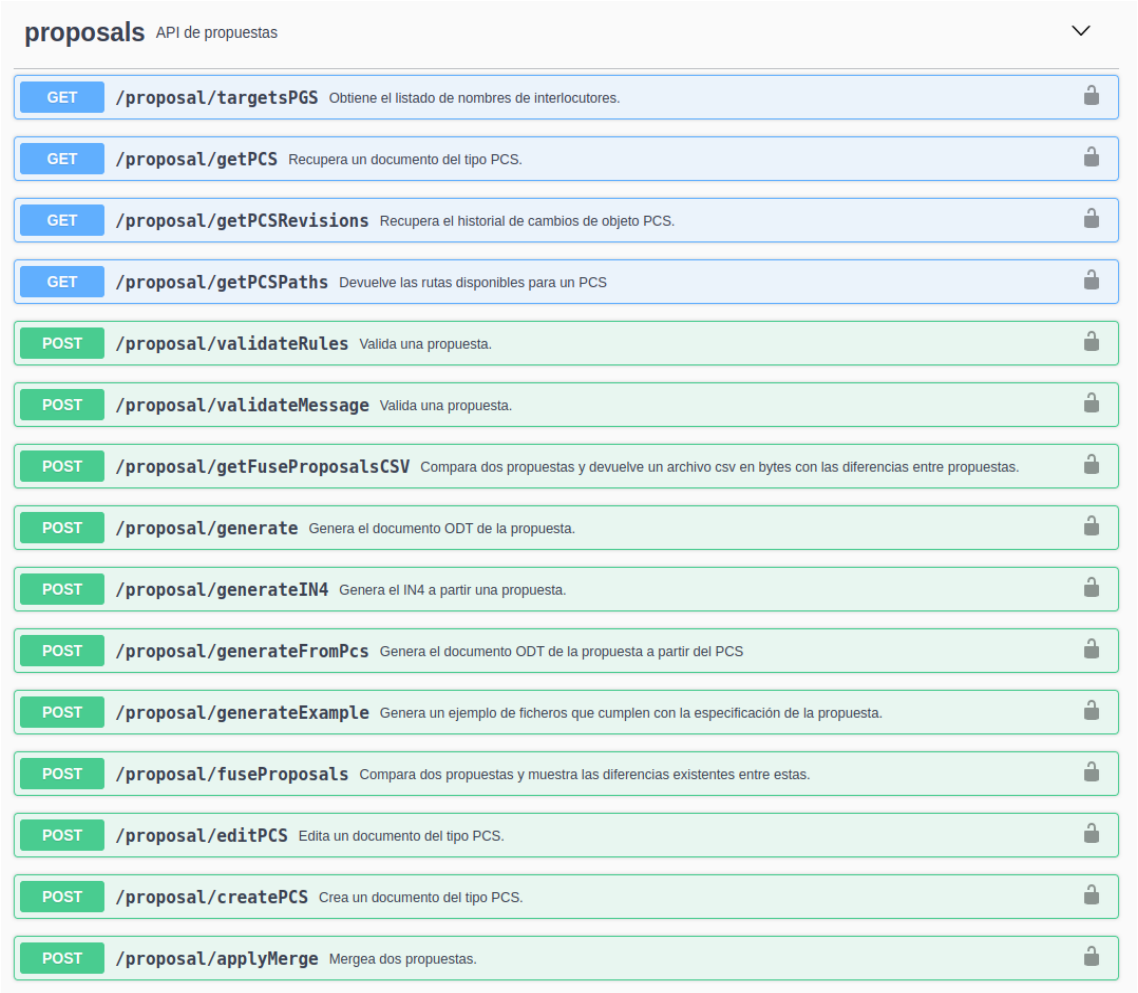

Figura 11. API del microservicio de Propuestas

Para la definición de la API usamos el protocolo HTTP haciendo uso de los verbos HTTP. La comunicación entre *Ebimap Web* y el microservicio *Proposals* se realizó implementando un patrón de invocación remoto síncrono donde el cliente, en nuestro caso *Ebimap Web* envía una petición al servicio *Proposals*. Si la petición es válida se procesa la petición ejecutando la lógica de negocio asociada a la petición. Finalmente, si la petición es procesada correctamente se envía de vuelta a *Ebimap Web* la respuesta. Si, por el contrario, la petición entrante no es procesada adecuadamente se envía al cliente una respuesta con la información pertinente. En la Figura 12 se representa el proceso de comunicación entre *Ebimap Web* y el microservicio *Proposals.*

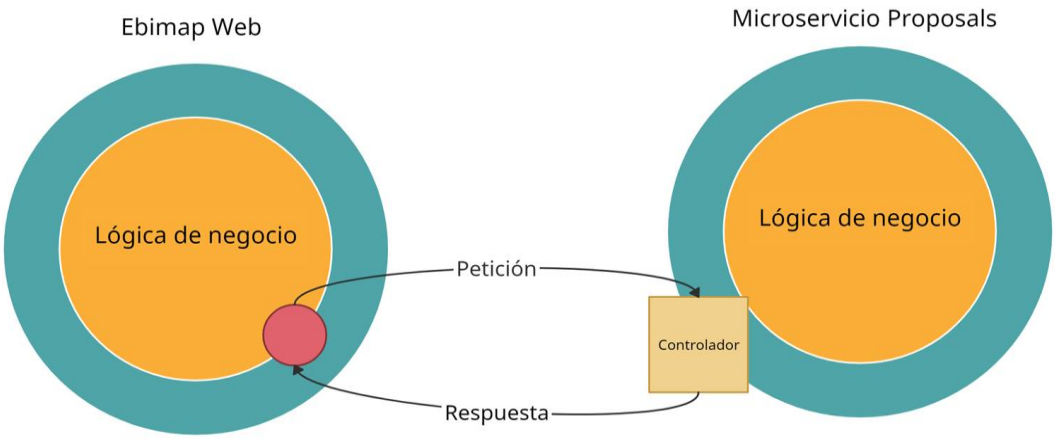

Figura 12. Esquema de comunicación entre *EbimapWeb* y el microservicio *Proposals*

Para el desarrollo de la capa API el framework Spring proporciona una serie elementos que facilitan la construcción de una arquitectura REST. La clase controladora permite, a partir de la petición, ejecutar la lógica de negocio devolviendo los resultados al servlet, una clase que responde a las peticiones HTTP. Para el desarrollo de la clase controladora normalmente usamos la anotación @Controller, que se trata de una especialización de una clase componente (@Component). En nuestro proyecto, sin embargo, usamos @RestController que combina la anotación @Controller y @ResponseBody en la clase controladora (serializa automáticamente el objeto de retorno en un Httpresponse). En la Figura 13 podemos observar la clase controladora del microservicio etiquetada con @RestController. Además, se puede apreciar que, mediante la anotación @Autowired que ofrece Spring, realizamos una inyección de dependencias automática de los servicios que hace uso el controlador para poder realizar la invocación de los métodos de dichos servicios.

```java
@RestController
@RequestMapping(ProposalsRestController.PROPOSAL_REST_CONTROLLER)
@Tag(name = "proposals", description = "API de propuestas")
public class ProposalsRestController {

    public static final String PROPOSAL_REST_CONTROLLER = "/proposal";
    private static final String BAD_REQUEST_STATUS_DESCRIPTION = "Bad request";
    private static final String OK_STATUS_DESCRIPTION = "Successful operation";

    @Autowired
    RunGeneratorProposal generator;

    @Autowired
    private ProposalService proposalService;

    @Autowired
    private FuseProposalsService fuseProposalsManager;

    @Autowired
    private MergeProposalsService mergeProposalsManager;
```

Figura 13. Detalle de la clase controladora ProposalsRestController.

Para modelizar el mapeado para los diferentes verbos HTTP, se suele emplear la anotación @RequestMapping, sin embargo, Spring proporciona variantes como @GetMapping y @PostMapping para simplificar la información de las anotaciones sin necesidad de especificar el verbo HTTP en el @RequestMapping. En este proyecto decidimos usar estas últimas variantes por su simplicidad a la hora de implementar los endpoints.

En la Figura 14 se muestra la estructura de los endpoints del microservicio. En este ejemplo concretamente se observa el endpoint */fuseProposals* encargado de comparar dos propuestas y devolver en un grid las diferencias entre las mismas. El endpoint recibe como parámetro un objeto PostFuseProposal que contiene las dos propuestas. Nótese que existe la anotación @RequestBody que permite pasar como parámetro el cuerpo de una petición al método. El cuerpo del método es el encargado de invocar la lógica de negocio que se encuentra en el servicio correspondiente. El formato de los mensajes está basado en JSON, que es un formato legible por los humanos y autodescriptivo y presenta el mensaje como una colección de propiedades. En la figura 14 también se puede observar que, acompañando al método, existen una serie de anotaciones que permiten describir la API con la herramienta Swagger. Concretamente la anotación @operation permite asignar un resumen y descripción de la funcionalidad del endpoind. La anotación @ApiResponses permite describir las posibles respuestas de éxito o error que podemos recibir tras la llamada al endpoint.

```
@Operation(
        summary = "Compara dos propuestas y muestra las diferencias existentes entre estas.",
        description = "Permite comparar dos propuestas y muestra las diferencias existentes entre estas.",
        tags = { "proposals" })
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
            description = OK_STATUS_DESCRIPTION,
            content = { @Content(
                    mediaType = MediaType.APPLICATION_JSON_VALUE) })
})
@PostMapping(path = "/fuseProposals", produces = "application/json;charset=utf-8")
public UIGrid fuseProposals (@RequestBody PostFuseProposal proposalMergeFiles)
    throws CloneNotSupportedException {

    ProposalPair proposals = fuseProposalsManager.getProposalsFromPostFuseProposal(proposalMergeFiles);
    return fuseProposalsManager.fuseProposals(proposals.getProposalBase(), proposals.getProposalSource());
}
```

Figura 14. Ejemplo de estructura de endpoind del microservicio *Proposals*

La capa lógica es la más compleja del microservicio y es la que procesa los datos recibidos a través de los controladores y devuelve a éstos la información ya procesada. La mayor parte de la lógica de negocio está implementada en los servicios de la aplicación. Durante el análisis del proceso de migración del microservicio se identificaron cuatro subdominios que se traducirían en las clases de tipo servicio implementadas. Estas clases se organizan en ProposalService que implementa la lógica para la generación de propuestas, FuseProposalService que permite la comparación de propuestas, MergeProposalService que donde se define la lógica para la unificación dos propuestas y ValidateService que posibilita la validación de las propuestas. La principal característica de estas clases es que

vienen etiquetadas con la anotación de Spring @Service. Esta anotación es un caso especial de la anotación @Component por lo que cualquier clase anotada con @Service será escaneada, instanciada e inyectada como dependencias por Spring donde sea necesario. La anotación @Service únicamente indica que la clase está soportando parte de la lógica de negocio de la aplicación.

Una funcionalidad interesante que ofrece Spring de forma nativa que es particularmente relevante para el proyecto de Propuestas se consigue mediante la anotación @Scheduled (ver Figura 15). Esta anotación permite programar tareas en el proyecto para que se lancen cada cierto intervalo de tiempo dependiendo de los parámetros de configuración. En el microservicio de Propuestas esta notación se ha usado para descargar, cuando se levante el proyecto y posteriormente cada seis horas, los diccionarios de las propuestas con el fin de tener actualizadas constantemente las cadenas de traducción.

En la imagen siguiente se puede observar que el método que actualiza los diccionarios se ha programado con la anotación @Scheduled. El parámetro fixedRate establece el tiempo que tiene que pasar entre dos ejecuciones y tiene el valor de 26100000 que es la cantidad de milisegundos contenidos en seis horas. Por otro lado, el parámetro initialDelay establece el tiempo que tiene que pasar tras levantar el proyecto para que se ejecute la tarea por primera vez que en nuestro caso es de un segundo (1000 milisegundos).

```
@Scheduled(fixedRate = 21600000, initialDelay = 1000)
public void downloadDictionaries () throws ProposalsException, IOException {
    String action = "downloadDictionaries";
    DictionaryCollection dicCollection = getDictionaries();
    Dictionary[] dictionaries = dicCollection.getDictionaries();
    for (Dictionary dic : dictionaries) {
        String dicName = dic.getName();
        downloadDictionary(dicName);

        int index = StringUtils.indexOf(dicName, ".res");
        String name = StringUtils.substring(dicName, 0, index);
        File oldDicName = new File(DICTIONARY_DIRECTORY + name + "_updated.res");
        File newDicName = new File(DICTIONARY_DIRECTORY + dicName);
        boolean success = oldDicName.renameTo(newDicName);
        if ( !success) {
            getLog().warn(this, action, "Error al descargar el diccionario" + dicName);
        }
    }
}
```

Figura 15. Ejemplo de anotación @Scheduled para la descarga de diccionarios cada seis horas

Simultáneamente al desarrollo del microservicio, en la aplicación *Ebimap Web* se fue desarrollando la interfaz gráfica de las propuestas y el gestor que permite realizar las llamadas al microservicio de Propuestas. De manera que, a medida que se implementaban los endpoints del microservicio de Propuestas, se desarrollaban los elementos del frontend y la capa del backend del *Ebimap Web*. Los elementos del frontend incluyen diferentes componentes de JavaScript que tiene la biblioteca DHTMLX para la creación de aplicaciones web. Al tratarse de componentes altamente personalizables y editables facilitan el diseño de la interfaz gráfica de usuario para las propuestas. En el anexo 11.*3 Pantalla de edición de propuestas PGS* puede observarse la interfaz gráfica de usuario para los objetos PGS. El anexo 11.4 muestra la pantalla de generación del odt de la propuesta y en el anexo 11.5 la comparación entre propuestas.

El backend del *Ebimap Web* está compuesto por los controladores responsables de procesar las solicitudes REST que envía en frontend y devolverle los datos una vez procesados para mostrar en la interfaz gráfica la respuesta. El servicio, por otro lado, recibe los datos del controlador y crea el objeto que será enviado al microservicio *Proposal*. Además, también es el encargado de transformar la información recibida del microservicio en el objeto que se enviará al frontend.

7. Implantación

En este apartado se presenta la fase de implantación del microservicio *Proposals* y viene determinada por el despliegue de este. El despliegue de una aplicación, entendido desde el punto de vista arquitectónico, define la estructura del entorno en el que el software se va a ejecutar. Durante los últimos años, el proceso de despliegue ha evolucionado juntamente con la adopción de las arquitecturas basadas en microservicios. En la actualidad, las empresas que adoptan estrategias DevOps consiguen una entrega continua de sus aplicaciones desplegándolas en contenedores [5]. Los contenedores, a diferencia de las máquinas virtuales, sólo virtualizan el sistema operativo, pero no el hardware de manera que crean un entorno ligero que hospeda la aplicación y las dependencias. Los contenedores favorecen los procesos DevOps al automatizar la compilación y las pruebas, así como la publicación y descarga de contenedores desde repositorios remotos [54]. Se trata de una tecnología muy ligera que, en general, tiene un tamaño reducido. Son fácilmente escalables, portables, reusables, inmutables y autocontenidos, esto es, que empaquetan los binarios de la aplicación y sus dependencias. Proporcionan un espacio privado permitiendo a los procesos ejecutarse en un ambiente aislado por encima de la capa del sistema operativo. Estos contenedores normalmente empaquetan todas aquellas librerías necesarias para ejecutar una determinada aplicación. Además, los contenedores tienen sus propios procesos internos, sistema de archivos, espacio de nombres, direcciones ips, interfaces de red, librerías del sistema operativo, dependencias y otros elementos de configuración. Los contenedores permiten gestionar los microservicios independientemente del lenguaje o la tecnología con la que estén desarrollados. Permiten el acceso a los servicios ya que, con independencia de la tecnología usado, los microservicios exponen su API REST. Una de las soluciones frecuentemente usada para generar contenedores de aplicaciones es la proporcionada por Docker. Se trata de una tecnología que posibilita construir y ejecutar contenedores ligeros basados en el kernel de Linux [55].

Para desplegar un microservicio como contenedor, se debe empaquetar como una imagen contenedora. Esta imagen es una plantilla de lectura que contiene las instrucciones necesarias para la creación del contenedor. Para crear esta imagen es necesario crear el Dockerfile, que es el archivo que describe como construir la imagen contenedora. En la figura 16 se muestra el detalle del Dockerfile para el microservicio de Propuestas. A partir

de este Dockerfile es posible construir la imagen contenedora mediante el comando *docker build* y generar una instancia de la imagen mediante *docker run*.

```
FROM java:8-jdk
RUN groupadd -g 31337 edicom && \
    useradd -u 31337 -r -g edicom edicom
COPY dockerInit.sh /dockerInit.sh
RUN sh /dockerInit.sh

USER edicom
WORKDIR /
ADD target/Proposals-0.0.1-SNAPSHOT.jar app.war
RUN mkdir -p /var/tmp/

# Idiomas
RUN mkdir -p /var/tmp/IDIOMAS/
COPY  ./src/main/resources/dictionaries/*.res /var/tmp/IDIOMAS/

COPY entrypoint.sh /entrypoint.sh
EXPOSE 8080
ENTRYPOINT ["sh", "/entrypoint.sh"]
```

Figura 16. Dockerfile del microservicio *Proposals*

A continuación, se describe con más detalle las propiedades del Dockerfile del microservicio de Propuestas:

FROM java:8.jdk indica a Docker usar java:8-jdk como la versión de la imagen base.

RUN ejecuta cualquier comando en una nueva capa arriba de la imagen actual y confirma el resultado. La imagen resultante es usada en el siguiente paso del Dockerfile. Específicamente con este comando se añade un usuario y grupo al sistema. Este comando se vuelve a ejecutar en el Dockerfile para la creación del directorio /var/tmp/ y /var/tmp/IDIOMAS/

USER establece que el usuario de la imagen es edicom

WORKDIR establece el directorio de trabajo que, en este caso, será el directorio raíz.

ADD target/Proposals-0.0.1-SNAPSHOT.jar app.war añade el archivo binario de la aplicación al contenedor con en nombre del archivo de destino especificado. En este caso, el Docker build copia target/Proposals-0.0.1-SNAPSHOT.jar al contenedor como app.war

COPY entrypoint.sh /entrypoint.sh permite copiar el archivo entrypoint.sh desde el origen y lo añade al path /entrypoint.sh del contenedor y los diccionarios desde la carpeta de recursos del proyecto al directorio /var/tmp/IDIOMAS/.

EXPOSE 8080 le dice al contenedor como realizar el mapeado del puerto. Asocia el 8080 con el puerto externo

ENTRYPOINT [“sh”, “/entrypoint.sh”] indica al contenedor que ejecutable por defecto usará el contenedor cuando empieza. Permite la configuración del contenedor.

En una organización donde se producen decenas o cientos de contenedores, gestionar manualmente cada uno de ellos resulta inviable. Para solventar este problema surgen herramientas de gestión de clústeres que permiten, entre otras muchas cosas, desplegar automáticamente contenedores de aplicaciones, escalar de forma automática y manual instancias de aplicaciones cuando sea necesario, gestionar el estado de las aplicaciones, nodos y clústeres, optimizar los recursos distribuyendo la carga de trabajo de los contenedores sobre las diferentes máquinas disponibles, asegurar la disponibilidad del servicio tratando de forma automática los fallos de servicio.

Entre las herramientas de gestión de clústeres más usadas en el mundo empresarial se encuentra Kubernetes que permite gestionar contenedores de aplicaciones mediante un clúster de nodos [53]. Kubernetes maneja los conceptos de master, nodos y pods. En kubernetes un cluster está formados por el conjunto de master y nodos. El nodo master es el encargado de distribuir la carga de trabajo en el resto de los nodos, que son sencillamente una máquina virtual o una máquina física. Un nodo puede contener varios pods, que son las unidades de computación desplegables más pequeñas que se pueden crear un gestionar. Un pod, por tanto, pueden estar formado por uno o más contenedores.

Para desplegar un servicio en kubernetes es necesario definir el despliegue y para ello se hace uso de un archivo de tipo YAML. En la Figura 17 podemos se muestra el detalle del archivo de despliegue para el microservicio de Propuestas.

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: $CI_PROJECT_NAME                    # spms
  namespace: $CI_ENVIRONMENT_NAME           # testing
  labels:
    app: $CI_PROJECT_NAME                   # spms
    stage: $CI_COMMIT_REF_SLUG              # master: lower-case branch, 63-byte, alphanumeric
    commit: "$X_CI_COMMIT_SHA_SHORT"        # short commit hash
spec:
  replicas: 1
  progressDeadlineSeconds: 600
  selector:
    matchLabels:
      app: $CI_PROJECT_NAME                 # spms
      stage: $CI_COMMIT_REF_SLUG            # master: lower-case branch, 63-byte, alphanumeric
  template:
    metadata:
      labels:
        app: $CI_PROJECT_NAME               # spms
        stage: $CI_COMMIT_REF_SLUG          # master: lower-case branch, 63-byte, alphanumeric
        commit: "$X_CI_COMMIT_SHA_SHORT"    # short commit hash
    spec:
      containers:
      - name: $CI_PROJECT_NAME              # spms
        image: $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_SLUG:$X_CI_COMMIT_SHA_SHORT
        env:
          - name: "X_EDICOM_STAGE"
            value: "$CI_ENVIRONMENT_NAME"
          - name: "CONFIG_URL"
            value: "http://config-server"
          - name: "CONFIG_PORT"
            value: "80"
        resources:
          requests:
            ephemeral-storage: "1Gi"
            memory: "500Mi"
            cpu: "200m"
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 240
          periodSeconds: 3
          successThreshold: 3
          timeoutSeconds: 3
        livenessProbe:
          httpGet:
            path: /actuator/liveness
            port: 8080
          initialDelaySeconds: 300
          periodSeconds: 10
          timeoutSeconds: 10
      imagePullSecrets:
          - name: gitlabr
      dnsConfig:
        options:
          - name: ndots
            value: "2"
---
```

Figura 17. Archivo YAML para el despliegue del microservicio *Proposals* en Kubernetes

En este archivo la propiedad *kind* en la línea 2 especifica que se trata de un objeto de tipo Deployment. En la línea 3 se especifican los metadatos como por ejemplo el nombre del proyecto. Dentro de la propiedad *spec* se establece el número de réplicas del pod (línea 11), que en nuestro caso será de 1. En la línea 20 establece para cada *pod* una etiqueta app cuyo valor es mismo que el nombre del proyecto. En la línea 23 se definen las especificaciones del deployment y, en la propiedad *env* de la línea 27, se indican las variables entorno del contenedor. En la línea 40 se establece el puerto del contenedor. A partir de la línea 41 se configura Kubernetes para invocar el endpoint que chequea la salud del sistema, esto es, permite a Kubernetes determinar la salud de la instancia del servicio. Hay dos chequeos diferentes. Uno de ellos es *readinessProbe* que se usa para determinar si se debería enrutar tráfico a una instancia del servicio. En este archivo de despliegue, kubernetes invoca el endpoint /actuator/health cada tres segundos después de un retraso inicial de 240 segundos para que dé tiempo a inicializar. Tras tres intentos de éxito consecutivo de readinessProbes kubernetes considera que el servicio está listo. Si, por el contrario, el número consecutivo de readinessProbes falla, se considera que el servicio no está listo. Kubernetes, por tanto, solo enrutará tráfico al servicio cuando el readinessProbes indica que el servicio está disponible. El segundo endpoint para comprobar la salud del servicio es el *livenessProbe* que determina si Kubernetes debería finalizar o comenzar una instancia del servicio. Si el número de livenessProbes consecutivos falla (por defecto 3) Kubernetes terminará y comenzará de nuevo el servicio.

Como los pods tienen una dirección ip asignada de forma dinámica no son útiles a la hora de que el cliente realice una petición HTTP. Este problema puede resolverse definiendo un servicio en Kubernetes. Un servicio es un objeto que proporciona a los clientes de uno o más pods un endpoint estable. Tiene una dirección ip y un nombre DNS que resuelve el problema de la dirección ip. El servicio enruta el tráfico a los pods definidos en el archivo de despliegue.

En la Figura 18 se muestra el archivo YAML para la definición del servicio en Kubernetes. En la línea 64 se especifica que se trata de un Servicio. En la línea 66 se especifica el nombre del servicio y también del DNS. En la línea 74 se especifica el puerto expuesto y en la 76 el puerto del contenedor al que enrutar el tráfico. En la propiedad app de selector (línea 78) se indica el contenedor al que enrutar el tráfico. Este es el punto clave de la definición del servicio ya que se selecciona el pod objetivo. Esto es, se seleccionan aquellos pods que tienen en app el valor del proyecto.

Finalmente es necesario definir otro objeto, Ingress, que permite gestionar accesos externos a los servicios en un cluster. En el archivo de despliegue se configura este objeto se forma similar a los anteriores. En la línea 82 se especifica que se trata de un objeto Ingress y en la propiedad name de los metadatos se indica el nombre del proyecto. En las reglas se especifica que las solicitudes entrantes se enruten al puerto 8080 del servicio llamada con el nombre del proyecto. La sección tls marca la ruta de entrada que usa el secretName para el host especificado.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: $CI_PROJECT_NAME                    # spms
  namespace: $CI_ENVIRONMENT_NAME           # testing
  labels:
    app: $CI_PROJECT_NAME                   # spms
    stage: $CI_COMMIT_REF_SLUG              # master: lower-case branch, 63-byte, alphanumeric
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: $CI_PROJECT_NAME                   # spms
    stage: $CI_COMMIT_REF_SLUG              # master (branch, lower-case, 63-byte, alphanumeric)
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: $CI_PROJECT_NAME                    # spms
  namespace: $CI_ENVIRONMENT_NAME           # testing
  annotations:
    kubernetes.io/ingress.class: nginx-$CI_ENVIRONMENT_NAME
  labels:
    app: $CI_PROJECT_NAME                   # spms
    stage: $CI_COMMIT_REF_SLUG              # master (branch, lower-case, 63-byte, alphanumeric)
spec:
  rules:
  - host: $X_CI_HOSTNAME                    # echo $CI_ENVIRONMENT_URL | awk -F '[:/]+' '{ print $2 }'
    http:
      paths:
      - backend:
          serviceName: $CI_PROJECT_NAME     # spms
          servicePort: 8080
  tls:
  - hosts:
    - $X_CI_HOSTNAME                        # echo $CI_ENVIRONMENT_URL | awk -F '[:/]+' '{ print $2 }'
    secretName: $X_CI_BASEHOSTNAME          # echo $X_CI_HOSTNAME | awk -F"." '{print $(NF-1)"."$NF}'

```

Figura 18. Archivo YAML para la definición del servicio e Ingress del microservicio *Proposals* en Kubernetes

8. Pruebas

Dentro de la construcción de microservicios, la etapa de pruebas supone una de las fases más importantes ya que al introducirse más cantidad de componentes en un servicio hay una mayor probabilidad de fallo [56]. El propósito de las pruebas es el de verificar el comportamiento del sistema. Permiten identificar bugs en etapas tempranas del desarrollo de manera que el producto, en su paso a producción, alcance la calidad deseada.

Aunque a nivel de microservicios se pueden emplear múltiples estrategias de testeo, las pruebas unitarias y funcionales son altamente recomendadas en estos sistemas [57]. Por un lado, las pruebas unitarias posibilitan la comprobación del correcto funcionamiento de una unidad de código. Estas pruebas son automatizables, completas, independientes y

repetibles y permiten a los desarrolladores entender mejor el propósito de la unidad de código que se está probando, asegurándose de su correcto funcionamiento [58]. Las pruebas funcionales, por otro lado, validan el sistema de acuerdo con los requisitos funcionales. El objetivo de estas pruebas es comprobar la funcionalidad de una aplicación software al proporcionar un determinado input al sistema verificando el output contra los requisitos funcionales. Además, permiten descubrir defectos que el programador no ha tenido en cuenta, prevenir defectos o asegurar que el resultado reúne los requisitos. Estas pruebas tienen en cuenta la perspectiva del usuario por lo que se llevan a cabo bajo un escenario del mundo real. La evaluación del sistema bajo estas pruebas permite, por tanto, mejorar el uso del sistema actual y la calidad del producto software [59].

El testeo unitario permite verificar la coherencia del resultado de ejecutar un fragmento de código de nuestro programa [60]. Existen numerosos frameworks de pruebas unitarias para Java, sin embargo, en nuestro proyecto usamos Junit, concretamente Junit 4. Junit es uno de los frameworks más usados y populares para realizar pruebas unitarias en programas Java y está basado en anotaciones. Tiene una buena integración con frameworks de apoyo como Mockito. Mockito es un framework open source de gran utilidad ya que uno de los problemas frecuentes a la hora de realizar tests unitarios es la presencia de dependencias de los métodos sobre otros servicios de la aplicación [60]. En estos casos se emplea normalmente lo que se llama un test doble que es un objeto que simula el comportamiento de la dependencia. Hay diferentes test dobles, sin embargo, durante el testeo del microservicio de empleó lo que se denomina mock. El mock es usado por el test para verificar que el test unitario invoca correctamente la dependencia. En definitiva, Mockito proporciona una API que permite mockear dependencias de manera que el test se ejecute de forma independiente y aislada.

En la Figura 19 podemos observar un ejemplo de clase de test para el manager ProposalManager. La anotación @RunWith(MockitoJunitRunner.class) ofrece una validación automática para evitar cometer errores en el uso del framework, esto es, informa de los potenciales errores en la utilización del framework. Con @InjectMock creamos una instancia de la clase e inyecta todos los mocks que son creados con la anotación @Mock. El test unitario propiamente dicho viene etiquetado con @Test. Dentro del método es importante destacar estructura when(…).thenReturn(…) del framework Mockito que permite especificar una condición y el valor de retorno de dicha condición. En este ejemplo se establece que, cuando se llama al método getProposalTargetNames del mock proposalService con cualquier String en sus dos parámetros devolverá siempre el HashMap especificado.

```
21
22  @RunWith(MockitoJUnitRunner.class)
23  public class ProposalManagerTest {
24
25      private static final String REPOSITORY = "Prueba";
26      private static final String XPATH = "Prueba";
27
28      @InjectMocks
29      private ProposalManager proposalManager;
30
31      @Mock
32      private ProposalService proposalService;
33
34      @Test
35      public void getProposalTargetsNamesTest () throws EbimapClientException, ProposalsException {
36
37          Map<String, String> targetNames = new HashMap<String, String>();
38          targetNames.put("1", "target1");
39          targetNames.put("2", "target2");
40
41          when(proposalService.getProposalTargetsNames(anyString(), anyString())).thenReturn(targetNames);
42
43          Map<String, String> proposalTargetNames = proposalManager.getProposalTargetsNames(REPOSITORY, XPATH);
44
45          assertTrue("Debera devolver datos", proposalTargetNames != null);
46      }
```

Figura 19. Ejemplo de test unitario para el microservicio *Proposals*

La integración continua permite que los commits realizados frecuentemente por los desarrolladores lancen la compilación del código. En esta etapa es importante que los tests se lancen también de forma automática y continua para garantizar la integridad del sistema. Por tanto, es altamente recomendable que el pipeline CI/CD contenga una etapa de pruebas que verifiquen el código. Si las pruebas fallan entonces el usuario será notificado de manera que pueda corregir el origen del fallo. En nuestro proyecto Gitlab posibilita integración y despliegue continuo. Gitlab, en caso de fallo en la etapa de pruebas tras una petición de mergeo, muestra un informe para identificar fácil y rápidamente el fallo. El testeo automático, por tanto, juega un papel significativo en las estrategias DevOps.

A diferencia de las aplicaciones monolíticas, una arquitectura basada en microservicios está constituida por componentes autónomos e individualizados. Debido a la naturaleza de esta arquitectura es de especial importancia probar sus puntos de integración ente sus diferentes componentes. En nuestro caso, la aplicación *Ebimap Web* se comunica mediante REST con el microservicio *Proposals*. Esta comunicación debe realizarse de acuerdo con su API REST, que incluye los endpoints y la estructura de la petición, así como el cuerpo de la respuesta. El uso de una API permite asegurar que el componente cliente esté de acuerdo con el "contrato" establecido en las especificaciones de la API. Esto es, el cliente debe enviar una petición HTTP con una estructura apropiada al endpoint adecuado y el servicio debe enviar de vuelta la respuesta esperada.

Para implementar estos test de integración se emplean tests end-to-end o, más abreviadamente, e2e, y consisten en peticiones HTTP y respuestas HTTP simulando escenarios que ocurren en entornos de producción. En este tipo de tests se especifica la petición HTTP que será enviada al cliente y la respuesta que el cliente espera de vuelta.

Estas pruebas, no obstante, son llevadas a cabo en el Departamento de Calidad de EDICOM a diferencia de las pruebas unitarias que son implementadas por el propio desarrollador. Sin embargo, estas pruebas también son automatizadas formando parte del pipeline para la integración continua.

Una vez implementada una nueva funcionalidad en el servicio de Propuestas es necesario revisar si el sistema cumple con las especificaciones logrando su cometido. Las pruebas de aceptación, al igual que las pruebas e2e, son llevadas a cabo en el Departamento de Calidad de EDICOM. El equipo de calidad es, por tanto, el encargado de probar la nueva funcionalidad de la aplicación o modificación de ésta en un entorno de pruebas antes de actualizar la aplicación en un entorno productivo.

Para la aplicación de estas pruebas el desarrollador, de acuerdo con las especificaciones del nuevo requisito, redacta en un documento los cambios que ha realizado en el sistema. En este documento se detalla, además, el objetivo de la prueba, los pasos a seguir para reproducir la funcionalidad, las partes del sistema a los que afecta el cambio realizado en la aplicación y los riesgos que supone dicho cambio. Con esta información, el tester prueba la funcionalidad con unos determinados inputs examinando los outputs. Si la prueba es satisfactoria el tester da por finalizada la tarea. Si, por el contrario, el tester encuentra que tras la prueba el sistema no cumple satisfactoriamente la funcionalidad o simplemente sugiere una posible mejora, la tarea es devuelta al desarrollador para su gestión.

Algunas herramientas como Sonar son añadidas al proceso de integración continúa para monitorizar la calidad del código y la cobertura de este. SonarQube es una herramienta de código abierto para la continua inspección de la calidad del código [61]. Se puede integrar con Eclipse y permite detectar bugs o bugs potenciales en el código, detecta duplicaciones, cobertura de código, problemas de complejidad, mal uso de los comentarios, etc.

9. Conclusiones

En este trabajo se ha propuesto y llevado a cabo el proceso de migración de un módulo software a microservicio en un entorno industrial. Concretamente, a partir de una aplicación monolítica se ha aislado en forma de microservicio el servicio de Propuestas de la empresa EDICOM. El microservicio ha permitido disminuir la carga de la aplicación monolítica, y dotado a los grupos de desarrollo mayor flexibilidad a la hora de experimentar con nuevas funcionalidades, lo que fomentará las estrategias DevOps a nivel empresarial. A lo largo de esta memoria se ha descrito el proceso de migración del microservicio profundizando en la solución y las herramientas empleadas para ello.

El proceso de construcción del microservicio de Propuestas fue llevado a cabo, con una estimación de aproximadamente 1450 horas de duración hasta la fecha, principalmente

por tres desarrolladores del equipo WS, Daniel Caamaño Panadero, Maksym Chmutov Derevianto y Francisco García Gonzálvez. Daniel Caamaño, es un desarrollador senior dentro del equipo con amplia experiencia en los servicios que ofrece EBIMAP y, junto con Oscar Albert, Scrum Master del equipo WS, han sido los encargados de asistir a las reuniones con los stakeholders del producto y realizar el análisis de requisitos. En la ejecución de las tareas de implementación del producto los tres desarrolladores anteriormente citados han participado independientemente de la naturaleza de estas. No obstante cabe destacar que Daniel Caamaño Panadero ha ejercido como líder experto en dirigir todo el proceso de migración y desarrollo del nuevo microservicio. Ha llevado a cabo las tareas de puesta en marcha del proyecto, generación, validación y estructura de propuestas, diseño e implementación de interfaces gráficas de usuario, corrección de bugs, etc. Maksym Chmutov Derevianto, por otro lado, ha realizado tareas de configuración del proyecto, dependencias, Swagger, despliegue, implementación de interfaces gráficas de usuario, testeos, corrección de bugs etc. Por último, Francisco García Gonzálvez, miembro de más reciente incorporación al equipo WS, ha desarrollado tareas relacionadas con la generación, comparación, fusión y validación de propuestas, gestión de los diccionarios, implementación de interfaces gráficas de usuario, testeos, corrección de bugs, etc., con un tiempo estimado de 650 horas dedicadas al proyecto.

Los microservicios son un estilo de arquitectura que posibilitan el paso rápido del producto a producción [62]. Sin embargo, los microservicios por sí solos no presentan estos beneficios a menos que se desarrollen en un contexto que apoyen estos procesos ágiles. Además, los procesos de migración y desarrollo de microservicios a escala industrial son costosos a nivel de recursos por lo que se requiere un marco que permita justificar su construcción. En el departamento de I+D de EDICOM la implantación de una estrategia DevOps en el desarrollo de sus productos permite obtener los beneficios intrínsecos que ofrecen este tipo de prácticas potenciando las ventajas del uso de una arquitectura basada en microservicios.

Entre los beneficios de la implantación de estrategias DevOps junto con el desarrollo de microservicios se observa que existe un incremento en la velocidad con la que un producto pasa a producción. Esto ocurre al evitar aquellas tareas que no aportan un valor real al negocio como, por ejemplo, la sobreestimación en el tiempo de desarrollo de un proyecto o problemas de productividad del equipo. Al llevar a cabo una metodología ágil basada en sprints, las funcionalidades migradas o desarrolladas en el nuevo microservicio se planifican en tareas con un tiempo determinado por el propio equipo de desarrollo. Este procedimiento permite ajustar de manera más fiable el tiempo que se dedicará al proyecto evitando problemas de productividad del equipo. Además, las estrategias DevOps buscan automatizar las diferentes fases del ciclo de vida del software lo que implica un aumento en la calidad del producto desarrollado, así como un incremento en la velocidad de paso a producción de este [10]. Por ejemplo, durante la fase de integración continua del microservicio *Proposals* las pruebas se lanzan de forma automática y continua para garantizar la integridad del sistema. Las estrategias DevOps también aumentan la ventaja competitiva de la empresa al tener con frecuencia nuevas versiones

mejoradas del servicio evitando tener que esperar largos periodos de tiempo como en el caso de proyectos con ciclos de vida largos. En el microservicio de Propuestas estas versiones mejoradas incorporan nuevos requisitos o mejoras solicitadas por consultores y otros clientes del servicio. Finalmente, adquirir una estrategia DevOps en la industria no solo permite reducir el coste de desarrollo de un producto, sino que también permite poner en producción rápidamente dicho producto reduciendo la amenaza de empresas competidoras.

Un aspecto que resulta clave en la migración y desarrollo de un microservicio es el conocimiento de la motivación y el valor del negocio [63]. En nuestro caso, tener un buen entendimiento del motivo que ha llevado decidir el servicio de Propuestas como un microservicio aislado de la aplicación monolítica ha permitido a los ingenieros desarrollarlo de forma rentable, de manera que la selección del microservicio estuviese justificada desde el punto de vista del negocio. Además, durante la ejecución del proyecto se ha percibido que el concepto de microservicio se encuentra muy ligado con todo el proceso de desarrollo de producto y con el propio equipo que lo construye. En este sentido los desarrolladores generan un sentimiento de propiedad hacia microservicio que deriva en la responsabilidad de generar un producto de calidad. Durante todo proceso de desarrollo del proyecto el equipo es consciente del valor del negocio generando un producto que cumpla las metas y objetivos para satisfacer las necesidades del usuario. Es importante que los miembros del equipo no solo conozcan el producto técnicamente, sino que también deben comprender el funcionamiento y objetivos de la aplicación y las operaciones de negocio que aporta a la empresa. El desarrollo ágil también promueve tener equipos autoorganizados de manera que sean capaces de distribuir las responsabilidades y sean capaces de tomar decisiones logrando los objetivos como un equipo. La necesidad de comunicación y colaboración entre los miembros del equipo es sumamente importante en el proceso de migración y desarrollo de microservicios.

La definición de un proceso para el ciclo de vida del microservicio siguiendo una estrategia DevOps resulta esencial a la hora de generar un mayor valor empresarial y capacidad de respuesta. En la siguiente figura se especifican las principales etapas dentro el proceso DevOps para la migración y desarrollo del microservicio que se han detectado en este trabajo. La primera etapa identificada en el proceso de generación del microservicio *Proposals* es la de *desarrollo ágil*. Esta etapa se ha llevado a cabo de forma ágil, permitiendo adaptar de forma rápida el desarrollo del microservicio a los nuevos requisitos y necesitades. En esta etapa, además, se han identificado todos aquellos requerimientos importantes que dan valor al negocio del servicio de Propuestas. En el caso de una migración también es importante priorizar qué funcionalidades básicas se migrarían primero para desarrollar a posteriori las funciones segundarias. En la etapa de *integración continua* se ha automatizado la compilación del código producido por los miembros del equipo de desarrollo. Como parte del pipeline de compilación en esta fase se verifican algunos aspectos de calidad del código como las pruebas unitarias o cobertura de código y, como resultado de esta este proceso, se entregan los archivos binarios a un repositorio de artefactos. La etapa de *testeo continuo* está parcialmente solapada con la

etapa de integración continúa. Esta etapa permite aumentar la calidad del producto que se produce. Las pruebas llevadas a cabo en esta fase son pruebas unitarias automatizadas en el pipeline del producto. En nuestro microservicio, además, el departamento de calidad de la empresa se encarga de la automatización de las pruebas de integración, así como de realizar pruebas funcionales en un entorno de test que verifiquen la funcionalidad contra los requisitos del sistema. Finalmente, en la etapa de *entrega continua* el producto para a producción lo que implica actividades como el despliegue y la provisión de una infraestructura. Esta fase es importante debido al feedback que tenemos sobre el microservicio por parte de los usuarios de este. Por tanto, de esta fase procede la mayor parte de la información necesaria para mejorar el producto en sucesivas versiones.

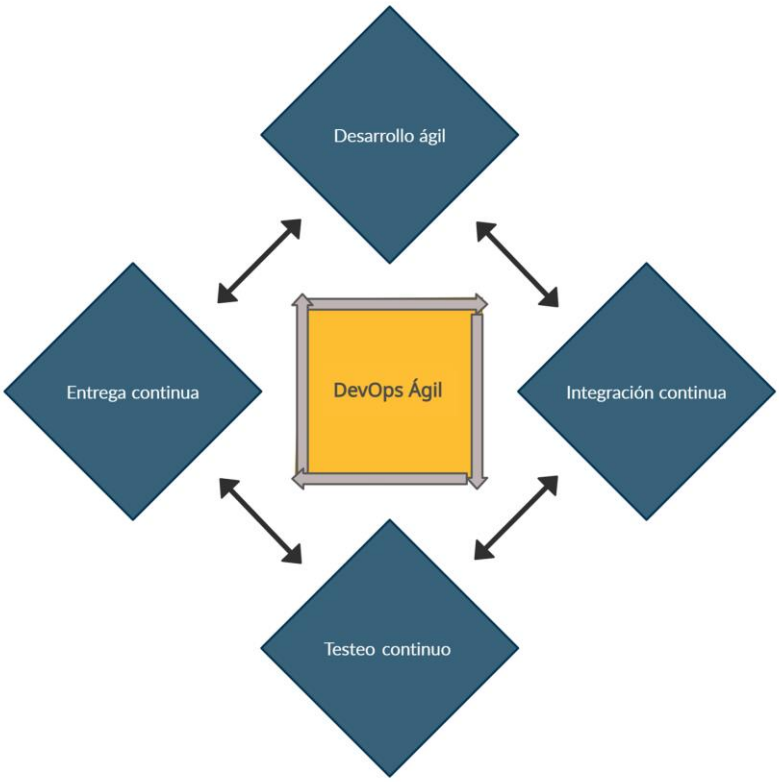

Figura 20. Etapas definidas para el proceso de migración y desarrollo del servicio de Propuestas

Un aspecto fundamental en el desarrollo y migración es el uso de herramientas que nos permiten llevar a cabo satisfactoriamente cada una de las etapas anteriormente expuestas. Estas herramientas abarcan el ciclo de vida de desarrollo de software desde el control de versiones, testeo o despliegue del microservicio. El principal objetivo de estas herramientas es posibilitar las estrategias DevOps de manera que se permita realizar frecuentes entregas del producto automatizando tantas tareas y procesos como sea posible. En el desarrollo de nuestro microservicio, dependiendo de la etapa del ciclo DevOps, podemos clasificar las herramientas empleadas como sigue:

- Desarrollo ágil. En el desarrollo del microservicio se utilizó Eclipse como IDE y Maven para la gestión, construcción y compilación del proyecto gestionando las

dependencias. Para la gestión ágil del trabajo se usó una aplicación interna de la propia empresa.

- Integración continua. GitLab es la principal herramienta que posibilita esta etapa. No obstante, este servicio de desarrollo de software colaborativo es igualmente relevante en el resto de las etapas para el desarrollo del microservicio.
- Testeo continuo. En esta fase las bibliotecas de JUnit y el framework Mockito posibilitaron la realización de pruebas unitarias del código del microservicio. La plataforma SonarQube realizó en análisis estático del código fuente para la obtención de métricas con las que mejorar la calidad del código.
- Entrega continua. Docker y Kubernetes son las principales herramientas de esta etapa permitiendo la automatización del despliegue del microservicio en contenedores.

10. Trabajos futuros

Aunque la investigación de microservicios en contextos DevOps es un área relativamente reciente, la mayoría de los estudios publicados enfatizan es aspectos teóricos y pocos a nivel industrial. Son necesarios más estudios que profundicen en el desarrollo los microservicios desde el punto de vista organizacional. Sería interesante, por ejemplo, incorporar trabajadores de ámbitos empresariales que reflejen sus experiencias al aplicar microservicios en procesos DevOps. Otra de las dificultades a la que nos enfrentamos cuando construimos o migramos un microservicio es la escasa información sobre qué técnicas y herramientas son las más adecuadas dependiendo del entorno industrial. Además, los trabajos sobre microservicios señalan los aspectos positivos de los microservicios, sin embargo, existe una carencia sobre en qué contextos industriales una migración es contraproducente. La mayoría de los estudios describen el desarrollo de microservicios desde una perspectiva muy generalista y, a veces, es complicado proyectar este conocimiento al sistema que estamos construyendo. Trabajos como el que se presenta en este proyecto fin de grado serían necesarios para señalar particularidades en la construcción de microservicios. Futuras investigaciones deberían apuntar en esta dirección. Esto ayudaría significativamente a reducir la diferencia entre la investigación académica y la práctica

A nivel industrial, este trabajo ha permitido al equipo mejorar en la práctica de la migración y desarrollo de microservicio. Habiendo desarrollado un nuevo microservicio a partir del módulo de propuestas sería posible plantearse, teniendo en cuenta la experiencia ganada durante el proceso, la migración de otros módulos dentro del servicio *Ebimap*. Estructurar este servicio en microservicios permitiría alcanzar múltiples beneficios entre los que destacaría mejora en el tiempo de mantenimiento y respuesta a la

implantación de nuevos requisitos de los stakeholders y la estimulación del proceso de innovación tecnológica a nivel empresarial.

11. Referencias

[1] "Java 2 Platform, Enterprise Edition (J2EE) Overview" 2022. https://www.oracle.com/java/technologies/appmodel.html#1 (accessed Mar. 06, 2022).

[2] A. Megargel, V. Shankararaman, D. K. Walker, A. Megargel, and V. Shankararaman, "Migrating from monoliths to cloud-based microservices: A banking industry example" in *Software Engineering in the Era of Cloud Computing*, Singapore: Singapore Management University, 2020, pp. 85–108.

[3] A. Messina, R. Rizzo, P. Storniolo, and A. Urso, "A Simplified Database Pattern for the Microservice Architecture" in *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications*, 2016, pp. 35–38.

[4] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps" in *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, Jul. 2018, pp. 39–46. doi: 10.1109/ICSA.2018.00013.

[5] R. V. Rajesh, *Spring microservices: build scalable microservices with Spring, Docker, and Mesos*, 1st edition. Birmingham: Packt Publishing, 2016.

[6] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation" *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[7] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices" *IEEE Softw.*, vol. 35, no. 3, pp. 96–100, 2018, doi: 10.1109/MS.2018.2141030.

[8] J. Lewis and M. Fowler, "MicroServices" 2014. https://www.martinfowler.com/articles/microservices.html (accessed Dec. 06, 2021).

[9] C. Pahl, P. Jamshidi, and O. Zimmermann, "Architectural Principles for Cloud Software" *ACM Trans. Internet Technol.*, vol. 18, no. 2, pp. 1–23, 2018, doi: 10.1145/3104028.

[10] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[11] M. Waseem, P. Liang, and M. Shahin, "A Systematic Mapping Study on Microservices Architecture in DevOps" *J. Syst. Softw.*, vol. 170, no. 110798, 2020.

[12] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation" *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, 2017, doi:

10.1109/MCC.2017.4250931.

[13] D. Taibi, V. Lenarduzzi, and C. Pahl, “Continuous architecting with microservices and DevOps: A systematic mapping study” in *Communications in Computer and Information Science*, 2019, vol. 1073, pp. 126–151. doi: 10.1007/978-3-030-29193-8_7.

[14] C. Pahl and P. Jamshidi, “Microservices: A Systematic Mapping Study” in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, 2016, pp. 137–146. doi: 10.5220/0005785501370146.

[15] K. Beck *et al.*, “Manifiesto por el Desarrollo Ágil de Software” 2001. https://agilemanifesto.org/iso/es/manifesto.html (accessed Dec. 27, 2021).

[16] K. S. Rubin, *Essential Scrum: a practical guide to the most popular agile process*, 1st edition. Addison-Wesley Professional, 2013.

[17] C. Richardson, *Microservices patterns: with examples in Java*, 1st edition. Shelter Island, NY: Manning Publications, 2019.

[18] S. Newman, *Building microservices: designing fine-grained systems*, First edit. Sebastopol, California: O’Reilly Media, Inc., 2015.

[19] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps Migration to a Cloud-Native Architecture” *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, 2016, doi: 10.1109/MS.2016.64.

[20] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, “Microservices migration patterns” *Softw. - Pract. Exp.*, vol. 48, no. 11, pp. 2019–2042, Nov. 2018, doi: 10.1002/spe.2608.

[21] J. Soldani, D. A. Tamburri, and W. J. Van Den Heuvel, “The pains and gains of microservices: A Systematic grey literature review” *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018, doi: 10.1016/j.jss.2018.09.082.

[22] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, “From monolithic systems to Microservices: An assessment framework” *Inf. Softw. Technol.*, vol. 137, Sep. 2021, doi: 10.1016/j.infsof.2021.106600.

[23] U. Zdun, E. Wittern, and P. Leitner, “Emerging Trends, Challenges, and Experiences in DevOps and Microservice APIs” *IEEE Softw.*, vol. 37, no. 1, pp. 87–91, Jan. 2020, doi: 10.1109/MS.2019.2947982.

[24] C. Richardson, “Refactoring a Monolith into Microservices” Mar. 08, 2016. https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/ (accessed Aug. 15, 2022).

[25] M. Eisele, *Modern Java EE Design Patterns*, 1st edition. O’Reilly Media, Inc., 2016.

[26] A. F. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, and N. C. Mendonça, “Migrating production monolithic systems to microservices using aspect oriented programming” *Softw. - Pract. Exp.*, vol. 51, no. 6, pp. 1280–

1307, Jun. 2021, doi: 10.1002/spe.2956.

[27] A. Sen and I. Skrobot, “Implementation of DevOps paradigm to deployment and provisioning of microservices” *Issues Inf. Syst.*, vol. 22, no. 1, pp. 136–148, 2021, doi: 10.48009/1_iis_2021_136-148.

[28] M. Yousif, “Microservices” *IEEE cloud Comput.*, vol. 3, no. 5, pp. 4–5, 2016, doi: 10.1109/MCC.2016.101.

[29] D. Farley and J. Humble, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[30] S. Wickramasinghe. “The Role of Microservices in DevOps” 2021. https://www.bmc.com/blogs/devops-microservices (accessed Dec. 06, 2021).

[31] H. Chen, R. Kazman, S. Haziyev, V. Kropov, and D. Chtchourov, “Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform” in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2015, vol. 2016, pp. 25–30. doi: 10.1109/SRDSW.2015.14.

[32] “Microservices and DevOps: Better Together” https://www.mulesoft.com/resources/api/microservices-devops-better-together (accessed Dec. 06, 2021).

[33] B. Familiar and J. Barnes, *Business in Real-Time Using Azure IoT and Cortana Intelligence Suite Driving Your Digital Transformation*, 1st ed. 20. Berkeley, CA: Apress, 2017. doi: 10.1007/978-1-4842-2650-6.

[34] V. Ivanov, “Implementation of a DevOps Pipeline for Serverless Applications” in *19th International Conference, PROFES*, 2018, pp. 28–30.

[35] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, “Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab” in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2020, pp. 327–337. doi: 10.1145/3368089.3409709.

[36] “DevOps Community” https://devops.com.vn/ (accessed Feb. 28, 2023).

[37] S. Narayanan, A. S. Marucheck, and R. B. Handfield, “Electronic Data Interchange: Research Review and Future Directions” *Decis. Sci.*, vol. 40, no. 1, pp. 121–163, 2009, doi: 10.1111/j.1540-5915.2008.00218.x.

[38] Z. M. Mousavi, M. Poormazahaeri, and A. Z. Khozani, “The role of electronic data interchange in electronic commerce” *Eur. Online J. Nat. Soc. Sci.*, vol. 2, no. 3, pp. 188–194, 2013, [Online]. Available: www.european-science.com

[39] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley Professional, 2004.

[40] R. T. Fielding, “Architectural styles and the design of network-based software architectures” University of California, Irvine, 2000.

[41] B. Shafabakhsh, R. Lagerström, and S. Hacks, “Evaluating the Impact of Inter Process Communication in Microservice Architectures” in *8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020)*, 2020, pp. 55–63.

[42] F. Montesi and J. Weber, “Circuit Breakers, Discovery, and API Gateways in Microservices” Sep. 2016.

[43] R. Chandramouli, “Security strategies for microservices-based application systems” Gaithersburg, MD, Aug. 2019. doi: 10.6028/NIST.SP.800-204.

[44] “Gitlab Documentation” 2022. https://docs.gitlab.com/ (accessed Jul. 23, 2022).

[45] “Eclipse Documentation” 2022. https://www.eclipse.org/documentation/ (accessed Jul. 23, 2022).

[46] “Visual Studio Documentation” 2022. https://code.visualstudio.com/docs (accessed Jul. 23, 2022).

[47] “Spring Boot Documentation” 2022. https://spring.io/projects/spring-boot (accessed Mar. 06, 2022).

[48] B. Varanasi, *Introducing Maven A Build Tool for Today’s Java Developers*, 2nd ed. 20. Berkeley, CA: Apress, 2019. doi: 10.1007/978-1-4842-5410-3.

[49] “Swagger Documentation” 2022. https://swagger.io/docs/ (accessed Jul. 23, 2022).

[50] “DHTMLX Documentation / Suite 5.X,” 2022. https://docs.dhtmlx.com/suite5.html (accessed Mar. 06, 2022).

[51] “Webpack Documentation” 2022. https://webpack.js.org/guides/ (accessed Jul. 23, 2022).

[52] “Docker Documentation” 2022. https://docs.docker.com/ (accessed Mar. 06, 2022).

[53] “Kubernetes Documentation” 2022. https://kubernetes.io/es/docs/home/ (accessed Mar. 06, 2022).

[54] H. Kang, M. Le, and S. Tao, “Container and microservice driven design for cloud infrastructure DevOps” in *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, Jun. 2016, pp. 202–211. doi: 10.1109/IC2E.2016.26.

[55] G. Schenker, *Learn Docker - Fundamentals of Docker 19.x : build, test, ship, and run containers with Docker and Kubernetes*, Second edi. Birmingham, England; Packt, 2020.

[56] R. Heinrich *et al.*, “Performance engineering for microservices: Research challenges & directions,” in *ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering*, Apr. 2017, pp. 223–226. doi: 10.1145/3053600.3053653.

[57] M. Waseem, P. Liang, G. Marquez, and A. Di Salle, “Testing microservices architecture-based applications: A systematic mapping study” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, Dec. 2020, vol. 2020-Decem, pp. 119–128. doi: 10.1109/APSEC51365.2020.00020.

[58] M. Albarka Umar, “A Study of Automated Software Testing: Automation Tools and Frameworks” *Int. J. Comput. Sci. Eng.*, vol. 8, no. 6, pp. 217–225, 2019.

[59] M. Shi, “Software Functional Testing from the Perspective of Business Practice” *Comput. Inf. Sci.*, vol. 3, no. 4, pp. 49–52, 2010, [Online]. Available: www.ccsenet.org/cis

[60] S. Acharya, *Mastering Unit Testing Using Mockito and JUnit*. Olton Birmingham: Packt Publishing, Limited, 2014.

[61] “SonarQube Documentation” https://docs.sonarqube.org/latest/ (accessed Jul. 26, 2022).

[62] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl, “Developing self-adaptive microservice systems: Challenges and directions” *IEEE Softw.*, vol. 38, no. 2, pp. 70–79, Mar. 2021, doi: 10.1109/MS.2019.2955937.

[63] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “Microservices in Practice, Part 1: Reality Check and Service Design” *IEEE Softw.*, vol. 34, no. 1, pp. 91–98, Jan. 2017, doi: 10.1109/MS.2017.24.

12. Anexos

12.1. Pantalla de edición de una interfaz

La pantalla de creación y edición de una interfaz en la aplicación tiene el aspecto que se observa en la figura 21. Como se aprecia, la pantalla está dividida en tres secciones principales. En la sección uno se muestra el árbol de registros de la interfaz como un árbol de nodos. En este árbol se observa de manera visual las relaciones padre-hijo entre registros. En la segunda sección se muestra el conjunto de campos que tiene el registro seleccionado en el árbol de registros y las propiedades que presenta cada campo. En la tercera sección se muestran diferentes opciones relativas a la interfaz, fundamentalmente información sobre las propiedades de los registros y campos de la interfaz.

Figura 21. Estructura de una interfaz en la aplicación *EbimapWeb*

12.2. Documento odt de las propuestas

En el documento odt generado a partir de una propuesta se recoge información sobre la normativa y recomendaciones de esta, descripción detallada de la propuesta y ejemplos de cómo se debe estructurar la información. En la figura 22 se muestra un detalle sobre la información que se especifica en la propuesta. En la tabla se observa información de varios campos del registro CAB. Concretamente, para cada campo se aprecia si es de tipo alfanumérico (X) o numérico (N), la longitud total del campo, la posición en la que empezaría el campo y donde acabaría. En la descripción se indican comentarios adicionales al campo, por ejemplo, comentarios relativos a la personalización del campo como cuando queramos añadir una anotación específica que haga referencia a un determinado interlocutor. En la columna "Situación en" vendrá indicado el campo en el que se tiene que mapear la información de dicho campo y, finalmente en la columna "Condición en" indicaremos la condición que se tiene que cumplir para poder mapear la información de dicho campo.

Cabecera de la confirmación de recepción (Datos globales al documento)
Nombre del registro: CAB
Nombre del fichero físico: CAB.TXT
Obligatoriedad: Condicional
Esta sección es repetible con las siguientes restricciones:
Mínimo número de ocurrencias: 0
Máximo número de ocurrencias: 9999999

Campo	Tipo	Long.	Ini	Fin	Descripción	Situación en	Condición en
IDCAB	X	10	1	10	**Identificador de cabecera**		
NUMCON	X	35	11	45	**Número de confirmación de recepción** ***Interlocutor***: Obligatorio	[BGM.1004]	
TIPO	X	3	46	48	**Tipo de documento** Lista de valores permitidos: 352: Confirmación de recepción ***Interlocutor***: Obligatorio	[BGM.1001]	
FECDOC	X	12	52	63	**Fecha de documento** Formato: AAAAMMDD o AAAAMMDDhhmm. ***Interlocutor***: Obligatorio	[DTM.2380]	([DTM.2005] = 137)

Figura 22. Detalle del documento odt de una propuesta en el que se muestra información sobre varios campos que forman parte de la propuesta

12.3. Pantalla de edición de propuestas PGS

En este subapartado de los anexos se describen algunos de los principales aspectos desarrollados de la interfaz gráfica de usuario para las propuestas. En la figura 23 se muestra la interfaz principal que se encuentra divida en cuatro regiones encabezadas por una barra de herramientas. La barra de herramientas posibilita realizar diversas funciones sobre la propuesta como la generación del documento odt de la propuesta, comparación, fusión y validación de propuestas entre otras. En la región 1 se representa el árbol de

registros de los que consta la propuesta. La interfaz permite jugar con los registros de manera que se posibilita el despliegue de estos en caso de que tengan registros hijos. Esto facilita la inspección de la propuesta en el caso de propuestas con una estructura compleja. Además, cada vez que clicamos en cada uno de los registros se actualizan las regiones dos y tres con los datos relativos al propio registro. La región dos se compone de un grid que contiene información sobre cada uno de los campos de datos que componen el registro. La región tres es un elemento de tipo acordeón donde se puede especificar propiedades generales del registro seleccionado, interlocutores, colaboradores, revisores y scripts de la propuesta. Finalmente, en la región número cuatro se ubican diferentes opciones para el uso de las propuestas como la posibilidad de introducir ficheros de datos que sea validados por la propuesta, la salida de la propuesta en forma de documentos odt, información asociada a los procesos de generación o validación de la propuesta, etc.

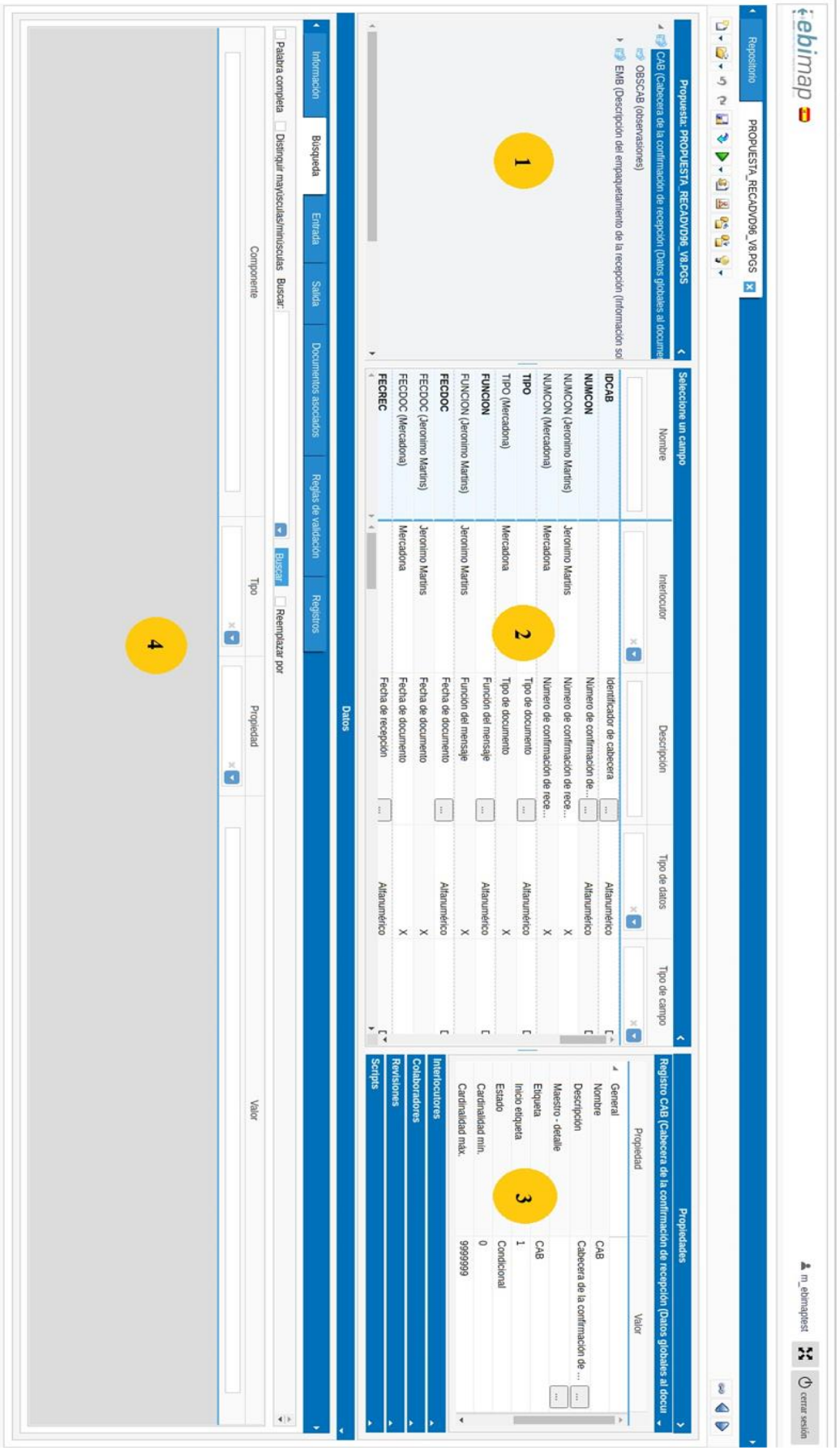

Figura 23. Pantalla de una propuesta en la aplicación *EbimapWeb*

12.4. Pantalla de generación del documento odt

En la figura 24 se representa la ventana que facilita la configuración de parámetros para la generación del documento odt de una propuesta. En la parte superior de la ventana se permite la selección de los interlocutores disponibles para los que queremos generar el documento de la propuesta. La parte inferior posibilita la selección de múltiples opciones para la generación de la propuesta como por ejemplo el tipo de propuesta o el idioma en el que se creará el documento de la propuesta.

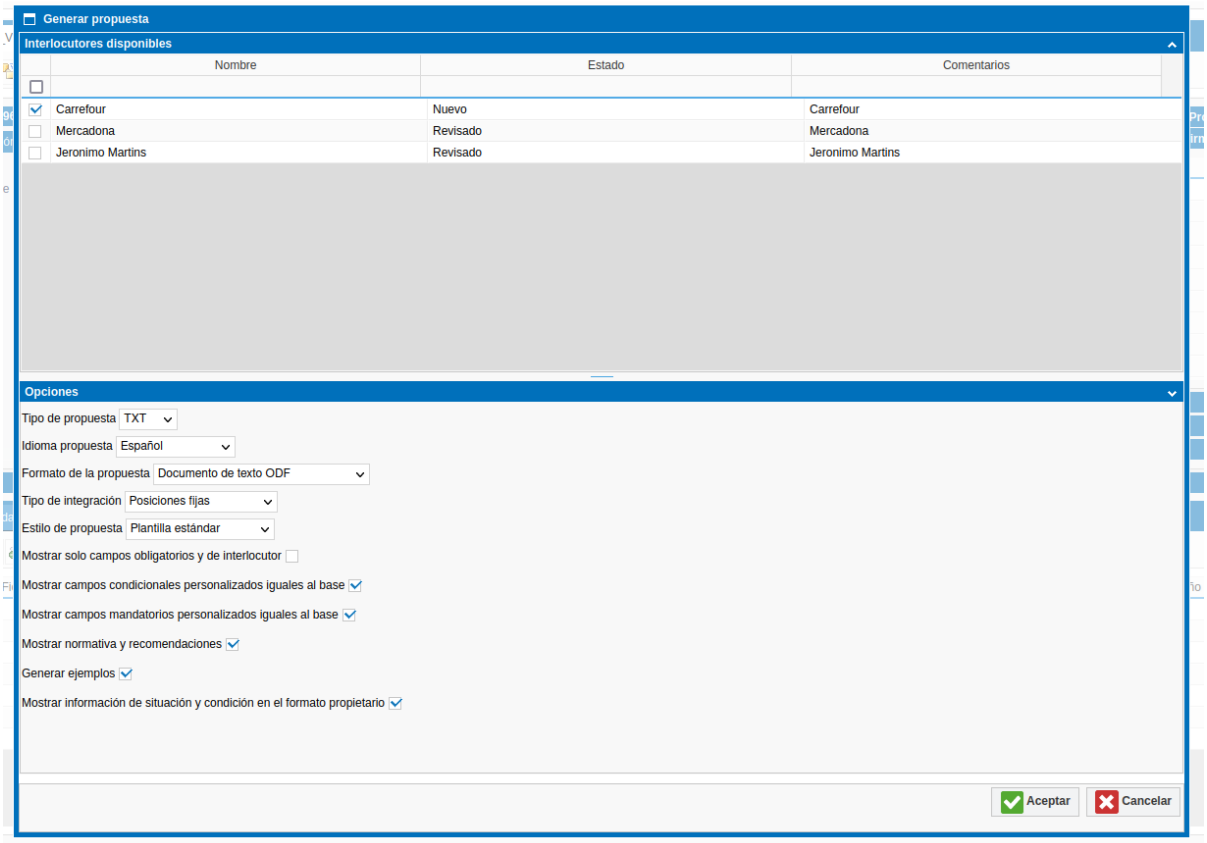

Figura 24. Pantalla que permite seleccionar las diferentes opciones para generar el documento odt de la propuesta para los interlocutores marcados.

12.5. Pantalla de comparación de propuestas

En la figura 25 y 26 se ejemplifica el proceso de comparación de propuestas. La ventana de la figura 25 permite la selección de la propuesta (localizada en la misma ruta del repositorio) que queremos comparar con la propuesta actual. Tras la elección de la propuesta se realiza la llamada que compara dos propuestas en el microservicio *Proposals* devolviendo las diferencias entre las mismas para representarlas en el grid que se observa en la figura 26. Estas diferencias pueden incluso ser exportadas en un documento CSV

tras clicar sobre el botón de “Exportar” arriba a la izquierda. Desde este grid, sería posible la fusión de ambas propuestas realizando un check en “Aplicar cambio” de la fila del grid que queramos modificar. A partir de los cambios seleccionados se volvería a llamar al microservicio *Proposals* para aplicar estas modificaciones.

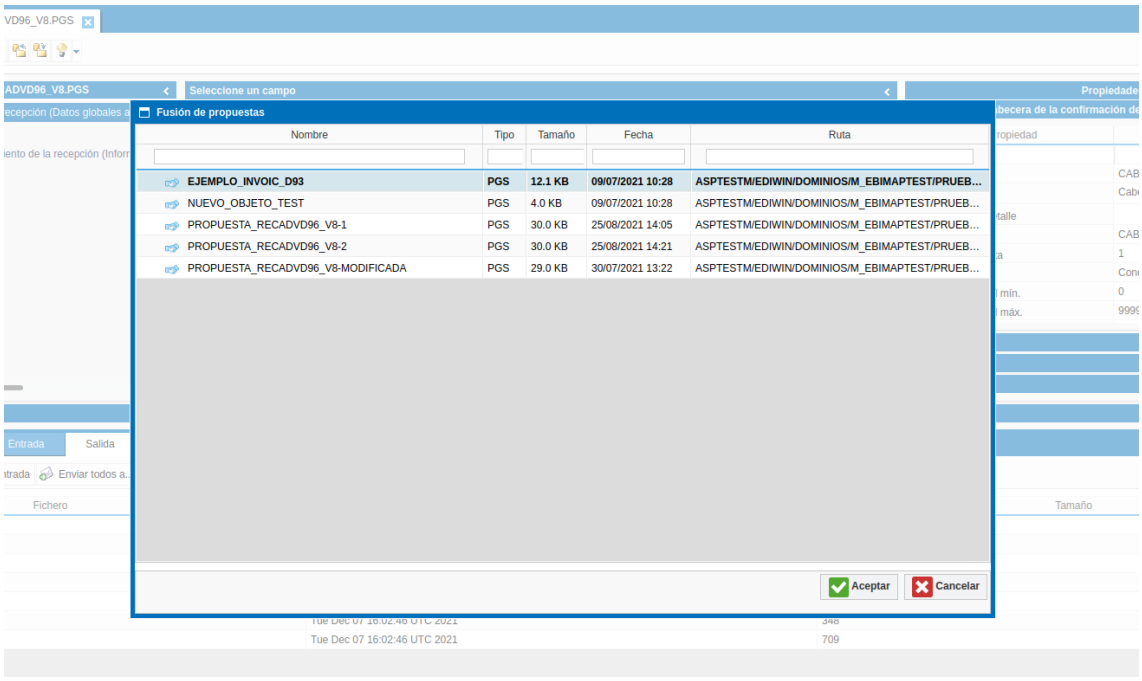

Figura 25. Ventana que permite seleccionar la propuesta a comparar

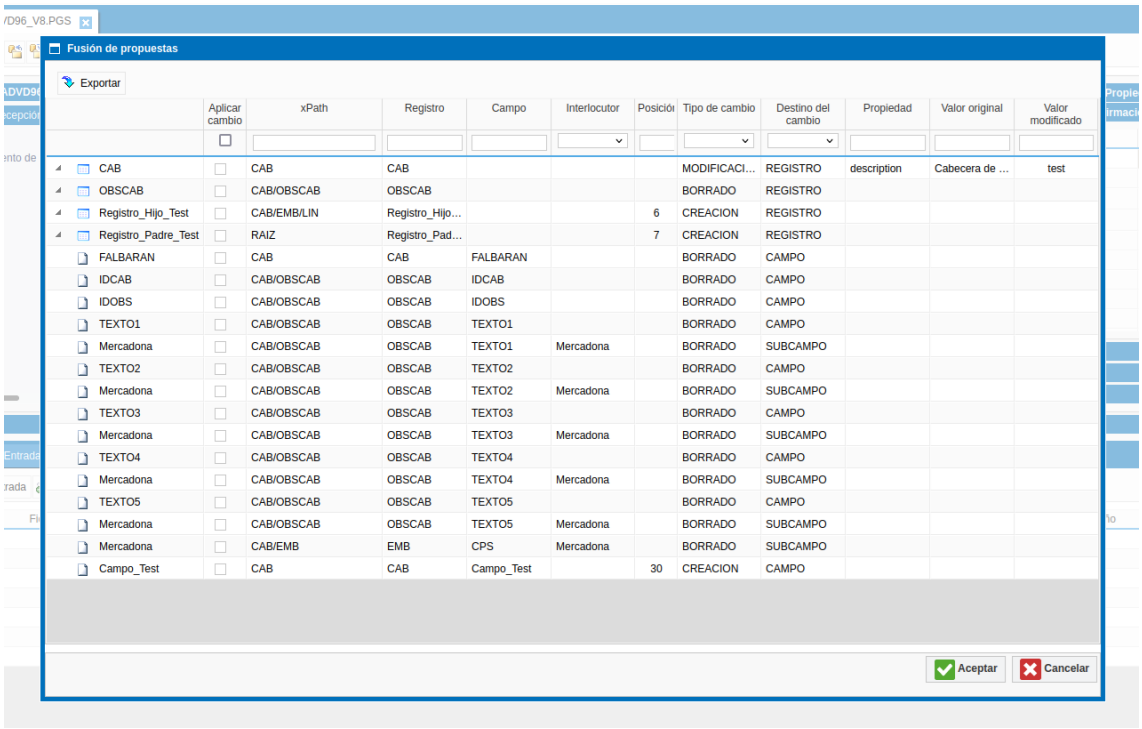

Figura 26. Ventana que muestra las diferencias entre las propuestas comparadas

12.6. Pantalla de edición de propuestas PCS

Aquí se describen algunos de las principales características de la interfaz gráfica de usuario para las propuestas PCS. Se trata de una interfaz similar a la de las propuestas PGS aunque en este caso distinguimos tres regiones (ver Figura 27). En la región uno se representa el árbol de registros de los que consta la propuesta. La región dos se compone de un grid que contiene información sobre cada uno de los campos de datos que componen el registro. En la región tres se ubican diferentes opciones para el uso de los PCS como la pestaña de información y la de entrada y salida de documentos.

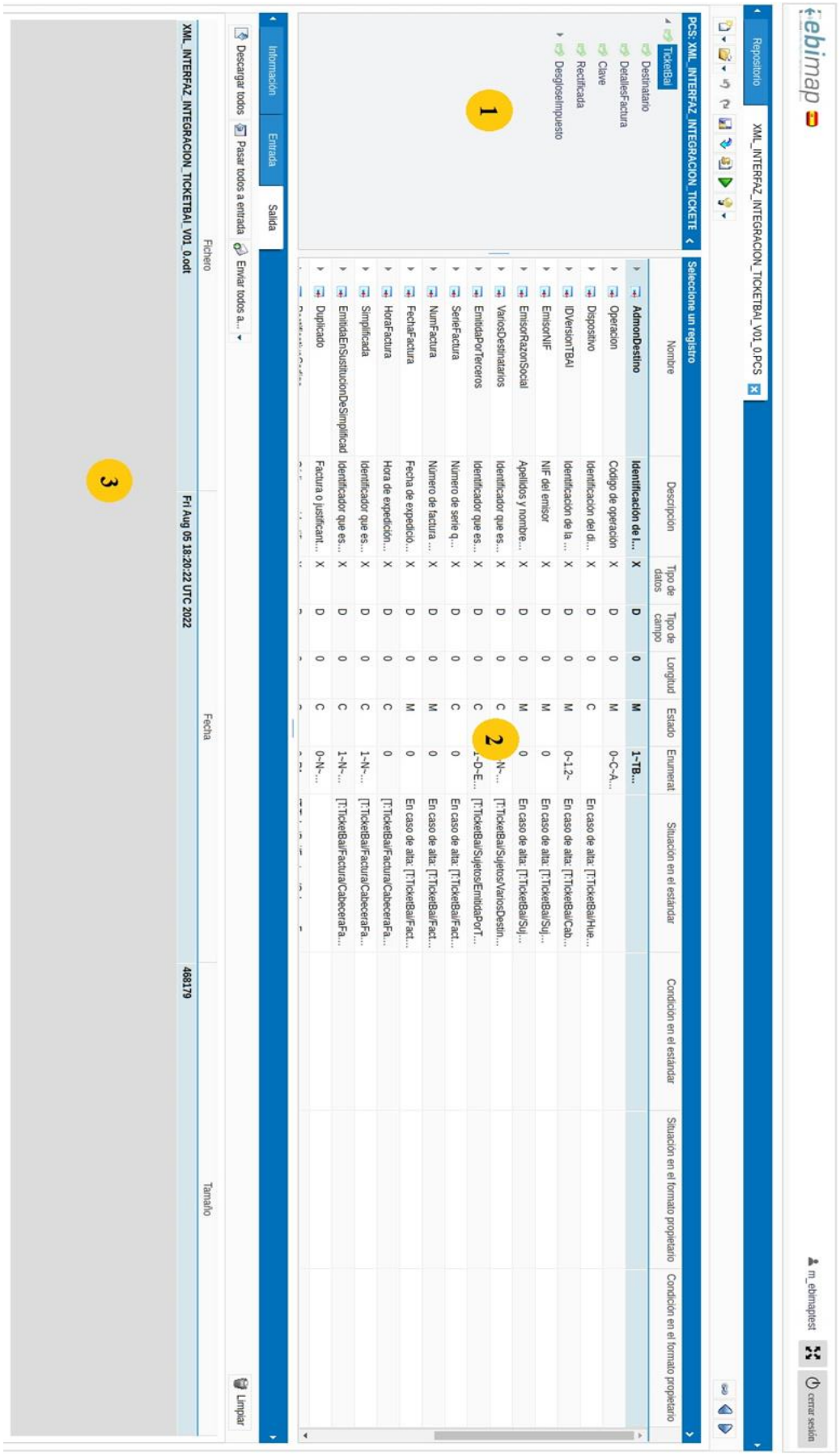

Figura 27. Pantalla de una propuesta PCS en la aplicación Ebimap Web

12.7. Objetivos de desarrollo sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

De los 17 ODS de la Agenda 2030 aprobada por las Naciones Unidas para el Desarrollo Sostenible tres se encuentran relacionados con el desarrollo del TFG.

1. Educación de calidad. Con la elaboración de un TFG se espera que los alumnos demuestren las habilidades adquiridas tras la superación de las asignaturas que conforman el grado en Ingeniería Informática. Estas habilidades no solo incluyen competencias técnicas, sino que, además, las habilidades blandas juegan un papel relevante. Asimismo, en este TFG en concreto, la dedicación intensiva del alumno al desarrollo de un proyecto en un contexto industrial, donde debe resolver problemas del mundo real, fomenta enormemente el desarrollo de habilidades profesionales. Por tanto, el desarrollo del TFG supone el compendio de todas las habilidades y competencias adquiridas por el alumno lo que deriva en una educación de calidad.
2. Trabajo decente y crecimiento económico. El crecimiento económico está estrechamente relacionado con la innovación y diversificación tecnológica ya que, generalmente, incrementa la producción y mejora los servicios. En este sentido, en el presente trabajo se profundiza en diversos aspectos relativos a la arquitectura de microservicios. Este tipo de arquitecturas, empleadas recientemente en la industria de las tecnologías de la información, permite la producción de desarrollo software más sostenible al no tener que lidiar con muchas de las dificultades del desarrollo tradicional de software. Esta tecnología hace posibles sistemas altamente productivos que mejoran los servicios ofrecidos a clientes.
3. Industria, innovación e infraestructuras. En la actualidad existe un incremento en el desarrollo de aplicaciones basadas en un conjunto de servicios independientes y autónomos. Cada uno de estos servicios, denominados microservicios, son desarrollados y desplegados de forma independiente. A escala industrial los microservicios permiten, entre otros beneficios, mejorar la productividad en el desarrollo de aplicaciones ya que pueden ser desarrollados, desplegados y gestionados de forma independiente haciendo mucho más sencillo su mantenimiento. Estos servicios individuales pueden escalar de forma independiente y permiten una mayor experimentación a la hora de desarrollar nuevas funcionalidades. Además, el concepto de microservicios ofrece facilidades a la industria a la hora de implantación de estrategias DevOps. En este TFG se ha detallado el proceso de migración de un módulo software de una aplicación monolítica a microservicio en un entorno industrial. Tras la implantación del nuevo microservicio se espera que la empresa pueda beneficiarse de las múltiples ventajas que supone una arquitectura basada en microservicios.