UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN – MENCIÓN EN INGENIERÍA DE SISTEMAS DE TELECOMUNICACIÓN

Framework de gestión de consumo de servicios

Autora:

Henar García Sánchez

Tutora:

Míriam Antón Rodríguez

Valladolid, 16 de Noviembre de 2016

TÍTULO:	***Framework* de gestión de consumo de servicios**
AUTORA:	**Henar García Sánchez**
TUTORA:	**Míriam Antón Rodríguez**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTA: **Míriam Antón Rodríguez**

VOCAL: **Mario Martínez Zarzuela**

SECRETARIO: **David González Ortega**

SUPLENTE: **Francisco Javier Díaz Pernas**

SUPLENTE: **Mª Ángeles Pérez Juárez**

FECHA:	**16 de Noviembre de 2016**
CALIFICACIÓN:	

RESUMEN

En la sociedad actual, el uso de las tecnologías es algo que forma parte de nuestra vida cotidiana. Para usar la mayoría de las tecnologías actuales, estas tecnologías poseen aplicaciones para hacer más fácil su uso a los usuarios, y estas aplicaciones normalmente necesitan conectarse con otras.

El objetivo de este TFG surge en este punto, en la comunicación de unas aplicaciones con otras. A través de un *ESB* (*Enterprise Service Bus*) estas comunicaciones se pueden realizar de forma simple, abstrayendo cada una de las partes de la aplicación. En estos *ESB* se pueden realizar muchas funciones, la aplicación desarrollada en este TFG va a comprobar que no se produzcan errores en esta comunicación a través del *ESB* y en caso de que se produzca se realizará una política de errores, previamente configurada.

La aplicación de este TFG va a permitir el envió de distintos tipos de peticiones, síncronas o asíncronas, y dependiendo de este tipo de peticiones se realizarán unas acciones u otras. Además de esto, la aplicación posee un servicio de auditoría para almacenar la información de determinados puntos clave de cada servicio y de esta forma llevar un control del funcionamiento de las distintas peticiones.

Palabras clave: *ESB*, *framework*, JMS*, MongoDB*, *Mule ESB*, servicios *Web*, política de errores, *JMS*, *ActiveMQ*

ABSTRACT

In current society, the use of technologies is something that is part of our daily live. To use the majority of actual technologies, these technologies have applications to do easier their use and these applications usually need to connect with others applications.

The goal of this end-of-degree project arises at this point, in the communication between applications. Through an ESB (*Enterprise Service Bus*), these communications can be performed simply abstracting each of the parts of the application. In these ESB, can execute many functions, the application developed in this end-of-degree project verify that no errors occur in this communication through the ESB and if an error occurs, the application will carry out an error policy pre-configured.

The application of this end-of-degree project will allow sending different types of requests, synchronous or asynchronous and will do an action depending on such request. In addition, the application owns an audit service to store information for certain key points in each service and in this way, have an operation control of different petitions.

Keywords: *ESB, framework, JMS, MongoDB, Mule ESB, servicios Web, política de errores, JMS, ActiveMQ*

AGRADECIMIENTOS

A mis padres, por todo su apoyo y la confianza depositada en mí.

A Javi, por su infinita paciencia y apoyo, por estar a mi lado durante esta etapa y no dejar que me rinda.

A Míriam, por la ayuda recibida en el desarrollo de este proyecto.

A Diego, por toda su ayuda y paciencia, por todo lo que me ha enseñado y por ser mi guía en el desarrollo de este proyecto.

ÍNDICE DE CONTENIDOS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE INSTRUCCIONES

CAPÍTULO 1

1. INTRODUCCIÓN

La historia del desarrollo *software* viene muy ligada a la evolución de la electrónica, y con ella a la de los ordenadores, ya que unos son necesarios para la evolución de los otros. A continuación, se hará una introducción a la historia del software, explicando brevemente algunos de los momentos de la evolución de los ordenadores.

La historia del desarrollo software se remonta al siglo XIX con Ada Byron, alias Lady Lovelace, quien fue la primera programadora de la historia. Ada escribió el que se considera el primer algoritmo destinado a ser procesado por una máquina. En este algoritmo se describieron los pasos que la permitían realizar los cálculos necesarios para obtener los valores de los números de Bernoulli. Este algoritmo fue diseñado para la maquina analítica Babbage, creada por Charles Babbage, el matemático británico que estableció los pilares fundamentales del desarrollo de la computación, sin embargo, esta máquina nunca llegó a funcionar correctamente.

En ese mismo siglo, unos años después, el famoso matemático George Boole demostró la relación entre las matemáticas y la lógica, algo que hasta el momento se relacionaba con la filosofía, creando la conocida álgebra de Boole. Sin embargo, el álgebra de Boole no se empezó a usar hasta casi 100 años después (Riha Linik, 2016).

En 1941 Konrad Zuse creó el ordenador Z3, la primera máquina programable y completamente automática.

Unos años después, en 1945 John Von Neumann desarrolló la técnica de programa compartido, la que indica que el hardware de una máquina debe ser simple y no necesita ser modificado por cada programa, sino que se deben usar instrucciones complejas para calcular ese *hardware*, permitiendo ser reprogramado más rápidamente. Un año más tarde la universidad de Pensilvania creo el ENIAC, la primera computadora digital electrónica (Peddie, 2013).

Durante los siguientes años, el software y los lenguajes de programación no pararon de evolucionar. Para tener una mejor visión de cómo ha ido esta evolución se comienza a dividir el tiempo en eras del desarrollo, cada una de las cuales posee su característica relevante en esta historia. Las eras de la historia del software son (Bhushan Agarwal & Prakash Tayal, 2008):

Primera era (1950-1965)

- Surge por primera vez el termino ingeniería del *software*.
- Se trabaja con la idea de "codificar y corregir"
- No existía un planteamiento previo.
- No existía documentación.
- Nace el uso del lenguaje ensamblador para la programación en ordenadores.
- El desarrollo es a base de prueba y error.
- Se crean los lenguajes FORTRAN y COBOL.

Segunda era (1965-1972)

- Se buscar simplificar código.
- Aparece la multiprogramación y los sistemas multiusuarios.
- Los sistemas de tiempo real apoyan la toma de decisiones.
- Aparece el *software* como producto.
- Se inicia la crisis del *software*.
- Se buscan procedimientos para el desarrollo del *software*.
- Se crea C y Pascal.

Tercera era (1972-1989)

- Aparece un nuevo concepto, el de los sistemas distribuidos.
- Se añade complejidad a los sistemas de información.
- Aparecen redes de área local y global y comunicadores digitales.
- Se amplía el uso de los microprocesadores.
- Se crea C++

Cuarta era (1989 – 1999)

- Hay un impacto colectivo de *software*.
- Aparecen redes de información y tecnologías orientadas a objetos.
- Aparecen redes neuronales, sistemas expertos y software de inteligencia artificial.
- La información adquiere más valor dentro de las organizaciones.
- Se crea Python, HTML, Java y PHP.

A día de hoy podemos decir que el desarrollo *software* no ha parado de evolucionar, al contrario, lo hace a pasos agigantados. Se puede decir que algunas de las características más importantes de la era actual son la reutilización de software y el desarrollo ágil de los proyectos.

Gracias a muchos de los innumerables *software* que existen en la actualidad, este no es el único campo que ha evolucionado, ya que los *software* se han introducido poco a poco en toda nuestra vida. Si nos ponemos a pensar usamos *software* para todo, ya sea para hablar con familiares o amigos como para trabajar o estudiar. Esta dependencia que poseemos hace que el *software* sea tan importante hoy en día.

Debido a esto, son muchas las empresas que dedican su actividad al desarrollo *software* y muchas más las que contratan sus servicios para mejorar los *software* actuales o crear otros nuevos.

1.1 MOTIVACIÓN Y OBJETIVOS

La motivación de cualquier aplicación es que se use y poderla sacar el máximo partido, ya sea vendiéndola o dándola el mayor uso posible. La aplicación desarrollada en este TFG no es menos, ya que al realizar el TFG en unas prácticas de empresa se intenta que la aplicación tenga utilidad futura.

Una de las cosas que más hay que tener en cuenta en cuenta y estudiar en cualquier tipo de software son los errores que se puedan producir en este. Es muy común que los desarrolladores incluyan en la planificación del desarrollo una parte de pruebas para minimizar los errores existentes. También es cierto, que hay trabajadores cuya función es esta, encontrar errores, para ello prueban el *software* e intentan encontrar bugs o errores que solucionar antes de que la aplicación llegue al cliente final. Sin embargo, muchas veces los errores que se producen en el *software* no son directamente culpa de estos, sino que son debidos a que la conexión a una base de datos esta caída, el dispositivo desde el que se ejecuta la aplicación no tiene internet, etc.

Con lo que en este punto surge la motivación para realizar este TFG. Una aplicación que se encargue de gestionar este tipo de errores y reintentar las peticiones que los generaron.

Como se ha dicho anteriormente este TFG se ha realizado en unas prácticas de empresa, en las cuales he visto cómo funcionan los sistemas *middleware* y la importancia de estos en la comunicación entre distintas aplicaciones.

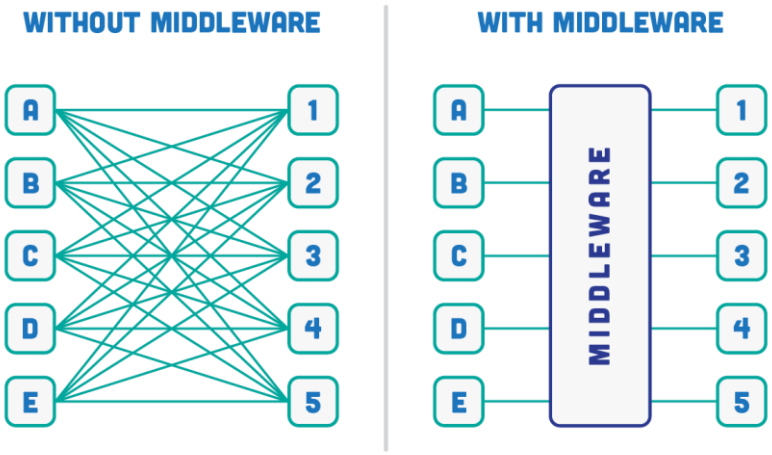

Ilustración 1: Middleware

Después de tener claro un objetivo y conocer la utilidad e importancia de los sistemas *middleware*, se decidió usar este tipo de sistemas para el desarrollo del TFG, ya que de esta manera sería mucho más fácil poder controlar los errores en caso de que estos se produzcan sin necesidad de que el cliente tenga constancia de ellos.

1.2 ESTRUCTURA DE LA MEMORIA

Este documento se ha estructurado en seis capítulos, a través de estos capítulos se pretende detallar algunas de las características más importantes del desarrollo de este TFG.

En el primer capítulo, se hace una introducción del tema que se trata, el desarrollo *software,* explicando la historia de este desarrollo a través de los años. A continuación, se expone la motivación y los objetivos de los que ha surgido este proyecto y lo han permitido desarrollar.

En el capítulo dos, se hace un estudio de las distintas tecnologías existentes en el mercado, que se pueden usar para este desarrollo. En este estudio se realiza una explicación detallada de cada una de las tecnologías y se exponen las ventajas y desventajas de estas. Finalmente, mediante una serie de criterios, se escoge la tecnología más idónea para este desarrollo, así como la mejor herramienta para estos propósitos de cada una de tecnologías elegidas.

En el tercer capítulo, se realiza una descripción detallada y técnica sobre la aplicación. Se muestran todas las funcionalidades de la aplicación mediante diagramas de flujos y se explica su funcionamiento en detalle. Una vez hecho esto, se hace una descripción detallada de cada uno de los servicios de la aplicación.

En el cuarto capítulo, se hace una descripción funcional de la aplicación, en otras palabras, se realiza un manual de usuario. En este manual de usuario, se explica el funcionamiento de la aplicación y como realizar las distintas peticiones de manera correcta. Debido al carácter de la aplicación, también se explica cómo generar un error para poder ver todas las funcionalidades que contiene dicha aplicación.

En el quinto capítulo, se expone la planificación que se estimó previamente al desarrollo de la aplicación. Esta planificación está separada en cinco fases, cada una de las cuales tiene unas funciones asignadas y unas horas previstas para su desarrollo. Para finalizar la planificación se incluye una tabla de cómo se cumplirá esta planificación en los cinco meses que se ha estimado de desarrollo. A continuación, en función de esta planificación se estima un presupuesto de los costes de la aplicación.

En el sexto y último capítulo, se exponen las conclusiones que se han obtenido durante el desarrollo de este proyecto, tanto conclusiones a nivel técnico como a nivel personal. Y para poner punto y final a esta parte se describen posibles líneas futuras que posee la aplicación para mejorarla.

A continuación, se expone la bibliografía, la cual se ha usado para desarrollar la documentación expuesta en esta memoria. Toda esta bibliografía sigue las normas del formato *APA*.

Por último, se exponen una serie de anexos técnicos. El primero de ellos explica como instalar *Mule ESB*, cómo crear un proyecto en él, una breve descripción sobre la estructura de carpetas y concluye con la explicación de cómo ejecutar un proyecto. A continuación, se explica cómo instalar el sistema de mensajería elegido, como arrancar el servidor y cómo acceder a la ventana de administración de este. Finalmente se explica cómo instalar la base de datos no relacional escogida, cómo levantar el servidor y cómo acceder a los datos de la base de datos de dos formas, mediante comandos y de forma gráfica con la herramienta *Robomongo*.

CAPÍTULO 2

2. ESTUDIO DE LAS DISTINTAS TECNOLOGÍAS

En este capítulo se van a estudiar las diferentes tecnologías disponibles en el mercado para el desarrollo de este proyecto. Debido a la naturaleza de este proyecto no se va a usar una única tecnología. Se va a explicar individualmente cada tipo de tecnología posible y los tipos existentes de estas. Al final de este capítulo, con una serie de criterios, se escogerán las tecnologías que se han considerado idóneas, así como las herramientas para usar dichas tecnologías, en el desarrollo de este proyecto.

2.1 ARQUITECTURAS DE *SOFTWARE*

Una arquitectura de *software* es una estructura o estructuras que comprenden elementos de un *software*, las propiedades visibles de estos elementos y las relaciones entre ellos. En otras palabras, la arquitectura de *software* define el conjunto de componentes de un sistema, las interfaces de comunicación entre ellos, y la manera en que estos componentes se comunican entre sí usando dichas interfaces.

La arquitectura de un sistema *software* se puede comparar con la arquitectura de un edificio. Si no se hace un buen estudio de la estructura y esta acaba mal diseñada, el edificio podría derrumbarse; de la misma manera sucede en un sistema software, si no realiza un estudio previo de este sistema *software* y no se establece una arquitectura correcta, el sistema podría ser poco eficiente o incluso podría no funcionar.

Además de la importancia descrita anteriormente, estos diseños arquitectónicos de *software* pueden ser reutilizados, reduciendo costes y aumentando la calidad de los sistemas futuros (Bass, Clements, & Kazman, 2012).

Una de las arquitecturas más usada es la arquitectura orientada a servicios, la cual se explica a continuación.

2.1.1. ARQUITECTURAS ORIENTADAS SERVICIOS (*SOA*)

Una arquitectura orientada a servicios se puede definir como un paradigma de integración basado en un principio fundamental de diseño, y proporciona servicios interoperables de arquitectura. Abarca sistemas heredados, componentes de *software* y procesamiento de mensajes.

Su función principal es la integración de servicios y el desarrollo de aplicaciones. A pesar de que algunos analistas han declarado que los sistemas basados en *SOA* están muertos, podemos decir que este tipo de arquitectura es más relevante que nunca (Mulesoft, 2015).

Gracias a este tipo de arquitecturas se pueden conseguir infraestructuras agiles y eficientes, ya que los servicios de *SOA* están débilmente acoplados y son altamente interoperables. Para comunicarse entre sí, estos servicios, usan definiciones independientes de las plataformas subyacentes y de los lenguajes de programación, como puede ser *WSDL (Erl, 2007)*.

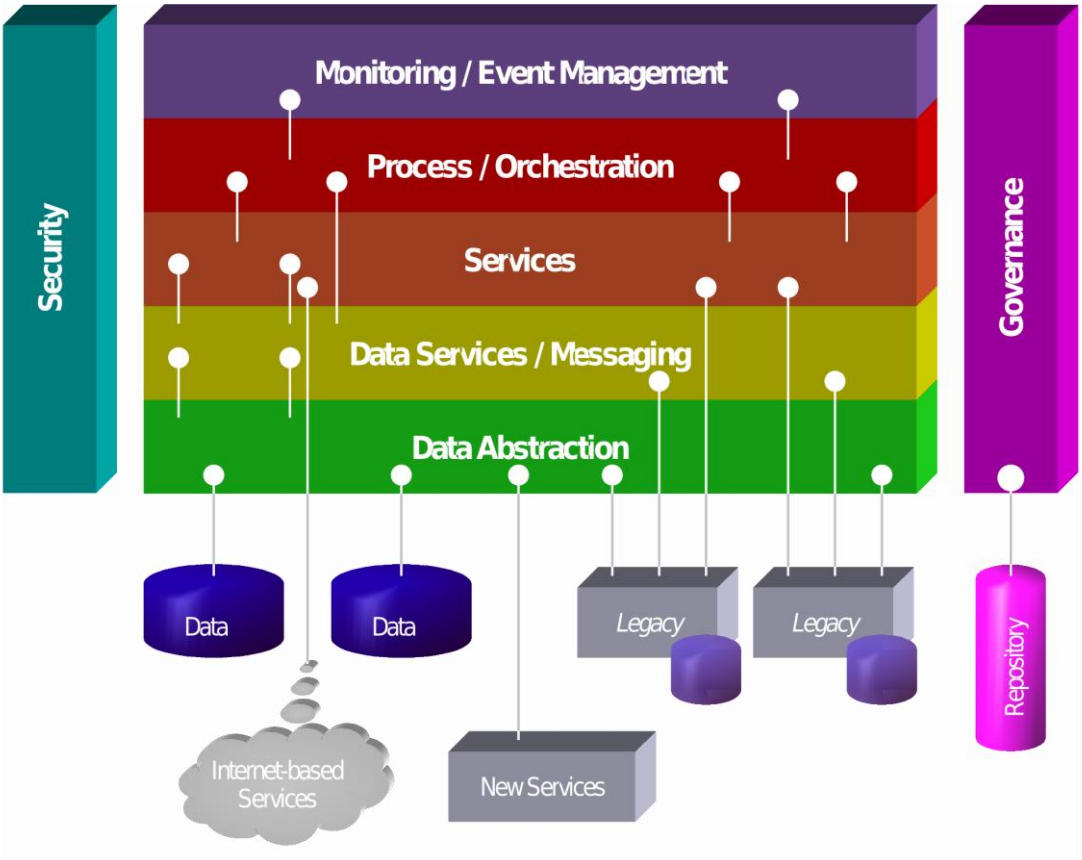

Ilustración 2: Modelo de arquitectura SOA

Las ventajas de usar SOA son (Open group, 2014):

- Mejoran el flujo de información en los servicios.
- Añaden una flexibilidad organizativa, es decir, permite cambiar los servicios software de forma rápida y sencilla.
- Poseen capacidad para exponer la funcionalidad interna, es decir, permiten que el cliente tenga visibilidad de las funcionalidades.
- Permiten la reutilización de los servicios, lo que produce un menor desarrollo y menor coste.
- Poseen una configuración de servicios de mensajería flexible.
- La monitorización de la actividad empresarial se vuelve una fuente valiosa de inteligencia de negocio.
- Se mejora el desempeño de la monitorización, permitiendo detectar ataques de seguridad.
- El control de mensajes posee una política de gestión y de seguridad de estos.
- Permiten la conversión de datos de un formato a otro a través de la asignación automática de campos, con lo que se ahorran costes.
- Permiten confidencialidad de los datos gracias al cifrado de mensajes a través de comprobación criptográfica.
- Permiten una simplificación en la estructura del software, mediante la eliminación de funcionalidades que no estén relacionadas con el software de negocios
- Poseen gran capacidad para adaptarse a diferentes ambientes externos.
- Mejoran la capacidad de administración y seguridad de los procesos.
- Permiten la detección de servicios, con lo que se aumenta el rendimiento y la funcionalidad de estos.
- Permiten la virtualización, mejorando la fiabilidad del sistema a través de la redundancia del funcionamiento de los activos.
- Gracias a la virtualización, se mejora la capacidad de escalar las funciones para satisfacer diferentes niveles de demanda.
- Poseen una implementación basada en modelos, lo que permite desarrollar nuevas funciones de forma rápida.

2.1.1.1. *FRAMEWORKS*

Un *framework* es un entorno de desarrollo que integra diversos componentes para facilitar el desarrollo *software*. Su objetivo principal es facilitar el desarrollo de una aplicación, para ello, usa *APIs*, que incorporan funcionalidades ya desarrolladas y probadas, para integrar en las aplicaciones de manera estandarizada, permitiendo simplificar la implementación de la arquitectura de las aplicaciones.

La mayoría de los *frameworks* contienen las siguientes características:

- Se encargan de manejar las sesiones y las *URLs*, abstrayéndolas del desarrollador.
- Incluyen las herramientas necesarias para conectarse a bases de datos.
- Suelen implementar controladores para la gestión de eventos y las peticiones que se realicen en la aplicación.
- Incluyen mecanismos para la autentificación y el control de acceso.

Algunos ejemplos del uso de estas tecnologías son *Spring* y *Struts* (AcensTechnologies, 2014).

2.1.1.2. *ESB*

Un *ESB* (*Enterprise Service Bus*) es un modelo de arquitectura *software* para gestionar la comunicación entre servicios web. Define un conjunto de reglas y principios para la integración de aplicaciones sobre una infraestructura similar a un bus. La idea básica de arquitectura *ESB* es permitir integrar diferentes aplicaciones poniendo en común un bus de comunicación y de esta manera desacoplar los sistemas entre sí. Gracias a que los sistemas están desacoplados es posible la comunicación entre estos sin necesidad de conocer lo que hay en el bus. A pesar de que las bases del *ESB* están en los *framework*, los *ESB* son mucho más potentes.

Ilustración 3: Bus de comunicación ESB

Algunas de las características de los *ESB* son:

- El concepto de bus permite desacoplar las aplicaciones entre sí.
- Los datos que viajan por el bus tienen un formato canónico, normalmente suele ser *XML*.
- Existe un adaptador entre la aplicación y el bus, el cual es responsable de comunicarse con la aplicación y transformar los datos. Sin embargo, también puede realizar otras actividades de seguridad, monitorización, etc.
- Los *ESB* no guardan el estado de los clientes, el estado se encuentra en los mensajes.

Algunos ejemplos del uso de estas tecnologías son *MuleESB* y *FuseESB* (MuleSoft, 2015).

2.1.1.3. SUITE DE INTEGRACIÓN

Una suite de integración proporciona las herramientas necesarias para las conexiones entre aplicaciones y de estas con la nube, además del desarrollo y exposición de microservicios y la gestión de *APIs*, todo ello en una misma herramienta. Gracias a estas características puede llegar a nuevos mercados y aprovechar las oportunidades de negocio de estos, proporcionando un software fiable, seguro y con mayor flexibilidad y escabilidad que los anteriores. Se puede decir que internamente lleva un *ESB*, ya que al igual que este, define un conjunto de reglas y principios para la integración de aplicaciones (IBM, 2015).

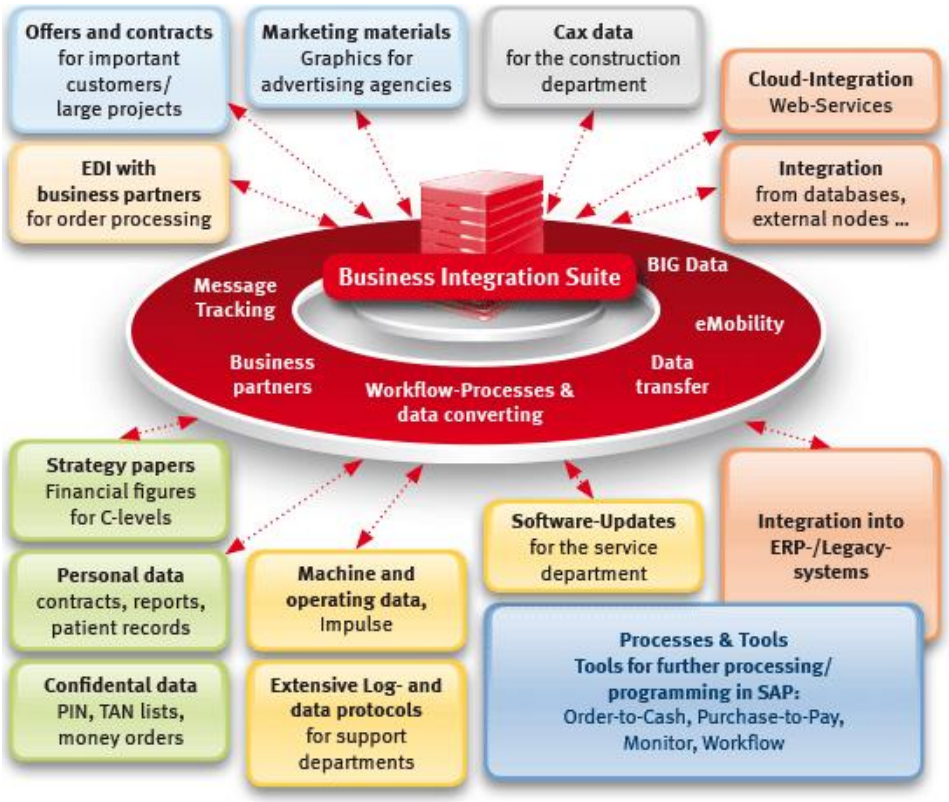

Ilustración 4: Suite de integración

Algunas de las características de las suites de integración son las siguientes (IBM, 2015):

- Permiten integrar de forma segura las aplicaciones, permitiendo optimizar recursos y la productividad en el software como servicio.
- Dan mayor flexibilidad a la hora de implementar las aplicaciones.
- Al usar un ESB como patrón arquitectónico, permiten eliminar las costosas conexiones punto a punto.
- El desarrollo de nuevas APIs es rápido y sencillo.
- Aceleran la creación y gestión de la integración de aplicaciones con el uso de patrones previamente definidos, conectores y herramientas fáciles de usar.
- Mejoran la visibilidad de los datos de negocio, eventos y mensajes de la capa de integración, lo que se traduce en una respuesta más ágil a los cambios.

Algunos ejemplos del uso de estas tecnologías son *webMethods* y *Tibco* (Information Builders, 2014).

2.1.1.4. COMPARACIÓN *FRAMEWORK* Y *ESB*

A continuación, se expone una breve comparación entre los *frameworks* y los *ESB* (Mulesoft, 2015):

- Un *ESB* es una plataforma de integración completa, mientras que un *framework* es un marco de mediación.
- Un *ESB* proporciona funcionalidades que un *framework* es incapaz de proporcionar, por no ser una plataforma completa.
- Un *ESB* mejora la fiabilidad, escabilidad y seguridad de un *framework*.
- Un *ESB* permite una integración más sencilla que un *framework* y permite desacoplar requisitos no funcionales.
- Un ESB posee un nivel de abstracción más alto que un *framework*, de manera que su integración es mejor.
- Un *ESB* reduce coste y complejidad respecto a un *framework*.

2.1.1.5. COMPARACIÓN *ESB* Y SUITE DE INTEGRACIÓN

A continuación, se expone una breve comparación entre los *frameworks* y los *ESB* (IBM, 2015):

- Una suite de integración ofrece todas las funciones de un *ESB* y otras a mayores, como gestión de datos, monitorización o gestión de procesos de negocio.
- Con una suite de integración se puede desarrollar lo mismo que con un *ESB* pero con un solo software.
- Los *softwares* más usados de suite de integración son de pago, mientras que los *ESB* más usados tienen versión *open*.

2.1.1.6. COMPARACIÓN GENERAL

Después de este este estudio, se va a realizar una breve comparación final de estas 3 tecnologías.

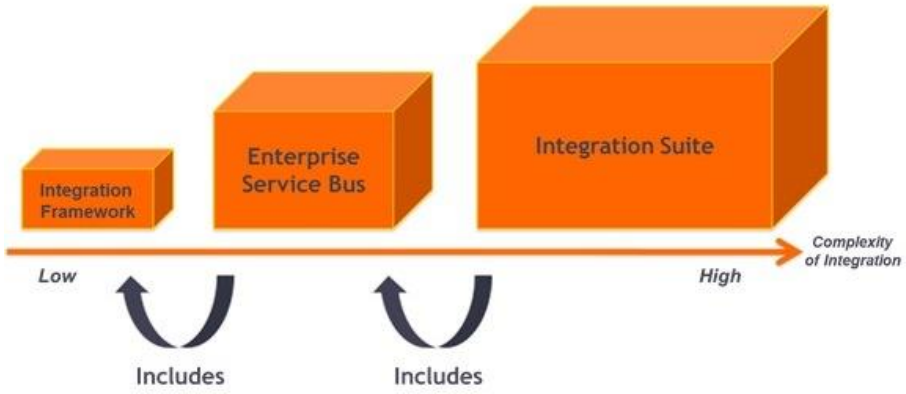

Ilustración 5: Comparación framework, ESB y suite de integración

Como se puede ver en la Ilustración 5, cada una de estas tecnologías tiene un distinto nivel de complejidad. La más compleja, que es el caso de la suite de integración, contiene las características y funcionalidades de los *ESB*, los cuales a su vez, contienen las características y funcionalidades de los *frameworks*. Con lo que podemos concluir que cuanto más compleja es la tecnología más completa es esta, aunque esto no siempre es lo deseado.

2.2 BASES DE DATOS

El concepto de base de datos como hoy en día se conoce nació en los años 60. A pesar de esto, el termino base de datos no fue nuevo en ese momento, ya que el almacenamiento de registros de información se remonta a la antigüedad. Y el almacenamiento de datos siempre ha existido, comenzó con tarjetas perforadas, continuó con cintas magnéticas hasta el almacenamiento en discos duros que se tiene en la actualidad.

No fue hasta 1963, en un congreso en California, cuando se dio la primera definición de este término como un conjunto de información relacionada entre sí, que se encuentra agrupada o estructurada.

En los años 70, el científico Edgar Frank Codd, publicó una serie de reglas para definir los sistemas de datos relacionales y gracias a estas reglas, se creó la empresa que se conoce hoy en día como Oracle. Posteriormente, en los años 80, se desarrolló el lenguaje de consultas de acceso a base de datos, conocido como *SQL*, el cual se convirtió en un estándar para la industria (Fortune, 2014).

A partir de este momento, el desarrollo de las bases de datos viene de la mano de las grandes compañías que dominan el mercado, como IBM, Microsoft y Oracle.

2.2.1 BASES DE DATOS ANALITICAS (*OLAP*)

Las bases de datos analíticas o también llamadas bases de datos estadísticas, son aquellas en las que la información en tiempo real no es afectada, es decir no se hacen operaciones con ella, solo consultas. Esto es debido a que los datos almacenados son datos archivados, normalmente datos históricos utilizados para el análisis, los cuales, son solo de lectura. Este tipo de bases de datos suele ser implementada para mejorar el desempeño de las consultas con grandes volúmenes de información (IBM, 2014; Sol, 2010).

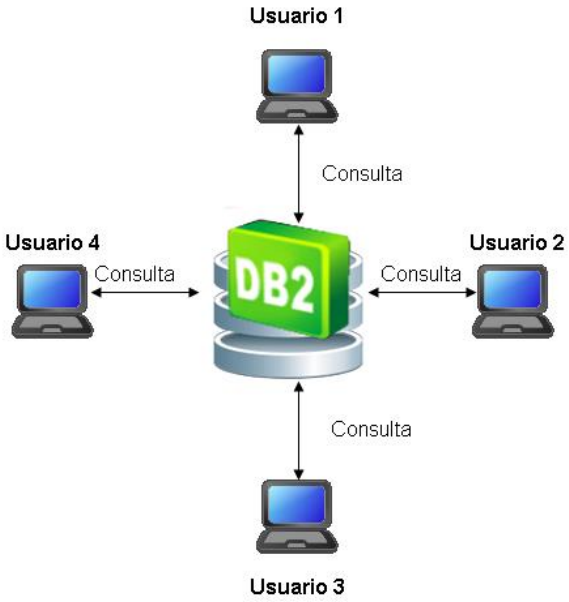

Ilustración 6: Base de datos analítica

2.2.2 BASES DE DATOS OPERACIONALES (*OLTP*)

Las bases de datos operacionales o también llamadas bases de datos dinámicas, son aquellas en las que la información en tiempo real es afectada, es decir, se hacen operaciones de insertar, modificar y borrar datos además de consultarlos. Normalmente este tipo de bases de datos se usan para realizar un seguimiento en tiempo real de la información (IBM, 2014; Sol, 2010).

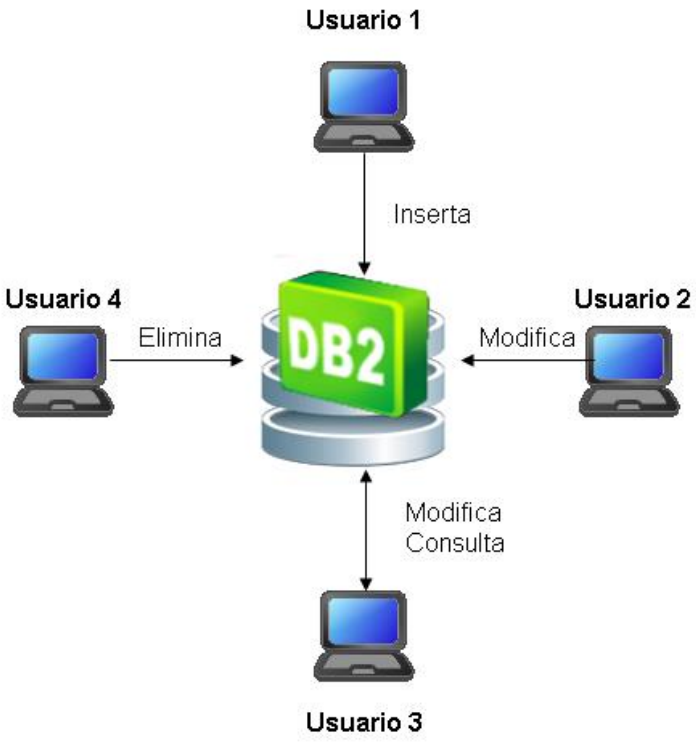

Ilustración 7: Base de datos operacional

2.2.3 BASES DE DATOS JERÁRQUICAS

Una base de datos jerárquica es aquella que organiza los datos jerárquicamente. En este tipo de bases de datos los datos se organizan como si fueran un árbol invertido, como se puede ver en la Ilustración 8 (Sol, 2010).

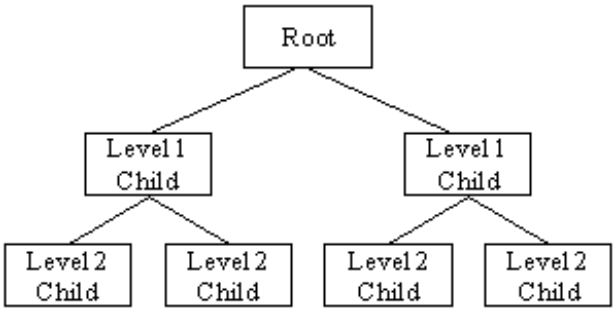

Ilustración 8: Base de datos jerárquica

Este tipo de base de datos tiene varios problemas que imposibilitaron su uso:

- No se pueden agregar registros a las tablas secundarias hasta que estos no se hayan agregado a las tablas primarias.
- Crea gran redundancia de los datos, lo que hace que se vuelva pesada y difícil de manejar.
- Debido a la redundancia de datos, no se pueden manejar correctamente las relaciones entre estos.

2.2.4 BASES DE DATOS EN RED

Inicialmente las bases de datos en red se crearon para solucionar los graves problemas que tenían las bases de datos jerárquicas. Este modelo es muy similar al jerárquico, incluso se podría decir que el jerárquico es un subconjunto del modelo de red, y gracias a él, resuelve el problema de redundancia que tenía el jerárquico.

Este modelo resuelve el problema de redundancia del modelo anterior, para ello, en lugar de utilizar una jerarquía de árbol de un solo padre, el modelo de red utiliza la teoría de conjuntos para proporcionar una jerarquía de árbol con la excepción de que los a los hijos se les permite tener más de un padre y de esta manera implementar las relaciones muchos a muchos.

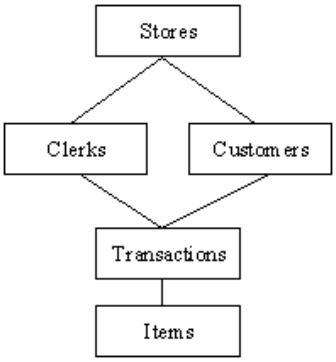

Ilustración 9: Base de datos en red

Sin embargo, a pesar de las mejoras que introdujo este tipo de base de datos, resultaba un modelo muy difícil de implementar y mantener, por lo que al igual que el modelo anterior, está en desuso (Sol, 2010).

2.2.5 BASES DE DATOS RELACIONALES

Una base de datos relacional es aquella en la que todos sus datos están relacionados entre sí. La información se organiza en tablas con filas y columnas. Una tabla se puede definir como una colección de objetos del mismo tipo (filas) cuyos datos están relacionados entre ellas mediante claves primarias.

Este tipo de bases de datos siguen unas reglas de integridad, para asegurar que sus datos están siempre accesibles y de forma precisa (Oracle, 2015).

- **Regla 1:** Cada fila o registro de una tabla debe ser único, esto se consigue gracias a una clave primaria única que posee cada registro, de forma que no permite que haya registros duplicados.
- **Regla 2:** Las claves primarias no pueden tener valores nulos, ya que esto afectaría a la relación entre tablas.

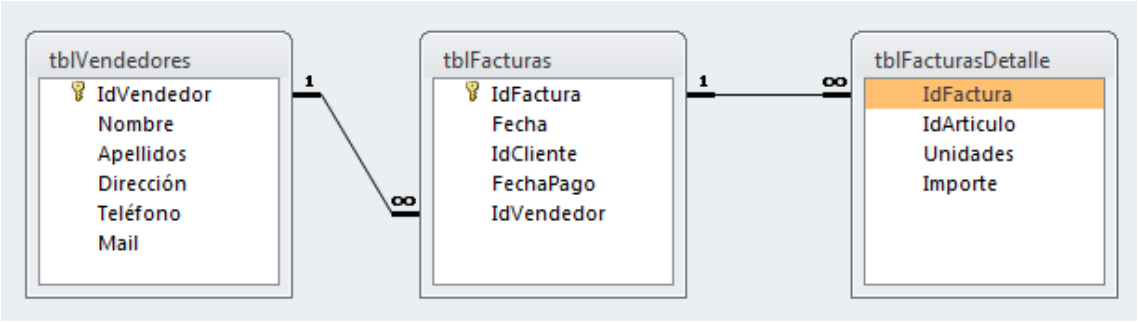

Ilustración 10: Base de datos relacional

Con el paso del tiempo, ha aparecido un problema en el manejo de grandes cantidades de información en este tipo de bases de datos. Las grandes cantidades de información suponen problemas en la gestión de datos y en la accesibilidad a ellos, lo que conlleva una gran disminución del rendimiento.

Algunos ejemplos de este tipo de bases de datos son *Oracle* y *MySQL*.

2.2.6 BASES DE DATOS NO RELACIONALES

Una base de datos no relacional es aquella en la que sus datos no están relacionados entre sí, no siguen una estructura predefinida, y no usan lenguaje *SQL* para las consultas, lo que ha llevado a que también sean conocidas como *NoSQL*.

Las principales ventajas de este tipo de bases de datos son (AcensTechnologies, 2014; Tiwari, 2011):

- **Consumen pocos recursos**, ya que los datos se almacenan en colecciones independientemente de su formato, con lo que la carga computacional a la hora de hacer búsquedas es mínima.
- **Poseen estabilidad horizontal,** mejorar el rendimiento es tan sencillo como indicar al sistema los nodos que están disponibles.
- **Permiten manejar gran cantidad de datos,** gracias a su estructura distribuida.
- **No genera cuellos de botella,** ya que no se usa *SQL*.

2.2.6.1 ORIENTADAS A OBJETOS

En este tipo de *NoSQL*, en lugar de guardar registros los datos se van a guardar en documentos semiestructurados, normalmente en *JSON* o *XML*, los cuales se almacenan en una colección. Esta implementación además de permitir acceder a toda la información de cada documento mediante su identificador único, permite realizar consultas más avanzadas sobre el contenido del documento. Es el modelo *NoSQL* más versátil (AcensTechnologies, 2014; Tiwari, 2011).

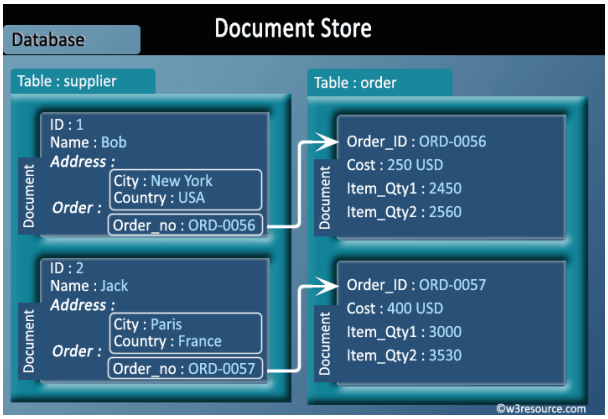

Ilustración 11: Base de datos no relacional orientada a documentos

Algunos ejemplos de este tipo de bases de datos son *Mongodb* y *Couchdb*.

2.2.6.2 ORIENTADAS A COLUMNAS

En este tipo de *NoSQL*, en lugar de almacenar registros, los datos se almacenan en columnas, permitiendo manejar grandes cantidades de datos. Su almacenamiento de datos resulta muy eficiente ya que evita el consumo de espacio al almacenar valores nulos, si no existen valores de una columna, esta no se almacena. Este tipo de *NoSQL* ofrecen un rendimiento muy alto y una arquitectura altamente escalable (AcensTechnologies, 2014; Tiwari, 2011).

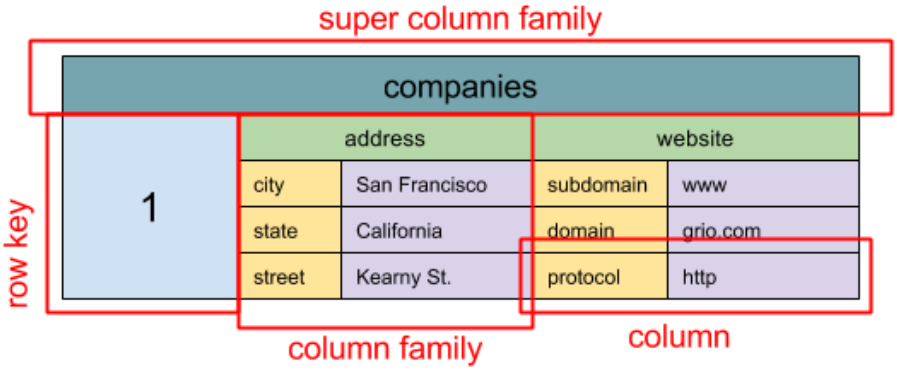

Ilustración 12: Base de datos no relacional orientada a columnas

Algunos ejemplos de este tipo de bases de datos son *HBase* e *Hypertable*.

2.2.6.3 DE CLAVE-VALOR

En este tipo de *NoSQL*, los datos se almacenan como una secuencia de valores agrupados o como un *array* asociativo. Cada elemento de este *array* contiene un identificador único el cual permite recuperar la información rápidamente y de forma eficiente. Son el modelo *NoSQL* más popular y el más sencillo en cuanto a funcionalidad (AcensTechnologies, 2014; Tiwari, 2011).

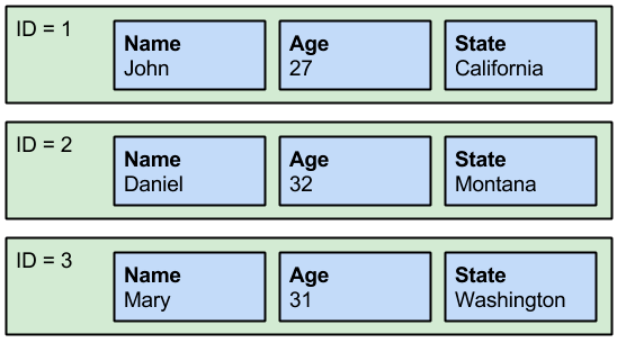

Ilustración 13: Base de datos no relacional de clave-valor

Algunos ejemplos de este tipo de bases de datos son *Cassandra*, *DynamoDB* y *Redis*.

2.2.6.4 EN GRAFO

En este tipo de *NoSQL*, están basadas en la teoría de grafos para enlazar los datos. La información se representa como los nodos de un grafo y las relaciones entre esta información son las aristas que unen los nodos. Son muy útiles a la hora de almacenar datos que poseen muchas relaciones entre ellos ofreciendo una navegación entre las relaciones mucho más eficiente que el tradicional modelo de base de datos relacional (AcensTechnologies, 2014), (Tiwari, 2011).

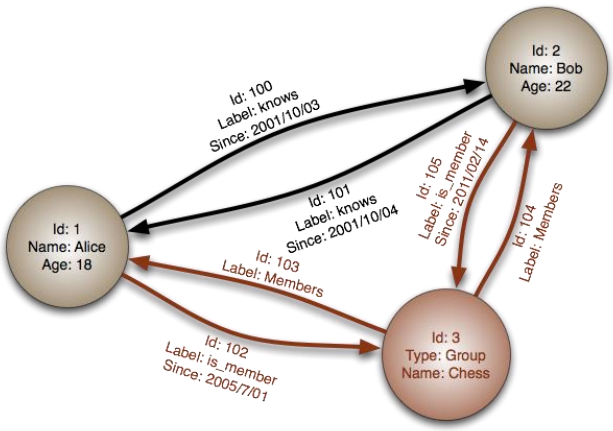

Ilustración 14: Base de datos no relacional en grafo

Unos ejemplos de este tipo de bases de datos son *Neo4j*, e *Infinite graph*.

Finalmente, para poder terminar el estudio de las bases de datos es necesario analizar estas según el teorema *CAP*.

El teorema *CAP* o teorema de *Brewer* dice que en los sistemas distribuidos solo es posible garantizar dos de las tres características que definen a continuación (Tiwari, 2011):

- **Consistencia:** garantiza que al realizar una operación en la base de datos siempre se reciba la misma información independientemente del nodo que procese la petición, esto se consigue replicando la información.
- **Disponibilidad:** garantiza que todos los nodos puedan realizar operaciones sobre los datos de manera correcta, esto se consigue actualizando los datos en todos los nodos periódicamente.
- **Tolerancia a particiones:** mide la capacidad del sistema para continuar funcionando de manera correcta al realizar particiones entre sus nodos, esto se consigue definiendo reglas que indiquen que hacer en caso de que un nodo no esté operativo.

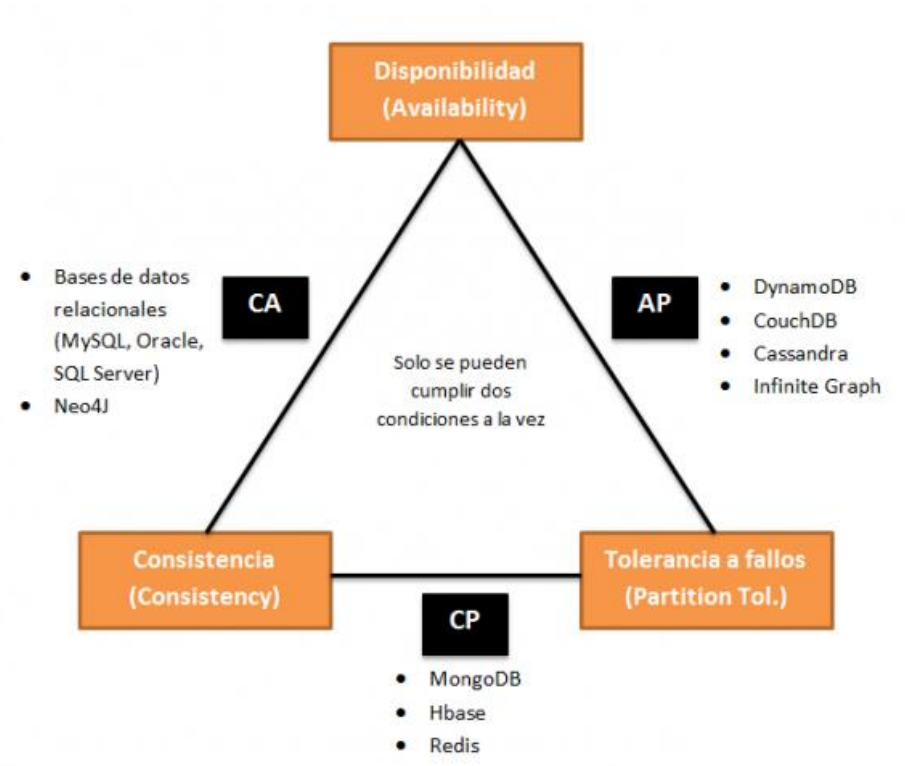

Ilustración 15: Triangulo del teorema CAP

En la Ilustración 15, se puede ver un gráfico del teorema GAP y las posibles combinaciones que se pueden obtener (Tiwari, 2011):

- **CA (consistencia y disponibilidad):** En este tipo se garantiza que el sistema estará siempre disponible atendiendo a peticiones y sus datos procesados serán consistentes, sin embargo, no se permite particionado.
- **CP (consistencia y particionado):** En este tipo se garantiza que los datos procesados serán consistentes, pero no asegura disponibilidad, con lo que es posible la perdida de comunicación al realizar particiones.
- **PA (particionado y disponibilidad):** En este tipo el sistema siempre va a estar disponible a las peticiones, pero se puede perder la comunicación al realizar particiones y como resultado se puede producir inconsistencia en la información.

2.3 SISTEMAS DE MENSAJERÍA

Con el desarrollo de aplicaciones surgió la necesidad de que estas se comunicaran entre sí. El problema de estas comunicaciones es que no todas las aplicaciones usaban los mismos protocolos o adaptadores. Debido a esto se creó la mensajería empresarial, que su objetivo era transmitir información entre sistemas mediante el envío de mensajes. A lo largo de los años ha habido distintas tecnologías:

- **Soluciones para llamadas a procedimientos remotos:** hacen funciones de *middleware* mediante una cola de mensajes, tales como *COM* y *CORBA*
- **Soluciones para la notificación de eventos, comunicación entre procesos y colas de mensajes:** los cuales se incluyen en los sistemas operativos, como *buffers FIFO*, colas de mensajes, tubos (*pipes*), señales, *sockets*,
- **Soluciones para una categoría de *middleware*:** ofrecen un mecanismo de mensajería fiable y asíncrono tales como *WebShpereMQ*, *SonicMQ*, *TIBCO*, *Apache ActiveMQ*, etc...

En este momento empezaron a aparecer los *MOM* (*Middleware* Orientado a Mensajes), una categoría de *software* creada para la intercomunicación de sistemas que ofrece una manera segura, escalable, confiable y con bajo acoplamiento. Los *MOMs* permiten la comunicación entre aplicaciones mediante un conjunto de *APIs* ofrecidas por cada proveedor y lenguaje, de esta manera, se tiene una *API* propietaria y diferente por cada *MOM* existente.
La idea principal de un *MOM* es actuar como un mediador entre los emisores y los receptores de mensajes. Esta mediación ofrece un nuevo nivel de desacoplamiento en la mensajería empresarial. Con lo que se puede decir que los *MOM* se utilizan para mediar en la conectividad y la mensajería, no solo entre las aplicaciones y el *mainframe*, sino de una aplicación a otra (Universidad de Alicante, 2014).

Existen dos modelos de mensajería (Oracle, 2015):

- **Punto a punto (*PTP*):** un mensaje se consume por un único consumidor. Como se puede ver en la Ilustración 16, un cliente envía un mensaje a la cola y ese mensaje se consume únicamente por otro cliente.

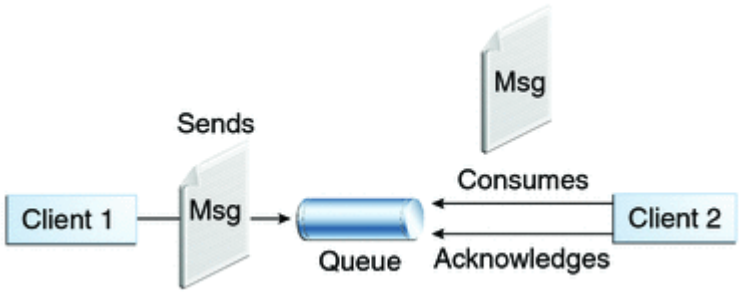

Ilustración 16: Modelo de mensajería PTP

- **Publicación/Suscripción (*Pub/Sub*):** un mensaje se consume por muchos consumidores. Como se puede ver en la Ilustración 17, un cliente publica un mensaje en un tema (topic) y ese mensaje es consumido únicamente por todos aquellos clientes que se hayan suscrito a ese tema (*topic*).

Ilustración 17: Modelo de mensajería Pub/Sub

Las ventajas de usar mensajes como sistema de comunicación son las siguientes (Universidad de Alicante, 2014):

- **Integración de sistemas:** las aplicaciones pueden estar desarrolladas con diferentes tecnologías.
- **Escabilidad:** se pueden añadir nuevos procesadores de mensajes sin que los emisores sean conscientes de los cambios.
- **Asincronía:** los mensajes se procesan e manera asíncrona, con lo que el emisor no tiene que esperar una respuesta de este. En el caso de procesar el mensaje, cuando exista un receptor este lo recibirá.
- **Desacoplamiento:** las aplicaciones no se conocen entre sí, por lo que se pueden remplazar sin que el resto se vea afectadas.

Las desventajas son (Universidad de Alicante, 2014):

- Falta de acceso transparente y eficiente para la comunicación entre componentes locales.
- El código para empaquetado y desempaquetado de los mensajes debe ser escrito por el programador, lo cual resulta incómodo para este.

2.3.1 *JMS*

JMS (*Java Message Service*) es una *API* de *Java* desarrollada por *Sun Microsystems* que permite la comunicación entre aplicaciones basadas en el intercambio de mensajes dando soporte a los modelos de mensajería *PTP* y *Pub/Sub*. *JMS* no es un *MOM* propiamente dicho, ya que abstrae la interacción entre los clientes de mensajería permitiendo una comunicación de forma independiente al proveedor como se puede ver la ilustración 12 (Universidad de Alicante, 2014).

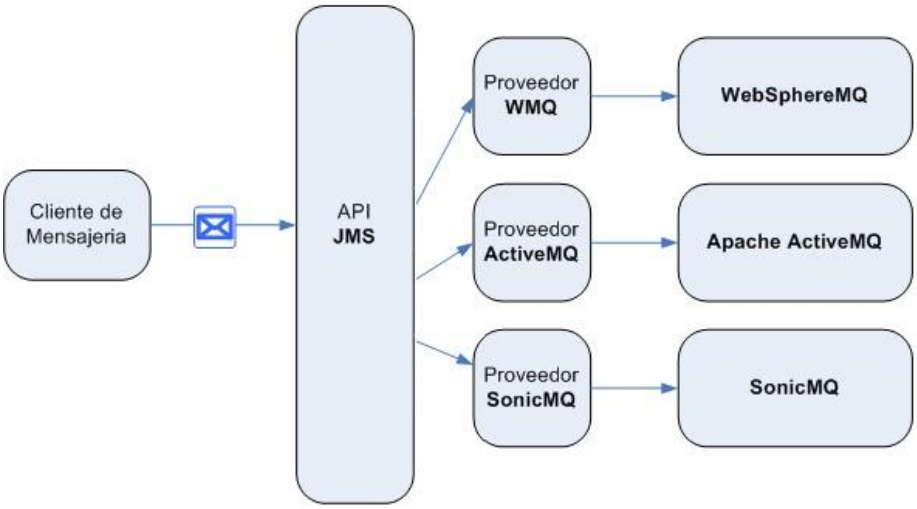

Ilustración 18: Comunicación JMS independiente de los proveedores

En el 2002 salió la última *release* de JMS y desde entonces se trata de una tecnología estable y madura.

Algunos ejemplos del uso de estas tecnologías son *WebSphereMQ*, *ActiveMQ* y *SonicMQ*.

2.3.2 *AMQP*

AMPQ (*Advanced Message Queuing Protocol*) es un protocolo de estándar para *MOM*, cuyo objetivo principal es la interoperabilidad de diferentes servicios. Se creó en el 2004 por diversas empresas como *OpenAMQ* y *RabbitMQ*. Su principal diferencia con otras tecnologías es que no define solamente una *API*, sino que también define el formato de los datos que van a ser enviados, de manera que cualquier aplicación pueda crear y leer mensajes en este formato de datos independientemente del lenguaje usado.

En esta tecnología los mensajes se almacenan en colas, las cuales garantizan que estos se entreguen en el mismo orden en el que han llegado siguiendo el mecanismo *FIFO* (*Fist In*, *First Out*) (RabbitMQ, 2008).

Algunos ejemplos del uso de estas tecnologías son *OpenAMQ* y *RabbitMQ*.

2.4 CONTROLADORES DE VERSIONES

Los controladores de versiones son sistemas que registran los cambios realizados sobre un archivo o un conjunto de archivos a lo largo del tiempo. Los datos se almacenan en repositorios, los cuales contienen versiones de cada archivo. Dentro de un repositorio pueden existir ramas, que son versiones paralelas de un proyecto, lo cual resulta muy útil a la hora de hacer desarrollos en paralelo por distintos desarrolladores. Además de todo esto, permite añadir comentarios sobre los cambios realizados en cada versión, para que a la hora de buscar los cambios de una versión resulte mucho más sencillo.

Estas herramientas resultan imprescindibles a la hora de hacer un desarrollo en equipo, pero también son muy útiles para desarrollos de una sola persona (Barrera González, 2004).

2.4.1 SISTEMA DE CONTROL DE VERSIONES LOCAL

Este es el sistema clásico de control de versiones, en el que se hace una copia de los archivos en distintos directorios. Es un sistema muy simple pero también muy propenso a errores si no se tiene una buena organización (Chacon & Straub, 2014).

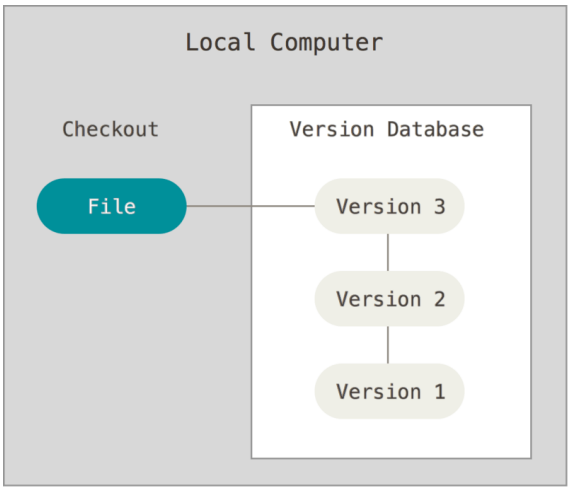

Ilustración 19: Sistema de control de versiones local

2.4.2 SISTEMA DE CONTROL DE VERSIONES CENTRALIZADO

En los sistemas de control de versiones centralizados existe un único servidor centralizado donde se guardan todos los archivos versionados. Al existir solo un servidor, los administradores tienen un control muy exacto de lo que hace cada persona, sin embargo, se depende 100% del buen funcionamiento de este, lo cual no resulta práctico.

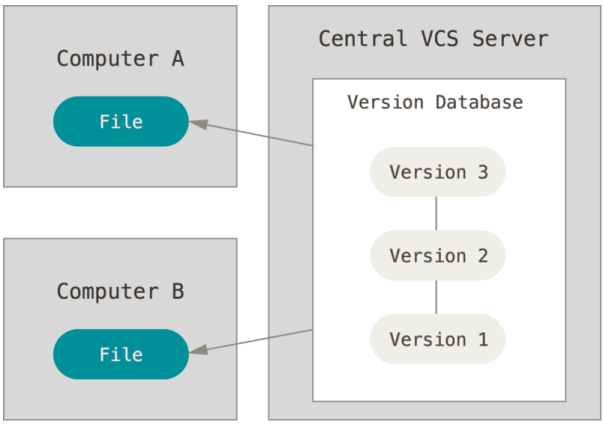

Ilustración 20: Sistema de control de versiones centralizado

En estos sistemas, los archivos que se suben al servidor son únicamente los que hayan sido modificados de la versión base. A la hora de ir a recuperar el conjunto entero de un proyecto hay que ir versión por versión escogiendo los archivos que han sido modificados, lo que hace de este proceso muy lento (Chacon & Straub, 2014).

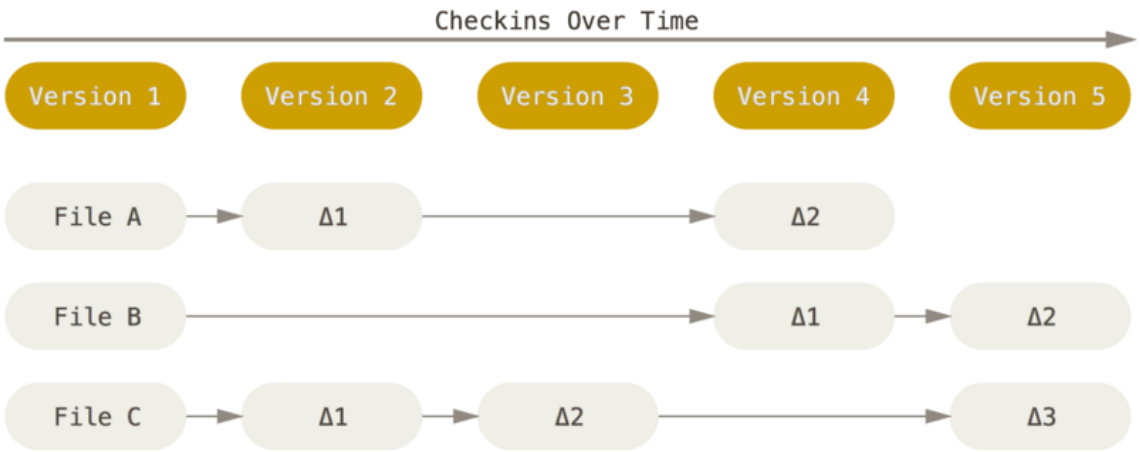

Ilustración 21: Almacenamiento de datos en sistemas centralizados

Durante mucho tiempo ha sido el estándar de control de versiones, pero actualmente está cayendo en desuso.

Algunos ejemplos del uso de estas tecnologías son los sistemas *CSV* y *Subversion*, el cual se podría decir que es una evolución de *CSV*.

2.4.3 SISTEMA DE CONTROL DE VERSIONES DISTRIBUIDO

En los sistemas de control de versiones distribuidos cada usuario tiene su propio repositorio local y una visión entera de todo el repositorio. Al tener cada usuario un repositorio local, la información está muy replicada y en caso de que se produzca un fallo en la máquina del repositorio remoto, es muy fácil que esta se recupere, ya que cada repositorio local puede funcionar como *backup*. Los recursos que requieren este tipo de sistemas es menor que el centralizado, ya que parte del trabajo lo hacen los repositorios locales.

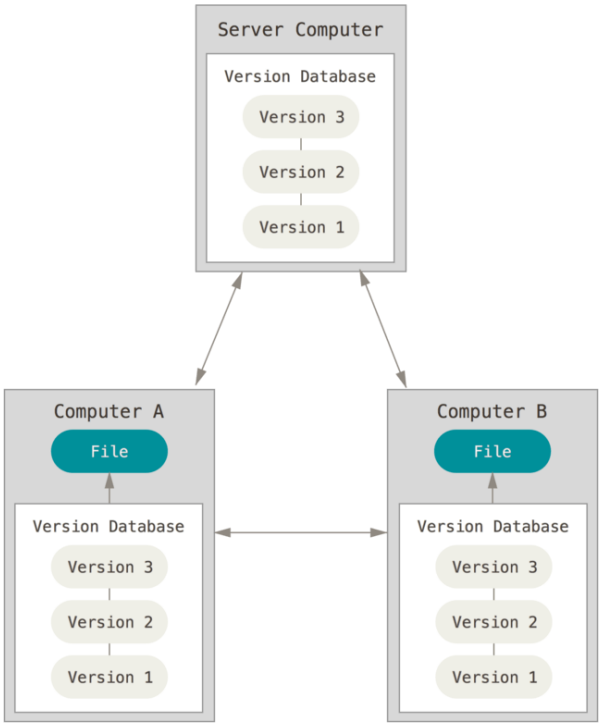

Ilustración 22: Sistema de control de versiones distribuido

En estos sistemas, los archivos que se suben al servidor son los que hayan sido modificados de la versión base, y el servidor almacena un paquete con todos los archivos del proyecto en ese momento, en otras palabras, es como si almacenara una foto del proyecto en cada momento. Más tarde cuando se quiere recuperar el proyecto, tan solo hay que buscar la foto de la versión deseada y descargarla, lo cual resulta mucho más fácil y eficaz que el método usado en los sistemas centralizados (Chacon & Straub, 2014).

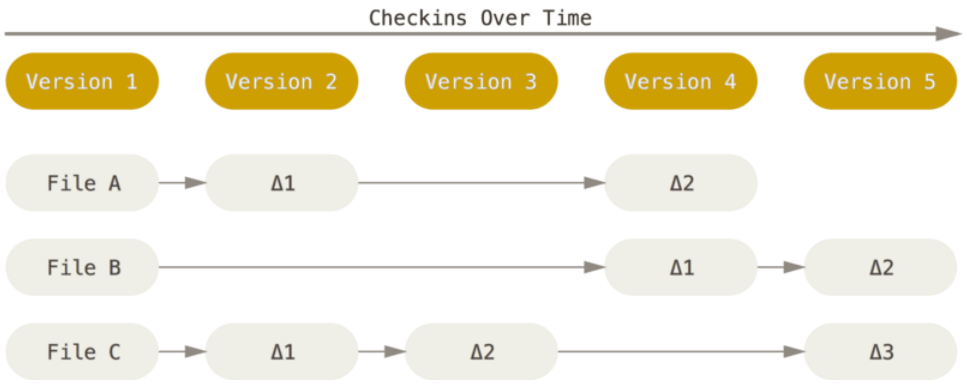

Ilustración 23: Almacenamiento de datos en sistemas distribuidos

Algunos ejemplos del uso de estas tecnologías son los sistemas *Git* y *Mercurial*, ambos desarrollados como alternativa a *BitKeeper*, un sistema de control de versiones del *kernel* de *Linux*, que en ese momento era privado.

2.5 ELECCIÓN DE LAS TECNOLOGÍAS

En este apartado se va a realizar la elección de las tecnologías previamente estudiadas.

2.5.1 ELECCIÓN DE ARQUITECTURA DE *SOFTWARE*

La *SOA* idónea para el desarrollo de este proyecto sería una suite de integración, ya que con una suite se tendrían todas las herramientas necesarias para el desarrollo de esta aplicación. Una de las grandes desventajas de las suites de integración es que no hay ninguna gratuita lo suficientemente madura como para desarrollar esta aplicación, sin embargo, es muy común usar un *ESB* junto con otras tecnologías para realizar una aplicación que en conjunto sea similar a una suite de integración.

Una vez escogido el tipo de tecnología, queda escoger la herramienta para el desarrollo, las dos herramientas más usadas de *ESB* son *Mule ESB* y *Fuse ESB*, las cuales se comparan a continuación:

Las semejanzas de ambas herramientas son:

- Son *open source* (*Mule ESB* posee una versión de pago con más componentes).
- Están basadas en *Java*.
- Su configuración está basada en *XML*.
- El soporte es de pago.
- Soportan la *API* de *Spring* (*Mule ESB* lo lleva integrado y en *Fuse ESB* se puede integrar fácilmente).

Las diferencias entre ambas herramientas son:

- Cada uno posee un *IDE* (Entorno de Desarrollo Integrado) propio.
- El servidor *standalone* de *Mule ESB* es mucho más robusto que el de *Fuse ESB*.
- Existe mucha más documentación sobre *Mule ESB* que sobre *Fuse ESB*.
- *Mule ESB* ofrece cursos gratuitos con la versión de pago por tiempo limitado mientras que los cursos gratuitos de *Fuse ESB* son muy limitados.
- *Mule ESB* es con diferencia mucho más usada a nivel empresarial que *Fuse ESB*.

Después de este estudio, la herramienta elegida, a pesar de que las dos son similares, ha sido *Mule ESB*, ya que a la hora de comenzar es mas fácil debido a la documentación existente y a los cursos gratuitos que ofrecen. Además, si pensamos en el futuro del proyecto, *Mule ESB* es la mejor opción ya que es la herramienta *ESB* más usada a nivel empresarial.

Una vez llegados a este punto, la elección del resto de tecnologías y herramientas de estas, se va a ver limitada por los componentes que proporciona *Mule ESB*, de esta manera se eliminarán las posibles incompatibilidades entre las distintas tecnologías y se facilitará el desarrollo de la aplicación.

2.5.2 ELECCIÓN DE LA BASE DE DATOS

La base de datos elegida ha sido una base de datos no relacional, ya que inicialmente se va a usar para guardar los registros de auditoría de la aplicación y dependiendo el punto en el que se encuentre el registro, los datos tendrán distinto formato. Más en profundidad, el tipo de base de datos no relacional elegido es el orientado a documentos, ya que de esta manera se tendrá un documento por cada petición, facilitando de este modo el acceso a los datos.

Las posibles herramientas más populares para esta tecnología son *MongoDB* y *CouchDB*, ambas gratuitas, pero la idónea para usar junto con *Mule ESB* es *MongoDB*, ya que *Mule ESB* posee un conector propio para la conexión con esta herramienta.

Por último, para la visualización de los datos de *MongoDB* se ha escogido la herramienta *Robomongo*, la cual es una herramienta más popular para ello, ya que fue la primera en mostrar los datos de forma asíncrona sin bloquear el hilo principal de la aplicación (Robomongo, 2016).

2.5.3 ELECCIÓN DEL SISTEMA DE MENSAJERÍA

Las posibles tecnologías de mensajería para implementar con *Mule ESB* son *JMS* y *AMQP*, ya que *Mule ESB* posee conectores para ambas. La elegida es *JMS* ya que es una tecnología estable y muy madura, lo cual va a facilitar su implementación en la aplicación.

La herramienta más utilizada por excelencia es *Apache ActiveMQ* la cual es gratuita y provee una interfaz gráfica para visualizar en todo momento las colas existentes, los mensajes en ellas, las conexiones e incluso permite mandar mensajes a las colas.

2.5.4 ELECCIÓN DEL CONTROLADOR DE VERSIONES

El controlador de versiones elegido es uno distribuido, ya que, comparando el funcionamiento de los sistemas distribuidos y centralizados, el de los distribuidos es notablemente mejor. La tecnología de los sistemas distribuidos más usada es *Git*, por lo que la elegida ha sido esta.

Dentro de *Git*, hay dos herramientas predominantes, *Github* y *BitBucket*. A pesar de que las dos son muy parecidas la elegida ha sido *Bitbucket*, la cual te permite usar tanto la tecnología *Git* como la tecnología *Mercurial*, además permite tener repositorios privados y públicos de forma gratuita, con un número máximo de colaboradores de 5 (ampliable en una versión de pago).

Finalmente, para facilitar el manejo de este repositorio se usará la herramienta *SourceTree*, la cual, permite gestionar de forma gráfica todas las tareas de gestión propias de un controlador de versiones de forma gratuita.

CAPÍTULO 3

3. DESCRIPCIÓN TÉCNICA

En este capítulo se va realizar una descripción detallada de la aplicación desde un punto de vista técnico.

Se comenzará explicando los tipos de llamadas que se pueden realizar. Se continuará explicando de forma detallada los tipos de servicios que se consumen para realizar esas llamadas. Más delante, se explicará el formato que contienen los documentos en la base de datos y por último se mostrará el uso del controlador de versiones escogido.

3.1 TIPO DE PETICIONES

Esta aplicación tiene tres peticiones claramente diferenciadas en función de sus actuaciones, las cuales se detallan a continuación.

3.1.1 PETICIONES SÍNCRONAS

En las peticiones síncronas, como se puede ver en la Ilustración 24, una vez la aplicación ha comprobado que se trata de una petición síncrona el mensaje pasa al SIB (Servicio de invocación base), el cual, es un servicio genérico que abstrae la aplicación del servicio que se va a invocar.

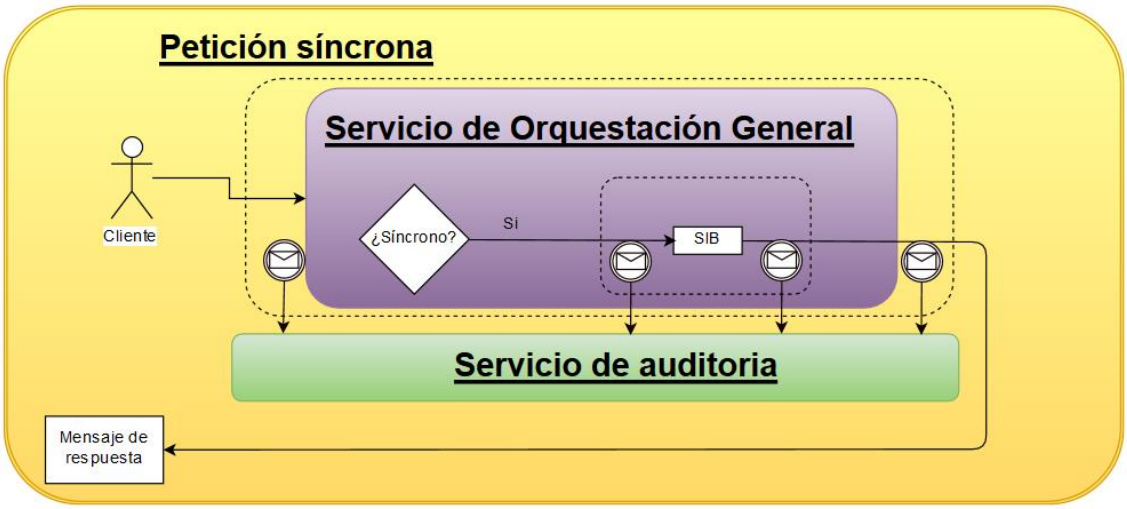

Ilustración 24: Petición síncrona

Finalmente, la aplicación muestra un mensaje de respuesta al cliente. Durante toda la petición hay una serie de banderas, en las cuales se registra el mensaje en la auditoría, para más tarde almacenarlo en la base de datos. Si en este tipo de peticiones se produce una excepción en alguna parte del flujo, esta es capturada y al cliente se le devuelve un mensaje con información sobre el error que ha producido esa excepción.

3.1.2 PETICIONES ASÍNCRONAS

En las peticiones asíncronas, como se puede ver en la Ilustración 25, una vez la aplicación ha comprobado que se trata de una petición asíncrona se establecen las pautas de gestión de errores y se envía el mensaje a una cola JMS. En este momento, se envía una respuesta al cliente indicándole que el mensaje ha sido recibido y almacenado correctamente.

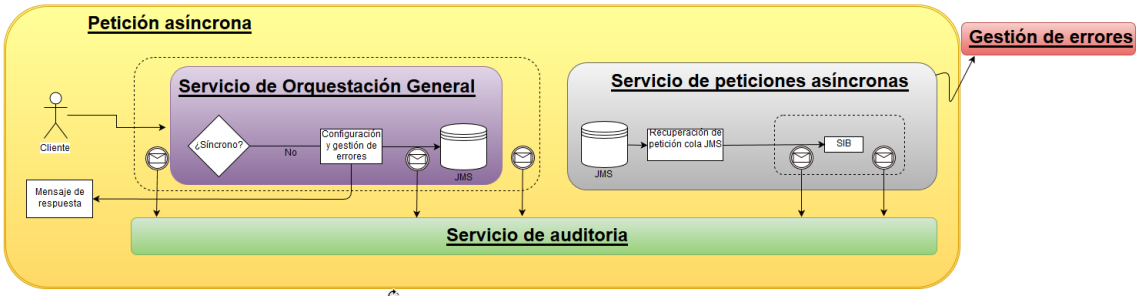

Ilustración 25: Petición asíncrona

El mensaje almacenado en la cola JMS es procesado por el servicio de peticiones asíncronas, el cual lo extrae y hace la llamada al servicio de invocación base. En el servicio de peticiones asíncronas es donde se realiza el control y gestión de errores que se explicará más adelante.

3.1.3 *BATCH*

En una petición de *batch*, como se puede ver en la Ilustración 26, se separan las peticiones para poder tramitarlas individualmente. Cada una de estas peticiones, es tratada como una petición asíncrona, con lo que el funcionamiento del *batch* una vez separadas las peticiones es el mismo que el de una petición asíncrona.

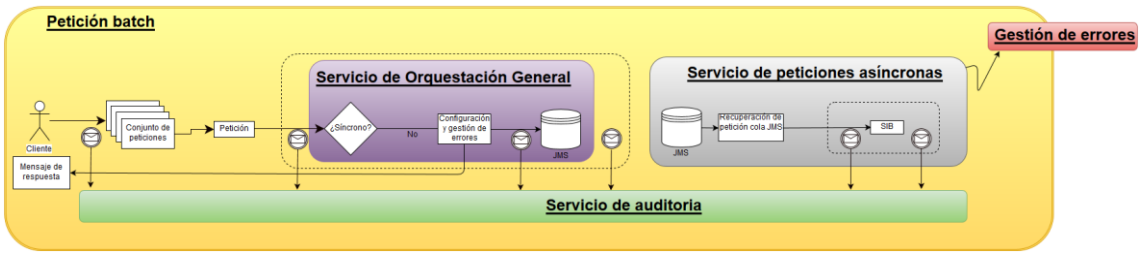

Ilustración 26: Batch

3.2 TIPO DE SERVICIOS

A lo largo de la ejecución de la aplicación se llaman a distintos servicios, los cuales, tienen funcionalidades distintas. Una característica general de estos servicios es que existe un gran nivel de desacoplo entre ellos.

3.4.1 SERVICIO DE ORQUESTACIÓN

Este servicio es al primero que se llama una vez se envía la petición. En él, se comprueba que tipo de petición se está haciendo y se actúa en consecuencia.

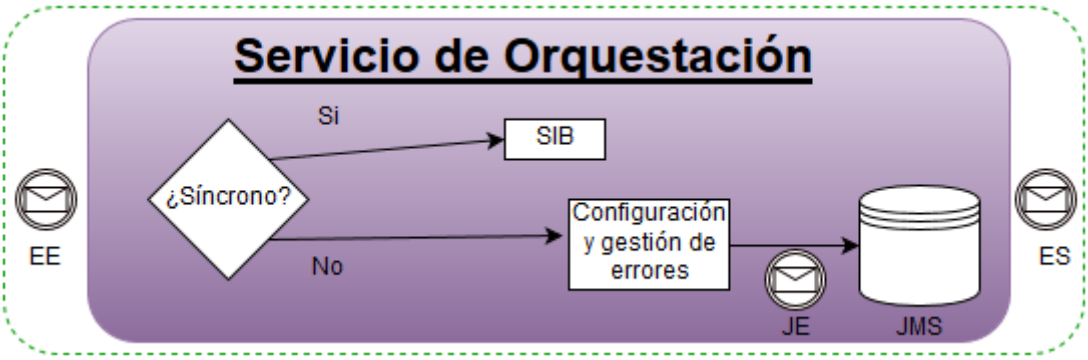

Ilustración 27: Servicio de orquestación

En caso de que la petición sea síncrona, se llama al servicio de invocación base, que se explicará más adelante, para realizar la llamada al servicio *backend* final. En caso de que la petición no sea síncrona, es decir sea asíncrona o de *batch*, se guardará en la cabecera del mensaje la prioridad de este y el número de reintentos a realizar en caso de error y se almacenará dicho mensaje en una cola *JMS*.

Todo este servicio está encapsulado en uno superior, el cual enviará el mensaje actual a la auditoría antes y después del servicio de orquestación, con las banderas EE y ES. En caso de que la petición no sea síncrona también se envía un mensaje a la auditoría antes de enviar el mensaje a la cola *JMS*, con la bandera JE.

3.4.2 SERVICIO DE INVOCACIÓN BASE (*SIB*)

Este servicio se encarga de realizar las llamadas al servicio *backend* final. En él se realizan las transformaciones de datos necesarias para adecuar la llamada a la del servicio *backend* y una vez se ha obtenido la respuesta se vuelven a transformar los datos para dejarlos en su formato original.

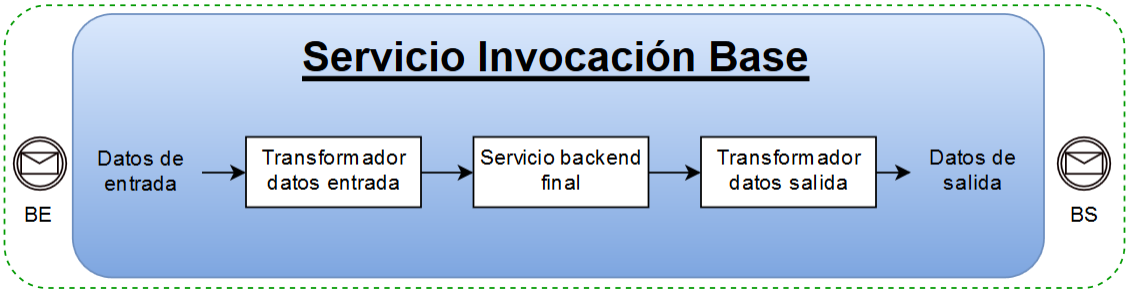

Ilustración 28: Servicio de invocación base

Todo este servicio va encapsulado en otro superior, el cual, es un servicio genérico por el que pasarán todas las peticiones y se encarga de enviar un mensaje a la auditoría antes y después de la llamada al *backend*, con las banderas BE y BS.

3.4.3 SERVICIO *BACKEND* FINAL PILOTO

Este es un servicio de prueba creado exclusivamente para las pruebas de esta aplicación. Se trata de un servicio, el cual, va a coger los datos del mensaje, para formar un *email* (asunto y cuerpo del mensaje) y mandarlo a un destinatario especificado en el mensaje.

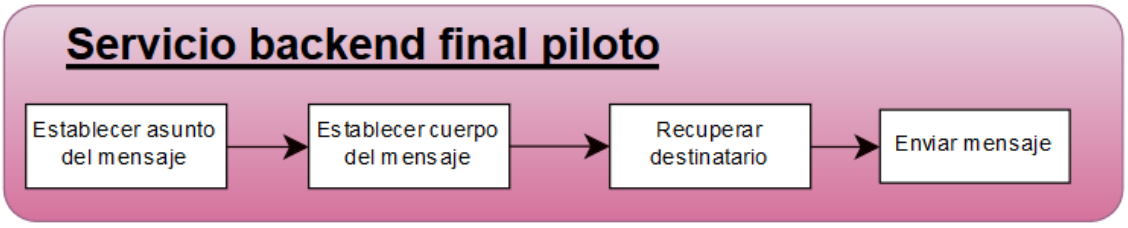

Ilustración 29: Servicio backend final piloto

Para el uso y prueba de esta aplicación se han implementado dos servicios iguales con nombres distintos, “mandarEmailSub_Flow” y “mandarEmail2Sub_Flow”.

3.4.4 SERVICIO DE PETICIONES ASÍNCRONAS (*SPA*)

Este servicio es el encargado de recuperar los mensajes de la cola *JMS*. Una vez recuperados estos mensajes, se llama al servicio de invocación base.

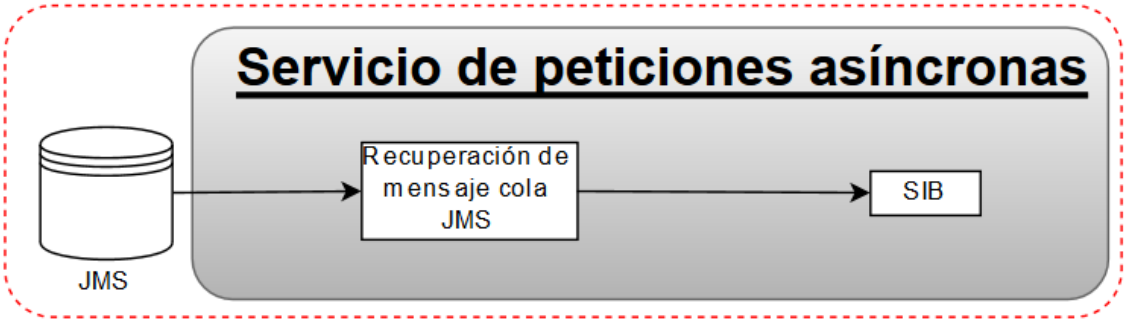

Ilustración 30: Servicio de peticiones asíncronas

Este servicio lleva alrededor una gestión de errores que se explicará más adelante.

3.4.5 SERVICIO DE AUDITORÍA

El servicio de auditoría consta de dos partes, el servicio productor de auditoría y el servicio consumidor de auditoría.

3.2.4.1 SERVICIO PRODUCTOR DE ADITORIA

En este servicio se lee una bandera que lleva el mensaje para saber desde donde ha sido mandado. Las banderas pueden ser:

- **EE:** *ESB* entrada, es la petición según le llega a la aplicación.
- **ES:** *ESB* salida, es la respuesta de la petición tal cual se devuelve al cliente.
- **JE:** *JMS* entrada, los datos que se almacenan en la cola *JMS*.
- **BE:** *Backend* entrada, son los datos que se envían al servicio de invocación base, para que este los transforme y haga la llamada al servicio *backend*.
- **BS:** *Backend* salida, son los datos que devuelve el servicio *backend* una vez transformados al formato común de la aplicación.
- **CM:** Cola muerta, son los mensajes almacenados en la cola muerta.
- ***Batch*:** se guarda la petición del *batch* tal cual se recibe.

Ilustración 31: Servicio productor de auditoría

Una vez identificado de donde procede cada el mensaje se forma un nuevo mensaje que será enviado a la cola *JMS*.

3.2.4.2 SERVICIO CONSUMIDOR DE ADITORIA

En este servicio se recuperan los mensajes de la cola *JMS* y se introducen en la base de datos.

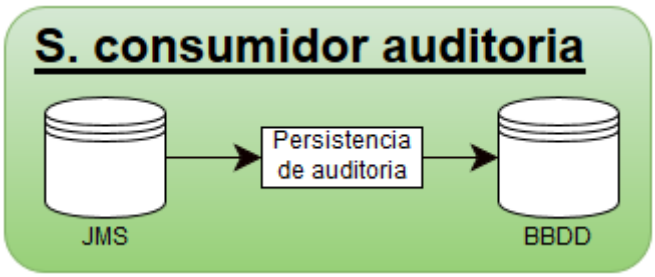

Ilustración 32: Servicio consumidor de auditoría

Teniendo el servicio de auditoría en dos partes se desacopla la formación del mensaje y la inserción en la base de datos, de manera que si la base de datos no está operativa los mensajes se pueden quedar almacenados en la cola *JMS* hasta que vuelva a estar operativa, evitando perder información.

3.4.6 GESTIÓN DE ERRORES

Este servicio se encarga de permanecer a la escucha por si se producen errores. En caso de que se produzca un error, se lanzará una excepción en el flujo que será capturada por este servicio.

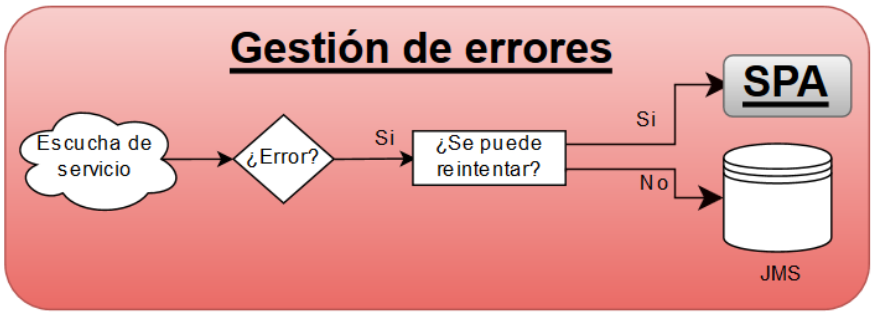

Ilustración 33: Gestión de errores

Cuando una excepción entra en este servicio, se comprueba una variable del mensaje, la cual, lleva la cuenta de los reintentos hechos por esa petición. En caso de que esta variable contador sea menor o igual que la variable de reintentos definida al enviar la petición, se reintenta esta petición, enviándola al servicio de peticiones asíncronas, y se incrementa el valor de la variable contador en 1. En caso de que no se pueda reintentar, porque la variable contador sea superior a la variable de reintentos, esta petición se almacena una cola *JMS* llamada "cola muerta" para su posterior inserción en la auditoría, de manera que se pueda llevar un registro de las peticiones que haya sido imposible realizar.

3.4.7 SERVICIO DE COLA MUERTA

Este servicio se encarga de estar a la escucha de la cola JMS llamada "cola muerta".

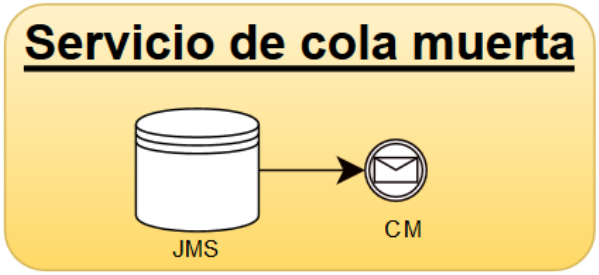

Ilustración 34: Servicio cola muerta

En caso de que se introduzca un mensaje en esta cola, este servicio lo extrae y lo manda al servicio de auditoría, con la bandera CM, para registrarlo junto con el resto de mensajes de esa petición.

3.3 LOGS

Todos estos servicios explicados anteriormente cuentan con unos componentes llamados *loggers*, situados en sitios estratégicos (normalmente al inicio y final de cada servicio, y en las capturas de excepciones), los cuales almacenan trazas con esta información en un fichero de log.

Esta aplicación dispone de tres ficheros de *logs*.

- **fgis:** *log* general donde se escriben las trazas de entrada y salida de los servicios, para poder llevar un seguimiento de las peticiones.
- **fgis-error:** *log* de error donde se escriben únicamente las trazas que contienen errores con información sobre estos.
- **fgis-audit:** *log* de auditoría donde se escriben las trazas de información general referente a la auditoría, es muy similar al general.

El principal problema que puede aparecer a la hora de almacenar los *logs* es el tamaño, es decir, que con el tiempo contengan mucha información, parte de ella demasiado antigua como para ser de utilidad y se vuelvan muy pesados. Este problema se ha solventado configurando estos *logs* con *Log4j,* una herramienta configurada por *Apache,* la cual añade *Mule ESB* en sus proyectos, para la configuración de este tipo de archivos.

En este archivo se han configurado específicamente estos 3 *logs* para que en cada uno se muestren las trazas correspondientes al nivel de prioridad adecuado, ya sea *ERROR* o *DEBUG*. Además de esto se ha incluido la configuración necesaria para que haya una rotación de los *logs*. Cuando un *log* llegue a ocupar 10MB se renombrará, por ejemplo, “fgis.log” pasará a ser “fgis-1.log”, hasta que haya un total de 10 *logs* de ese tipo, en ese momento el último se irá perdiendo, pero como la información de este será antigua no se perderán datos de importancia.

3.4 *APIKIT* Y FORMATO MENSAJES

3.4.1 *APIKIT*

APIkit es un conjunto de herramientas de *Maven* (herramienta software para la gestión y construcción de proyectos *Java*) provisto por *Mule ESB,* para crear una *API REST*. Esta *API* se diseña con un archivo *RAML* (lenguaje basado en *YAML* para el diseño de peticiones *REST*), en el cual se puede especificar las peticiones que va a contener la aplicación, el tipo de estas y sus esquemas y ejemplos de entrada y salida. Esta herramienta en *Mule ESB* también permite ver de forma gráfica, a través del navegador, una consola donde poder realizar las peticiones y consultar los datos requeridos en cada petición o ver un ejemplo de esta. Además de esto, con esta herramienta se pueden gestionar errores en los datos de entrada a la petición, ya que, si los datos introducidos no siguen el esquema definido en el archivo RAML, se indica donde se ha producido el error y de qué tipo, de esta manera aseguramos que los datos de las peticiones van a ser correctos (Mulesoft, 2015).

3.4.2 FORMATO DE LOS MENSAJES

El formato de los mensajes de las peticiones lo podemos separar en dos tipos, mensajes de entrada, es decir, lo que envía la petición y mensajes de salida, los que responde la petición.

3.4.2.1 FORMATO MENSAJES DE ENTRADA

Estos mensajes tienen dos partes, los metadatos y los datos.

Ilustración 35: Formato de mensaje entrada

Los metadatos son los datos necesarios para la aplicación, nombre del servicio *backend* al que se va a invocar, el tipo de petición que se va a realizar y la política de errores en caso de las peticiones asíncronas y *batch*. La política de errores va a tener dos campos requeridos, el número de reintentos y la prioridad del mensaje en la cola JMS.

Por otro lado, están los datos, indiferentes al tipo de petición, que van a ser los datos que va a necesitar el servicio *backend*. En el caso del servicio *backend* final piloto, los datos requeridos son el destinatario, el asunto y el mensaje del correo a enviar.

En caso de que la petición sea un *batch*, los datos van a ser un *array* de datos, es decir, un elemento del *array* con los datos correspondientes a la llamada *backend* por cada petición que se desee hacer en el *batch*, con lo que, si se quiere mandar 3 correos, el formato del mensaje será el de la Ilustración 36.

Ilustración 36: Formato de mensaje entrada batch

3.4.2.2 FORMATO MENSAJES DE SALIDA

Estos mensajes son más simples que los de entrada, ya que la respuesta contiene menos datos.

Ilustración 37: Mensaje de salida

Estos mensajes solo tienen tres campos. El idPadre, es un identificador único por cada petición *batch*, todas las peticiones individuales del *batch* contienen este identificador padre para poder consultar los registros de auditoría. En caso de que la petición no sea *batch*, este valor es "*null*". El id, es un identificador único para cada petición individual, independientemente del tipo de petición que la haya lanzado. La respuesta, es la respuesta que devuelve la aplicación, ya sea una respuesta de OK o de error.

3.5 EXTERNALIZACION DE VARIABLES

Como cualquier otra aplicación, esta posee una serie de variables que se usan frecuentemente en los componentes de estas. Estas variables son los datos de conexiones a base de datos, al servidor de colas *JMS*, direcciones y puertos de *host*, etc.

En una aplicación creada por una empresa es muy común tener varios entornos, normalmente son el de desarrollo, donde se desarrollan las funcionalidades; el de pase a producción, donde se realizan pruebas exhaustivas de la aplicación; y finalmente el de producción, donde el cliente final usa la aplicación. Debido a la existencia de estos entornos muchas veces estas variables cambian. Para simplificar el pase de un entorno a otro, se han creado 3 archivos de propiedades, cada uno para un entorno, con todas estas variables centralizadas.

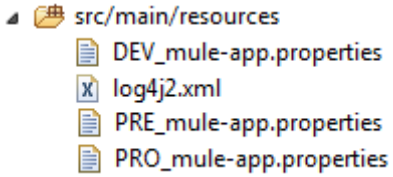

Ilustración 38: Archivos de propiedades

Para usar un archivo de propiedades u otro, basta con especificar, en una variable de entorno, antes de ejecutar la aplicación, indicando en que entorno te encuentras, como se ve en la Ilustración 39.

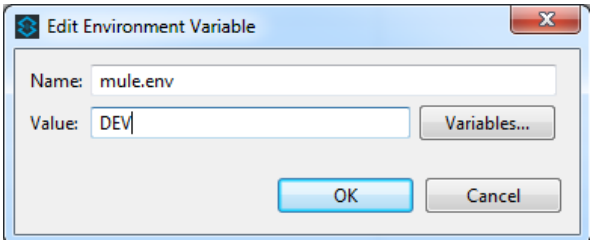

Ilustración 39: Especificación variable de entorno

3.6 FORMATO DOCUMENTOS BASE DE DATOS

La base de datos, como se ha dicho previamente, es una base de datos no relacional orientada a documentos, y si bien es cierto que las bases de datos no relacionales no tienen una estructura predefinida, las orientadas a objetos normalmente siguen una estructura. A continuación, se mostrará la estructura seguida en un lenguaje *JSON*.

Ilustración 40: Ejemplo forma de árbol de un documento

En la Ilustración 40, se puede ver un ejemplo en forma de árbol de un documento. Los campos generales en estos documentos son los siguientes:

- **_id:** id única definida por la base de datos.
- **id:** id del mensaje que genera la petición, a través de esta id se pueden ver todos los mensajes de auditoría almacenados para cada petición.
- **idPadre:** en caso de se haya hecho una petición *batch*, a esta se la asigna una idPadre que tendrán igual todas las peticiones del *batch*, y con ella se pueden ver los mensajes de auditoría de todas las peticiones de un *batch*.
- **Servicio:** nombre del servicio *backend* que se invoca.
- **Tipo servicio:** tipo de petición, síncrona, asíncrona o *batch*.

Después de estos campos generales, cada registro tendrá un campo por cada bandera de auditoría, estos campos pueden cambiar dependiendo el tipo de petición que sea y si se han producido errores o no.

Ilustración 41: Campos de banderas auditoría

Como se puede ver en la Ilustración 41, no todos los campos contienen los mismos datos. Los campos "Esb entrada", "Esb salida", "Backend entrada", "*Backend* salida" y "Cola muerta" son iguales, contienen un *payload* que son los datos existentes en ese momento en la aplicación y la fecha en la que se hizo la llamada a auditoría. El campo "*Jms* entrada", además de tener los campos descritos anteriormente, tiene un campo a mayores con la política de errores que se seguirá en caso de que se produzca algún error. Finalmente, el campo "Errores" es un *array* que contiene todos los errores producidos en esa petición, cada error contiene la excepción que ha producido el error y la fecha en la que se ha producido este.

3.7 CONTROL DE VERSIONES

Antes de crear la aplicación, se creó un repositorio en *Bitbucket* para ir versionándola a medida que se iba desarrollando.

Este tipo de repositorios permite crear ramas para el desarrollo de las funcionalidades. Como esta es una aplicación inicial y desarrollada solo por una persona solo se ha creado una rama *master* y una rama *develop*. En la rama *master* se encuentra la aplicación inicial, sin funcionalidades y en la rama *develop* cada vez que se ha añadido una funcionalidad nueva o se ha hecho un campo importante se han subido los cambios como se ve en la Ilustración 42.

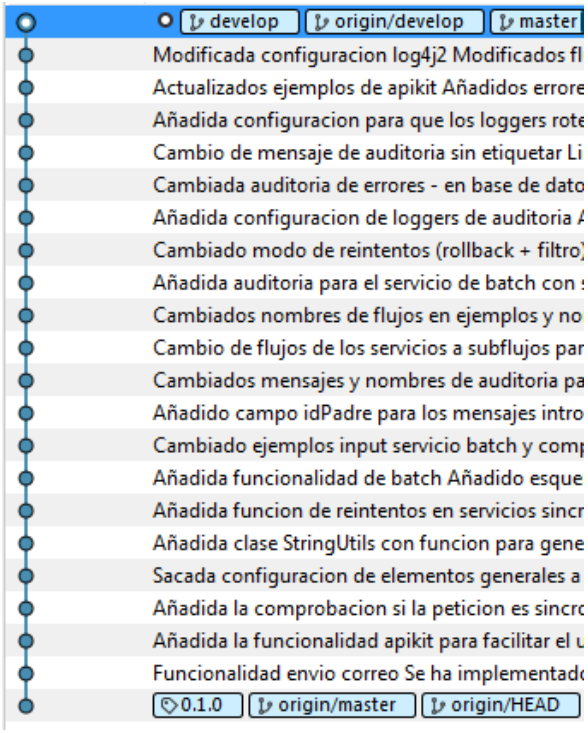

Ilustración 42: Cambios subidos al repositorio

Finalmente, una vez terminado la aplicación se sube todo a la rama *master* y se cierra la *develop*, en un futuro si se encuentran *bugs* o se quieren añadir nuevas funcionalidades, se pueden crear nuevas ramas y repetir el proceso.

CAPÍTULO 4

4. DESCRIPCIÓN FUNCIONAL

Este capítulo tiene como objetivo presentar una descripción funcional de la aplicación de forma que cualquier usuario pueda usar todas las funcionalidades de manera correcta y eficaz. Para ello, se explican las funcionalidades de manera clara, apoyándose en capturas de los pasos que hay en cada funcionalidad.

Para realizar esta descripción funcional o manual de usuario se parte de que la aplicación está desplegada correctamente en un servidor, en este caso local (Anexo 1), y los programas externos como son el de mensajería de colas y el de base de datos están correctamente arrancados (Anexos 2 y 3).

4.1 CONSOLA *APIKIT*

El paso inicial, una vez arrancada la aplicación, es acceder a la consola del *APIKit*, la cual nos indica las peticiones que podemos realizar, el tipo de petición que son, el esquema de como tienen que conformarse y un ejemplo de ellas.

Para acceder a esta consola es suficiente con poner en un navegador la url: http://localhost:8081/api/console/

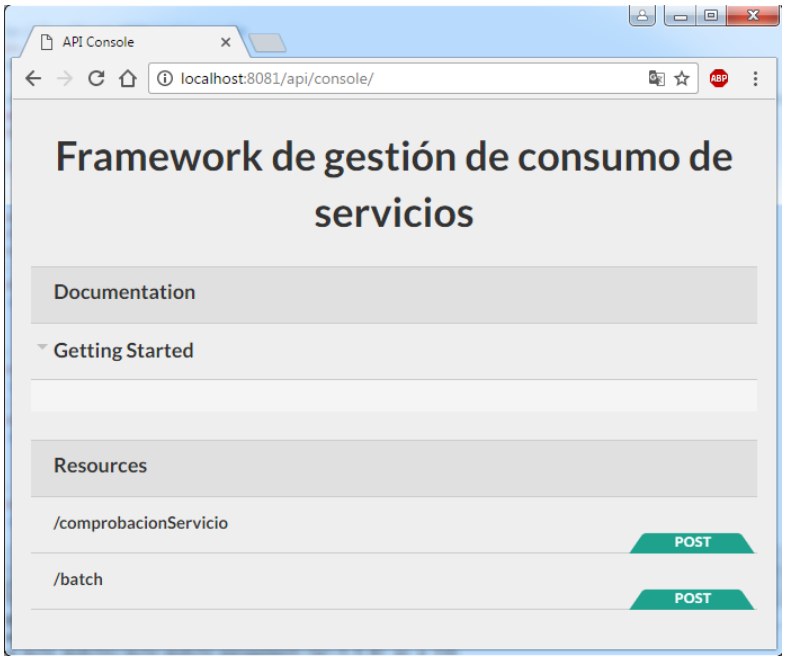

Ilustración 43: ConsolaAPIKit

En la Ilustración 43, se puede ver de una manera visual el nombre de la aplicación junto con los dos servicios *POST* de los que esta dispone, el de comprobación de un servicio y el de *batch*, los cuales son detallados más adelante.

Al hacer *click* en la pestaña *POST* de cualquiera de estos servicios, va a aparecer una ventana con tres secciones claramente diferenciadas, la de petición, la de respuesta y la de probar.

4.1.1 SECCIÓN DE PETICIÓN (*REQUEST*)

Como se puede ver en la Ilustración 44, al comienzo de esta sección hay una descripción del funcionamiento de la petición. A continuación, se encuentra el espacio para los esquemas de seguridad, en caso de que estén definidos y después se sitúa el cuerpo del mensaje.

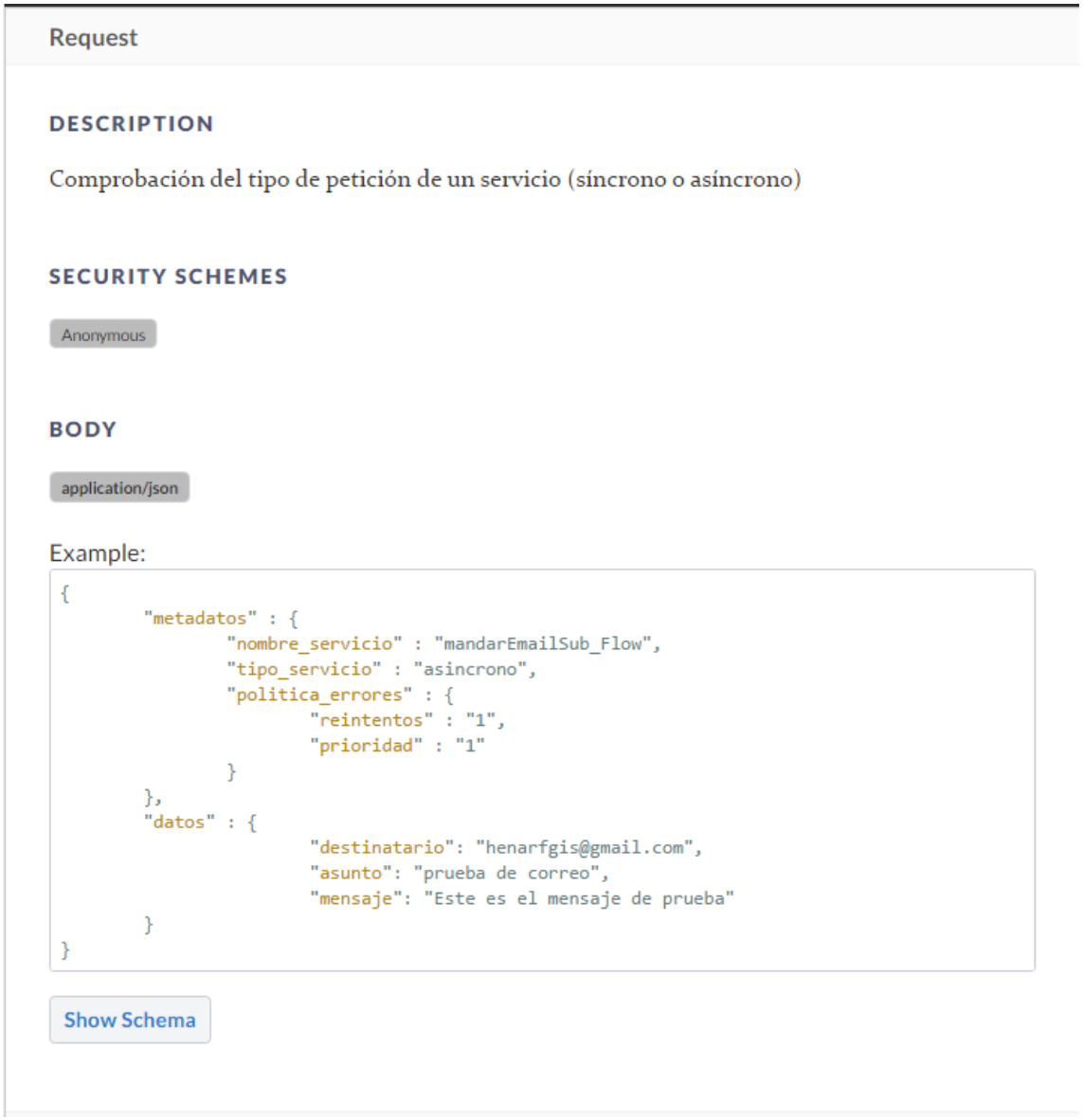

Ilustración 44: Sección de petición

Inicialmente, en el cuerpo del mensaje se especifica el tipo de lenguaje en el que se mandan los datos, a continuación, hay un ejemplo de una posible petición y debajo un botón que muestra el esquema que ha de cumplir el mensaje de la petición.

4.1.2 SECCIÓN DE RESPUESTA (*RESPONSE*)

En esta sección, aparecen las posibles respuestas que se obtienen al realizar la petición, en función del estado esta devuelva. Si todo es correcto, el estado es 200 y si hay algún error, 400 o 500. Dentro de cada pestaña, al igual que en la sección de petición, se encuentra inicialmente el lenguaje en que se va a mostrar la respuesta, a continuación, un ejemplo de esta y finalmente el esquema.

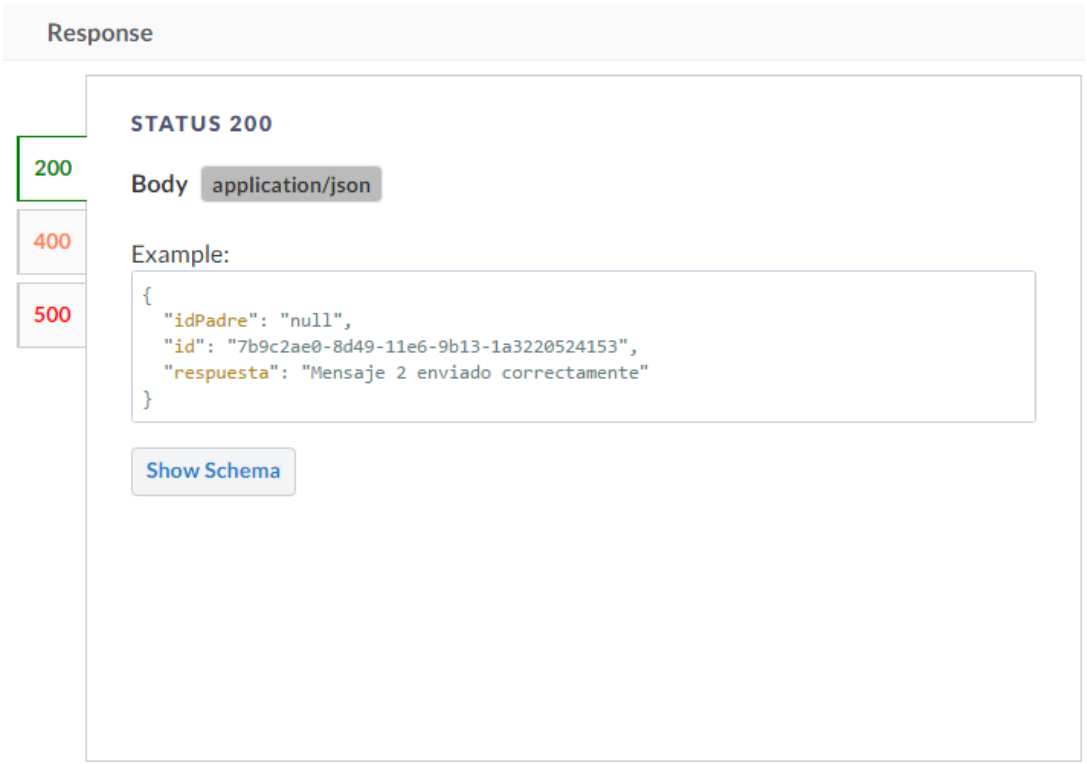

Ilustración 45: Sección de respuesta

4.1.3 SECCIÓN DE PRUEBA (*TRY IT*)

En esta sección, se encuentra todo lo necesario para probar la petición de manera correcta.

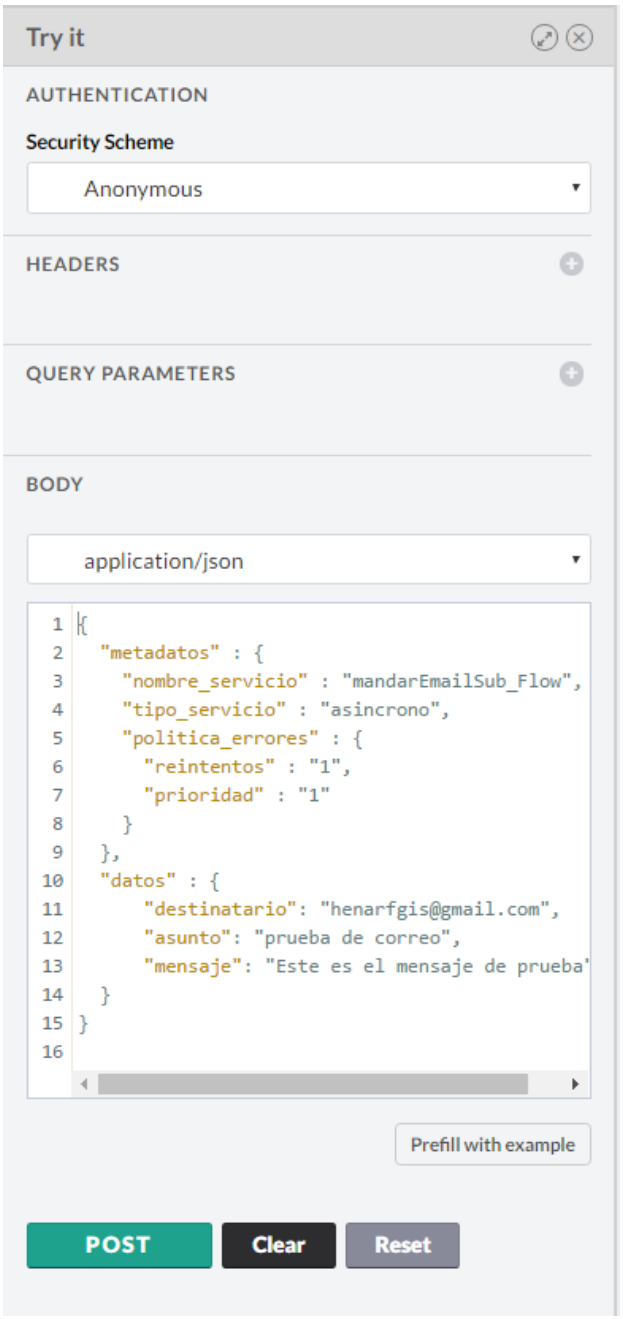

Ilustración 46: Sección de prueba - Realizar la petición

Inicialmente, se encuentran los datos de autentificación, en caso de que se haya definido un esquema de seguridad, a continuación, las cabeceras que llevara la petición, parámetros y finalmente el cuerpo del mensaje.

A la hora de realizar la petición hay dos opciones, la primera que se escriban los datos del mensaje a mano y la segunda que se use el ejemplo dado por la consola, presionando el botón "*Prefill with example*". Una vez definidos los datos del mensaje, para ejecutar la petición basta con pulsar el botón *POST*, en caso de que la petición sea de otro tipo, *GET*, *PUT* o *DELETE*, el contenido del botón cambiará por este.

Una vez ejecutada la petición, a esta sección se le añaden dos subsecciones más, una de petición y otra de repuesta.

4.1.3.1 SUBSECCION DE PETICIÓN (*REQUEST*)

En esta subsección, aparecen los datos de la petición realizada, la URL, las cabeceras y el cuerpo del mensaje. Esta subsección es meramente informativa.

Ilustración 47: Sección de prueba - Subsección de petición

4.1.3.1 SUBSECCION DE RESPUESTA (*RESPONSE*)

En esta subsección, se muestra la respuesta de la petición realizada. Se muestra el estado que indica si la petición se ha hecho correctamente o se ha producido algún error, a continuación, están las cabeceras del mensaje y finalmente el cuerpo de este.

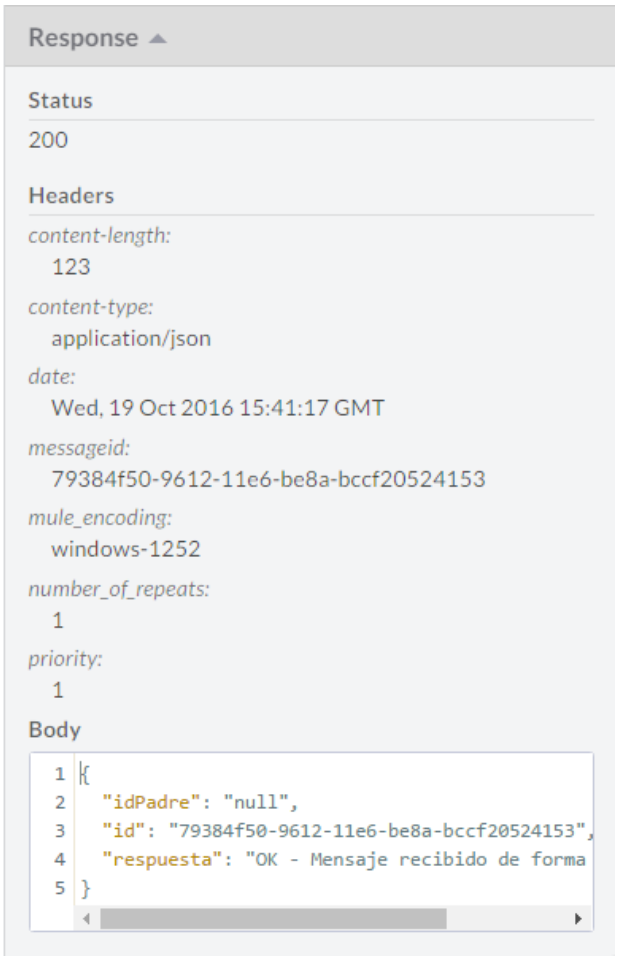

Ilustración 48: Sección de prueba - Subsección de respuesta

4.2 REALIZAR PETICIONES

Antes de realizar la petición hay que tener en cuenta que en la herramienta *APIKit* se definen unos esquemas de datos de entrada y datos de salida. Los datos de entrada son datos modificables por el usuario, con lo que tienen que seguir el esquema definido, ya que, de lo contrario, la herramienta *APIKit* mostrará un error y no se lanzará la petición.

Antes de hacer una descripción de cómo se hacen las peticiones, se va a estudiar brevemente el esquema de estas.

Ilustración 49: Formato de mensaje de peticiones

Como se puede ver en la Ilustración 49, el mensaje consta de dos partes, los metadatos y los datos.

Los metadatos son datos requeridos por la aplicación para realizar la petición. Los campos obligatorios para cualquier petición son:

- **Nombre del servicio,** se refiere al nombre del servicio *backend* al que se quiere invocar. En esta aplicación hay dos servicios *backend* piloto y en este campo solo se permite el nombre de esos dos servicios: "*mandarEmailSub_Flow*", "*mandarEmail2Sub_Flow*".
- **Tipo de petición,** se refiere a si la petición es síncrona, asíncrona o *batch*, estos tres tipos de peticiones son los únicos permitidos en este campo.

En caso de que la petición sea asíncrona o *batch* se requiere otro campo:

- **Política de errores**, en él se define la política de errores a seguir. Consta de dos campos también obligatorios:
  - **Prioridad,** la prioridad que va a tener el mensaje a la hora de almacenarse en la cola JMS.
  - **Reintentos,** el número máximo de reintentos posibles en caso de que se produzca un error.

Los datos, son datos requeridos por el servicio *backend* final. En caso de los dos servicios piloto implementados, los cuales mandan un email, los campos requeridos son el destinatario, asunto y mensaje del email a mandar. De estos datos no se realiza ninguna comprobación, ya que son datos externos a la aplicación.

En caso de que la petición sea *batch*, el campo datos va a ser un *array* con tantos elementos como peticiones se deseen realizar, cada una de ellas con los datos que se acaban de definir.

4.2.1 PETICIÓN COMPROBACIÓN DEL SERVICIO

Con esta petición se van a lanzar peticiones síncronas y asíncronas. La aplicación cuando le llega una petición, sea del tipo que sea, va a realizar una comprobación para saber de qué tipo es y en función de ello llevar a cabo una u otra acción.

Para ejecutar la aplicación lo primero que hay que realizar es pulsar la pestaña *POST* de este servicio, y de esta manera obtendremos todas las secciones explicadas anteriormente. Puesto que este servicio hace comprobaciones en función de los datos que enviemos en el cuerpo vamos a separar dos casos, petición síncrona y petición asíncrona.

4.2.1.1 PETICIÓN SÍNCRONA

Como su nombre indica esta petición es síncrona, de tal manera que al ejecutar esta petición se hace una llamada directamente al servicio *backend* deseado, esperando la respuesta de este y devolviéndola. En caso de que en esta petición se produzca un error, no se lleva a cabo ninguna política de errores, simplemente la respuesta que se muestra al usuario indica el error que se ha producido.

Los datos a introducir para ejecutar este servicio son los que se ven en la Ilustración 50.

```
{
    "metadatos" : {
        "nombre_servicio" : "mandarEmail2Sub_Flow",
        "tipo_servicio" : "sincrono"
    },
    "datos" : {
            "destinatario": "henarfgis@gmail.com",
            "asunto": "prueba de correo sincrona",
            "mensaje": "Este es el mensaje de prueba para la petición sincrona"
    }
}
```

Ilustración 50: Datos entrada - Petición síncrona

Y la respuesta que se obtiene es la que se ve en la Ilustración 51.

```
{
    "idPadre": "null",
    "id": "fa18b600-9612-11e6-be8a-bccf20524153",
    "respuesta": "Mensaje 2 enviado correctamente"
}
```

Ilustración 51: Datos salida - Petición síncrona

En esta respuesta se obtiene una *idPadre*, la cual se usa únicamente en las peticiones de *batch*, con lo que en esta petición no tiene ningún interés, una id con la que se podrá ver el registro de auditoría de esta petición y además se indica que el mensaje se ha enviado correctamente, lo cual se puede comprobar accediendo a la cuenta de correo.

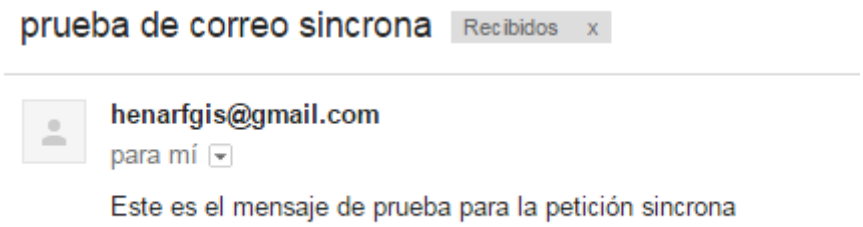

Ilustración 52: Correo enviado petición síncrona

En la Ilustración 52, se puede ver que el correo se ha mandado correctamente y que este tiene el asunto y el mensaje especificados en la petición.

4.2.1.2 PETICIÓN ASÍNCRONA

Como su nombre indica esta petición es asíncrona, de tal manera que al ejecutarla los datos se almacenan en una cola JMS para ser procesados más tarde.

Los datos a introducir para ejecutar este servicio son similares a los de la petición síncrona, añadiendo el campo de política de errores, como se ve en la Ilustración 53.

```
{
    "metadatos" : {
        "nombre_servicio" : "mandarEmailSub_Flow",
        "tipo_servicio" : "asincrono",
        "politica_errores" : {
            "reintentos" : "1",
            "prioridad" : "1"
        }
    },
    "datos" : {
            "destinatario": "henarfgis@gmail.com",
            "asunto": "prueba de correo asincrona",
            "mensaje": "Este es el mensaje de prueba para petición asincrona"
    }
}
```

Ilustración 53: Datos entrada - Petición asíncrona

Y la respuesta que se obtiene es la que se ve en la Ilustración 54.

```
{
    "idPadre": "null",
    "id": "e99fdfe0-9614-11e6-be8a-bccf20524153",
    "respuesta": "OK - Mensaje recibido de forma correcta"
}
```

Ilustración 54: Datos salida - Petición asíncrona

En esta respuesta se obtiene una *idPadre*, la cual se usa únicamente en las peticiones de *batch*, con lo que en esta petición no tiene ningún interés, una id con la que se podrá ver el registro de auditoría de esta petición y además se indica que el mensaje se ha recibido correctamente y esta se ha almacenado en una cola *JMS*.

4.2.2 *BATCH*

Con esta petición se van a mandar un conjunto varias peticiones asíncronas de una vez.

Para ejecutar la aplicación, como en el servicio anterior, lo primero que hay que realizar es pulsar la pestaña *POST* de este servicio, y de esta manera obtendremos todas las secciones explicadas con anterioridad.

Los datos a introducir para ejecutar este servicio son similares a los de la petición asíncrona, pero el campo datos en este caso será un *array*, como se ven en la Ilustración 55.


```
{
  "metadatos": {
    "nombre_servicio": "mandarEmailSub_Flow",
    "tipo_servicio": "batch",
        "politica_errores": {
    "reintentos": "1",
    "prioridad": "1"
    }
  },
  "datos": [
    {
    "destinatario": "henarfgis@gmail.com",
    "asunto": "prueba de correo batch 1",
    "mensaje": "Este es el mensaje de prueba batch 1"
    },
    {
    "destinatario": "henarfgis@gmail.com",
    "asunto": "prueba de correo batch 2",
    "mensaje": "Este es el mensaje de prueba batch 2"
    }
  ]
}
```


Ilustración 55: Datos entrada - Petición batch

Y la respuesta que se obtiene es la que se ve en la Ilustración 56.


```
{
    "idPadre": "9a5a5880-8d49-11e6-9b13-1a3220524153",
    "id": "9a63ce61-8d49-11e6-9b13-1a3220524153",
    "respuesta": "OK - Mensaje recibido de forma correcta"
}
```


Ilustración 56: Datos salida - Petición batch

En esta respuesta, al contrario que en las anteriores peticiones el campo *idPadre* no es *null*, esta id nos sirve para identificar todas las peticiones realizadas mediante el *batch* en la auditoría. Como en las anteriores peticiones, también se obtiene una id con la que se puede ver el registro de auditoría de esta petición y, además, igual que en la petición asíncrona, se indica que el mensaje se ha recibido correctamente y esta se ha almacenado en una cola *JMS*.

4.3 FORZAR APARICIÓN DE ERROR

Tanto en las peticiones asíncronas como en las de *batch*, los mensajes almacenados en la cola *JMS* se procesan a medida que el servidor de colas está activo y disponible. Una vez un mensaje sale de esta cola *JMS*, se procesa y en caso de que se produzca algún fallo es cuando se aplica la política de errores que definida previamente.

Para poder observar este funcionamiento en caso de error, lo más sencillo es desconectar internet para forzar que se produzca un error a la hora de mandar un email en la invocación de este servicio *backend* final piloto. Al estar ejecutando la aplicación en modo local no va a existir ningún problema por no tener acceso a internet.

La política de errores que está definida es una política de reintentos, es decir, si se produce un error la petición, esta se va a reintentar tantas veces como se haya especificado, antes de destruir el mensaje. La mejor forma de ver esto es mediante los *logs* previamente configurados.

Si se accede a la carpeta *logs*, se puede ver que hay tres *logs* con distintos nombres:

- **fgis:** en él se recogen las trazas generales.
- **fgis-error:** en él se recogen las trazas de error.
- **fgis-audit:** en él se recogen las trazas procedentes al registro de auditoría.

Una vez almacenados los mensajes en la cola *JMS*, es indiferente si han llegado a esta como una petición de *batch* o como petición asíncrona. Los datos para realizar esta petición son similares a los descritos previamente en la petición asíncrona.

```
{
    "metadatos" : {
        "nombre_servicio" : "mandarEmail2Sub_Flow",
        "tipo_servicio" : "asincrono",
        "politica_errores" : {
            "reintentos" : "3",
            "prioridad" : "1"
        }
    },
    "datos" : {
            "destinatario": "henarfgis@gmail.com",
            "asunto": "prueba de correo",
            "mensaje": "Este es el mensaje de prueba"
    }
}
```

Ilustración 57: Datos entrada - Petición asíncrona

Como se puede ver en la Ilustración 57, se ha establecido un número máximo de reintentos de tres.

Al enviar esta petición, se obtiene una respuesta con un estado 200 y un mensaje la petición se ha almacenado correctamente en la cola *JMS*.

Una vez el mensaje de petición sale de esta cola es cuando se va a producir el error, y si se observan las trazas del *log* de error (fgis-error) se ve que se han realizado tres reintentos como se ha especificado en la política de errores.

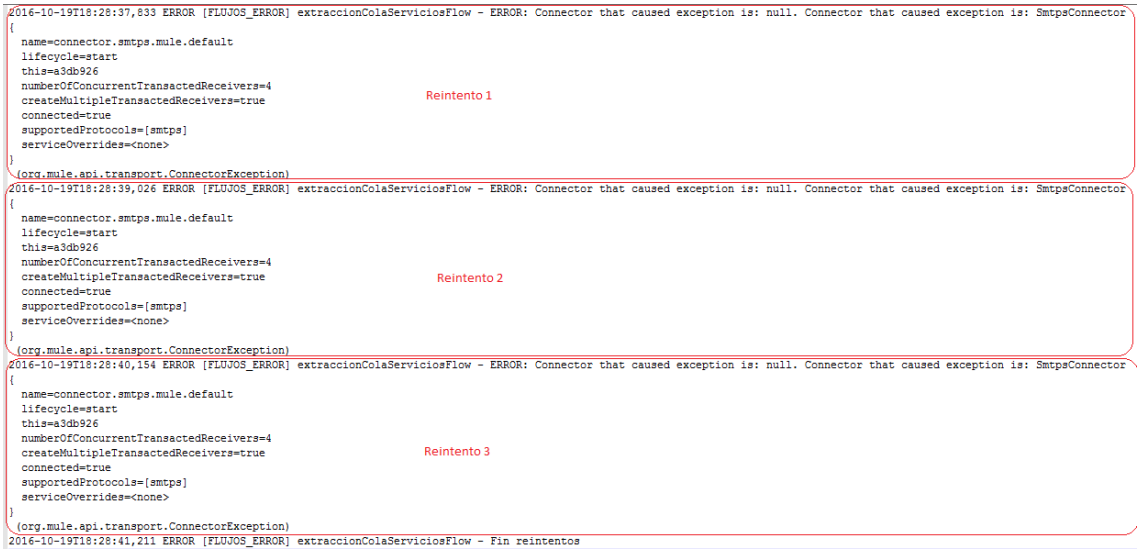

Ilustración 58: Reintentos log error

Finalmente, cuando se acaba el número de reintentos, el mensaje es enviado a otra cola distinta, la cola muerta, la cual, después de procesarlo, lo almacena en la base de datos para que quede constancia de que ha sido imposible realizar esa petición.

4.4 *APACHE ACTIVEMQ*

Apache ActiveMQ es una herramienta para el uso de mensajería de colas *JMS*. En esta sección se va a explicar brevemente el uso de la consola de administración de esta herramienta, para la visualización de los mensajes existentes en las colas (Apache ActiveMQ, 2016).

Para acceder a la ventana de administración tan solo hay que escribir en el navegador la *URL* http://localhost:8161/admin/ e introducir los datos de autentificación, por defecto, el usuario es *admin* y la contraseña es *admin*.

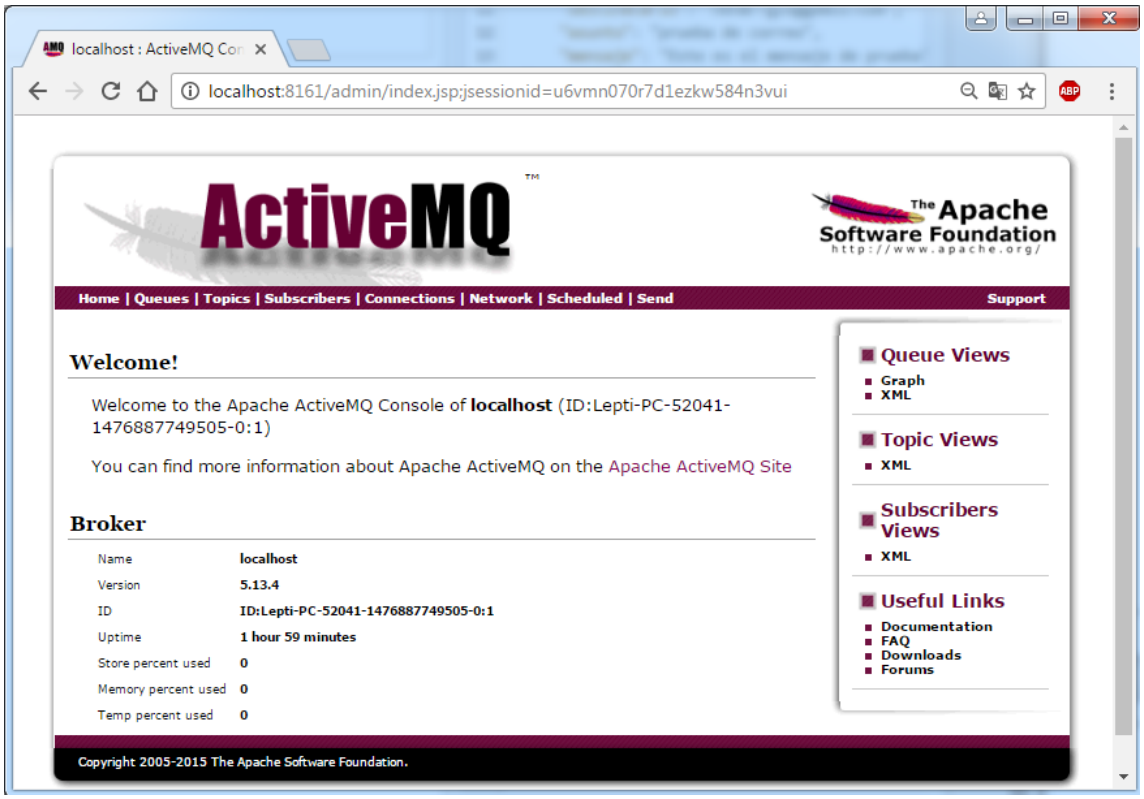

Ilustración 59: Ventana administración ActiveMQ

En esta pantalla inicial, como se ve en la Ilustración 59, se pueden ver los datos de la conexión existente en este momento.
Una vez en esta ventana se hace *click* en "*Queues*" se pueden ver las colas existentes en ese momento.

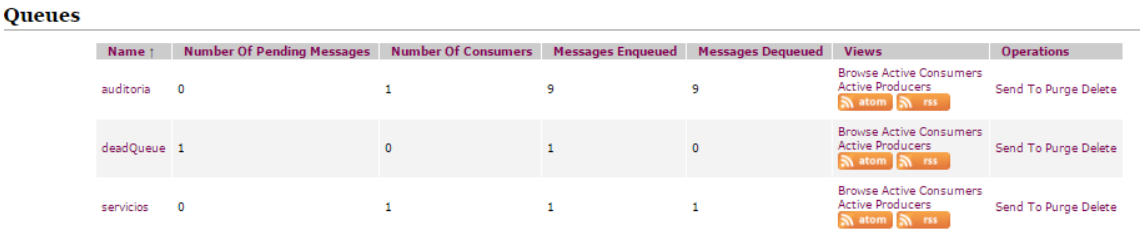

Ilustración 60: Colas JMS

En la Ilustración 60, se puede ver que actualmente existen tres colas, una para almacenar los mensajes de auditoría, que ha procesado 9 mensajes; una de las peticiones asíncronas (servicios), que ha procesado el mensaje de la petición que se acaba de realizar; y la *deadQueue* tiene un mensaje sin procesar. En esta ventana, además de ver las colas existentes y los mensajes en ellas, se pueden mandar mensajes a las colas y borrar estas con o sin mensajes, con las operaciones "*Send To*" y "*Delete*".

Si se pincha en la *deadQueue,* se pueden ver los detalles del mensaje almacenado.

Headers

Message ID	ID:Lepti-PC-54548-1476895182954-1:1:24:1:1
Destination	queue://deadQueue
Correlation ID	
Group	
Sequence	0
Expiration	0
Persistence	Non Persistent
Priority	1
Redelivered	false
Reply To	
Timestamp	2016-10-19 18:40:19:271 CEST
Type	

Properties

MULE_SESSION	rO0ABXNyACNvcmcubXVsZS5zZXNzaW9uLkRlZmF
Number_of_repeats	3
MULE_ROOT_MESSAGE_ID	b61563b0-961a-11e6-9e1e-bccf20524153
MULE_ENDPOINT	jms://deadQueue
messageId	b61563b0-961a-11e6-9e1e-bccf20524153
Priority	1

Message Actions

Delete

Copy

Move

Message Details


```
{
        "metadatos" : {
                "nombre_servicio" : "mandarEmailSub_Flow",
                "tipo_servicio" : "asincrono",
                "politica_errores" : {
                        "reintentos" : "3",
                        "prioridad" : "1"
                }
        },
        "datos" : {
                        "destinatario": "henarfgis@gmail.com",
                        "asunto": "prueba de correo",
                        "mensaje": "Este es el mensaje de prueba"
        }
}
```


Ilustración 61: Mensaje almacenado

Este mensaje se puede mover de cola, copiar a otra o incluso borrar de esta.

Para mandar un mensaje a una cola tan solo hay que pinchar en *Send* y aparece el menú con los datos necesarios para mandarlo.

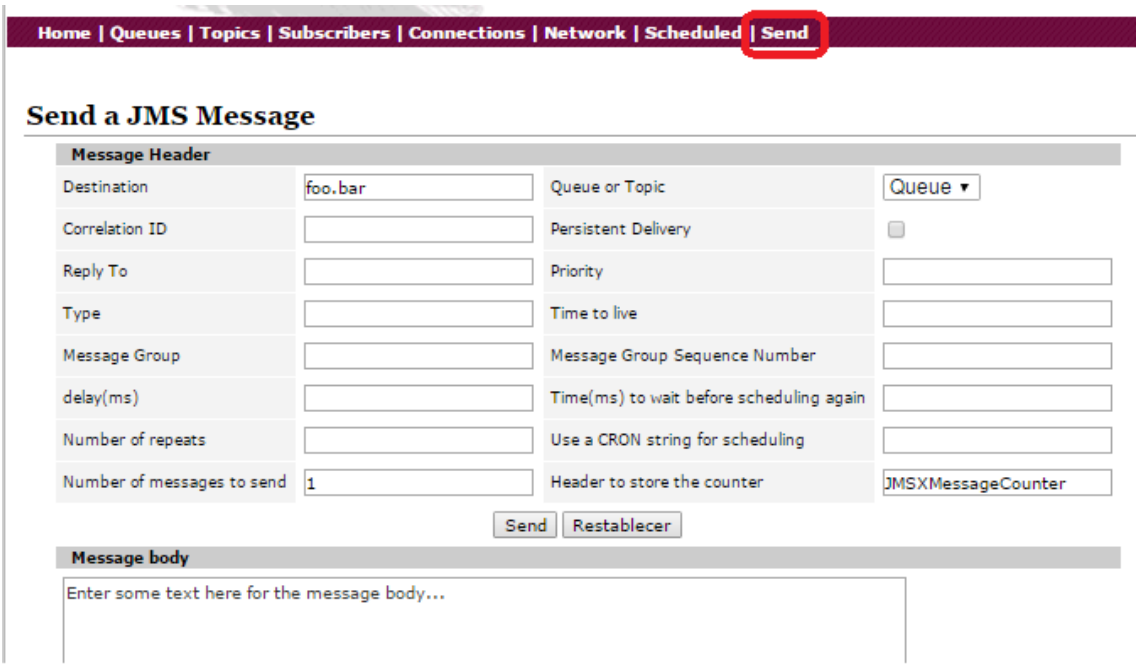

Ilustración 62: Ventana de mandar un mensaje

En este menú, además de introducir el mensaje, se puede indicar la cola o tema donde se almacenará y otros campos como, el número de repeticiones, la prioridad del mensaje, el tiempo de vida del mensaje en la cola, etc.

4.5 BASE DE DATOS

Para simplificar el uso de la base de datos y ver estos de forma gráfica, se usa la herramienta *Robomongo*. Una vez esta herramienta está conectada a la base de datos se pueden ver las colecciones existentes en ella.

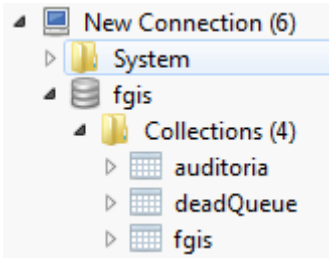

Ilustración 63: Colecciones base de datos

Si hace *click* en una de ellas, automáticamente hace una búsqueda de todos los datos existentes en dicha colección y para ver estos datos existen varios modos de vista, en forma de árbol, en forma de tabla y en forma de texto.

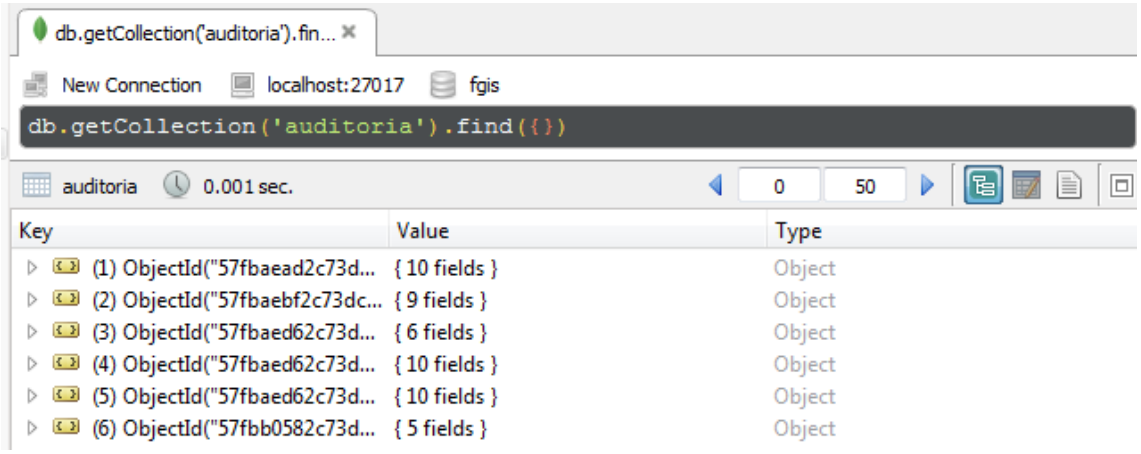

Ilustración 64: Vista forma de árbol

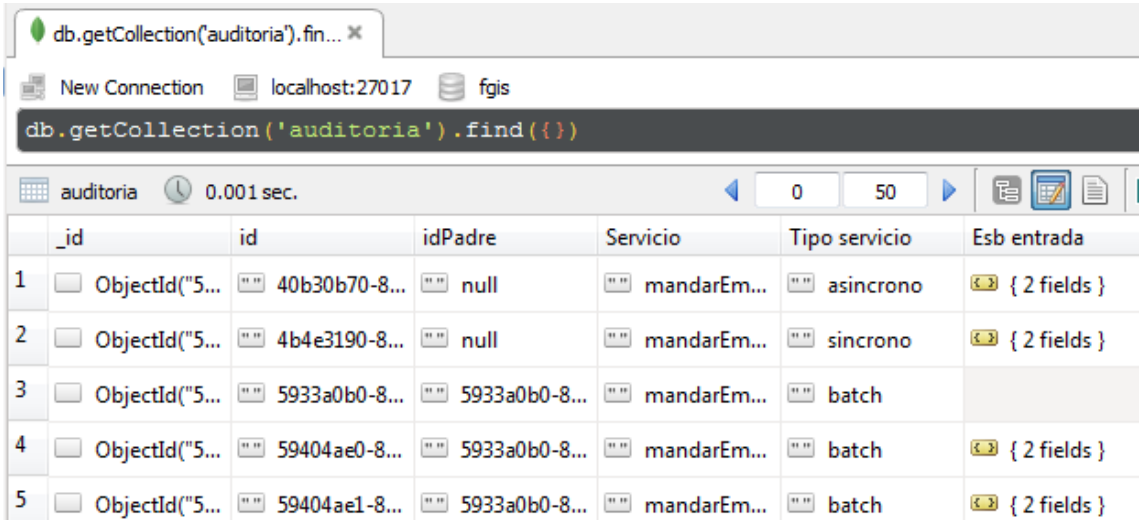

Ilustración 65: Vista forma de tabla

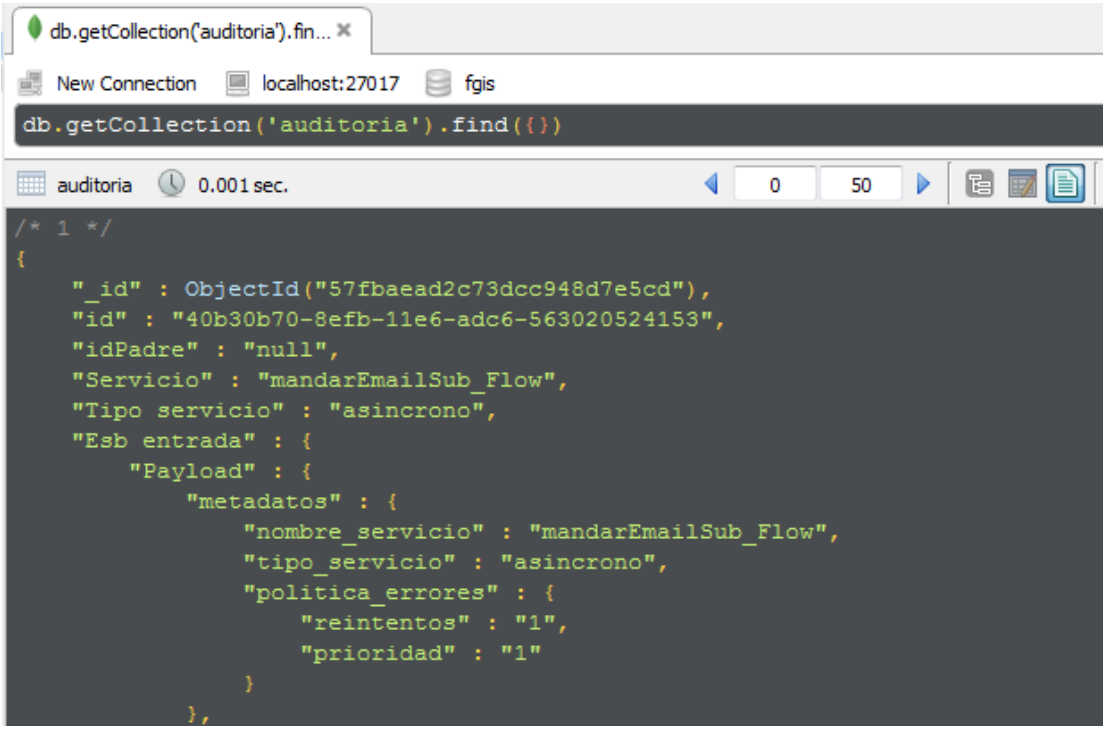

Ilustración 66: Vista forma de texto

Si en modo de vista en forma de árbol, se despliega uno de los objetos, se pueden ver cómodamente los datos almacenados en ese documento.

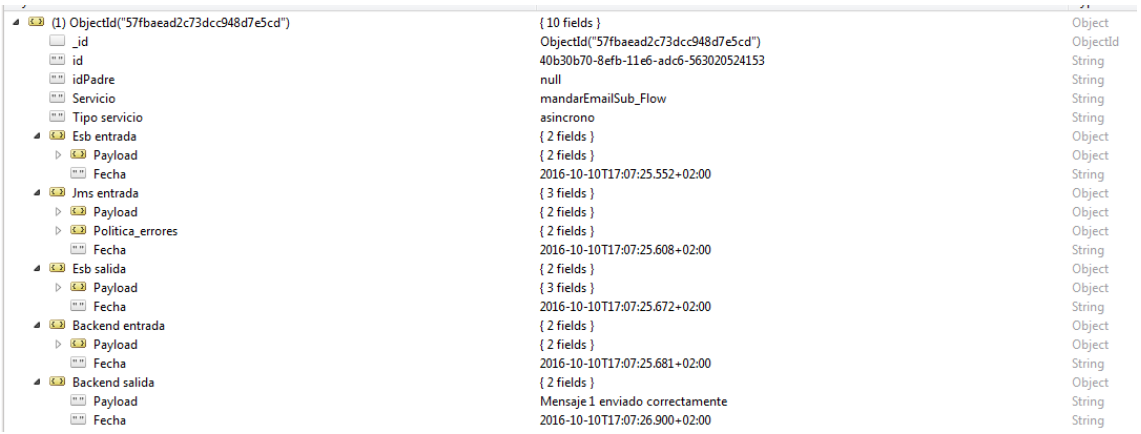

Ilustración 67: Datos del documento

4.5.1 BÚSQUEDA EN LA BASE DE DATOS

Para realizar una búsqueda especifica de un elemento en este tipo de bases de datos hay que ejecutar una *query* como la Instrucción 1.

```
db.getCollection('auditoría').find({"id" : "40b30b70-8efb-11e6-adc6-563020524153"})
```

Instrucción 1: Query búsqueda especifica

Después de *getCollection* hay que especificar la colección en la que se quiere hacer la búsqueda, a continuación, se ponen los datos a buscar, en el ejemplo anterior, se van a buscar todos los documentos cuya "id" sea "40b30b70-8efb-11e6-adc6-563020524153".

Este tipo de bases de datos tiene un motor de búsqueda mucho más potente. Debido a la estructura que suelen tener estos documentos, es común que dentro de un campo existan más campos, con lo que es posible hacer búsquedas en los campos dentro de campos de los documentos.

Para realizar una búsqueda de este tipo la *query* es similar a la anterior, como la Instrucción 2.

```
db.getCollection('auditoría').find({"Backend salida.Payload" : "Mensaje 1 enviado
correctamente"})
```

Instrucción 2: Query búsqueda especifica avanzada

En esta búsqueda se especifica inicialmente el nombre de la colección a buscar, y a continuación, se especifica que se busque "Mensaje 1 enviado correctamente" dentro del campo "*Payload*" que se encuentra dentro del campo "*Backend* salida".

CAPÍTULO 5

5. PLANIFICACIÓN Y PRESUPUESTO

Como todo proyecto propiamente dicho, antes de desarrollar este, es necesario hacer una estimación de la planificación y del presupuesto. En este apartado se va a explicar la planificación estimada al comienzo de este y con ello se va a determinar un presupuesto.

5.1 PLANIFICACIÓN

La planificación se va a separar en fases, cada fase va a tener unas tareas especificadas y unas horas previstas para su desarrollo.

- **Fase 0**
  - Toma de requisitos – 10h
  - Estudio de las soluciones de mercado – 5h
  - Definición de la arquitectura – 20h
    - Elección del software principal
    - Elección de las herramientas secundarias
  - Documentación – 10h

- **Fase 1**
  - Creación y configuración del proyecto *Mule ESB* – 10h
  - Desarrollo del servicio *backend* final piloto – 20h
  - Desarrollo del servicio de invocación base – 10h
    - Definición de los formatos de datos de entrada y salida
    - Transformación de datos de entrada y salida
  - Documentación – 10h

- **Fase 2**
  - Desarrollo de servicio de orquestación – 5h
  - Diseño de la *API* – 5h
    - Definición de archivo *RAML*
    - Definición de esquemas de entrada y salida de los servicios
    - Implementación de *APIkit*
  - Implementación de colas *JMS* – 25h
    - Implementación del servicio suscriptor del mensaje *JMS*
    - Implementación del servicio consumidor del mensaje *JMS* (extraccionColaServicio)
  - Documentación – 10h

- **Fase 3**
  - Definición de estrategia de auditoría – 10h
    - Definición de formato de mensajes de auditoría
    - Definición de la estructura de documentos para la base de datos
    - Definición de las banderas y la ubicación de estas
  - Configuración de base de datos *MongoDB* – 20h
    - Configuración de conector *MongoDB* en *Mule ESB*
  - Desarrollo del servicio de auditoría – 15h
  - Documentación – 10h

- **Fase 4**
  - Definición de la estrategia de gestión de errores – 10h
  - Implementación de la política global de errores – 20h
  - Implementación de la captura de excepciones de errores particulares – 15h
  - Documentación – 10h

- **Fase 5**
  - Documentación final – 50h
    - Manual de usuario
    - Anexos
    - Conclusiones
    - Líneas futuras

Una vez realizada la planificación, se puede ver que en total la estimación de horas es de 300h. Si se tiene en cuenta que el desarrollo de este proyecto se va a compaginar con clases podemos establecer que habrá una dedicación diaria de 3h. Si se hacen los cálculos se obtiene que la duración del desarrollo del proyecto será de 5 meses, 3 horas/día x 5 días/semana x 4 semanas/mes x 5 meses = 300 horas.

Toda esta planificación se puede ver de forma más grafica en la Tabla 1**¡Error! No se encuentra el origen de la referencia.**. En esta tabla se encuentran las tareas específicas de cada fase y en verde se puede ver en qué momento de los 5 meses de desarrollo se llevarán a cabo, finalizando el quinto y último mes con la tarea de la documentación final.

		Mes 1	Mes 2	Mes 3	Mes 4	Mes 5
Fase 0	Documentación					
	Estudio soluciones mercado					
	Definición de la arquitectura					
	Toma de requisitos					
Fase 1	Creación proyecto					
	Desarrollo *backend* final piloto					
	Desarrollo servicio de invocación base					
	Documentación					
Fase 2	Desarrollo servicio de orquestación					
	Diseño de la API					
	Implementación colas JMS					
	Documentación					
Fase 3	Definición estrategia de auditoria					
	Configuración base de datos					
	Desarrollo del servicio de auditoria					
	Documentación					
Fase 4	Definición de la estrategia de gestión de errores					
	Implementación de la política global de errores					
	Implementación de captura de errores particulares					
	Documentación					
Fase 5	Documentación final					

Tabla 1: Planificación

5.2 PRESUPUESTO

Una vez definida la planificación y el número de horas que va a llevar el proyecto, se puede hacer un presupuesto de este, pero previamente hay que analizar el material (ordenador y programas) que será necesario para desarrollar dicho proyecto.

El ordenador con el que se va a desarrollar, es un *MSI GE70 2PE Apache Pro* el cual tiene un valor actual en el mercado de 1099€ sin sistema operativo, a lo que hay que sumarle 146€ de la licencia de *Windows 7*, con lo que supone un total de 1245€. Este coste se va a tener en cuenta para calcular el gasto de depreciación que se puede incluir anualmente en la declaración de impuestos, este valor es del 25% del coste del portátil, con lo que deja 311,35€/año, si eso se calculamos para los 5 meses que durará el desarrollo del proyecto, se obtiene que el coste es de 129,69€, lo que se puede redondear a 130€.

Todo el software que se va a usar tiene licencias *open,* con lo que es completamente gratuito y por este lado no va a haber gastos de ningún tipo.

Debido a que el proyecto no se va a realizar en casa, se pueden estimar unos gastos de luz, internet, etc. de 5€ diarios, lo que hace 100€ mensuales.

Una vez definido esto, queda definir el salario del programador. Un programador con experiencia puede cobrar por hora entre 15 y 25 €, el salario de un programador junior sin experiencia puede descender aproximadamente hasta los 13€/h. Si se toma el precio de 13€/h como correcto y el proyecto va a llevar 300 horas, se calcula que el coste del programador es de 3900€.

Concepto	Coste
Coste programador junior	3.900,00 €
Portátil	130,00 €
Software	0,00 €
Costes de lugar de trabajo	500,00 €
TOTAL	**4.530,00 €**

Tabla 2: Presupuesto

Como se puede ver en la Tabla 2, la estimación final del presupuesto para el desarrollo este proyecto es de 4530€

CAPÍTULO 6

6. CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo se van a exponer una serie de conclusiones obtenidas tras realizar este TFG y las posibles líneas futuras que deja este a la hora de mejorar la aplicación desarrollada.

6.1 CONCLUSIONES

Hace tres o cuatro décadas, a los desarrolladores *software* se les llamaba artistas, debido a la dificultad que suponía desarrollar un programa, los cuales en ese entonces eran escasos. Sin embargo, ahora mismo cualquiera puede hacer un programa y la necesidad de estos ha hecho que este campo haya crecido a pasos agigantados. Se puede decir que hoy en día no se puede vivir sin tecnología, la cual siempre lleva un *software* para facilitar su uso. La evolución de estas tecnologías conlleva que siempre se necesiten nuevos *software*.

A lo largo de esta memoria, se ha hecho un estudio de las tecnologías actuales y una explicación de cuales se han elegido. Este puede parecer el paso más fácil, sin embargo, es el que más dificultad ha supuesto, ya que cuantas más tecnologías se tienen al alcance de la mano más difícil es elegir las adecuadas.

La idea de esta aplicación surgió debido a las prácticas en empresa que me encontraba realizando en la empresa EVERIS, en el equipo de servicios de arquitectura basados en soluciones *ESB*. A partir de ese momento comencé a desarrollar el proyecto desde cero, algo que no es tan sencillo como puede parecer. Inicialmente necesité hacer un estudio del proyecto que quería realizar, con las funcionalidades que iba a llevar y una planificación detallada de este. Gracias a lo cual se pudo estimar un presupuesto.

Una vez estaba realizada la planificación, llegó la hora de elegir las tecnologías. Esta elección resultó algo complicada, por el hecho de tener innumerables tecnologías para realizar cada función. El problema viene en que cuantas más hay, más opciones tienes y más cuesta elegir, ya que, muchas de ellas son igualmente buenas. Finalmente, conseguí decidirme por un grupo de ellas principalmente por ser gratuitas y su madurez en el mercado. Y al fin, llegó el punto en el que empezar realmente a desarrollar la aplicación.

A día de hoy, puedo decir que la planificación que hasta este momento nunca me había planteado hacer, me ayudó mucho a seguir un orden y unos plazos establecidos, terminando la aplicación en el tiempo estimado.

Por último, este TFG pone un punto y final a una etapa muy importante para mí que ha sido el estudio de este grado. Este proyecto me ha supuesto un reto, que se ha llevado muchas horas de esfuerzo y dedicación, pero finalmente se ha superado. A través de él, he aprendido mucho y he podido poner en practica algunos de los conocimientos adquiridos durante el grado.

Desde un punto de vista personal, he aprendido lo que es realizar un proyecto desde cero y desde el punto de vista de una empresa, algo durante el estudio del grado no te imaginas como es. Este TFG ha supuesto un aprendizaje continuo, no solo en el uso de tecnologías, sino también personal, como el trabajo autónomo a la hora de superar los problemas surgidos durante su desarrollo.

6.2 LÍNEAS FUTURAS

Cualquier aplicación funcional que existe en el mercado va a tener propuestas de mejora, por lo que la aplicación desarrollada en este TFG es lógico que también las tenga. A continuación, se van a explicar una serie de posibles futuras implementaciones, para mejorar la funcionalidad de esta aplicación.

- **Mejorar la funcionalidad de reintentos:** La funcionalidad de reintentos que está implementada esta en base a la captura de excepciones en *Mule ESB*. Cuando se produce un error, se captura y se reintenta la petición. Una mejora de esto sería volver a almacenar el mensaje en la cola, de manera que cuando llegue el turno de procesarlo es posible que el error que se produjera en la primera ejecución este solventado.

- **Cambio de tecnología de mensajería *JMS* por *AMQP*:** La tecnología usada de mensajería es *JMS*, una tecnología muy usada y muy madura. Sin embargo, su tecnología competidora, *AMQP*, a pesar de ser una tecnología relativamente nueva, ofrece funcionalidades que *JMS* no tiene y podrían resultar muy útiles para la aplicación.

- **Catálogo de servicios *backend* en base de datos:** Actualmente la aplicación solo tiene dos servicios *backend* final, los servicios piloto implementados. En el momento en el que esta aplicación tenga llamadas a más servicios, sería conveniente crear un catálogo en una base de datos con los nombres de los servicios *backend* y los datos que estos servicios requieren.

- **Implementación de funcionalidad de comprobación de los datos de entrada:** Actualmente no se realiza una comprobación de los datos que van a ir al *backend* antes de llegar a este, una mejora de la aplicación sería que en función del *backend* al que se llama, se hiciera una breve comprobación de los datos que se van a enviar.

BIBLIOGRAFÍA

AcensTechnologies. (2014). *Framework para el desarrollo ágil de aplicaciones*. Recuperado 17 de Octubre de 2016, de https://www.acens.com/wp-content/images/2014/03/frameworks-white-paper-acens-.pdf

AcensTechnologies. (2014). *Bases de datos NoSQL*. Recuperado 17 de Octubre de 2016, de https://www.acens.com/wp-content/images/2014/03/frameworks-white-paper-acens-.pdf

Apache ActiveMQ. (2016). *Introduction*. Recuperado 24 de Octubre de 2016, de http://activemq.apache.org/

Apache Maven. (2016). *What is Maven*. Recuperado 24 de Octubre de 2016, de https://maven.apache.org/what-is-maven.html

Barrera González, P. (2004). Sistemas de Control de Versiones. Universidad Rey Juan Carlos. Recuperado 17 de Octubre de 2016, https://gsyc.urjc.es/~barrera/docs/curso-svn-2004.pdf

Bass, L. & Clements, P. & Kazman, R. (2012). *Software architecture in practice.* Recuperado 17 de Octubre de 2016, de http://ptgmedia.pearsoncmg.com/images/9780321815736/samplepages/0321815734.pdf

Bhushan Agarwal, B. & Prakash Tayal, S. (2008). *Software Engineering*. Nueva Delhi: Firewall Media

Chacon, S. & Straub, B. (2014). *Pro Git.* Recuperado 17 de Octubre de 2016, de https://git-scm.com/book/en/v2

Erl, T. (2007). *SOA: Principles of Service Design*. Recuperado 17 de Octubre de 2016, de http://searchitchannel.techtarget.com/feature/Service-oriented-computing-and-SOA-explained

Fortune, S. (2014). *A Brief History of Databases*. Recuperado 17 de Octubre de 2016, de http://avant.org/project/history-of-databases/

IBM. (2015). *IBM Application Integration Suite.* Recuperado 25 de Octubre de 2016, de http://www-03.ibm.com/software/products/lv/application-integration-suite

IBM. (2014). *Características y tipos de bases de datos.* Recuperado 25 de Octubre de 2016, de https://www.ibm.com/developerworks/ssa/data/library/tipos_bases_de_datos/

Information Builders. (2014). *Integration suite.* Recuperado 17 de Octubre de 2016, de http://www.informationbuilders.es/products/integration/suite

MuleSoft. (2015). *APIKit Introduction.* Recuperado 24 de Octubre de 2016, de https://docs.mulesoft.com/apikit/

MuleSoft. (2015). *Integration solutions.* Recuperado 25 de Octubre de 2016, de https://www.mulesoft.com/resources/esb/mule-vs-camel-comparison

MuleSoft. (2015). *The Evolution of SOA.* Recuperado 17 de Octubre de 2016, de https://www.mulesoft.com/lp/whitepaper/soa/service-oriented-architecture

MuleSoft. (2015). *What is an ESB.* Recuperado 17 de Octubre de 2016, de https://www.mulesoft.com/resources/esb/what-esb

MuleSoft. (2015). *What is Mule ESB.* Recuperado 24 de Octubre de 2016, de https://www.mulesoft.com/resources/esb/what-mule-esb

Open Group. (2015). *Service Oriented Architecture*. Recuperado 25 de Octubre de 2016, de https://www.opengroup.org/soa/source-book/soa/soa_features.htm

Oracle. (2015). *A Relational Database Overview*. Recuperado 17 de Octubre de 2016, de https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

Oracle. (2015). *The Java EE 6 Tutorial*. Recuperado 17 de Octubre de 2016, de http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html

Peddie, J. (2013). *The history of visual magic in computers*. Londres: Springer.

RabbitMQ. (2008). *Advanced Message Queuing Protocol*. Recuperado 17 de Octubre de 2016, de https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

Riha Linik, J. (2016). Ada Lovelace: The First Computer Programmer. *iQ by Intel*. Recuperado 22 de Octubre de 2016, de https://iq.intel.com/ada-lovelace-the-first-computer-programmer/

Robomongo. (2016). *Introduction.* Recuperado 24 de Octubre de 2016, de https://robomongo.org/

Sol, S (2010). Types of Databases. The UK Web Design Company. Recuperado 25 de Octubre de 2016, de http://www.theukwebdesigncompany.com/articles/types-of-databases.php

Tiwari, S. (2011). *NoSQL*. Indiana: John Wiley & Sons.

Universidad de Alicante. (2014). *Introducción a JMS.* Recuperado 17 de Octubre de 2016, de http://expertojava.ua.es/j2ee/publico/mens-2010-11/sesion01-apuntes.html#Un+Poco+de+Historia

ANEXOS TÉCNICOS

1. *MULE ESB*

Mule es motor en tiempo de ejecución de la plataforma *Anypoint*. Es un *ESB*, con una plataforma de integración basada en *Java*, que permite a las aplicaciones el intercambio de datos de forma rápida y sencilla. Esta comunicación, se realiza independiente de los lenguajes que tengan los sistemas implementados. *Mule ESB* puede ser desplegado en cualquier lugar, ya que tiene conectividad universal y se puede integrar y orquestar los eventos en tiempo real o por lotes (*batch*) (MuleSoft, 2015).

Para el desarrollo de proyectos en *Mule ESB* se usa la herramienta *Anypoint Studio*.

1.1 INSTALAR Y ARRANCAR *ANYPOINT STUDIO*

Anypoint Studio, es un programa que no requiere instalación, tan solo hay que ir a su página de descargas https://www.mulesoft.com/lp/dl/studio, registrarse y descargarlo. Una vez descargado y descomprimido tan solo hay que ejecutar el archivo "*AnypointStudio*" y seleccionar la ubicación del *workspace*.

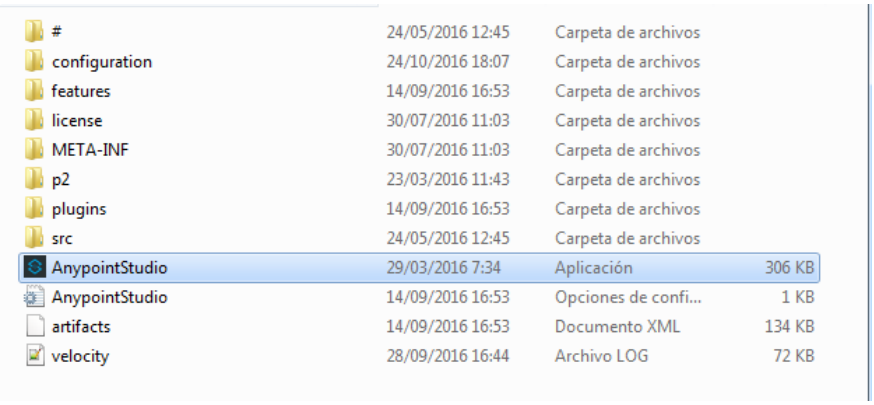

Ilustración 68: Carpeta Anypoint Studio

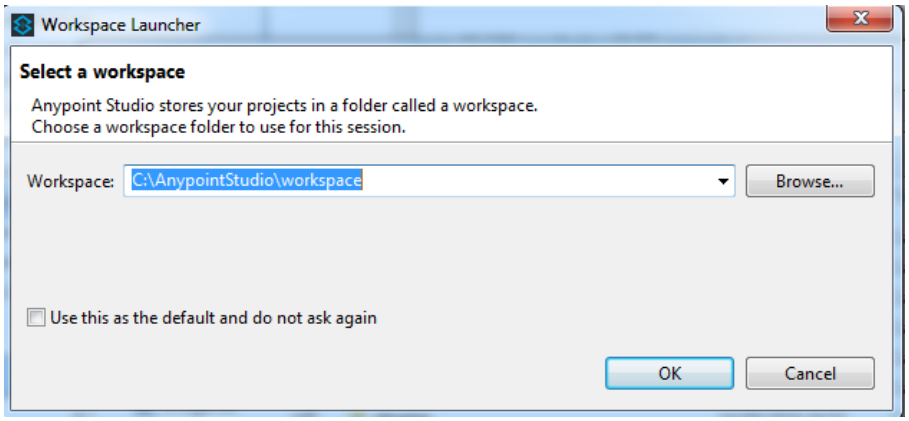

Ilustración 69: Workspace Anypoint Studio

Muchas veces a la hora de desarrollar una aplicación es necesario el uso de programas externos, para usar este tipo de programas hay que definir sus ubicaciones antes de arrancar *Anypoint Studio*, con lo que resulta más cómodo arrancar este desde una línea de comandos.

Para este TFG, es necesario definir la ubicación del programa externo *ActiveMQ*, usado para las colas *JMS,* del *JDK* (*Java SE Development Kit*) y de la herramienta *Maven*, una herramienta usada para la construcción y gestión de proyectos *Java* (Apache Maven, 2016).

Para definir las ubicaciones de estos programas tan solo hay que poner en una línea de comandos la Instrucción 3.

```
set JAVA_HOME=%RUTA% (Ubicación JDK)
set M2_HOME=%RUTA%(Ubicación Maven)
set ACTIVEMQ=%RUTA%(Ubicación ActiveMQ)

set PATH=%JAVA_HOME%\bin;%M2_HOME%\bin;%ACTIVEMQ%\bin;
```

Instrucción 3: Instrucciones ubicaciones programas externos

Y a continuación, ir a la ubicación de *Anypoint Studio* y ejecutar este, mediante la Instrucción 4.

```
AnypointStudio.exe
```

Instrucción 4: Instrucción ejecutar Anypoint Studio

1.2 CREAR UN NUEVO PROYECTO

Una vez arrancado el *Anypoint Studio*, para crear un proyecto hay que ir a la pestaña *File* → *New* → *Mule Project*

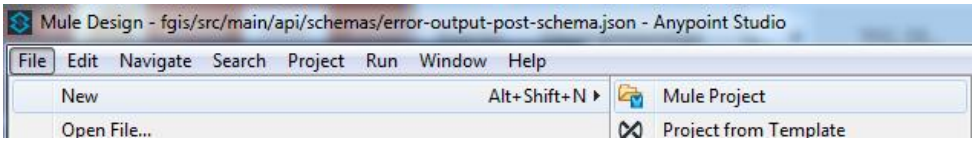

Ilustración 70: Nuevo proyecto Mule

En la siguiente ventana, como se puede ver en la Ilustración 70, se puede establecer el nombre del proyecto y seleccionar la versión de *Mule* deseada (*EE*, *Enterprise Edition*, versión de pago; *CE*, *Community Edition*, versión gratuita). A continuación, se puede elegir si usar *Maven* o no, y en caso de que se use, establecer el nombre del *Group Id, Artifact Id* y Versión. Después, da el soporte al controlador de versiones *Git* y a continuación permite establecer la configuración del *APIkit*.

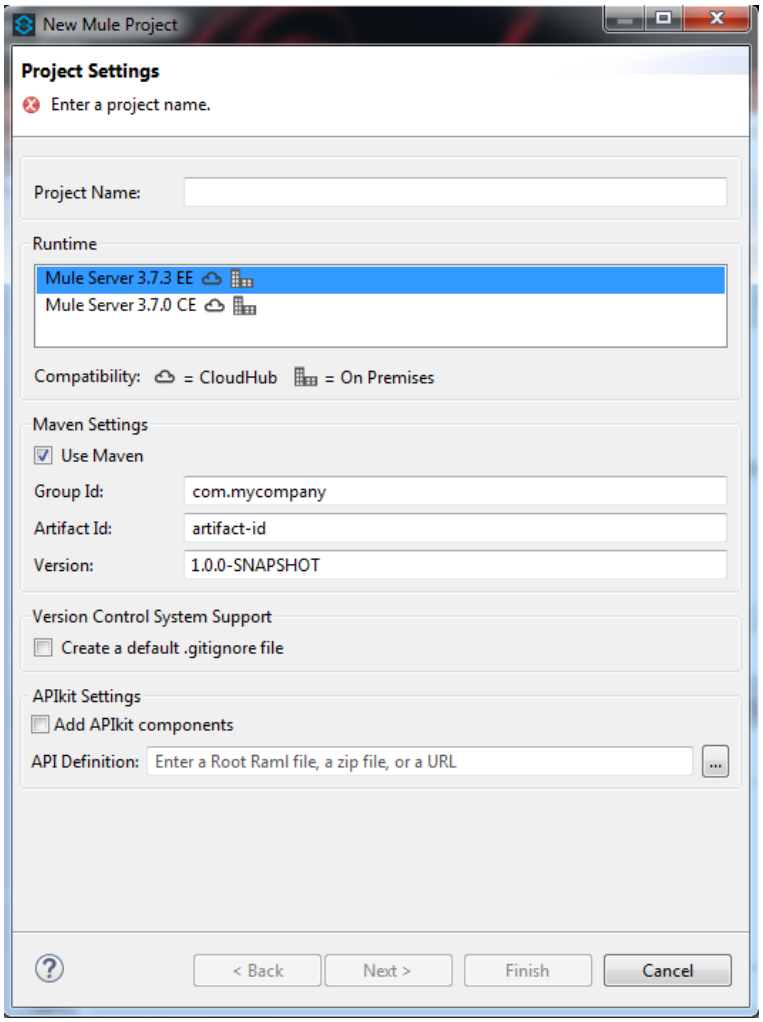

Ilustración 71: Configuración proyecto Mule

Una vez creado el proyecto, se puede ver la estructura de carpetas que sigue este, la cual se puede ver en la Ilustración 72. Las carpetas principales son:

- ***src/main/app*:** carpeta donde van a ir todos los flujos necesarios para el funcionamiento del proyecto y los archivos con las propiedades del proyecto.
- ***src/main/api*:** carpeta donde va a ir toda la configuración referente al *APIKit*, archivo *RAML* y archivos de esquemas y ejemplos.
- ***src/main/java*:** carpeta donde van los archivos *java* en los cuales se implementan funciones en *java*.
- ***src/main/resources*:** carpeta donde van otros archivos necesarios para el proyecto.

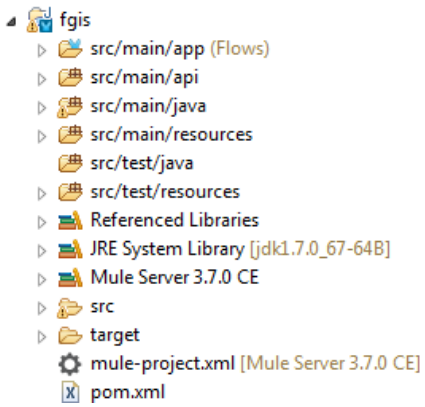

Ilustración 72: Estructura de carpetas proyecto Mule

1.3 EJECUTAR UN PROYECTO Y ASIGNAR VARIABLE DE ENTORNO

Una vez un proyecto esta desarrollado, para probarlo es aconsejable configurar algunos parámetros de configuración. Para ello se va a la pestaña *Run* → *Run Configurations*… y se selecciona la opción "*Mule Application with Maven*".

En la Ilustración 73, se puede ver que en la pestaña *Environmet* se puede definir una variable de entrono, muy útil a la hora de separar las variables en distintos archivos para los disantos entornos. Una vez definidas las variables deseadas para ejecutar el proyecto es suficiente con dar al botón *Run*.

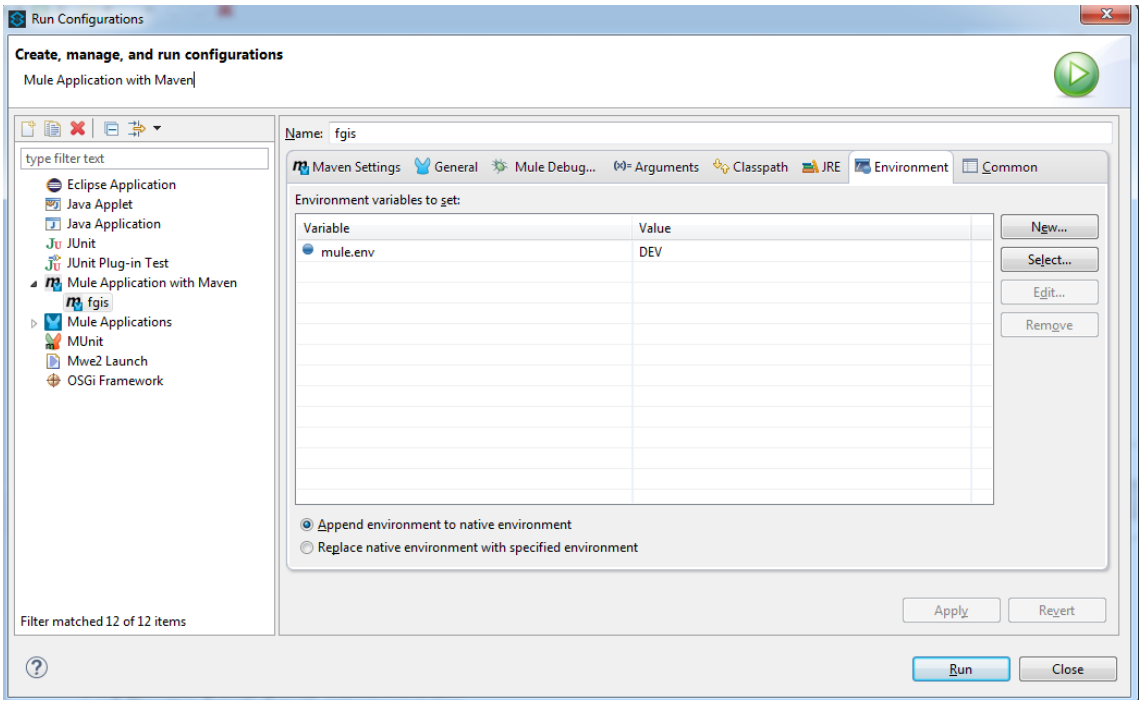

Ilustración 73: Variable de entorno

Una vez hecho esto en la consola se puede ver cómo va la ejecución y si el proyecto se ha desplegado correctamente.

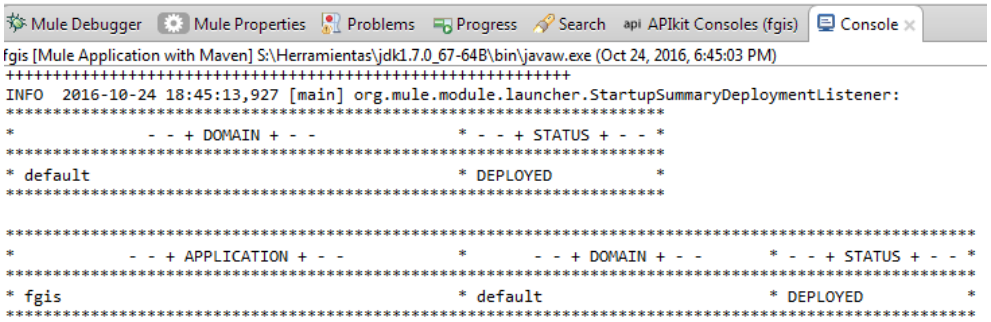

Ilustración 74: Consola ejecución

1.4 CONFIGURACIÓN DE *LOGS*

Anypoint Studio posee un archivo *logs* llamado *log4j2.xml* (*src/main/resources*) mediante el cual se pueden configurar los *logs* mediante el *framework log4j*.

En este archivo, mediante los *appenders,* se puede configurar la ubicación de los *logs*, así como el nivel de prioridad de trazas a escribir, incluso la configuración para que los *logs* roten o no.

```
<Appenders>
    <RollingFile name="servicios" fileName="c:${sys:file.separator}logs${sys:file.separator}fgis.log"
            filePattern="c:${sys:file.separator}logs${sys:file.separator}fgis-%i.log">
        <PatternLayout pattern="%d{ISO8601} %-5p [%c] %m%n" />
        <SizeBasedTriggeringPolicy size="10 MB" />
        <DefaultRolloverStrategy max="10"/>
    </RollingFile>

    <RollingFile name="audit" fileName="c:${sys:file.separator}logs${sys:file.separator}fgis-audit.log"
            filePattern="c:${sys:file.separator}logs${sys:file.separator}fgis-audit-%i.log">
        <PatternLayout pattern="%d{ISO8601} %-5p [%c] %m%n" />
        <SizeBasedTriggeringPolicy size="10 MB" />
        <DefaultRolloverStrategy max="10"/>
    </RollingFile>

    <RollingFile name="error" fileName="c:${sys:file.separator}logs${sys:file.separator}fgis-error.log"
            filePattern="c:${sys:file.separator}logs${sys:file.separator}fgis-error-%i.log">
        <PatternLayout pattern="%d{ISO8601} %-5p [%c] %m%n" />s
        <SizeBasedTriggeringPolicy size="10 MB" />
        <DefaultRolloverStrategy max="10"/>
    </RollingFile>
</Appenders>
```

Ilustración 75: Appenders log4j

Además de esto, se pueden establecer categorías, las cuales se usan en los componentes *logger* de *Anypoint Studio* para indicar el tipo de traza que es y en que *logger* se va a almacenar.

```
<Loggers>
    <logger name="FLUJOS_AUDIT" level="DEBUG">
        <AppenderRef ref="audit"/>
    </logger>

    <logger name="FLUJOS_SERVICIOS" level="DEBUG">
        <AppenderRef ref="servicios"/>
    </logger>

    <logger name="FLUJOS_ERROR" level="ERROR">
        <AppenderRef ref="error"/>
    </logger>

    <AsyncRoot level="DEBUG">
        <AppenderRef ref="consola" />
    </AsyncRoot>
</Loggers>
```

Ilustración 76: Configuración trazas log

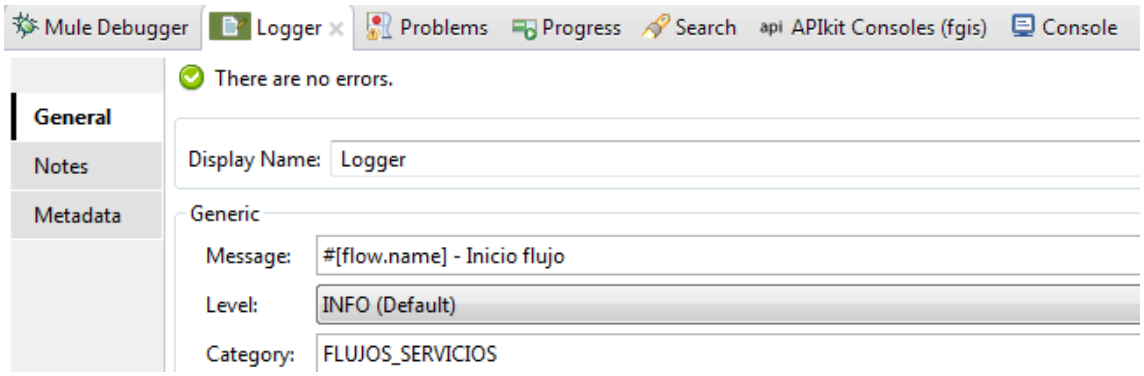

Ilustración 77: Componente Logger

1.5 *APIKit*

APIkit es un conjunto de herramientas de *Maven* (herramienta software para la gestión y construcción de proyectos *Java*) provisto por *Mule ESB* para crear una *API REST*, esta *API* se diseña con un archivo *RAML* (lenguaje basado en *YAML* para el diseño de peticiones *REST*). En este archivo, se pueden especificar las peticiones que va a contener la aplicación, el tipo de estas y sus esquemas y ejemplos de entrada y salida (Mulesoft, 2015).

Añadir el *APIKit* a un flujo es tan sencillo como arrastrar el componente *APIKit Router* al flujo en sí, una vez hecho esto, es necesario darle la ubicación del archivo *RAML*, previamente creado en *src/main/api*), que contiene la definición de la *API*.

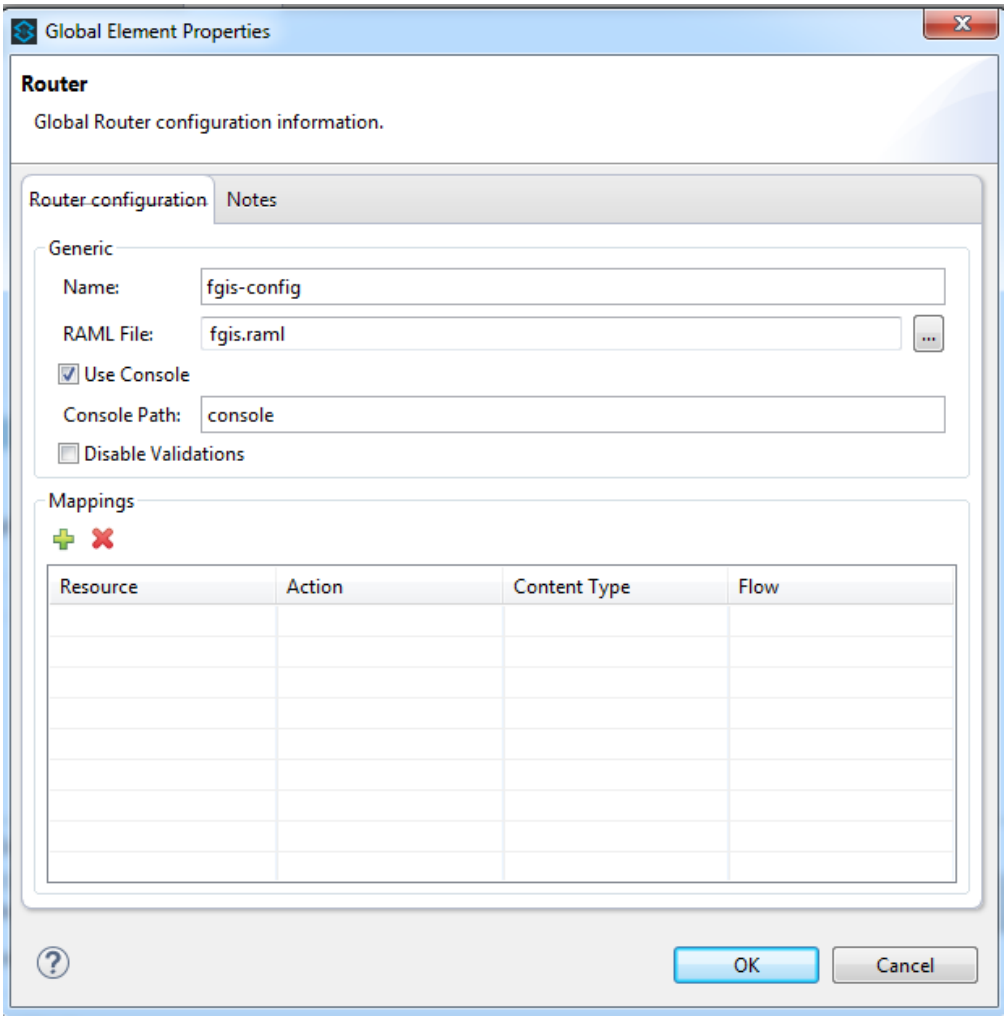

Ilustración 78: Configuración APIKit Router

En el archivo *RAML,* se especifican los flujos que va a tener la *API*, el tipo de petición y los campos de entrada y salida, esto último, se puede hacer en el propio archivo *RAML* o incluyendo los archivos externos de los esquemas y ejemplos de la entrada y salida.

fgis.raml

```
#%RAML 0.8
title: Framework de gestión de consumo de servicios
version: v1.0

baseUri: http://localhost:8081/api/console
documentation:
  - title: Getting Started
    content: |
      This sample API has been created as part of your APIkit project. It is located in src/main/api.
/comprobacionServicio:
    displayName: comprobacionServicio
    post:
      description: Comprobación del tipo de petición de un servicio (síncrono o asíncrono)
      body:
        application/json:
          example: !include examples/comprobacionServicio-input-post-example.json
          schema: !include schemas/comprobacionServicio-input-post-schema.json
      responses:
        200:
          body:
            application/json:
              example: !include examples/comprobacionServicio-output-post-example.json
              schema: !include schemas/comprobacionServicio-output-post-schema.json
```

Ilustración 79: Archivo RAML

Una vez definido el *RAML*, el propio *Anypoint Studio* crea los flujos definidos y un control de errores para comprobar que se cumplen los esquemas en los datos de entrada y salida.

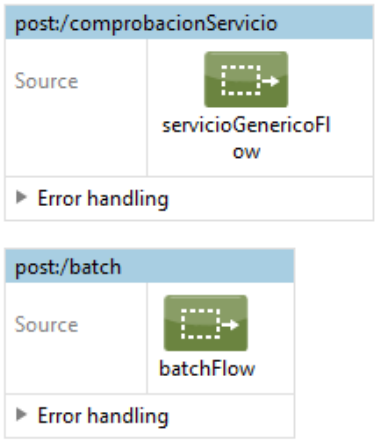

Ilustración 80: Flujos definidos en el RAML

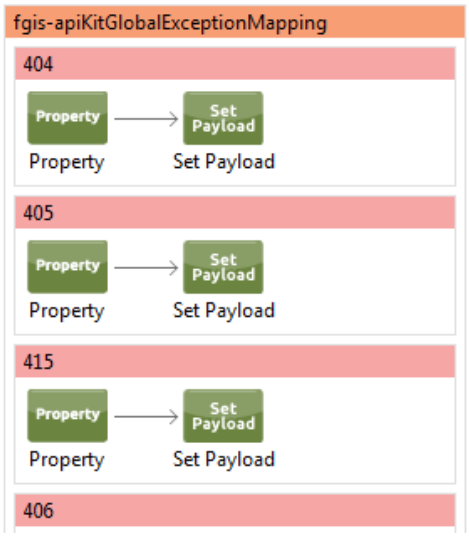

Ilustración 81: Excepciones APIKit

Una vez ejecutado un proyecto que posea esta *API*, se puede acceder a ella fácilmente desde cualquier navegador con la *URL*: http://localhost:8081/api/console/

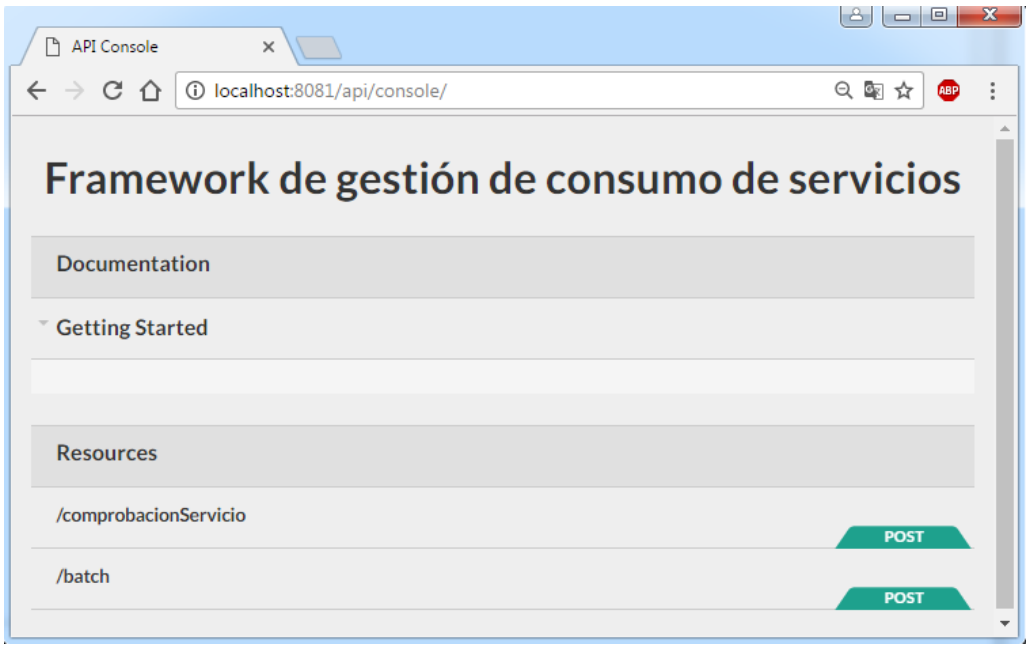

Ilustración 82: Consola APIKit

2. *APACHE ACTIVEMQ*

Apache ActiveMQ es una herramienta para el uso de mensajería de colas *JMS*. En esta sección se va a explicar brevemente el uso de la consola de administración de esta herramienta, para la visualización de los mensajes existentes en las colas (Apache ActiveMQ, 2016).

2.1 INSTALACCIÓN

Apache ActiveMQ, es un programa que no requiere instalación, tan solo hay que ir a su página http://activemq.apache.org/download.html, descargarlo y descomprimirlo en el lugar deseado.

2.2 EJECUTAR *ACTIVEMQ*

Es necesario ejecutarlo mediante línea de comandos, ya que hay que definir previamente la ubicación del *JDK*, mediante la Instrucción 5.

```
set JAVA_HOME=%RUTA% (Ubicación JDK)
set PATH=%JAVA_HOME%\bin;
```

Instrucción 5: Definir ubicación JDK

Para ejecutar el ActiveMQ tan solo hay que situarse en la ruta donde se ha descomprimido, una vez ahí, ir a la carpeta *bin* y ejecutar la Instrucción 6.

```
activemq start
```

Instrucción 6: Ejecutar ActiveMQ

2.3 VENTANA ADMINSITRACIÓN

Una vez arrancado *ActiveMQ,* se puede acceder a la ventana de administración desde cualquier navegador con la *URL*: http://localhost:8161/admin/ e introducir los datos de autentificación. Por defectos estos datos son, usuario "admin" y contraseña "admin".

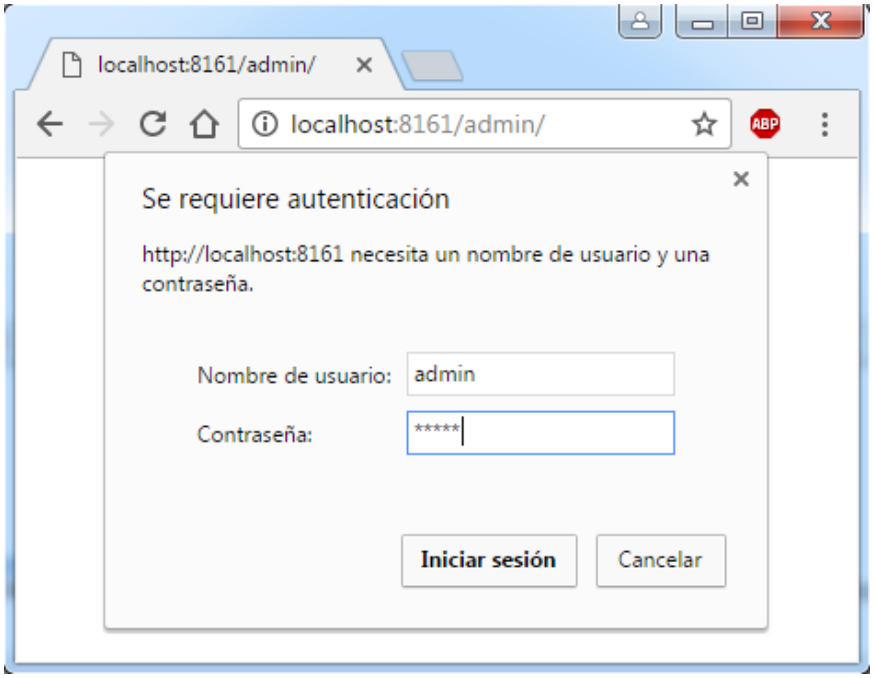

Ilustración 83: Autenticación ventana administración ActiveMQ

Ilustración 84: Ventana administración ActiveMQ

3. *MONGODB*

MongoDB es una de las más usadas tecnologías de bases de datos no relacional orientada a documentos.

3.1 INSTALACCIÓN

MongoDB, es un programa que no requiere instalación, tan solo hay que ir a su página https://www.mongodb.com/download-center?filter=enterprise#enterprise, descargarlo y descomprimirlo en el lugar deseado.

3.2 EJECUTAR MONGODB

Es necesario ejecutarlo mediante línea de comandos, para ello, tan solo hay que situarse en la ruta donde se ha descomprimido, una vez ahí, ir a la carpeta *bin* y ejecutar la Instrucción 7.

```
mongod.exe
```

Instrucción 7: Ejecutar MongoDB

3.3 ACCEDER A LOS DATOS DE *MONGODB*

Una vez ejecutado el programa podemos acceder a los datos de la base de datos de dos maneras, mediante comandos o con un programa externo.

3.3.1 ACCEDER A LOS DATOS MEDIANTE COMANDOS

Para acceder a los datos mediante comandos es necesario abrir otra ventana de comandos, que situarse en la ruta donde se encuentra *MongoDB*, una vez ahí, ir a la carpeta *bin* y ejecutar la Instrucción 8.

```
mongo.exe
```

Instrucción 8: Acceder a datos MongoDB

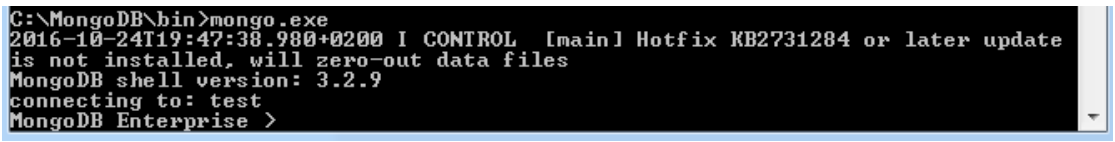

Ilustración 85: Acceso a datos de MongoDB mediante comandos

Una vez se ha accedido a la base de datos, tan solo hay que ejecutar los comandos para ver los datos de esta.

```
MongoDB Enterprise > show dbs
admin    0.000GB
fgis     0.000GB
local    0.000GB
pruebas  0.000GB
test     0.000GB
workers  0.000GB
MongoDB Enterprise > db.getCollection('workers').find({});
{ "_id" : ObjectId("57bf085c735cdc8bbb3dbe03"), "firstName" : "Carlos", "lastNa
es" : "Garcoa", "age" : 30 }
{ "_id" : ObjectId("57bf0af5735cdc8bbb3dbe04"), "firstName" : "Carlos", "lastNa
es" : "Garcoa", "age" : 30, "created" : ISODate("2016-09-24T22:00:00Z") }
{ "_id" : ObjectId("57d961dfba274009b817d1c2"), "content" : "pio pio pi" }
{ "_id" : ObjectId("57d96316ba27401d540f86c5"), "content" : "llllll" }
{ "_id" : ObjectId("57d9633bba274004485a4206"), "content" : "jjtgchv" }
MongoDB Enterprise >
```

Ilustración 86: Comandos y datos MongoDB

3.3.2 ACCEDER A LOS DATOS MEDIANTE *ROBOMONGO*

Robomongo, la cual es una herramienta más popular para ello, ya que fue la primera en mostrar los datos de forma asíncrona sin bloquear el hilo principal de la aplicación (Robomongo, 2016).

3.3.2.1 INSTALACCIÓN

Para instalar *Robomongo,* primero hay ir a su página de descargas https://robomongo.org/download y seleccionar la versión deseada. Una vez instalada o descomprimida la versión portable, se ejecuta el archivo *Robomongo* y se configura con la conexión con la base de datos. Los datos necesarios para esta conexión se pueden ver en la Ilustración 87, al acceder a una base de datos local la dirección es *localhost* y el puerto es el propio de *MongoDB*, 27017.

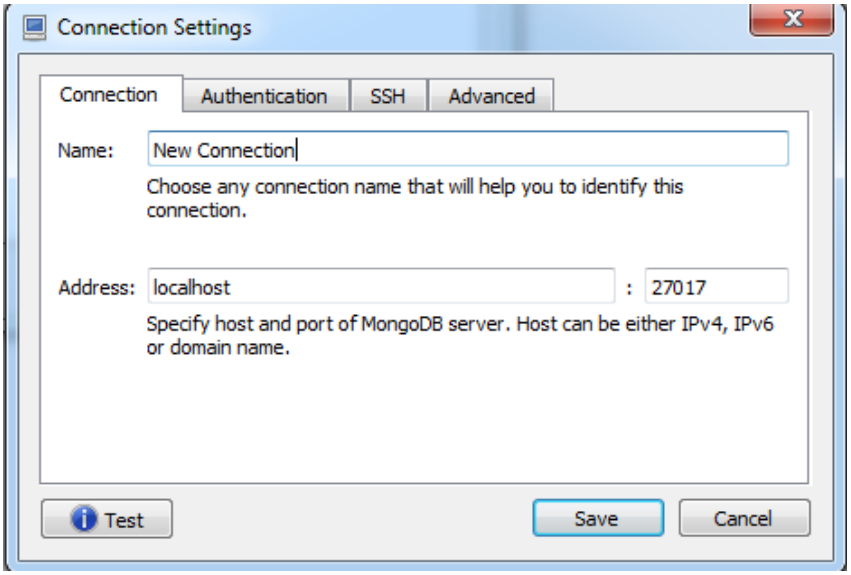

Ilustración 87: Conexión Robomongo – MongoDB

Una vez establecida la conexión con la base de datos, se obtienen los datos de esta, donde se pueden ver las bases de datos existentes y las colecciones de estas.

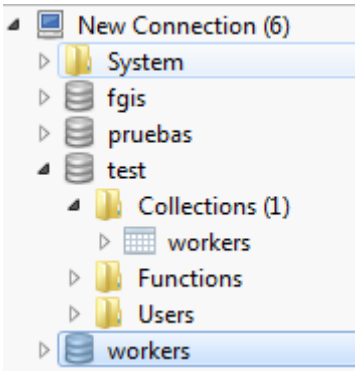

Ilustración 88: Bases de datos y colecciones

Para acceder a los datos de una de estas colecciones, tan solo hay que hacer doble click sobre una de ellas y el programa se encarga de ejecutar el comando necesario.

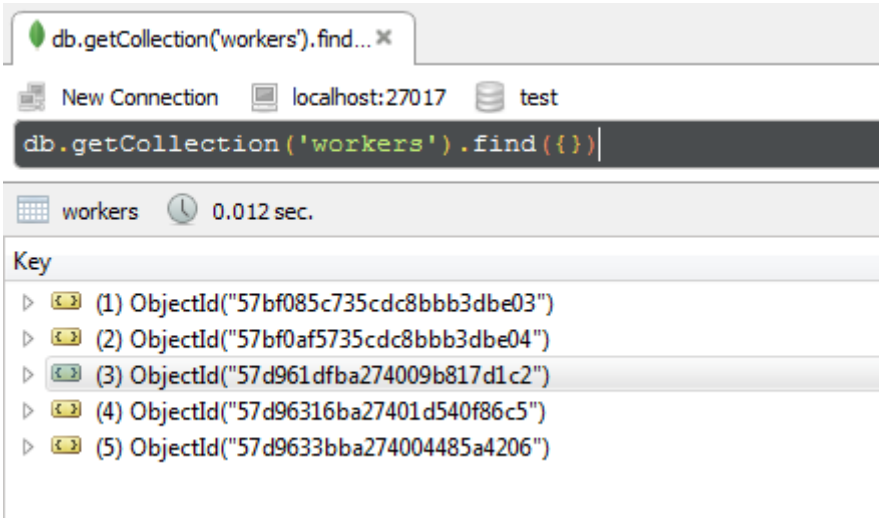

Ilustración 89: Datos Robomongo

Mediante esta herramienta se puede acceder a los datos de forma visual y fácilmente sin necesidad de usar comandos, ya que para las distintas acciones insertar datos, actualizar o borrar, la herramienta te da la estructura necesaria para cada instrucción.

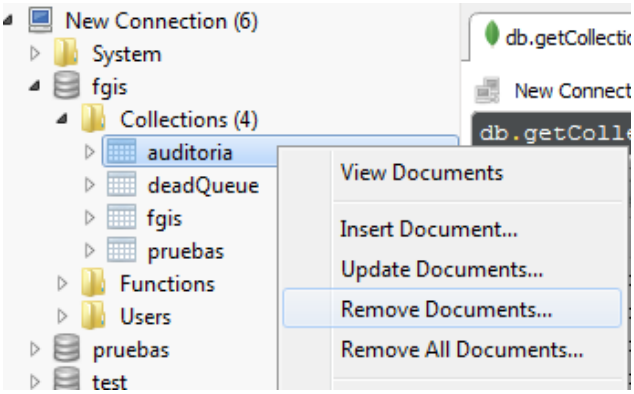

Ilustración 90: Acciones base de datos

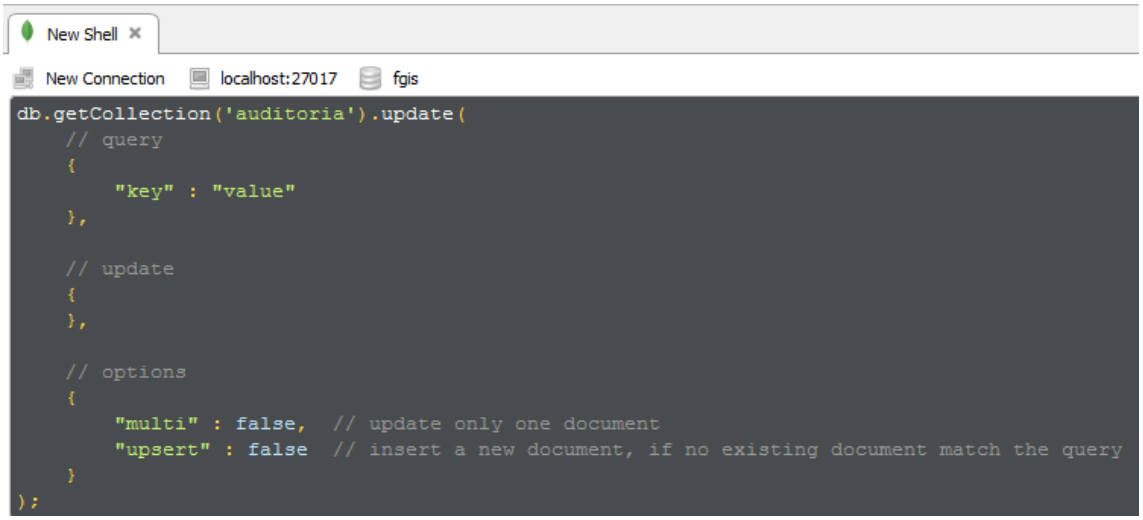

Ilustración 91: Estructura acción de actualizar