Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Herramienta para la visualización del estado tecnológico en una arquitectura de microservicios

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Erick de la Cruz Castellano

Tutor: Francisco Daniel Muñoz Escoí

Tutor Externo: Hernandez Herranz, Carlos

Curso 2019-2020

Resumen

El crecimiento de una empresa de servicios TIC suele venir acompañado de la reformulación de su modelo tecnológico. Frecuentemente esto significa cambiar de una arquitectura monolítica o centralizada a una construida con microservicios o distribuida. No resulta extraño que durante el proceso aparezcan dificultades que impidan la adecuada gestión de los proyectos y que se desatiendan aspectos como la calidad del código, la uniformidad de los proyectos o elevados tiempos de despliegue, requiriendo un coste de tiempo extra para tareas que antes no lo necesitaban.

Este documento propone una solución para uno de los problemas que pueden aparecer en este cambio, en concreto el de la visualización de diferentes métricas y diagramas relacionados con la calidad del software en los microservicios de los diferentes proyectos de la empresa. Por lo que se va a desarrollar una herramienta que recopilará información relevante sobre los proyectos, los cuales componen una arquitectura de microservicios y de esta manera se podrá inferir el estado tecnológico de la misma.

Como información relevante se van a considerar los análisis estáticos de código que se realizan en los proyectos a través de SonarQube, los resultados de las tareas de despliegue en producción de Jenkins y así como las versiones de Java y de determinadas librerías, también los resultados de pequeños análisis sobre los repositorios de GitHub.

La información mencionada, que se visualizará a través de paneles de Metabase, permitirá a los responsables del departamento de desarrollo de la empresa Jeff tomar decisiones sobre en qué invertir esfuerzo y recursos para mejorar la calidad del servicio que ofrecen.

Palabras clave: microservicios, SonarQube, Spring, Jenkins, calidad del software

Abstract

A software company growth is usually accompanied by its technological model reformulation. Frequently, this implies a change from a monolithic architecture to one built with microservices. In this process, difficulties arise that prevent the proper management of projects. Besides, aspects such as the quality of the code, the uniformity of the projects or long deployment times are neglected, requiring an extra time cost for tasks that previously did not need it.

This document proposes a solution to one of the problems that may appear in this change; specifically, the visualization of software quality metrics and diagrams for the existing company projects. Therefore, a tool will be developed that will collect relevant information about projects, which compose a microservices architecture, so in this way the technological status of the architecture will be inferred.

As relevant information we will consider the static code analyses carried out in projects through SonarQube, the results of Jenkins' production deployment tasks, the versions of Java and of certain libraries, as well as the results of small analysis on GitHub repositories.

That information, which will be visualized through Metabase panels, will allow the Jeff's development department to decide on which are the components or resources that need to be improved in order to boost the quality of the service they offer.

Keywords: microservices, SonarQube, Spring, Jenkins, software quality.

Índice

Tabla de Figuras

1. Introducción

No es reciente el problema de evaluar la calidad del software. Su estudio ha derivado en diferentes taxonomías que han permitido distinguir entre la adecuación a lo requerido a la aplicación, de lo óptimo de la solución empleada.

Es decir, se contrapone la calidad externa o funcional, entendida como la medida en la que la funcionalidad de la aplicación se ciñe a lo demandado por el cliente, a la interna, referida a cómo se emplea la tecnología para satisfacer esas demandas.

Dado que en este trabajo se va a hacer énfasis en la calidad interna, donde se ubica el concepto de mantenibilidad del código fuente[1], una de las soluciones que han surgido para enmendar este problema es que el programador trate de cumplir los principios SOLID (Martin, 2015), cuyo resultado es un código que a un programador ajeno a su desarrollo le cueste poco entenderlo y modificarlo.

Sin embargo, al no poder determinar cuánto de mantenible es una aplicación, el problema sigue presente. Tradicionalmente, la solución consistía en que un programador experto revisase el programa, para detectar posibles fallos o elementos poco mantenibles, pero el resultado sigue sin ser parametrizable.

Actualmente la solución pasa por la obtención de métricas que permitan objetivar la calidad de software gracias al uso de tecnologías de testeo y de la clasificación de fragmentos de código que son susceptibles de provocar fallos conocidos. Esto se conoce como inspección continua (Gaudin, 2013) y algunas herramientas, como SonarQube[2] permiten recoger estos valores.

Este avance permite la integración continua (Fowler M. , 2006), entendida como una manera de trabajar que pretende acortar el tiempo que sucede entre que un desarrollador termina una nueva característica de la aplicación y que un usuario final hace uso de la misma. Ahora que existen maneras de medir la calidad del código, se puede automatizar el proceso de despliegue, sabiendo que el código es robusto.

Cabe destacar que está presente el problema de las versiones, tanto del lenguaje como de las librerías, de las herramientas empleadas en el desarrollo de software, que determinarán cuales serán en su mantenimiento. Descuidar las versiones de

[1] Datos de la industria muestran como el 80% del tiempo que se destina a un software es al mantenimiento (Gaudin, 2013).

[2] https://www.sonarqube.org/

un producto, permitiendo que se queden atrasadas bajo el lema de “si funciona, no lo toques”, puede derivar en dificultades a largo plazo, tales como incompatibilidad con los nuevos módulos, que la incorporación de nuevas funcionalidades suponga una enorme carga de trabajo, pérdida de soporte e, incluso, tener que volver a desarrollarlo.

El último problema consiste en la visualización de esta información. La automatización de un proceso no lo exime de posibles fallos, por lo que un ojo experto debe poder recoger valores del correcto funcionamiento de estos procesos, para adelantarse a sus fallos.

Nos encontramos en un contexto cambiante en el que las empresas necesitan poder adaptarse a nuevas situaciones, las aplicaciones deben ser robustas, fáciles de mantener y a las que se le puedan añadir nuevas funcionalidades respetando las anteriores.

Por tanto, este trabajo pretende desarrollar una herramienta que aporte una visión global de la situación de un número elevado de proyectos, así como su evolución con el fin de poder tomar decisiones sobre la calidad del software.

He de señalar que este trabajo cuenta con un glosario de términos que pretende facilitar la lectura de este documento.

1.1. Motivación

La oportunidad que supone el desarrollo de un proyecto de estas características me permite aplicar mis conocimientos sobre una serie de herramientas tecnológicas no solo potentes, sino que el mercado laboral requiere de profesionales que sepan manejarse con ellas.

Además de ello, el desarrollo de mi proyecto puede suponer una contribución valiosa para una empresa, permitiéndome familiarizarme con su actividad laboral y demostrar mis habilidades como graduado a fin de acabar formando parte del equipo.

Por tanto, mis motivaciones en cuanto a aspectos tecnológicos son los siguientes:

El primero sería familiarizarme con el modelo tecnológico de Spring[3], un *framework* que engloba un conjunto de librerías que se han utilizado típicamente en las aplicaciones web construidas sobre Java. Las englobadas corresponden a

[3] https://spring.io/

Hibernate[4] , JPA[5] y Maven[6] entre otras; las utilizadas serán descritas con detalle más adelante.

Querer mejorar mis destrezas sobre este modelo de desarrollo se debe a que en los últimos años se ha hecho muy popular entre los desarrolladores, ya que aplica una capa de abstracción que facilita configuraciones que anteriormente resultaban complejas, agilizando el proceso de desarrollo del software. Motivo por el cual muchas empresas buscan a desarrolladores expertos en estas tecnologías.

Otra motivación correspondería a conocer en profundidad la herramienta SonarQube. Este tipo de herramientas de análisis del código fuente goza de popularidad entre los desarrolladores, ya que ayuda a resolver algunos de los problemas planteados en la introducción, como el análisis de la calidad interna del software.

En concreto, SonarQube permite su integración con los IDE más populares y su sistema de reglas y documenta con claridad y profundidad aspectos a mejorar del código fuente. Supone una fuente de aprendizaje y de mejora en la práctica de la programación.

La siguiente motivación alude a comprender mejor el proceso de la integración continua, ya que, siendo un concepto desarrollado teóricamente durante el grado, mis conocimientos prácticos sobre este proceso son escasos.

Se aprecia que cada vez más empresas adoptan este modelo de trabajo, ya que agiliza el desarrollo y la corrección de errores. Además, comienzan a ser demandadas a los programadores, puesto que, aunque su función principal no esté directamente relacionada con la implementación de este proceso, sí se suele desarrollar en un contexto de integración continua.

Aunque la carrera aporta una importante base de conocimiento sobre Java, es cierto que algunos conceptos más modernos quedan fuera del temario. En consecuencia, otra de mis motivaciones es la de desarrollar haciendo uso de los cambios de Java en sus últimas versiones.

En concreto, pretendo aprovechar este proyecto para aprender en el uso de las características introducidas en Java 10[7], siendo las más valiosas los *optional*, la inferencia de tipos y el uso de los *streams*.

[4] https://hibernate.org/
[5] https://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html
[6] https://maven.apache.org/
[7] https://docs.oracle.com/javase/10/

El último de los aspectos tecnológicos dentro de mis motivaciones es el de ampliar mis conocimientos al respecto del uso de las API REST (Fielding, 2000).

Aunque el grado incluía asignaturas que trataban este concepto, es un aspecto en el que quiero aumentar mis conocimientos. Es un tipo de servicios web muy utilizado, y pretendo mejorar mis conocimientos tanto en consumir estos servicios, como en ofrecerlos.

Paso a comentar las motivaciones no relacionadas directamente con la tecnología, donde la primera de ellas es la de enfrentarme a un proyecto complejo y resolverlo, pues plantea un reto donde aplicar destrezas organizativas, así como de búsqueda de información.

Finalmente, pretendo que mi proyecto sea de utilidad para la empresa, de manera que mi aportación justifique una posible contratación por el valor añadido que podría llegar a aportar.

1.2. Objetivos

El proyecto se va a desarrollar en el contexto de unas prácticas curriculares en la empresa Jeff[8]. Una *Startup*, fundada en 2016, cuyo producto principal es una aplicación móvil desde la que gestiona una red de lavanderías, otra de gimnasios y otra de centros de belleza.

Jeff tiene una cultura empresarial que cuida su propio producto, por lo que es afín a mis motivaciones mencionadas anteriormente sobre el desarrollo con tecnologías punteras, así como en mejorar la calidad de su software a través de la inspección y la integración continuas.

Entre el que ha sido mi tutor de prácticas y yo, planteamos un proyecto cuyos objetivos se detallan a continuación.

El proyecto consistirá en desarrollar una herramienta cuyo objetivo principal es que permita conocer el estado de la calidad del software en proyectos que utilicen las herramientas SonarQube, Jenkins[9] y GitHub[10]. Estos proyectos deben estar desarrollados de acuerdo con el modelo de Maven.

[8] https://wearejeff.com/es/
[9] https://jenkins.io/
[10] https://github.com/

La empresa se propone objetivos trimestrales establecidos de manera parametrizable. Uno de los propuestos es el de mejorar sus productos, y para mejorar la calidad del software, necesitan una herramienta que les permita tener una instantánea de la situación actual, así como conocer la evolución de la misma. Esto permite cuantificar variables cualitativas y generar objetivos cuatrimestrales de mejora, así como obtener valores objetivos sobre su progresión.

Para tal fin se quiere desarrollar un producto que, de manera autónoma tome medidas establecidas, las procese y almacene en una base de datos.

Los objetivos concretos de la herramienta son los siguientes:

1. Tomar medidas de:
    1.1. Los análisis estáticos de código de SonarQube.
        1.1.1. Las variables a cuantificar de cada proyecto serán las siguientes:
            1.1.1.1. Número de *Bugs* en el código.
            1.1.1.2. Número de Vulnerabilidades detectadas.
            1.1.1.3. Número de *code smells*.
            1.1.1.4. Número de líneas duplicadas en el código.
            1.1.1.5. Porcentaje del código cubierto por los tests.
            1.1.1.6. Número total de líneas de las que consta el proyecto.
            1.1.1.7. Fecha en la que se obtienen estos resultados.
    1.2. Las tareas de Jenkins.
        1.2.1. Las variables a cuantificar se obtienen del resultado del *pipeline* que automatiza el despliegue del proyecto en el entorno de producción, las cuales son:
            1.2.1.1. Resultado de la ejecución del proceso.
            1.2.1.2. Fecha de ejecución del proceso.
            1.2.1.3. Proyecto sobre el que se aplicó el proceso.
    1.3. Los repositorios de GitHub.
        1.3.1. Las variables se derivan del análisis del archivo `pom.xml`, de donde se recogerá lo siguiente:
            1.3.1.1. Versión de Java del proyecto.
            1.3.1.2. Versión de Spring Boot[11].
            1.3.1.3. Versión de Spring Cloud[12].
2. Estas medidas recibirán el oportuno procesamiento a fin de que sean coherentes con el modelo de datos. Este procesamiento consistirá en alguno de los siguientes procesos.

[11] https://spring.io/projects/spring-boot
[12] https://spring.io/projects/spring-cloud

   2.1. Convertir el dato al tipo oportuno.
   2.2. Evitar el almacenamiento de duplicados.
3. Estas medidas se persistirán en una base de datos, para mantener un histórico.
4. El funcionamiento de la aplicación debe ser autónomo, por tanto:
   4.1. Se programará que una vez al día ejecute los procesos de obtención, procesamiento y persistencia de métricas.
   4.2. Para ello la aplicación debe estar desplegada en un contenedor de AWS[13], y estará siempre en ejecución, con acceso a los servicios necesarios y cada día ejecutará los procesos responsables de sus funcionalidades.
5. La aplicación debe permitir la inserción de mediciones de nuevas variables del repositorio por parte de programadores no familiarizados con el proyecto.
   5.1. Esto debe de hacerse sin alterar el modelo de datos.
   5.2. Sin tener que comprender los procesos internos de la aplicación.
6. La aplicación debe mostrar la información de manera clara y específica. Además, esta visualización podrá ser alterada por el usuario final.

1.3. Impacto esperado

La adecuada visualización de la calidad de los proyectos permitirá tomar decisiones sobre si destinar esfuerzos a la mejora de determinados proyectos o no. Más adelante se ofrecen ejemplos concretos de las decisiones que se pueden tomar sobre los datos obtenidos por la aplicación.

Gracias a tener una visión global del estado del *backend*, se pueden tomar decisiones si algún microservicio se encuentra en peor estado que otro, y si se debe priorizar su actualización sobre otras tareas, o bien que otros microservicios adquieran sus funcionalidades para que este desaparezca, por la deuda tecnológica contraída.

El conocimiento sobre la tarea de Jenkins que se mide permite conocer la cantidad de despliegues que se hacen sobre el entorno de producción cada día.

Esto permite hacer inferencias sobre:

1. Si la gente confía en sus habilidades para subir a producción cambios.
2. Si hay muchos despliegues, puede ser un indicador de que la empresa responde con flexibilidad a las demandas de los clientes y otros departamentos.

[13] https://aws.amazon.com/es/

Por lo que respecta a GitHub, las medidas sobre los repositorios permiten tener un control de las versiones sobre las que trabajan los microservicios.

Añadiéndole la información sobre los despliegues, se podría determinar si se debe subir de versión un proyecto, dado que se encuentra en riesgo de quedarse desfasado y está recibiendo muchas actualizaciones.

En la situación inversa, es decir, cuando el proyecto recibe pocos cambios y no se usa demasiado, las medidas obtenidas apoyarían la decisión de no invertir particular esfuerzo en mantenerlo y actualizar sus versiones.

Básicamente, mi aplicación facilitará datos objetivos en los que basarse para tomar decisiones sobre acciones que afectarán a todo el departamento de *backend*, y por extensión al producto final.

Resaltar que, como impacto secundario, conocer las métricas de SonarQube permite conocer cuánto esfuerzo se requiere para arreglar los defectos señalados por este.

Por lo que, al arreglarse, se tendrían aplicaciones más robustas, que afectarían positivamente al producto.

El cumplir con las reglas de SonarQube genera un código legible y fácil de mantener, favoreciendo el crecimiento de la empresa al agilizar las tareas de los programadores.

1.4. Metodología

Para el desarrollo del proyecto de utilizó el desarrollo o modelo en espiral (Bohem, 1988), es decir, se plantearon una serie de iteraciones, teniendo la primera de ellas ya funcionalidad, de manera que las siguientes mejorarían esta característica o añadirían una nueva.

Esta aproximación, al ser menos rígida que el modelo en cascada (Royce, 1970), permite variar la estrategia a partir de la cual se van a alcanzar los objetivos, o poder extender el alcance de alguno de los requisitos planteados, así como plantear algunos nuevos.

Este modelo se adaptó a nuestro caso estableciendo que cada iteración se compondría de las siguientes fases:

1. Propuesta y discusión: donde se establece la funcionalidad que se va a desarrollar, con sus requisitos, y se plantean una o dos aproximaciones posibles.
2. Investigación: donde se revisaría la documentación acerca de cómo se implementa la característica planteada en la fase anterior y se decide qué aproximación se prefiere. Es posible que esta fase plantee alternativas no consideradas en un principio, pero en caso de aparecer se tendrán en cuenta para la decisión.
3. Implementación: en esta fase se escribe el código fuente de la aproximación elegida.
4. Pruebas: se somete al código a diferentes tipos de pruebas para comprobar su correcto funcionamiento.

La elección de este modelo se apoya en dos motivos, el primero de ellos es que la posibilidad de que la fase de investigación descubriera nuevas características susceptibles de generar una nueva iteración.

La situación al principio era que, solo teníamos conocimiento de las funcionalidades más básicas de las APIs de SonarQube y Jenkins, así como de la librería de JGit[14], por lo que sabíamos que los requisitos más sencillos se podían satisfacer, pero era posible que un análisis más pormenorizado de estas tecnologías nos abriera las puertas a desarrollar nuevas funcionalidades.

El segundo motivo es que, al ser este proyecto parte de unas prácticas, ser una herramienta de uso interno y encontrarme en contacto directo con el tutor, podían surgir nuevas características a implementar o, por necesidades de la empresa, profundizar en alguna de las iteraciones y el modelo en espiral permite esta flexibilidad.

1.5. Estructura

Seguidamente se van a describir los apartados que componen este documento y su contenido.

En Contexto tecnológico se va a explicar el modelo de negocio y el funcionamiento de Jeff, para seguidamente comparar el modelo centralizado de arquitectura monolítica con la arquitectura distribuida basada en microservicios.

[14] https://www.eclipse.org/jgit/

En la crítica se comentarán diferencias y semejanzas con otros TFGs publicados en la ETSInf y se finalizará el apartado con la propuesta planteada.

En Análisis del problema se va a hablar del que será el usuario de la aplicación, así como de demandas concretas que determinarán las características de la solución y se hablará del esquema general de la solución planteada. El siguiente subapartado profundizará en el comportamiento de la aplicación, así como la mención de sus componentes principales y el apartado terminará con un plan de trabajo donde se muestra un desglose de tareas y su planificación.

Seguidamente, en el apartado de Diseño de la solución se definirá el modelo de datos, para posteriormente, en Arquitectura del sistema comentar el comportamiento general de los procesos de la aplicación y en Diseño detallado, las particularidades de cada uno de ellos, así como la explicación de los paneles de visualización. Terminará este apartado definiendo las tecnologías utilizadas para el desarrollo del proyecto.

En el apartado de Desarrollo de la solución propuesta se van a mencionar algunos aspectos destacables que sucedieron durante esta fase, ya que el desarrollo no se distanció mucho de la planificación.

Por lo que respecta al apartado Implantación, se va a describir el entorno en el que se desplegó la aplicación, así como las características que tenían cada uno de los componentes de este proceso.

El apartado Pruebas expondrá las diferentes validaciones llevadas a cabo sobre el proyecto, así como los análisis de SonarQube sobre el mismo y comentará las conclusiones extraídas de estos datos.

Posteriormente se encuentran las conclusiones, así como la relación entre lo aprendido durante el grado y las tecnologías empleadas en este proyecto.

Finalmente, se va a destacar un posible proyecto futuro a partir del desarrollo expuesto.

1.6. Convenciones

- Los nombres de ficheros relevantes para las tecnologías se mostrarán en letra `Courier` para resaltar. Como por ejemplo el `pom.xml`.

- Las palabas extranjeras se destacarán en letra cursiva, señalar que los nombres de las tablas de la base de datos, por haberse denominado en inglés, también se destacarán en letra cursiva. Por ejemplo: *backend*.

- Fragmentos de código en el texto se resaltarán con el tipo de letra Abadi Extra Light, por ejemplo: project.get().

- Los nombres de los formatos de los archivos se mostrarán en mayúsculas, asumiendo que son acrónimos del inglés. Ejemplos: XML, JSON, JAR, …

2. Contexto tecnológico

La empresa Jeff se fundó en 2015. Es una startup o empresa emergente que ha cambiado su estrategia comercial en varias ocasiones a partir de las rondas de financiación recibidas. Empezó gestionando lavanderías, para posteriormente convertir ese modelo de negocio en un franquiciado, lo que les permitió llegar a más países. En estos momentos Jeff tiene franquicias en más de 20 países.

El modelo de negocio actual es la venta de franquiciados de lavanderías, centros de belleza y gimnasios. Cuando una persona quiere abrir una franquicia de Jeff, primero decide en cuáles tiene interés y Jeff aporta infraestructura informática para la gestión del negocio, así como asesoramiento y facilidades para el buen desarrollo de ese punto de venta.

Lo que tecnológicamente ofrece Jeff a sus franquiciados es lo que se conoce como *Software as a service*, es decir, se ofrece acceso a los servicios que ofrecen sus aplicaciones, por una parte, a las aplicaciones web y móviles para los clientes desde donde el usuario realiza pedidos y por otro lado, herramientas para la gestión del negocio que permiten visualizar el estado de los pedidos, así como llevar a cabo la facturación y otras funcionalidades que requieren los franquiciados.

El departamento de producto e ingeniería de Jeff, donde realicé las prácticas y este proyecto, tiene la estructura mostrada en la Figura 1. Existen dos responsables de todo el departamento; inmediatamente tras ellos existe un responsable del área de ingeniería, encargado de la gestión de aspectos puramente tecnológicos, otro responsable del diseño de producto, que gestiona a los diseñadores del departamento, un responsable de cliente de producto, quien es responsable de aquellos que se encargan del contacto con el cliente y exploran posibles mejoras de los servicios ofertados. Por último, existe un responsable del departamento de datos, donde se realizan los análisis estadísticos pertinentes a demanda de otras partes de la empresa.

Bajo ellos se encuentran los diferentes equipos categorizados por funciones, donde por cada vertical, es decir, por cada tipo de franquicia (lavandería, centro de belleza, gimnasio y recientemente centro de relajación) existen dos equipos, uno que gestiona los servicios que reciben los clientes y otro que gestiona la aplicación que se le proporciona a los franquiciados. El equipo de *backoffice* amplía y mantiene la aplicación que permite la gestión interna de la empresa y el

equipo de logística se ocupa de desarrollos relacionados con el estado de los pedidos.

Finalmente, los equipos transversales tienen impacto en la actividad del resto de equipos, por eso se crea un equipo específico para cada uno de ellos. Ejemplos de estas funcionalidades son los sistemas de pago, el servicio de atención al cliente (*tech support eng.*) o el diseño de componentes web.

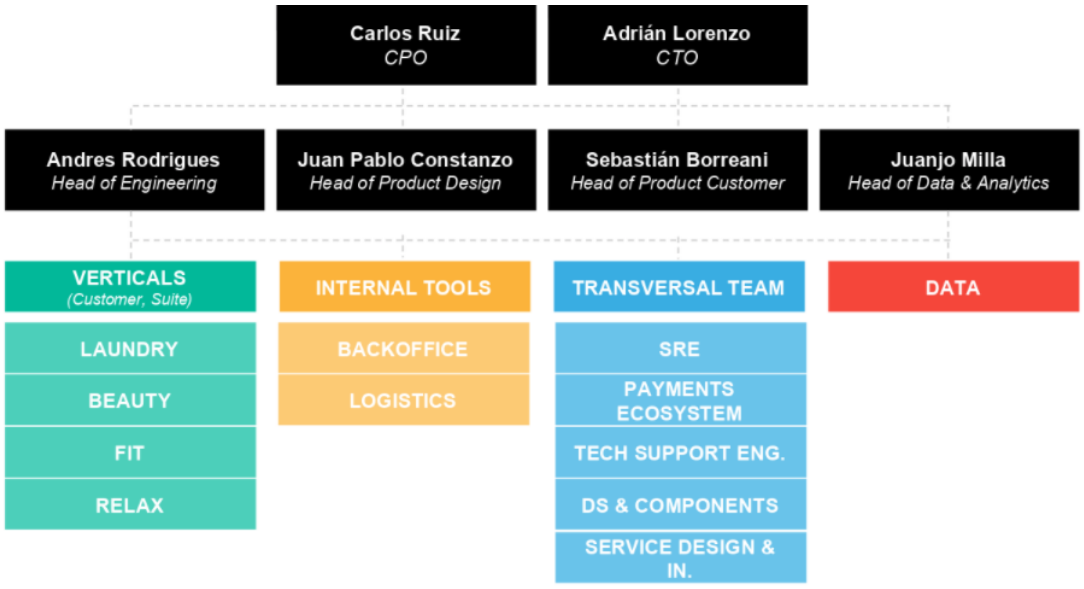

Figura 1. Estructura del departamento

Dentro de cada equipo existen cinco perfiles profesionales, desarrollador *backend*, desarrollador *frontend*, *product manager*, desarrollador móvil y diseñador. Sobre las demandas de cada tarea, su número varía. La consecuencia de esto es que existen departamentos horizontales sobre cada uno de estos roles que se reúnen periódicamente para poner en común posibilidades de mejora o posibles fallos encontrados. Mis funciones durante las prácticas estaban relacionadas con el apoyo al responsable del departamento horizontal del *backend*.

Jeff hace gala de una cultura empresarial basada en la transparencia, la comunicación y el apoyo mutuo. La Figura 2 muestra lo que ellos llaman “los cuatro pilares de Jeff” y aparece frecuentemente en las charlas y presentaciones que la compañía lleva a cabo internamente.

Figura 2. Pilares de la cultura de Jeff.

Quiero destacar de la Figura 2 el pilar de *never enough*, del lema de nunca es suficiente surgen muchas ramificaciones siendo una de ellas este proyecto. Se interpreta como que siempre es posible mejorar; esa motivación desemboca en cuidar el producto propio, es decir, que sus aplicaciones sean lo más atractivas visualmente posible, que sea lo más funcionales posibles, que la experiencia de usuario sea lo más agradable posible, etc. De manera que cuando esto se consigue, sigue pudiéndose mejorar.

Por tanto, parte de los esfuerzos del departamento del *backend* se dirigen hacia la mejora de la calidad de los servicios. Como Jeff utiliza el modelo de OKRs, es decir, para su progresión y mejora constante plantea objetivos trimestrales cuantificables que se comparten con toda la empresa, tanto en su presentación como en su evolución.

Entre estos objetivos planteados figura la mejora de la calidad del código de las aplicaciones, aspecto sobre el que va a intervenir directamente este proyecto, ya que al ofrecer una solución a la visualización del estado del backend, esta tendrá la característica de ser medible.

Proseguiremos contrastando dos modelos de desarrollo de aplicaciones en red que se han utilizado en Jeff. Para ello se añade la Figura 3 a modo de orientación visual de lo que se va a desarrollar en los siguientes párrafos.

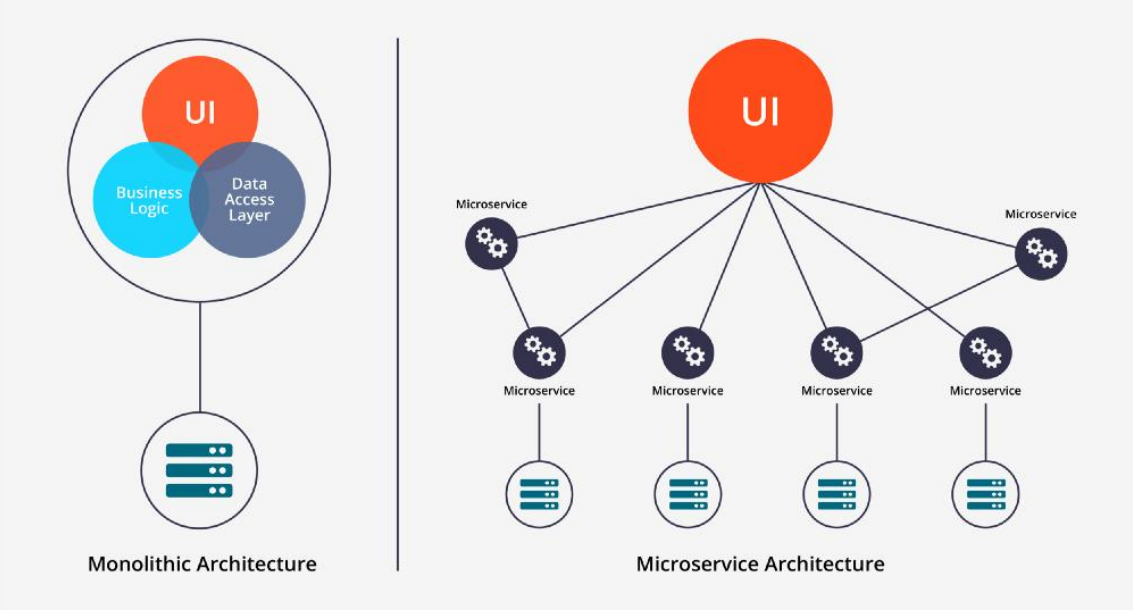

Figura 3. Arquitectura monolítica y arquitectura de microservicios[15]

Tradicionalmente los desarrollos de las aplicaciones en red se desarrollaban de acuerdo con una arquitectura monolítica o centralizada, esto es que la aplicación se diseñaba para que estuviera autocontenida en un mismo servidor, lo que quiere decir que este procesa todas las peticiones que le llegasen de manera íntegra, haciendo el uso de servicios externos cuando es estrictamente necesario.

De acuerdo con esta filosofía, la aplicación de uno de estos monolitos debe de contener como mínimo una capa de aplicación que expone puntos de acceso donde recibir peticiones, una capa de servicio que soportaría la capa de negocio y una capa de acceso a datos, que permita la conexión con la base de datos, también alojada en el servidor.

Esta arquitectura presenta una serie de ventajas. La primera es la seguridad; al exponer unos pocos puntos a internet, solo se debe cuidar la seguridad en estos lugares.

Otra ventaja es que son aplicaciones muy eficientes. Al ser autocontenidas, todo el procesamiento sucede internamente en el servidor y se encuentra bajo el control del lenguaje de programación, y no es necesario que el programador lo gestione manualmente.

Esta característica gana relevancia al compararla con la comunicación entre microservicios, la cual, adelanto, sucede a través de la red, añadiendo un retardo,

[15] Tomado de https://medium.com/@goodrebels/microservicios-ventajas-y-contras-de-la-arquitectura-descentralizada-a3b7fc814422

que, aunque sea normalmente breve, suele ser variable y siempre más alto que si esa comunicación se hiciera entre componentes de la misma aplicación.

Uno de los motivos por el que se adopta esta arquitectura, es que resulta como muy lógico empezar por una funcionalidad concreta y que conforme los usuarios vayan haciendo uso de la aplicación, hagan sugerencias de mejora y estas se implementen. Este proceso mantenido a lo largo de cuatro años da como resultado una arquitectura monolítica.

Sin embargo, cuenta con numerosas desventajas, entre ellas que su excesivo acoplamiento conlleva una elevada rigidez. No es difícil añadirle funcionalidades nuevas, pero sí modificar las que ya están funcionando, por lo que se corre el riesgo de acabar teniendo un proyecto con mucho código que no se está utilizando. Además de que actualizar las versiones de los diferentes *frameworks* o librerías puede resultar una ardua tarea, ya que hay demasiados componentes que dependen de ellas.

Otras desventajas se derivan del hecho de que la aplicación se ubique en la misma máquina. Aunque esta sea robusta y resistente, en caso de que falle se cae todo el servicio irremediablemente. También encuentra problemas a la hora de escalar el servicio, al ser un único dispositivo, la única manera de aumentar sus prestaciones es comprar un servidor más potente, además de no permitir la gestión de los cuellos de botella que internamente puedan suceder.

Se puede dar el caso de que el mismo servidor contenga varias aplicaciones, pero estas aplicaciones siguen diseñadas con la misma filosofía de contener los componentes necesarios para gestionar sus peticiones.

En contraposición a la arquitectura monolítica se encuentran las arquitecturas basadas en microservicios, definidas por Martin Fowler como "…un enfoque para desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno de los cuales se ejecuta en su propio proceso y se comunica con mecanismos ligeros, a menudo una API de recursos HTTP." (Fowler M. , 2014).

La idea básica es dividir los componentes necesarios para resolver determinada petición en aplicaciones separadas y conectadas entre ellas. Estos microservicios deben ser específicos en la funcionalidad que realizan y evitar realizar más de una acción. Por ejemplo, un microservicio que haga la función de *Gateway* solo redireccionará peticiones y no procesará más allá de lo necesario para saber a quién la debe transmitir.

Esta arquitectura pretende superar las desventajas de los monolitos. Por tanto, estas son algunas de sus ventajas.

Al ser componentes pequeños es fácil escalarlos para aumentar las prestaciones del servicio, y gracias a tecnologías como Docker[16] y Kubernetes[17], se puede automatizar tanto el generar nuevas réplicas del microservicio, como su apagado en función de la carga que está recibiendo. Es decir, se aprovecha la elasticidad de los microservicios para resolver el problema de los cuellos de botella.

Al no encontrarse fuertemente acoplados los microservicios pueden tolerar fallos. Si algún componente cae, no solo no afecta al resto del proceso, sino que se puede levantar otra réplica, dado que esta gestión es automatizable, y el usuario final no tiene por qué percibir un cese del servicio.

El añadido de funcionalidades al producto es sencillo, dado que para desarrollar un nuevo microservicio solo es necesario conocer el protocolo de comunicaciones entre ellos, para saber qué se va a recibir y qué se va a devolver, pero no es necesario conocer la estructura global. De la misma manera, alterar una funcionalidad es sencillo, pues si se respeta el formato de los mensajes, se puede alterar la lógica de un microservicio sabiendo que otros no dejarán de funcionar.

Por otra parte, esta arquitectura no está libre de desventajas. La primera es que la gestión de las comunicaciones puede resultar muy compleja, ya que el buen funcionamiento de los microservicios depende de ella. Se requiere el trabajo adicional de desarrollar un protocolo de comunicaciones entre microservicios que permita la escalabilidad del sistema. Una posible solución sería un sistema de colas que permita la comunicación asíncrona, pero es algo que no requería el servidor monolítico.

Otro problema derivado de la gestión de comunicaciones es que la respuesta puede ser demasiado lenta, por requerir la participación de demasiados microservicios en ella. El problema del retardo se agravaría en caso de que alguno de estos microservicios falle. Sin embargo, existen aportaciones, como el proyecto Hystrix de Netflix (Netflix, 2018) que da solución a estos problemas, dotando de mayor resiliencia a la arquitectura de microservicios.

Otro inconveniente es la visualización del estado global de la red de los microservicios, pese a la complejidad del monolito, es posible hacer que registre su actividad en logs que, aunque sea un fichero muy grande, todo lo que ha sucedido está ahí. En cambio, en el ecosistema de los microservicios, cada componente tendría un fichero de logs, lo que dificulta la recopilación de información.

[16] https://www.docker.com/
[17] https://www.kubernetes.io/

Este problema se extiende en la mayoría de las gestiones que pretenden afectar a todo el sistema. Por ejemplo, si bien es cierto que subir la versión de Java de un microservicio es una tarea mucho menor que hacerlo en un monolito y con mucho menos riesgo de fallo, subirlo en 80 microservicios no es cómodo.

Cuando Jeff llevó a cabo su internacionalización, el departamento del *backend* llevó a cabo la tarea de fragmentar el monolito con el que gestionaban el servicio, en microservicios para aprovecharse de las ventajas mencionadas anteriormente.

Pero se encontraron con los siguientes problemas en la visualización. El primero alude a SonarQube, que recoge las métricas de calidad del código de alrededor de 100 microservicios y ofrece la interfaz que se muestra en la Figura 4.

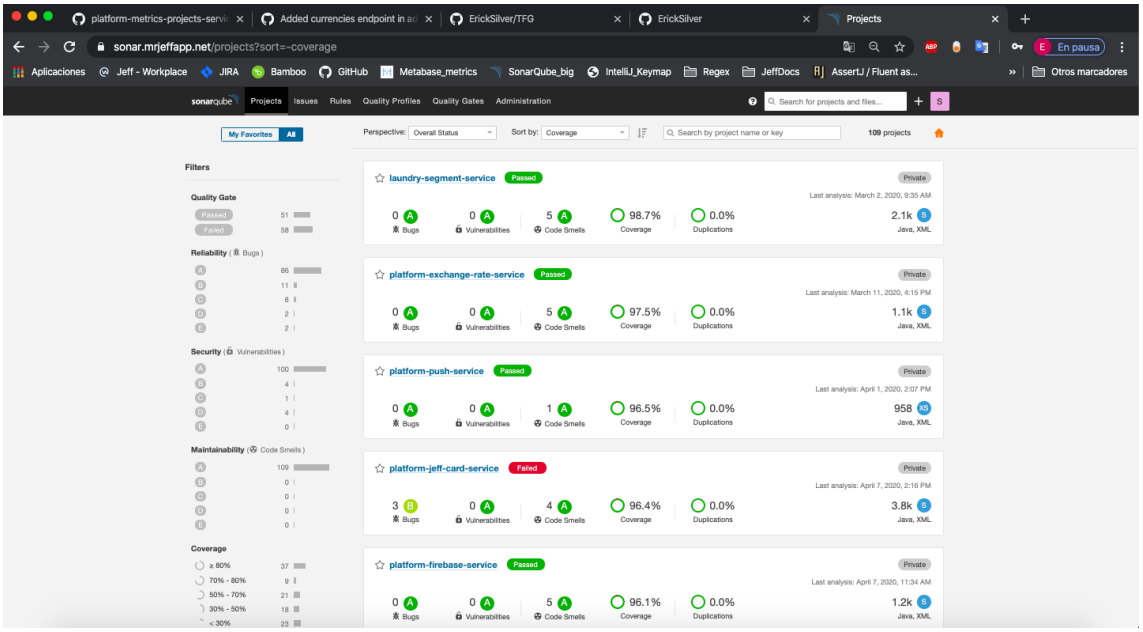

Figura 4: Interfaz de SonarQube

Esta lista tiene una fila por cada proyecto, por lo que revisar el estado de cada uno de ellos supone una tarea pesada. Además, sus filtros son muy limitados; permiten poco más que ordenar por determinada métrica o buscar un proyecto concreto. Se echan en falta funciones como recoger aquellos proyectos que tengan una métrica por encima de determinado valor y, sobre todo, el tener un histórico de análisis de cada proyecto, pues este panel no permite saber si los valores han mejorado o empeorado.

Es cierto que una vez dentro del proyecto, la información que ofrece está mucho más detallada y ofrece una distinción entre métricas tomadas en el código nuevo y las del código ya analizado, pero esto no satisface las necesidades de la empresa.

Por otra parte, tenemos a Jenkins, la herramienta que se utiliza para la integración continua. La Figura 5 muestra la interfaz que se ofrece del proceso que lleva a cabo esta función.

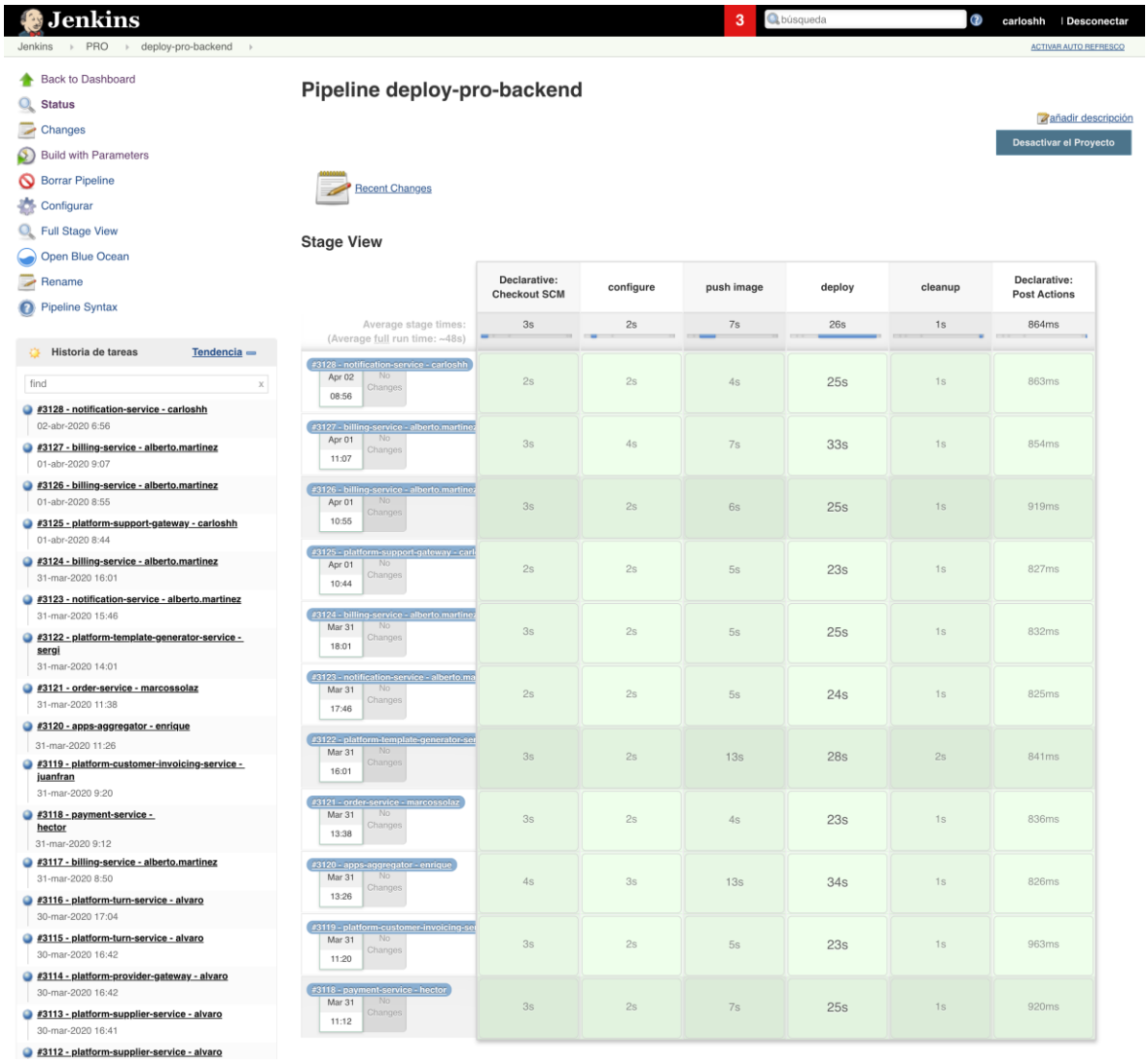

Figura 5: Interfaz de Jenkins

Es cierto que aquí disponemos de un historial de ejecuciones de la tarea, pero resulta una lista extensa y con escasas funciones de filtrado.

La conclusión es que Jenkins, al igual que SonarQube, ofrecen mucha información sobre lo concreto, como puede ser los pasos de la tarea o en análisis de un proyecto concreto, pero poco sobre lo global, como sería un histórico detallado de la evolución de determinadas métricas o un historial de despliegues sobre un proyecto determinado.

Este problema se repite en los repositorios de GitHub, solo que, con aún menos información, lo que resulta lógico, ya que el objetivo de GitHub es almacenar código fuente sin analizarlo.

Se aprecia que conforme la arquitectura basada en microservicios fue creciendo en complejidad los problemas de visualización empezaron a aparecer hasta llegar a la situación actual, en la que cuando el departamento del backend tiene que tomar medidas de eficiencia tecnológica para plantear mejoras de cara a los siguientes OKRs, encontraba que había que dedicar demasiado tiempo a tareas muy repetitivas.

Es en este contexto donde surge la idea de centralizar una serie de medidas que permitan visualizar el estado del *backend*.

2.1. Crítica al estado del arte

Se van a revisar algunos trabajos anteriormente presentados relacionados con las tecnologías y la problemática que se aborda.

Tras revisar algunos trabajos de fin de grado presentados en la ETSInf que utilizan SonarQube, se aprecia que su uso se ciñe a una comprobación de la calidad del software. Este proyecto pretende tratar esta herramienta desde la perspectiva de una empresa y aprovechar otras funcionalidades de esta. Son ejemplo de este uso el trabajo de Morant Navarro, J. (Morant Navarro, 2016) y el de Taberner Aguas, B. (Taberner Aguas, 2015).

También se encuentran trabajos que implementan la integración continua, utilizando la herramienta Jenkins entre otras. Un ejemplo de ello es el proyecto de Borja , P. (Hernández, 2017) donde se muestra la implementación del proceso de automatización del despliegue y el de Todea, V.C. (Todea, 2016), que describe el proceso de integración continua y muestra su implementación a través de Jenkins, en un entorno parecido al de Jeff, ya que para el proceso utiliza contenedores Docker. Mi TFG pretende dar un paso más y facilitar la visualización de los resultados de estos procesos en un entorno donde ya se despliega de acuerdo con este paradigma.

El trabajo de Tortosa (Tortosa Calabuig, 2019) y el de Pérez (Pérez Andrés, 2019) son ejemplos de desarrollos software empleando la arquitectura de microservicios. Ambos trabajos ofrecen soluciones para la primera implementación de un servicio, por lo que las desventajas mencionadas anteriormente aún no han aparecido. El proyecto en el que se basa este TFG se ubica en un contexto de microservicios mucho más desarrollado, donde sus desventajas pueden acabar siendo limitantes para el buen funcionamiento del desarrollo.

2.2. Propuesta

La situación actual es que cuando un programador hace alguna modificación en uno de los múltiples proyectos de Jeff y quiere llevarlo al entorno de producción, subirá los cambios al repositorio de GitHub correspondiente e iniciará una tarea de Jenkins, quien le enviará el proyecto a SonarQube para que lo analice y luego empezará el proceso de integración continua, cuyo resultado es que se despliegue en producción y los cambios sean visibles para los usuarios finales.

Lo que quiere decir, que cuando se quiere saber el estado de un proyecto concreto, es necesario buscarlo en la interfaz de SonarQube, luego revisar las ejecuciones de la tarea de Jenkins y, por último, entrar a GitHub, y comprobar los últimos *commits*.

La propuesta es que mi aplicación recoja los aspectos relevantes de estas aplicaciones y que, a través de Metabase[18], ofrezca una visualización que permita conocer el estado global del *backend*, así como el estado concreto de los proyectos.

Esto permitirá reducir el tiempo destinado a tareas repetitivas y destinar ese tiempo a buscar soluciones a situaciones inesperadas o generar estrategias de mejora de la calidad del código, así como poder comprobar en poco tiempo si las estrategias o las soluciones están siendo efectivas. Destacar que cuantificar estas variables cualitativas va a permitir generar informes numéricos sobre la salud de los desarrollos del departamento de *backend*, permitiendo visibilizar un tipo de trabajo que es difícil de percibir por profesionales no expertos en la materia. Por último, se va a distinguir entre los aspectos que venían determinados por la empresa de los que fueron aportación mía.

Jeff determinó que el proyecto debería recopilar los datos concretos, mencionados en los objetivos, de las tres fuentes mencionadas, que el proyecto debía realizarse en Spring, que debía validarse el comportamiento de la aplicación haciendo uso de test, la estructura de carpetas del proyecto y que debe de poder desplegarse en un contenedor Docker. La finalidad se debe a la funcionalidad exigida y a mantener homogeneidad entre las aplicaciones del *backend*.

Bajo mi criterio quedó la implementación de estas características. Esto quiere decir que el diseño general, las tecnologías de Spring que se iban a emplear, así

[18] https://www.metabase.com/

como las características de las clases Java de la aplicación han sido aspectos que he aportado yo.

3. Análisis del problema

Habiendo explicado el contexto en el que se encuentra este proyecto, se va a desarrollar un análisis más pormenorizado de los objetivos, y de cómo se van a conseguir.

Se pretende resolver un problema de visibilidad, dado que, como se ha comentado con anterioridad, la información relevante se encuentra dispersa en diferentes aplicaciones.

La demanda de una solución venía por parte del responsable del *backend*, quien era mi tutor de prácticas. Esta persona tiene una extensa experiencia como programador y utiliza Metabase, una herramienta de inteligencia de negocio, para la obtención de informes sobre los datos. Por tanto, la información recogida debe de estar preparada para este contexto. Aunque algunos de los gráficos que se vayan a desarrollar durante este proyecto, deben de poder ser sometidos a cambios o generar nuevos a demanda de este responsable.

Un requisito por parte de mi tutor fue que la base de datos estuviera en MySQL[19], por cumplir con el estándar de Jeff y no romper la homogeneidad de sus bases de datos.

Por tanto, mi aplicación va a ser un hilo conductor entre Metabase y las diferentes aplicaciones e integrará estas plataformas para cumplir con las demandas de la empresa, tal y como se muestra en la Figura 6. La información será recogida de las APIs de SonarQube y Jenkins, por un lado, y por otro desde la librería JGit se clonarán los repositorios de GitHub a un directorio temporal y se analizarán. Esto será almacenado en la base de datos, donde Metabase las consultará cuando el usuario lo solicite.

He de comentar que el nombre de mi aplicación, *platform-metrics-projects-service*, cumple con el patrón de denominado de proyectos de Jeff. De esta manera el sufijo *service* marca qué tipo de proyecto es, para distinguirlo de un *gateway*, por ejemplo. El prefijo *platform* pretende dar a entender que es un proyecto independiente, que no forma parte de divisiones como *laundry* o *beauty*. Por último, *metrics-projects*, resume la funcionalidad del proyecto, que es recoger métricas de los proyectos.

[19] https://www.mysql.com/

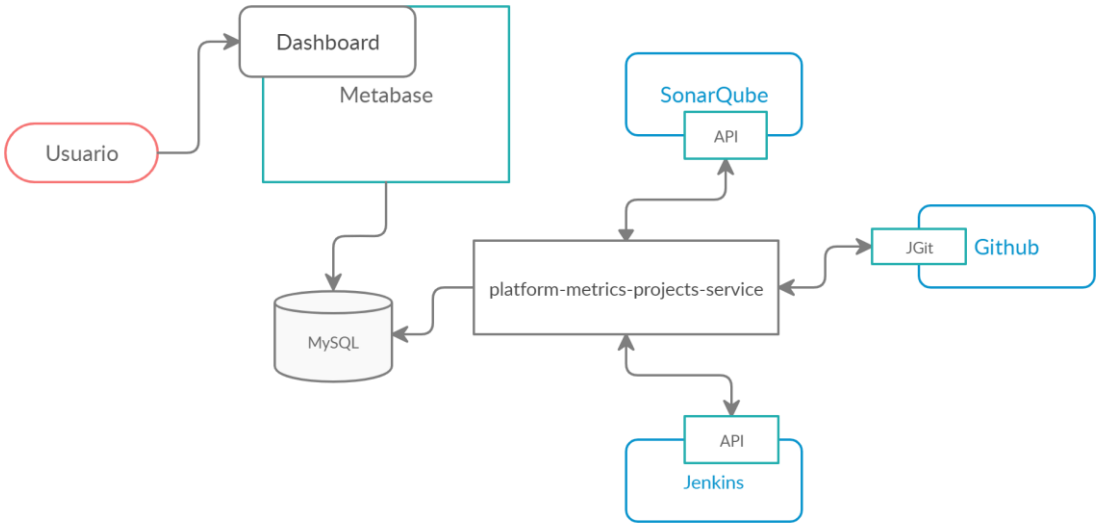

Figura 6. Esquema general de la aplicación platform-metrics-projects-service

Cabe destacar que estas cajas son componentes separados y se encontrarán en contenedores separados. De esa manera solo el fallo de la base de datos originará una caída del servicio, ya que, si cae Metabase, la aplicación seguirá recogiendo métricas, el fallo en alguno de los procesos de recogida de datos no afecta a los otros y finalmente, si falla la aplicación, los datos seguirán siendo visibles desde Metabase.

Las medidas para evitar el fallo en la base de datos están a cargo del departamento de DevOps de Jeff y se detallarán en la fase de implantación.

3.1. Solución propuesta

Recopilando lo mencionado anteriormente, la aplicación *platform-metrics-projects-service* tendrá las siguientes características.

La aplicación se desplegará desde un docker-compose, que lanzará un contenedor con Metabase, otro con un servidor MySQL y otro contenedor con la aplicación, de manera que se encuentren interconectados. Quiero señalar que esta base de datos la requiere Metabase, ya que la base de datos de la aplicación a la que también se conectará este servicio corresponderá a una propia de la organización

y sus tablas se construirán en la primera ejecución de la aplicación utilizando las migraciones de Flyway[20].

Con respecto a la base de datos, su esquema se va a simplificar lo máximo posible, manteniendo las claves ajenas mínimas y evitando que tenga comportamiento, sencillamente se utilizará para acumular datos en las seis tablas que la componen. La finalidad es que, si en un futuro la empresa decide utilizar otra tecnología para las bases de datos, la migración resulte sencilla.

El proyecto estará construido a través de Maven, para poder hacer uso de sus librerías a través del `pom.xml`, para utilizar el ciclo de vida de Maven a la hora de compilar, ejecutar test y generar ejecutables y para utilizar su estructura de carpetas. Además, Spring está muy integrado con los proyectos Maven y la mayoría de sus librerías se encuentran en los repositorios oficiales que ofrece esta tecnología.

La aplicación se conectará a la API de SonarQube y extraerá las métricas de *bugs*, vulnerabilidades, *code smells*, cobertura por los test, número de líneas de código duplicadas, número de líneas de código totales y fecha del último análisis de cada uno de los proyectos. También se conectará a la API de Jenkins para recoger los detalles de la ejecución de la tarea *deploy-pro*, y de donde recogerá el nombre del proyecto desplegado, el resultado de la ejecución y la fecha de ejecución.

Las conexiones mencionadas se harán a través de un cliente OpenFeign[21], al que se le añadirá un control de errores utilizando la librería de Spring Retry, que implementa un número de reintentos ante el fallo de la petición y la ejecución de determinado método una vez esos intentos se agoten.

Por lo que respeta a los repositorios, se descargarán a través de JGit, librería que aporta implementaciones en Java para ejecutar los comandos de git, por lo que se clonará el repositorio a partir de su URL, se analizará el `pom.xml` y se extraerán las versiones de Java, Spring Boot y Spring Cloud.

Se va a utilizar el patrón de diseño estructural de las factorías (Eric Freeman, 2004) para convertir lo que se recibe del exterior de la aplicación a objetos que se almacenarán en la base de datos. Este patrón de diseño consiste en llevar la gestión de la creación de objetos a un tipo de servicios nombrado como factorías. Su finalidad es aislar la instanciación de un objeto a un lugar concreto de la aplicación, para reducir la lógica de los servicios.

[20] https://flywaydb.org/
[21] https://spring.io/projects/spring-cloud-openfeign

Finalmente, y para cumplir el quinto objetivo (permitir la inserción de nuevas mediciones sobre los repositorios por parte de programadores ajenos al proyecto), se va a utilizar el patrón de diseño estratégico. Se creará una interfaz con un método, que se invocará en la capa de la factoría, de manera que todas las implementaciones de esa interfaz ejecuten su trozo de código. Así, un programador ajeno solo tiene que implementar esta interfaz para escribir código ejecutable dentro de la aplicación. La ventaja por la que se va a utilizar este patrón es porque permite introducir nuevos análisis sobre los repositorios o eliminar algunos ya creados sin alterar el funcionamiento del resto de la aplicación (Guru, 2020).

Estos procesos se automatizarán para que se lleven a cabo todos los días a las 6 salvo los fines de semana. Sin embargo, para tener cierto control sobre el proceso, se crearán *endpoints* que permitan lanzar cada uno de los tres procesos de recogida de datos de manera manual, para que en caso de fallo o de querer renovar los datos, no haya que esperarse a que el proceso se ejecute por sí solo.

Estos puntos de acceso se crearán haciendo uso de los RestController de Spring y estarán protegidos con el protocolo de autenticación de acceso básica de HTTP, y devolverán en formato JSON los datos obtenidos del proceso después de almacenarlos en la base de datos.

Finalmente, se gestionará Metabase, donde habrá una serie de paneles donde se visualice el contenido de la base de datos en forma de gráficos con el contenido de los resultados de determinadas consultas SQL.

Se pretende que, partiendo de un elemento mínimo con funcionalidad al que se le irán añadiendo elementos, el proyecto atraviese las siguientes fases de desarrollo:

1. Crear un proyecto con una funcionalidad mínima: Una aplicación Spring generada desde Maven que ejecute llamadas a la API Rest de SonarQube.
2. Primer *endpoint*: Permitará ejecutar el proceso anterior desde fuera de la aplicación.
3. Persistir la información: Almacenar el resultado del proceso en base de datos.
4. Duplicar el proceso: Replicar el modelo para Jenkins.
5. Añadir JGit: Incorporar al proyecto de la librería JGit para tener acceso a los repositorios y replicar el proceso.
6. Gestionar la seguridad en los puntos de acceso: Exigir credenciales en los puntos de acceso.
7. Gestión de fallos: Añadir control de fallos en las peticiones que realiza mi aplicación.

8. Imagen Docker con la aplicación: Añadir los componentes necesarios para que el proyecto se ejecute en un contenedor Docker generado por Maven.
9. Configurar Metabase: Así como sus paneles de visualización.
10. Test: Desarrollar a cabo los test unitarios y de integración necesarios.
11. Estratégico: Incorporar el patrón de diseño estratégico para los repositorios.
12. Documentación: Generar la documentación de las URL expuestas.
13. Crear un docker-compose: Este archivo debe desplegar los contenedores de mi aplicación, el de MySQL y el de Metabase y deben estar interconectados.

3.2. Plan de trabajo

Antes de abordar el proyecto se desarrolló un plan de trabajo, con el fin de ser orientativo y poder aproximar la evolución del proyecto durante su desarrollo, así como su fecha de finalización. Por lo tanto, la división de tareas que se llevó a cabo es de carácter orientativo y no pretenden definir la tarea en sí misma, sino etiquetarla.

Las tareas en las que se desglosan las trece fases mencionadas en el apartado anterior son las siguientes:

1. Crear una aplicación con funcionalidad mínima.
   1.1. Exploración de la API de SonarQube.
   1.2. Generación de un proyecto Spring.
   1.3. Crear un cliente que llame a la API.
2. Primer *endpoint*, el de SonarQube.
   2.1. Añadir un *endpoint* que ejecute el proceso del cliente de la API.
3. Persistir la información.
   3.1. Crear un contenedor con un servidor MySQL.
   3.2. Incorporar al proyecto el sistema de migraciones Flyway.
   3.3. Conectar el proyecto con el servidor MySQL.
   3.4. Generar la base de datos desde Flyway.
   3.5. Añadir la factoría de los objetos de SonarQube.
   3.6. Crear la capa de repositorios y el repositorio para los objetos de SonarQube.
   3.7. Modificar el servicio para incorporar el repositorio y el almacenado en base de datos.
4. Duplicar el proceso para Jenkins.
   4.1. Exploración de la API de Jenkins.

4.2. Creación de un cliente que llame a la API.
4.3. Añadir un *endpoint* que ejecute el proceso del cliente de la API.
4.4. Añadir una factoría.
4.5. Crear su repositorio.
4.6. Crear la tabla en el sistema de migraciones.
4.7. Modificar el servicio para incorporar el repositorio y el almacenado en base de datos.

5. Incorporar al proyecto la librería JGit.
   5.1. Crear un Servicio que ejecute los métodos de la librería.
   5.2. Gestionar el sistema de carpetas temporales de los repositorios.
   5.3. Añadir un *endpoint* que ejecute el clonado de un repositorio.
   5.4. Añadir una factoría.
   5.5. Crear su repositorio.
   5.6. Crear la tabla oportuna en el sistema de migraciones.
   5.7. Modificar el servicio para incorporar el repositorio y el almacenado en base de datos
6. Gestionar la seguridad en los puntos de acceso.
   6.1. Exigir credenciales a través de autenticación básica en las peticiones a mi aplicación.
7. Añadir el control de fallos a los clientes.
   7.1. Generar el servicio que ejecutará las llamadas.
   7.2. Desarrollar el método de reintentos.
   7.3. Desarrollar el método de recuperación.
8. Que el proyecto se ejecute desde un contenedor Docker.
   8.1. Añadir el plugin que lo permite y programar su ejecución dentro del ciclo de vida de Maven.
   8.2. Conectar ese contenedor con el contenedor de MySQL.
   8.3. Programar la ejecución automática de los procesos de petición de datos y almacenamiento del resultado de su procesamiento.
9. Configurar Metabase.
   9.1. Crear un contenedor con un servidor Metabase.
   9.2. Conectar ese contenedor de Metabase con el de MySQL.
   9.3. Generar *dashboards* y consultas contra la base de datos.
10. Generación de Test.
    10.1. Tests Unitarios de las factorías.
    10.2. Tests Unitarios de los servicios.
    10.3. Tests de Integración de los controladores Rest.
11. Patrón de diseño estratégico sobre los repositorios.
    11.1. Crear las interfaces que implementarán las diferentes estrategias.
    11.2. Generar la clase contexto de las implementaciones.

11.3. Programar la ejecución de estas interfaces.

11.4. Generar la primera implementación

12. Documentar el proyecto.

12.1. Documentación de los *endpoints*.

13. Crear un docker-compose para el despliegue.

13.1. El docker-compose desplegará las 3 imágenes.

13.2. Se configurarán las conexiones entre ellas.

La Figura 7 muestra un diagrama muy general donde se muestra las semanas en las que se planificaron cada una de las tareas. Esta figura es una imagen del original (Cruz, Diagrama de Gantt breve, 2020), que es una versión reducida del diagrama de Gantt extendido (Cruz, Diagrama de Gantt Extendido, 2020).

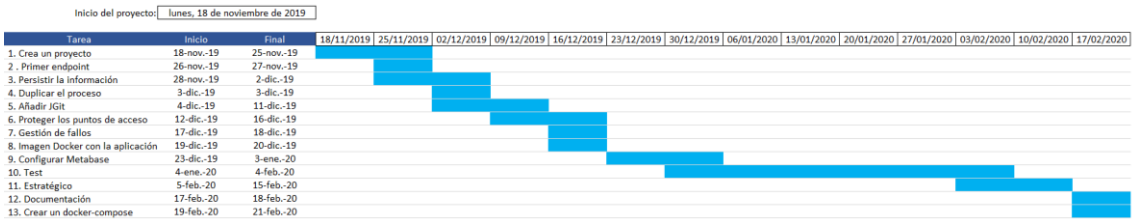

Figura 7. Diagrama de Gantt.

Con respecto a la duración de las tareas, mencionar que a la fase 4, destinada al proceso de Jenkins, se le otorgó una duración muy breve, dado que consistía en duplicar el modelo de SonarQube en una estructura ya creada en fases anteriores, de ahí su corta duración.

Otro detalle es que se planteó una fase de test muy exhaustiva, por lo que se le destinó una gran cantidad de tiempo. Aunque casi el mismo tiempo que para el proyecto pueda parecer una exageración, no resulta ser así en la práctica ya que el código que se necesita para validar una clase es mucho más voluminoso que la propia clase.

4. Diseño de la solución

En este capítulo se va a detallar el funcionamiento concreto de *platform-metrics-project-service*, en Arquitectura se especificará el patrón de comunicaciones entre los componentes y en Diseño se ilustrará el comportamiento de cada uno de los procesos.

Antes de eso se va a describir el modelo de datos para ofrecer una visión global de la estructura de la información de la aplicación.

Hay que señalar que los campos *id* de todas las tablas no es el clásico valor numérico auto incremental, sino lo que se conoce como UUID (Leach, Mealling, & Salz, 2005), acrónimo de *universally unique identifier* o Identificador único universal. Un valor de 32 dígitos hexadecimales que tienen una probabilidad de duplicado muy pequeña incluso en diferentes sistemas, a diferencia del id clásico, que casi siempre empieza por uno, existiendo este valor por tanto en varias tablas del sistema y en diferentes sistemas.

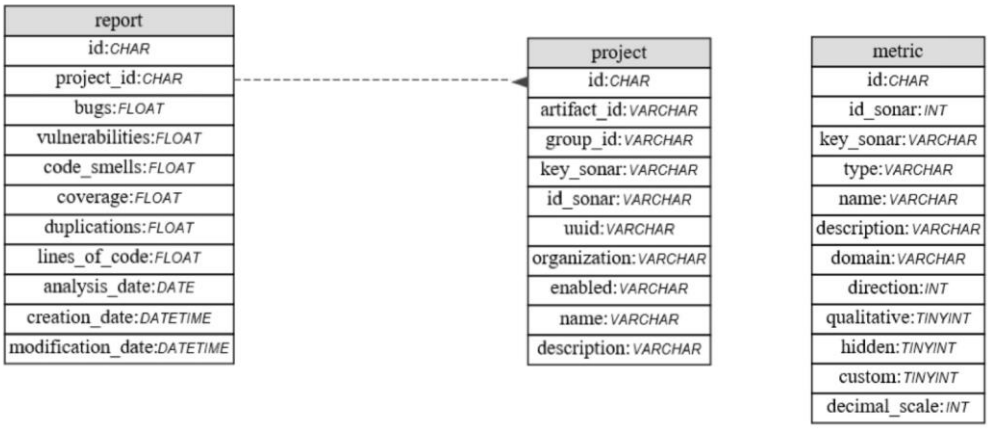

Figura 8. Tablas de SonarQube

Cuando SonarQube analiza un proyecto con sus más de 500 reglas, agrupa los resultados en un número más reducido de métricas. Estas se almacenan en nuestro sistema en la tabla *metric* por si en una nueva versión cambian estos valores y dificulta el seguimiento de algunas de ellas.

En la tabla *project* se almacenan las representaciones de los proyectos que tiene SonarQube, ya que resulta necesario obtener sus identificadores, campo *sonar_key*, para poder solicitar los resultados del último análisis de un proyecto concreto. Quiero destacar los campos *group_id* y *artifact_id* que identificarán al proyecto en todo el sistema de Jeff.

Finalmente, la tabla *report* contiene el resultado final que se quiere mostrar en Metabase. Tiene una relación con *project* de manera que un proyecto pueda tener muchos informes, pero un informe solo tenga un proyecto. Destacar los campos de fecha de creación (*creation_date*) y fecha de modificación (*modification_date*) que aportan información referente al propio sistema, mientras que fecha de análisis (*analisys_date*) se refiere a cuando Sonar lo analizó. Es en esta tabla donde se encuentran los valores mencionados en los objetivos de *bugs*, vulnerabilidades, *code smells*, cobertura de los test (*coverage*), líneas duplicadas (*duplications*) y líneas de código del proyecto (*lines_of_code*). Este último valor se guarda ya que, aunque no sea una métrica de calidad del código, sí que clarifica los valores de los anteriores.

He de señalar que se almacenan metadatos de SonarQube que no son relevantes en este proyecto, pero que en alguna ocasión resultaron ser útiles, como fue el caso de que no aparecía la métrica vulnerabilidades y explorando estos metadatos fue posible hallarla, por lo que se mantienen de cara a posibles cambios de versión de SonarQube que afecten a esta aplicación.

En las tablas de la Figura 9 encontramos *flyway_schema_history*, que es ajena al proyecto. Es una tabla que genera Flyway para gestionar las migraciones. La tabla *build* es la que almacena el resultado de la ejecución del proceso *deploy_pro* de Jenkins mencionado anteriormente. Contiene información sobre el nombre del proyecto sobre el que se ha ejecutado el proceso, su resultado y las tres fechas siguen el esquema de la tabla *report*, donde *build_date* se refiere a la fecha en la que Jenkins finalizó el proceso y las otras son referidas al sistema de mi aplicación. El campo *build_jenkins* es un histórico de ejecuciones de Jenkins que permite identificar el proceso en el servidor. Se almacena para en caso de fallo poder buscarlo en Jenkins de manera unívoca.

Las otras tres tablas corresponden al proceso de GitHub, donde *url_repo* contiene las urls necesarias para clonar los repositorios en el sistema local. Esta tabla es estática, no se va a modificar a través de la aplicación y solo se le añadirán filas de manera externa. Perfectamente podría haber sido un archivo dentro del proyecto, pero la habría hecho más inaccesible de cara a futuras modificaciones.

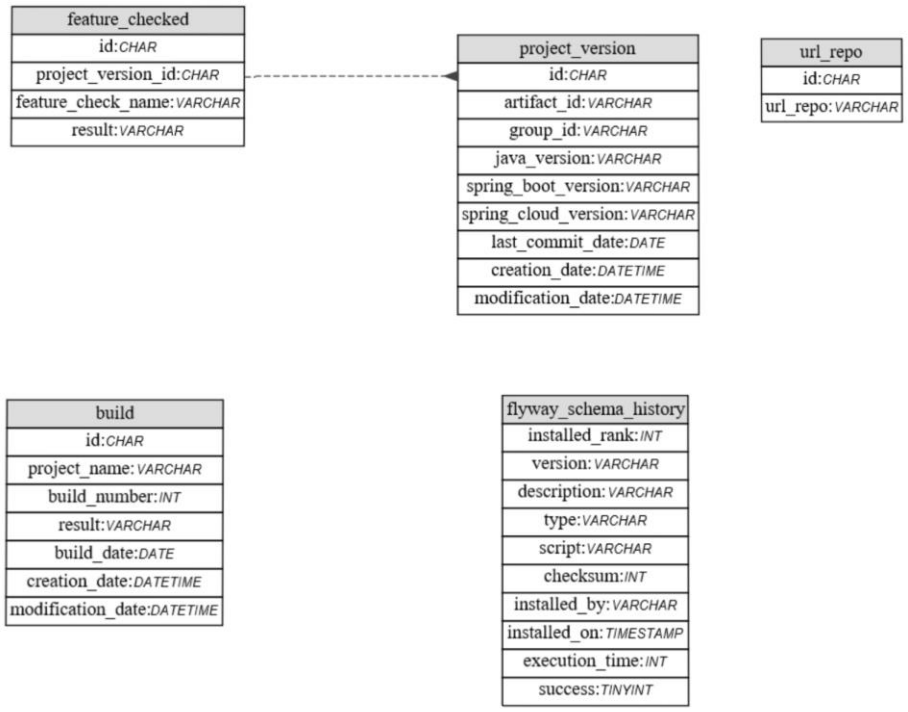

Figura 9. Tablas de *project_version* y *build*

Por lo que se refiere a *project_version*, contiene el resultado del análisis del `pom.xml`, donde *artifact_id* y *group_id* identifican al proyecto, *java_version*, *spring_boot_version* y *spring_cloud_version* almacenan las versiones del proyecto de esas tecnologías en cuestión. Se añade la fecha del último *commit* que ha recibido el proyecto para saber si ha recibido algún cambio reciente o si el proyecto lleva tiempo sin sufrir cambios. También están las dos fechas de creación y modificación que funcionan igual que las de *build* o *report.*

Finalmente, la tabla de *feature_checked* tiene una asociación con *project_version* de maneara que a una entrada de *project_version* le corresponden muchas de *feature_checked*. Una de estas últimas solo tiene una entrada de *project_version*. Esta tabla es la encargada de almacenar los análisis realizados por un programador ajeno. Solo se almacenan dos valores relevantes, el nombre de análisis, *feature_checked_name*, y su resultado, *result*, ya que se busca que sean cuestiones sencillas.

4.1. Arquitectura del sistema

El comportamiento del sistema es la ejecución de tres procesos que recogen datos y los almacenan. La Figura 10 muestra el esquema general que siguen y posteriormente se mencionarán las particularidades de cada uno de ellos.

He de señalar que la función del servicio es coordinar el orden de llamadas a los diferentes componentes de acuerdo con las necesidades del proceso que se esté abordando, así como la de proporcionar a cada componente los datos que necesita y recoger sus resultados para ofrecérselos al siguiente componente.

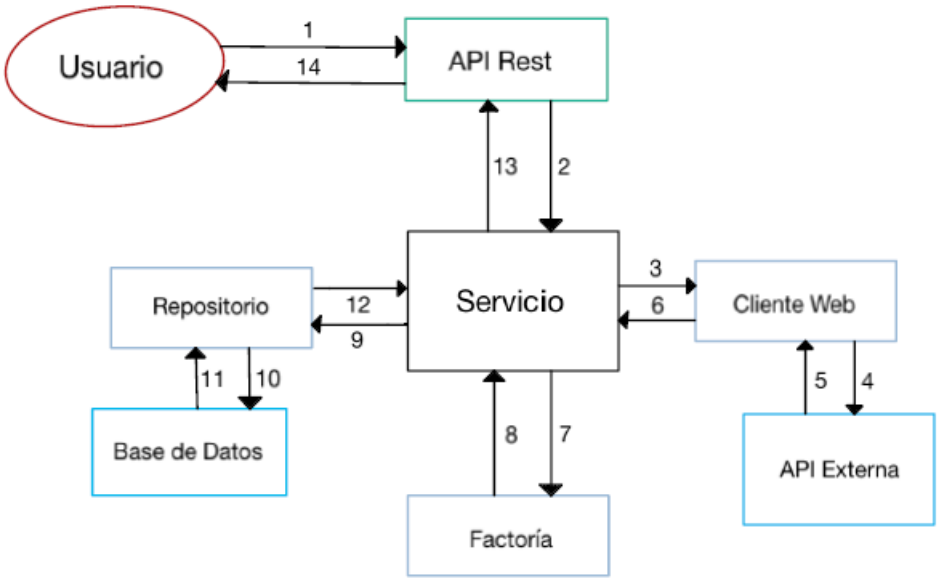

Figura 10. Esquema general

La idea es que cuando la API Rest que expone mi aplicación reciba una petición con una URL que reconozca como válida y autentificada con credenciales válidos llamará a un método concreto del servicio, el cual le pedirá a una fuente de datos externa a través de un cliente web lo necesario para satisfacer la petición del usuario.

Destacar que al cliente web se accede a través de un servicio que implementa Spring Retry. Este segundo servicio ofrecerá un método anotado con @Retrayable, el cual ejecutará el método del cliente web y si no recibe respuesta, volverá a ejecutarlo tras 500 milisegundos y si al tercer intento sigue sin responder, ejecutará el método anotado como @Recover, quien devolverá un objeto vacío, que el servicio principal desestimará. De esa manera se previene el posible fallo de que el servidor no conteste porque se encuentra saturado.

Otro aspecto del cliente web, es que las peticiones que realice a la API externa, entendida como la fuente de datos, es decir SonarQube, Jenkins o Github, están

autentificadas con credenciales válidas. En los tres casos se utilizó un token que daba acceso a los datos necesarios.

Con los datos recibidos, el servicio los pasará por la factoría que le devolverá objetos pertenecientes al modelo de datos de la aplicación. Estos objetos se almacenarán en la base de datos a través de un repositorio, tras asegurarse de que no se van a guardar duplicados. Tras guardarlos, el servicio le pedirá los datos y se los devolverá al usuario. Se busca mostrar el estado de la base de datos tras la inserción de nueva información.

Quiero destacar que las peticiones que realiza el usuario no requieren ningún tipo de parámetro más allá de las credenciales. El diseño está realizado para que los valores necesarios para conectarse con las aplicaciones externas se encuentren dentro del sistema. En este caso datos como los tokens para autentificar las peticiones, las URLs de las API Rest, credenciales de la base de datos y demás valores sensibles se almacenan en el contenedor Docker como variables de entorno a las que la aplicación accede al arrancar.

Una característica común a los tres procesos es que no se devolverá un objeto del dominio del sistema directamente, sino que se devolverá una Projection. Esta interfaz permite modificar los atributos del objeto que vamos a devolver, como se describe en el apartado de Tecnologías Utilizadas dentro del subapartado de Spring, esto va a evitar que se generen JSON infinitos.

Aunque el esquema general incluya la acción de un usuario, este no es el caso más frecuente. Lo que sucederá habitualmente es que los tres procesos se lanzarán automáticamente. Esto es así porque hay una clase que al estar anotada como @Component, Spring la reconocerá al arrancar y la instanciará en su contexto. Esta clase tiene un método anotado como @Scheduled y programado para que se ejecute del lunes al viernes a las seis de la tarde. Esta tarea ejecutará los métodos de los servicios que desencadenan los procesos de recolección de datos.

4.2. Diseño detallado

En este apartado se van a describir las particularidades de cada uno de los procesos de recogida de información.

Jenkins es el proceso más prototípico y se ciñe al proceso general, su API es muy sencilla. Solo es necesario proporcionarle los valores del nombre del proceso del

que se quieren solicitar los datos y los atributos que se quieren recibir. La gestión de duplicados es muy sencilla al tener el *build_number*, que identifica unívocamente esa ejecución en Jenkins, y ser una secuencia numérica ascendente, es cuestión de comprobar el último almacenado en la base de datos, y eliminarlos de los recibidos los menores o iguales a ese valor.

El proceso de SonarQube se distancia del esquema general en cuanto al cliente web. Debido a la estructura de su API Rest, el proceso es hacer una petición de todos los proyectos que tiene ese servidor SonarQube para posteriormente pedir las métricas de cada uno de los proyectos, asociado el identificador de este y las métricas que se solicitan en cada petición.

La gestión de duplicados es muy sencilla. Como la información relevante para Metabase se encuentra en la tabla *report* y SonarQube realiza un análisis diario, antes de insertar los nuevos objetos en la tabla, se eliminan los del día actual, de esa manera se asegura que no haya duplicados.

Es cierto que este método es poco eficiente de cara al número de borrados e inserciones que se realizan y se podía optimizar mucho más, pero se quiere evitar caer en la optimización prematura (Watson, 2017). Este concepto se entiende como dedicarles demasiado tiempo a aspectos como la escalabilidad o la optimización de procesos en etapas prematuras del desarrollo, cuando en esas etapas debería recibir más atención la aceptación del producto por parte de los clientes. En este caso, el exceso de operaciones sobre la base de datos se considera menos relevante que el estado actualizado de los informes de SonarQube, por lo que se busca una solución sencilla acorde con el problema actual.

El proceso de recolección de datos sobre los repositorios es el que más se sale del esquema, ya que no utiliza un cliente web convencional. Antes de nada, este proceso le pedirá a la base de datos las URLs de los repositorios, necesarias para realizar el clonado en el sistema local.

Con el código del proyecto clonado, se accederá al archivo `pom.xml` y haciendo uso de otra librería, Maven Model[22], se transformará este archivo a un objeto Java para acceder a sus propiedades de manera más fiable y cómoda. Este objeto es el que se le pasará a la factoría, que es donde se aplicará el patrón de diseño estratégico.

Con el objeto de la tabla *project_version* ya creado y con un UUID asignado, es cuando se iniciará el proceso de ejecución de los *feature checker*. Estas clases

[22] http://maven.apache.org/ref/3.6.0/maven-model/

implementan una interfaz con el método *check*, que recibe por parámetro un objeto contexto que contiene la ruta del repositorio en el sistema de ficheros local, y el objeto Java con el `pom`. Con esta estructura, en la factoría se ejecután todas las clases que implementen esta interfaz. A cada una de estas clases se le asigna lo necesario para ejecutar sus análisis y el resultado se almacena en la base de datos vinculado al *project_version* de su proyecto.

Algunos ejemplos de los análisis *feature checker* implementados son registrar la versión de la imagen de Docker que genera el proyecto, comprobar si tiene determinada configuración de seguridad, ver si el proyecto utiliza imágenes de JIB[23] o contar las veces que se genera un UUID en los archivos de migración

Por último, se va a mostrar la estructura de los paneles de Metabase. La Figura 11 muestra un menú con los seis paneles, donde *Archetype versions*, *Artifact versions* y *Migrations* contienen datos obtenidos de los repositorios, *Delivery metrics* sobre los despliegues ejecutados mediante Jenkins y finalmente *Quality metrics* y *Quality metrics evolution* ofrecen la información de los análisis de SonarQube.

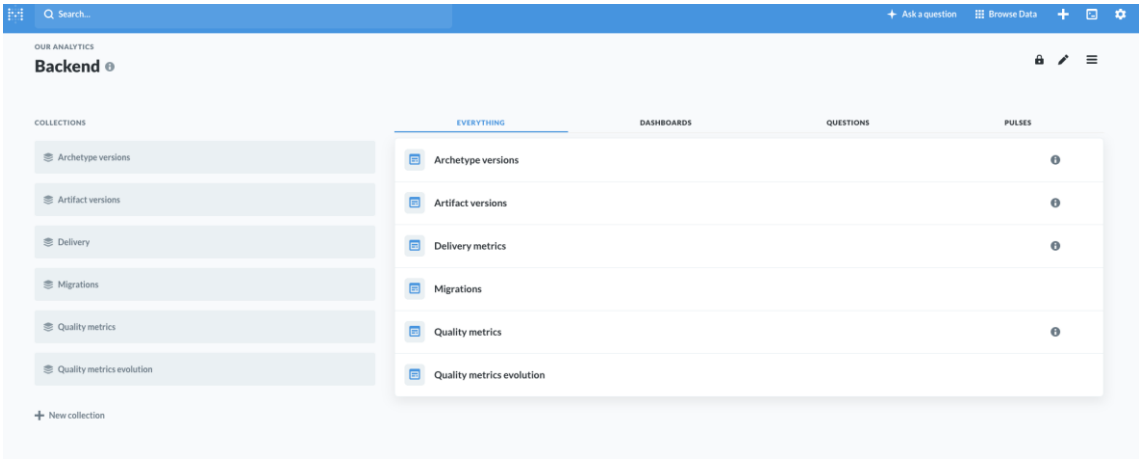

Figura 11. Menú de Paneles

La Figura 12 muestra en el gráfico la cantidad de despliegues que se han producido cada semana, así como los diez proyectos que más cambios han recibido.

[23] https://github.com/GoogleContainerTools/jib

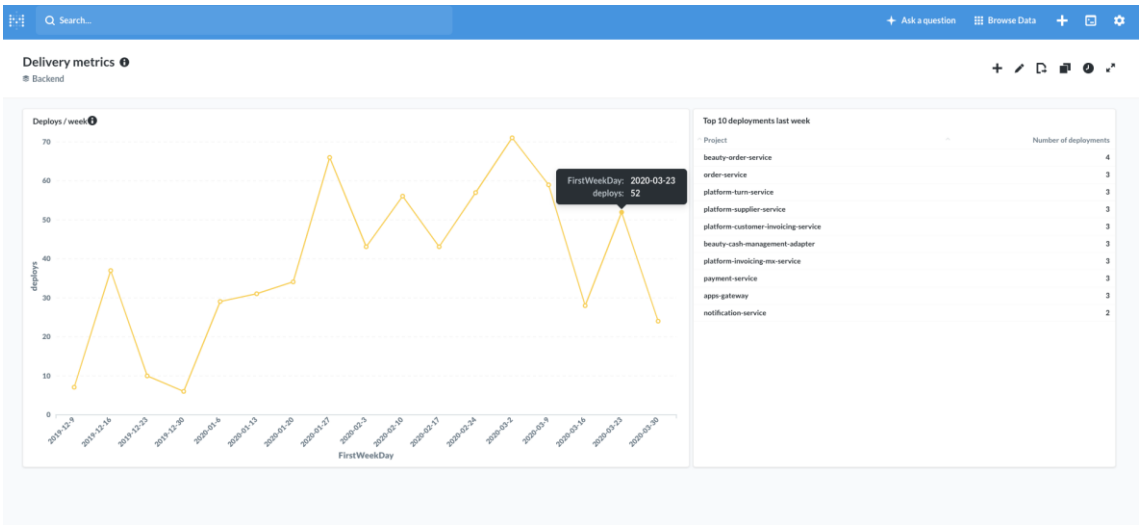

Figura 12. Métricas de despliegue

En la Figura 13 vemos cuatro gráficos, cada uno de ellos muestra la evolución semanal de cada una de las métricas en todos los proyectos de la empresa, así como la marca, *goal*, que el departamento se propone alcanzar en el próximo trimestre.

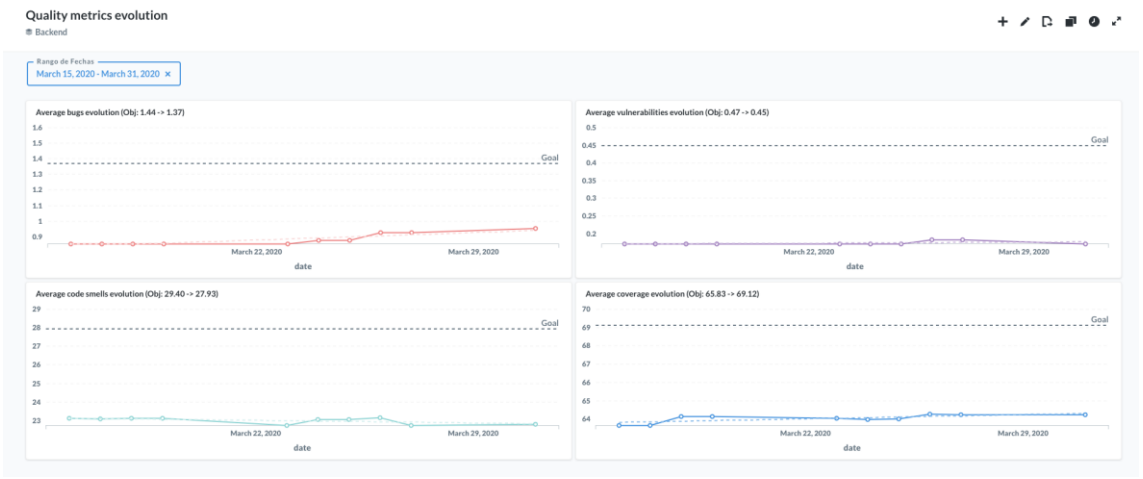

Figura 13.Evolución de las métricas de calidad

La siguiente figura muestra la evolución semanal de cada uno de los proyectos, comparando la mejora que ha habido de una semana a otra y en la vista de *Last reports ordered*, muestra las métricas actuales de los proyectos. Está ordenada alfabéticamente por el nombre del proyecto. Los recuadros de la derecha muestran un resumen de la información mostrada en la Figura 13.

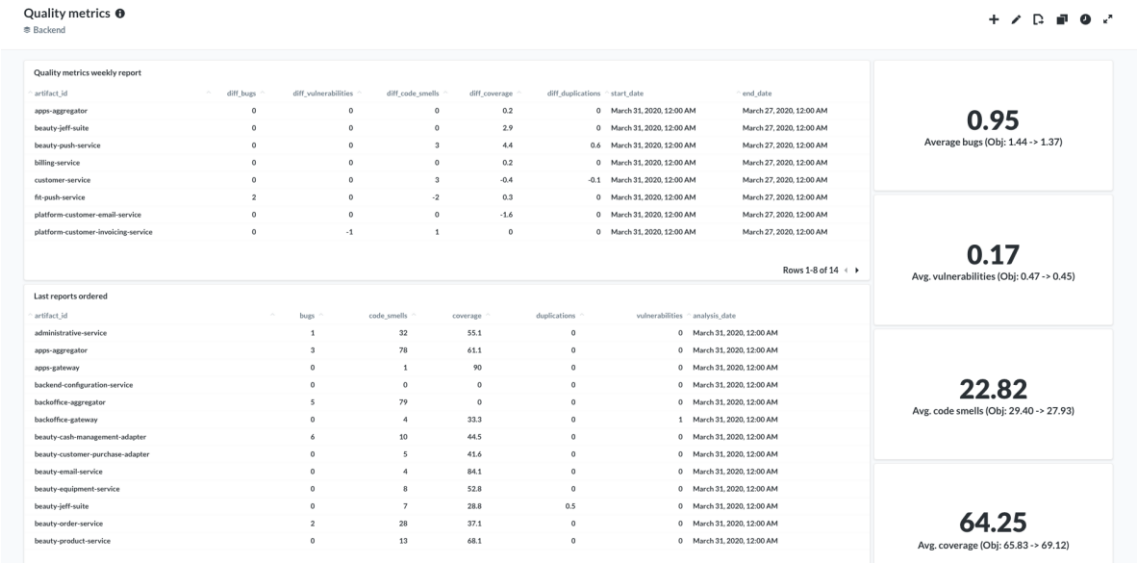

Figura 14. Métricas de calidad

La Figura 15 muestra los resultados de los análisis de los repositorios, en el primero se ven las versiones de Java y cuántos proyectos tienen cada una de ellas, y lo mismo para las versiones de Spring Boot y Spring Cloud.

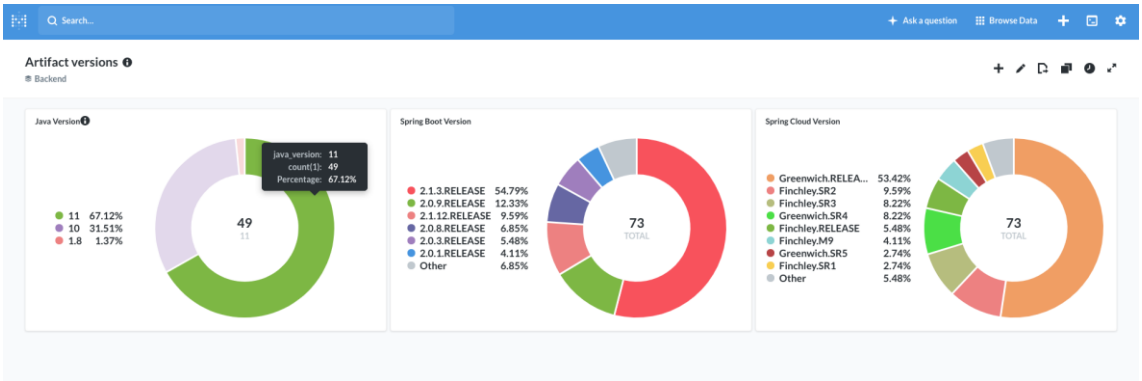

Figura 15. Versiones de los proyectos

La figura 16 ofrece los resultados de dos *feature checker*, el primero referido a los proyectos que han migrado de los contenedores Docker a los Jib y el segundo sobre si existe una determinada configuración de seguridad en el archivo `bootstrap.yml`.

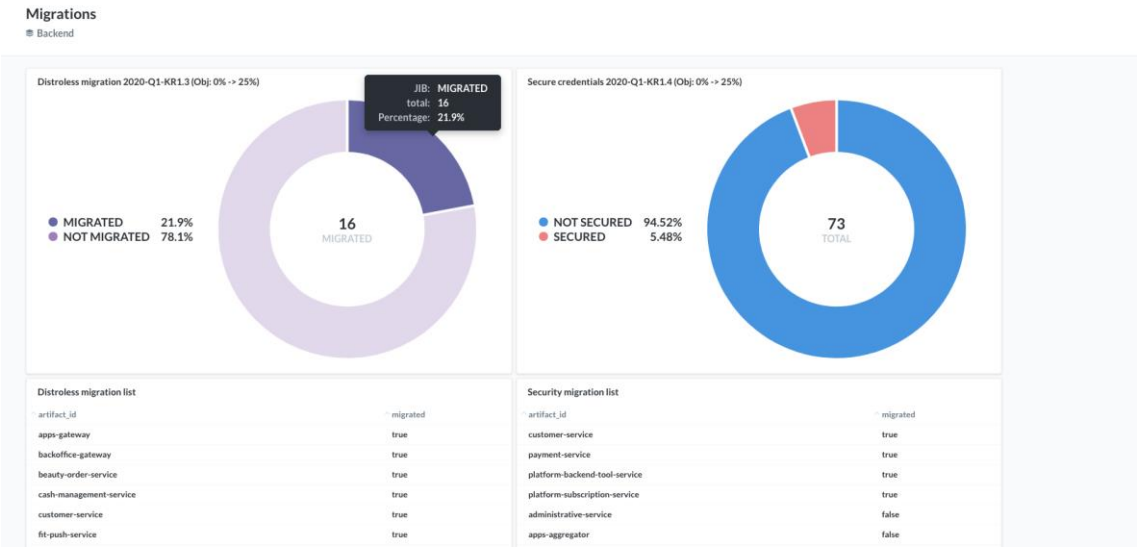

Figura 16. Migraciones

4.3. Tecnología utilizada

En este apartado se van a describir en profundidad las diferentes herramientas utilizadas para el desarrollo de este trabajo de fin de grado, así como las motivaciones por las que se han incluido estas y no otras.

4.3.1. SonarQube

Es una plataforma de código abierto para evaluar código fuente. Lleva a cabo análisis estáticos que se basan en un sistema de reglas obtenidas de acuerdo con los estándares de las siguientes convenciones:

1. *Common Waekness Enumeration* (CWE) [24]: Una lista creada y mantenida por una comunidad de desarrolladores enfocada a señalar debilidades comunes tanto a nivel de hardware como de software. Pretende aportar una medición para las herramientas de seguridad, así como una línea base para la identificación de debilidades y el esfuerzo que implica su prevención.

2. *Open Web Application Security Project* (OWASP) [25]: Una fundación sin ánimo de lucro dedicada a la mejora de la seguridad web. Uno de sus

[24] https://cwe.mitre.org/
[25] https://owasp.org/www-project-top-ten/

proyectos es el OWASP Top Ten, un documento que se actualiza anualmente con las 10 vulnerabilidades más frecuentes de ese año y cómo evitarlas. Representa un amplio consenso sobre los riesgos de seguridad más críticos para las aplicaciones web.

3. SANS *top25*[26]: El instituto Sans se estableció en 1989 como una organización cooperativa de investigación y educación, como OWASP también mantiene una lista de los 25 errores software más dañinos.

4. *The Motor Industry Software Reliability Association* (MISRA)[27]: Esta asociación creó el documento MISRA C que agrupa recomendaciones para el desarrollo en el lenguaje C. Pretende proveer directrices sobre seguridad y fiabilidad del código fuente, así como buenas prácticas para los sectores automovilísticos, ferroviarios, espaciales y otros relacionados.

5. *CERT Coordination Center*[28]: Con su iniciativa *The Vulnerability Notes Database* publica en GitHub las diferentes vulnerabilidades software que ha identificado, añadiéndole detalles técnicos e información sobre la recuperación ante ataques. Este proyecto permite que gente ajena al mismo contribuya con vulnerabilidades que encuentre.

El análisis al que somete el equipo de SonarQube a estas fuentes da como resultado 540 reglas aproximadamente, dado que su número varía ligeramente entre versiones, que agrupan en los siguientes factores:

1. Complejidad.

En este factor agrupa las métricas de complejidad ciclomática y cognitiva, ambas referidas a la dificultad que puede tener el código fuente de ser leído y entendido por un programador ajeno al desarrollo del proyecto. Si determinados métodos del código fuente pudieran convertirse en frases, estas métricas indicarían cuanto de difícil es la frase de entender por la cantidad de condiciones o acciones que en ella se mencionan.

2. Duplicados.

Estas métricas se refieren a la cantidad de código repetido que hay dentro de la aplicación. La presencia de estas duplicidades afecta negativamente al tiempo de compilación y de carga del proyecto.

[26] https://www.sans.org/top25-software-errors/

[27] https://www.misra.org.uk/

[28] https://www.kb.cert.org/vuls/

3. Fiabilidad.

Aquí se agrupan mediciones relacionadas con la presencia de fallos dentro del código que pueden ocasionar una interrupción del servicio que presta la aplicación, es decir, los *bugs*.

También incluye el concepto de deuda tecnológica, entendida como el tiempo que a un programador le puede costar arreglar estos defectos.

4. Mantenibilidad.

Los *code smells* se encuentran en esta categoría, siendo la métrica más representativa de la misma.

Estos se traducen como código que huele mal, y se definen como fragmentos del código fuente que deberían estar formulados más de acuerdo con la lógica subyacente del lenguaje o del *framework* con el fin de adecuarse a un estándar.

Además, añade la deuda tecnológica asociada a estas métricas.

5. Seguridad.

Aquí es donde SonarQube ubica características del código que pueden fallar por acciones de agentes externos a la aplicación, las que denomina vulnerabilidades.

6. Issues.

Esta categoría es residual, es donde caen las métricas que se consideran relevantes, pero cuya funcionalidad no está tan delimitada como las anteriores.

7. Tamaño.

Alude a la magnitud del proyecto, permite valorar el impacto de las medidas anteriores, ya que no es lo mismo que se detecten 100 *code smells* en un programa de 200 líneas que en uno de 200.000.

También incluye métricas de interés como número de líneas de comentarios o número de clases que utiliza el programa.

8. Test.

Aunque SonarQube no ejecuta los tests, sí que está preparado para recoger los informes que dejan otras aplicaciones que los ejecutan. De estos informes recoge métricas como la cobertura global que ofrecen estas pruebas, entendida como la cantidad de líneas del código que cubren los tests, así como el número de líneas no cubiertas o el número de pruebas que tiene el proyecto.

Se eligió SonarQube frente a otras alternativas como Kiuwan[29] por hallarse implantado en la empresa y tener más conocimiento de la misma. Las diferencias entre ambas no son notables, pues el disponer de una API REST, la integración con la mayoría de los IDE, tener una base de reglas y ejecutarlas en un servidor externo son características compartidas por ambas.

4.3.2. Maven

Es una herramienta de código libre, bajo la licencia de *Apache Software* que en su origen se desarrolló para sistematizar el proceso de compilación de un proyecto basándose en el formato XML.

Esto anteriormente se hacía a través de procesos Ant[30], pero resultaba ser un proceso costoso, delicado y susceptible de modificarse a menudo. Maven ofrece un estándar de compilación y la posterior generación de un ejecutable a partir de un proyecto Java de manera muy sencilla. No depende del sistema operativo en el que se encuentra, está desarrollado en Java y se ejecuta sobre su máquina virtual.

Su modelo se basa en el archivo `pom`, donde en formato XML se almacenan los diferentes módulos de los que hace uso el proyecto, así como sus versiones y los repositorios de donde descargarlos.

Por tanto, en el proceso de compilación, el motor de Maven lee el fichero `pom`, y descarga los módulos de los que depende el proyecto de los repositorios indicados, y procede a su compilación y posterior generación de un ejecutable.

Otro concepto valioso de Maven es el del ciclo de vida, donde se establece el orden y las acciones a las que Maven somete a un proyecto. Las fases más importantes de este ciclo son las siguientes:

1. Compilación: Generación de archivos ejecutables a partir de las clases Java.
2. Test: ejecuta las pruebas que contenga el proyecto, abortando el proceso de ciclo de vida en caso de que alguno falle.
3. Empaquetado: Como consecuencia de la compilación genera un archivo ejecutable.

[29] https://www. kiuwan.com
[30] https://ant.apache.org/

4. Instalación: Lleva el fichero ejecutable a su directorio, donde tiene los otros módulos compilados.
5. Despliegue: Copia el fichero ejecutable a un repositorio, para que el proyecto pueda ser importado como un módulo dentro de otro proyecto.

Este ciclo de vida está preparado para añadir pasos adicionales como puede ser que SonarQube analice el proyecto o que se genere una imagen Docker tras el empaquetado haciendo uso de la librería dockerfile-maven[31]. Lo que resulta muy conveniente dado que estos ejemplos son requisitos de mi proyecto.

Una ventaja en el uso de Maven es que dispone de una librería que permite generar un Maven *Wrapper*, es decir, un envoltorio que se introduce en el proyecto de manera que sea posible ejecutar su ciclo de vida en una máquina que no tenga instalado Maven, lo que permite aislar aún más el proyecto del sistema en el que se encuentre.

Una alternativa a Maven es Gradle, que aporta un modelo similar, con la particularidad de estar construido sobre Groovy, en lugar de sobre Java. Un aspecto positivo de Gradle es que funciona con Kotlin, además de con Java.

En este proyecto se ha optado por Maven, por ser un proyecto con más trayectoria y estar ampliamente documentado, además de que al no desarrollarse en Kotlin, Gradle no aporta ninguna ventaja significativa, pues ambas herramientas están integradas en la mayoría de los IDE.

4.3.3. Docker

La idea que subyace a Docker es la de facilitar el despliegue de una aplicación aislándola del sistema operativo de la máquina en la que se encuentra. De esta necesidad surge el concepto de contenedor, que pretende ser una imagen preconfigurada con todo lo necesario para que la aplicación funcione.

Docker implementa esta funcionalidad y permite su fácil configuración a través del `Dockerfile`, fichero que define la imagen.

Esta tecnología aporta la herramienta Docker-compose, que simplifica el uso de los contenedores. A partir del fichero `docker-compose.yml`, crea una red donde genera y conecta los contenedores que se encuentran mencionados con la configuración indicada.

[31] https://github.com/spotify/dockerfile-maven

Una característica interesante, es que existen librerías que permiten generar una imagen que contenga el proyecto de Spring, de manera que cuando Maven genere el empaquetado del proyecto, esta se genere con el ejecutable del proyecto.

Una posible alternativa a esta tecnología es Jib, desarrollada por Google, que permite generar contenedores que no tengan instalada una consola. Esto aporta seguridad, ya que en caso de que consigan entrar al contenedor, no podrán ejecutar nada por línea de comandos.

Se ha decidido trabajar con Docker por tener más experiencia y familiaridad en su uso, ya que Jib es de aparición relativamente reciente y no existe tanta documentación sobre su uso en la red. Además de que la ventaja no es notable en este proyecto, ya que se desplegará en un entorno ya securizado.

Sin embargo, más adelante se puede considerar cambiar los contenedores Docker por los de Jib.

4.3.4. GitHub

Esta herramienta ofrece un espacio en la nube, donde subir los ficheros que componen un proyecto. A este espacio se le llama repositorio, y permite tanto acceder a él desde diferentes dispositivos, como que otras personas puedan acceder a él. Motivo por el que se utiliza para compartir código y para trabajar en equipo, ya que permite que varios programadores puedan hacer cambios en un repositorio.

GitHub supone solo la interfaz. Las acciones que se pueden realizar sobre un repositorio son las que define Git, proyecto al que GitHub envuelve y amplia.

Estas acciones están diseñadas para permitir y facilitar un control de versiones. Uno de los casos de uso más frecuentes de esta gestión de versiones es el de añadir una nueva funcionalidad a un proyecto que se encuentra desplegado en producción. Para evitar que un fallo no previsto origine una caída del servicio, lo habitual es copiar el repositorio en un ordenador ajeno al entorno de producción y hacer las ampliaciones, cambios y prueba pertinentes, de modo que cuando se esté seguro del funcionamiento, fusionarlo con lo que se encuentra en el repositorio y que esa versión se despliegue en producción.

Una característica valiosa es su integración con Jenkins, que facilita la práctica de la integración continua, como se detallará más adelante.

Adicionalmente, existe una librería, JGit, que permite ejecutar desde Java los comandos propios de Git para la gestión de los repositorios. Este es el motivo fundamental por el que mi proyecto se desarrolla enfocándose a proyectos que estén almacenados en GitHub.

4.3.5. Jenkins

Es una de las herramientas que permiten la práctica de la integración continua. Es un servidor de automatización que permite secuenciar pequeñas tareas, las cuales pueden estar directamente relacionadas con el proceso de despliegue.

Las ventajas por las que se ha elegido trabajar con Jenkins es que permite ejecutar los procesos de Maven (la compilación, los test…) y además está integrado con GitHub, de manera que puede descargar repositorios y responder a eventos lanzados por dicha web. Además, dispone de una API REST que permite acceder a los resultados de la ejecución de determinada tarea sin necesidad de acceder al panel de control de Jenkins.

Una posible configuración de Jenkins que ejemplificaría su funcionamiento puede ser la siguiente.

Se ha configurado un repositorio de GitHub, para que cuando reciba una solicitud de aplicar un cambio en la rama principal del repositorio (generalmente en esta rama se encuentra la versión del proyecto que está desplegada en producción), avise al servidor de Jenkins para que este se descargue el archivo `jenkinsFile` del repositorio. Este archivo contiene las tareas que Jenkins debe de ejecutar, las cuales son pedirle a Sonar que analice el proyecto, luego compilarlo, ejecutar los test unitarios y los de integración, para posteriormente empaquetar el proyecto, generar una imagen de Docker y subirla al repositorio con el resto de las imágenes, donde esperará ser desplegada.

4.3.6. MySQL

Es un sistema gestor de bases de datos relacionales de código abierto que se basa en el modelo de cliente servidor, lo que facilita su uso a través de la red.

Aunque existan numerosas alternativas se ha elegido esta por su facilidad de uso, por la extensa documentación que existe en internet sobre su uso y porque el

modelo de datos de la aplicación no exige ningún requisito que MySQL no pueda satisfacer.

4.3.7. Sequel Pro

Es una aplicación de escritorio con herramientas que facilitan la gestión de bases de datos, permite ejecutar consultas, conectarse con varias bases de datos simultáneamente, visualizar la base de datos y realizar exportaciones.

Se ha elegido sobre otros gestores más potentes, como Oracle SQL Developer, ya que las funcionalidades que aportan son las requeridas para este proyecto y su interfaz es mucho más sencilla que las otras opciones.

4.3.8. Flyway[32]

Esta herramienta permite llevar un control sobre el estado de la estructura de la base de datos.

Dentro del proyecto existirán unos archivos `.sql`, que contendrán las instrucciones necesarias para construir la base de datos de la aplicación, así como un orden de ejecución. Cuando el proyecto se ejecute, Flyway leerá estos archivos y comprobará si se han aplicado en la base de datos. De no ser así, averiguará cuál es el último que se ha aplicado, y ejecutará los siguientes. Pero, si no encuentra ninguno, detendrá la ejecución del proyecto.

Esta herramienta fuerza al programador a tener que generar un archivo de migración cada vez que pretenda alterar la base de datos de producción, de esta manera, se tratan de evitar modificaciones en la estructura que puedan originar un fallo en las bases de datos desplegadas, que derive en una suspensión del servicio, o, en el peor de los casos una pérdida de información.

Aunque este tipo de herramientas son propias de la fase de desarrollo, la práctica de la integración continua, así como la constante mejora del producto, requiere el uso de este tipo de herramientas para facilitar el mantenimiento de la aplicación, así como para la implementación de nuevas características solicitadas.

[32] https://www.flyway.org/

4.3.9. Spring

Es un gran *framework* que agrupa todas las librerías necesarias para generar una aplicación web Java. Su componente principal, Spring Boot Starter, contiene librerías para gestionar las conexiones HTTP, para la gestión de mensajes en formato JSON y para el manejo del modelo de datos, no solo el envío de información con una base de datos, sino también la correspondencia entre los objetos Java de la aplicación con las tablas donde persistirán sus valores.

Se ha utilizado en este proyecto por lo sencillo de su configuración. Spring permite que toda la configuración del servidor web, que antes consistía en puntillosas modificaciones en extensos archivos en formato XML, sea transparente al programador. Además, aporta la facilidad de generar un archivo JAR que ya contiene dentro un servidor Tomcat, evitando el pesado paso de tener que desplegar el archivo WAR en el servidor web y esperar a observar sus cambios.

Una de las principales características de Spring, implicada en la escasa configuración que requiere para su funcionamiento más básico, es la inversión de control. Básicamente cambia el funcionamiento habitual en Java en la que el programador debe crear los objetos que se van a introducir como parámetros en el constructor de otro objeto. Este *framework* permite que el programador indique, mediante anotaciones, que determinados objetos van a ser utilizados por otros componentes de la aplicación, de manera que Spring automatiza su gestión, tanto en la instanciación como en la gestión de la paralelización.

A estos objetos anotados se les conoce como *Beans*, y corresponden a componentes como los servicios, los repositorios, etcétera. Básicamente, son objetos troncales para el funcionamiento de la aplicación.

La mayoría de los componentes que se van a utilizar en este proyecto son *Beans*, ya que su funcionamiento es muy genérico, por lo que se especificará el comportamiento concreto cuando se hable del componente más adelante.

Dentro de este *framework* se han empleado las siguientes librerías:

OpenFeign

Estos son los clientes web que se van a utilizar para llevar a cabo las llamadas a las APIs externas de Sonar y Jenkins.

Este cliente permite la sencilla construcción de URLs y la asignación de parámetros, también permite la configuración de la autentificación de las

llamadas, la asignación del método de la operación HTTP y la construcción del cuerpo del mensaje.

Una alternativa serían los clientes web RestTemplate que permiten una configuración más precisa, pero OpenFeign tiene una implementación más sencilla y sus funcionalidades se ajustan a los requisitos de la aplicación.

Spring Data

De esta librería se utilizará el componente Entity o entidad, que permite asignar correspondencias entre las clases Java con la estructura de la base de datos y el componente Repository o repositorio, que facilita la construcción de las consultas gracias al conocimiento de las entidades.

RestController

Estos componentes permiten exponer URLs REST, que se van a utilizar para ejecutar los procesos de la aplicación. Su configuración permite definir qué operaciones HTTP van a soportar, qué parámetros van a coger de la URL y qué objetos Java se van a devolver en formato JSON como respuesta al proceso ejecutado.

Spring Security

Con el fin de añadir seguridad a las URLs expuestas se incorpora este componente que permite la configuración del protocolo *Basic Authentication*, que exige credenciales válidas a las peticiones HTTP que lleguen a la aplicación.

Scheduling

Para cumplir con el requisito de que se ejecuten periódicamente una serie de procesos desde la aplicación, se emplea este componente, que permite programar la ejecución de determinado método con una frecuencia temporal concreta.

Spring Retry

Este pequeño componente es una medida frente a posibles fallos de sobrecarga del servidor al que se realiza la llamada HTTP. Permite anotar algunos métodos para que en caso de fallo lo vuelva a intentar tras un tiempo determinado, y cuando agota los intentos permite ejecutar otros métodos para que la aplicación continúe funcionando. No es más que repetir el backoff de TCP, pero eligiendo los pasos a seguir en caso de fallo.

Projection

Este componente permite a partir de un objeto Java, elegir los atributos con los que se va a generar el JSON e incluso cambiar los campos en esos mismos atributos.

Se utiliza por dos motivos, el primero es para que los objetos del dominio de datos de la aplicación, lo que son las entidades, no salgan de la aplicación, por seguridad y por resistencia a fallos, ya que son objetos susceptibles de tener comportamientos extraños al tener fuertes relaciones con la base de datos.

El otro motivo es uno de los fallos más evidentes. Para imitar el comportamiento de las claves ajenas de la base de datos, estas entidades guardan relaciones entre ellas del tipo de uno a muchos o muchos a muchos. Si esto no se gestiona debidamente, pueden dar lugar a JSONs infinitos, ya que en una relación muchos a muchos, una entidad contiene a la otra y viceversa.

4.3.10. Test

Los tests son una herramienta que permiten comprobar la funcionalidad del código de la aplicación, son un elemento necesario para asegurar el correcto funcionamiento de la aplicación y validar que las funcionalidades establecidas en los requisitos suceden tal y como se han especificado.

Son de utilidad también para asegurar el buen funcionamiento de la aplicación en caso de errores.

Dentro de las diferentes librerías que ofrece Java para esta funcionalidad se ha optado por las siguientes:

JUnit5[33]

Esta tecnología se utiliza para llevar a cabo los test unitarios, lo que permite comprobar el funcionamiento de funciones a partir de la variación de los parámetros de entrada y la comprobación de los valores de salida.

Su funcionamiento es el de generar un contexto de ejecución mínimo para ejecutar las funciones de manera aislada al comportamiento del resto de aplicación y así poder llevar a cabo las comprobaciones oportunas.

Se va a utilizar para comprobar que los métodos hacen lo que se espera de ellos y no hacen lo que no se espera de ellos.

[33] https://junit.org/junit5/

Con la finalidad de facilitar la legibilidad del código se va a utilizar la implementación AssertJ [34], que reescribe los métodos de JUnit5 para la comprobación de la salida de las funciones facilitando su lectura.

Mockito[35]

Uno de los problemas de JUnit es que no permite comprobar el funcionamiento de métodos que dependen de otras clases, debido a lo reducido del contexto de ejecución que genera, lo que impide realizar los test adecuados sobre los servicios.

Para solventar este inconveniente se va a utilizar la librería Mockito, que permite generar *mocks*, objetos que dan una implementación "falsa" a una interfaz, de manera que se puede simular su comportamiento determinando qué valores van a devolver sus funciones cuando se les invoque con determinados parámetros de entrada o la generación de excepciones ante determinadas situaciones.

Estos *mocks* permiten además recoger valores tales como los parámetros de entrada con los que se han llamado y las veces que se ha llamado a determinada función entre otros.

MockMvc[36]

Los últimos tests que se van a emplear son los de integración, los cuales comprueban el funcionamiento de aquellas partes de la aplicación que dependen de componentes externos.

La librería que se va a utilizar permite desarrollar tests que aseguren el funcionamiento de las URLs expuestas por la aplicación.

Esta tecnología va a levantar el contexto de Spring que permite reconocer peticiones HTTP, por lo que se pueden realizar *mocks* que simulen la respuesta de la aplicación.

Se va a utilizar esta librería porque además de encontrarse integrada con Mockito, facilita la generación de documentación de las URLs.

Spring Rest Docs[37]

[34] https://joel-costigliola.github.io/assertj/

[35] https://site.mockito.org/

[36] https://www.baeldung.com/integration-testing-in-spring

[37] https://www.baeldung.com/spring-rest-docs

Esta librería, que amplía la funcionalidad de los test de integración que se van a emplear en este proyecto, facilita la documentación de las URL expuestas en la aplicación.

Son métodos aplicables a los objetos MockMvc que generan documentación sobre los parámetros de entrada y de salida de cada uno de los *endpoints*, destacar la relevancia de que esto se haga a la vez que los test, pues es allí donde se valida el comportamiento de los procesos.

4.3.11. Java 10

Desde la versión 9, que salió en septiembre del 2017, Oracle ofrece una nueva versión de Java cada 6 meses. Esto facilita que se puedan ir subiendo los proyectos de versión poco a poco, ya que no habrá un cambio tan drástico que impida la actualización de los proyectos. Además, el ir añadiendo poco a poco funcionalidades, facilita la adaptación de los programadores a las nuevas características añadidas.

En este proyecto se va a utilizar Java 10 o cualquiera de las versiones subsiguientes, pues el TFG utiliza características del lenguaje introducidas en esta versión 10. A continuación, se van a detallar algunas de las funcionalidades que se encuentran en esta versión y que será utilizadas en el proyecto:

Inferencia de tipos local

Esta característica fue introducida en la décima versión y pretende eliminar la redundancia en la declaración de tipos. Muchos lenguajes modernos deducen el tipo de la variable a partir de su instanciación, Oracle pretende incorporar esta funcionalidad gracias al tipo *var*. Aunque no aporta un cambio en el comportamiento del lenguaje, sí que elimina la información redundante en el código fuente, facilita la legibilidad del código para el programador.

Streams

Esta característica se introdujo en Java 8, pero se ha ido mejorando con las siguientes versiones y supone un acercamiento a la programación funcional.

Ya que una de las actividades más frecuentes es el trabajar con listas de objetos, se incorporaron al lenguaje los *streams*, que permiten substituir bloques *for* por llamadas a funciones que ofrecen una implementación de los procesos más habituales a la hora de recorrer colecciones de objetos, como puede ser la

ordenación, la agrupación en una colección, el filtrado sobre el valor de un atributo o la conversión de un tipo a otro.

Esto favorece la lectura del código, así como reduce la carga cognitiva a la hora de programar.

Optional

Una de las comprobaciones más típicas y tediosas es del *null*. Es ineludible cuando el valor de una variable depende de agentes externos a nuestra aplicación y supone un aumento en la complejidad ciclomática del código.

En Java 8 aparecieron los *optional*, que envuelven cualquier objeto, y aportan una serie de métodos, entre ellos isPresent() o get(), que añaden una capa de abstracción, facilitando la comprensión en la lectura, y acercándose al lenguaje coloquial en lo que supone la resolución del problema.

Anotaciones

Aunque no es un elemento que en este proyecto se vayan a crear nuevas anotaciones, sí que se van a utilizar, ya que es la manera en la que Spring establece que se deben hacer uso de sus funcionalidades.

Básicamente, marcan determinados elementos como objetos que recibirán la implementación de una interfaz de una librería concreta haciendo uso de la reflexión de Java.

Un ejemplo es cuando se anota una clase como @Entity, el compilador la reconocerá como objeto que tiene una representación en la base de datos, y le otorgará los métodos oportunos para su interacción con la misma.

4.3.12. Metabase

Es una herramienta para la visualización de datos. A través de la conexión con una o varias bases de datos, puede ejecutar consultas SQL, relativamente complejas, y mostrar sus resultados.

Su potencial reside en que puede formatear los datos con gráficos, y generar agrupaciones en tableros, dando estructura y organización a la información obtenida.

El único requisito para su uso es cierto conocimiento de SQL, para poder generar las consultas, pero a partir de ahí, la herramienta ofrece una gran versatilidad.

Una posible alternativa sería Tableau, que, aunque ofrece más herramientas para su visualización de los datos, no está tan adaptada a la construcción de consultas, por lo que se optará por el uso de Metabase, ya que algunos de los requisitos planteados son más fáciles de conseguir realizando una consulta a la base de datos.

4.3.13. Postman[38]

Es una aplicación de escritorio que permite construir y ejecutar peticiones HTTP. Ofrece una interfaz amigable a la hora de añadirle campos de cabecera y parámetros en la petición, así como para un cuerpo del mensaje con determinado formato. Además, permite guardar las consultas realizadas.

Es muy utilizada a la hora de hacer pruebas en el momento de desarrollo tanto para comprobar si funcionan los servicios expuestos, como para explorar las API REST de otras webs.

Su alternativa es una aplicación similar que ofrece Google en forma de extensión, pero se ha elegido esta por no depender del funcionamiento de un navegador.

[38] https://www.postman.com/

5. Desarrollo de la solución propuesta

Cabe recordar que, al encontrarnos en la metodología en espiral, cada una de las fases planeadas en el análisis del problema corresponde con una iteración del modelo donde se atravesaban las fases de propuesta, investigación, implementación y pruebas. Señalar que la mayoría de los resultados de la fase de investigación están expuestos en este documento describiendo posibles alternativas y justificando las decisiones tomadas y que las soluciones concretas sobre las que se realizó la implementación coinciden con lo expuesto.

Durante el proceso de desarrollo el seguimiento de tareas pendientes, finalizadas y en proceso se llevó a cabo mediante Jira[39], un tablero de tareas sencillo que permite describirlas y asignarlas a determinados usuarios.

Es en ese tablero donde se planteaba la siguiente tarea, se proponían algunas posibilidades de implementación, y se señalaba la escogida. Facilitaba además el seguimiento por parte del tutor de la empresa de las tareas realizadas.

El proceso de desarrollo sucedió siguiendo las fases planteadas en el apartado de análisis del problema. Sin embargo, la temporalidad de las tareas no sucedió de acuerdo con lo planteado en un principio. La fase de test se extendió aún más, por ser tecnologías con las que no estaba tan familiarizado y requirieron mayor investigación por mi parte.

Sin embargo, el proyecto de desplegó antes de que esa fase terminara, por lo que se pudo empezar a recopilar datos a mediados de enero.

Otro cambio con respecto al planteamiento original es que solo había planteado un análisis del repositorio por parte del patrón de diseño estratégico, y al final se acabaron desarrollando 12, de los cuales muchos no llegaron a tener un panel en Metabase por revelarse como aspectos que afectaban a una cantidad muy reducida de proyectos.

Hay que destacar que cuando se superó la fase de obtener las métricas de SonarQube, almacenarlas y devolverlas, el tutor de la empresa añadió un nuevo objetivo. Quería crear un *endpoint*, que devolviera esa información en un formato CSV. Esta característica se implantó, pero en las fases finales del proyecto, en concreto cuando se configuraron los paneles de Metabase, se eliminó del mismo, ya que era información redundante.

[39] https://www.atlassian.com/es/software/jira

Un inconveniente que surgió en este desarrollo fue que Jeff tomó la decisión de realizar un ERTE el 31 de marzo de 2020, a causa de la recesión económica causada por el COVID-19. Esta medida tuvo la consecuencia de la rescisión de mi convenio de prácticas.

Esto no afectó a este desarrollo, ya que en ese momento la aplicación llevaba algo más de dos meses desplegada y recogiendo métricas con sus principales funcionalidades ya implementadas. Lo único que se hizo fuera del contexto de la empresa fue acabar de pulir algunos casos de uso en los tests y algunas refactorizaciones que no alteraban la funcionalidad de los procesos.

6. Implantación

Mi aplicación se ejecutará sobre un contenedor Docker, que será iniciado desde un docker-compose, de manera que la misma máquina anfitriona contenga los contenedores de Metabase y el de la aplicación. Esta máquina será una imagen de AWS de tipo EC2[40], que permiten una asignación de recursos computacionales programable, de manera que se pueda ajustar a la demanda del momento.

La decisión de desplegar el proyecto sobre una instancia de AWS fue del departamento de DevOps de Jeff, que son quienes gestionan la infraestructura de los servicios ofertados. Fue decisión mía las características de la imagen Docker y del docker-compose utilizado.

La Figura 17 muestra el `DockerFile` desde donde se ejecutará la imagen. Mi proyecto hace uso de un *plugin* que realiza la construcción de una imagen Docker que contenga el archivo JAR generado por Maven durante su proceso de ciclo de vida, pero esta construcción requiere de este archivo de configuración para su correcta ejecución.

```
1  FROM openjdk:11-jre-slim
2  EXPOSE 8080
3  ARG JAR_FILE
4  ADD  ${JAR_FILE} app.jar
5  ENTRYPOINT ["java","-jar","/app.jar"]
```

Figura 17. Dockerfile

Las indicaciones que recibe son que partirá de la imagen de un contenedor Docker que tenga Java 11 con sistema operativo Linux, indicado en la línea FROM. Utilizará el puerto 8080 para sus comunicaciones con el exterior, lo que se especifica en la línea de EXPOSE. Recibirá como argumento algo llamado JAR_FILE, línea de ARG, el cual copiará en el directorio local de la imagen renombrándolo como app.jar, lo que se indica en la línea 4 y finalmente, que cuando el contenedor arranque ejecutará este archivo desde Java, lo que dejará a mi aplicación en ejecución. Ya que la entrada de ENTRYPOINT específica el comando que va a ejecutar el contenedor tras finalizar el arranque de su sistema operativo.

El docker-compose desplegará tres contenedores, uno con Metabase, otro con un servidor MySQL conectado con el anterior y otro con la imagen con mi aplicación generada con el proceso antes mencionado.

[40] https://aws.amazon.com/es/ec2/

La configuración común se resume en que los contenedores tendrán limitado el uso que pueden hacer de la memoria RAM a 512MB y que cuando la máquina anfitriona se reinicie, ellos se volverán a ejecutar automáticamente, de esa manera se facilita la gestión en caso de fallo o de un apagado por mantenimiento.

Los contenedores de MySQL y Metabase se encuentran enlazados y se ejecutará antes el de la base de datos. Dado que Metabase necesita este servicio activo para funcionar, este orden de ejecución supone un requisito para su correcto despliegue.

El contenedor de la base de datos tiene asociado un volumen donde se persistirá la configuración de los paneles de Metabase, para que en caso de reiniciarse el contenedor, no se pierda la estructura de Metabase.

La información sensible como las credenciales de acceso a diferentes servicios y las URLs que permiten el acceso a los mismos se encuentran aquí, como variables de entorno. De esta manera estas variables quedan almacenadas dentro del sistema operativo del contenedor y no sean accesibles desde el exterior.

La base de datos donde la aplicación *platform-metrics-projects-service* almacenará las métricas está fuera de este sistema. Durante el desarrollo no fue así; se utilizó la misma que usaba Metabase. Pero cuando se desplegó en este sistema, se migraron los datos a una instancia de AWS de la que se realizan copias de seguridad periódicas.

Por tanto, el despliegue consistió en generar la imagen con Maven desde el ordenador que se utilizó para el desarrollo para después extraerla del repositorio local de Docker y llevarla hasta la instancia de AWS mediante una conexión SSH[41] junto con el `docker-compose.yml`.

Tras ejecutar el docker-compose desde la terminal, la aplicación *platform-metrics-projects-service* quedaría preparada para ejecutar el proceso de recolección de datos a la hora programada, así como recibir peticiones a partir de los *endpoints* expuestos.

[41] https://www.ssh.com/

7. Pruebas

Para validar el funcionamiento del proyecto se realizaron tests unitarios utilizando JUnit y Mockito, y tests de integración con MockMvc para comprobar que los puntos de acceso de la aplicación ofrecen lo necesario para la interacción con las aplicaciones.

Recordar que la ejecución de los tests es diferente de la ejecución del proyecto, ya que para los primeros se levanta un contexto mínimo de ejecución y en ningún caso el proyecto entero, de esa manera se puede comprobar aisladamente el comportamiento de los métodos.

No se llevó a cabo el test de integración con la base de datos, ya que esta se encuentra fuera del sistema y la gestión de su correcto funcionamiento no depende de este proyecto.

Las clases Java para las que se construyeron tests fueron las factorías y los servicios, ya que son las que tienen el comportamiento de la aplicación. El procedimiento a seguir fue generar una clase de test para cada clase de estos paquetes y para cada método generar varios métodos de test, donde cada uno de estos comprobaría uno de los comportamientos del método a validar.

Para cada método se validó su funcionamiento correcto, suponiendo entradas de datos válidas y buscando una ejecución sin excepciones. Posteriormente se validaron los casos que no coinciden con el comportamiento prototípico, situaciones en las que faltaba algún campo, que se recibe un objeto vacío como parámetro de entrada o que genera algún tipo de excepción. Esto quiere decir que para los métodos menos ambiciosos se han generado mínimo entre dos o tres casos de test.

Un ejemplo es el caso de la factoría de los objetos de Jenkins. Cabe recordar que la funcionalidad de esta clase, que solo tiene un método, es instanciar un objeto del modelo de datos de mi sistema a partir de los datos proporcionados por Jenkins. En este caso se desarrollaron seis tests, cubriendo los casos de recibir un objeto con los datos correctos, recibir un objeto sin fecha o sin nombre de proyecto, así como un test en caso de que el nombre del proyecto estuviera completo, otro para cuando se encontrara incompleto y finalmente, un test para cuando se reciba una lista de objetos con uno correcto, otro sin fecha y otro sin nombre.

Para las clases con métodos que dependen de otras clases para su ejecución se utilizó la librería Mockito, ya que permite simular el comportamiento de las clases de las que se depende. Esto es posible por el desacoplamiento que ofrecen las interfaces de Java, al separar la declaración del método de su implementación, es posible cambiar esa última por otra.

Las particularidades que lo diferencian del ejemplo anterior es que la generación de *mocks*, como se denomina a las clases generadas por Mockito a partir de una interfaz, requiere un contexto de ejecución un poco más complejo. Esto se resuelve con una anotación en los test que vayan a hacer uso de esta tecnología.

Más allá de esto, el procedimiento es el mismo que el explicado con anterioridad, salvo que cuando la ejecución del método sobre el que se realiza el test realiza una llamada a un método de una clase ajena, se le debe de programar el comportamiento indicándole qué debe devolver cuando reciba determinados parámetros de entrada.

Se ha utilizado Mockito especialmente en los servicios, pues son las clases que gestionan el orden de llamadas a los diferentes componentes de la aplicación.

Los tests de integración validan el correcto funcionamiento de los *endpoints* que expone este proyecto. Aumentan la complejidad del contexto de ejecución ya que se requiere parte del comportamiento de HTTP para su funcionamiento.

Para su programación se debe señalar cual es la clase cuyo comportamiento se va a simular, que en este caso es aquella que recibe las peticiones web, y a partir de ahí se pueden generar objetos de la clase MockMvc que comprueban que cuando se recibe una petición en esa URL, está correctamente autentificada y devuelve en formato JSON los valores esperados.

Tras la validación de la aplicación se generó la documentación sobre el proyecto, ofreciendo un archivo HTML como el que se muestra en la Figura 18, donde se detalla el propósito general de la aplicación, se describen las clases de su modelo de datos (apartado *Domain* del menú lateral), y se documentan los puntos de acceso tal y como aparece en la Figura 18. Indicando el nombre de la fuente de datos externa, el objeto que va a almacenar, un ejemplo de consulta y otro de respuesta. Estos ejemplos se generan a partir de los tests de integración, que permiten la construcción de fragmentos de texto inyectables en archivos de formato ASCIIDOC[42]. La documentación se encuentra en inglés por exigencia de la empresa, ya que en ella trabaja gente de diferentes nacionalidades.

[42] http://asciidoc.org/

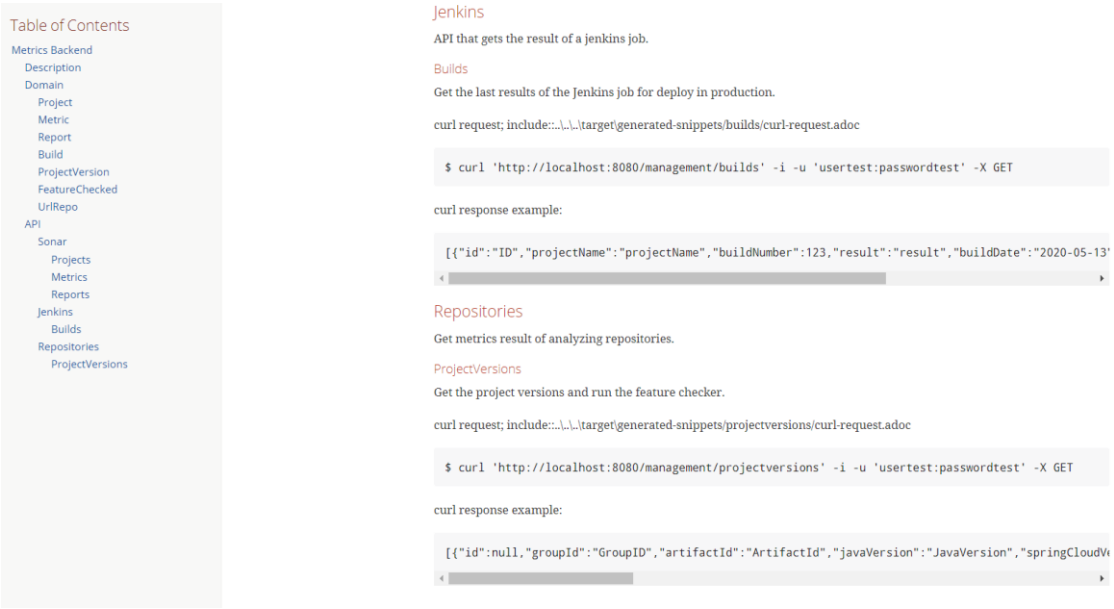

Figura 18. Documentación

En la Figura 19 se muestra el resultado de la cobertura de código de los tests generados. La cobertura total es del 68%, pero este valor no es adecuado, ya que considera que las clases del modelo de datos no están cubiertas por los tests. Sin embargo esas clases no se pretenden validar, ya que no contienen lógica y sus métodos son *getters* y *setters*, así como implementaciones de *toString* que no tienen particular interés.

Las clases relevantes son las factorías, los servicios y la clase ManagementController, donde se ubica la lógica de los *endpoints*. Salvo en los servicios la cobertura es del 100%. Los servicios no están del todo cubiertos. Se aprecia que los que se encuentran en el paquete *gitrepositories*, no tienen test, ya que estos métodos ejecutan las llamadas a los métodos de la librería de JGit, no solo se confía en el buen funcionamiento de esta, sino que, dadas las limitaciones del contexto de ejecución de los tests, estos tests serían excesivamente complejos. Sencillamente no compensa el esfuerzo de validar el funcionamiento de una librería que se sabe que funciona perfectamente.

Otro paquete que no está totalmente cubierto es el que contiene los *feature checker*; esas comprobaciones pequeñas ejecutadas sobre los repositorios. El motivo es su carácter temporal. Como se ha comentado con anterioridad, su funcionalidad era comprobar el estado de pequeños aspectos que se corregirían o no. Una vez acaba su acción carecen de relevancia y se eliminarán. Sin embargo, algunos fueron testeados por iniciativa propia con la finalidad de mejorar mi práctica en el uso de los tests. Dado el cese de las prácticas por el ERTE que

realizó la empresa, no se testearon todas las clases, siendo un problema menor, porque las clases más importantes están ya correctamente validadas de acuerdo con lo requerido por la empresa. Sencillamente los tests de los *feature checker* era algo que no se exigía y que me tomé la libertad de realizar unos cuantos que me resultaron interesantes una vez finalizadas las prácticas en Jeff.

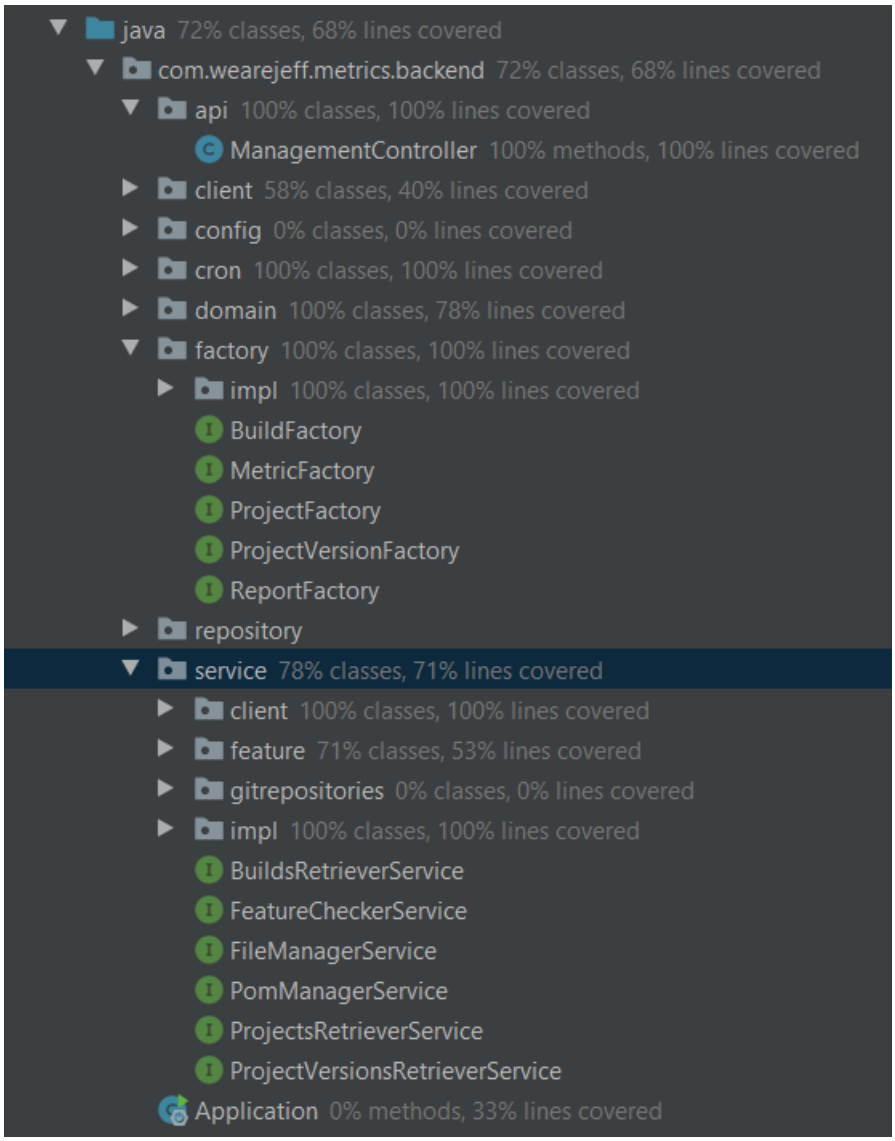

Figura 19. Cobertura de los test

Por último, se va a mostrar los resultados del análisis del servidor SonarQube de Jeff sobre mi proyecto en la Figura 20. Nótese que la cobertura de este test es menor, ya que no están los tests del paquete *feature*, por los motivos antes mencionados el análisis se hizo sobre la versión del proyecto que tienen ellos, sin los arreglos que he realizado yo por mi cuenta.

Estos resultados deben interpretarse como orientativos, pues no todas las reglas de SonarQube tienen la misma relevancia y no se persigue un resultado perfecto,

ya que, aunque algunos de sus avisos son muy relevantes y acertados, otros identifican mal la situación en la que se encuentran. Un ejemplo es el problema de seguridad que detecta, sencillamente ve un trozo de una URL que es una ruta de un archivo y señala que es un defecto del código ya que la información sensible debe ubicarse fuera de la aplicación. Esto se ha hecho así: la información de credenciales de acceso a las aplicaciones externas se encuentra en las variables de entorno del sistema operativo, pero esta ruta a una carpeta no se ha considerado información sensible, por lo que no se ha extraído de la aplicación. No solo eso, sino que puede resultar útil para otro programador conocer esta ruta y resulta conveniente ubicarla en un lugar de fácil acceso.

De la misma manera el código duplicado sucede en clases de configuración y debe de estar duplicado para su correcto funcionamiento. Los *code smells*, que muestra a su vez son aspectos menores que no dificultan el mantenimiento de la aplicación. Uno de ellos es que SonarQube considera que lo correcto es que un constructor no tenga más de siete parámetros, y encuentra en su análisis uno con nueve. Pero es necesario que tenga más de los recomendados ya que es una clase construida para recibir los JSON de SonarQube, y estos datos tienen nueve atributos. A la hora del diseño se consideró que todos eran relevantes y no se eliminó ninguno.

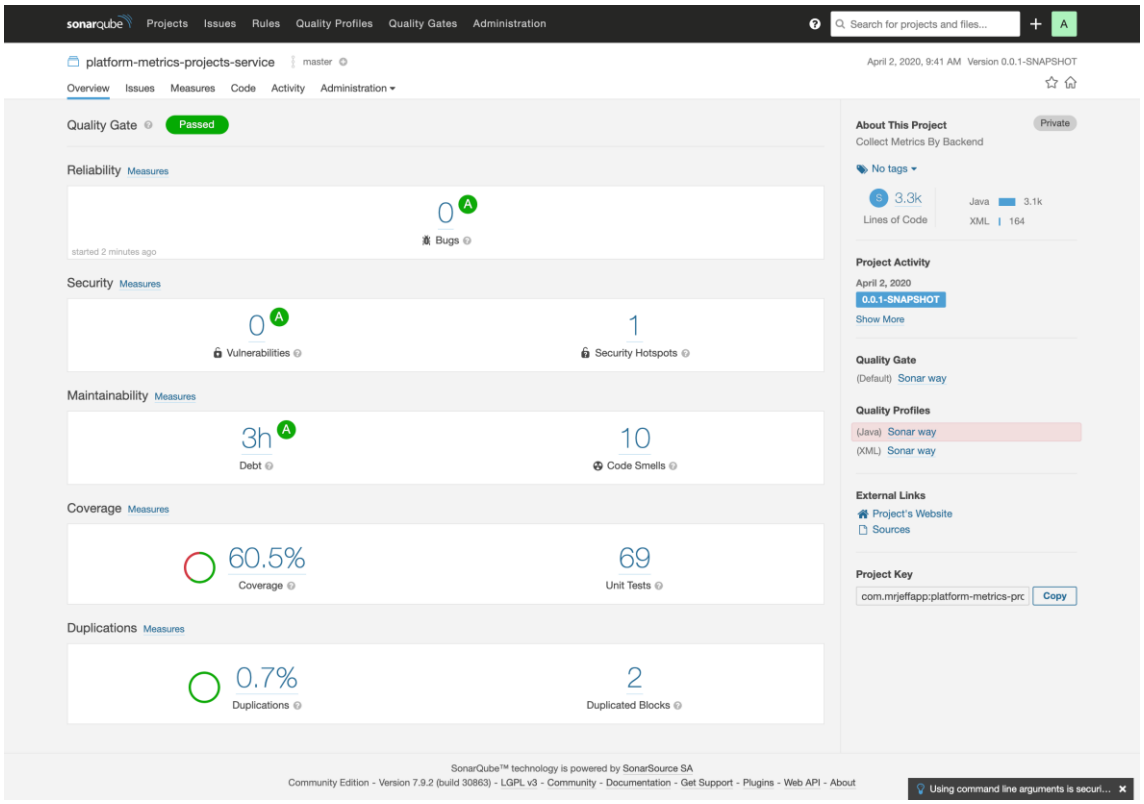

Figura 20. Análisis SonarQube

Un dato que sí que es relevante del análisis de SonarQube es la cantidad de test unitarios, que como se aprecia en la Figura 20 son 69, que son más del doble de los métodos de la aplicación. Esta métrica, junto con la cobertura de los paquetes, sí que es garantía de que las clases se encuentran debidamente validadas.

8. Conclusiones

El avance de la tecnología aporta nuevas maneras de resolver los problemas antiguos. Se ha visto como la implantación de una arquitectura de microservicios aporta soluciones y conlleva la aparición de nuevos inconvenientes.

Este proyecto ha dado solución a uno de ellos. Se puede apreciar a través de los paneles de Metabase que la información sobre el estado tecnológico de esta arquitectura resulta clara, manejable y cuantificable tras la implantación de esta herramienta.

La Figura 21 muestra la evolución del OKR relacionado con la calidad del código, la segunda columna aporta el valor de enero, la siguiente febrero y finalmente marzo. Ya que las primeras versiones de la aplicación se desplegaron a mediados de enero, se puede apreciar que la visualización del estado del *backend* ha permitido formular objetivos cuantificables cuya progresión es apreciable, así como que este valor se ha ido mejorando progresivamente.

4: Quality			
4.1: Improve sonar metrics in 5%	50%	75%	75%

Figura 21.OKR de calidad

Este proyecto no queda cerrado del todo, ya que gracias a la implementación del diseño estratégico en los *feature checker*, deja la herramienta abierta a nuevas implantaciones que puedan ser de interés de cara a la formulación de nuevos OKRs relacionados con la calidad del software.

Mis objetivos personales también se encuentran ampliamente satisfechos, ya que he podido profundizar en el conocimiento de Spring, SonarQube y Jenkins hasta el punto de hallarme dotado y desenvuelto en su uso, gracias a desarrollar este proyecto. Los conocimientos adquiridos durante la titulación han cristalizado y me han permitido el aprendizaje de nuevas destrezas como queda demostrado en este proyecto.

8.1 Relación del trabajo desarrollado con los estudios cursados

Para la realización de este proyecto he utilizado una serie de conceptos en los que ya había sido introducido mientras cursaba la titulación.

La asignatura de Tecnologías de los sistemas de información en la red aportó el marco teórico de los sistemas centralizados y distribuidos que ha sido la base para entender los conceptos de arquitecturas monolíticas y basadas en microservicios. También en esa asignatura aprendí a trabajar con los contenedores Docker y finalmente el modelo de *Software as a Service*, que me permitió entender el modelo de negocio de Jeff.

La gestión de las comunicaciones de las APIs Rest no supuso ningún problema ya que la asignatura de Integración de Aplicaciones destina gran parte de su temario al uso de esta tecnología.

Quero mencionar las aportaciones de Ingeniería del Software, donde se introdujo el concepto de las capas de la aplicación para organizar los proyectos, ya que en este proyecto se incluyeron las estudiadas y se añadieron algunas más, como por ejemplo la capa de las factorías. Otro aporte de la asignatura fueron los conceptos de calidad del código, así como las validaciones del código a través de test. Estos han sido ampliamente utilizados en este proyecto tanto para entender el funcionamiento de SonarQube como para realizar los test adecuados sobre mis clases. He de destacar también que fue en esa asignatura donde se nos introdujo a trabajar con repositorios y con GitHub por primera vez en la carrera.

Tanto de la asignatura Bases de datos como de Tecnología de bases de datos tomé los conocimientos necesarios para gestionar la base de datos expuesta en este documento.

La utilización de los *streams* de Java requería tener ciertas nociones sobre computación paralela. Es cierto que la librería no exige su uso para utilizarla, pero es útil saber cómo funciona internamente para entender algunos fallos o problemas que puedan surgir en su implementación. Fue en las asignaturas de Concurrencia y sistemas distribuidos y en Computación paralela donde estos conceptos se desarrollaron ampliamente.

Quiero destacar que los conocimientos adquiridos sobre el protocolo TCP/IP me facilitaron enormemente la gestión de las comunicaciones entre contenedores, así como con las API Rest. Este protocolo se desarrolla en las asignaturas de Redes de computadores, Redes de área Local, Redes corporativas y Sistemas y servicios

en red. Sus contenidos me sirvieron para interpretar los mensajes de error, así como para entender qué estaba fallando cuando las cosas no sucedían como yo esperaba.

9. Trabajos futuros

Como se ha expuesto, *platform-metrics-projects-service* resuelve los problemas de visualización de un contexto tecnológico concreto, lo que agiliza la gestión del *backend*, dejando en manos del departamento el llevar a cabo las modificaciones que pueda decidir el responsable que recibe la información.

Sería interesante como proyectos futuros el desarrollo de una aplicación que permita llevar a cambios en los proyectos.

Las tecnologías que utiliza este proyecto para acceder al `pom.xml`, también se pueden utilizar para modificarlo y generar *commits* o fusiones con la rama principal del repositorio. Por tanto, es posible automatizar la modificación de ficheros.

El problema al que se enfrentaría este proyecto es que esas modificaciones hagan fallar otras funcionalidades de la aplicación sobre la que se ha ejecutado el cambio. Sin embargo, una posible solución puede ser que, tras llevar a cabo el cambio, Jenkins ejecute los tests del proyecto para comprobar que todo sigue funcionando como se espera y que en caso de fallo, notifique a los responsables de ese proyecto para que lo revisen.

Como este proceso se haría a través de GitHub, quedaría registrado en el historial de cambios del proyecto que es lo que ha sido modificado por parte de los revisores.

Es cierto que pretender corregir los posibles defectos señalados por las métricas de SonarQube puede resultar demasiado ambicioso, pero no lo serían tanto los cambios sobre versiones de determinadas librerías o incluso la modificación de algunos procesos de Jenkins.

Este proyecto facilitaría aún más la gestión de la arquitectura de microservicios, ya que no habría que hacer un cambio en la configuración en 80 microservicios de uno en uno, sino que se lanzaría a todos de golpe y se revisarían solo aquellos que han fallado.

Bibliografía

Bohem, B. (1988). A Spiral Model of Software Development and Enhacement. *IEEE Computer, 21*(5), 61-72.

Cruz, E. d. (2020, Abril 15). *Diagrama de Gantt breve*. Retrieved from https://github.com/ErickSilver/excels/blob/master/Diag%20Grantt%20Corto.xlsx

Cruz, E. d. (2020, Abril 15). *Diagrama de Gantt Extendido*. Retrieved from https://github.com/ErickSilver/excels/blob/master/Diagrama%20de%20Gant%20Extendido.xlsx

Eric Freeman, E. F. (2004). Head First Design Patterns. Sebastopol: O'Reilly. doi:ISBN 0596007124

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, California, EEUU: University of California. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fowler, M. (2006, Mayo 01). *Continuous Integration*. Retrieved from https://www.martinfowler.com/articles/continuousIntegration.html

Fowler, M. (2014, Marzo 24). *https://martinfowler.com/*. Retrieved from https://martinfowler.com/articles/microservices.html

Gaudin, O. (2013). *Continuous Inspection: A Paradigm Shift in Software Quality Management.* Ginebra, Suiza: SonarSource. Retrieved from https://www.sonarsource.com/resources/white-papers/continuos-inspection.html

Guru, R. (2020, Abril 13). *Refactoring Guru*. Retrieved from https://refactoring.guru/design-patterns/strategy

Hernández, P. B. (2017). *Automatización de despliegues mediante VMware, Jenkins y Robot Framework.* Retrieved from http://hdl.handle.net/10251/88614

Jeff. (n.d.). *We are Jeff*. Retrieved from https://wearejeff.com/

Leach, P., Mealling, M., & Salz, R. (2005). *A Universally Unique IDentifier (UUID) URN Namespace.* (I. E. Force, Ed.) doi:10.17487/RFC4122

Martin, R. C. (2015). *Design Principles and Design Patterns.*

Morant Navarro, J. (2016). *iMediador*. Retrieved from http://hdl.handle.net/10251/71538.

Netflix. (2018, Noviembre 19). *Github*. Retrieved from https://github.com/Netflix/Hystrix

Pérez Andrés, S. (2019). *Red social temática para la clasificación de libros, películas y videojuegos.* Retrieved from http://hdl.handle.net/10251/125732

Royce, W. (1970). Managing the Developlment of Large Software Systema. *Proceedings of IEEE WESCON, 26 (August)*, 1-9.

Taberner Aguas, B. (2015). *Informatización de una PYME aplicando una metodología ágil: un caso real.* Retrieved from http://hdl.handle.net/10251/50364.

Todea, V. (2016). *Diseño e implementación de un sistema de entrega continua para aplicaciones web sobre contenedores Docker.* Retrieved from http://hdl.handle.net/10251/71386

Tortosa Calabuig, M. (2019). *Diseñando una fábrica del Futuro con la Internet de las Cosas. Una experiencia práctica.* Retrieved from http://hdl.handle.net/10251/128454

Watson, M. (2017, Noviembre 28). *Stackify*. Retrieved Abril 18, 2020, from https://stackify.com/premature-optimization-evil/

10. Glosario

1. Ant: herramienta software que permite la secuenciación de tareas repetitivas, usada habitualmente para desarrollar los procesos de compilación de un proyecto Java.
2. API: acrónimo de *Application Programming Interface*; es un punto de comunicación entre diferentes componentes software.
3. API REST: un tipo de API concreta dentro del protocolo HTTP a partir de la cual los verbos del estándar tienen significado en la aplicación que las recibe más allá del necesario para HTTP.
4. ASCIIDOC: formato de documentos de texto utilizado para redactar documentación compatible con HTML y PDF entre otros.
5. AWS: acrónimo de Amazon Web Services; una colección de servicios ofertada por Amazon que, en su conjunto, permiten la computación en la nube.
6. Backend: división de las tareas de programación, en concreto corresponde a las actividades relacionadas con la gestión de las comunicaciones entre diferentes componentes.
7. Code Smells: traducido como código maloliente, es una categoría de métricas de SonarQube que se refiere a un código que resulta poco legible, que utiliza elementos en desuso o que resuelve un problema sencillo con demasiada complejidad.
8. Docker: herramienta de gestión de contenedores.
9. Docker-compose: tecnología que permite la ejecución de varios contenedores en un mismo anfitrión y la configuración de sus comunicaciones desde un único fichero denominado `docker-compose.yml`.
10. Endpoint: es un punto de acceso a un sistema, en el contexto de este trabajo se refiere a una URL con determinados parámetros de entrada en el contexto de una API Rest.
11. Flyway: sistema de control de versiones de la base de datos gestionado por la propia aplicación.
12. Frontend: división de las tareas de programación, se corresponde con actividades relacionadas con la estructura visual del contenido web.
13. Gateway: puerta de enlace; en el contexto de los microservicios sería el encargado de recibir todas las peticiones y redirigirlas al microservicio que deba empezar a procesarlas.
14. GitHub: sistema de control de versiones construido sobre Git. Ofrece una interfaz web sobre las acciones que ofrece Git.

15. Gradle: herramienta de compilación y generación de ficheros ejecutables a partir de un proyecto en Java o en Kotlin.
16. Groovy: lenguaje de programación construido sobre Java que ofrece instrucciones ejecutables por línea de comandos, sin necesidad de hacerse sobre una máquina virtual. Pretende ser una aproximación menos orientada a objetos y más funcional
17. Hibernate: implementación de JPA que gestiona las comunicaciones de una aplicación con la base de datos.
18. IDE: acrónimo de *Integrated Development Environment*; es una aplicación de escritorio que tiene los elementos necesarios para programar.
19. JAR: fichero de código Java ejecutable.
20. JGit: librería sobre Java que permite ejecutar comandos de Git sobre los repositorios de GitHub, como pueden ser el clonado del repositorio o la fusión de diferentes ramas.
21. JPA: acrónimo de Java Persistent API, modelo de Java para gestionar bases de datos relacionados desde el paradigma de orientación a objetos.
22. JSON: acrónimo de *JavaScript Object Notation*, es un formato estandarizado que permite distinguir un objeto de sus propiedades a partir del uso de llaves y corchetes.
23. Kotlin: lenguaje de programación tipado de aparición relativamente reciente, cuya ventaja es que permite incluir fragmentos de código Java dentro de un proyecto Kotlin y que puede ser compilado a código fuente JavaScript.
24. Kubernetes: es un sistema para la automatización del despliegue de contenedores y la gestión de los mismos.
25. Maven: herramienta de compilación y generación de ficheros ejecutables a partir de un proyecto en Java.
26. Microservicio: de acuerdo con la arquitectura basada en microservicios, es un componente de una aplicación que tiene un comportamiento autónomo al resto del sistema.
27. MySQL: gestor de bases de datos relacionales
28. OKR: acrónimo de Objective and Key Results, objetivos planteados como resultados medibles con finalidad de ser alcanzados en un plazo de tiempo concreto. Es parte de una metodología empresarial empleada por Google.
29. `Pom.xml`: archivo propio de los proyectos Maven donde se describe el proyecto software y sus dependencias con otras librerías. Su nombre se deriva de las siglas de *Project Object Model*.
30. Reflexión de Java: propiedad de las clases de Java a partir de la cuál se puede acceder a los metadatos de la clase, tales como su nombre, sus atributos o los métodos que contiene.

31. Repositorio: en el contexto de GitHub, un repositorio envuelve un proyecto y registra metadatos ajenos al proyecto tales como los *commits* realizados, las ramas que tiene, número de colaboradores, etc.
32. SonarQube: herramienta para el análisis del código fuente.
33. Spring: *framework* en Java para el desarrollo de aplicaciones web.
34. WAR: acrónimo de *Web Application Archive*, contiene código Java compilado interpretable y ejecutable por un servidor web.
35. XML: acrónimo de *Extensible Markup Language*, formato para la comunicación entre dispositivos que permite distinguir entre el objeto y sus propiedades a través de un sistema de marcas, que permiten distinguir entre el atributo y su valor.