ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

GRADO EN INGENIERÍA TELEMÁTICA

IMPLANTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA EL CALIBRADO DE VOLATILIDADES

Autor: Pablo Iglesia Fernández-Tresguerres

Director: Atilano Fernández-Pacheco Sánchez-Migallón

Madrid

Junio 2018

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

"Implantación de una arquitectura de microservicios para el calibrado de volatilidades"

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2017/18 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Pablo Iglesia Fernández-Tresguerres Fecha: 18/06/2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Atilano Fernández-Pacheco Sánchez-Migallón Fecha: 11/07/2018

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D.__Pablo Iglesia Fernández-Tresguerres_____

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: __Implantación de una arquitectura de microsservicios para el calicrado de volatilidades que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente).*

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a ……11….. de …Julio……... de …2018

ACEPTA

Fdo……

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

GRADO EN INGENIERÍA TELEMÁTICA

IMPLANTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA EL CALIBRADO DE VOLATILIDADES

Autor: Pablo Iglesia Fernández-Tresguerres

Director: Atilano Fernández-Pacheco Sánchez-Migallón

Madrid

Junio 2018

IMPLANTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA EL CALIBRADO DE VOLATILIDADES

Autor: Iglesia Fernández-Tresguerres, Pablo.
Director: Fernández-Pacheco Sánchez-Migallón, Atilano.
Entidad Colaboradora: Banco Bilbao Vizcaya Argentaria, S.A. (BBVA)

RESUMEN DEL PROYECTO

El objetivo de este proyecto es estudiar y analizar las arquitecturas orientadas a los microservicios y, posteriormente, implantar un caso de uso real para realizar el calibrado de volatilidades en el banco **BBVA**.

Palabras clave: Microservicios, Aplicación, Plataforma, Volatilidad, FRTB

1. Introducción

Después de la crisis financiera (2007-2008), el Comité de Supervisión Bancaria de Basilea (BCBS) se centró en proporcionar las medidas y herramientas necesarias para mejorar la capacidad de respuesta del sistema bancario ante perturbaciones económicas y financieras. Por este motivo, en enero de 2016, publica la Revisión Fundamental del libro de Negociación (FRTB), que exige a los bancos reservar un capital mínimo en función al riesgo asumido en el mercado, que será calculado utilizando el método estándar revisado. Además, añade una serie de restricciones como la obligación de utilizar una ventana temporal de 10 años a la hora de realizar el calibrado de la volatilidad.

De esta forma, sumado a que los modelos matemáticos para realizar el cálculo de la volatilidad son muy complejos, con una arquitectura tradicional se tardaría más de medio día en realizar el calibrado de la volatilidad de un subyacente. Por este motivo, y dado que las grandes entidades financieras internacionales necesitan realizar el cálculo sobre más de 400 subyacentes, en este artículo se pretende realizar un estudio sobre la aplicación de una arquitectura de microservicios que permita paralelizar y escalar rápidamente los procesos para reducir al máximo el tiempo de computación de la aplicación de calibrado.

2. Definición del proecto

Teniendo en cuenta las restricciones impuestas por el **BCBS**, el objetivo del presente proyecto es la creación de una aplicación para realizar la calibración de la volatilidad de los activos subyacentes del **Banco Bilbao Vizcaya Argentaria (BBVA)**. Esta aplicación, como ya se ha comentado en la introducción, utilizará una arquitectura de microservicios que permita escalar de una forma flexible, así como separar la funcionalidad de la aplicación de tal forma que determinados microservicios puedan ser compartidos por distintas aplicaciones.

Este tipo de arquitecturas ha avanzado mucho en la teoría, pero aún no se ha reflejado en la práctica. Aunque la eficacia de los microservicios para mejorar muchos aspectos de las clásicas arquitecturas monolíticas es un hecho, también está suficientemente probado que añaden una complejidad extra a la hora de realizar el desarrollo.

Ante esta circunstancia, y dado que apenas se han desarrollado aplicaciones con arquitecturas de microservicios en el banco, otro de los objetivos del proyecto es comprobar la viabilidad de la utilización de una arquitectura de este tipo dentro de la industria financiera y su importancia en la trasformación digital.

3. Descripción del modelo/Sistema

El Sistema desarrollado consta de nueve microservicios que se comunican entre sí a través de diferentes métodos. Con el fin de realizar la comunicación asíncrona, de tal forma que la falta de disponibilidad de un microservicio concreto no afecte al rendimiento de otro, la comunicación entre los microservicios con más carga de trabajo se realican con colas **Tibco** de mensajes con prioridad.

Además, se ha desarrollado un interfaz web desde el que se podrán lanzar nuevas tareas de calibración y consultar los resultados de las mismas.

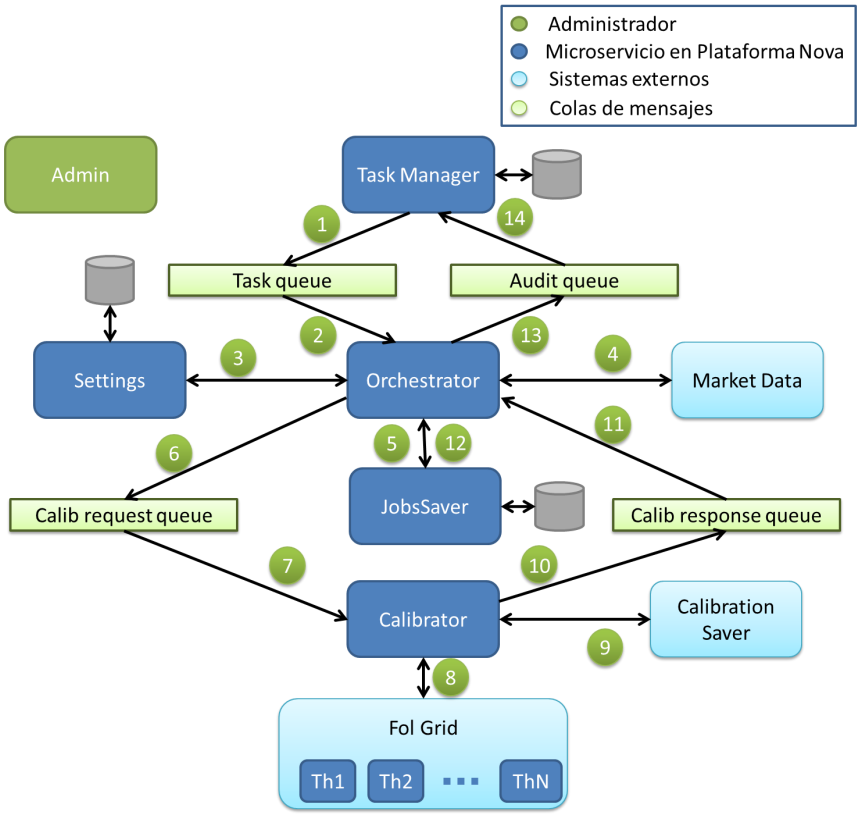


Ilustración 1: Estructura de microservicios utilizada

En la , se puede ver el flujo de ejecución de la aplicación, que está formada por los siguientes microservicios:

- **Task Manager:** Encargado de comunicarse con el exterior. En este caso recibe las peticiones realizadas desde el interfaz web y las deposita en la cola "**Task Queue**" en el formato apropiado.
- **Orchestrator:** Gestiona el funcionamiento de toda la aplicación. Recoge la tarea de calibración de la cola **"Task Queue**", recaba toda la información de calibración necesaria y, al final de su ejecución, deposita un mensaje en la cola "**Calibration Request Queue**" con la tarea segregada en tareas de calibración más simples.
- **Settings:** Encargado de recabar los ajustes de calibración de cada tarea. Estos ajustes son únicos para cada subyacente.
- **Market Data:** Encargado de recabar la información de mercado de cada subyacente en las fechas seleccionadas.
- **Jobs Saber:** Se encarga de dividir la tarea de calibración solicitada por el usuario en calibraciones de un subyacente en un día.
- **Calibrator:** Recoge las tareas simples de calibración depositadas en la cola "**Calibration Request Queue**" y se las envía al **Fol Grid** para que las realice. La respuesta obtenida la deposita en la cola "**Calibration Response Queue**".
- **Fol Grid:** Potente grid encargado de realizar las complejas tareas matemáticas de calibración. Recibe tareas simples.
- **Calibration Saver:** Este microservicio guarda el resultado de la calibración para que otros sistemas externos puedan acceder a ellos.
- **Admin:** Encargado de gestionar el correcto funcionamiento de la aplicación. Puede hacer un purgado selectivo de mensajes en las colas para cancelar una calibración.

4. Resultados

Los resultados obtenidos cumplen con las expectativas marcadas inicialmente. La aplicación es funcional y es capaz de realizar las tareas de calibración en un tiempo mínimo.

Además, aunque el hecho de haber utilizado una arquitectura de microservicios ha generado muchas complicaciones, sobre todo a la hora de optimizar el rendimiento, la aplicación no solo está optimizada para realizar las tareas de calibración complejas lo más rápido posible, sino que también es capaz de priorizar las calibraciones destinadas a ser consultadas en tiempo real para que estas se realicen lo más rápido posible, independientemente de si el sistema estaba realizando otras calibraciones.

En la ilustración 2 podemos ver el formulario de solicitud de calibración que se muestra en la interfaz web, junto con el resultado de calibración obtenido.

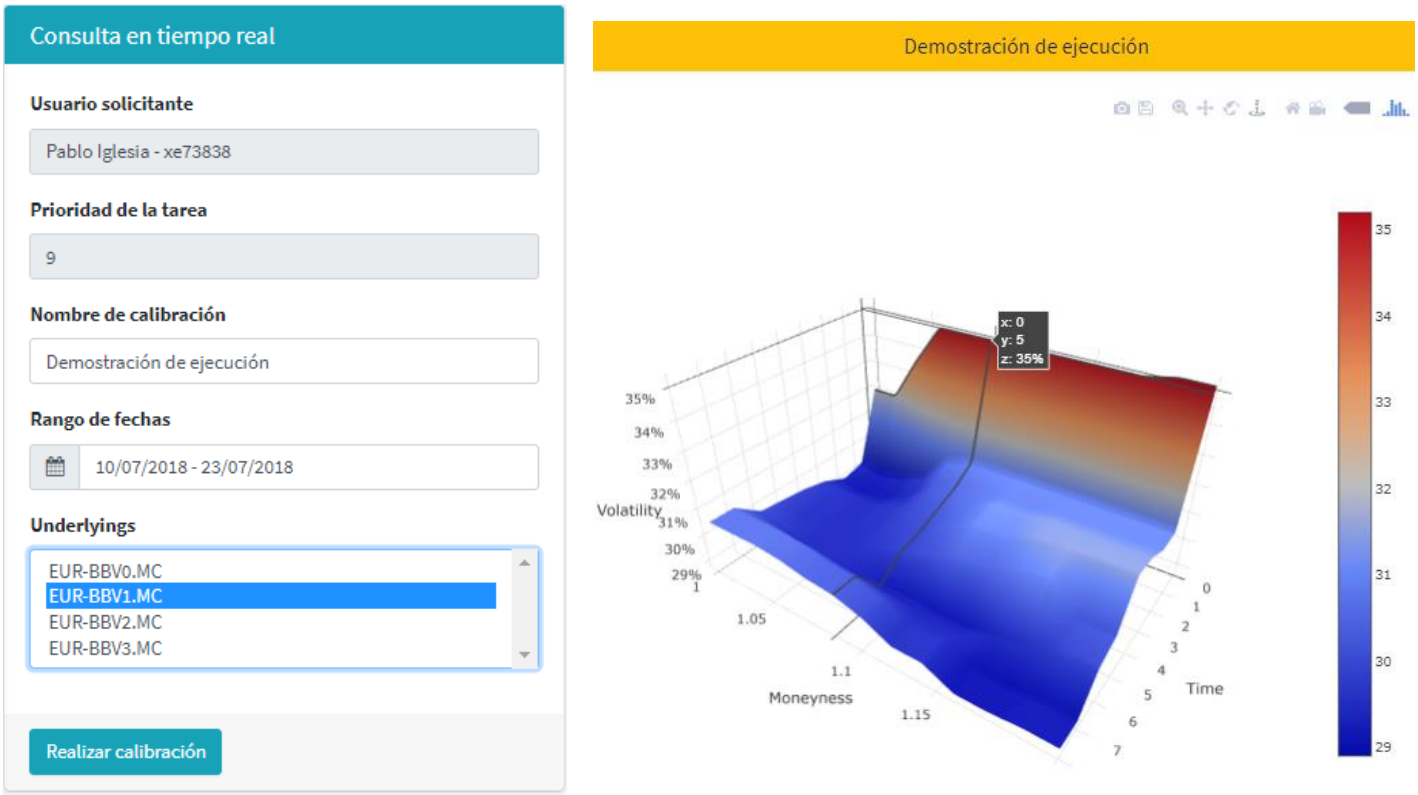


Ilustración 2: Formulario de envío de calibración junto con su resultado

5. Conclusiones

Después de todo el proceso de diseño e implementación de la aplicación, el resultado obtenido ha cumplido con los objetivos iniciales. La aplicación es funcional y realiza las tareas de calibración en un tiempo muy ajustado. Además, la arquitectura de microservicios utilizada ha permitido optimizar el rendimiento de la aplicación, facilitando, además, la escalabilidad del sistema.

Por otro lado, la arquitectura utilizada ha añadido mucha complejidad a la gestión de la aplicación y ha incrementado el coste de mantenimiento de la misma. Este mismo problema, salvo en casos excepcinales, se encontrará en todos los desarrollos realizados sobre una arquitectura de microserivicios. De esta forma, en la actualidad, las arquitecturas de microservicios no son válidas para todas las aplicaciones y es muy importante cuestionarse en cada caso concreto si los beneficios que aportan, que son muchos, realmente tienen el valor suficiente como para cubrir las complejidades y los sobrecostes que pueden introducir.

IMPLEMENTATION OF A MICROSERVICE ARCHITECTURE FOR THE CALIBRATION OF VOLATILITIES

Author: Iglesia Fernández-Tresguerres, Pablo.
Supervisor: Fernández-Pacheco Sánchez-Migallón, Atilano.
Collaborating Entity: Banco Bilbao Vizcaya Argentaria, S.A. (BBVA)

ABSTRACT

This proyect is focused on studying and analyzing microservices-oriented architectures and, subsequently, to implement a real-use case to calibrate volatilities in the BBVA bank.

Keywords: Microservices, Application, Platform, Volatility, FRTB

1. Introduction

After the financial crisis (2007-2008), the Basel Committee on Banking Supervision (BCBS) focused on providing the necessary measures and tools to improve the response capacity of the banking system in the face of economic and financial disturbances. For this reason, in January 2016, they publish the Fundamental Review of the Trading Book (FRTB), which requires banks to reserve a minimum capital based on the risk that they assume in the market. That risk will be calculated using the revised standard method, which adds a series of restrictions such as the obligation to use a 10-year time window when calibrating volatility.

In this way, taking into account that the mathematic calculations to perform volatility calibrations are very complex, a traditional architecture would take more than half a day to calibrate the volatility of an underlying asset. Therefore, as large international financial institutions need to perform the calibration of more than 400 underlying, this article intends to study the implementation of an microservice-architecture application that allows the bank to parallelize and quickly scale up the processes to reduce the maximum computation time of the calibration application.

2. Project definition

Taking into account the restrictions imposed by the **BCBS**, the objective of this project is the creation of an application to perform the volatility calibration of the BBVA bank's underlying assets. This application will use a microservice-architecture. That means that the application will be able to scale flexibly, and that its functionality will be divided in different microservices so that they could be shared by different applications.

The theory of this type of architecture has advanced a lot in the past few years, but it has not yet been reflected in practice. Although the effectiveness of microservices to improve many aspects of the classic monolithic architectures is a fact, it is also sufficiently proven that they add extra complexity when carrying out its development.

Therefore, as few projects with this type of architecture have been developed in the BBVA bank, another objective of the project is to verify the feasibility of using such an architecture within the financial industry and its importance in the transformation digital.

3. Description of the model / System

The developed system consists of nine microservices that communicate with each other through different methods. In order to perform asynchronous communication, so that the lack of availability of a specific microservice does not affect the performance of another, the communication between the high-loaded microservices is made through Tibco queues of messages.

In addition, a web interface has been developed from which new calibration tasks can be launched and the results of these can be consulted.

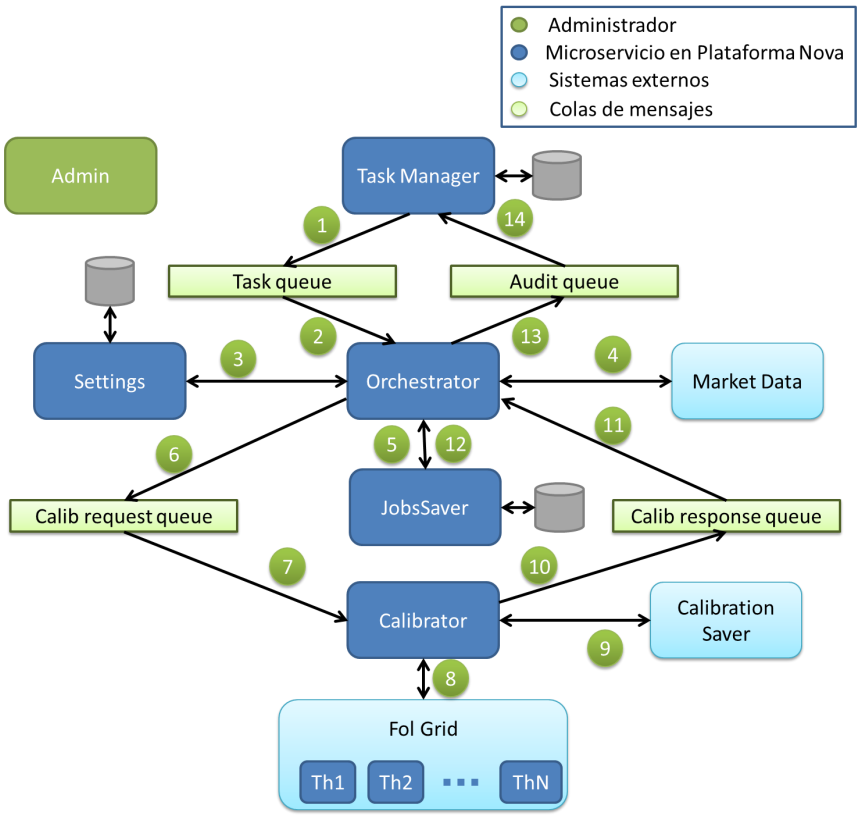


Ilustración 3: microservices arquitecture used

In **Figure 3**, we can see the execution flow of the application, where the following microservices come into action:

- **Task Manager:** Responsible for communicating with the outside world. In this case, it receives the requests made from the web interface and places them in the queue "**Task Queue**" in the appropriate format.
- **Orchestrator:** Manages the operation of the entire application. It collects the calibration task from the queue "**Task Queue**", collects all the necessary calibration information and, at the end of its execution, place a message in the

queue "**Calibration Request Queue**" with the task segregated in simpler calibration tasks.

- **Settings:** Responsible for collecting the calibration settings of each task. These settings are unique for each underlying.
- **Market Data:** Responsible for collecting the market data of each underlying on the selected dates.
- **Jobs Saber:** It is responsible for dividing the calibration tasks requested by the user into simpler calibrations of an underlying in a specific day.
- **Calibrator:** Collects the simple calibration tasks placed in the queue "**Calibration Request Queue**" and sends them to the **Fol Grid** for them to be performed. The calibration response is placed in the queue "**Calibration Response Queue**".

- **Fol Grid:** High-performance grid which is responsible for performing the complex mathematical calibration tasks. It receives simple calibration tasks.
- **Calibration Saver:** This microservice stores the result of the calibration so that other external systems can access them.
- **Admin:** Responsible of managing the correct operation of the application. It can selectively purge messages in the queues to cancel a calibration.

4. Results

The results obtained meet the expectations initially set. The application is functional and is capable of performing calibration tasks in a minimum time.

In addition, although using a microservice architecture has generated many complications, especially when it comes to optimizing performance, the application is not only optimized to perform complex calibration tasks as quickly as possible, but is also capable of prioritizing the calibrations intended to be consulted in real time so that they are carried out as quickly as possible, regardless of whether the system was performing other calibrations.

In Figure 4 we can see the calibration request form that is displayed in the web interface, along with the calibration result obtained.

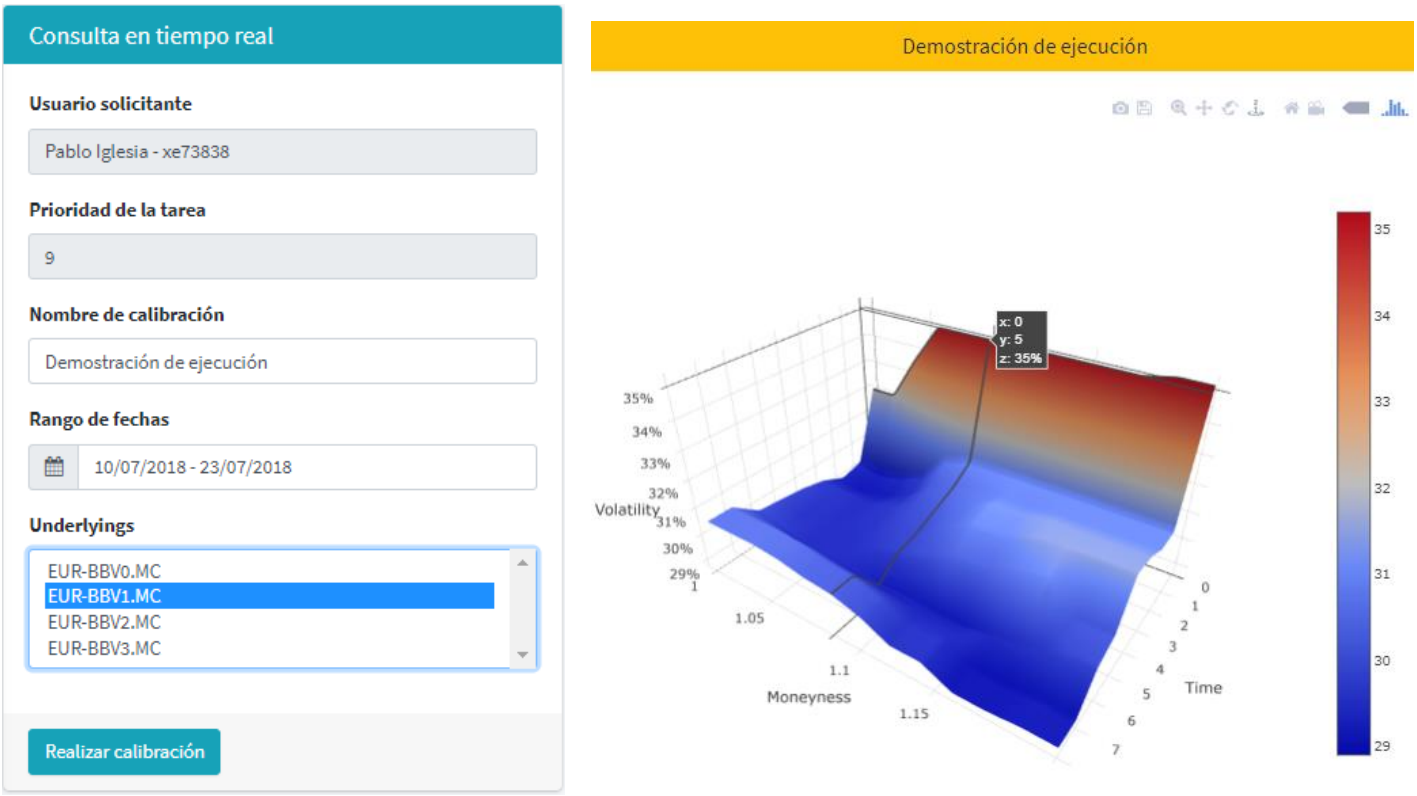


Ilustración 4: Calibration Task request and its results

5. Conclusions

After all the process of design and implementation of the application, the result obtained has met the initial objectives. The application is functional and performs calibration tasks in a very tight time. In addition, the microservice architecture used has made it possible to optimize the performance of the application, facilitating the scalability of the system.

On the other hand, the architecture used has added much complexity to the management of the application and has increased the maintenance cost of it. This same problem, will be found in all the developments carried out on a microservice architecture. Therefore, nowadays, microservices architectures are not valid for all applications and it is very important to ask ourselves, in each specific case, if the benefits they provide, which are many, really have enough value to cover the complexities and extra costs that can be introduced.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1: INTRODUCCIÓN

Después de la **crisis financiera (2007-2008)**, el **Comité de Supervisión Bancaria de Basilea (BCBS)** se centró en proporcionar las medidas y herramientas necesarias para mejorar la capacidad de respuesta del sistema bancario ante perturbaciones económicas y financieras y conseguir así una mayor estabilidad financiera mundial.

Dentro de este marco, el **BCBS** publicó en **enero de 2016** la última versión de la **Revisión Fundamental del Libro de Negociación** (**FRTB**)**,** que es un conjunto de reglas de capital de riesgo de mercado diseñadas para reemplazar una serie de parches introducidos después de la crisis financiera. Busca capturar mejor el riesgo de cola, redibujar el límite entre la banca y los libros de negociación, y elevar el nivel de los modelos internos [1]**.** Las normas serán de obligado cumplimiento para todas las entidades financieras a partir de **enero de 2019**.

El proyecto se realiza en el **Banco Bilbao Vizcaya Argentaria, S.A. (BBVA)** y consistirá en adaptar el proceso de calibrado de volatilidad de mercado a lo exigido por la **FRTB**, añadiendo, además, una serie de funcionalidades extra que se expondrán más adelante.

El calibrado de la volatilidad utiliza unos modelos que implican unos complejos cálculos matemáticos. Además, la **FRTB** exige incorporar una ventana temporal de información histórica de 10 años por cada subyacente, por lo que para realizar el calibrado se necesita una gran cantidad de computación.

Teniendo en cuenta que **BBVA** trabaja con un rango de entre 400 y 2500 subyacentes, se ha realizado una estimación previa de la cantidad de computación necesaria para ejecutar el proceso de calibración y se ha determinado que se emplearían 116 días para realizar el calibrado si este se realizara en un único proceso.

De esta forma, necesitamos una arquitectura que nos facilite la paralelización de procesos y la escalabilidad horizontal en función de la demanda de tal forma que podamos realizar una

aplicación flexible disminuya el tiempo de calibración repartiendo el proceso en diferentes hilos distribuidos en diferentes máquinas.

A su vez, también es necesario diseñar e implantar el software de forma modular para poder añadir nuevas funcionalidades a la aplicación sin afectar el funcionamiento del conjunto del software de tal forma que obtengamos una aplicación fácil de administrar y con un coste de mantenimiento relativamente bajo.

Para conseguir estos objetivos se utilizará una arquitectura orientada a microservicios. Este tipo de arquitecturas se basa en una descomposición del conjunto del software en pequeños bloques funcionales de tal forma que cada uno de estos bloques se ejecute de forma autónoma. Estos bloques funcionales, o microservicios, se comunican entre sí formando el conjunto de la aplicación y nos permitirá desarrollar el software de una forma más ágil y mucho más escalable.

Por otro lado, cada uno de estos microservicios estará alojado en un contenedor **Docker**. Los contendores **Docker** son ligeros y portables. Su despliegue es muy rápido y por tanto nos podremos realizar escalado horizontal de cada microservicio en tiempo real, ya sea en un mismo servidor o repartido entre varios. [2]

1.1. ESTRUCTURA DEL DOCUMENTO

Para la realización de este proyecto se ha querido seguir la estructura clásica de un documento técnico. Todas las secciones de este documento han sido debidamente numeradas y aparecen en el **Índice de contenido**.

Para facilitar la comprensión del documento, siempre que se habla sobre algún elemento que tenga una sección específica dentro de la memoria se incluye una referencia a dicha sección. Las referencias pueden corresponder a ilustraciones, tablas o secciones de contenido y estarán siempre subrayadas y en negrita.

Además, todos los nombres y fechas que se mencionen en el texto estarán representados en negrita con el objetivo de facilitar al lector la búsqueda de palabras clave dentro del texto.

Finalmente, es importante recalcar que la información recabada de otros documentos o artículos web está debidamente referenciada en formato **IEEE**. El conjunto de estas referencias se puede encontrar en el apartado de **Bibliografía.**

Capítulo 2: DESCRIPCIÓN DE LAS HERRAMIENTAS

En este capítulo se describirán las tecnologías utilizadas en el proyecto. Se incluirán únicamente las tecnologías más importantes ya que el conjunto de las herramientas utilizadas es muy amplio. Se incluyen también las tecnologías sobre las que corren los frameworks empleados ya que, aunque muchas de ellas no han sido utilizadas directamente, es importante conocerlas para entender el funcionamiento óptimo de la aplicación.

2.1. POSTGRESQL

PostgreSQL es un sistema de gestión de bases de datos relacional orientado a objetos y libre, publicado bajo la licencia **PostgreSQL**,1 similar a la **BSD** o la **MIT**.

Como muchos otros proyectos de código abierto, el desarrollo de **PostgreSQL** no es manejado por una empresa o persona, sino que es dirigido por una comunidad de desarrolladores que trabajan de forma desinteresada, altruista, libre o apoyados por organizaciones comerciales. Dicha comunidad es denominada el **PGDG** (**PostgreSQL Global Development Group**). [3]

2.2. LUCIDCHART

Lucidchart es una herramienta de diagramación basada en la web, que permite a los usuarios colaborar y trabajar juntos en tiempo real, creando diagramas de flujo, organigramas, esquemas de sitios web, diseños **UML**, mapas mentales, prototipos de software y muchos otros tipos de diagrama. [4]

2.3. JIRA

JIRA es una herramienta en línea para la administración de tareas de un proyecto, el seguimiento de errores e incidencias y para la gestión operativa de proyectos. La herramienta fue desarrollada por la empresa australiana **Atlassian**. Inicialmente **Jira** se utilizó para el desarrollo de software, sirviendo de apoyo para la gestión de requisitos, seguimiento del estado de desarrollo y más tarde para la gestión de errores. **Jira** puede ser utilizado para la gestión y mejora de los procesos, gracias a sus funciones para la organización de flujos de trabajo. [5]

2.4. SOURDETREE

Sourcetree es un cliente gratuito de **Mercurial** y **Git** para **Windows** y **Mac** que ofrece una interfaz gráfica para tus repositorios de **Hg** y **Git**. Esta herramienta nos permite hacer todo tipo de operaciones sobre **Git**, y sirve para simplificar la utilización de esta herramienta, ya que trabajar con la consola o con la **UI** integrada de **Git** es mucho menos intuitivo.

2.5. APACHE MAVEN

Maven es un software libre creado por la comunidad **Apache**. Sirve para simplificar los procesos de build, unificando en un mismo lugar las dependencias del proyecto. De esta forma, **Maven** nos permite correr el código siempre de la misma forma, independientemente de las dependencias que haya incluido otra persona del equipo. Con un único comando se puedan instalar dichas dependencias, compilar y ejecutar el código.

Pero **Maven** no es únicamente una herramienta dedicada a hacer builds de código, sino que gestiona el software completo. Con **Maven** se puede desplegar la aplicación, ejecutar pruebas y generar informes y documentación del proyecto.

A cambio, es necesario configurar un documento **XML,** por lo general llamado **pom.xml**, en el que se incluirá toda la configuración del proyecto de tal forma que el programa sea capaz de identificar y procesar cada una de necesidades del proyecto.

2.6. DOCKER

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de Virtualización a nivel de sistema operativo en **Linux.**

Al desplegar la aplicación en un contenedor, **Docker** permite encapsular todo el entorno de trabajo de tal forma que al llegar a producción, los desarrolladores tienen la garantía de que la aplicación se ejecutará en el mismo entorno en el que se han realizado las pruebas funcionales, evitando errores inesperados por cualquier cambio de configuración en el host de producción.

El tiempo de despliegue de una aplicación que está empaquetada en un contenedor **Docker** es muy bajo. Por otro lado, dado que comparten el mismo sistema operativo, los contenedores son portátiles entre diferentes distribuciones de Linux, y son significativamente más pequeños que las imágenes de máquinas virtuales (VM). [6]

2.7. JENKINS

Jenkins es un servidor de integración continua. Ayuda a automatizar los procesos del desarrollo del software facilitando técnicas de entrega continua. Corre sobre contenedores servlet como **Apache Tomcat** y soporta herramientas de control de versiones como **Git**.

Puede ejecutar proyectos con estructura **Apache Maven** y estas ejecuciones pueden ser lanzadas de varias formas, desde el commit del sistema de control de versiones hasta una ejecución programada o ejecutada al acceder a una determinada url.

2.8. SPRING FRAMEWORK

Spring es un framework de **java** creado como resultado de la evolución de la ingeniería del software. Su principal función es actuar de "core" de la aplicación para integrar los distintos frameworks utilizados dentro de la misma. De esta forma, administra las dependencias funcionales del proyecto para que la integración de estas sea más limpia y se eviten errores.

Spring ahorra líneas de código evitando repeticiones, aplicando así una buena práctica de programación. También facilita la creación de aplicaciones más autónomas y con funcionalidades más encapsuladas.

Existen multitud de aplicaciones desarrolladas por la comunidad sobre la tecnología **Spring**. A continuación, se explican algunas de las utilizadas en el proyecto.

2.8.1. SPRING BOOT

Spring Boot es una de las tecnologías más utilizadas dentro del mundo **Spring**. Su misión es reducir al máximo las tareas de despliegue de los proyectos de tal forma que los desarrolladores puedan centrarse únicamente en el desarrollo del software.

Spring Framework es muy útil a la hora de desarrollar proyectos por los motivos anteriormente expuestos. Sin embargo, a veces, puede resultar tedioso crear y configurar un proyecto **Spring**. De esta forma, la funcionalidad básica de **Spring Boot** es ofrecer una capa de personalización extra sobre **Spring Framework** para simplificar al máximo su utilización.

A su vez, incluye **Spring Initializr** que nos permite seleccionar el tipo de aplicación que queremos desarrollar y crea automáticamente una aplicación **Spring Boot** con las dependencias necesarias para ese tipo de proyecto. El proyecto creado es un proyecto **Gradle** o **Apache Maven**, aunque para este proyecto utilizaremos esta última.

2.8.1.1. Spring Tool Suite

Spring Tool Suite (**STS**) es un **entorno de desarrollo Integrado** (**IDE**) basado en **Eclipse**. Combina las herramientas existentes de **Java**, Web y **Java EE** con herramientas integradas para facilitar el desarrollo de software **Spring**.

STS nos facilita implementar, depurar, ejecutar y desplegar las aplicaciones **Spring**. Incluye un editor de texto integrado en el que el código se compila automáticamente según escribimos, y nos permitirá ejecutar el proyecto en su conjunto con un solo click.

Permite también integrar el **Entorno Local** de la **Plataforma Nova** en el IDE de tal forma que podremos utilizar todas las herramientas necesarias para desarrollar y probar el software localmente dentro de un mismo entorno de desarrollo.

2.8.1.2. Spring Cloud Netflix

Spring Cloud es un conjunto de herramientas utilizado para construir algunos patrones comunes de sistemas distribuidos. Aporta distintas herramientas y permite a los desarrolladores levantar rápidamente este tipo de sistemas.

Algunas de las herramientas utilizadas provienen de **Netflix Open Source Service** (**OSS**), que es la arquitectura de microservicios que utiliza la conocida aplicación de streaming de video para alojar su sistema. Dos de las herramientas más importantes son los siguientes:

Netflix Eureka

Servidor de "**Service Discobery**" que permite a los microservicios registrarse según se van creando en el sistema. En otras palabras, **Eureka** lleva la cuenta de los microservicios existentes en el sistema y su localización, creando, así, un mapa completo de la arquitectura lógica del sistema.

Netflix Zuul

Su función es hacer de API Gateway del sistema. Se encarga de permitir o denegar el acceso a los microservicios y actúa como único punto de entrada que redirige el tráfico al microservicio correspondiente.

Actúa de enrutador y se apoya en **Netflix Eureka** para conocer la información de los microservicios disponibles.

2.9. PLATAFORMA NOVA

La **Plataforma Nova** es una plataforma de desarrollo, gestión y despliegue de aplicaciones internas del banco **BBVA**. Ha cambiado la forma de gestionar el ciclo de vida de las aplicaciones en CIB mediante una arquitectura de componentes orientada a los microservicios. Ofrece integración con componentes de baja latencia, integración con otros productos y computación intensiva y facilita el desarrollo y despliegue de aplicaciones de una forma ágil y flexible.

Cómo es de esperar, la **Plataforma Nova** tiene algunos puntos de mejora. Aunque están trabajando en incorporarlo, en la actualidad no permite escalar el software horizontalmente en función de la demanda. Sin embargo, ofrece una política de mantenimiento y monitorización del software, así como seguridad a nivel de plataforma que ha hecho que nos decantemos por esta plataforma antes que otras alternativas como AWS.

Como se puede ver en la **Ilustración 5**, la plataforma se basa en la tecnología **Spring Cloud Netflix**. De esta forma, **Netflix Zuul** realizará la función de enrutar todas las peticiones externas e internas y administrar los permisos de tal forma que toda petición autorizada pueda acceder al microservicio correspondiente, mientras que **Netflix Eureka** se encargará de conocer en todo momento el estado y la ubicación de cada uno de los microservicios, que tendrán que registrarse en **Netflix Eureka**.

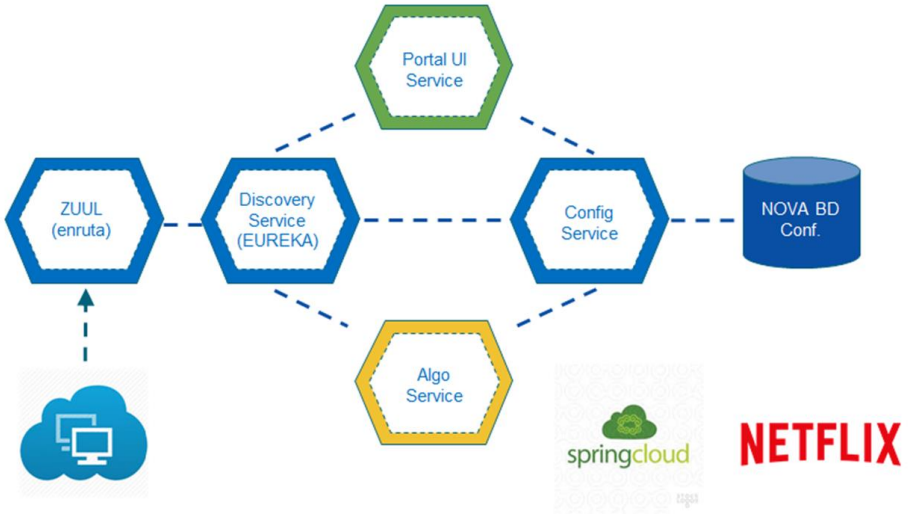


Ilustración 5: Arquitectura simplificada de la Plataforma Nova

Por otro lado, en la **Ilustración 6** podemos ver el flujo de trabajo de la integración continua dentro de la **Plataforma Nova**. Como se puede observar, **Jenkins** lee el código del repositorio **git** indicado. Cómo ya habíamos indicado, los microservicios se desplegarán en contenedores **Docker** distribuidos en un cluster de servidores. Para ello, la imagen del microservicio se almacenará en **Docker Registry** mientras que **Docker Swarm** se encargará de desplegar y orquestar las distintas imágenes del microservicio en el cluster.

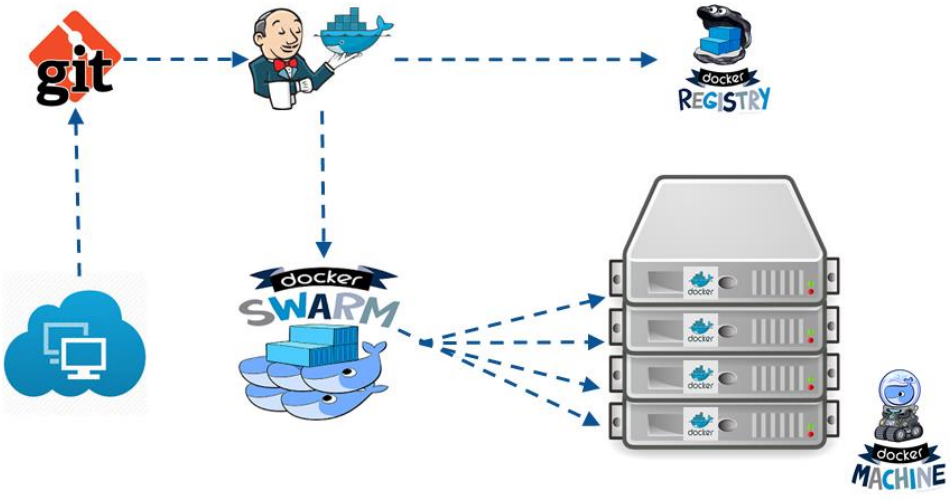


Ilustración 6: Integración continua en la Plataforma Nova

2.9.1.1. Entorno Local

Aunque se ha mencionado anteriormente que la plataforma Nova utiliza un servidor de integración continua como **Jenkins**, la plataforma está ideada para albergar aplicaciones en producción y no para desarrollar directamente sobre ella. Por supuesto, la plataforma incluye un entorno de preproducción e incluso un entorno previo, el entorno integrado, pero estos entornos están pensados para probar el software en su conjunto antes de subirlo al entorno definitivo de producción.

Por este motivo, existe un entorno local de la plataforma Nova disponible para instalarlo en los ordenadores locales. Este entorno proporciona una arquitectura similar a la de la plataforma, pero ejecutada en una única máquina. De esta forma, podremos desarrollar y probar el software localmente antes de desplegarlo en la plataforma.

Capítulo 3: ESTADO DE LA CUESTIÓN

3.1. COMPARACIÓN DE LOS MICROSERVICIOS CON EL ENFOQUE TRADICIONAL MONOLÍTICO

Uno de los mayores problemas existentes en el sector bancario es que las aplicaciones desarrolladas en el pasado están comenzando a quedar obsoletas. Las aplicaciones ya no son capaces de escalar al ritmo que exige el mercado y las empresas de este sector no están siendo capaces de migrar estas aplicaciones a un modelo tan flexible y escalable como son los microservicios. Esto es debido a que las aplicaciones no han sido diseñadas para desagregar su funcionalidad, sino que son un conjunto de millones de líneas de código monolíticas con funcionalidades entrecruzadas.

Por esta razón, una buena forma de entender los microservicios es comparar este tipo de arquitecturas con las arquitecturas tradicionales que las empresas venían utilizando en el pasado.

Esta diferencia está muy bien reflejada en la **Ilustración 7**, donde, a parte de las diferencias entre las arquitecturas, se puede apreciar también alguna de las ventajas o desventajas de una frente a la otra. Una aplicación monolítica reúne toda su funcionalidad en un único proceso, mientras que una arquitectura de microservicios distribuye cada elemento funcional de la aplicación en diferentes microservicios que se comunican entre sí. Como se puede apreciar en la **Ilustración 7**, los microservicios pueden estar distribuidos en máquinas separadas, y cada máquina no tiene por qué disponer de la funcionalidad completa de la aplicación, sino que, para poder obtenerla, se debe de comunicar con el resto de los componentes de la aplicación.

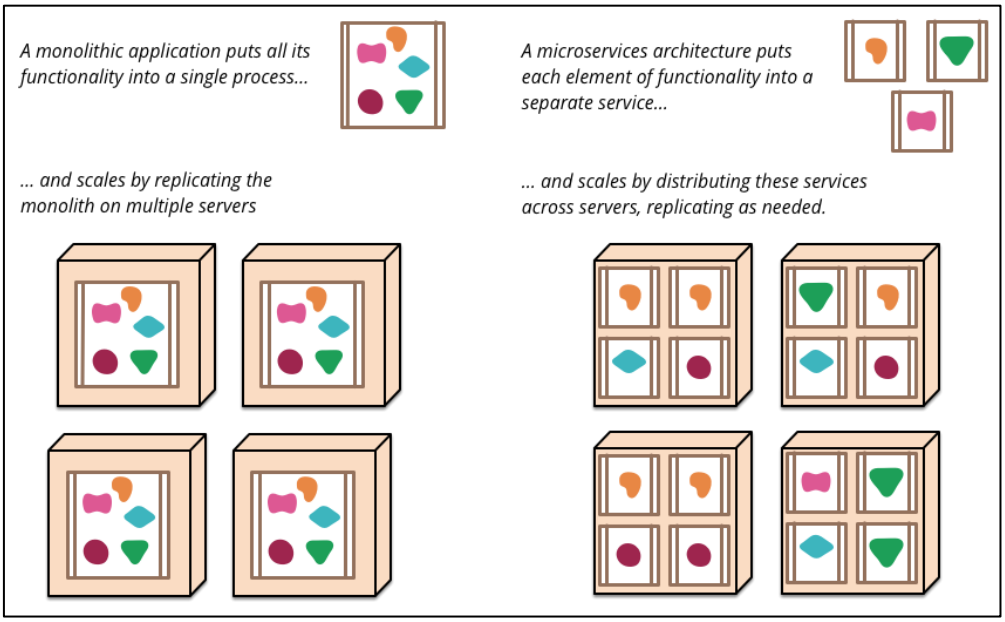


Ilustración 7: Comparación de microservicios con aplicaciones monolíticas [7]

3.2. PRINCIPIOS DE LOS MICROSERVICIOS

Esta sección del documento está basada en el libro "**Building Microservices**" escrito por **Sam Newman**. [8] El libro es un ensayo muy interesante que habla sobre los problemas más comunes a la hora de desarrollar una arquitectura orientada a microservicios. Hace una descomposición de la teoría de los microservicios que, paradójicamente, recuerda a la que debe hacer un equipo de desarrollo a la hora de implementar una arquitectura de este tipo.

Durante el desarrollo del libro, el autor insiste en varias ocasiones en que es importante que cada persona u organización debe de definir sus propios principios ya que son estos los que definirán nuestra forma de trabajar y los que nos ayudarán a salir del paso cuando nos encontremos un dilema arquitectónico. Sin embargo, y dado que puede ser útil definir nuestros propios principios basándonos en unos ya predefinidos, **Sam Newman** nos expone los 7 principios que él considera esenciales para llevar un proyecto de este tipo a buen puerto.

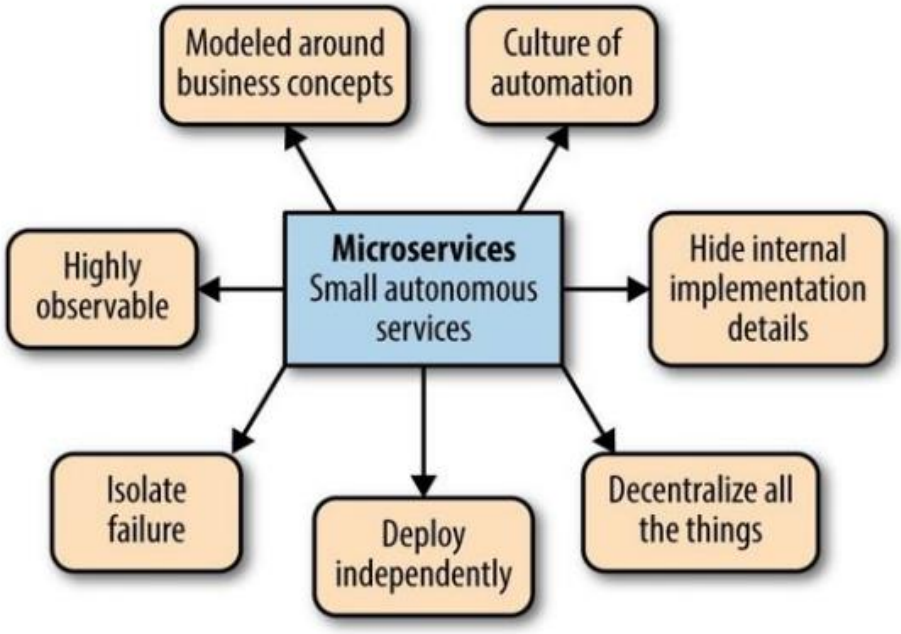


Ilustración 8: Los 7 principios de Sam Newman para crear un servicio autónomo.

En la **Ilustración 8** se muestran los 7 principios de Sam Newman para crear una arquitectura de microservicios:

- ***Modelos basados en los conceptos de negocio***

  Está más que demostrado que los modelos estructurados en torno a las líneas de negocio son mucho más estables que aquellos estructurados en torno a conceptos técnicos.

- ***Adoptar una cultura de automatización***

  Una arquitectura orientada a microservicios añade mucha complejidad a la hora de orquestar el funcionamiento de las aplicaciones. La desagregación del software supone un reto, por ejemplo, en la monitorización del software en producción. Si adoptamos una cultura de automatización, la gestión se simplificará en gran medida.

❖ ***Ocultar los detalles internos de implementación***

Nos debemos centrar únicamente en la función que cada servicio debe de desarrollar. Es indiferente la forma en que se desarrolle, a los otros servicios con los que se comunica solo les interesa que cumpla con su función.

Una buena forma de aislar los detalles internos de la implementación para hacer las aplicaciones agnósticas a la tecnología es la utilización de **APIs,** ya sean de tipo **REST** o de cualquier otro tipo.

❖ ***Descentralizar todo***

Sobre todo, para el gobierno de los equipos de desarrollo. Lo que buscamos es que cada equipo sea dueño de su servicio, y tome las decisiones en función de lo que es óptimo para su desarrollo, no lo que es más común dentro de la empresa. De esta forma se facilita que todos los microservicios cumplan con su función.

❖ ***Microservicios desplegables de forma independiente***

La mejor forma de controlar los errores y de proporcionar a cada microservicio la potencia y los requisitos arquitectónicos que realmente necesita en cada momento es desplegar cada uno de ellos de forma independiente. Si es posible, en máquinas separadas.

❖ ***Aislar fallos***

Hay que tener en cuenta que los microservicios pueden y van a fallar en algún momento. Así, es importante desarrollar las aplicaciones de tal forma que el fallo de un componente no genere un fallo en cascada del resto de microservicios aislando el dominio de error a un único microservicio.

En esta línea, los desarrolladores de **Netflix** crearon en **2011** un servicio llamado **Chaos Monkey.** La única función de este servicio es probar la resiliencia de su infraestructura **IT** deshabilitando intencionadamente y de forma aleatoria sus servidores. De esta forma, los desarrolladores de la conocida empresa desarrollan los microservicios con la

seguridad de que en algún momento van a fallar y, por tanto, su infraestructura está preparada en caso de fallo.

❖ ***Trazabilidad en un vistazo***

A parte de aislar el dominio del fallo y de automatizar los procesos de tal forma que la participación de los trabajadores en la gestión de la plataforma sea mínima, una buena práctica consiste en agrupar la máxima información posible en el menor espacio posible de tal forma que el administrador pueda conocer el estado y la ubicación de un microservicio con un solo vistazo a los logs. Por ejemplo, una buena idea es agrupar información en los ID de error o incluso en las direcciones **IP** de los servidores.

3.3. BENEFICIOS DE LA ARQUITECTURA ORIENTADA A MICROSERVICIOS

La arquitectura de microservicios se está empezando a convertir en una tendencia a la hora de desarrollar aplicaciones en el mundo empresarial.

"A pesar de que su extensión en uso no ha llegado tan allá donde sí lo ha hecho su teoría, muchos desarrolladores están descubriendo cómo esta forma de creación de software favorece el tiempo, rendimiento y estabilidad de sus proyectos." *[9]*

Como se indica en la cita anterior, las empresas se están dando cuenta de que esta nueva forma de desarrollar aplicaciones es más cómoda y eficiente. Los principales motivos para implementar una arquitectura de este tipo se muestran a continuación.

En primer lugar, analizando la **Ilustración 7** en profundidad, nos damos cuenta de las diferentes formas de escalar que tienen ambas arquitecturas. La principal diferencia es que en una arquitectura orientada a los microservicios no escala la aplicación como tal sino cada uno de estos microservicios por separado. De esta forma somos capaces de gestionar de forma eficiente los recursos que utiliza la aplicación y nos permitirá escalar los servidores horizontalmente en tiempo real sin la necesidad de que el servicio se pare.

Por otro lado, la lógica de la aplicación está bien separada, por lo que es fácil de entender y modificar. Además, los microservicios simplifican los procesos de integración y de gestión de cambios. Cada bloque se puede desplegar de forma independiente y un cambio en uno de ellos no afectará al resto de la aplicación.

Finalmente, estas arquitecturas facilitan la gestión de equipos multifuncionales y autónomos, ya que, por sí mismo, cada microservicio es multifuncional. Además de ser independiente de los demás, cada microservicio tiene una parte de base de datos, de back-end, etc. Esto nos permitirá escalar el proceso de desarrollo de una forma más sencilla formando equipos multifuncionales que se encarguen de varios microservicios.

3.4. INCONVENIENTES DE LA ARQUITECTURA ORIENTADA A MICROSERVICIOS

A pesar de todos los beneficios que aportan, las arquitecturas orientadas a microservicios también tienen algunos inconvenientes a los que nos tendremos que enfrentar a la hora de desarrollar la aplicación.

La desagregación que ofrecen los microservicios nos da muchas posibilidades para mejorar el proceso de desarrollo del software y mejorar la disponibilidad de las aplicaciones. Sin embargo, también introducen bastante complejidad, sobre todo a la hora de gestionar y monitorizar el estado de la aplicación. Hay que hacer un gran esfuerzo en despliegues automáticos (si se cuenta con 300 microservicios no se puede desplegar cada uno de ellos manualmente), monitorización (si falla uno, no se puede mirar uno para mirar el problema), gestionar fallos, consistencia de datos, estrategia de pruebas y más factores que introducen los sistemas distribuidos. [10]

Además, un sistema distribuido puede significar más trabajo de planificación. La separación funcional de la aplicación puede ser algo muy complejo de planificar, y las pruebas en una arquitectura de este tipo pueden llegar a ser muy tediosas debido a que se añaden más componentes al desarrollo y por tanto más puntos de fallo.

Por último, estos sistemas pueden conllevar un alto consumo de memoria, por lo que la arquitectura debe de estar especialmente preparada para ellos. Los desarrolladores tendrán que lidiar con problemas como la latencia de la red (aspecto especialmente crítico al no estar los servicios en la misma máquina) o el balanceo de la carga entre los diferentes servidores.

3.5. DOCKER COMO SOLUCIÓN PARA ALOJAR LOS MICROSERVICIOS

Para alojar los microservicios dentro de una máquina utilizaremos contenedores **Docker**. Esta solución es popular a la hora de implementar microservicios porque los contenedores son: [11]

- **Flexibles:** Hasta las aplicaciones más complejas se pueden alojar en un contenedor **Docker**.
- **Ligeros:** Los contenedores aprovechan y comparten el núcleo de la máquina.
- **Intercambiables:** Se pueden realizar actualizaciones y mejoras en caliente.
- **Portables:** Se pueden desarrollar localmente, desplegar en la nube y ejecutar en cualquier sitio.
- **Escalables:** Se puede incrementar y distribuir automáticamente réplicas de los contenedores.
- **Apilables:** Permite apilar servicios verticalmente en caliente.

Un contenedor se ejecuta de forma nativa en **Linux** y comparte el núcleo del servidor con otros contenedores. De esta forma, ejecuta procesos discretos y no ocupa más memoria que cualquier otro ejecutable, haciéndolo más ligero.

Otra forma de implementar microservicios sería creando una máquina virtual para cada uno de ellos, sin embargo, de esta forma perderíamos la flexibilidad que nos aportan los contenedores **Docker** y la escalabilidad sería más complicada. Además, una máquina virtual provee un sistema operativo completo, por lo que es mucho más pesada, ya que proporciona, por lo general, muchos más recursos de los que consume una aplicación. Cómo podemos ver en la **Ilustración 9**, **Docker** corre sobre el sistema operativo de la máquina y proporciona

un servicio al mismo nivel que el de una máquina virtual, administrando contenedores, que solo contienen los binarios, las librerías y la aplicación como tal.

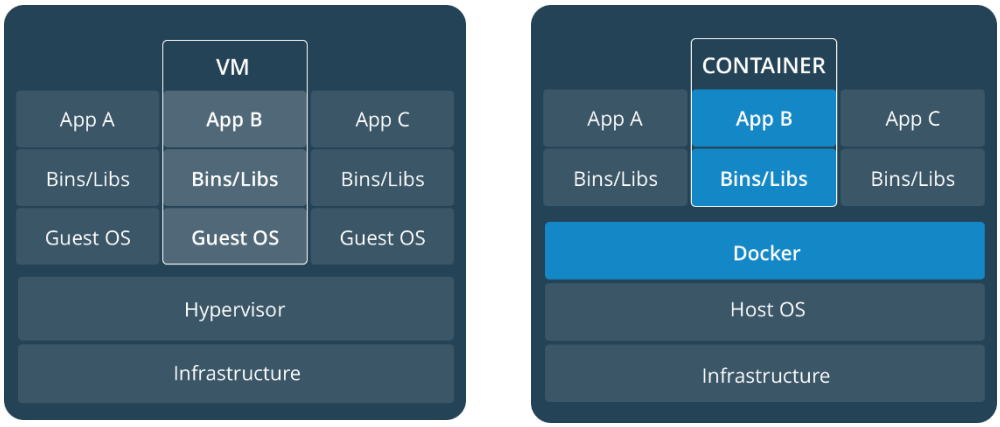


Ilustración 9: Máquinas virtuales vs contenedores Docker [11]

3.6. EJEMPLOS DE UTILIZACIÓN EXITOSA DE LA ARQUITECTURA DE MICROSERVICIOS

La arquitectura orientada a microservicios cada vez es más utilizada. Una demostración de ello es que muchas de las grandes empresas cuyo modelo de negocio gira entorno a internet han decidido implementar sus infraestructuras utilizando microservicios. Grandes ejemplos de ellos son **Amazon**, **Netflix**, **Twitter**, **Ebay**, **Uber** o **Spotify**. Todas ellas son empresas importantes que deben escalar sus recursos en tiempo real para adaptarse a las necesidades de los consumidores.

A continuación, analizaremos alguno de estos ejemplos en más profundidad:

Netflix: Es uno de los pioneros en este tipo de arquitecturas (cómo todas las empresas mencionadas). Su arquitectura está compuesta por más de 500 microservicios, y cuenta con más de 50 millones de suscriptores que realizan unas 2.000 millones peticiones al día.

Es capaz de adaptarse a más de 800 tipos de dispositivos mediante su **API** de straming de video, que, con el objetivo de ofrecer un servicio más estable, realiza cinco solicitudes a diferentes servidores para no perder nunca la continuidad de transmisión. [9]

Ebay: También fue pionera en la utilización de estas tecnologías, tanto en arquitecturas orientadas a microservicios como en la utilización de **Docker**. Las diferentes áreas funcionales de su negocio componen un conjunto de microservicios agrupados por su aplicación principal.

Uber: Es uno de los mejores ejemplos de este tipo de arquitecturas. No solo porque su apuesta por esta arquitectura le ha permitido escalar su aplicación a lo largo de todo el mundo sino porque tiene su propio blog de ingeniería en el que muestran su experiencia utilizando los microservicios. [12]

3.7. ¿QUÉ ES LA NORMATIVA FRTB?

En este capítulo se explicará en profundidad qué es y qué implicaciones tiene la revisión fundamental de la cartera de negociación, o normativa **FRTB (Fundamental Review of the Trading book).**

3.7.1. HISTORIA DE LOS ACUERDOS DE REGULACIÓN FINANCIERA

Para entender bien la normativa **FRTB**, revisaremos brevemente la historia de los acuerdos de regulación del **Comité de Supervisión Bancaria de Basilea (BCBS)**, ya que son los antecedentes a esta normativa y nos ayudará a comprender mejor sus implicaciones.

- ***Basilea I***

En **diciembre de 1988**, el **BCBS**, que estaba formado por los gobernadores de los bancos centrales del G-10, publicó el primero de los **Acuerdos de Basilea**. Este acuerdo establecía, por primera vez, un conjunto de recomendaciones para calcular el capital mínimo que debía tener una entidad bancaria en función de los riesgos que soportaba. En concreto, se estableció que el capital mínimo debía ser de, al menos, un 8% de los activos ponderados por su riesgo. [13]

❖ *Basilea II*

A principios de siglo XX, este mismo comité se dio cuenta de que el Acuerdo de **Basilea I** tenía bastantes limitaciones a la hora de calcular el capital mínimo. El cálculo era insensible a las variaciones del riesgo y no tenía en cuenta la calidad crediticia, es decir, la capacidad de una persona o entidad para devolver los préstamos concedidos en el tiempo acordado. Por tanto, en los cálculos de **Basilea I**, se consideraba que todos los créditos concedidos presentaban el mismo riesgo de no devolución.

Ante estas limitaciones, el comité de Basilea aprueba en **2004** el **Acuerdo de Basilea II**. Este acuerdo es un nuevo conjunto de recomendaciones que se basa en tres pilares fundamentales:

Pilar 1: Disponer de unos requisitos mínimos de capital

El objetivo de este pilar es calcular el valor de los riesgos de mercado, de crédito y operacional. Para ello, por lo general, se utilizan modelos propios de evaluación de riesgos, existiendo incentivos para la mejora de estos modelos. [14]

El riesgo de crédito se calcula a través de tres componentes fundamentales:

- La probabilidad de incumplimiento, **PD (Probability of Default).**
- La gravedad de la pérdida dado el incumplimiento, **LGD (Loss Given Default).**
- La exposición en el momento del incumplimiento, **EAD (Exposure At Default).**

De esta forma, el cálculo del riesgo de crédito es más dinámico y utiliza variables que no se tenían en cuenta en **Basilea I**.

Pilar 2: Supervisar la gestión de fondos propios

Se establece una entidad estatal encargada de supervisar que las entidades mantengan un capital suficiente para soportar los riesgos en los que incurren y de que el cálculo de estos riesgos se hace de manera correcta en función de lo establecido en el pilar 1.

En el caso de **España** el encargado de realizar esta función es el **Banco de España.**

Pilar 3: Disciplina de mercado

El acuerdo pretendía aumentar la transparencia de las entidades bancarias exigiendo la publicación periódica de información acerca de su exposición a los diferentes riesgos y la suficiencia de sus fondos propios.

Otro de los objetivos más importantes de este pilar también era la generalización y homogeneización internacional de las buenas prácticas bancarias.

- ***Basilea III***

Con el acuerdo de **Basilea II** se pretendía lograr una mayor alineación de los requerimientos de capital de las entidades financieras con los verdaderos riesgos que éstas enfrentan.

Sin embargo, durante el inicio de la crisis financiera **(2007-2008)** quedó en evidencia que el riesgo derivado de la cartera de negociación era superior al nivel de capital mínimo exigido por **Basilea II** para cubrirlo. De esta forma, bajo una condición crítica de mercado como la que aconteció en el año **2007**, el capital reservado resultó insuficiente para absorber las pérdidas acontecidas. [15]

Tras estos acontecimientos, en **2010,** el **BCBS** presentó el último conjunto integral de reformas, **Basilea III**, que se centra principalmente en proporcionar las medidas y herramientas necesarias para mejorar la capacidad de respuesta del sistema bancario ante perturbaciones económicas y financieras y conseguir así una mayor estabilidad financiera mundial. [16]

Este nuevo acuerdo no pretende sustituir a los dos anteriores sino complementarlos. Añade nuevos requisitos en torno a tres principios básicos, el capital, el apalancamiento y la liquidez. Para asegurar que una entidad tiene capital de calidad suficiente para superar una crisis, introduce el concepto de los "colchones de capital", que los bancos tendrán que construir gradualmente **entre 2016 y 2019**. Además, introduce conceptos nuevos como el ratio de apalancamiento, que tendrá que ser mayor del 3%, el **LCR (Coeficiente de Cobertura de Liquidez)** o el **NSFR (Coeficiente de Fondeo Estable Neto)**.

3.7.2. NORMATIVA FRTB Y SUS IMPLICACIONES

Se puede definir la normativa **FRTB** como un conjunto de propuestas definidas por el **BCBS** para las grandes entidades financieras internacionales. Es la siguiente generación de las reglas de capital reglamentario de riesgo de mercado.

La versión final de esta normativa vino incluida dentro de los **Requerimientos mínimos de Capital para el Riesgo de Mercado** (**MCRMR-2016**) y será de obligado cumplimiento a partir de **enero de 2019**. Los **MCRMR-2016** fueron publicados por el **BCBS** en **enero de 2016** y muchos los consideran como el acuerdo de **Basilea 3.5** o incluso **Basilea IV**. [17]

A continuación, se exponen algunas de las novedades que introduce la **FRTB**: [18]

1. Estandariza el criterio para definir el límite entre la cartera de negociación y la cartera bancaria. Con esto se pretende que la cuantificación del capital entre los diferentes bancos sea más homogénea y evitar así el arbitraje financiero.
2. Para calcular el riesgo de las posiciones sometidas a titulización no se podrá utilizar modelos internos, sino que tendrá que usarse el **"Método estándar revisado".**
3. El modelo de justificación y de graduación necesitará al menos 10 años de datos históricos para el 75% de las clases de activos.

El impacto de la normativa está siendo muy alto, una muestra de ello es que el 74% de las entidades financieras utilizan modelos internos para realizar el cálculo del capital regulatorio cuando, a partir de **enero de 2019**, todas ellas estarán obligadas a utilizar el **"Método Estándar Revisado"** para realizar esta tarea. [15]

Además, la potencia básica de procesamiento aumentará drásticamente a la hora de calibrar la volatilidad del mercado. La **FRTB** exige utilizar una ventana temporal de información de 10 años y un modelo de calibración mucho más complejo para realizar el calibrado. De esta forma, las entidades bancarias se verán obligadas a modificar las aplicaciones de calibración de la volatilidad y a replantearse la arquitectura de las mismas. En **BBVA**, por ejemplo, se ha decidido implementar una arquitectura de microservicios para realizar el calibrado.

Capítulo 4: DEFINICIÓN DEL TRABAJO

4.1. MOTIVACIÓN DEL PROYECTO

Cuanto mayor es la presencia de entidades financieras en el mercado capitales mayor es la sensibilidad que estas adquieren sobre el riesgo de mercado. La volatilidad opera como instrumento de control y auditoría que suministra información sobre la tendencia del riesgo en las inversiones realizadas, por lo que es de vital importancia realizar un buen cálculo de esta para conseguir una mayor estabilidad financiera en las entidades. [19]

Conscientes de esto, el **Comité de Supervisión Bancaria de Basilea** (**BCBS**) publicó en **enero de 2016** la **Revisión Fundamental de la Cartera de Negociación** (**FRTB**), que estandariza la forma de realizar el cálculo del riesgo de mercado entre las diferentes entidades financieras mundiales. Así, todos los bancos tendrán que cumplir con las restricciones mínimas impuestas por el **BCBS** de tal forma que la información obtenida de estos cálculos sea más fiable y precisa que la obtenida hasta ahora.

En esta línea, el banco **BBVA** es consciente de que mejorar su procedimiento de calibración de la volatilidad del mercado es importante, más allá de las sanciones que podría recibir por parte del **BCBS.** Por esta razón, aunque la motivación inicial del proyecto es cumplir con la normativa **FRTB,** y aprovechando la flexibilidad que exige la arquitectura a implementar, el proyecto incluye una serie de funcionalidades extra orientadas, sobre todo, a realizar informes del mercado en tiempo real, ya sea para incorporarlas a la base de datos histórica o para responder a la consulta realizada por un trader.

En cualquier caso, la columna vertebral del proyecto, sobre la que se sustentan el resto de las funcionalidades, es crear una arquitectura orientada a los microservicios para cumplir con las restricciones impuestas por la **FRTB**. De forma más concreta, la **FRTB** pide a los bancos que establezcan un límite objetivo entre su cartera de negociación y su cartera de inversión, cambiando su enfoque de **modelo interno de valor en riesgo** (**VAR**) a una medida de riesgo de **déficit esperado** (**ES**), obteniendo así una mejor comprensión del riesgo

de cola y la adecuación del capital cuando los mercados financieros están bajo estrés**.** [20]

Atendiendo al proyecto, esto se traduce en que el banco debe ampliar el número de variables utilizadas para realizar la calibración de la volatilidad del mercado, incluyendo una ventana temporal de 10 años de información por cada subyacente. De esta forma, se conseguirá medir el riesgo de mercado de una forma mucho más precisa, ya que el riesgo se puede medir como la volatilidad de los resultados esperados.

En cualquier caso, para realizar la calibración se necesita una arquitectura mucho más potente que la existente actualmente en la entidad para realizar esta función. Además, incorpora restricciones muy específicas que requieren diseñar un nuevo sistema de tal forma que no se pueden reutilizar los sistemas actuales, por lo que el proyecto incluye tanto el desarrollo del software como el diseño e implementación de la arquitectura, que en este caso será una arquitectura orientada a microservicios.

4.2. OBJETIVOS

El desarrollo del proyecto se basa en la implantación de una arquitectura de microservicios para realizar una aplicación que permita al banco **BBVA** cumplir con la normativa exigida por el **BCBS** y crear una serie de funcionalidades extra. Concretamente, los objetivos más importantes son:

- Estudiar y analizar una arquitectura orientada a los microservicios.
- Comprobar la viabilidad de la utilización de microservicios dentro de la industria financiera y su importancia en la trasformación digital.
- Realizar un caso de uso real con microservicios en la industria financiera.

Las arquitecturas orientadas a microservicios son cada vez más populares a la hora de crear una aplicación en el mundo empresarial. Sin embargo, su utilización en el mundo financiero y, en concreto, en el banco **BBVA**, es escasa, por lo que el resultado de este proyecto será importante, y nos permitirá analizar la viabilidad de la utilización de microservicios dentro del banco y su impacto en la trasformación digital.

Este tipo de arquitecturas ha avanzado mucho en la teoría, pero aún no se ha reflejado en la práctica. Aunque la eficacia de los microservicios para mejorar muchos aspectos de las clásicas arquitecturas monolíticas es un hecho, también está suficientemente probado que añaden una complejidad extra a la hora de realizar el desarrollo.

De esta forma, otro de los objetivos del proyecto es realizar un estudio de las arquitecturas de microservicios con el fin de detectar y documentar todos los retos con los que tendremos que enfrontarnos durante el desarrollo del proyecto.

Es importante recalcar que, hasta ahora, hemos hablado de objetivos principalmente teóricos en los que se pretende estudiar y analizar las necesidades y los beneficios que puede tener una arquitectura de microservicios en un gran proyecto dentro de una entidad como el banco **BBVA.** Sin embargo, no es posible desarrollar un prototipo de la totalidad del proyecto para presentar en este trabajo ya que es un proyecto en el que están involucradas muchas más personas.

De esta forma, el objetivo final de este trabajo es el de implementar un caso de uso real que nos permita comprobar y sacar conclusiones sobre las hipótesis alcanzadas en el estudio teórico.

4.3. METODOLOGÍA Y PLANIFICACIÓN

Una metodología es el conjunto de métodos que se siguen en un proyecto con el propósito de alcanzar las metas propuestas al principio de este. Se podría entender como una plantilla que nos dice cómo actuar durante el desarrollo.

Es una de las claves para conseguir el éxito en un proyecto. Una metodología define la forma de trabajar y la planificación a seguir dentro de un proyecto y, por tanto, una metodología mal escogida podría suponer el fracaso del mismo o un retraso en los tiempos de entrega. Por tanto, es importante que sea la metodología la que se ajusta al proyecto y no al revés.

En este caso, el proyecto desarrollado se compone, en alta medida, de desarrollo software, donde es muy habitual que los requisitos y soluciones evolucionen con el tiempo. De esta forma, no tendría sentido utilizar una metodología tradicional, donde se planifica la totalidad

del proyecto al principio del mismo y realizar cambios en la planificación resulta altamente costoso.

En lugar de ello, utilizaremos una metodología '**Agile'**, que es un conjunto de metodologías para el desarrollo de proyectos que precisan de rapidez y flexibilidad para adaptarse a condiciones cambiantes del sector o mercado, aprovechando dichos cambios para proporcionar ventaja competitiva. Es decir, el proyecto se "trocea" en pequeñas partes que tienen que completarse y entregarse en pocas semanas. [21]

En la **Ilustración 10** se puede ver la herramienta el tablón de **Scrum** facilitado por la herramienta **Jira**. En este se van poniendo y asignando las tareas relacionadas con el proyecto de tal forma que todos los integrantes del equipo de desarrollo sepan cómo está avanzando el proyecto.

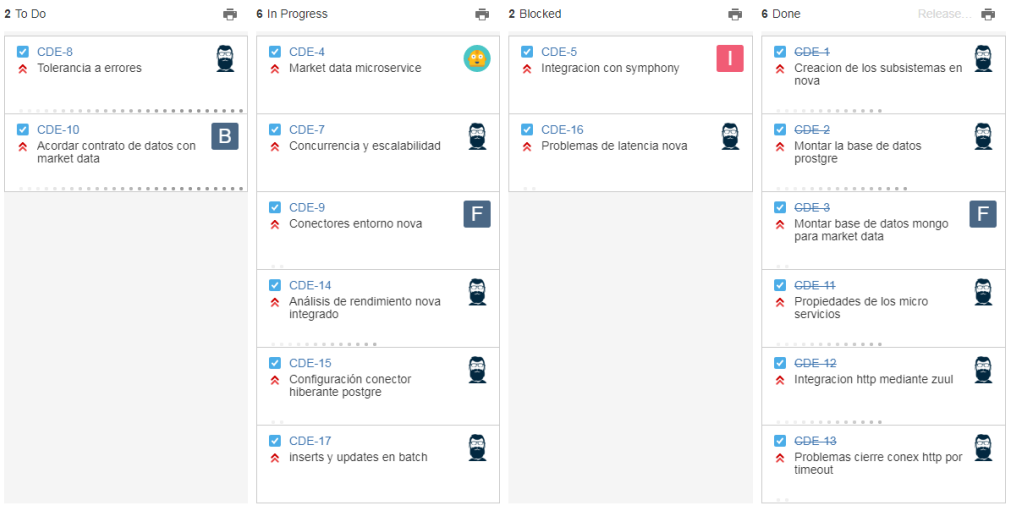


Ilustración 10: Tablón Scrum de Jira

Precisamente, esta metodología concuerda perfectamente con el tipo de proyecto al que nos enfrentamos ya que, por definición, una arquitectura orientada a microservicios consiste en dividir el conjunto del software en pequeños bloques funcionales.

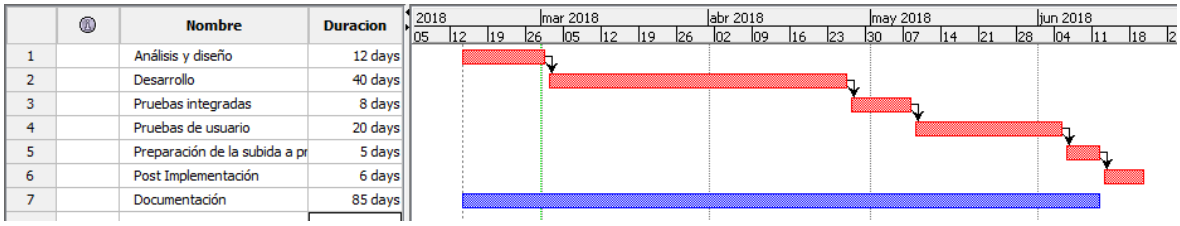


Ilustración 11: Planificación del proyecto

En la **Ilustración 11**, se muestra la planificación que se seguirá para desarrollar el trabajo. Como se puede observar, el proyecto está dividido en 6 tareas resumen. Es una planificación a cinco meses en la que no está incluida la planificación de los sprints a realizar en cada tarea resumen.

Estas tareas se realizarán en cascada ya que es necesario finalizar una para poder hacer la siguiente. A su vez, hay una tarea en paralelo durante todo el proyecto, que consiste en documentar las diferentes etapas realizadas durante el desarrollo del proyecto.

Capítulo 5: SISTEMA/MODELO DESARROLLADO

En este apartado se explicará en profundidad el sistema desarrollado y estará estructurado de la misma forma que la planificación del proyecto, ya que en el desarrollo de este tipo de proyectos cada etapa concreta un poco más lo hecho en la anterior, por lo que es muy apropiado seguir esta misma estructura para comprender bien el funcionamiento del sistema. Las etapas de desarrollo del proyecto son:

1. **Diseño y especificación de los bloques funcionales del sistema (microservicios)**
2. **Diseño de cada microservicio para cumplir con las especificaciones**
3. **Integración de los microservicios**
4. **Optimización del rendimiento de la aplicación**

A continuación, se desarrollarán los apartados indicados anteriormente.

5.1. DISEÑO Y ESPECIFICACIÓN DE LOS BLOQUES FUNCIONALES DEL SISTEMA (MICROSERVICIOS)

En este apartado se realizará la descomposición funcional a alto nivel del sistema, dividiendo la aplicación en diferentes bloques funcionales independientes entre sí. Además, se especificará de forma clara la funcionalidad de cada elemento y cómo se comunicará con el resto de los componentes de tal forma que el equipo de desarrollo de cada microservicio sepa de antemano cómo conectar con los microservicios adyacentes.

La división de la arquitectura en bloques funcionales debe hacerse teniendo en cuenta dos conceptos clave, el bajo acoplamiento y la alta cohesión [8]. Bajo acoplamiento de tal forma que un cambio en un servicio no afecte a otros, pero alta cohesión, para evitar desarrollar la misma funcionalidad en distintos microservicios. Además, la comunicación entre distintos microservicios debe de ser estrecha, pero es importante limitar los tipos de llamadas entre

ellos, ya que una comunicación frecuente entre servicios, aparte de un potencial problema de rendimiento, puede conllevar aumentar el nivel de acoplamiento entre servicios.

Así, la parte más complicada de esta fase de desarrollo es determinar los límites de actuación de cada microservicio para conseguir que estos dos conceptos se cumplan al máximo.

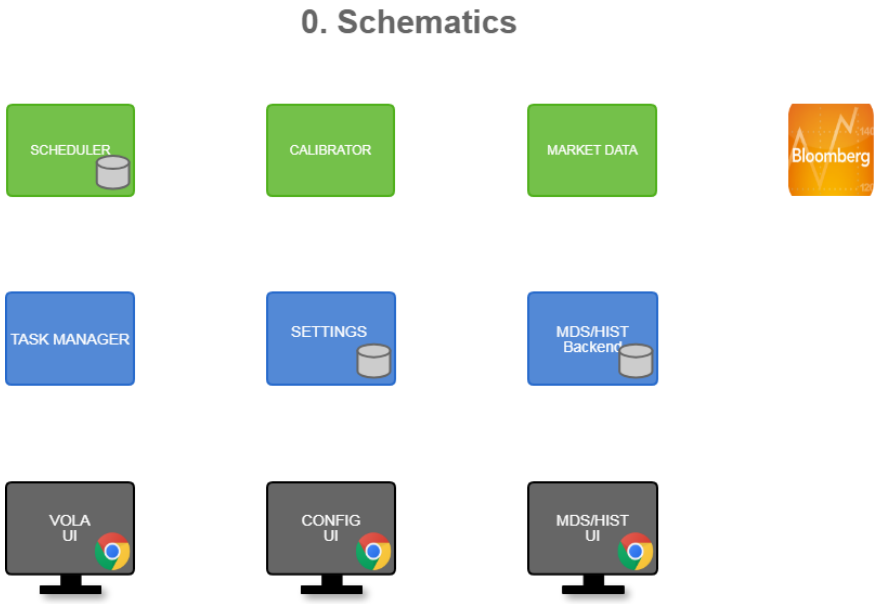


Ilustración 12: Bloques funcionales del sistema a implementar

La descomposición realizada se muestra en la **Ilustración 12**. Cada uno de los bloques mostrados en dicha ilustración tiene una funcionalidad concreta dentro del proyecto, aunque no todos se utilizan en el caso de uso principal. Para entender mejor la funcionalidad de cada bloque explicaremos a continuación los diferentes casos de uso planteados para el proyecto.

5.1.1. CASOS DE USO

A continuación, se expondrán los casos de uso concretos que se quieren desarrollar. Cómo ya se ha indicado anteriormente, el primero de los casos de uso se centrará en cumplir con la normativa FRTB, mientras que los otros dos crearán un servicio complementario sobre la arquitectura del primero.

❖ *Calibrador Batch*

Es el caso de uso inicial, obligado por el **FRTB.** Su función es realizar el calibrado de la volatilidad de cada subyacente cada equis tiempo, por ejemplo, cada dos semanas.

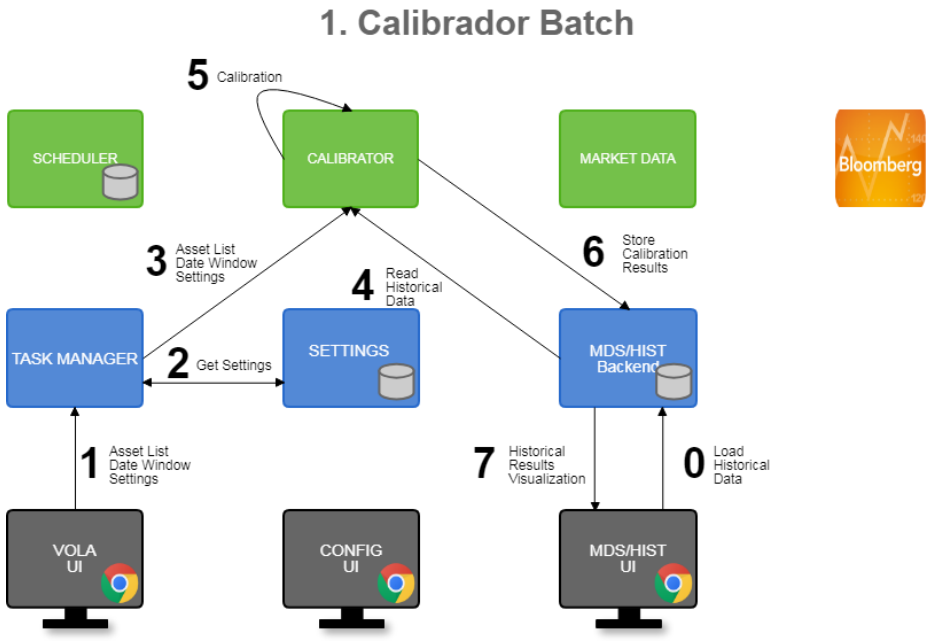


Ilustración 13: Esquema utilizado para cumplir con la normativa FRTB

La aplicación se lanzará manualmente. En ese momento el usuario indica al Gestor de Volatilidad (**VOLA Backend**) la lista de subyacentes, la duración de la ventana temporal y los ajustes específicos de calibración.

Con los ajustes de calibración especificados por el usuario, el **VOLA Backend** se comunica con el **Calibrador**, un cluster de servidores encargado de calcular, en paralelo, la volatilidad de cada subyacente. A medida que se realiza, la calibración de cada subyacente se almacena en base de datos. Es importante que el calibrador vaya realizando commits parciales para que no tenga que volver a empezar el proceso desde cero si por cualquier motivo se callera el servicio.

Finalmente, se envían los resultados a la entidad reguladora y se muestran los resultados en la interfaz de usuario.

❖ *Real Time*

En la **Ilustración 14** se muestra la arquitectura utilizada para este caso de uso. Como indica su nombre (**Real-Time**) este caso de uso es el encargado de lanzar ejecuciones programadas en tiempo real para recoger la información del mercado en ese momento y guardarla en la base de datos histórica.

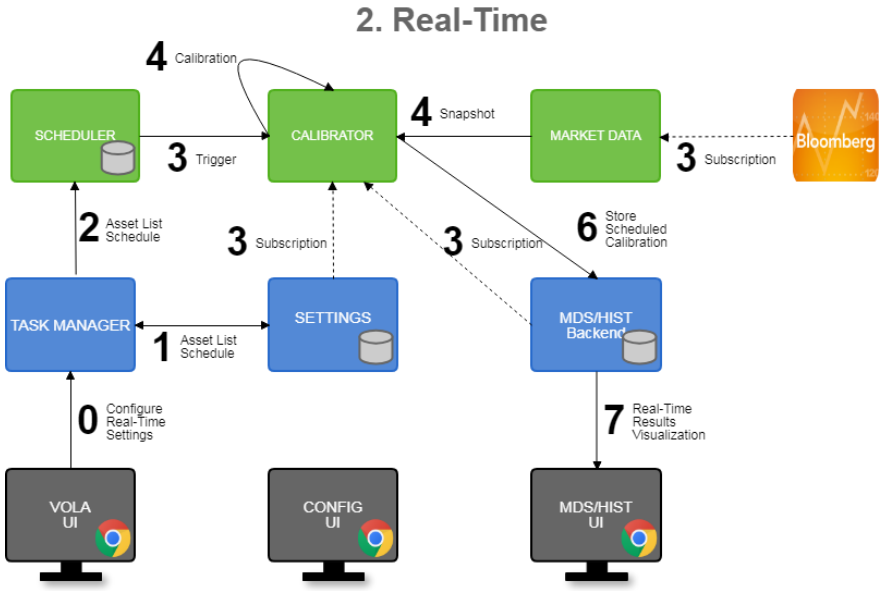


Ilustración 14: Esquema utilizado para cumplir con el caso de uso "Real-Time"

Para realizar este caso de uso, la aplicación tendrá una interfaz de usuario desde la que se podrán configurar ejecuciones programadas. Estas ejecuciones se realizarán con los parámetros de calibración indicados por el usuario de tal forma que se podrá realizar una calibración parcial de los subyacentes y una ventana temporal de información menor a la exigida por la **FRTB**.

❖ *On demand*

El caso de uso "**On demand"** es similar al anterior, pero tiene un enfoque distinto. permitiría al empleado consultar la volatilidad de uno o varios subyacentes en un momento concreto. Por ejemplo, la realizaría el Trader cuando se encuentra en negociación con el cliente. En ambos casos, la información en tiempo real del mercado provendrá del bloque "feeder", que creará una imagen del mercado con la información recogida de la API de Bloomberg.

La arquitectura utilizada para realizar el caso de uso "**On Demand"** se muestra en la **Ilustración 15**. Cómo se puede observar, en este caso de uso no se almacena nada en la base de datos histórica. Esto es debido a que el objetivo de este caso de uso es el de hacer pequeñas consultas puntuales que no tendría sentido almacenar.

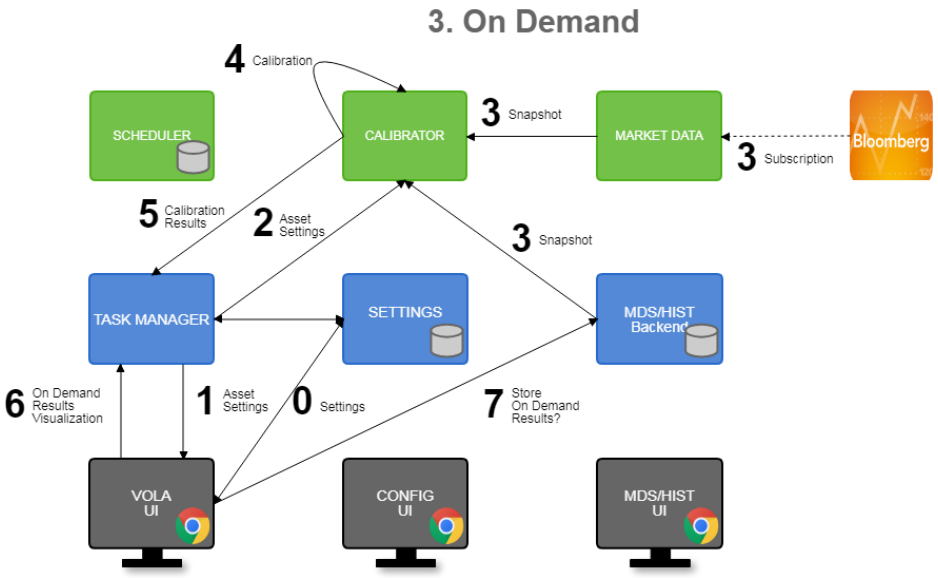


Ilustración 15: Esquema utilizado para el caso de uso "On Demand"

Tanto la función "**On demand**" como la de "**Real Time**" permitirá a los empleados del banco tener información actualizada y veraz sobre el mercado de tal forma que, por ejemplo, la decisión sobre ejercer o no las opciones de compra disponibles sobre los activos (**equitys**) se tome sobre información contrastada y se consiga ahorrar dinero. Por otro lado, también es útil a la hora de comprar nuevos valores, ya que al disponer de información actualizada del mercado se evitarían arbitrajes y se conseguiría comprar al valor real del mercado.

5.1.2. ESPECIFICACIÓN FUNCIONAL DE LOS MICROSERVICIOS

En el apartado anterior hemos visto una descomposición funcional de la aplicación. Esta descomposición se ha hecho pensando en todos los posibles casos de uso que se pretende implementar en el futuro. Esto es importante, porque nos permite diseñar el software de forma que cumpla con los requisitos de todos los casos de uso. Sin embargo, para este

proyecto, desarrollaremos únicamente el primero de los casos de uso indicados, el **Calibrador Batch**.

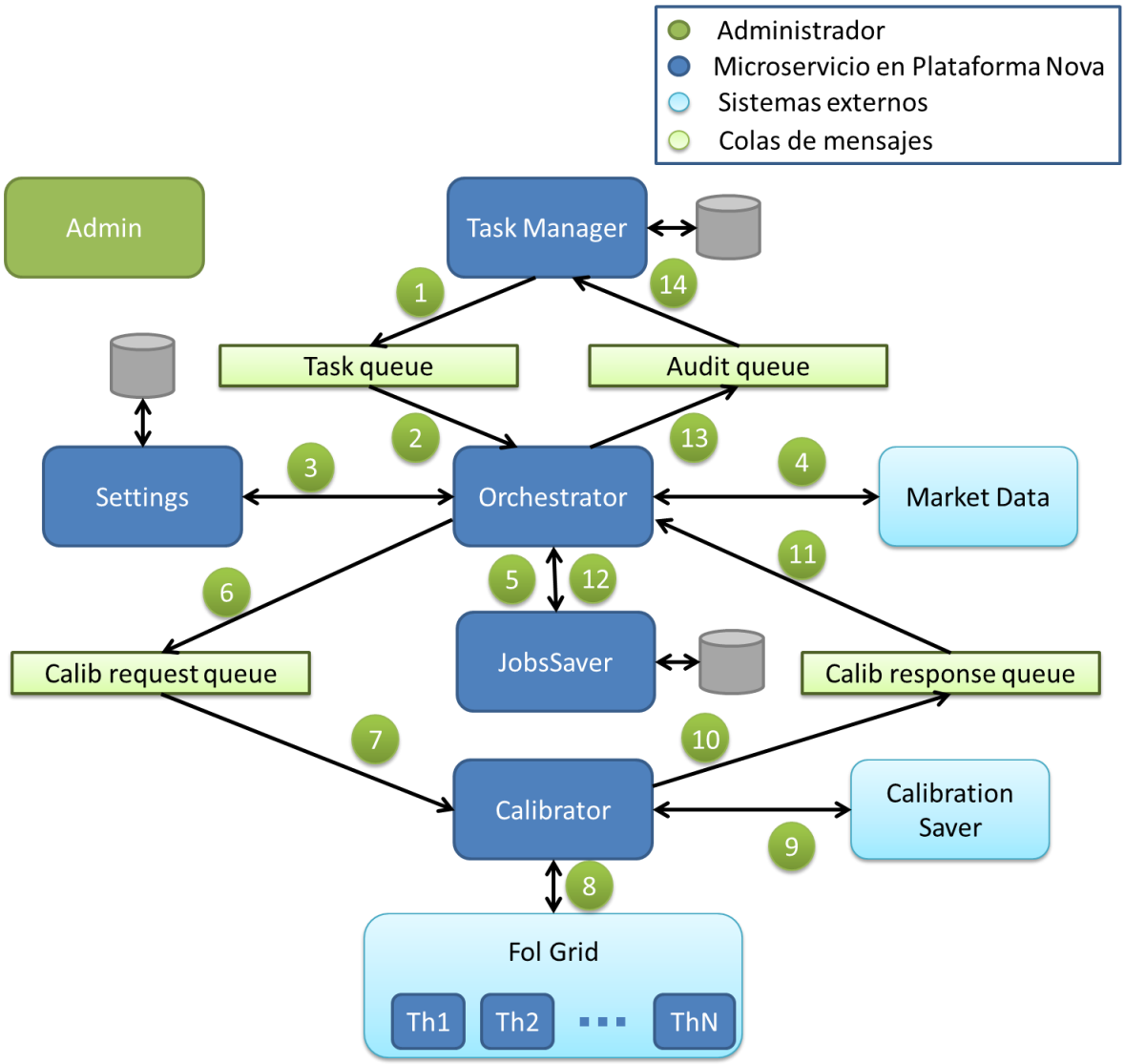


Ilustración 16: Esquema de microservicios utilizado para el Calibrator Batch

En la **Ilustración 16** podemos ver cómo sería el flujo de una ejecución básica de la aplicación, en ella, vemos 9 microservicios, totalmente independientes. En este apartado se especificará la función de cada uno de los microservicios, que conjuntamente conforman la totalidad de la aplicación.

Comparando esta imagen con la **Ilustración 13**, contemplamos algunas diferencias significativas. En primer lugar, encontramos un microservicio, "**Task Manager**", que no estaba en el anterior esquema y cuya función es, básicamente, la de hacer de launcher para realizar el lanzamiento de las ejecuciones deseadas. Por otro lado, también encontramos el "**Orchestrator**", cuya misión es hacer de punto de unión entre todos los microservicios. Por último, el "**Fol Grid**" es un Grid de cálculo que tiene **BBVA** para realizar el cálculo de productos financieros y de riesgos.

A continuación, profundizaremos un poco más en la función de cada uno de estos microservicios, dejando definidas sus funciones y la forma de comunicarse con el resto del ecosistema de la aplicación.

5.1.2.1. Task Manager

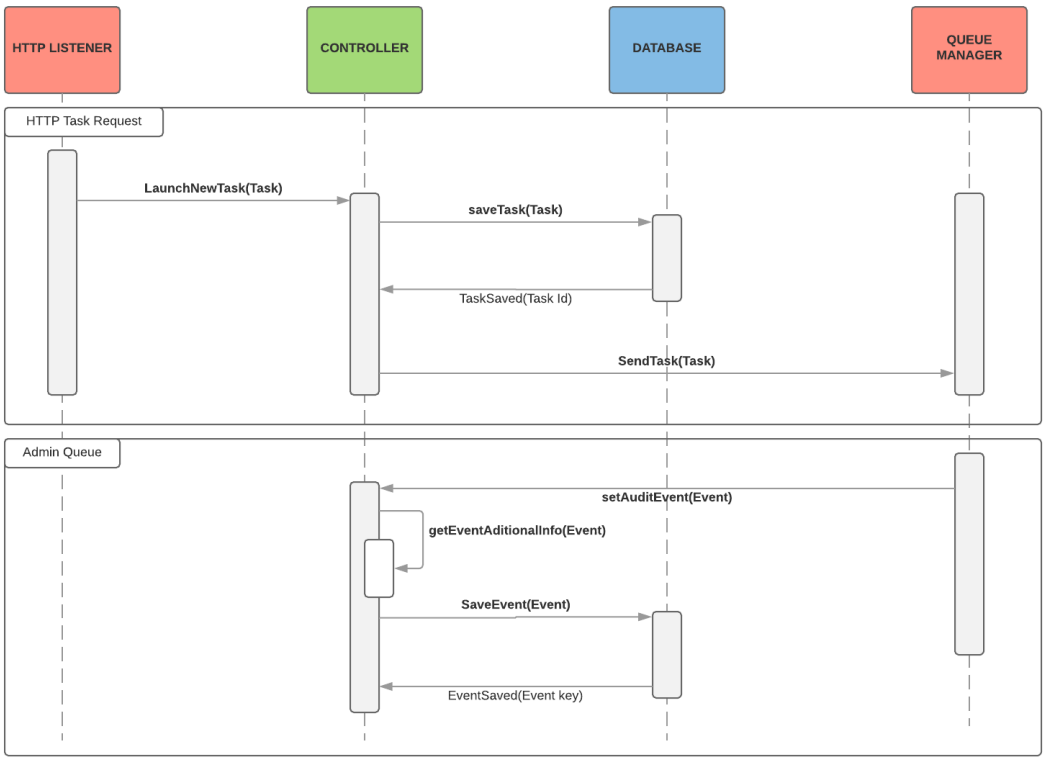


Ilustración 17:Diagrama de secuencia básico Task Manager

Como ya se ha comentado anteriormente, este servicio es el encargado de recibir las solicitudes de ejecución y lanzar la aplicación con los parámetros indicados por el usuario. El usuario podrá indicar los siguientes parámetros de ejecución:

- Subyacentes sobre los que se desea realizar la calibración
- Ventana temporal utilizada (fecha de inicio y fecha de fin)

La información de mercado de cada subyacente no hará falta proporcionarla ya que será la propia aplicación quien la recopile.

El **Task Manager** se encargará, además, de auditar los tiempos de ejecución de las diferentes etapas de la ejecución de la aplicación. En concreto, se guardará las fechas de inicio y fin de ejecución de las siguientes etapas:

- Obtención de ajustes de calibración.
- Obtención de los datos de mercado.
- Planificación de tareas a realizar (calibraciones).
- Tiempo de realización de cada una de las tareas planificadas.

Comunicación con otros microservicios:

El usuario se podrá conectar con este microsercicio a través del protocolo **HTTP** de tal forma que pueda lanzar una ejecución directamente desde una página web.

Por otro lado, la comunicación con el resto de los microservicios se realizará de forma asíncrona, utilizando colas de mensajes por prioridad. De esta forma, el microservicio simplemente se encargará de escuchar las ejecuciones de los usuarios y depositarlas en la cola "**Task Queue**", con la prioridad deseada. Finalmente, el resto de los microservicios depositarán en la cola "**Admin Queue**" sus tiempos de ejecución de tal forma que este microservicio las reciba y las almacene en base de datos.

5.1.2.2. Settings

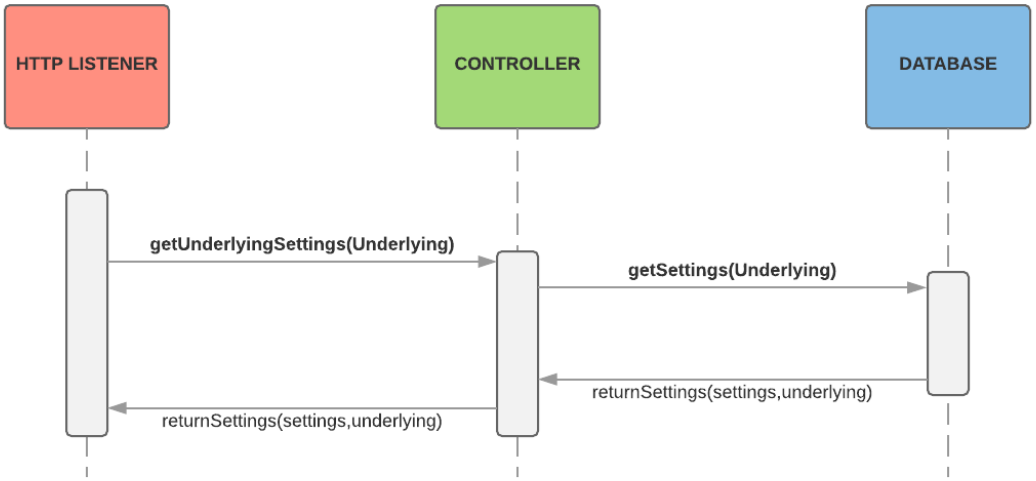


Ilustración 18: Diagrama de secuencia Settings

Cada subyacente tiene ajustes de calibración diferentes. Los parámetros de calibración estarán guardados en una base de datos y no tienen por qué ser iguales para todos los subyacentes. Los parámetros exactos de configuración se especificarán más adelante. En cualquier caso, antes de comenzar el proceso de calibración, la aplicación debe recabar los ajustes de calibración específicos de cada subyacente.

El microservicio de **Settings** se encargará de realizar esta tarea de tal forma que será el único microservicio que lea de la base de datos de **Settings,** y todos los microservicios que necesiten esta información tendrán que comunicarse con este servicio.

Comunicación con otros microservicios:

Atendiendo al esquema mostrado en la **Ilustración 16**, se puede observar que, en el flujo de ejecución mostrado, la respuesta de este microservicio se espera que sea instantánea, ya que el resto de la ejecución depende de esta respuesta. De esta forma, no tendría sentido que la comunicación con el **Orchestrator** se realizase mediante colas, por lo que la comunicación se realizará mediante **HTTP**.

5.1.2.3. Market Data

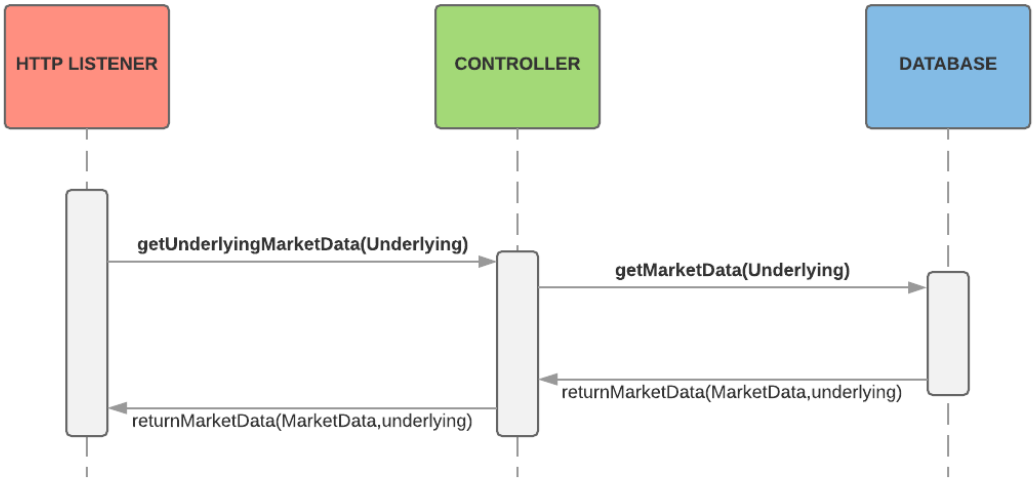


Ilustración 19: Diagrama de secuencia Market Data

Al igual que el microservicio anterior, este microservicio responde a la necesidad de la aplicación de obtener información específica de cada subyacente para realizar la calibración de la volatilidad. En este caso, como indica el nombre del microservicio, la información necesaria son los datos de mercado de cada subyacente para el periodo solicitado por el usuario.

Esta información estará almacenada en una base de datos propia, por lo que este microservicio también se encargará de mantener actualizada la información almacenada. Para ello, se comunicará con otros sistemas externos para obtener la información de mercado diaria de cada subyacente para incorporarla a la existente.

Comunicación con otros microservicios:

En este aspecto, este microservicio es muy similar al microservicio de ***Settings***, por lo que la comunicación con el **Orchestrator** se realizará también por **HTTP.** Además, la comunicación del microservicio con los sistemas externos de información de mercado se realizará por **API REST**, por lo que esta comunicación también se realizará por **HTTP.**

5.1.2.4. Orchestrator

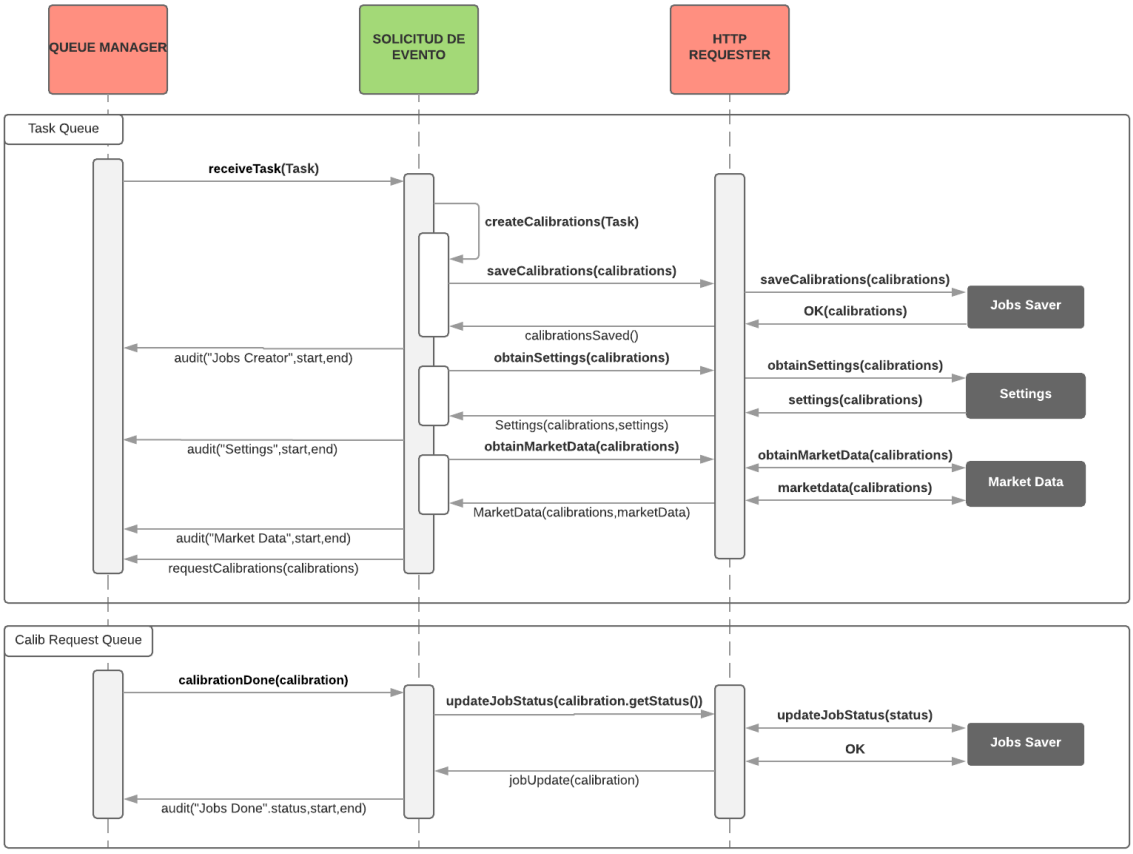


Ilustración 20: Diagrama de secuencia Orchestrator

Este microservicio es, probablemente, el más importante de los desarrollados en la aplicación ya que es el microservicio que recibe la petición del **Task Manager** y gestiona la ejecución del resto de microservicios para cumplir con la tarea solicitada.

Se podría decir que el **Orchestrator** es el "cerebro" de la aplicación, ya que es el único que tiene visión completa de todo el flujo de ejecución. Al igual que el resto de microservicios, recibe una tarea por parte del **Task Manager** y él mismo la ejecuta y le devuelve el resultado obtenido. Sin embargo, al contrario que la mayoría de los microservicios de este proyecto, el **Orchestrator** no tiene la lógica para obtener directamente la respuesta, sino que se vale de otros microservicios para obtenerla.

El flujo seguido por el **Orchestrator** para realizar el caso de uso principal (**Calibrador Batch**) es el siguiente:

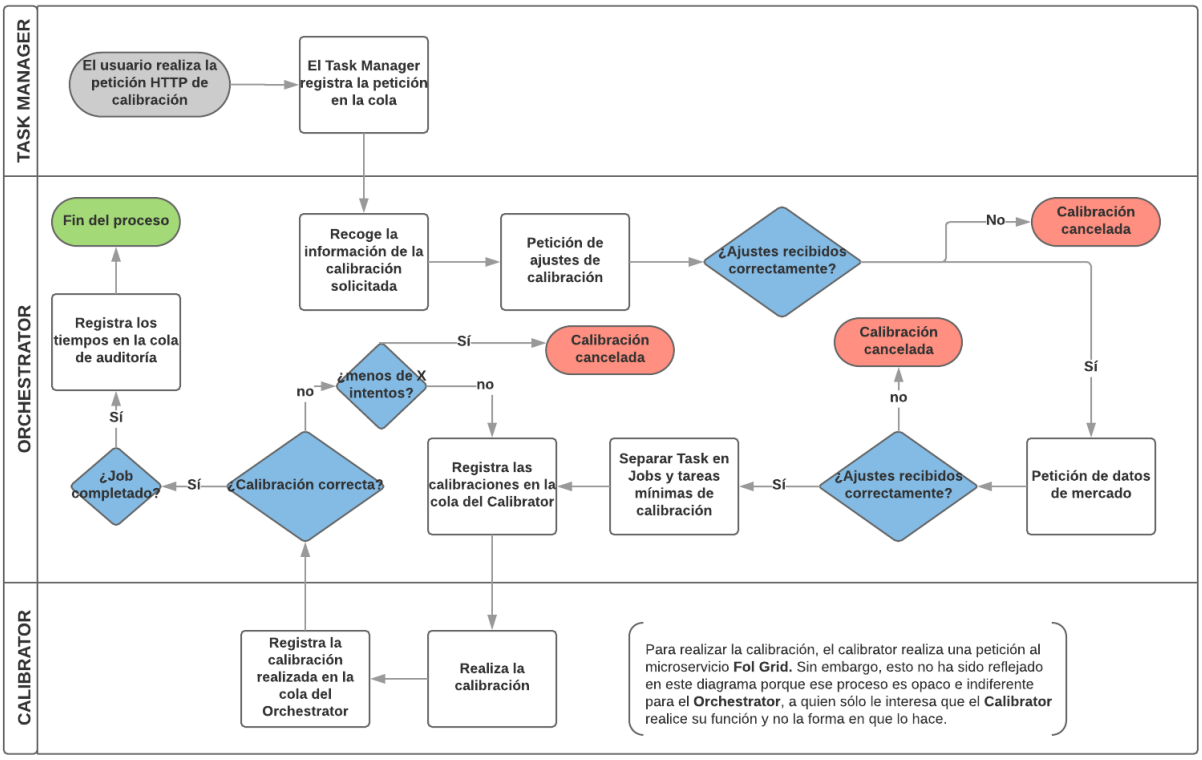


Ilustración 21: Diagrama de flujo del Orchestrator

1. El **Orchestrator** recibe los subyacentes y la ventana temporal sobre los que se desea realizar la calibración.
2. Realiza una petición **HTTP** al microservicio ***Settings*** para obtener los ajustes de calibración de cada subyacente. Realizará una petición por subyacente.
3. De la misma forma, realiza una petición **HTTP** al microservicio **Market Data** para obtener los datos de mercado de cada subyacente en la ventana de tiempo seleccionada.
4. Una vez obtenido los datos, el **Orchestrator** se comunica con el microservicio **Jobs Saver** para desglosar la tarea en tareas mínimas de calibración. La tarea mínima de calibración es la calibración de un subyacente en un día, por lo que el número total de tareas de calibración será la multiplicación del número de subyacentes por el número de días laborables de la ventana temporal seleccionada.

5. Almacena las tareas obtenidas en la cola **Calib request queue** para que el **Calibrator** las ejecute.
6. Finalmente, escucha la respuesta del **Calibrator**, almacenada en la cola **Calib response queue**, y se lo comunica a **Jobs Saver** para que actualice el estado de la tarea ejecutada.

Es importante recalcar que al ir finalizando cada una de las tareas realizadas, el **Orchestrator** envía un mensaje a la cola **Audit queue** para que el **Task Manager** pueda almacenar los tiempos de ejecución de cada etapa.
Este microservicio es uno de los que más carga tendrá ya que es el que se encargará de controlar todo el flujo de ejecución. Por este motivo, este microservicio tendrá que ser especialmente escalable, y será de vital importancia calcular bien el número de servicios a utilizar en paralelo para no generar un bloqueo en este punto de ejecución.

Comunicación con otros microservicios:

Este microservicio se debe de hablar con todos los demás, por lo que debe de soportar tanto comunicación mediante colas de mensajes como comunicación mediante el protocolo **HTTP**.
Utilizará el protocolo **HTTP** para comunicarse con los siguientes microservicios:

- ***Settings***
- **Market Data**
- **Jobs Saver**

Por otro lado, para comunicarse con los microservicios **Task Manager** y **Calibrator** utilizará las siguientes colas:

- **Task queue**
- **Audit queue**
- **Calib request queue**
- **Calib response queue**

5.1.2.5. Jobs Saver

Como ya se ha explicado en el anterior apartado, este microservicio se encarga de desglosar una tarea de calibración de la volatilidad de un subyacente en un periodo de tiempo determinado en un conjunto de subtareas más simples. Se creará una subtarea de calibración para cada día laborable del periodo de tiempo indicado por el usuario, y será la calibración de la volatilidad del subyacente en ese día.

De esta forma, la primera tarea que deberá realizar este microservicio es obtener los días laborables del periodo de tiempo seleccionado para posteriormente, almacenar en base de datos cada una de estas tareas y devolvérselas al **Orchestrator.**

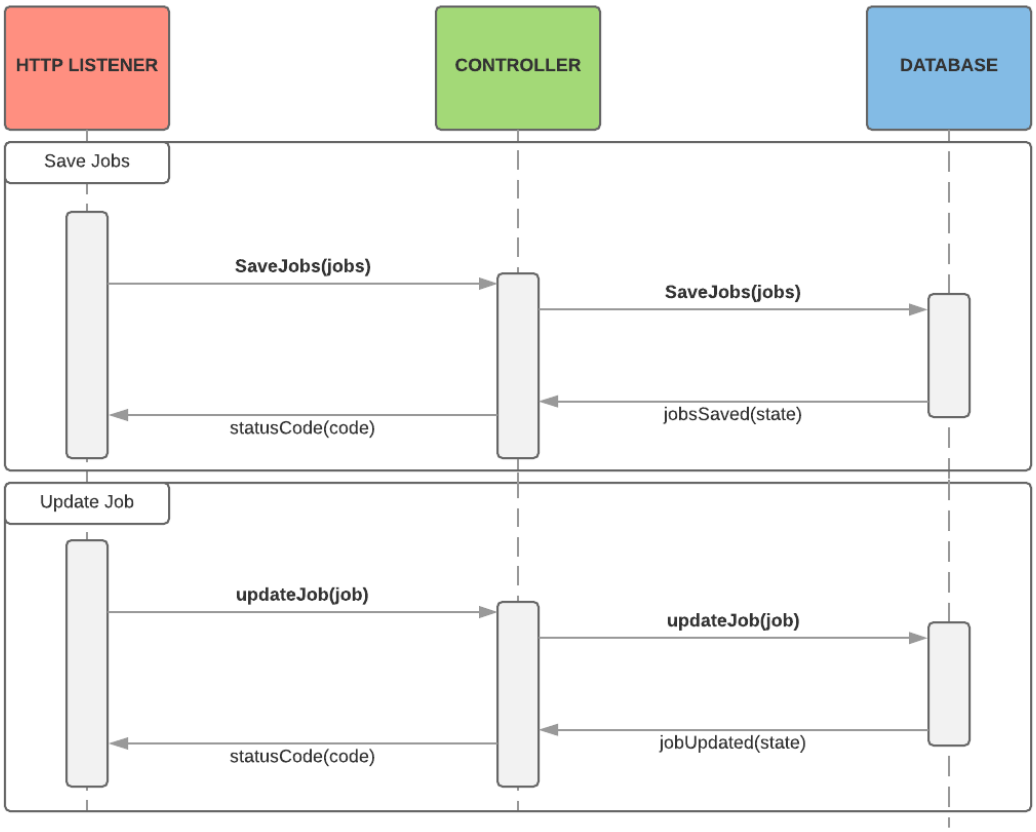


Ilustración 22: Diagrama de secuencia Jobs Saver

Estas tareas se almacenan en base de datos porque se quiere realizar un seguimiento de su cumplimiento. Por este motivo, uno de los campos almacenados debe de ser el estado de la tarea, y cuando se haya cumplido, el **Orchestrator** deberá de ponerse en contacto otra vez con este microservicio para actualizar el estado de cumplimiento de la tarea.

El número de tareas creado es muy grande. Teniendo en cuenta que una ejecución para cumplir con la normativa **FRTB** puede necesitar fácilmente realizar la calibración de la volatilidad de 400 subyacentes en una ventana temporal de 10 años, el número de tareas creado sería el siguiente:

$$400\ subyacentes \times 10\ años\ \times 251\ días = 1.004.000\ de\ tareas$$

Comunicación con otros microservicios:

Este microservicio sólo se conectará con el **Orchestrator** y, por tanto, solo tendrá esta conexión, que se realizará mediante **HTTP.**

5.1.2.6. Calibrator

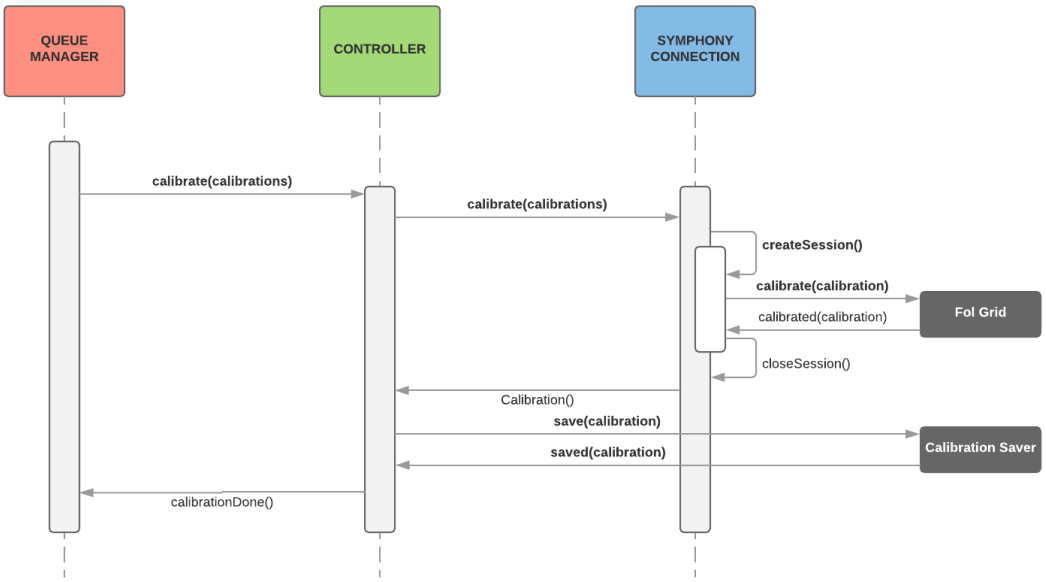


Ilustración 23: Diagrama de secuencia Callibrator

Este microservicio también es de vital importancia para la realización de la tarea de calibración. Su función será recoger las tareas de calibración creadas por **Jobs Saver** y enviárselas al **Fol Grid** para que las realice.

Este servicio debe de ser lo más rápido posible, por ello, se creará un pool de N hilos de ejecución (por ejemplo 50) que, realizarán una comunicación síncrona con el **Fol Grid.**

Una vez obtenida la respuesta con la tarea completada, se almacenará la calibración utilizando el **Calibration Saver**. Por último, se enviará un mensaje a la cola **Calib Response Queue** en el que se incluirá la calibración realizada y los tiempos de ejecución de tal forma que el **Orchestrator** pueda actualizar el estado de estas tareas y proporcionar al **Task Manager** los tiempos de ejecución de cada tarea.

Comunicación con otros microservicios:

El **Calirator** se comunica con tres microservicios, el **Orchestrator**, el **Calibration Saver** y el **Fol Grid**.

La comunicación con el **Orchestrator** se realizará utilizando las colas **Calib Request Queue** y **Calib Response Queue**. Es muy importante que la comunicación se realice por colas ya que esto permite que se pueda hacer de forma asíncrona, y, teniendo en cuenta el número masivo de tareas a procesar, esto nos permite no tener que prodesarlas todas de golpe y hacerlo cuando el microservicio esté disponible para responder a más peticiones.

Por otro lado, la comunicación con el **Calibration Saver** se realizará mediante el protocolo **HTTP** y, finalmente, la comunicación con el **Fol Grid** se hará por **TCP**.

5.1.2.7. Fol Grid

Este microservicio es el que realizará finalmente las tareas de calibración de volatilidades con los parámetros de calibración obtenidos para cada subyacente.

Para realizar la calibración de la volatilidad se utilizan cálculos matemáticos muy complejos que hacen que la carga soportada por este microservicio sea muy alta. De hecho, se estima

que se tararía aproximadamente un segundo en realizar la calibración de la volatilidad de un subyacente para un día determinado. Esto es mucho tiempo, sobre todo teniendo en cuenta que cada ejecución puede contar con más de un millón de tareas de calibración.

Por este motivo, este microservicio estará orientado a obtener el máximo rendimiento para realizar las calibraciones. Estará compuesto por un cluster de N servidores sobre el que correrá una aplicación creada con el lenguaje de programación óptimo para este tipo de operaciones, que comúnmente es **C++** o **Python.**

Dado que este microservicio requiere de unas necesidades especiales de optimización de recursos, no se desplegará sobre la **plataforma NOVA,** cómo el resto de los microservicios, sino que se desplegará sobre la **plataforma IBM Symphony**, que es una plataforma que facilita el despliegue y administración de aplicaciones distribuidas. Los detalles de esta plataforma se explicarán más adelante con mayor detalle.

Comunicación con otros microservicios:

Este microservicio sólo se comunica con el **Calibrator** y, cómo ya se ha indicado, la conexión entre ellos se realizará mediante una conexión **TCP**.

5.1.2.8. Calibration Saver

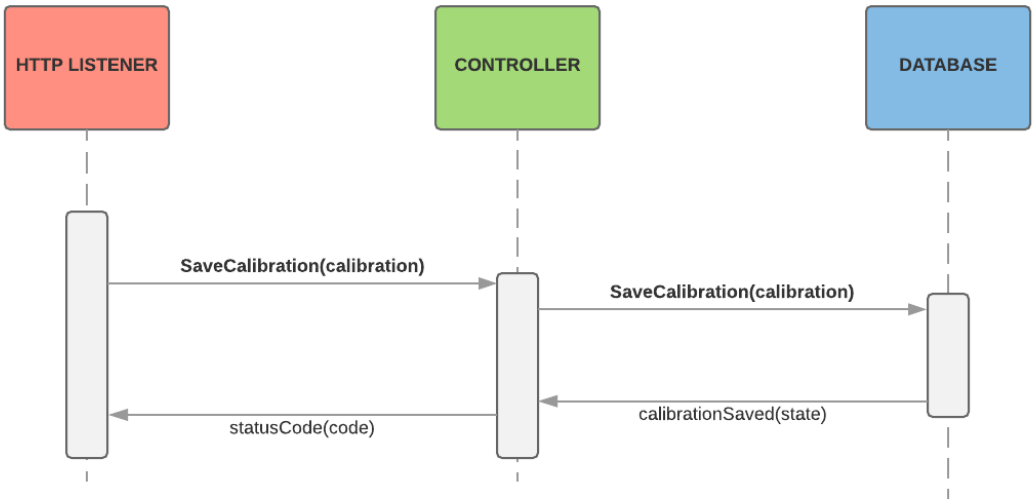


Ilustración 24: Diagrama de secuencia Calibration Saver

La función de este microservicio es especialmente simple, y consiste en almacenar las calibraciones simples obtenidas por el **Fol Grid**, Esto será especialmente útil cuando otras aplicaciones independientes quieran acceder a los resultados obtenidos por la ejecución de la aplicación.

Comunicación con otros microservicios:

El **Calibrator Saver** está conectado únicamente con el **Calibrator**, y la comunicación se realizará utilizando el protocolo **HTTP.**

5.1.2.9. Admin

Este microservicio está dedicado a supervisar el correcto funcionamiento de la aplicación. Debe de tener acceso a todas las colas utilizadas por el sistema y, en caso de que el usuario final lo solicite, o si detecta un error múltiple en el sistema, este microservicio podrá realizar una purga masiva de mensajes de las colas de tal forma que los microservicios dejen de procesar mensajes.

Los mensajes introducidos en las colas estarán identificados con un ID de tarea. De esta forma, el **Admin** podrá verificar de qué tarea es cada mensaje. Además, si se quiere cancelar alguna tarea en concreto, este microservicio podrá purgar todos los mensajes pertenecientes a esta tarea que estén pendientes de procesar.

5.2. DISEÑO DE CADA MICROSERVICIO PARA CUMPLIR CON LAS ESPECIFICACIONES INICIALES

En el apartado anterior se ha explicado funcionalmente el proyecto desarrollado, los microservicios que lo componen y la función que cada uno de ellos debe realizar. A continuación, se profundizará en el diseño de la aplicación, indicando los aspectos técnicos necesarios para el correcto funcionamiento de esta.

5.2.1. MODELO DE DATOS UTILIZADO

El modelo de datos es uno de los puntos más importantes a la hora de diseñar una aplicación con una arquitectura orientada a los microservicios. En este tipo de arquitecturas es muy complicado mantener la trazabilidad del flujo de ejecución.

Para solucionar esto, se necesita un servicio centralizado en el que se pueda guardar y actualizar la información referente a cada ejecución. Se puede optar tanto por un modelo relacional como un modelo no relacional como podría ser una **Mongo DB**.

En la aplicación desarrollada utilizaremos ambas, una base de datos relacional **PostrgreSQL**, y una base de datos no relacional **Mongo DB**. Se ha decidido utilizar una **Mongo DB** para almacenar la información de mercado de los activos subyacentes debido a que estos datos son desestructurados y, por tanto, difícilmente modelables en una base de datos relacional. Por otro lado, la información de los ajustes de calibración y otros datos internos de la aplicación se almacenarán en una base de datos **PostgreSQL**, ya que son datos estructurados y es muy importante mantener la integridad de los mismos.

Este ejemplo de las bases de datos refleja la flexibilidad que aportan las aplicaciones de microservicios, ya que para caso concreto se elige la opción óptima. Además, como cada base de datos es gestionada por microservicios diferentes, la decisión sobre la base de datos

elegida es transparente para el resto de los microservicios, y les es indiferente que motor de base de datos se está utilizando.

En la base de datos relacional se guardará la información referente a la ejecución, que será actualizada en cada proceso de ejecución de tal forma que, en caso de caída de la aplicación, se pueda saber en qué punto estaba la ejecución y, por tanto, podamos tener tolerancia a errores.

Tabla 1: Contenido de la base de datos PostgreSQL "Calibrador"

Esquema	Tabla	Contenido
Tasks	Tasks	Tareas de calibración programadas por el usuario
	Audit	Estado de ejecución de la tarea de calibración
Jobs	Jobs	Tasks segmentados en underlyings (Subyacentes)
	calibrations	Jobs segmentados en días (Unidad mínima de calibración)
Settings	Settings	Información de calibración única para cada subyacente
Authentication	Users	Usuarios dados de alta en la aplicación
	Groups	Grupos de usuarios dentro de la aplicación

En la **Tabla 1** se puede ver la división de la base de datos **PostgreSQL** utilizada en la aplicación, cuyo nombre es "**Calibrador**". Esta base de datos está alojada en un servidor de base de datos externo a la aplicación y el contenido de las tablas indicadas en la **Tabla 1** se indicará a continuación.

Tabla Tasks

Esta tabla está contenida dentro de esquema **Tasks** y en ella se guarda el contenido de las solicitudes de calibración solicitadas por los usuarios de la aplicación, así como el número de unidades mínimas de calibración pendientes.

Tabla 2: Contenido de la tabla "Task"

Índice	Fila	Típo	Comentario
PK	**id**	**integer**	**Id de la tarea**
FK	**userId**	**character varying(50)**	**Id del usuario**
	name	**character varying(50)**	**Nombre de la tarea**
	underlyings	**text**	**Subyacentes a calibrar la volatilidad**

	initdate	**timestamp**	**Fecha de inicio de la calibración**
	enddate	**timestamp**	**Fecha de fin de la calibración**
	arrivaldate	**timestamp**	**Fecha de llegada de la petición**
	lastupdate	**timestamp**	**Fecha de la última actualización recibida**
	status	**character varying(50)**	**Estado de la tarea**
	priority	**integer**	**Prioridad de la tarea [0-9]**
	pendingCalibrations	**integer**	**Calibraciones pendientes**

Tabla Audit

La tabla "**audit**" es una tabla dedicada exclusivamente al almacenamiento de los tiempos de ejecución de la aplicación. Al contrario que en la **Tabla Tasks**, donde sólo se mantiene el estado y los detalles del tiempo de ejecución de la última iteración sobre esa tarea, en la tabla **Audit** se almacena el timestamp de cada una de las fases de ejecución de la aplicación.

De esta forma, quedará constancia de cuánto tarda cada microservicio en realizar sus tareas y se podrá detectar fácilmente donde puede haber un cuello de botella en el flujo de ejecución de la aplicación.

Tabla 3: Contenido de la Tabla "audit"

Índice	Fila	Típo	Comentario
PK	**id**	**integer**	**Id de auditoría**
FK	**taskId**	**character varying(50)**	**Id de la tarea**
	phasename	**character varying(50)**	**Nombre de la tarea**
	initdate	**timestamp**	**Fecha de inicio de la calibración**
	enddate	**timestamp**	**Fecha de fin de la calibración**
	lastupdate	**timestamp**	**Fecha de la última actualización recibida**
	status	**character varying(50)**	**Estado de la tarea**

Tabla Jobs

En la **Tabla Tasks**, guardamos las calibraciones pedidas por los usuarios de la aplicación, que puede constar de varios subyacentes. En la tabla Jobs, sin embargo, se almacenan estas tareas segregadas por subyacente. Esto se hace así, porque en esta fase se almacenan en base de datos los ajustes de calibración, que son distintos para cada subyacente.

Tabla 4: Contenido de la tabla "Jobs"

Índice	Fila	Típo	Comentario
PK	**id**	**integer**	**Id del job**

FK	**taskId**	**character varying(50)**	**Id de la tarea**
FK	**userId**	**character varying(50)**	**Id del usuario**
	name	**character varying(50)**	**Nombre del job**
	underlying	**text**	**Subyacente a calibrar la volatilidad**
	initdate	**timestamp**	**Fecha de inicio de la calibración**
	enddate	**timestamp**	**Fecha de fin de la calibración**
	creationtime	**timestamp**	**Fecha de creación del job**
	lastupdate	**timestamp**	**Fecha de la última actualización recibida**
	status	**character varying(50)**	**Estado del job**
	priority	**integer**	**Prioridad del job[0-9]**
	settings	**text**	**JSON con los ajustes del subyacente**

Tabla Calibrations

Al igual que en la

Tabla Jobs, en la tabla **Calibrations** se vuelve a segregar la información. En este caso, se dividen los **Jobs** creados en unidades mínimas de calibración, que es la calibración de un subyacente en un día. Esto se hace de esta forma porque la información de mercado de los subyacentes varía en función del día, por lo que, en caso de fallo, esto nos permitirá retomar la calibración donde la habíamos dejado sin tener que volver a recoger la información de mercado.

Tabla 5: Contenido de la tabla "Calibrations"

Índice	Fila	Típo	Comentario
PK	**id**	**integer**	**Id del job**
FK	**taskId**	**character varying(50)**	**Id de la tarea**
	name	**character varying(50)**	**Nombre del job**
	underlying	**text**	**Subyacente a calibrar la volatilidad**
	date	**timestamp**	**Fecha de la calibración**
	creationtime	**timestamp**	**Fecha de creación del job**
	lastupdate	**timestamp**	**Fecha de la última actualización recibida**
	status	**character varying(50)**	**Estado del job**
	priority	**integer**	**Prioridad del job[0-9]**
	settings	**text**	**JSON con los ajustes del subyacente**

	marketdata	**text**	**JSON con la información de mercado**

Tabla Settings

En esta tabla se almacena la información de los ajustes de calibración de cada uno de los subyacentes. Los parámetros incluidos en la tabla de settings se pueden ver en la

Tabla 6.

Tabla 6: Contenido de la tabla settings

Settings		
PK	id	integer
AK	model_name	character varying(50)
	discard_arbitrable	numeric
	max_bid_ask_spread	numeric
	max_atm_bid_ask_spread	numeric
	merge_strike_threshold	numeric
	min_moneyiness_for_repo_calc	numeric
	max_moneyiness_for_repo_calc	numeric
	min_num_buckets	numeric
	min_num_options	numeric
	define_smooth	boolean
	force_monotonic_slope	boolean
	force_convex_slope	boolean
	slope_function_weight	numeric
	force_monotonic_put_curv	boolean
	force_convex_put_curv	boolean
	put_curv_function_weight	numeric
	force_monotonic_call_curv	boolean
	force_convex_call_curv	boolean
	call_curv_function_weight	numeric
	extrapolation	character varying(50)
	last_user	character varying(50)
	last_update	timestamp without time zone
	slope95_cap	numeric
	put_curve80_cap	numeric
	call_curv120_cap	numeric
	min_no_days	numeric
	cone_jump	numeric
	cone_std_dev	numeric
	strike_lwing	numeric
	strike_rwing	numeric
	dcut	numeric
	ucut	numeric
	premium95_percentile_bound	numeric
	rp_margin_type	character varying(15)
	rp_margin_value	numeric
	desk_id	character varying(25)
	calibrator_id	character varying(50)
	min_days_to_expiry	numeric
	totem_method	character varying(50)
	repo_management	character varying(50)

Datos recibidos por parte del microservicio Market Data

La aplicación se conecta al microservicio **Market Data** para pedir los datos de mercado de cada subyacente. Este microservicio se conecta a su vez con un una base de datos **MongoDB**, no relacional que también es externa al proyecto. La información devuelta por este microservicio está pactada desde el principio de la ejecución del partido y tiene la siguiente estructura de datos.

5.2.2. TECNOLOGÍA DE COLAS

Cómo ya se ha comentado anteriormente, la arquitectura empleada utiliza colas para la transmisión de mensajes. Esto permite enviar mensajes asíncronos de tal forma que la falta de potencia o de memoria de un microservicio no afecte al rendimiento de otro.

Así, cuando un microservicio genera un mensaje, no tiene que preocuparse de si el receptor está listo para recibirlo ni esperar la respuesta, sino que simplemente debe depositarlo en las colas. Por otro lado, las colas utilizadas admiten prioridades de tal forma que los primeros mensajes que un microservicio reciba de una cola sean los de mayor prioridad.

Las colas utilizadas para este propósito son colas Tibco funcionando sobre la arquitectura del banco BBVA.

5.2.3. DISEÑO ESPECÍFICO DE CADA MICROSERVICIO

Ya hemos hablado de la funcionalidad que debe de tener cada microservicio, y sobre la arquitectura que los acompañará para asegurar su correcto funcionamiento. En este apartado, se detallará el diseño de cada microservicio para cumplir con los requerimientos expuestos en el apartado **5.1**.

5.2.3.1. Plataforma y lenguaje utilizados

El lenguaje de programación utilizado es una decisión muy importante a la hora de diseñar cualquier aplicación, ya que, dependiendo de las características de esta, es más recomendable utilizar un lenguaje u otro. Por ejemplo, los lenguajes más apropiados para desarrollar una aplicación que requiera realizar un gran número de operaciones matemáticos son **C** o **C++**,

ya que son lenguajes compilados, mucho más eficientes que otros lenguajes interpretados o parcialmente interpretados como **Java**, que, sin embargo, son mucho más apropiados para el desarrollo de proyectos multiplataforma.

En un proyecto con una arquitectura orientada a microservicios es muy importante analizar cada uno de los microservicios para comprobar si tiene alguna característica especial que haga que sea recomendable la utilización de un lenguaje de programación frente a otro. Por otro lado, si ninguno de los microservicios tiene alguna necesidad especial, es altamente recomendable utilizar el mismo lenguaje para todos dado que esto facilita en gran medida la administración y el mantenimiento de la aplicación.

En el proyecto desarrollado, formado por nueve microservicios, sólo se han detectado necesidades especiales en el microservicio **Fol Grid**, que precisa de un lenguaje óptimo, ya que será el encargado de realizar todas las tareas de calibración de volatilidades, teniendo que realizar cálculos matemáticos muy complejos. Por este motivo, este microservicio estará desarrollado en **C++** y se implementará sobre un Grid utilizando la tecnología de la **Plataforma Symphony**.

El resto de los microservicios serán desplegados sobre la **Plataforma Nova**, que como ya se ha comentado anteriormente es una plataforma enfocada al microservicio. El lenguaje de desarrollo utilizado será **Java**, ya que es un lenguaje muy robusro, que facilita mucho el desarrollo debido a que el conjunto del equipo de desarrollo tiene más experiencia en la utilización de este lenguaje.

En la **Tabla 7** podemos ver un resumen de los lenguajes de programación y las plataformas de desarrollo utilizadas por los microservicios.

Tabla 7: Lenguajes y plataformas utilizadas por los microservicios

Microservicio	Lenguaje	Plataforma
Admin	Java	Nova
Task Manager	Java	Nova
Orchestrator	Java	Nova
Jobs Saber	Java	Nova
Market Data	Java	Nova
Settings	Java	Nova
Calibrator	Java	Nova
Calibrator Saber	Java	Nova
Fol Grid	C++	Symphony

5.2.3.2. Inputs y Outputs. Especificación de las peticiones HTTP

Después de la etapa de diseño de la aplicación, un microservicio debe de poder ser desarrollado por un equipo de desarrollo que no conozca el funcionamiento ni los detalles de diseño del resto de microservicios. Para este equipo de desarrollo, el resto de los microservicios deben de ser cómo cajas negras de las que sólo se sabe el tipo de petición que requieren para empezar a funcionar, la información que se le debe de enviar y, en caso de que devuelva información, la información devuelta.

De esta forma, es muy importante definir los inputs y los outputs de cada microservicio y esto es lo que se definirá en este apartado.

En primer lugar, se definirán las peticiones **HTTP** que deben de admitir los microservicios.

TASK MANAGER

El task manager es el microservicio encargado de interactuar con el usuario final, por lo que la mayoría de las peticiones **HTTP** de las que disone están enfocadas a esta función.

- **Launch new task:**
  - **URL:** http://{{host}}:{{launcherport}}/taskmanager/
  - **Tipo de petición:** POST
  - **Tipo de contenido:** application/json

- **Plantilla del cuerpo del mensaje:**

```
[{
    "name": "Task 1",
    "priority": 9,
    "user": {
        "id": "xe72389",
        "name": "Pablo",
        "surname": "Iglesia"
    },
    "underlyings": ["EUR-BBV1.MC","EUR-BBV2.MC"],
    "initDate": 1514792548000,
    "endDate": 1514792548000
},
{
    "name": "Task 2",
    ...
},

...

{
    "name": "Task N",
    ...
}]
```

- **Respuesta obtenida:** Sin respuesta. Sólo código HTTP.

- **Geta all Tasks:**
  - **URL:** http://{{host}}:{{launcherport}}/taskmanager/
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
[{
    "name": "Task 1",
    "priority": 9,
    "user": {
        "id": "xe72389",
        "name": "Pablo",
        "surname": "Iglesia"
    },
    "underlyings": ["EUR-BBV1.MC","EUR-BBV2.MC"],
    "initDate": 1514792548000,
```

```
    "endDate": 1514792548000
},
{
    "name": "Task 2",
    ...
},

...

{
    "name": "Task N",
    ...
}]
```

SETTINGS

El propósito de la comunicación **HTTP** de este microservicio es que el **Orchestrator** pueda pedir y recibir los ajustes de calibración de un subyacente. Además, el administrador de la aplicación también debe de poder añadir, borrar o modificar contenido.

- **Get all Settings:**
  - **URL:** http://{{host}}:{{settingsport}}/setting/
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
[{
        "modelName": "testmodel",
        "discardArbitrable": 1,
        "maxBidAskSpread": 0.25,
        "maxAtmBidAskSpread": -1,
        "mergeStrikeThreshold": 0.005,
        "minMoneyinessForRepoCalc": 0.9,
        "maxMoneyinessForRepoCalc": 1.1,
        "minNumBuckets": 4,
        "minNumOptions": 6,    "initDate": 1514792548000,
         "endDate": 1514792548000
         ...
},
{
        "modelName": "testmode2",
        ...
```

```
}

...

{
        "modelName": "testmodelN",
        ...
}]
```

- **Set new Settings:**
  - **URL:** http://{{host}}:{{settingsport}}/setting/
  - **Tipo de petición:** POST
  - **Tipo de contenido:** application/json
  - **Plantilla del cuerpo del mensaje:**

```
[{
        "modelName": "testmodel",
        "discardArbitrable": 1,
        "maxBidAskSpread": 0.25,
        "maxAtmBidAskSpread": -1,
        "mergeStrikeThreshold": 0.005,
        "minMoneyinessForRepoCalc": 0.9,
        "maxMoneyinessForRepoCalc": 1.1,
        "minNumBuckets": 4,
        "minNumOptions": 6,    "initDate": 1514792548000,
         "endDate": 1514792548000
         ...
},
{
        "modelName": "testmode2",
        ...
}

...

{
        "modelName": "testmodelN",
        ...
}]
```

  - **Respuesta obtenida:** Sin respuesta. Sólo código HTTP.
  -

- **Get setting by id:**
  - **URL:** http://{{host}}:{{settingsport}}/setting/{{settingId}}
  - **Tipo de petición:** GET

- **Tipo de contenido:** --
- **Plantilla del cuerpo del mensaje:** --
- **Respuesta obtenida:**


```
{
        "modelName": "testmodel",
        "discardArbitrable": 1,
        "maxBidAskSpread": 0.25,
        "maxAtmBidAskSpread": -1,
        "mergeStrikeThreshold": 0.005,
        "minMoneyinessForRepoCalc": 0.9,
        "maxMoneyinessForRepoCalc": 1.1,
        "minNumBuckets": 4,
        "minNumOptions": 6,    "initDate": 1514792548000,
         "endDate": 1514792548000
         ...
}
```


- **Get setting by modelname:**
  - **URL:**

    http://{{host}}:{{settingsport}}/setting/?modelname={{modelname}}
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**


```
{
        "modelName": "testmodel",
        "discardArbitrable": 1,
        "maxBidAskSpread": 0.25,
        "maxAtmBidAskSpread": -1,
        "mergeStrikeThreshold": 0.005,
        "minMoneyinessForRepoCalc": 0.9,
        "maxMoneyinessForRepoCalc": 1.1,
        "minNumBuckets": 4,
        "minNumOptions": 6,    "initDate": 1514792548000,
         ...
}
```


- **Modify setting by id:**
  - **URL:** http://{{host}}:{{settingsport}}/setting/{{settingId}}
  - **Tipo de petición:** PUT
  - **Tipo de contenido:** --

- **Plantilla del cuerpo del mensaje:**

```
{
        "modelName": "testmodel",
        "discardArbitrable": 1,
        "maxBidAskSpread": 0.25,
        "maxAtmBidAskSpread": -1,
        "mergeStrikeThreshold": 0.005,
        "minMoneyinessForRepoCalc": 0.9,
        "maxMoneyinessForRepoCalc": 1.1,
        "minNumBuckets": 4,
        "minNumOptions": 6,    "initDate": 1514792548000,
         ...
}
```

- **Respuesta obtenida:** Código de confirmación HTTP

- **Modify setting by id:**
  - **URL:** http://{{host}}:{{settingsport}}/setting/{{settingId}}
  - **Tipo de petición:** DEL
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:** Código de confirmación HTTP

JOB STATUS

Como ya se ha mencionado anteriormente, un job es una tarea de calibración dividida por subyacentes de tal forma que se pueda almacenar conjuntamente la tarea con loajustes de calibración de dicho subyacente. Además, este microservicio también será el encargado de almacenar las unidades mínimas de calibración. De esta forma, el servicio **HTTP** de este microservicio está principalmente destinado a interactuar con el **Orchestrator.**

- **Save Jobs:**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/
  - **Tipo de petición:** POST
  - **Tipo de contenido:** application/json

- **Plantilla del cuerpo del mensaje:**

```
[
    {
        "taskName": "Task1",
        "underlying": "bbva1",
        "priority": 0,
        "jobGroupId": 1272,
        "user": {
            "id": "E043891",
            "name": null,
            "surname": null
        },
        "setting": {
               "id": 1,
               "modelName": "DEFAULT",
               "discardArbitrable": 1,
               "maxBidAskSpread": 0.25,
               "maxAtmBidAskSpread": -1,
                ...
         },
        "creationdate": 1527585438123,
        "lastupdate": 1527585438123,
        "initdate": 1527838948000,
        "enddate": 1527838948000,
        "status": "init",
        "underliyingName": "1mes 1valor"
    },
    {
        "taskName": "Task1",
        "underlying": "bbva2",

        ...
    },

       ...

    {
        "taskName": "Task1",
        "underlying": "bbvaN",
        ...
    }
]
```

  - **Respuesta obtenida:** Sin respuesta. Sólo código HTTP.

- **<u>Save Calibrations:</u>**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/calibration/
  - **Tipo de petición:** POST
  - **Tipo de contenido:** application/json

- **Plantilla del cuerpo del mensaje:**

```
[
    {
        "jobid": 1234,
        "taskid": 1313,
        "underliying": "10años 4valores nova",
        "settings": null,
        "marketdata": null,
        "date": 1527804000000,
        "creationdate": 1527840523598,
        "lastupdate": 1527687822689,
        "status": "done",
        "jobsByJobid": null
    },
    {
        "jobid": 1235,
        "taskid": 1313,
        ...
    },

       ...

    {
        "jobid": N,
        "taskid": 1313,
              ...
    }
]
```

- **Respuesta obtenida:** Sin respuesta. Sólo código HTTP.

- **Get all Jobs:**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
[
    {
 "taskName": "Task1",
        "underlying": "bbva1",
        "priority": 0,
        "jobGroupId": 1272,
        "user": {
            "id": "E043891",
```

```
            "name": null,
            "surname": null
        },
        "setting": {
               "id": 1,
                "modelName": "DEFAULT",
               "discardArbitrable": 1,
               "maxBidAskSpread": 0.25,
                ...
  },
        "creationdate": 1527585438123,
        "lastupdate": 1527585438123,
        "initdate": 1527838948000,
        "enddate": 1527838948000,
        "status": "init",
        "underliyingName": "1mes 1valor"
    },
    {
 "taskName": "Task1",
        "underlying": "bbva2",

        ...
    },

...

    {
 "taskName": "Task1",
        "underlying": "bbvaN",
        ...
    }
]
```


- **Get all Calibrations:**
    - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/calibration/
    - **Tipo de petición:** GET
    - **Tipo de contenido:** --
    - **Plantilla del cuerpo del mensaje:** --
    - **Respuesta obtenida:**


```
[
    {
        "jobid": 1234,
        "taskid": 1313,
        "underliying": "10años 4valores nova",
        "settings": null,
        "marketdata": null,
        "date": 1527804000000,
        "creationdate": 1527840523598,
```

```
        "lastupdate": 1527687822689,
        "status": "done",
        "jobsByJobid": null
    },
    {
 "jobid": 1235,
        "taskid": 1313,
        ...
    },

...

    {
 "jobid": N,
        "taskid": 1313,
        ...
    }
]
```


- **Get Job by id:**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/{{jobId}}
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**


```
{
         "taskName": "Task1",
        "underlying": "bbva1",
        "priority": 0,
        "jobGroupId": 1272,
        "user": {
            "id": "E043891",
            "name": null,
            "surname": null
        },
        "setting": {
               "id": 1,
                "modelName": "DEFAULT",
               "discardArbitrable": 1,
               "maxBidAskSpread": 0.25,
                ...
         },
        "creationdate": 1527585438123,
        "lastupdate": 1527585438123,
        "initdate": 1527838948000,
        "enddate": 1527838948000,
```

```
        "status": "init",
        "underliyingName": "1mes 1valor"
}
```

- **Get Calibration by id:**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/calibration/{{calibrationId}}
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
{
    "id": 5,
    "jobid": 152,
    "userid": "E043891",
    "name": "primerLanzamiento",
    "underliying": "EUR-BBV.MC",
    "initdate": 1523433072840,
    "enddate": 1523515583459,
    "creationdate": 1523514990660,
    "lastupdate": 1523514990660,
    "status": "scheduled"
}
```

- **Modify Job by Id:**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/{{jobId}}
  - **Tipo de petición:** PUT
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:**

```
{
        "taskName": "Task1",
        "underlying": "bbva1",
        "priority": 0,
        "jobGroupId": 1272,
        "user": {
            "id": "E043891",
            "name": null,
            "surname": null
        },
        "setting": {
              "id": 1,
```

```
                "modelName": "DEFAULT",
                "discardArbitrable": 1,
                "maxBidAskSpread": 0.25,
                ...
        },
        "creationdate": 1527585438123,
        "lastupdate": 1527585438123,
        "initdate": 1527838948000,
        "enddate": 1527838948000,
        "status": "init",
        "underliyingName": "1mes 1valor"
}
```


- **Respuesta obtenida:** Código **HTTP.**

- **Modify Calibration by id:**
  - **URL:**

    http://{{host}}:{{jobstatusport}}/jobssaver/calibration/{{calibrationId}}
  - **Tipo de petición:** PUT
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:**


```
{
    "id": 5,
    "jobid": 152,
    "userid": "E043891",
    "name": "primerLanzamiento",
    "underliying": "EUR-BBV.MC",
    "initdate": 1523433072840,
    "enddate": 1523515583459,
    "creationdate": 1523514990660,
    "lastupdate": 1523514990660,
    "status": "scheduled"
}
```


  - **Respuesta obtenida:** Código **HTTP**.

- **Delete Job by id**
  - **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/{{jobId}}
  - **Tipo de petición:** DEL
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:** Código **HTTP**.

- **Delete Calibration by id:**

- **URL:** http://{{host}}:{{jobstatusport}}/jobssaver/calibration/{{calibrationId}}
- **Tipo de petición:** DEL
- **Tipo de contenido:** --
- **Plantilla del cuerpo del mensaje:** --
- **Respuesta obtenida:** Código **HTTP**.

ADMIN

El microservicio **Admin**, es el encargado de comprobar si el flujo de la aplicación está funcionando correctamente. Por el tipo de sistema que se está implementando, con varias colas de mensajes activas, es necesario un microservicio como este, que pueda comprobar el contenido de estas colas y purgarlo si se cancela una ejecución o si no está siendo consumido por los microservicios.

- **Get all messages:**
  - **URL:** http://{{host}}:{{adminport}}/admin/
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
[
	{
		"queue": "X"
		"message": {
		...
		}
	},
	{
		"queue": "Y"
		"message": {
		...
		}
	},
```

```
        ...

        {
                "queue": "Z"
                "message": {
                ...
                }
        }
]
```

- **Get all messages by queue name:**
  - **URL:** http://{{host}}:{{adminport}}/admin/{{queueName}}
  - **Tipo de petición:** GET
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:**

```
[
        {
                "queue": "{{queueName}}"
                "message": {
                ...
                }
        },
        {
                "queue": "{{queueName}}"
                "message": {
                ...
                }
        },

        ...

        {
                "queue": "{{queueName}}"
                "message": {
                ...
                }
        }
]
```

- **Purge all Queues:**
  - **URL:** http://{{host}}:{{adminport}}/admin/
  - **Tipo de petición:** DEL
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --

- **Respuesta obtenida:** Código **HTTP**.

- **Purge queue by queue name:**
  - **URL:** http://{{host}}:{{adminport}}/admin/{{queueName}}
  - **Tipo de petición:** DEL
  - **Tipo de contenido:** --
  - **Plantilla del cuerpo del mensaje:** --
  - **Respuesta obtenida:** Código **HTTP**.

Una vez especificadas las peticiones **HTTP** realizadas por cada microservicio, se detallará a continuación el contenido de los mensajes depositados en las colas.

TASK QUEUE

Esta cola es la cola utilizada por el **Task Manager** para depositar las tareas de calibración de volatilidades solicitadas por el usuario. De esta forma, la información depositada en la cola tendría la siguiente estructura:

```
{
    "name": "Task 1",
    "priority": 9,
    "user": {
        "id": "xe72389",
        "name": "Pablo",
        "surname": "Iglesia"
    },
    "underlyings": ["EUR-BBV1.MC","EUR-BBV2.MC"],
    "initDate": 1514792548000,
    "endDate": 1514792548000
}
```

ADMIN QUEUE

En esta cola se almacenan los tiempos de ejecución de cada una de las fases de ejecución de la aplicación. Estos tiempos son depositados en esta cola por el varios microservicios con la siguiente estructura:

```
{
    "auditId": 12,
    "stepPhase": "Calibration",
```

```
    "stepAditionalInfo": {
       ...
    }
    "initAuditableDateTime": 1233321155688788,
    "endAuditableDateTime": 1233321155652316
}
```

CALIBRATION REQUEST QUEUE

En esta cola se almacenan las tareas simples de calibración. Estas calibraciones son generadas y depositadas en la cola por el **Orchestrator**. Las calibraciones son depositadas con la siguiente estructura:

```
{
    "id": 5,
    "jobid": 152,
    "userid": "E043891",
    "name": "primerLanzamiento",
    "underliying": "EUR-BBV.MC",
    "initdate": 1523433072840,
    "enddate": 1523515583459,
    "creationdate": 1523514990660,
    "lastupdate": 1523514990660,
    "status": "scheduled"
}
```

CALIBRATION RESPONSE QUEUE

Finalmente, el **Calibrator** devuelve a la cola **Calibration response Queue** el estado de las calibraciones realizadas, tanto si han sido exitosas cómo fallidas, de tal forma que el sistema tenga información del job realizado y pueda tomar las decisiones necesarias para completarlo de forma exitosa. El contenido del mensaje sería el siguiente:

```
{
    "id": 5,
    "jobid": 152,
    "status": "done/failed"
}
```

Por último, es importante detallar un caso especial que existe dentro de la arquitectura. Este caso es la comunicación entre el **Calibrator** y el **Fol Grid**. Estos dos microservicios crean

una conexión **TCP** entre sí de tal forma que la comunicación entre ambos es directa. Cómo es de esperar, se crearán tantas conexiones como instancias del microservicio **Calibrator** estén levantadas.

Cómo ya se ha comentado anteriormente, el **Fol Grid** es el único microservicio de la arquitectura que no está desarrollado en Java. De esta forma, es muy importante que la información transmitida esté serializada con una estructura conocida e interpretable por ambos microservicios. Aunque inicialmente se comenzó utilizando la serialización facilitada por la **Plataforma Symphony**, esta estaba limitada a datos primitivos y finalmente se optó por crear un mensaje **JSON** con la siguiente estructura:


```
{
  "priority": 0,
  "name": "ARS BASIS EUR",
  "currency": null,
  "spot": {
    "value": 6075.41,
    "localdate": {
      "year": 2018,
      "month": "JANUARY",
      "chronology": {
        "id": "ISO",
        "calendarType": "iso8601"
      },
      "era": "CE",
      "dayOfMonth": 1,
      "dayOfWeek": "MONDAY",
      "dayOfYear": 1,
      "leapYear": false,
      "monthValue": 1
    }
  },
  "liquidity": {
    "value": 7.0E8
  },
  "dividends": [
    {
      "ex_date": {
        "year": 2026,
        "month": "JANUARY",
        "chronology": {
          "id": "ISO",
          "calendarType": "iso8601"
        },
        "era": "CE",
        "dayOfMonth": 25,
```

```
        "dayOfWeek": "SUNDAY",
        "dayOfYear": 25,
        "leapYear": false,
        "monthValue": 1
      },
      "pay_date": {
        "year": 2026,
        "month": "JANUARY",
        "chronology": {
          "id": "ISO",
          "calendarType": "iso8601"
        },
        "era": "CE",
        "dayOfMonth": 25,
        "dayOfWeek": "SUNDAY",
        "dayOfYear": 25,
        "leapYear": false,
        "monthValue": 1
      },
      "relative": false,
      "net_value": 1.2
    }
  ],
  "repo": {
    "buckets": [
      {
        "date": "2018-01-30",
        "df": 1.0000123288431229
      },
      ...
      {
        "date": "2022-12-15",
        "df": 1.0405997818901656
      }
    ]
  },
  "vol_orc": {
    "orc_buckets": [
      {
        "date": {
          "year": 2018,
          "month": "FEBRUARY",
          "chronology": {
            "id": "ISO",
            "calendarType": "iso8601"
          },
          "era": "CE",
          "dayOfMonth": 16,
          "dayOfWeek": "FRIDAY",
          "dayOfYear": 47,
          "leapYear": false,
          "monthValue": 2
        },
```

```
      "vol_ref": 0.100844892428,
      "slope_ref": 0.100844892428,
      "put_curv": 0.100844892428,
      "call_curv": 0.100844892428,
      "down_cut": 0.100844892428,
      "up_cut": 0.100844892428,
      "ref_fwd": 0.100844892428,
      "down_smooth": 0.100844892428,
      "up_smooth": 0.100844892428
    },
    ...
    {
      "date": {
        "year": 2018,
        "month": "DECEMBER",
        "chronology": {
          "id": "ISO",
          "calendarType": "iso8601"
        },
        "era": "CE",
        "dayOfMonth": 20,
        "dayOfWeek": "THURSDAY",
        "dayOfYear": 354,
        "leapYear": false,
        "monthValue": 12
      },
      "vol_ref": 0.100844892428,
      "slope_ref": 0.100844892428,
      "put_curv": 0.100844892428,
      "call_curv": 0.100844892428,
      "down_cut": 0.100844892428,
      "up_cut": 0.100844892428,
      "ref_fwd": 0.100844892428,
      "down_smooth": 0.100844892428,
      "up_smooth": 0.100844892428
    }
  ],
  "atm_buckets": [
    {
      "date": {
        "year": 2018,
        "month": "FEBRUARY",
        "chronology": {
          "id": "ISO",
          "calendarType": "iso8601"
        },
        "era": "CE",
        "dayOfMonth": 16,
        "dayOfWeek": "FRIDAY",
        "dayOfYear": 47,
        "leapYear": false,
        "monthValue": 2
      },
```

```
        "strike": 110.0,
        "vol": 0.2
      },
     ...
      {
        "date": {
          "year": 2018,
          "month": "DECEMBER",
          "chronology": {
            "id": "ISO",
            "calendarType": "iso8601"
          },
          "era": "CE",
          "dayOfMonth": 20,
          "dayOfWeek": "THURSDAY",
          "dayOfYear": 354,
          "leapYear": false,
          "monthValue": 12
        },
        "strike": 110.0,
        "vol": 0.2
      }
    ]
  },
  "discrete_vol": {
    "volatilityBuckets": [
      {
        "date": {
          "year": 2024,
          "month": "DECEMBER",
          "chronology": {
            "id": "ISO",
            "calendarType": "iso8601"
          },
          "era": "CE",
          "dayOfMonth": 28,
          "dayOfWeek": "SATURDAY",
          "dayOfYear": 363,
          "leapYear": true,
          "monthValue": 12
        },
        "points": [
          {
            "strike": 110.0,
            "vol": 0.2
          },
          ...
          {
            "strike": 110.0,
            "vol": 0.2
          }
        ]
      },
```

```
      ...
      {
        "date": {
          "year": 2026,
          "month": "JANUARY",
          "chronology": {
            "id": "ISO",
            "calendarType": "iso8601"
          },
          "era": "CE",
          "dayOfMonth": 25,
          "dayOfWeek": "SUNDAY",
          "dayOfYear": 25,
          "leapYear": false,
          "monthValue": 1
        },
        "points": [
          {
            "strike": 110.0,
            "vol": 0.2
          },
          {
            "strike": 110.0,
            "vol": 0.2
          },
          {
            "strike": 110.0,
            "vol": 0.2
          }
        ]
      }
    ]
  },
  "localdate": {
    "year": 2018,
    "month": "JANUARY",
    "chronology": {
      "id": "ISO",
      "calendarType": "iso8601"
    },
    "era": "CE",
    "dayOfMonth": 1,
    "dayOfWeek": "MONDAY",
    "dayOfYear": 1,
    "leapYear": false,
    "monthValue": 1
  }
}
```

5.3. INTEGRACIÓN DE LOS MICROSERVICIOS

La integración de una aplicación con una arquitectura orientada a los microservicios es uno de los puntos más complejos del desarrollo de esta ya que no se trata del despliegue de una sola aplicación, sino que hay que desplegar todos los microservicios por separado.

Esto supone que en el momento del despliegue no solo hay que comprobar el correcto comportamiento de cada uno de los microservicios por separado, sino que también será necesario comprobar al comportamiento de todos ellos en su conjunto que, al fin y al cabo, es lo que aporta la funcionalidad a la aplicación.

5.3.1. PLATAFORMA NOVA

Para realizar el despliegue conjunto de los microservicios son muy útiles las plataformas orientadas a los microservicios, como la **Plataforma Nova,** que nos facilita entorno único sobre el que desplegar cada uno de los microservicios.

Una de las herramientas más importantes y necesarias que aporta esta plataforma es la plataforma local de desarrollo, que nos ofrece, en nuestro host local, un entorno similar al que se encontrarán las aplicaciones en producción, ofreciendo un servidor **Tomcat** para cada uno de los microservicios.

Esta plataforma ofrece menos limitaciones que la plataforma de producción, ya que las aplicaciones desplegadas sobre la plataforma de producción están encapsuladas en un contenedor **Docker** mientras que en la plataforma local de Nova no. Por este motivo, es muy importante comprender todas las limitaciones que tiene la **Plataforma Nova**, de tal forma que a la hora de subir la aplicación a producción sea compatible con la plataforma.

Creación de un microservicio

Las aplicaciones desplegadas en la **Plataforma Nova** deben de tener unas características determinadas. La aplicación debe de estar desarrollada en Java con el software de gestión y construcción de proyectos **Apache Maven**. Además, la clase principal del proyecto debe de incluir y utilizar algunas dependencias propias de la plataforma.

Para asegurarnos de que la aplicación creada tiene la estructura necesaria para su correcto funcionamiento sobre la **Plataforma Nova**, esta ofrece un generador de código que nos permite crear un proyecto con la estructura y las dependencias necesarias para que pueda funcionar sobre la plataforma.

En la **Ilustración 25** podemos ver el generador de código, que es muy fácil de utilizar.

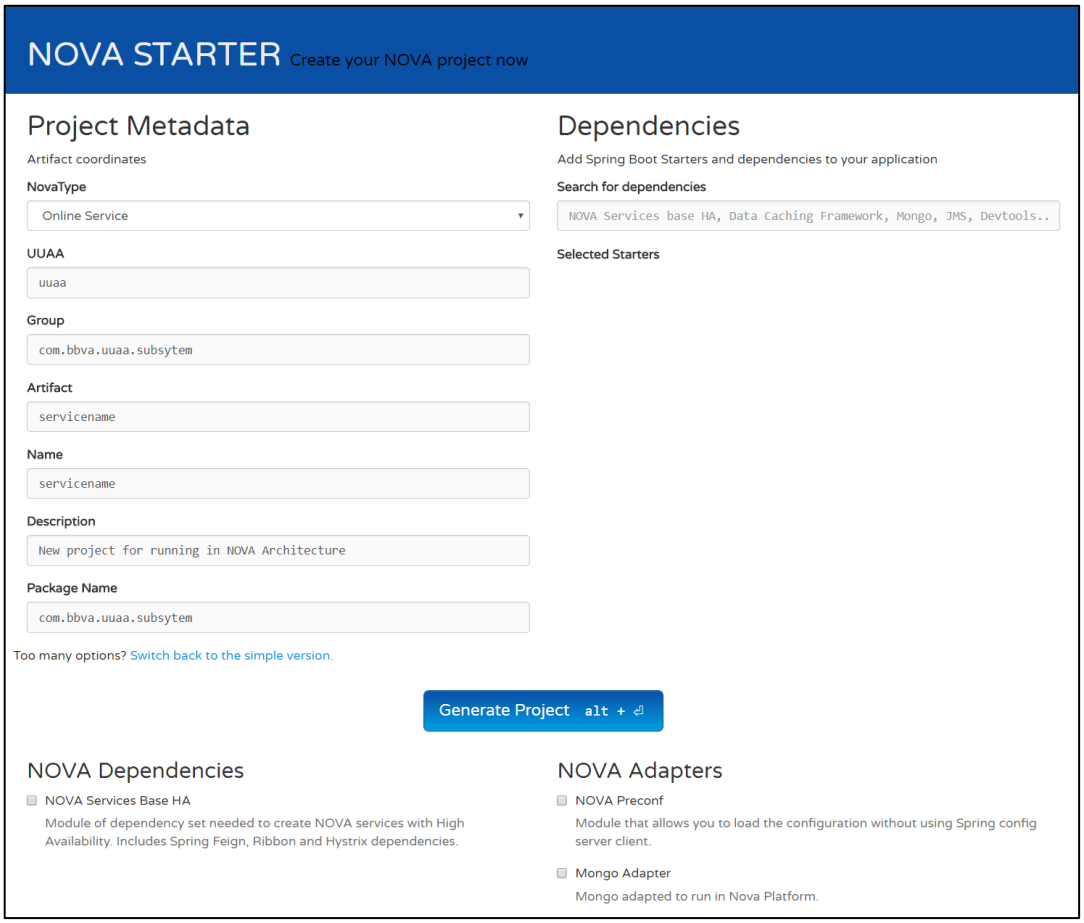


Ilustración 25: Generador de código de la Plataforma Nova

Despliegue en de la plataforma

La **Plataforma Nova** tiene una jerarquía que es necesario respetar a la hora de desplegar proyectos sobre ella. Esta jerarquía es la mostrada en la **Ilustración 26**.

Según esta ilustración se pueden deducir los pasos a seguir para crear o actualizar un microservicio sobre la plataforma. El primer paso sería crear un producto, para ello, hay que seguir la burocracia interna del banco, ya que ese producto tiene que ser creado por los encargados de la plataforma.

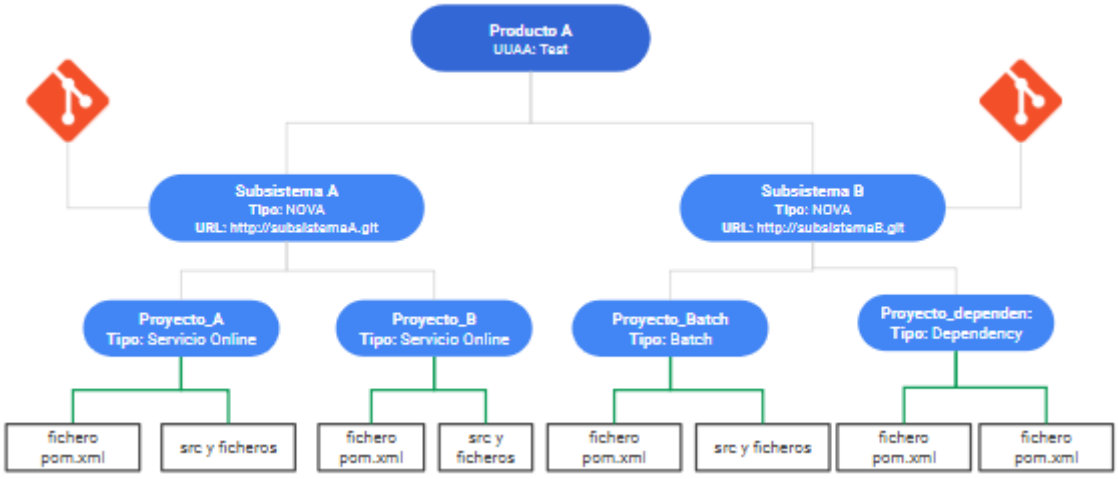


Ilustración 26: Jerarquía de la Plataforma Nova

Una vez creado el producto, ya se puede operar libremente sobre la plataforma con la única restricción del presupuesto asignado al proyecto. De esta forma, el siguiente paso será crear un subsistema para cada uno de los microservicios que se quieran desplegar. Un producto puede tener varios subsistemas (en nuestro caso tendría nueve), y cada uno de estos subsistemas debe de tener un repositorio **Git** asignado.

Finalmente, el último paso a seguir es la creación de los proyectos. Un proyecto es la unidad de ejecución en la plataforma Nova, hasta ahora, la jerarquía que hemos seguido es meramente organizativa, pero con la creación de un proyecto estamos creando, finalmente, una aplicación sobre un servidor.

Para poder crear un proyecto, se debe de crear un tag en **git** de tal forma que, a la hora de crearlo, simplemente se tenga que elegir el tag que se desea desplegar. Un tag es una funcionalidad de **git** que permite almacenar una versión congelada del software sin que le afecten nuevas actualizaciones. Una vez realizado esto, la **plataforma Nova** realizará un test

de calidad sobre el código y, si es superado, creará una versión empaquetada de la aplicación sobre uno o varios contenedores **Docker.**

La creación de un proyecto es el punto más sensible de todo el proyecto ya que es dónde se producen más incompatibilidades. Cómo ya se ha comentado anteriormente, el proyecto debe de ser un proyecto **Maven**, por lo que todos los proyectos deben de incluir un pom.xml con todas las dependencias del microservicio correspondiente. Las dependencias del proyecto deben de estar alojadas en un repositorio accesible desde la **Plataforma Nova,** en el caso de nuestro proyecto, se accede a un repositorio global del banco por lo que es necesario asegurarse de que el repositorio contiene todas las dependencias utilizadas y, si no es así, subirlas.

Cuando se crea un proyecto, se eligen las características computacionales que se quieren asignar al mismo. Las características más importantes son las vCPUs (virtual CPUs) y la memoria RAM asignable. Más adelante, en el capítulo de optimización de la aplicación hablaremos de esto en mayor profundidad.

Como también se puede apreciar en la **Ilustración 26**, un subsistema puede estar compuesto de varios productos. Así, los nueve microservicios desarrollados podrían estar también bajo un mismo subsistema. Sin embargo, y dado que los productos desplegados sobre un mismo subsistema deben de pertenecer al mismo repositorio **git**, no tendría sentido hacerlo de esta forma cuando lo que nos interesa es que haya equipos de desarrollo diferentes para cada microservicio.

Conectores Nova

Para desarrollar una aplicación en la **Plataforma Nova** es importante tener en cuenta que, por restricciones de los equipos de seguridad, los contenedores **Docker** están aislados de los sistemas externos a la **Plataforma Nova.** De esta forma, para establecer una conexión directa con un sistema externo a la plataforma es necesario utilizar un **Conector Nova**, que es un **HAProxy** que gestiona este tipo de conexiones.

Los **Conectores Nova** son gestionados por el equipo de la **Plataforma Nova** y son capaces de realizar balanceo de carga en los sistemas que lo soporten.

5.3.2. PATAFORMA SYMPHONY

La **Plataforma Symphony** es uno de los puntos más importantes y complejos del desarrollo de la aplicación. En el grid de esta plataforma se realizan todas las operaciones matemáticas relacionadas con las calibraciones de la volatilidad solicitadas por el usuario, por lo que el proceso tiene que estar optimizado al máximo. Cómo ya se ha comentado anteriormente, el **Calibrator** está desarrollado en **Java** y el **Fol Grid** en **C++**, por lo que el mensaje recibido por el **Fol Grid** debe de estar serializado en una estructura entendible para ambos lenguajes.

De esta forma, el primer paso que debemos de llevar a cabo para integrar la plataforma Symphony con nuestra aplicación es realizar esta serialización de los datos. La solución de serialización adoptada ha sido envíar un **String** de **JSON.** Esta serialización se ha realizado de forma muy simple en **Java**, ya que se ha utilizado un **JSONMapper**, que de forma automática un objeto **Java** a **JSON.** Sin embargo, en el **Fol Grid,** la serialización y la deserialización ha sido un poco más compleja ya que en **C++** no existen librerías que automaticen este proceso. Además, por las características y las necesidades de los servicios ejecutables **Plataforma Symphony** es complicado incorporar librerías de terceros, por lo que la serialización y la deserialización se ha tenido que "mapear" manualmente.

El servicio ejecutado sobre la **Plataforma Symphony** debe de tener una estructura determinada, por lo que lo más fácil es crearlo utilizando las herramientas facilitadas por la propia plataforma, que aporta un asistente de creación de código que nos facilita esta tarea.

Además, el grid de la **Plataforma Symphony** utiliza el sistema operativo **Linux**. El entorno de desarrollo utilizado, en cambio, es **Windows**, por lo que, dado que el servicio está creado en **C++,** que es un lenguaje compilado, es necesario crear un paquete de ejecución distinto para el entorno de desarrollo y el entorno de despliegue**.** Este paquete se comprime en formato **.zip**, y es enviado el equipo de la **Plataforma Symphony** para que sea desplegado en su sistema.

Por otro lado, el microservicio **Calibrator** debe de incorporar algunas dependencias de la **Plataforma Symphony** para poder establecer la conexión **TCP** con la misma. En un principio, se intentó hacer funcionar el proyecto con las dependencias **.jar** indicadas en la documentación de la plataforma, sin embargo, pronto nos dimos cuenta de que era necesario algo más para hacerlo funcionar, instalar el cliente de la **Plataforma Symphony**.

Este último caso debería de ser un paso trivial en la mayoría de los proyectos, solo deberíamos de instalar la nueva dependencia y problema solucionado. Pero en este caso el problema era un poco más grande. Cómo ya se ha explicado con anterioridad, la **Plataforma Nova** despliega automáticamente sobre un **Docker** un proyecto **Maven** contenido en un tag de **git.** No deja editar la configuración del contenedor **Docker** ni instalar dependencias extra, que es lo que se necesitaría para crear un cliente **Symfony.**

De esta forma, fue necesario encontrar otra forma para crear un cliente nova sobre la **Plataforma Nova**. Finalmente, para poder crear el cliente, y dado que la **Plataforma Nova** impide modificar la imagen del contenedor creado por ellos, se empaquetaron **librerías de acceso dinámico (dll)** en un archivo **.jar** para, posteriormente, desempaquetarlas en tiempo de arranque de la aplicación de tal forma que está pudiese acceder a las mismas. Además, también fue necesario "settear" dos variables de entorno necesarias para el correcto funcionamiento de las librerías **dll** incorporadas.

5.4. OPTIMIZACIÓN DEL RENDIMIENTO DE LA APLICACIÓN

La optimización del rendimiento es uno de los puntos críticos a la hora de configurar cualquier aplicación, pero lo es más si cabe en una arquitectura orientada a los microservicios, donde el mal funcionamiento de uno de los microservicios puede influir en el rendimiento del conjunto de la aplicación. De esta forma, la aplicación debe de ser optimizada a conciencia para todos los casos de uso que pueda tener.

Para poder optimizar la aplicación de forma correcta es importante tener claro qué aspectos de la optimización de la aplicación son más importantes. En este caso nos centraremos en los siguientes aspectos:

- **Alta disponibilidad:** La aplicación debe de estar disponible en cualquier momento.
- **Tiempos de ejecución:** La carga computacional de una calibración es muy alta, pero uno de los objetivos de la aplicación es disminuir al máximo estos tiempos de calibración.
- **Ejecución con prioridades:** La calibración de las volatilidades de los subyacentes exigida por la normativa **FRTB** podría durar, en principio varias horas, por lo que si queremos admitir consultas en tiempo real da la volatilidad debemos de admitir prioridades.

5.4.1. PARAMETRIZACIÓN DE LOS TIEMPOS DE EJECUCIÓN DE CADA MICROSERVICIO

El primero de los pasos para poder optimizar el rendimiento de una aplicación es la monitorización de esta para tener información de los tiempos de ejecución y los posibles bloqueos que se puedan estar produciendo en cada momento.

Tenemos varias opciones para poder obtener esta información. La primera de ellas es el análisis de la información almacenada en los logs producidos por cada microservicio. Para que esta opción sea viable es muy importante que el log de cada aplicación esté nivelado y sea consistente, además de evitar “logear” información poco relevante. En los logs de cada microservicio debemos de almacenar información relevante al funcionamiento del microservicio en si mismo, uno de los errores que se podrían cometer a la hora de generar los “logs” es registrar la información relevante a la lógica de la aplicación en su conjunto y no la información relevante a este microservicio.

Es importante tener en cuenta, además, que, dado que uno de los objetivos de optimización es conseguir alta disponibilidad para la aplicación, pueden coexistir varias instancias del mismo microservicio, por lo que es muy importante que los “logs” de la aplicación se

almacenen de forma centralizada. De esta forma, los "logs" no se almacenan en la memoria de cada microservicio, sino que se almacenarían en un ente centralizado facilitado, en este caso, por la **Plataforma Nova**.

Como hemos comentado, en cada microservicio solo se almacenaría la información relevante a su propio funcionamiento. De esta forma, debemos asegurarnos de que la información referente a los tiempos de ejecución del conjunto de la aplicación es almacenada en algún lugar. El encargado de realizar esta tarea, como se ha comentado en apartados anteriores, es el **Task Manager**. Este microservicio recoge los mensajes almacenados en la cola **Audit Queue** por el resto de los microservicios y los almacena en la tabla **Audit** de la base de datos central de la aplicación.

El primer paso para revisar el rendimiento de la aplicación sería revisar la tabla **Audit**, así, podremos entender en qué punto de la ejecución se están produciendo los bloqueos. A continuación, una vez que ya sabemos en qué microservicio estamos encontrando problemas, revisaríamos los "logs" específicos de la aplicación para comprender mejor donde está el problema.

Finalmente, otro punto muy importante es estandarizar las pruebas de ejecución utilizadas de tal forma que siempre se utilicen las mismas. De esta forma, tendremos registrados los tiempos habituales de ejecución y podremos detectar fácilmente si los cambios realizados para optimizar la aplicación están obteniendo resultado. Además, en el momento de la integración de la aplicación con la **Plataforma Symphony** podremos realizar estas mismas pruebas para comprobar si el rendimiento obtenido en esta plataforma es el esperado. Las pruebas de ejecución estándar utilizadas son las siguientes:

- **Prueba de Carga:** Calibración de la volatilidad de 4 subyacentes con una ventana temporal de 10 años.
- **Prueba de prioridades:** Calibración de la volatilidad de 8 subyacentes con una ventana temporal de 1 año. Cada uno de los subyacentes con una prioridad distinta de tal forma que los subyacentes con mayor prioridad deban de acabar primero.

- **Real-Time Calibration:** Calibración de la volatilidad de un subyacente en un día con la máxima prioridad.

5.4.2. BLOQUEOS ENCONTRADOS

Una vez parametrizados los tiempos de ejecución de la aplicación es necesario analizar los resultados para identificar los puntos en los que la aplicación toma más tiempo durante la ejecución. Este análisis será realizado en dos entornos diferentes, en un primer lugar, se utilizará el entorno de desarrollo local de la **Plataforna Nova** y de la **Plataforma Symphony** para encontrar las primeras flaquezas de la aplicación antes de subir la aplicación a los entornos de preproducción de ambas plataformas.

Para realizar este análisis se partió de cero, dando las mismas características a todos los microservicios, intentando no asumir antes de tiempo cual de ellos tendría más carga y de qué tipo sería esta. Así, al realizar este análisis, todos los microservicios disponían de un único hilo de ejecución, por lo que al lanzar la prueba estándar de carga (4 subyacentes con una ventana temporal de 10 años), comprobamos que el tiempo de ejecución del sistema era mayor de cuatro horas.

Este tiempo de ejecución era de esperar, en cualquier caso, dado que para realizar esta prueba el sistema tiene que realizar 4x10x362=14600 calibraciones. Teniendo en cuenta que cada una de ellas tarda un segundo ser calibrada, el tiempo de calibración fue razonable.

Además, también se identificó un bloqueo temporal en el **Orchestrator** que, de igual forma, también era esperado dado que es el microservicio encargado de gestionar el comportamiento de toda la aplicación, por lo que la carga que debe de soportar es muy alta.

Entendiendo, por tanto, que estos bloqueos son provocados por la alta carga del sistema y no por una falta de optimización del mismo, se ha optado por paralelizar estas tareas para multiplicar la velocidad de ejecución de la aplicación. El proceso seguido para realizar esta tarea se explicará más adelante, en este mismo apartado.

Con la paralelización del sistema, el rendimiento de la aplicación mejoró en gran medida, pasando de las 4 horas de ejecución a 7 minutos y medio. Sin embargo, está configuración estaba lejos de ser la óptima, y esto se pudo ver reflejado al subir el proyecto a preproducción, sobre la **Plataforma Nova**.

En este momento, debido a varios motivos, el tiempo de ejecución se dobló, pasando de 7,5 minutos a 15. En primer lugar, los tiempos de obtención de información de la base de datos eran mucho mayores que los registrados en el entorno local debido a un problema de latencia, y a problemas de rendimiento, ajenos al desarrollo, en los **Conectores Nova**.

El tiempo extra debido a la latencia era predecible, teniendo en cuenta que la **Plataforma Nova** y las bases de datos utilizadas se encuentran en distintos edificios, sin embargo, el rendimiento de los conectores era pésimo, por lo que se solicitó a los gestores de la **Plataforma Nova** los revisaran y, efectivamente, encontraron un problema que estaba mermando el rendimiento.

Una vez solucionado el problema de los **Conectores Nova**, el tiempo de ejecución de la aplicación seguía siendo espacialmente superior a los obtenidos en el entorno local, y esto era debido exclusivamente a la latencia existente en la transmisión de información. Para solucionar estos problemas, se decidió empezar a transmitir la información en bloques de N calibraciones en lugar de hacerlo de una en una. De esta forma, se el tiempo perdido por la latencia de la red se disminuyó en un factor de N debido a que este tiempo sólo se perdía en una de cada N calibraciones.

Una vez solucionado el problema con los tiempos de obtención y guardado de información en la de datos, el tiempo final de ejecución bajó hasta los 8 minutos que, aunque era un tiempo considerablemente inferior a los 15 minutos que se habían obtenido en la primera ejecución sobre la **Plataforma Nova**, aún era superior al que se había obtenido en local.

Tabla 8: Optimizaciones de rendimiento

	Plataforma	Acción	Tiempo inicial	Tiempo final
1	Local	Paralelización de tareas	4 horas	7,5 minutos
2	Nova	Cambio de plataforma	7,5 minutos	15 minutos
3	Nova	Mejorar conectores	15 minutos	10 minutos
4	Nova	Transmisión por lotes	9 minutos	8 minutos
5	Nova	Aumentar potencia	7 minutos	6,5 minutos

Llegados a este punto, hubo que profundizar un poco más en los logs de las aplicaciones, ya que el tiempo extra que estábamos obteniendo no se estaba consumiendo en un punto concreto, sino que se repartía entre todos los microservicios. De esta forma, y tras investigar los rendimientos de los microservicios, se llegó a la conclusión de que el problema radicaba en la potencia que cada uno de los microservicios tenía asignada.

La **Plataforma Nova** permite elegir entre las siguientes opciones de configuración para elegir la máquina sobre la que se desplegarán los microservicios:

Tabla 9: Posibles especificaciones de la Plataforna Nova

Configuración	Núcleos virtuales	Memoria RAM
Configuración 1	0.5 vCPU	1 Gb
Configuración 2	1 vCPU	2 Gb
Configuración 3	2 vCPUs	4 Gb
Configuración 4	4 vCPUs	8 Gb

Se decidió utilizar la configuración 1 para todos los microservicios, exceptuando al **Calibrator** y al **Fol Grid**, que fueron desplegados utilizando la configuración 2. Estas

configuraciones se antojaban suficientes teniendo en cuenta que se desplegaron al menos dos instancias de cada microservicio.

Sin embargo, como ya se ha comentado, los tiempos de calibración obtenidos sobre la **Plataforma Nova**, ampliamente superiores a los obtenidos realizando la ejecución en local, pusieron de manifiesto que estas configuraciones no eran suficientes. El microservicio **Market Data**, por ejemplo, tenía suficiente con la potencia de la configuración 1 pero la memoria de esta configuración era insuficiente dado que guarda información en caché para evitar realizar consultas repetidas a la base de datos.

Finalmente, se aumentó la potencia de todos los microservicios a la configuración 2, y los resultados obtenidos fueron muy satisfactorios, obteniendo un tiempo de calibración de 6 minutos y medio en la prueba estándar de carga realizada. Este tiempo es prácticamente el mínimo que podemos obtener ya que esta ejecución realiza 14600 calibraciones, por lo que, teniendo en cuenta que cada una de ellas tarda un segundo en ser realizada y que el **Fol Grid** tiene únicamente 40 hilos de ejecución, el tiempo mínimo de calibración es de 6,03 minutos.

5.4.3. PARALELIZACIÓN DE TAREAS

Uno de los objetivos de la optimización de la aplicación es conseguir alta disponibilidad, para ello, es importante que exista redundancia en el sistema. Con este objetivo, todos los microservicios tendrán, al menos, dos instancias de cada microservicio. Para conseguir una disponibilidad aun mayor, las distintas instancias de cada microservicio deberían de estar en edificios o plataformas distintas, pero se considera que la disponibilidad alcanzada con la redundancia de microservicios sobre la **Plataforma Nova** ya es suficientemente grande como para cubrir con los objetivos iniciales.

La paralelización de tareas no se refiere únicamente a la redundancia horizontal de los servidores de aplicaciones, sino que una parte muy importante es la creación de Pools de hilos de ejecución internos en cada uno de los microservicios. Los pools de hilos tienen como objetivo mejorar el rendimiento de la aplicación y el número de hilos creados en cada microservicio debe de ser consistente con el número de hilos de los microservicios

adyacentes. Por ejemplo, no tendría sentido que el **Calibrator** tuviera un pool de 200 hilos cuando el **Fol Grid** está limitado en producción a 40 hilos

Todos los microservicios tendrán hilos concurrentes, pero son el **Calibrator** y el **Orchestrator** quienes deben de tener más cantidad de hilos debido a que su objetivo es el de realizar un gran número de operaciones simultáneas y estas no son especialmente complejas.

5.4.4. TRANSMISIÓN DE INFORMACIÓN POR LOTES

Otra de las medidas adoptadas para mejorar el rendimiento de la aplicación fue la transmisión de información por lotes. En un principio, la información de cada una de las calibraciones generadas por el sistema (Calibración de la volatilidad de un subyacente en un día concreto) era transmitida por el sistema de una en una.

En la primera prueba realizada en local, el rendimiento de la aplicación no se vio afectado, ya que la base de datos de la aplicación estaba replicada en local y, por tanto, la latencia en la transmisión era mínima. Sin embargo, al subir la aplicación al entorno de preproducción de la **Plataforma Nova** la latencia de la conexión con la base de datos aumentó considerablemente debido a que esta se encontraba en un sistema externo a la plataforma.

Para mitigar el problema generado por la latencia de la red, se decidió agrupar estas calibraciones en lotes de calibraciones de tal forma que el sistema solo tenga que esperar la latencia de una de las transmisiones. De esta forma, sumado a la presencia de varias instancias de los microservicios trabajando en paralelo, se consiguió mitigar el problema generado por la latencia de la red que, aunque no se llegó a solucionar, el rendimiento del sistema no se vio afectado por esta.

5.4.5. UTILIZACIÓN DE PRIORIDADES

El caso de uso principal de la aplicación es el **Calibrador Batch**, que es el que se utilizaría para cumplir con la normativa **FRTB**. Para cumplir con esta normativa, la aplicación debe de ser capaz de realizar la calibración de 400 subyacentes con una ventana temporal de 10

años que, con la configuración actual, tardaría 10 horas. Obviamente, la aplicación es escalable y bastaría con poner más instancias de cada microservicio para disminuir este tiempo de calibración a unos minutos.

En cualquier caso, para poder más casos de uso que el citado **Calibrador Batch**, como las calibraciones en tiempo real, es necesario que la aplicación admita prioridades de tal forma que esta sea capaz de anteponer las calibraciones destinadas a ser consultadas en tiempo real a las calibraciones destinadas al caso de uso **Calibrador Batch**.

Para gestionar la prioridad de los mensajes se asume que en ningún caso una calibración entrante se debe de anteponer a aquellas tareas que un microservicio esté realizando en el momento de su llegada. De esta forma, el primer punto donde se deben de gestionar las prioridades es en las colas de mensajes **Tibco**, que admiten prioridades del 1 al 9.

Así, todas las peticiones de calibración serán depositadas en la cola **Task Queue** por el **Task Manager** de tal forma que el **Orchestrator**, cuando libere uno de los hilos de ejecución, recogerá la tarea de calibración con mayor prioridad de todas las depositadas en esta cola. Una vez finalizada su tarea, este microservicio depositará las calibraciones solicitadas en la cola **Calib request Queue**, con la misma prioridad del mensaje recibido.

De igual forma trabajarán el **Calibrator** y el **Fol Grid**, de tal forma que una calibración en tiempo real siempre será recogida de las colas **Tibco** antes que las calibraciones pendientes de cualquier calibración batch.

Finalmente, el segundo punto donde hay que gestionar prioridades es en los receptores de mensajes **JMS**, que son los encargados de escuchar las colas **Tibco** y de recoger los mensajes almacenados en estas de tal forma que cuando el microservicio esté listo pueda recoger los mensajes almacenados en ellas. Es importante gestionar las prioridades en este componente ya que, dado que el propio sistema receptor de **JMS** genera una cola interna de mensajes que por defecto no admite prioridades, por lo que es preciso modificarla para que lo las admita, o, generar un receptor de mensajes **JMS** encargado de recibir los mensajes con prioridad

inferior a 8 y otro para recoger los mensajes con prioridad 9 (Prioridad utilizada para el caso de uso **Real-Time**).

Esta última solución fue la adoptada finalmente en el proyecto obteniendo unos resultados satisfactorios dado que el tiempo de calibración de una calibración en tiempo real apenas se ve afectado cuando el sistema está procesando una calibración Batch.

Capítulo 6: ANÁLISIS DE RESULTADOS

En este capítulo se realizará un análisis crítico del proyecto y se compararán los resultados obtenidos con las expectativas generadas al principio de este. Además, se analizarán los puntos fuertes y débiles del proyecto desarrollado, haciendo hincapié en los beneficios e inconvenientes que ha generado la utilización de una arquitectura orientada a los microservicios.

6.1. PUNTOS FUERTES DEL PROYECTO

En primer lugar, se explicará en este apartado los resultados más positivos del proyecto, analizando las ventajas que ha aportado realmente la utilización de una arquitectura orientada a los microservicios.

6.1.1. RENDIMIENTO DE LA APLICACIÓN

Cómo ya se ha comentado en el capítulo **5.4 Optimización del rendimiento de la aplicación**, la aplicación desarrollada ha conseguido cumplir los objetivos iniciales de rendimiento, y lo ha hecho, además, con resultados muy positivos.

Se puede decir que la aplicación a cumplido con todos los objetivos de rendimiento propuestos al principio del proyecto. En los siguientes apartados se muestran los resultados obtenidos en cada uno de los aspectos relevantes de rendimiento.

6.1.1.1. Alta disponibilidad

Es necesario que la aplicación desarrollada tenga una disponibilidad muy cercana al 100%.

La **Plataforma Nova** indica que la disponibilidad de un microservicio desplegado sobre su plataforma es del 99%, mientras que en la disponibilidad de la **Plataforma Symphony** es del 99.9%. Supongamos que sobre la **Plataforma Nova** sólo se dispone de una instancia por microservicio y que los ocho microservicios desplegados sobre esta plataforma son

necesarios para el correcto funcionamiento del caso de uso **Calibrador Batch**. En este caso, la disponibilidad de la aplicación para el caso de uso sería:

$$\boldsymbol{disponibilidad = 0.99^8 \times 0.999 = 0.9218 = 92.18\%}$$

Cómo vemos, la disponibilidad de la aplicación para el caso de uso **Calibrador Batch** estaría más cercana al 90% que al 100%. Estos resultados, aunque no son especialmente malos, tampoco son los deseados para la aplicación, por lo que se decidió utilizar al menos dos instancias por cada microservicio de tal forma que la disponibilidad de la aplicación sería la siguiente:

$$\boldsymbol{disponibilidad = (1 - (1 - 0.99)^2)^8 \times 0.999 = 0.9982 = 99.82\%}$$

Cómo se puede ver, la disponibilidad de la aplicación sería del 99.82%, es decir, la aplicación fallaría una vez cada año y medio. Aunque ya es un resultado muy positivo, este se podría mejorar añadiendo más instancias de cada microservicio en paralelo.

6.1.1.2. Tiempos de ejecución

El segundo de los objetivos iniciales de rendimiento era rebajar los tiempos de ejecución de la aplicación. Como ya se ha explicado anteriormente, la aplicación será muy exigida cuando sea lance una ejecución batch para cumplir con la normativa **FRTB**, por lo que el rendimiento de la aplicación debe de estar optimizado al máximo para conseguir rebajar el tiempo total de ejecución al máximo.

Una vez optimizado el rendimiento de la aplicación, esta consiguió ejecutar la prueba estándar de carga en 6 minutos y medio. La prueba estándar de carga consiste en realizar la calibración de la volatilidad de 4 activos subyacentes para un periodo temporal de 10 años. Teniendo en cuenta que el grid de la **Plataforma Symphony** tiene únicamente 40 hilos de ejecución y que el tiempo estimado de calibración de la volatilidad de un subyacente para un día es de un segundo, el tiempo mínimo de ejecución de la tarea sería de 6 minutos, por lo que el tiempo de 6 minutos y medio obtenido por la aplicación es muy positivo.

Si la aplicación es capaz de enviar tareas de calibración al grid de la **Plataforma Symphony** a una velocidad suficiente como para que este nunca deje de trabajar, el tiempo de calibración obtenido por la aplicación será inversamente proporcional al número de hilos de ejecución del grid. Una de las mayores ventajas de la arquitectura utilizada es que puede escalar hasta el infinito, por lo que los tiempos de ejecución se podrían disminuir infinitamente. De esta forma, es muy importante realizar una reflexión sobre el impacto que tiene el tiempo de calibración sobre la entidad y sobre los trabajadores para poder encontrar un balance entre el número de recursos dedicados a la aplicación y el tiempo de ejecución medio de la misma.

6.1.1.3. Ejecución con prioridades

Cómo se ha explicado en el apartado anterior, los tiempos de ejecución de la aplicación pueden ser muy grandes, por lo que, si se quieren compatibilizar distintos casos de uso sobre la misma arquitectura, se debe asignar una prioridad de ejecución a cada uno de los casos de uso de tal forma que si la plataforma se encuentra con varias ejecuciones simultaneas sea capaz de identificar cuál es la más importante.

El sistema de prioridades se desarrolló de forma satisfactoria sobre el sistema, y la prueba estándar de prioridades desarrollada, donde se envían ocho tareas de calibración con distintas prioridades, muestra que el sistema prioriza correctamente las tareas de calibración y que la tarea con mayor prioridad apenas se desvía de los tiempos normales de calibración.

6.1.2. FLEXIBILIDAD DE LA APLICACIÓN

En el capítulo **3.3** se comentan los beneficios de realizar una aplicación con una arquitectura orientada a los microservicios entre ellos se encontraba la flexibilidad que aporta la desagregación de funcionalidades dentro de la aplicación.

Haber utilizado una arquitectura de microservicios para la realización del proyecto, aporta a la aplicación gran flexibilidad ya que permite añadir requerimientos específicos a cada microservicio sin que estos afecten en ningún aspecto al resto de la aplicación.

Durante el periodo de diseño de los microservicios se especifica qué acciones debe de realizar cada microservicio y la forma de comunicarse con el resto del sistema. Estos dos aspectos son los únicos que deben de cumplir los microservicios, y es indiferente la forma en la que lo hagan siempre y cuando cumplan ambos requisitos. De hecho, un microservicio podría en un momento dado delegar su funcionalidad a otros sistemas siempre y cuando siga haciendo de enlace entre el microservicio solicitante y el microservicio ejecutor.

Precisamente ha sido este principio de las arquitecturas orientadas a los microservicios el que ha permitido disminuir el tiempo de ejecución de la aplicación al máximo, ya que se decidió utilizar un grid de la **Plataforma Symphony** para realizar las operaciones de calibración de la volatilidad y, gracias a eso, reducir el tiempo de ejecución de estas.

6.1.3. ESCALABILIDAD DE LA ARQUITECTURA

Esta fortaleza de la aplicación está relacionada en gran medida con el apartado anterior ya que gracias a la flexibilidad que aportan las arquitecturas orientadas a los microservicios la aplicación es capaz de escalar de una forma eficiente.

Hoy en día, uno de los puntos más importantes en el desarrollo del software es la escalabilidad de las aplicaciones desarrolladas. Por norma general hay dos formas de escalar una aplicación, que comúnmente son llamadas escalado vertical y escalado horizontal.

El escalado vertical consiste en aumentar la capacidad de los servidores de la aplicación mientras que el escalado horizontal consiste en desplegar tantos servidores en paralelo como se necesiten. Por lo general, el escalado horizontal es más barato y beneficioso, ya que aparte de aumentar el rendimiento de la aplicación también aumentan su disponibilidad.

Sin embargo, el escalado horizontal exige una complejidad extra a la aplicación, ya que necesita de un dispositivo centralizado que se encargue de repartir la carga uniformemente entre todas las instancias desplegadas de la aplicación, un balanceador de carga. Esta complejidad que se comenta es asumida desde el primer momento en una arquitectura de microservicios y todas las plataformas orientadas a los microservicios incluyen una herramienta centralizada encargada de realizar la tarea del balanceador de carga. Así, todos

los microservicios serán capaces de escalar horizontalmente, como demuestra el hecho de que la aplicación desarrollada utiliza, al menos, dos instancias de cada uno de los microservicios. La herramienta encargada de realizar el balanceo de carga en la **Plataforma Nova** es **Netflix Eureka.**

Además, la flexibilidad aportada por las arquitecturas de microservicios permite a la aplicación escalar de una forma mucho más eficiente, ya que no existe la necesidad de escalar la aplicación en su conjunto, sino que solo se deben de escalar las funcionalidades que realmente tengan esta necesidad, es decir, los microservicios.

6.1.4. FACILIDAD PARA AÑADIR NUEVAS FUNCIONALIDADES

Igualmente, gracias a la arquitectura de microservicios, el proyecto desarrollado tiene la gran fortaleza de que, en el momento en el que se quiera añadir una nueva funcionalidad no es necesario tocar ninguno de los microservicios que la componen. Tal y como está diseñada solo habría que añadir un nuevo "módulo" a la aplicación que se aproveche de las actuales funcionalidades de la misma para realizar alguna otra opción.

Este nuevo "módulo" sería otro microservicio y para poder aprovechar toda la potencia de la aplicación puede que también hubiera que modificar el **Orchestrator**, pero en ningún caso habría que modificar ningún otro microservicio. En caso de tener que hacerlo significaría que la aplicación no ha sido diseñada correctamente.

6.2. PUNTOS DÉBILES DEL PROYECTO

En el anterior apartado se habla de las fortalezas del proyecto, y coinciden con bastante exactitud con los beneficios de las arquitecturas orientadas a los microservicios que se exponían en el capítulo **3.3**. De la misma forma, en el capítulo **3.4**, se exponían los posibles inconvenientes de desarrollar una aplicación con una arquitectura de microservicios. Durante el desarrollo de la aplicación, el equipo de desarrollo no solo se encontró con los problemas que se exponían en dicho capítulo, sino que también se encontró con otros problemas que no habían sido previsto. En cualquier caso, quedó patente la importancia de desarrollar este tipo

de proyectos sobre una plataforma especializada orientada a los microservicios, ya que, sin su ayuda, la complejidad del desarrollo crecería en gran medida.

6.2.1. COMPLEJIDAD DE GESTIÓN

Una de las primeras complejidades que la aplicación ha heredado de las arquitecturas de microservicios es la gestión de la aplicación. Aunque existen herramientas para centralizar la gestión de todos los microservicios bajo un mismo software, la aplicación está compuesta por nueve microservicios que, al fin y al cabo, no dejan de ser nueve aplicaciones totalmente independientes.

Lo que se puede ver como una ventaja, ya que aporta una gran flexibilidad a la aplicación, también es un quebradero de cabeza a la hora de monitorizar el correcto funcionamiento de la aplicación. El primer punto donde se ha visto esta complejidad es en la optimización del sistema para cumplir con los requisitos iniciales, donde, para detectar los puntos de bloqueo de la ejecución fue necesario crear logs con distintos niveles de abstracción, nivel de sistema, de aplicación o de microservicio.

Así mismo, otro punto donde se encontraron grandes problemas fue a la hora de crear una solución para aportar trazabilidad a la información de las calibraciones. Finalmente, se creó una solución completamente externa a la **Plataforma Nova**, donde se almacenaba a información referente a estos sistemas en una base de datos **PostgreSQL**.

6.2.2. AUTOMATIZACIÓN DE LA GESTIÓN DE ERRORES

Continuando con el punto anterior, la gestión de errores es uno de los puntos de mayor complejidad en una arquitectura de microservicios, sobre todo si se quiere gestionar de forma automática.

La automatización de las gestiones es uno de los principios básicos cuando se desarrolla una aplicación con una arquitectura de microservicios. En el momento en el que, para la correcta ejecución de un caso de uso de una aplicación, en este caso el **Calibrador Batch**, se necesita

la participación de nueve microservicios, es muy complicado monitorizar y asegurar que todos ellos han realizado su función correctamente.

Además, revisarlo de forma manual se antoja literalmente imposible debido a las grandes cantidades de información almacenada y a la complejidad existente para comprobar la traza de ejecución del sistema. De esta forma, es de vital importancia que todos los sistemas tengan su propio sistema para comprobar que su comportamiento y el comportamiento de sus vecinos es el correcto y poder actuar en consecuencia.

Dado que el sistema desarrollado es, hasta el momento, un **MVP** (Minimum Viable Product), la aplicación no tiene ningún sistema encargado de realizar este tipo de comprobaciones y realmente se asume que las ejecuciones realizadas se realizan sin ningún tipo de fallo.

6.2.2.1. Recuperación ante fallos

Profundizando un poco más en el problema, la aplicación tampoco está dotada de un sistema de recuperación ante fallos de tal forma que, si el conjunto de la aplicación falla durante la ejecución de una tarea de calibración, la única forma de obtener el resultado de esta terea es volviendo a lanzar la tarea de calibración de la volatilidad.

En cualquier caso, este punto es una de las próximas actualizaciones que se realizarán sobre la aplicación actual ya que la información y el estado de la calibración se guarda y se actualiza en cada una de las etapas de ejecución de la aplicación. Además, también se almacena toda la información obtenida durante el proceso de calibración, por lo que el único sistema necesario para reanudar la ejecución desde el punto en que se encontraba en el momento del fallo es un sistema que se encargue de revisar el estado de las calibraciones almacenadas en base de datos e introduzca las calibraciones pendientes en la cola de mensajes correspondiente al punto de ejecución en el que se encontraba.

6.2.3. ESCASO APROVECHAMIENTO DEL HARDWARE

Hoy en día se pueden encontrar muchos ejemplos de aplicaciones que permiten el escalado horizontal en tiempo real. Plataformas como **AWS** (Amazon Web Services) incluyen este

servicio de forma gratuita en sus sistemas y cobran únicamente por el número de horas de computación utilizadas.

Este tipo de escalado aporta gran flexibilidad a la aplicación y, sobre todo, permite a las aplicaciones orientadas a los microservicios optimizar al máximo el hardware utilizado, ya que cada uno de los microservicios es capaz de escalar en tiempo real en función de la demanda de un instante temporal concreto.

Lamentablemente, la **Plataforma Nova** no incluye la posibilidad de utilizar escalado en tiempo real. Esto significa que cada uno de los microservicios debe de tener en todo momento tantas instancias desplegadas como le sean necesarias para cubrir el pico de ejecución del sistema.

Por tanto, la gran mayoría del tiempo, los microservicios tendrán instancias inútiles sin realizar ningún tipo de trabajo. Además, por las características de la aplicación desarrollada, durante todo el flujo de ejecución de una tarea de calibración hay microservicios al 100% de su capacidad mientras que hay otros que no están realizando ninguna tarea y que, por tanto, están desaprovechando la capacidad computacional de sus instancias.

De esta forma, en la gran mayoría de los casos, una aplicación desarrollada sobre una arquitectura orientada a los microservicios que no es capaz de escalar horizontalmente en tiempo real es menos eficiente en cuanto a la utilización del hardware que una aplicación monolítica desarrollada sobre un único servidor.

6.3. INTERFAZ WEB

Por último, se ha desarrollado un interfaz web, que ya no puede ser considerado un microservicio porque aúna varias funcionalidades dentro de una misma aplicación. Sin embargo, esta interfaz es capaz de comunicarse con los distintos microservicios de la aplicación y de realizar distintas acciones, como lanzar nuevas tareas de calibración, con distintas prioridades.

En esta primera versión, la interfaz muestra el menú que podría ver un usuario de la aplicación. Sin embargo, en futuras iteraciones se incorporarán funcionalidades de

administrador de tal forma que estos puedan actuar directamente sobre el sistema, ejecutando acciones como el purgado de colas o la modificación de los ajustes de calibración de los distintos subyacentes.

A continuación, se mostrarán algunas capturas de la aplicación, y en el "**Anexo A: Manual de Usuario**", se detallará el funcionamiento de la aplicación.

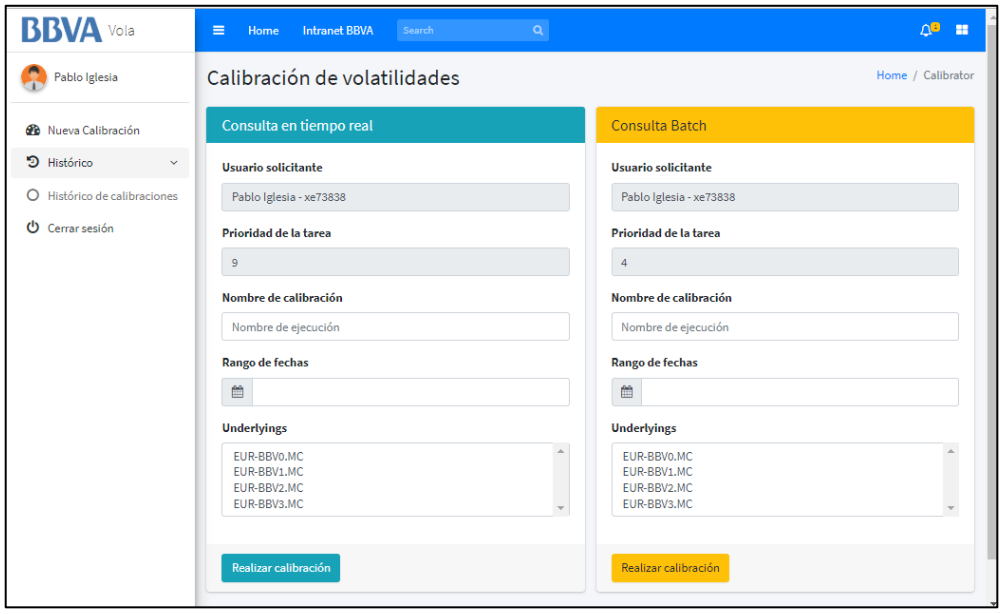


Ilustración 27: Lanzamiento de tareas de calibración desde la aplicación web

Ilustración 28: Visualización de calibraciones realizadas

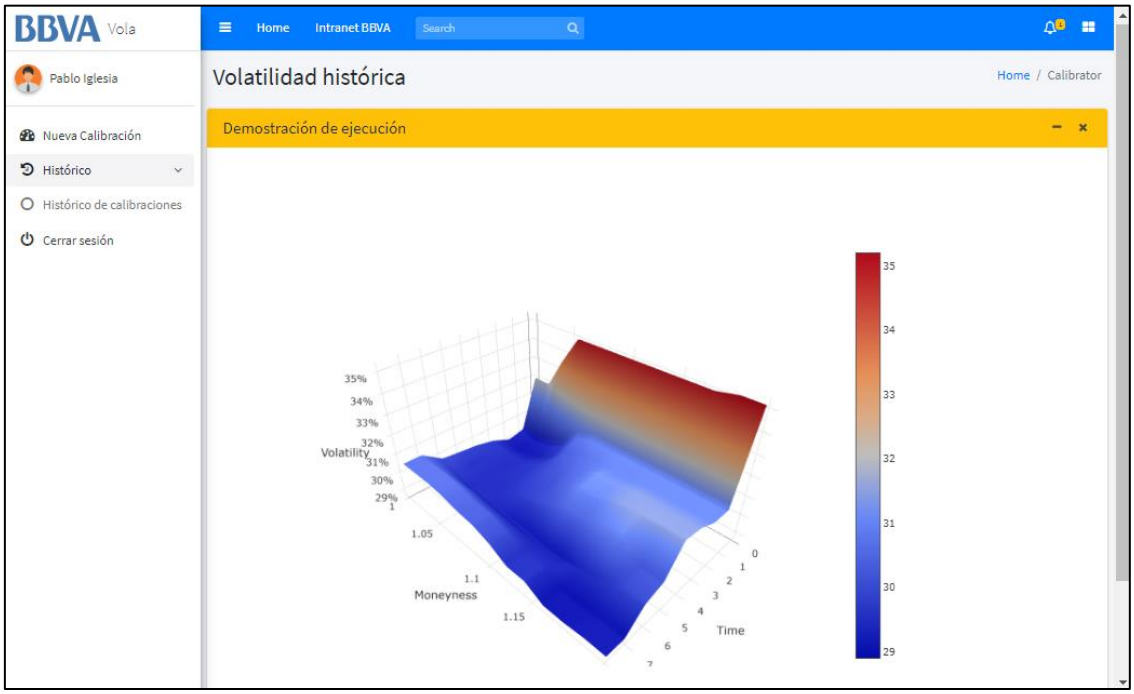


Ilustración 29: Resultados de la calibración

Capítulo 7: CONCLUSIONES Y FUTUROS TRABAJOS

En este capítulo se recuperarán los objetivos iniciales del proyecto y se analizará si han sido cumplidos o no. Además, se expondrán los trabajos futuros que no se han podido realizar durante el desarrollo de este trabajo.

Los objetivos planteados al comienzo del proyecto fueron los siguientes:

- Estudiar y analizar una arquitectura orientada a los microservicios.
- Comprobar la viabilidad de la utilización de microservicios dentro de la industria financiera y su importancia en la trasformación digital.
- Realizar un caso de uso real con microservicios en la industria financiera.

Para argumentar el cumplimiento de los dos primeros objetivos planteados se realizará a continuación un análisis sobre la utilización de los microservicios para la creación de nuevas aplicaciones. Por otro lado, el tercero de los objetivos se ha cumplido y su desarrollo ha sido explicado durante los capítulos 4, 5 y 6.

7.1. ANÁLISIS SOBRE LA UTILIZACIÓN DE MICROSERVICIOS

Durante el transcurso de las prácticas en el banco **BBVA** y durante la realización de este trabajo he realizado una aproximación teórica y práctica a las arquitecturas orientadas a los microservicios. Este tipo de arquitecturas se está convirtiendo, cada vez más, en una tendencia a la hora de desarrollar aplicaciones en cualquier ámbito y, cómo toda nueva tecnología o metodología, se pueden encontrar en internet numerosos artículos enumerando las ventajas y los inconvenientes de esta, pero no es hasta que la pruebas cuando realmente te das cuenta de los pros y los contras de utilizar esa tecnología, en este caso, las arquitecturas orientadas a los microservicios. En este análisis se pretende exponer las conclusiones

subjetivas obtenidas después de haber realizado una aplicación con una arquitectura orientada a los microservicios.

La primera aproximación que realicé a las arquitecturas orientadas a los microservicios fue meramente teórica y, en ella, me daba la sensación de que este tipo de arquitecturas era el presente, que aportan muchas más ventajas que inconvenientes y que los inconvenientes introducidos eran de una importancia menor o fácilmente solucionables. Ahora, después de haber desarrollado un caso de uso real, sigo pensando que es el presente, pero sólo para algunas aplicaciones.

La teoría de las arquitecturas de microservicios dicta que cada microservicio debe de ser desarrollado como una aplicación en sí misma que cumpla con una función determinada y atómica de la aplicación a desarrollar. De esta forma, el primer paso para desarrollar una arquitectura de microservicios es desagregar la funcionalidad de la aplicación en bloques funcionales totalmente independientes entre sí, pero… ¿Es posible desagregar la funcionalidad de todas las aplicaciones?

Tras mucho reflexionar, la conclusión a la que he llegado es que cualquier aplicación desarrollada se puede desagregar en distintos bloques funcionales y, por tanto, desarrollar con una arquitectura de microservicios. Si consideramos una aplicación como un "Programa o conjunto de programas informáticos que realizan un trabajo específico, diseñado para el beneficio del usuario final", se puede determinar que toda aplicación se puede dividir, por lo menos, en dos bloques funcionales, un bloque encargado de la interacción con el usuario, y otro encargado de realizar el trabajo específico solicitado por este. Sin embargo, mi opinión, hoy por hoy, es que para la gran mayoría de las aplicaciones desarrolladas no tendría sentido utilizar arquitecturas orientadas a microservicios.

En la actualidad, este tipo de arquitecturas aportan muchas ventajas, como una gran flexibilidad en el desarrollo, facilidad de escalado u optimización de los recursos de la aplicación, pero añaden una complejidad extra a los proyectos que, en la gran mayoría de los pequeños proyectos generaría un impacto mucho más grande que las ventajas que realmente genera. Entre los impactos negativos que incorporan este tipo de arquitectura se

incluye una mayor complejidad de gestión y de desarrollo, así como un mayor coste debido a la necesidad de incorporar más de un servidor y a la gestión del Hardware que hacen muchas organizaciones tradicionales.

De esta forma, en mi opinión, las arquitecturas de microservicios deberían de utilizarse, principalmente, en grandes desarrollos donde realmente se puedan aprovechar los beneficios de los microservicios como son la optimización del rendimiento de las aplicaciones, la facilidad de desarrollo para equipos multidisciplinares o la facilidad de añadir nuevas funcionalidades o actualizaciones cuando la aplicación ya está en funcionamiento.

En cualquier caso, creo que el razonamiento planteado será válido únicamente en la actualidad, ya que creo firmemente que en un futuro, más cercano que lejano, las plataformas orientadas a los microservicios conseguirán mitigar, en gran medida, los problemas que actualmente introducen y que se crearán **PaaS** (Platform as a Service) que no solo faciliten este tipo de aplicaciones sino que realmente fomenten el uso compartido de microservicios de tal forma que el desarrollo de aplicaciones se realice realmente desarrollando un microservicio que aúne las funcionalidades de otros microservicios ya creados.

En conclusión, después de haber estudiado la teoría y la práctica de los microservicios, considero que las plataformas existentes actualmente no están preparadas para el desarrollo de cualquier tipo de aplicación pero que, en un futuro cercano, las arquitecturas de microservicios será la solución que utilice la gran mayoría de las aplicaciones.

7.2. FUTUROS TRABAJOS

Finalmente, se especificarán en este apartado los futuros trabajos a desarrollar para completar la aplicación ya desarrollada.

En primer lugar, queda pendiente la realización del caso de uso **Scheduled Calibration**, en el que las calibraciones se pueden dejar programadas para que se ejecuten en la fecha y la hora que se deseen. Este caso de uso es especialmente útil para poder dejar programadas las ejecuciones del caso de uso **Calibrador Batch**, exigido por la normativa **FRTB** de tal forma que el responsable no solo se asegure de que esta calibración se realice periódicamente, en

los intervalos especificados por la normativa, sino que también pueda programar la ejecución para una hora de madrugada de tal forma que el resto de calibraciones no se vea afectada por el lanzamiento de esta. El desarrollo de este caso de uso sería muy simple y consistiría en la creación de un microservicio que almacenase las calibraciones planificadas, y, en el momento indicado por el usuario, realice una petición **HTTP** al **Task Manager** para lanzar la ejecución.

Por otro lado, como ya se ha comentado en el capítulo **6.2.2** la aplicación no cuenta con un control global de errores ni de un sistema de recuperación ante fallos.

Ambas funcionalidades son funcionalidades totalmente fundamentales para poder lanzar la aplicación a producción, y este sería el próximo paso a seguir una vez comprobado que el **MVP** de la aplicación funciona correctamente y cumple con sus objetivos de optimización.

Cómo también se ha comentado en el capítulo **6.2.2.1**, la implementación de un sistema de recuperación ante fallos sería bastante sencilla debido a que la aplicación, actualmente, almacena en base de datos la traza completa de la ejecución de la aplicación, por lo que simplemente se necesitaría que el microservicio **Admin** incorpore la funcionalidad de, en caso de fallo de la aplicación, revisar la información de las ejecuciones almacenada en base de datos e introducir las calibraciones pendientes en la cola de mensajes correspondiente.

Finalmente, la implementación del control de errores sería un poco más compleja porque se necesitaría añadir pequeñas funcionalidades de control en todos los microservicios desarrollados. Además, se debería de especificar las acciones a realizar en caso de error ya que, en el caso de uso actual, como se puede ver en el diagrama de flujo de la **Ilustración 21**, en caso de error, la aplicación actual simplemente cancelaría la ejecución

ANEXO A: MANUAL DE USUARIO

Este manual está destinado a los usuarios que deseen aprender a utilizar el **Calibrador de volatilidades** del banco **BBVA**. Para poder acceder a esta herramienta es necesario realizar una petición interna para dar de alta un usuario con los permisos correspondientes.

La pestaña de inicio de sesión es la que se muestra en la **Ilustración 30**, y el usuario estará creado con el usuario y contraseña del directorio activo del banco.

Ilustración 30: Login de la aplicación

Una vez que se ha accedido correctamente a la aplicación, el usuario podrá ver el dashboard de inicio, donde se muestra directamente la opción de lanzar tareas de calibración sobre el sistema desarrollado. Cómo se muestra en la **Ilustración 31** y en la **Ilustración 32**, dependiendo de los permisos asignados al usuario, este podrá lanzar uno o dos tipos de ejecuciones. La **consulta en tiempo real** está disponible para todos los usuarios ya que es una consulta ligera, de un único subyacente y, como mucho con una ventana temporal de un mes. Por otro lado, la **Consulta Batch** está restringida a algunos usuarios ya que está destinada a realizar calibraciones muy pesadas que podrían colapsar el sistema. Cómo se

puede apreciar en las ilustraciones, la **consulta en tiempo real** tiene una prioridad más alta que la C**onsulta Batch** con el objetivo de mejorar la experiencia de usuario realizando estas calibraciones lo más rápido posible.

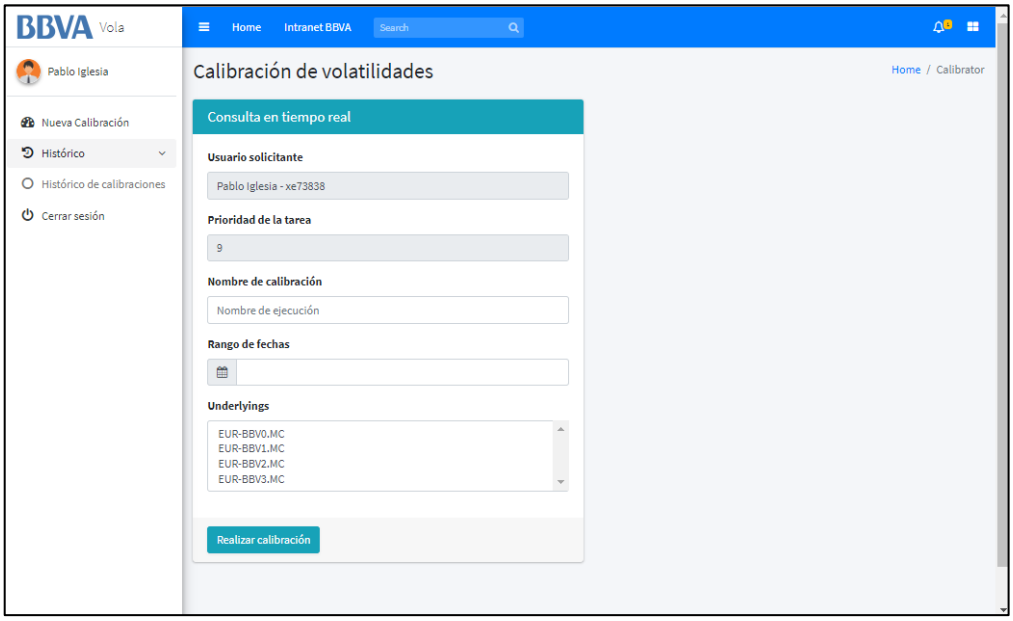


Ilustración 31: Lanzamiento de tareas de calibración desde la aplicación web

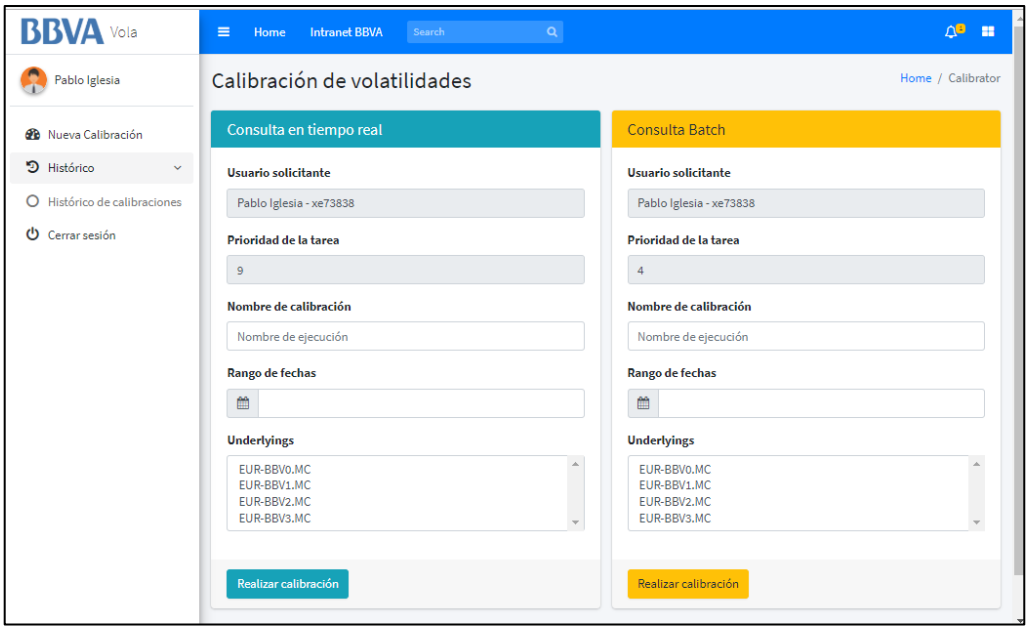


Ilustración 32: Lanzamiento de tareas de calibración desde la aplicación web (Con privilegios)

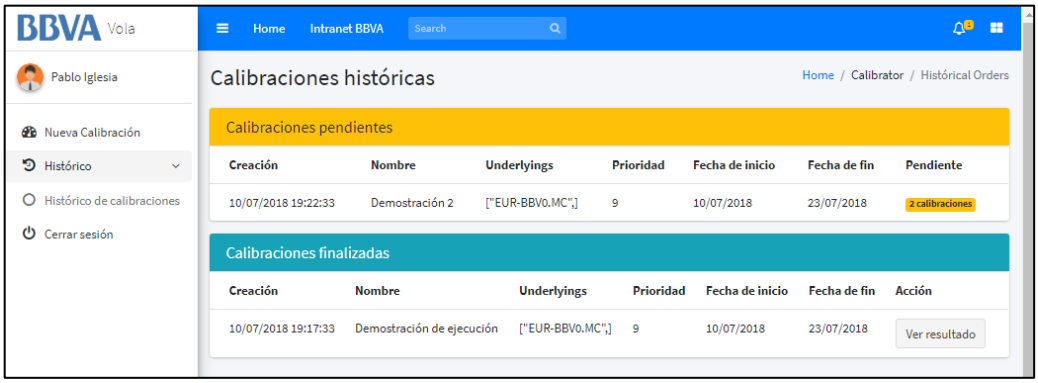


Ilustración 33: Tarea de calibración pendiente

Una vez enviada la tarea de calibración, el usuario será redirigido automáticamente a la pestaña de histórico de calibraciones. En esta pestaña se muestra todo el histórico de las calibraciones del usuario y desde ella se podrá acceder a los resultados de calibración de todos ellas. Durante el periodo de calibración de la tarea solicitada, sobre el histórico de las tareas de calibración realizadas se mostrará una tabla independiente con las tareas de calibración pendientes. Esta tabla podría ser como la que se muestra en la **Ilustración 33** o en la **Ilustración 34**. A la derecha de la tabla se muestra un contador, donde se muestran las tareas de calibración pendientes. Este contador se actualiza sin necesidad de actualizar la página y en el momento en el que llega a cero, como se muestra en la **Ilustración 34**, se sustituye por un botón desde el que se podrá ver el resultado de la calibración.

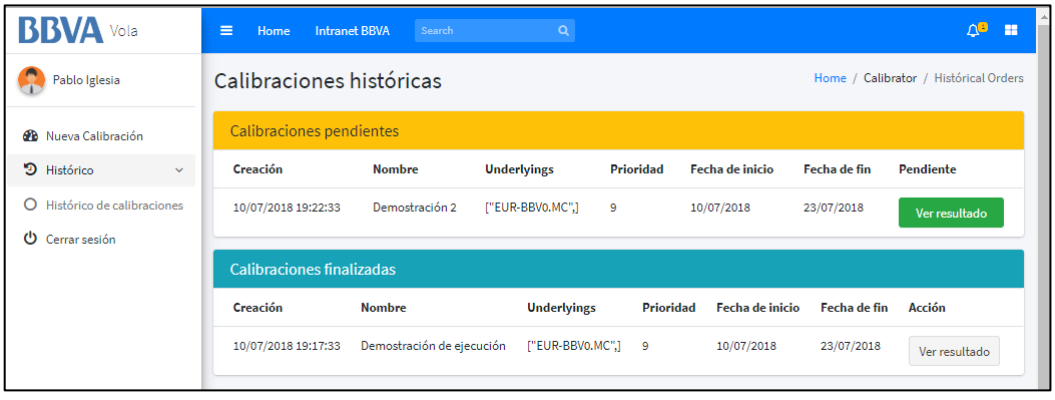


Ilustración 34: Tarea de calibración finalizada

Finalmente los resultados de calibración muestrán la superficie de volatilidad calculada como se puede ver en la **Ilustración 35**.

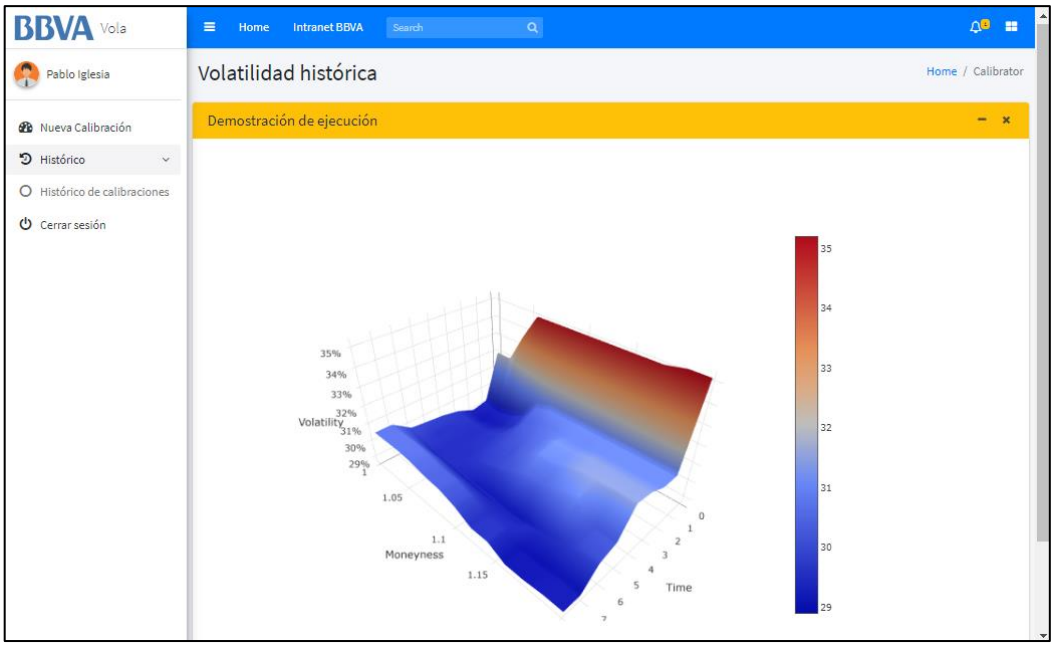


Ilustración 35: Resultados de la calibración

ANEXO B: GUÍA DE INSTALACIÓN DEL PROYECTO

En este Anexo se detallan los pasos que debe de seguir un desarrollador para continuar con instalarse todas las dependencias necesarias para continuar con el desarrollo de la aplicación.

REQUERIMIENTOS DE LA APLICACIÓN

- **Apache Maven**
- **Source Tree** o un gestor de **git** similar.
- Entorno local de la **Plataforma Nova**
- Configuración **Plataforma Symphony**

CLONAR MICROSERVICIOS

El primer paso es preparar el proyecto clonando la última versión del código de cada uno de los microservicios. Para hacerlo es necesario estar conectado a una red interna del banco **BBVA** ya que el proyecto se encuentra en un repositorio interno.

Los repositorios que se deben de clonar son los siguientes:

- **Task Manager:** http://lwnov602.igrupobbva:35000/groups/XEQY/launcher
- **Orchestrator:** http://lwnov602.igrupobbva:35000/groups/XEQY/orchestrator
- **Settings:** http://lwnov602.igrupobbva:35000/groups/XEQY/settings
- **Jobs Saver:** http://lwnov602.igrupobbva:35000/groups/XEQY/jobssaver
- **Market Data:** http://lwnov602.igrupobbva:35000/groups/XEQY/marketdata
- **Calibrator:** http://lwnov602.igrupobbva:35000/groups/XEQY/calibrator
- **Admin:** http://lwnov602.igrupobbva:35000/groups/XEQY/admin
- **Utils:** http://lwnov602.igrupobbva:35000/groups/XEQY/utils
- **Models:** http://lwnov602.igrupobbva:35000/groups/XEQY/models

Los microservicios **Fol Grid** y **Calibrator Saver** no es necesario descargarlos ya que son sistemas externos y no hará falta conectarse a ellos. Por otro lado los repositorios **Utils** y

Models no corresponden a un microservicio sino que son librerias que se utilizaran varios microservicios.

Sabiendo la URL de todos los microservicios, el siguiente paso es clonar los proyectos con la herramienta **SourceTree** de la forma que se muestra en la **Ilustración 36**.

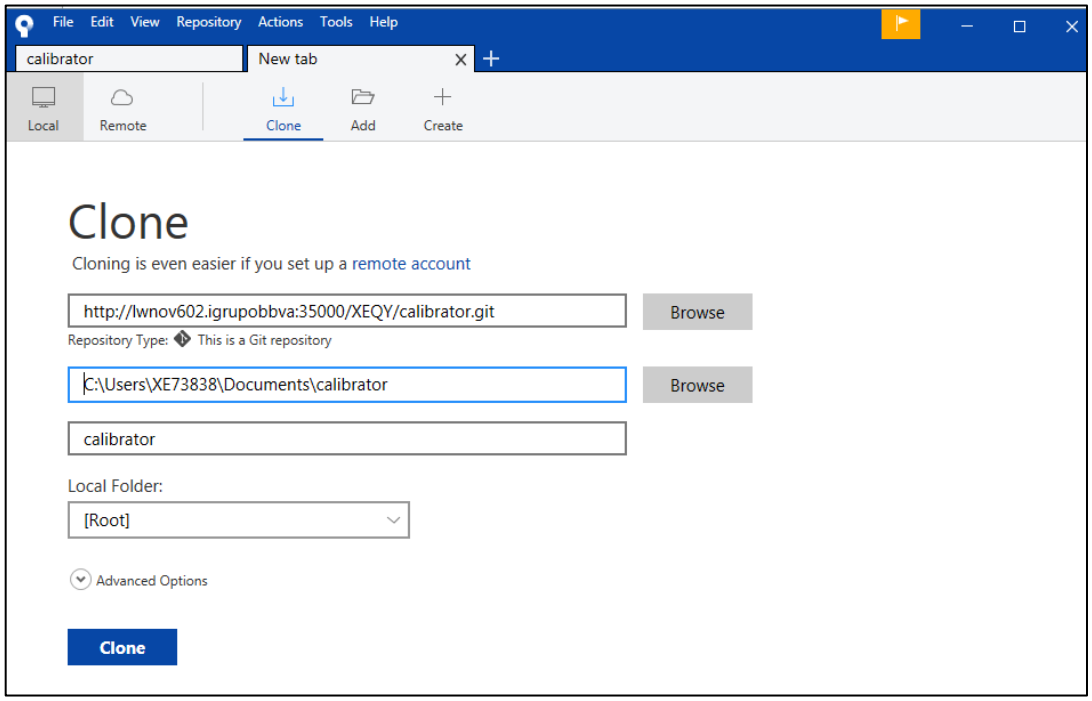


Ilustración 36: Clonar proyectos con Source Tree

Una vez hecho esto, se creará un fichero **pom.xml** que incluya todos los microservicios de la aplicación. Esto es necesario para generar la estructura de proyecto **Apache Maven**. El fichero **pom.xml** debe de tener la siguiente extructura:

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.bbva.xeqy</groupId>
    <artifactId>CalibratorAggregator</artifactId>
    <version>0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
```

```
        <module>utils</module>
        <module>model</module>
        <module>launcher</module>
        <module>orchestrator</module>
        <module>marketdata</module>
        <module>jobstatus</module>
        <module>calibrator</module>
        <module>settings</module>
        <module>admin</module>
    </modules>
</project>
```

El Proyecto debe de quedar con la estructura mostrada en la **Ilustración 37**.

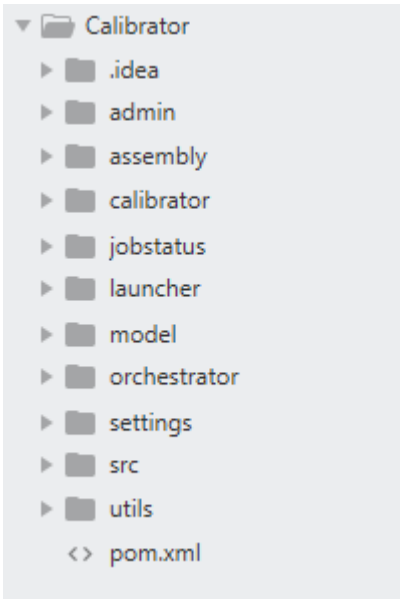


Ilustración 37: Estructura del proyecto

PLATAFORMA NOVA

El primer para instalar la aplicación es instalar la **Plataforma Nova**. La instalación y la puesta en marcha de esta aplicación es bastante sencilla.

REQUERIMIENTOS

- **Java JDK:** se requiere una version 1.8.0_1XXX o superior de JDK de Java como instalación mínima.

- **Memoria**: El puesto local de NOVA con todos los servicios arrancados al mismo tiempo consumen aproximadamente 1,5 GB de memoria. Pero se recomienda tener al menos 8 Gb de memoria RAM.
- **CPU:** Se recomienda disponer de al menos 2/4 cores para poder ejecutar con fluidez el conjunto de operaciones y servicios de la plataforma NOVA con el resto de operaciones normales del sistema operativo.
- **Sistema Operativo:** Windows XP o versiones superiores.
- **Apache Maven**

Además, es imprescindible que dentro del fichero de configuración de Maven se añadan los repositorios de NOVA. Cuando la aplicación esté lista para subir al entorno de producción de la **Plataforma Nova**, es necesario asegurarse de que todas las dependencias del proyecto se encuentren en los repositorios de esta aplicación.

El fichero **settings.xml** de configuración debe de tener una estructura similar a la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>%MAVEN_LOCAL_REPOSITORY_PATH%</localRepository>
    <servers>
        <server>
            <username>%USER%</username>
            <password>%PASSWORD%</password>
            <id>nova_central</id>
        </server>
        <server>
            <username>%USER%</username>
            <password>%PASSWORD%</password>
            <id>snapshots</id>
        </server>
        <server>
            <username>%USER%</username>
            <password>%PASSWORD%</password>
            <id>nova</id>
        </server>
    </servers>
    <profiles>
        <profile>
            <repositories>
                <repository>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
```

```
                    <id>nova_central</id>
                    <name>RELEASE_NO_PLUGIN_VIRTUAL_REPO</name>
<url>http://cibartifactory.igrupobbva:8084/artifactory/RELEASE_NO_PLUGIN_VIRTUAL_
REPO</url>
                </repository>
                <repository>
                    <snapshots/>
                    <id>snapshots</id>
                    <name>SNAPSHOT_NO_PLUGIN_VIRTUAL_REPO</name>
<url>http://cibartifactory.igrupobbva:8084/artifactory/SNAPSHOT_NO_PLUGIN_VIRTUAL
_REPO</url>
                </repository>
                <repository>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <id>nova</id>
                    <name>NOVA_remote</name>
<url>http://cibartifactory.igrupobbva:8084/artifactory/NOVA_remote</url>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <id>nova_central</id>
                    <name>RELEASE_PLUGIN_VIRTUAL_REPO</name>
<url>http://cibartifactory.igrupobbva:8084/artifactory/RELEASE_PLUGIN_VIRTUAL_REP
O</url>
                </pluginRepository>
                <pluginRepository>
                    <snapshots/>
                    <id>snapshots</id>
                    <name>SNAPSHOT_PLUGIN_VIRTUAL_REPO</name>

<url>http://cibartifactory.igrupobbva:8084/artifactory/SNAPSHOT_PLUGIN_VIRTUAL_RE
PO</url>
                </pluginRepository>
                <pluginRepository>
                    <snapshots/>
                    <id>nova</id>
                    <name>NOVA_remote</name>
<url>http://cibartifactory.igrupobbva:8084/artifactory/NOVA_remote</url>
                </pluginRepository>
            </pluginRepositories>
            <id>artifactory</id>
        </profile>
    </profiles>
    <activeProfiles>
        <activeProfile>artifactory</activeProfile>
    </activeProfiles>
</settings>
```

SERVICIOS INCLUIDOS EN LA INSTALACIÓN

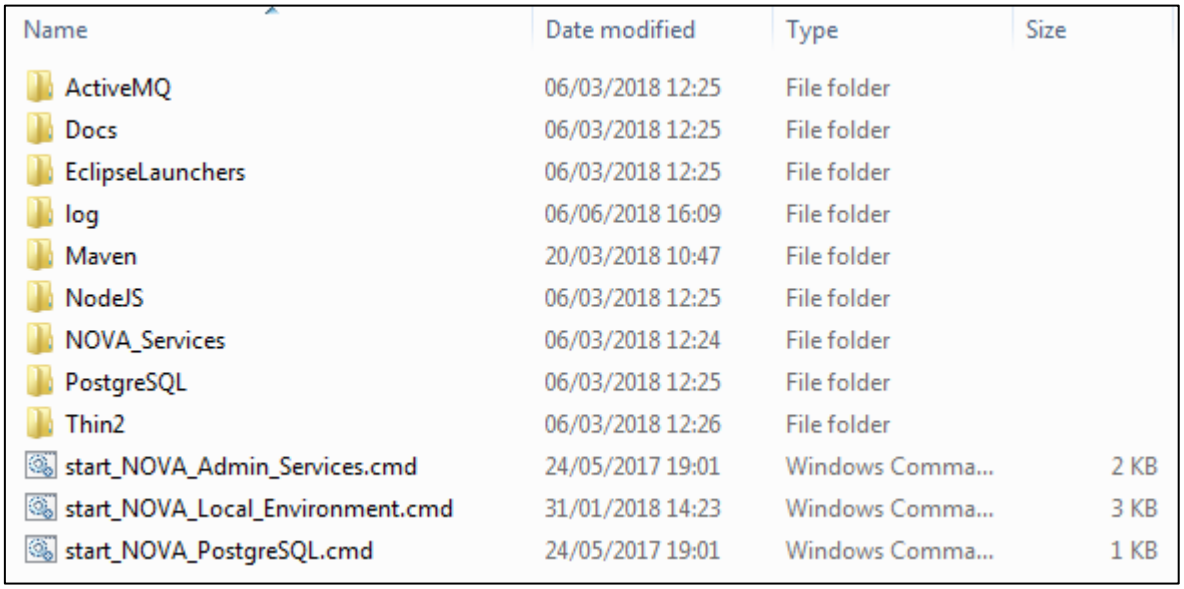
La instalación de Nova consiste en instalar los siguientes elementos:

- Config service.
- Eureka service.
- Spring boot admin console service. (desde la versión 3.0.1)
- Zuul service.
- Base de datos PostgreSQL
- NOVA Starter service (desde la versión 0.0.2)

INSTALACIÓN

Descargar la última versión disponible de la **Plataforma Nova** y descomprimir el archivo **com.bbva.enoa.local_environment-4.0.0.zip**. Este archivo ocupa alrededor de 1,5 Gb por lo que el tiempo de descompresión puede ser largo.

Configurar la variable de sistema **%NOVA_LOCAL_ENVIRONMENT_PATH%**, que indicará el directorio de instalación de la **Plataforma Nova**.



Name	Date modified	Type	Size
ActiveMQ	06/03/2018 12:25	File folder	
Docs	06/03/2018 12:25	File folder	
EclipseLaunchers	06/03/2018 12:25	File folder	
log	06/06/2018 16:09	File folder	
Maven	20/03/2018 10:47	File folder	
NodeJS	06/03/2018 12:25	File folder	
NOVA_Services	06/03/2018 12:24	File folder	
PostgreSQL	06/03/2018 12:25	File folder	
Thin2	06/03/2018 12:26	File folder	
start_NOVA_Admin_Services.cmd	24/05/2017 19:01	Windows Comma...	2 KB
start_NOVA_Local_Environment.cmd	31/01/2018 14:23	Windows Comma...	3 KB
start_NOVA_PostgreSQL.cmd	24/05/2017 19:01	Windows Comma...	1 KB

Ilustración 38: Estructura de ficheros de NOVA Local Enviroment

Para iniciar la **Plataforma Nova** se debe de ejecutar el siguiente comando en un cmd.

```
%NOVA_LOCAL_ENVIRONMENT_PATH%\start_NOVA_Local_Environment.cmd
```

Este comando arrancará de forma simultánea todos servicios necesarios en la **Plataforma Nova**, que son los incluidos en la siguiente lista:

- **PostgreSQL:** El primer que realiza NOVA Local Environment es arrancar la consola de PostgreSQL. En ella está almacenadas las configuraciones del servicio Config Server, por lo tanto, es un requisito inicial de arranque.
- **Eureka Service:** servicio donde cualquier servicio se registra y sirve para tener identificados cada uno de los servicios de la plataforma, de tal manera que los servicios pueden conocer donde están otros servicios a través de Eureka.
- **Config Service:** que sirve para almacenar las configuraciones de los servicios y disponerlas cuando los estos arrancan.
- **Zuul Service:** enrutador por el que se accede a cualquiera de los servicios desde dentro y fuera de la plataforma. Conoce la ubicación de cualquier servicio gracias a Eureka.
- **Thin:** aunque no es un servicio como tal, crea en el SS.OO las variables de entorno necesarias para trabajar con las herramienta de la capa de presentación Thin2, dejando el entorno local listo.

Una vez ejecutado el comando en la consola, debemos de esperar a ver una pantalla como la mostrada en la **Ilustración 39**, en la que se indica que la **Plataforma Nova** ya está lista para ser utilizada.

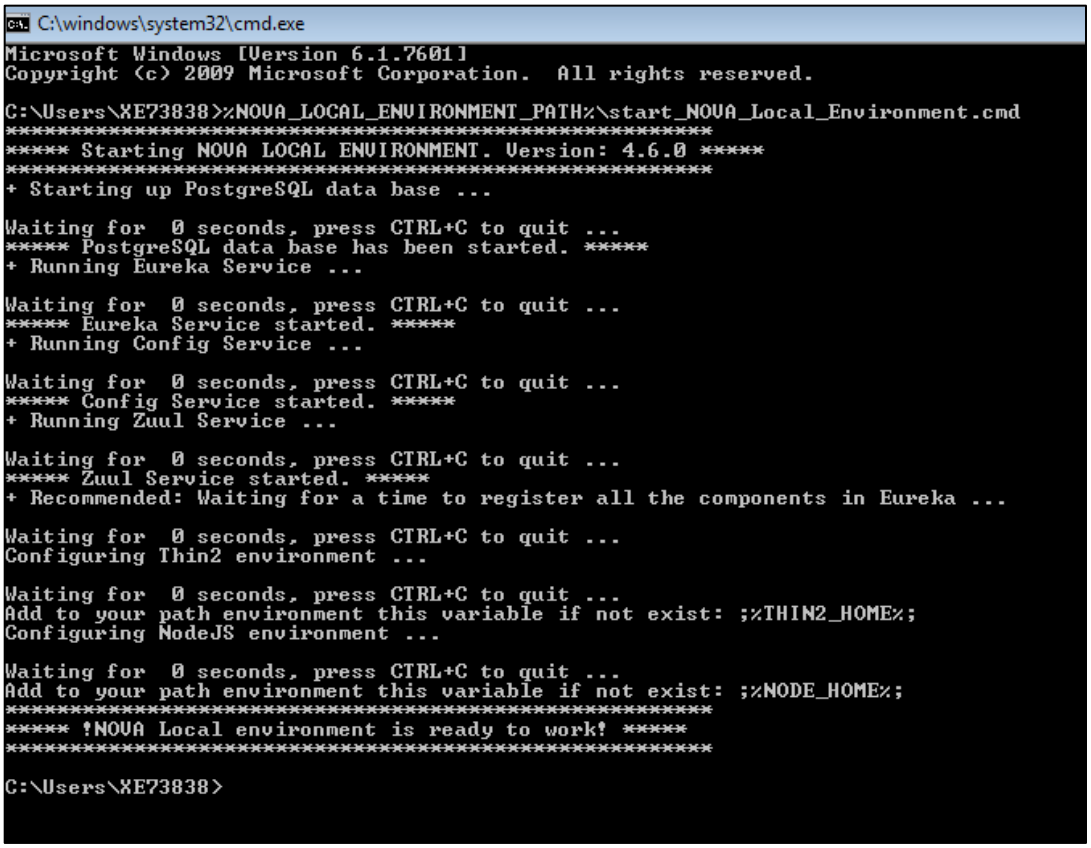


Ilustración 39: Arranque de la plataforma Nova

PREPARACIÓN DE LA PLATAFORMA SYMPHONY

No es necesario instalar la **Plataforma Symphony**. Debido a que la **Plataforma Nova** no permite instalar dependencias directamente sobre los contenedores **Docker**, ha habido que desarrollar una solución para poder ejecutar el flujo de la aplicación únicamente con las dependencias instaladas con **Apache Maven**.

La solución adoptada para evitar instalar el cliente de la **Plataforma Nova** pasa por los siguientes pasos:

- Compilar el **Calibrator** con **Maven**, seleccionando el perfil **win64**. Las librerías de la **Plataforma Symphony** son diferentes para el entorno de desarrollo local (**Windows**) que para el entorno de la **Plataforma Nova** (**Linux**) por lo que es

necesario la utilización de perfiles para que **Maven** instale las librerías necesarias en cada uno de los entornos.

- En el directorio **Maven** local, donde se descargan todas las dependencias, navegar hasta el directorio com\platform\symphony\symphony-native\6.1.1, y descomprimir el archivo **symphony-native-6.1.1-win64.jar** en un directorio local ("**symphony-path**")
- Añadir el directorio anterior ("**symphony-path**") al la variable de sistema **PATH**.
- Añadir una nueva variable de sistema **EGO_SEC_PLUGIN=sec_ego_default**

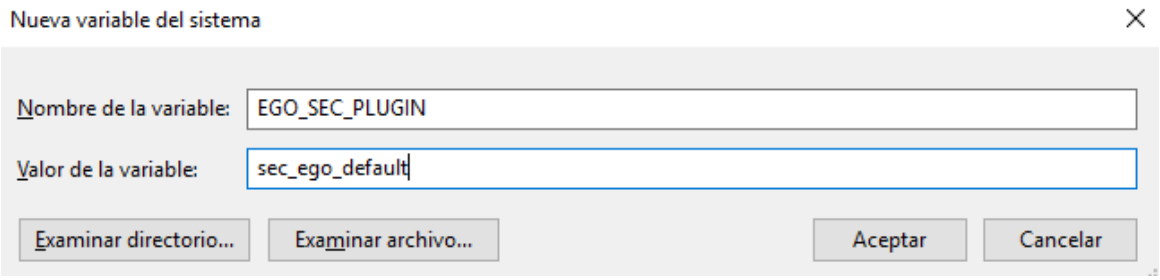


Ilustración 40: Añadir variable EGO_SEC_PLUGIN

BIBLIOGRAFÍA

[1] FRTB, «FRTB : A Summary of the Regulations,» 6 Febrero 2016. [En línea]. Available: http://frtb.info/summary-of-the-frtb-regulations/. [Último acceso: 22 Febrero 2018].

[2] A. M. d. C. G. Oterino, «¿Qué es Docker?,» 24 Julio 2015. [En línea]. Available: http://www.javiergarzas.com/2015/07/que-es-docker-sencillo.html. [Último acceso: 20 Febrero 2018].

[3] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/PostgreSQL. [Último acceso: 2018 07 11].

[4] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Lucidchart. [Último acceso: 2018 07 11].

[5] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/JIRA. [Último acceso: 11 07 2018].

[6] . M. Rouse, «Docker,» searchdatacenter. [En línea]. [Último acceso: 14 Marzo 2018].

[7] J. A. Saiz, «Microservicios, ¿Están las empresas preparadas?,» 2 Enero 2016. [En línea]. Available: http://joseantoniosaiz.com/microservicios-introduccion-estan-empresas-preparadas/. [Último acceso: 20 Febrero 2018].

[8] S. Newman, Buildeing Microservices, Sebastopol: O'Reilly Media, 2015.

[9] E. A., «Qué son Microservicios y ejemplos reales de uso,» 19 Abril 2016. [En línea]. Available: https://openwebinars.net/blog/microservicios-que-son/. [Último acceso: 20 Febrero 2018].

[10] A. M. d. C. G. Oterino, «¿Qué es eso de los microservicios?,» 19 Junio 2015. [En línea]. Available: http://www.javiergarzas.com/2015/06/microservicios.html. [Último acceso: 20 Febrero 2018].

[11] «Get Started with Docker,» [En línea]. Available: https://docs.docker.com/get-started/#docker-concepts. [Último acceso: 20 Febrero 2018].

[12] E. Reinhold, «Rewriting Uber Engineering: The Opportunities Microservices Provide,» 20 Abril 2016. [En línea]. Available: https://eng.uber.com/building-tincup/. [Último acceso: 20 Febrero 2018].

[13] Economipedia, «Basilea I,» [En línea]. Available: http://economipedia.com/definiciones/basilea-i.html. [Último acceso: 04 Abril 2018].

[14] Economipedia, «Basilea II,» [En línea]. Available: http://economipedia.com/definiciones/basilea-ii.html. [Último acceso: 04 Abril 2018].

[15] M. V. G. Amoraga, «Comparación entre el modelo estándar actual y el modelo estándar bajo FRTB,» Universidad Complutense de Madrid, Madrid, 2016.

[16] B. BBVA, «Basilea III,» 24 Enero 2015. [En línea]. Available: https://www.bbva.com/es/economia-todos-basilea-iii/. [Último acceso: 04 Abril 2018].

[17] «Summary of FRTB,» [En línea]. Available: http://www.optimissa.com/frtb/. [Último acceso: 04 Abril 2018].

[18] B. f. i. settlements, Fundamental review of, 2014.

[19] J. J. S. Ávila, «Metodologías de medición del riesgo de mercado,» [En línea]. Available: https://revistas.unal.edu.co/index.php/innovar/article/view/21641/34906. [Último acceso: 27 Febrero 2018].

[20] A. Blog, «MASTERING THE FRTB,» [En línea]. Available: https://www.accenture.com/us-en/insight-fundamental-review-trading-book-theory-action. [Último acceso: 22 Febrero 2018].

[21] B. BBVA, «¿Qué es la metodología 'Agile'?,» 2016 Enero 20. [En línea]. Available: https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/.