UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

MÁSTER EN INGENIERÍA WEB

Proyecto Fin de Máster

Arquitectura de Microservicios con RESTful

Autores

Alberto Cols Fermín

Mariana Salcedo Real

Tutor

Luis Fernández Muñoz

Julio de 2017

RESUMEN

Desarrollar un sistema de software es un trabajo arduo desde su concepción hasta su despliegue. Durante todo este proceso se toman en consideración factores como los requerimientos del negocio, la afluencia de usuarios esperada, la escalabilidad del sistema, la forma en que se desplegará el sistema, entre otros tantos factores. Sin embargo, a pesar de que a lo largo del proceso de desarrollo del sistema se toman en cuenta continuamente estos factores, existe una parte del proceso que resulta vital para el buen devenir del sistema: el diseño de la arquitectura.

Ahora bien, dentro de esta delicada tarea existen dos principales vertientes: optar por una arquitectura monolítica más tradicional, u optar por un diseño orientado a microservicios. Este trabajo busca explicar sobre qué trata la Arquitectura basada en Microservicios, las ventajas y desventajas que tiene este tipo de diseño, como llevar a cabo la división de un sistema en microservicios, y finalmente cuando debería o no ser utilizada esta arquitectura.

Para apoyar esta investigación y dar sustento a las conclusiones obtenidas se tomó como dominio un sistema de Gestión Universitaria, pero, al ser tan extenso se escogieron a su vez tres(03) subdominios: gestión de profesores, de materias y de nómina. Teniendo en cuenta este dominio, se realizaron tres(03) implementaciones, estas cubren un diseño monolítico y dos posibles soluciones basadas en microservicios. Concluidas dichas implementaciones se pasa a resumir qué ventajas poseen y que problemas fueron encontrados en cada una.

Por último se concluye en que circunstancias es recomendable aplicar microservicios tomando como base los resultados obtenidos anteriormente.

Palabras clave

Arquitectura, Microservicio, Spring, RESTful

ABSTRACT

Developing a software system is hard work from conception to deployment. Throughout this process, many factors have to be taken into account, such as: business requirements, expected user inflows, system scalability, how the system will be deployed, and many others. However, although these factors are continually taken into account throughout the development process of the system, there is a part of the process that is vital to the good development of the system: architecture design.

Within this delicate task there are two main approaches: opting for a traditional monolithic architecture, or opt for a microservice-oriented design. This work seeks to explain what is a Microservices based architecture, the advantages and disadvantages of this type of design, how to carry out the division of a monolithic system into microservices one, and, finally, when this architecture should be used.

In order to support this research and support the conclusions obtained, a University Management system was taken as a domain, but, since this is a really vast domain, three (03) subdomains were chosen: management of teachers, subjects and paysheet. Taking into account this domain, three (03) implementations were done: these cover a monolithic design and two possible solutions based on microservices. Once these implementations have been completed, it is possible to summarize what advantages they have and what problems were found in each one.

Finally, it is concluded in what circumstances it is advisable to apply microservices based on the results obtained previously.

Keywords

Architecture, Microservice, Spring, RESTful

TABLA DE CONTENIDOS

OBJETIVOS

Los objetivos principales de este Trabajo Final de Máster son:

- Entender en qué consiste un Microservicio y conocer sus características principales.
- Comprender en qué situaciones es recomendable aplicar una Arquitectura basada en Microservicios.
- Comprender las distintas maneras que existen de dividir una aplicación en Microservicios.
- Comparar una Arquitectura Monolítica versus una basada en Microservicios.
- Proponer la base de un sistema de gestión para profesores y materias basado en una arquitectura de microservicios.

CAPÍTULO 1: DEFINICIÓN DE ARQUITECTURAS

Arquitectura Monolítica

La arquitectura monolítica es aquella en la que todos los componentes o módulos de un *software* se encuentran desarrollados y desplegados en una única aplicación. Sin embargo, esto no la limita a no tener dependencias de terceros desplegados en servidores independientes para poder satisfacer ciertas funcionalidades (Ej. manejadores de cola, manejadores de bases de datos, etc.).[1]

Toda la lógica de negocio está auto-contenida en si misma, cuyos componentes están interconectados y son interdependientes, a diferencia de arquitecturas modulares cuyos componentes están poco acoplados.[2]

Ventajas iniciales de las Arquitecturas Monolíticas[3]

- **Rápido desarrollo inicial**

  Los desarrolladores pueden enfocarse en implementar todas las funcionalidades sin preocuparse por la integración entre distintos componentes en distintos servidores.

- **Ubicación del código fuente**

  El código fuente de toda la aplicación se encuentra en un único paquete por lo que navegar a través del mismo es sencillo utilizando algún *IDE* (*Integrated Development Environment*).

- **Fácil despliegue**

  Sólo es necesario desplegar el paquete de la aplicación en un servidor. Además, todos los *IDEs* actuales están enfocados al desarrollo de aplicaciones monolíticas por lo cual es relativamente sencillo generar el paquete la aplicación (archivos *.WAR*, *.JAR*, *.EAR*) desde el mismo *IDE.*

- **Fácil de escalar horizontalmente**

  Basta con correr varias copias de la misma en varios servidores manejados con un manejador de cargas.

Desventajas de las Arquitecturas Monolíticas

Sin embargo, a medida que la aplicación va creciendo y el equipo de desarrollo se va haciendo más grande, aparecen diversos problemas que se vuelven cada vez más significativos:

- **Dificultad de adaptación**

  La adaptación de nuevos desarrolladores al equipo de trabajo tiende a ser lenta si el monolito tiene gran tamaño, esto conlleva a que la velocidad de desarrollo del equipo se vea afectada. Más aún, si no existe una buena metodología de trabajo, la aplicación cada vez será más difícil de entender y de modificar, lo cual conlleva que cualquier nuevo cambio por parte de los nuevos desarrolladores puedan influir negativamente en la calidad del código.[4]

- **Incremento en el tiempo de despliegue**

  Mientras más grande sea el monolito, más tiempo tomará la inicialización del mismo lo cual influye negativamente en la productividad de los desarrolladores puesto que tendrán que esperar períodos de tiempo más prolongados para iniciar la aplicación. Esto también influye en el tiempo de despliegue a producción de la aplicación.

  Aunado a esto se tiene el problema de que cualquier cambio que se realice en un componente (así este no afecte a los demás componentes) implica re-empaquetar, re-ensamblar y re-desplegar la aplicación completa, aumentando los tiempos de espera de todos los miembros del equipo.[4]

- **Problemas para escalar eficientemente la aplicación**

  A pesar de que la aplicación puede escalar horizontalmente desplegándola en varios servidores, existen varios problemas con este enfoque:

  - Todas las instancias de la misma tienen acceso a toda la data lo cual dificulta el *cacheo* de la misma e incrementa los accesos a memoria, ralentizando la aplicación.
  - Si existe algún *bug* en un componente en específico (por ejemplo, un *memory leak*) esto podría afectar a toda la aplicación. Además, como todas las instancias de la aplicación son iguales, el error afectará la disponibilidad global de la misma sin importar en qué servidor se encuentre desplegada.

- Si un servicio de la aplicación es el que posee más carga de trabajo y se desea tener varias instancias de este únicamente, esto no podría ser posible en este tipo de arquitectura.

Tomando por ejemplo una aplicación de venta de productos que tenga tres servicios: *Servicio de Clientes*, *Servicio de Carrito de Venta* y *Servicio de Productos*. En el caso que el *Servicio de Carrito de Venta* requiera más recursos para poder manejar las peticiones de los clientes tendríamos que crear instancias completas de la aplicación.[4]

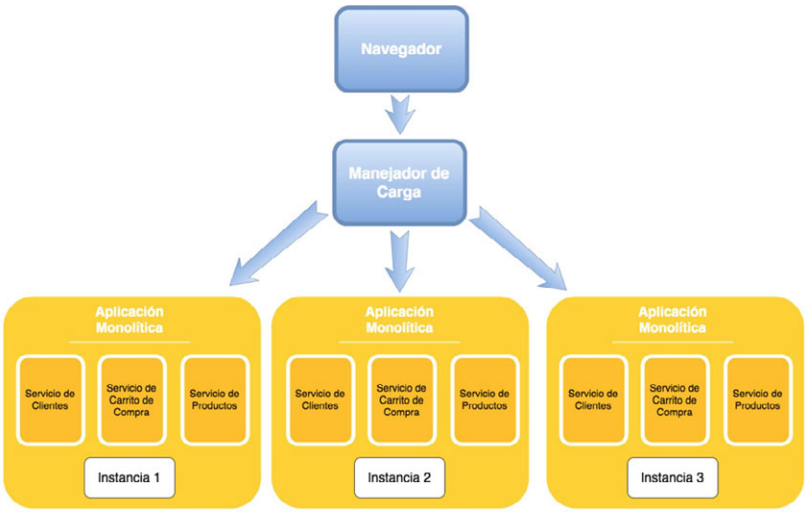


Figura 1

Problema en la escalabilidad horizontal cuando se necesita escalar un único componente

Fuente: Elaboración Propia

Como puede observarse en la Figura 1, para tener varias instancias del *Servicio de Carrito de Compra* se debe instanciar el monolito en su totalidad, lo cual implica un gasto innecesario de recursos puesto que los demás servicios no necesitan ser escalados.

- **Componentes con distintos requerimientos**

Cada componente del *software* puede necesitar distintos tipos de recursos (uno puede requerir más capacidad de procesamiento mientras otro más capacidad de memoria), al tener un solo monolito no se podría escalar verticalmente cada componente independientemente según sus necesidades.[4]

- **Difícil de adaptar a nuevas tecnologías**

Una arquitectura monolítica fuerza a la aplicación a estar atada al mismo *stack* tecnológico desde el inicio de la misma. Si se desea utilizar un lenguaje distinto para algún componente, esto no podría ser implementado puesto que la aplicación está atada a la decisión inicial.[4]

Además, en caso de tener que cambiar de lenguaje o de *framework* porque el actual se vuelva obsoleto o cambien a grandes rasgos los requerimientos del sistema, se vuelve más complicado el proceso de migrar gradualmente la aplicación para adaptarse a las nuevas tecnologías, quizás teniendo que llegar al punto de reescribir la aplicación en su totalidad lo cual es un proceso bastante riesgoso y costoso.

Diseño Orientado a Dominio

El Diseño Orientado a Dominio es un enfoque para desarrollar software para necesidades complejas conectando profundamente la implementación a un modelo en evolución de los conceptos básicos de negocio.

Tiene las siguientes premisas:

- El foco principal del proyecto es el dominio y su la lógica.
- Basar el diseño complejo en un modelo.
- Se debe realizar una colaboración entre el equipo técnico y los expertos del dominio para ir encontrando de manera evolutiva el centro del problema

Las premisas es simple, pero hacer uso de ellas como debe ser, en el día a día es complicado. Requiere nuevas habilidades y disciplina, y un enfoque sistemático al momento de desarrollar software.

El Diseño Orientado a Dominio no es una tecnología o una metodología. DDD proporciona una estructura de prácticas y terminología para tomar decisiones de diseño que enfocan y aceleran proyectos de software que se ocupan de dominios complicados.[5]

Arquitectura basada en Microservicios

La arquitectura basada en Microservicios es una variante de la arquitectura orientada a servicios (SOA), cada servicio debe estar finamente separados y se comunican entre ellos utilizando métodos de comunicación ligeros como puede ser un recurso HTTP a través de un API.[6]

Es una forma particular de diseñar aplicaciones de software como *suites* de servicios siendo ejecutado cada uno en un proceso propio y que a su vez pueden ser desplegados de forma independiente, para esto se utiliza toda una maquinaria de despliegues continuos. Cada Microservicio puede estar implementado utilizando distintos *frameworks*, lenguajes de programación e incluso diferentes tecnologías de almacenamiento de datos.

Aún no existe una definición exacta de este tipo de arquitectura, pero sí existen una serie de características que se deben cumplir.

Características de una Arquitectura basada en Microservicios

- **Componentización vía servicios**

  Un componente se define como una unidad de software que es independientemente reemplazable y actualizable, un ejemplo son las librerías.

  Los microservicios harán uso de librerías que formarán parte de sus dependencias, pero, la manera en que se llevan a cabo los componentes en este tipo de arquitecturas es separando el software en distintos servicios, que se comunicarán entre ellos mediante un servicio web o una llamada a un procedimiento remoto (RPC).

  La razón principal para utilizar servicios como componentes, en vez de utilizarlos como librerías, es que estos son desplegables individualmente. Si una aplicación está compuesta por múltiples librerías un cambio en una de estas requeriría que la aplicación completa fuese re-desplegada, en cambio si la aplicación está separada en microservicios si es necesario modificar uno, sólo habría que desplegar la nueva versión del microservicio, por supuesto habrán cambios que necesiten cambios en más de un microservicio, pero, esto se puede minimizar a través de límites de servicios cohesivos y mecanismos de evolución en los contratos de estos.[6]

- **Organización entorno a las capacidades del negocio**

  Cada Microservicio implementará un área del negocio con todo lo que esto conlleva: interfaz de usuario, lógica de negocio, almacenamiento de datos y cualquier colaboración con algún servicio externo que sea necesaria, como consecuencia de esto el equipo encargado del desarrollo debe ser un equipo multidisciplinario, que sea capaz de realizar el desarrollo de las funcionalidades requeridas.[6]

- **Productos no proyectos**

  El enfoque tradicional al momento de llevar a cabo el desarrollo de una aplicación es de un proyecto, esto quiere decir que al culminar la implementación de la misma el mantenimiento es realizado por otras personas o incluso otra empresa y el equipo es disuelto.

  En cambio la visión que se propone con este tipo de arquitecturas es orientada a producto, dígase, que al momento de comenzar un desarrollo el equipo es responsable de él, desde su concepción hasta su mantenimiento. Esto también viene ligado con las capacidades de negocio, porque en vez de ver una aplicación como conjunto de funcionalidades que se debe realizar, se ve como una relación en la que se busca mejorar continuamente la capacidad empresarial mediante el software.

  Este enfoque puede ser utilizado también en una arquitectura Monolítica, pero por la granularidad de los equipos es más sencillo aplicarlo en una arquitectura de Microservicios.[6]

- ***Smart Endpoints Dumbs Pipes***

  Una aplicación basada en microservicios deber ser lo más desacoplada y cohesiva posible, cada uno debe ser dueño de su dominio y comportarse como un filtro al estilo de Unix (se recibe una solicitud, se aplica la lógica necesaria y se produce una respuesta).

  Estos se pueden comunicar usando un protocolo estilo REST y/o mensajería ligera, para este último se suele utilizar estructuras sencillas, que sirvan como un enrutador de mensajes como por ejemplo: *RabbitMq* o *ZeroMq*, estos funcionan como un intermedio sin lógica ya que los extremos son los que la poseen, que en este caso serían los Microservicios.[6]

- **Administración Descentralizada**

  Una de las consecuencias de la administración centralizada es que tiende a utilizar una sola tecnología, se ha demostrado que este enfoque puede no ser el más apropiado, ya que no todos los problemas se solucionan de la misma manera, por lo que tiene sentido que una solución posea múltiples tecnologías.

  Cuando se separa una aplicación Monolítica en microservicios se deja la posibilidad de seleccionar la mejor opción para cada uno a nivel de lenguaje de programación, *frameworks*, librerías, sistemas de almacenado, etc.

  Además de seleccionar con las tecnologías que serán utilizadas el equipo puede definir los estándares que seguirán, se pueden crear librerías o herramientas que sean útiles entre equipos que presenten algún problema en común.[6]

- **Manejo de Datos Descentralizado**

  La manera más abstracta de ver la descentralización de los datos significa que el modelo conceptual del "todo" va a cambiar entre microservicios. Esto suele ser un inconveniente al integrar un sistema muy extenso.

  Puede verse el caso en que la visión de cliente sea distinta en el módulo de ventas y el módulo de soporte, por lo que puede ser que existan atributos comunes o aún peor atributos comunes pero con semánticas distintas.

  Esto suele suceder entre aplicaciones pero también se puede ver dentro de las aplicaciones, sobre todo cuando las aplicaciones están divididas en componentes separados. Una manera de verlo es siguiendo la noción de *Bounded Context* del Diseño Orientado a Dominio, este divide un dominio complejo en varios *bounded context* y mapea las relaciones entre ellos.

  Así como se descentralizan las decisiones sobre el modelo conceptual, los Microservicios descentralizan las decisiones sobre el almacenaje de datos. Mientras que un sistema monolítico suele tener una sola base de datos lógica para persistir la data, los Microservicios prefieren que cada uno controle su propia base de datos, así sea con diferentes instancias del mismo manejador de base de datos, o incluso con un sistema de base de datos totalmente diferente.[6]

- **Automatización de Infraestructura**

  Las técnicas de automatización de infraestructura han evolucionado muchísimo a través de los últimos años, dada a la evolución de la nube y AWS en

particular han reducido la complejidad de construir, desplegar y la operatividad en general de los Microservicios.

Muchos de los productos o sistemas que se construyen con Microservicios son hechos por equipos que tienen bastante experiencia en *Continuos Delivery*, estos suelen utilizar este tipo de técnicas de forma extensiva.

Cuando se ha utilizado esta técnica de despliegue en aplicaciones Monolíticas de manera continua hasta hacerlo parte del día a día realizar el despliegue de más aplicaciones(en este caso Microservicios) debería ser natural.[6]

- **Diseño para fallos**

Una de las consecuencias del desarrollo orientado a Microservicios es el hecho que estos deben ser programados para soportar que otros fallen ya que esto puede suceder y el Microservicio debe ser capaz de responder de la manera más elegante posible.

Ya que estos pueden fallar en cualquier momento, es importante que estos fallos se detecten rápidamente y de ser posible que el Microservicio sea restaurado de manera automática.

Monitorear el comportamiento de los microservicios es sumamente importante ya que ayuda a prevenir comportamientos no deseados y corregirlos rápidamente.

Los equipos procuran realizar un buen plan y herramientas para realizar el monitoreo y *logging* para cada Microservicios, tales como tableros que muestren el estado de cada uno y algunas métricas que sean relevantes para poder detectar fallos.[6]

- **Diseño Evolutivo**

Las personas que siguen este tipo de arquitecturas, suelen utilizar el diseño evolutivo y la descomposición en Microservicios como una herramienta para los desarrolladores que permite controlar los cambios en las aplicaciones sin necesidad de hacerlos más lentos.

Un sistema que haya sido creado inicialmente como una aplicación monolítica puede ir evolucionando a Microservicios, puede hacerlo de manera progresiva, es decir, ir separando poco a poco sus módulos en componentes y estos irlos llevando a Microservicios, teniendo aún como su centro una

aplicación monolítica; o ir agregando funcionalidades en forma de Microservicios.

Colocar los componentes como servicios aparte permiten realizar salidas a producción de una manera más granular y controlada, por ejemplo, si se realizó un cambio solo en un Microservicio no es necesario desplegar la aplicación completa, con volver desplegar ese Microservicio es suficiente.[6]

Maneras de dividir una aplicación en Microservicios

Idealmente un Microservicio debe hacerse cargo de un conjunto de responsabilidades reducido. La responsabilidad de una clase puede ser definida como la razón de cambio, y esta solo debe tener una razón de cambio, esto siguiendo el Principio de Única Responsabilidad[7], por lo que tiene sentido aplicar lo mismo a los Microservicios.

Se puede tomar como ejemplo los servicios en Linux, por ejemplo *grep, cat* o *find*; cada uno de estos se encarga de una sola responsabilidad y normalmente lo hacen excepcionalmente bien, además estos se pueden combinar entre sí para realizar tareas complejas, lo que encaja perfectamente con la forma en que se desea que los Microservicios se comporten.

No existe una única forma en la que se puede dividir una aplicación en microservicios, y se puede decir que es un proceso que se considera un arte, pero existen estrategias que se pueden seguir para llevar a cabo esta separación. A continuación son descritas:

- **Por Capacidades de Negocio**

  Se conoce a las Capacidades de Negocio como las partes que permiten generar valor al negocio, por ejemplo: la gestión de órdenes o la gestión de cliente; por lo que una de las formas naturales de dividir un sistema sería justamente siguiendo esa división que ya existe en el negocio como tal.

  Se puede tomar como ejemplo un negocio que tenga las siguientes Capacidades de Negocio:

  - *Product catalog management* (gestión de catálogo de productos)
  - *Inventory management* (gestión de inventario)
  - *Order management* (gestión de ordenes)
  - *Delivery management* (gestión de envíos)

Nuestros sistema estaría compuesto por un microservicio por Capacidad de Negocio

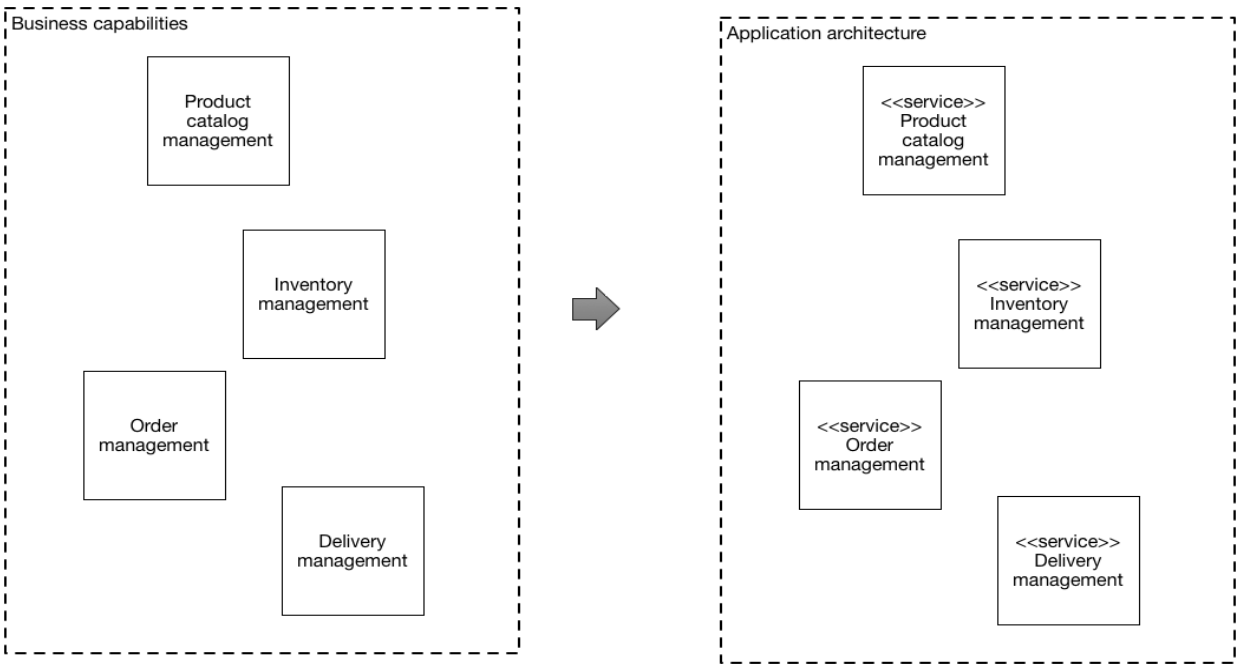


Figura 2
Fuente: Pattern: Decompose by subdomain
http://microservices.io/patterns/decomposition/decompose-by-subdomain.html

De esta manera podemos obtener una arquitectura estable ya que la capacidades de negocios cambian muy poco, se puede tener un sistema cohesivo, poco acoplado con equipos multifuncionales y organizados para generar valor al negocio.[8]

- **Por Dominio**

Al dividir por dominio un sistema se debe seguir el Diseño Orientado a Dominio (DDD[5] en sus siglas en inglés), este se encuentra basado en los modelos del dominio. Un modelo funciona como un Lenguaje Ubicuo[9] que ayuda a los desarrolladores del software a comunicarse con los expertos del dominio. Actúa también como la base conceptual para el diseño del software en sí, por ejemplo cómo descomponer en objetos o funciones; para ser eficiente un modelo necesita estar unificado, es decir que sea consistente y que no haya contradicciones en el.

El Diseño Orientado a Dominio se enfoca en resolver el problema de una aplicación como su dominio, y este se encuentra conformado por múltiples subdominios, cada uno de estos corresponden a una parte distinta del negocio y pueden estar clasificados de la siguiente forma:

- **Central**: es lo que diferencia al negocio, por lo tanto es la parte más importante de la aplicación.
- **Soporte**: son las partes que componen el negocio pero no las más resaltantes.
- **Genérico**: no son específicos del negocio.

Podemos tomar un sitio de ventas como ejemplo y obtener dos subdominios de él, como podrían ser:

- Ventas
- Soporte

Por lo que nuestro sistema tendría un microservicio para el área de Ventas y otro para el área de Soporte como lo podemos observar en la siguiente imagen.[9]

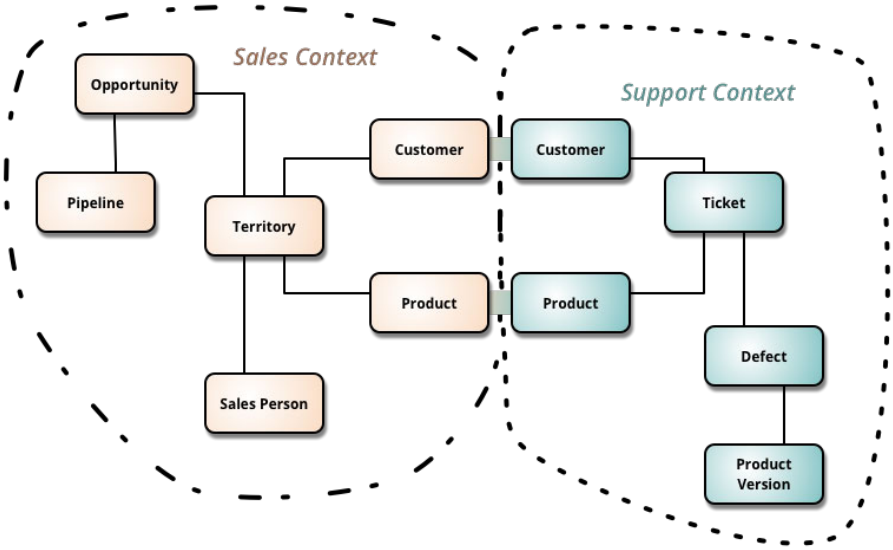


Figura 3.
Fuente: Bounded Context,
https://martinfowler.com/bliki/BoundedContext.html

Ventajas de una Arquitectura basada en Microservicios

- **Fronteras delimitadas entre Módulos**

Este es un beneficio importante pero extraño, porque no hay razón, en teoría, por qué los microservicios deben tener límites de módulo más fuertes que un monolito.

El desacoplamiento con módulos funciona porque los límites del módulo son una barrera para las referencias entre módulos. El problema es que, con un

sistema monolítico, por lo general es bastante fácil de saltarse esta barrera. Distribuir los módulos en servicios separados hace que los límites sean más firmes, lo que hace mucho más difícil encontrar la forma de evitar las divisiones entre módulos.[11]

- **Despliegue Independiente**

  Varios desarrollos de microservicio fueron desencadenados por la dificultad de desplegar monolitos grandes, donde un pequeño cambio en cualquier parte del monolito podría causar el despliegue entero fallara.

  Un principio clave de microservicios es que los servicios son componentes y, por lo tanto, son desplegables de forma independiente. Así que ahora cuando se realiza un cambio, sólo tiene que ser probado y desplegado un sólo servicio. Si falla, no se habrá caído todo el sistema, solamente el servicio que ha necesitado un cambio. Después de todo, debido a la necesidad que existe en este tipo de diseños de diseñar para el fallo, incluso un completo fallo del componente no debe impedir que otras partes del sistema sigan funcionando.

  El gran beneficio de la entrega continua es la reducción del tiempo en el ciclo entre una idea y el software en ejecución. Las organizaciones que hacen esto pueden responder rápidamente a los cambios del mercado, e incluso introducir nuevas características más rápido que su competencia.[12]

- **Diversidad Tecnológica**

  Puesto que cada microservicio es una unidad que se puede desplegar independientemente, cada uno tiene una libertad considerable en sus opciones de tecnología. Los microservicios se pueden escribir en diferentes lenguajes, utilizar diferentes librerías y utilizar diferentes base de datos. Esto permite a los equipos elegir una herramienta adecuada dependiendo del sistema a desarrollar, algunos lenguajes y librerías son más adecuados para ciertos tipos de problemas.

  La discusión de la diversidad técnica a menudo se centra en la mejor herramienta para el trabajo, pero a menudo el mayor beneficio de los microservicios es la cuestión más básica del control de versiones. En un monolito sólo se puede usar una versión única de una librería, una situación que a menudo conduce a actualizaciones problemáticas. Una parte del sistema puede requerir una actualización para usar sus nuevas funcionalidades, pero no puede porque esta actualización rompe otra parte del sistema. Tratar los problemas de versionado de las librerías es uno de esos problemas que se vuelven exponencialmente más difíciles a medida que la base de código se hace más grande.

Con un sistema monolítico, las decisiones tempranas sobre lenguajes y marcos son difíciles de revertir. Después de mucho tiempo de desarrollo, tales decisiones pueden bloquear a los equipos en tecnologías incómodas. Los microservicios permiten a los equipos experimentar con nuevas herramientas, y también migrar gradualmente los sistemas de un servicio a la vez en caso de que una tecnología superior sea relevante.[11]

- **Facilidad para Escalar**

Dado que una aplicación está compuesta de múltiples microservicios que no comparten dependencias externas, el escalado de una instancia de microservicio en el flujo se simplifica mucho: si un microservicio en un flujo se convierte en un cuello de botella debido a la ejecución lenta, este microservicio se puede ejecutar en un servidor potente para un mayor rendimiento si es necesario, o se pueden ejecutar varias instancias del Microservicio en diferentes máquinas para procesar elementos de datos en paralelo.

Contraste la escalabilidad fácil de microservicios con sistemas monolíticos, donde el escalamiento no es trivial; si un módulo tiene un pedazo interno lento del código, no hay manera de hacer ese pedazo individual de código funcionar más rápidamente. Para escalar un sistema monolítico, uno tiene que ejecutar una copia del sistema completo en una máquina diferente e incluso hacer eso no resuelve el cuello de botella de un paso interno lento dentro del monolito o incrementar la capacidad del servidor donde está siendo ejecutado el sistema.[11]

Desventajas de una Arquitectura basada en Microservicios

- **Latencia agregada por la comunicación entre microservicios**

Rendimiento golpeado debido a HTTP, serialización y deserialización, y sobrecarga en la red. En lugar de las llamadas API programáticas, es decir, realizar una llamada a un método dentro de la misma aplicación, se tiene que realizar llamadas HTTP. Esto puede ser incluso más problemático si sus solicitudes resultan en múltiples solicitudes posteriores a otros servicios. Sin embargo, existen enfoques que pueden aplicarse para reducir peticiones en cascada y mejorar los tiempos de respuesta (redundancia de datos en una base de datos de servicios y sincronización de datos mediante mensajería o sondeo de alimentación).

- **Consistencia Eventual**

Este es un problema de usabilidad, y es casi seguro que se debe a los peligros de la consistencia final, es decir, que si se realizó una solicitud de actualización y esta fue recibida por el nodo rosado, pero su solicitud fue

manejada por el nodo verde, hasta que el nodo verde reciba la actualización de color rosa, este quedará atrapado en una ventana de inconsistencia. Eventualmente será consistente, pero hasta entonces el usuario se está preguntando si algo ha ido mal.

A continuación otro ejemplo del problema de Consistencia Eventual:

- Se hace una petición para eliminar un profesor en un microservicio *Profesor*.
- Se elimina el profesor en *Profesor* y se envía un mensaje para eliminar las instancias del id del profesor en un microservicio *Materia*.
- Al no tener certeza de cuándo se eliminará dicho id podría llevar a problemas como que se haga una solicitud de los profesores de una materia antes de ser borrado el id, y al intentar buscar dicho profesor en *Profesor* no se pueda encontrar.

Inconsistencias como esta son bastante irritantes, pero pueden ser mucho más graves. La lógica de negocios puede terminar tomando decisiones sobre información inconsistente, en caso que esto suceda, puede ser extremadamente difícil diagnosticar lo que salió mal porque cualquier investigación ocurrirá mucho después de que la ventana de inconsistencia se haya cerrado.

Los microservicios introducen problemas de consistencia eventuales debido a su insistencia elogiable en la gestión descentralizada de datos. Con un monolito, puede actualizar muchos registros en una sola transacción. Microservicios requieren múltiples recursos para actualizar, y las transacciones distribuidas son mal vistas (por una buena razón). Así que ahora, los desarrolladores deben ser conscientes de los problemas de consistencia, y averiguar cómo detectar cuando las cosas están fuera de sincronización.[11]

- **Complejidad operacional**

Ser capaz de desplegar rápidamente pequeñas unidades independientes es una gran bendición para el desarrollo, pero pone una presión adicional sobre las operaciones ya que media docena de aplicaciones ahora se convierten en cientos de pequeños microservicios. Muchas organizaciones encontrarán que la dificultad de manejar tal enjambre de herramientas que cambian rápidamente es sumamente complicado.

Esto refuerza el importante papel de la entrega continua. Mientras que la entrega continua es una habilidad valiosa para los monolitos, uno que es casi siempre vale la pena el esfuerzo para conseguir, se convierte en esencial para una configuración de microservicios. Simplemente no hay manera de manejar docenas de servicios sin la automatización y la colaboración que fomenta la entrega continua. La complejidad operacional también se incrementa debido a

las mayores demandas en la gestión de estos servicios y el monitoreo. De nuevo, un nivel de madurez que es útil para aplicaciones monolíticas se hace necesario si los existen microservicios en el sistema.[11]

- **Incremento en el uso de recursos**

  La arquitectura de microservicios reemplaza instancias de aplicación N monolíticas con instancias de servicios NxM. Si cada servicio se ejecuta en su propia JVM (o equivalente), que suele ser necesario aislar las instancias, entonces hay la sobrecarga de M veces como muchos tiempos de ejecución JVM. Además, si cada servicio se ejecuta en su propia VM (por ejemplo, instancia de EC2), como es el caso en Netflix, la sobrecarga es incluso más alta.[12]

Backend For Frontend

Este concepto es más una solución a un problema en específico que un concepto en sí, al trabajar con microservicios existen casos en los que llamar a un solo servicio web no es suficiente para obtener toda la información necesaria, por ejemplo si existe un servicio para el manejo de Inventario y otro para el manejo de Productos, pero se quiere desplegar en la interfaz gráfica la información de los Productos con su Inventario.

Para este tipo de situaciones en las que la interfaz gráfica debe hacer peticiones a más de un servicio y dependiendo de lo que este retorne solicitar cierto fragmento de información, se puede realizar un *Backend for Frontend,* este *backend* recibirá las peticiones que requieran de información compuesta e irá a cada Microservicio que sea necesario para retornar la información completa.

Otro uso para los *Backend For Frontend* es cuando se poseen distintos tipos de clientes(móviles, web..) en estos casos la información que se espera recibir en cada uno de estos no suele ser la misma, por lo que una solución es crear un *backend* especializado para cada tipo de cliente que se desee soportar y este será el encargado de solicitar y manipular la información que sea requerida.

Al agregar una capa extra al sistema se corre el riesgo que ésta comience a tener lógica que no le corresponda, por lo que se debe tener mucho cuidado en no agregar lógica de negocio en los *Backend For Frontend*, estos solo deben encargarse de entregar la información de cierta manera para una experiencia de usuario particular, no de realizar lógica de negocio que debería encontrarse en los Microservicios.[13]

CAPÍTULO 2: HERRAMIENTAS UTILIZADAS

Eureka

Eureka es un servidor basado en el protocolo REST desarrollado por Netflix, utilizado para el registro y localización de microservicios, balanceo de carga y tolerancia a fallos. La función de Eureka es registrar la información de las diferentes instancias de microservicios existentes del ecosistema.[14]

Para poder registrar dicha información, cada microservicio durante su arranque se comunicará con el servidor Eureka para notificar que está disponible, dónde está situado y sus metadatos. Luego de registrarse, continuarán notificando a Eureka su estado cada 30 segundos (esta notificación se denomina *heartbeats*). Si después de tres periodos Eureka no recibe notificación de dicho microservicio lo eliminará del registro. De la misma forma una vez vueltos a recibir tres notificaciones considerará el servicio disponible de nuevo.[15]

Los clientes (microservicios registrados) de Eureka podrán recuperar la información de otros microservicios registrados para así conocer su localización y poder comunicarse con ellos. Dicha información de registro se *cachea* en el cliente. Además Eureka se puede configurar para funcionar en modo clúster donde varias instancias *peer* intercambiarán su información. Esto, junto al *cacheo* de la información de registro en los clientes da a Eureka una alta tolerancia a fallos.

Ventajas de utilizar Eureka

- Abstracción de la localización física de los microservicios: cualquier microservicio que sea un cliente Eureka sólo necesita conocer el identificador del microservicio al que desea invocar y Eureka resolverá su localización.

  Esto significa que no importa en qué ambiente esté desplegado el servicio (desarrollo, producción, prueba, etc.), Eureka devolverá al cliente la localización de los microservicios del ambiente especificado por este en la configuración del mismo. Es decir, si un desarrollador está probando localmente una funcionalidad y necesita conectarse con un microservicio desplegado en el ambiente local, Eureka resolverá dicho requisito; así mismo, si el desarrollador (desde un ambiente local) desea conectarse al mismo microservicio pero desplegado en ambiente de producción, basta con indicarle a Eureka de qué ambiente se desea obtener la información del microservicio y se encargará de devolver la localización y puertos de este.[15]

- Conocer el estado de nuestro ecosistema de microservicios: Eureka proporciona un panel de control que permite ver los microservicios existentes actualmente en el registro.[15]

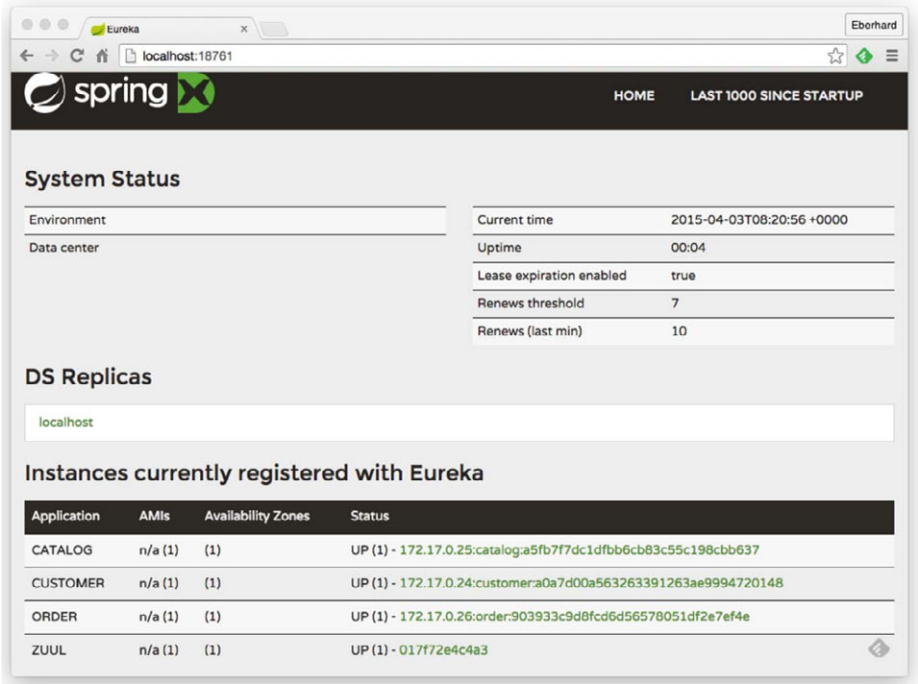


Figura 4
Panel de control de Eureka
Fuente:http://meuslivros.github.io/microservices-flexible-software-architectures/text/part0019.html

- Se puede configurar como clúster incrementando notablemente su tolerancia a fallos. Esto quiere decir que en caso de que un servidor de Eureka no esté disponible, los clientes podrán comunicarse con otro de los servidores para obtener la información deseada.

- Eureka proporciona soporte a multiregión, pudiendo definir diferentes agrupaciones de microservicios. En la figura 4 se explica cómo puede ser desplegado un clúster de Eureka en una región. Existe un clúster por región y este sólo sabe la información de los registros que se encuentren en las instancias desplegadas en esta región.

  Los servicios se registran en una de las instancias del clúster y dicha información es replicada a todos los nodos del mismo. De esta forma se mejora la tolerancia a fallos puesto que, en caso de que uno de los nodos no esté disponible, un cliente puede comunicarse con cualquiera de los nodos ya que la información de registro de todos los clientes está replicada en todos los nodos.[15]

En la Figura 5 se puede observar la arquitectura de multiregión desplegada por Netflix.

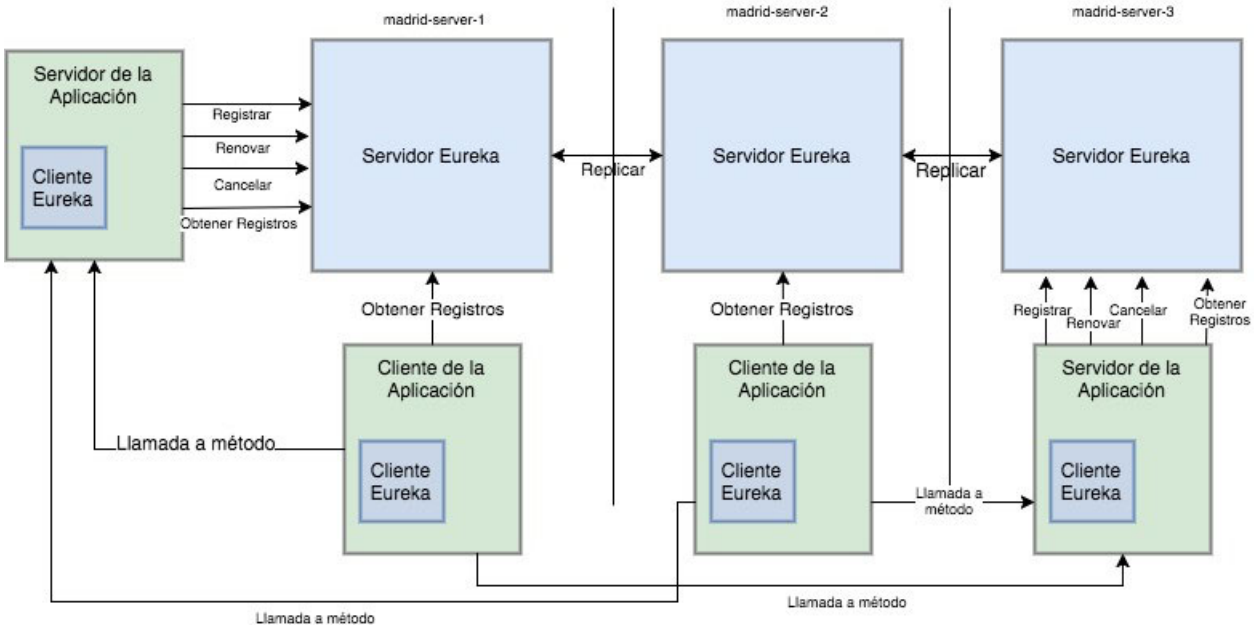


Figura 5
Arquitectura multiregión desplegada por Netflix
Fuente: Documentación Eureka
https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Zuul

Zuul es la puerta principal para todas las solicitudes de dispositivos y sitios web al *backend* de la aplicación de *streaming*. Siendo un componente expuesto al público, Zuul está construido para permitir enrutamiento dinámico, monitoreo, resiliencia y seguridad.

El enrutamiento es una parte integral de una arquitectura de microservicio. Por ejemplo, */api/users* se asigna al servicio de usuario y /api/shop se asigna al servicio de tienda. Zuul es un enrutador basado en JVM y equilibrador de carga del lado del servidor de Netflix.

El volumen y la diversidad del tráfico de la API de Netflix a veces dan lugar a problemas en el ambiente de producción que surgen rápidamente y sin previo aviso. Necesitamos un sistema que nos permita cambiar rápidamente el comportamiento para reaccionar ante estas situaciones.

Zuul utiliza una gama de diferentes tipos de filtros que nos permite aplicar rápida y ágilmente funcionalidad a nuestro servicio de borde. Estos filtros nos ayudan a realizar las siguientes funciones:[16]

- **Autenticación y seguridad**: identificación de requisitos de autenticación para cada recurso.
- ***Insights* y monitoreo**: seguimiento de datos y estadísticas significativos.

- **Enrutamiento dinámico**: enrutamiento dinámico de las peticiones a los diferentes *backend*.
- **Pruebas de estrés:** aumentar gradualmente el tráfico.
- **Desconexión de carga**: asignación de capacidad para cada tipo de solicitud y solicitud de descarte.
- **Manejo de respuestas estáticas:** construyendo algunas respuestas directamente.
- **Resiliencia multiregión**: solicitudes de enrutamiento en las regiones AWS.

Zuul contiene múltiples componentes:

- **Zuul-core:** librería que contiene la funcionalidad principal de compilar y ejecutar filtros.
- **Zuul-simple-webapp**: aplicación web que muestra un ejemplo simple de cómo construir una aplicación con zuul-core.
- **Zuul-netflix**: librería que añade otros componentes de NetflixOSS a Zuul - por ejemplo, utilizando Ribbon[17] para las solicitudes de enrutamiento.
- **Zuul-netflix-webapp**: aplicación web que agrupa zuul-core y zuul-netflix en un paquete fácil de usar.

Protocolo REST

La **Transferencia de Estado Representacional** (en inglés *Representational State Transfer*) o **REST** es un estilo de arquitectura software para sistemas hipermedia distribuidos como la *World Wide Web*, también conocido como servicios web *RESTful*, estos permiten realizar solicitudes a sistemas exponen estos servicios, para acceder y manipular un recurso web usando un conjunto de acciones predefinidas y que no poseen estado.[18]

MySQL

MySQL es un sistema de gestión de base de datos relacional desarrollado por Oracle Corporation y está considerada como la base de datos de código abierto más popular del mundo[19], y es una de las más populares en general junto a Oracle y Microsoft SQL Server, sobre todo para desarrollos web.

Spring

Creado por Rod Jhonson en 2003 bajo la licencia Apache 2.0, Spring es el *framework* más popular para el desarrollo de aplicaciones basado en Java puesto que permite crear aplicaciones de alto rendimiento, fáciles de probar y con código reutilizable. Si bien las características fundamentales de Spring *Framework* pueden ser usadas en cualquier aplicación desarrollada en Java, existen variadas extensiones para la construcción de aplicaciones web sobre la plataforma Java EE.[20]

Está organizado por módulos independientes y puede integrarse con otros *frameworks.*

- Spring *Framework*: Apoyo básico para la inyección de dependencias, gestión de transacciones, aplicaciones web, de acceso a datos, mensajería, pruebas.
- Spring Data: Proporciona un modelo de programación amigable para el acceso de datos. Este es un proyecto paraguas que contiene muchos sub-proyectos que son específicos de una base de datos dada (relacional, no relacional, en la nube).
- Spring Boot: Simplifica la creación de aplicaciones de Spring embebiendo los servicios subyacentes en la aplicación (servidor web, motor BD...).

Node.Js

Node.Js es una plataforma de desarrollo construida sobre la máquina virtual de Javascript V8 desarrollada por Google. Mientras que los motores de Javascript tradicionalmente son ejecutados en el navegador del lado del cliente, las librerías de Node.Js están enfocadas a construir aplicaciones del lado del servidor en Javascript.

Node.Js es un entorno en tiempo de ejecución multiplataforma asíncrono orientado a eventos. Permite manejar múltiples conexiones simultáneamente, y luego que cada petición concluye, Node entra en modo descanso hasta recibir la próxima petición.

El hecho que Node.Js está diseñado sin el manejo de hilos no implica que no se puedan aprovechar los múltiples núcleos de su entorno. Los procesos hijos se pueden generar mediante el uso del método *child_process.fork()* y están diseñados para ser fáciles de comunicarse con estos métodos. Construido sobre esa misma interfaz está el módulo de clúster, que le permite compartir sockets entre procesos para permitir balanceo de carga sobre sus núcleos.[21]

CAPÍTULO 3: IMPLEMENTACIÓN

En primera instancia se seleccionó un dominio para realizar este Trabajo de fin de Máster, siendo el de Gestión Universitaria el elegido, pero, siendo tan extenso se limitó a tres subdominios:

- Gestión de Profesores.
- Gestión de Materias.
- Gestión de Nómina.

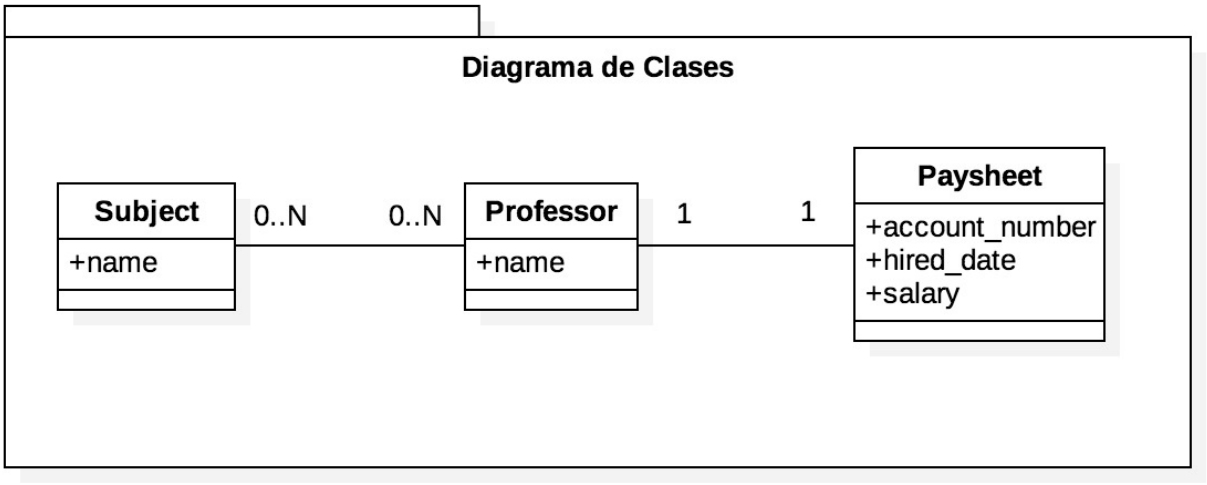


Figura 7
Diagrama de Clases del dominio
Fuente: Elaboración propia

Teniendo en cuenta este dominio, se comenzó el desarrollo de este trabajo basándose en un Arquitectura Monolítica, luego se fue iterando sobre esta solución y se llegó a una Arquitectura final basada en Microservicios.

La arquitectura que se propone es sólo del lado del servidor. Aunque esta puede ser utilizada para el desarrollo de la interfaz gráfica, para efectos de este trabajo sólo se realizará el desarrollo del servidor. Como se mencionó anteriormente se utilizará Spring *Framework* para el desarrollo de la aplicación, MySQL como manejador de base de datos y Eureka[14].

Caso Base: Arquitectura Monolítica

Como se dijo anteriormente el desarrollo de esta solución se inició con una Arquitectura Monolítica, es decir, los tres (03) subdominios seleccionados se encuentran dentro de una misma aplicación.

La aplicación fue estructurada de la siguiente manera: existe un módulo por cada subdominio y dentro de cada módulo se encuentran los componentes necesarios para implementar las funcionalidades requeridas.

Cada módulo se divide en:

- Service
- Entity
- REST
- Repository
- Exception
- Converter

A continuación se puede ver la estructura que se siguió para crear este caso base.

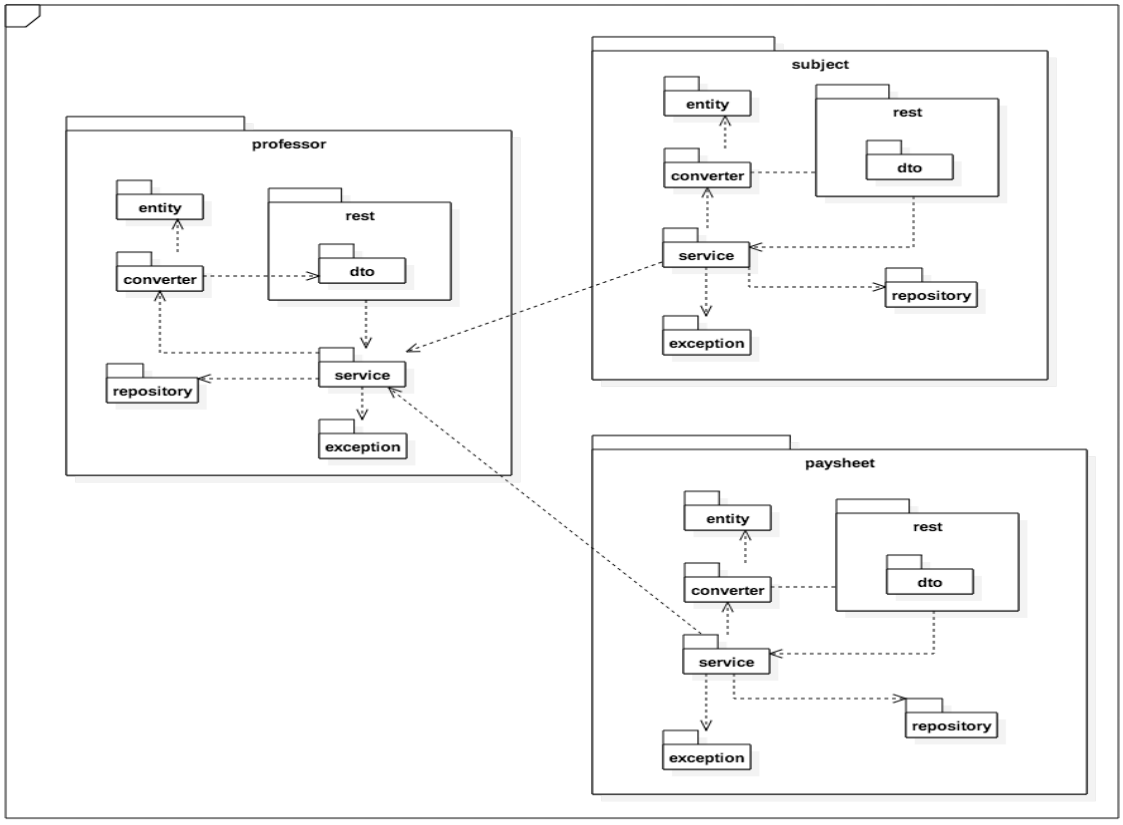


Figura 8
Diagrama de paquetes de la aplicación Monolítica
Fuente: Elaboración propia

En la figura 8 se puede observar cómo cada módulo posee un paquete REST que a su vez contiene un paquete DTO, en este paquete se agrupan las clases que sirven como la representación de la data que se espera recibir en una solicitud vía un servicio web o la respuesta que este debe proveer. Cada una de estas clases reciben el nombre de DTO.

La base de datos que se plantea para esta primera fase, posee las siguientes tablas:

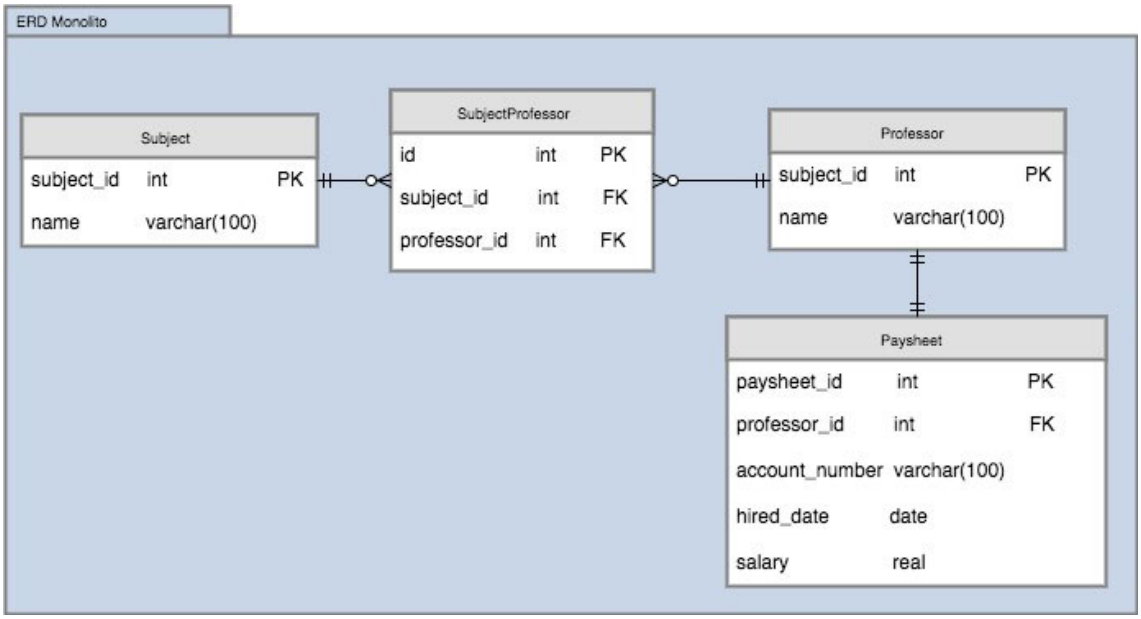


Figura 9
Diagrama ERD de la aplicación Monolítica
Fuente: Elaboración propia

Para este caso base se plantea una arquitectura de despliegue que está conformada por: un servidor web que albergará la aplicación, un servidor de base de datos y el cliente que se desee comunicar con la aplicación, en la Figura 10 se puede apreciar dicha distribución.

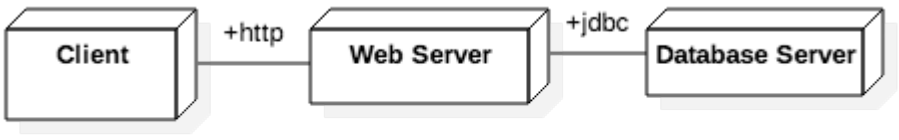


Figura 10
Diagrama de despliegue de la aplicación Monolítica
Fuente: Elaboración propia

Desventajas de esta Arquitectura

- **Mantenibilidad**

  Un sistema de gran escala como es la gestión Universitaria suele tener una base de código de gran tamaño, esto afecta directamente a la dificultad de entender el proyecto, de realizar nuevas modificaciones o de corregir errores. Además, se debe resaltar que al no haber una separación de servicios estricta, la modularidad suele perderse con el tiempo y esto también incrementa la dificultad de entender cómo se debería implementar correctamente una funcionalidad nueva.

- **Despliegue Continuo**

  Ya que una aplicación monolítica necesita ser desplegada en su totalidad cada vez que se quiera publicar una nueva funcionalidad, o algún cambio de las actuales, sin importar el (o los) componente que se haya visto afectado, los riesgos que se corren al realizar un despliegue son mucho más altos por esta misma razón. Esto suele ser un problema por ejemplo para el desarrollo de interfaz de usuario, que suelen necesitar cambios frecuentes.

- **Escalabilidad**

  Tomando en cuenta el dominio utilizado se puede presentar el caso en que un módulo es más demandado que otro, por ejemplo el módulo de nómina, por lo que lo más conveniente sería aumentar la capacidad del servidor para ese módulo en particular, pero al tener todo el sistema en una sola aplicación esto no es posible (como se puede apreciar en la Figura 10), por lo que se tendría que aumentar la capacidad de servidor para toda la aplicación o crear una nueva instancia de esta, en ambos casos existe un desperdicio de recursos.

Iteración 1: Primera separación en Microservicios

Luego de desarrollar una aplicación utilizando una arquitectura monolítica, ahora pasamos a realizar una división en microservicios.

La primera división que se realizó fue separar los tres subdominios en un microservicio cada uno, quedando así el diagrama ER:

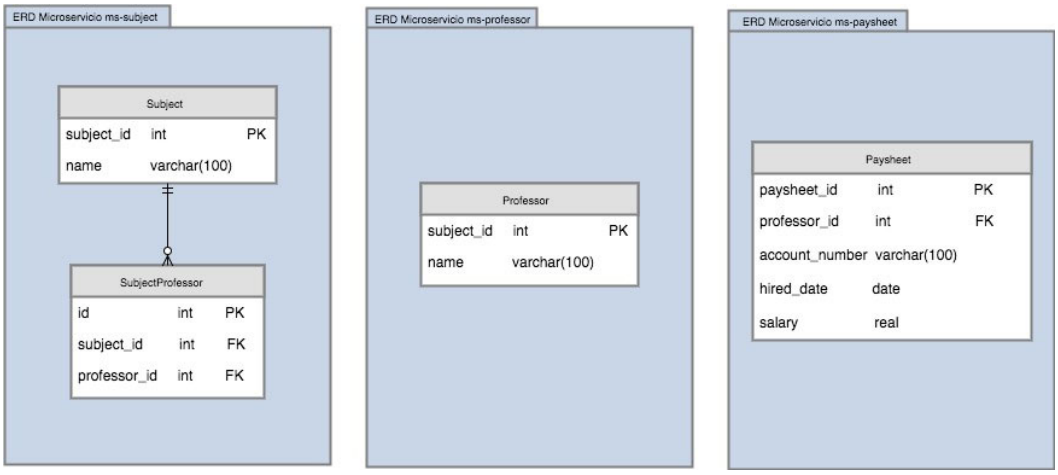


Figura 11
Diagrama de Base de Datos para esta primera división en Microservicios
Fuente: Elaboración propia

La primera división que se realizó fue separar los tres subdominios en un Microservicio cada uno, es decir, que cada uno de los microservicios tendrá una estructura de paquetes similar a la que vimos en el punto anterior, a continuación se muestra cada estructura:

- Microservicio de Nóminas (*ms-paysheet*):

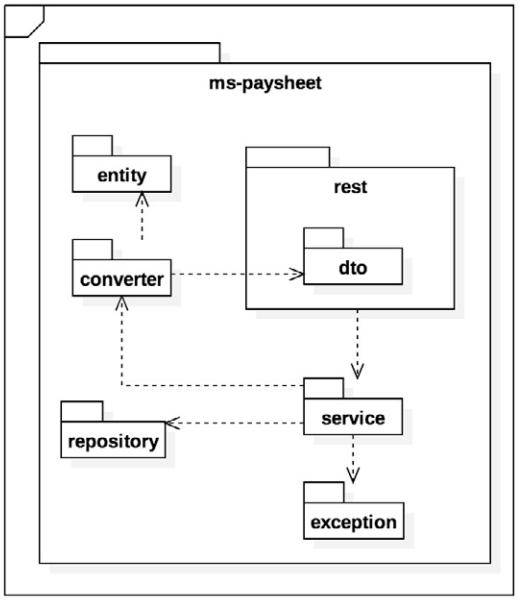


Figura 12
Fuente: Elaboración propia

- Microservicio de Profesores (*ms-professor*):

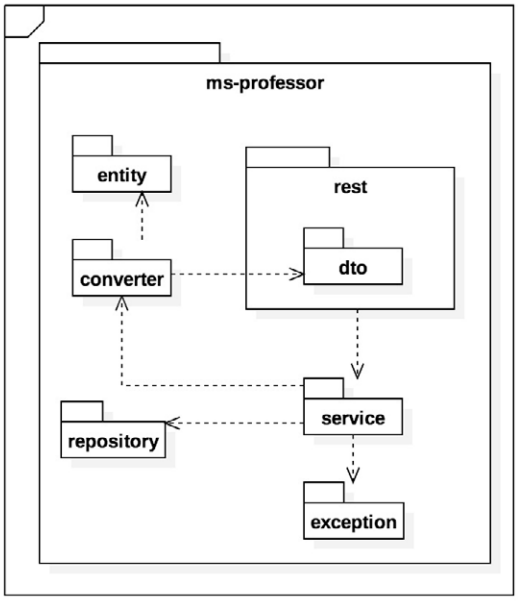


Figura 13
Fuente: Elaboración propia

- Microservicio de Materias (*ms-subject*):

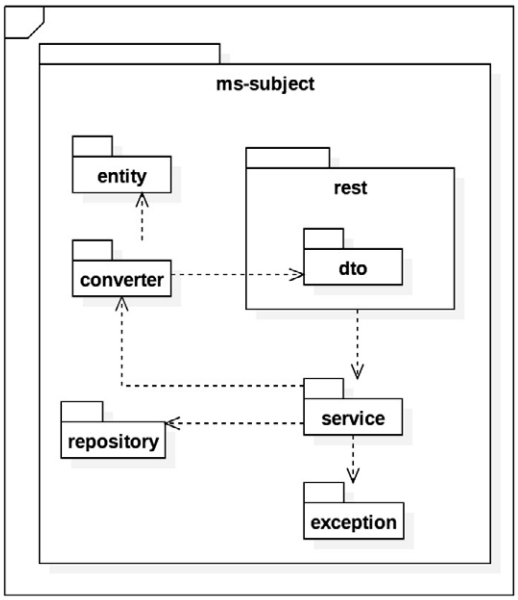


Figura 14
Fuente: Elaboración propia

En las Figuras 12, 13 y 14 se puede observar cómo cada microservicio contiene la estructura de paquetes correspondiente al módulo que va a ser utilizado como el dominio pertinente.

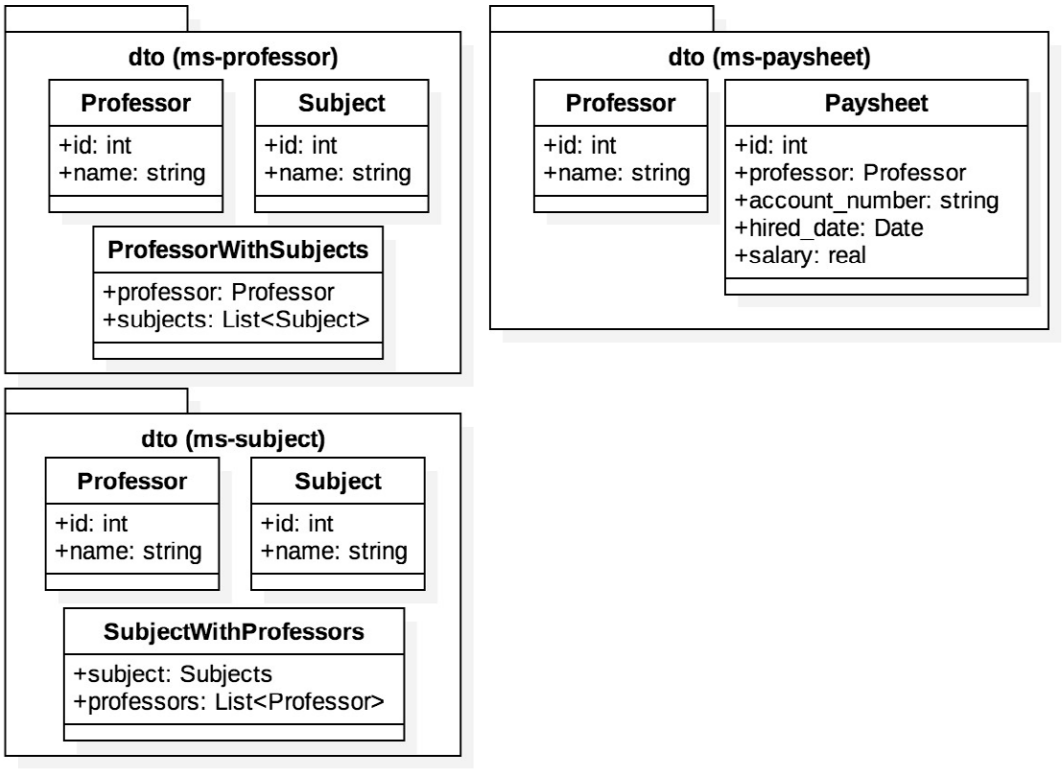


Figura 15
Especificación de clases pertenecientes a cada paquete DTO de cada microservicio
Fuente: Elaboración propia

En la Figura 15 se encuentra el detalle de las clases que posee el paquete DTO de cada Microservicio, estos contienen los DTOs correspondientes a su dominio. Sin embargo, como puede observarse en la imagen, en esta solución se tuvieron que agregar las distintas clases ajenas a su dominio necesarias para poder devolver las respuestas que contengan información anidada. Esto ocurre en:

- *ms-subject*: cuando se solicita la lista de profesores para una materia, se necesita el DTO de *Professor* para poder agregar dicha información en la respuesta.
- *ms-professor*: cuando se solicita la lista de materias para un profesor, se necesita el DTO de *Subject* para poder agregar dicha información en la respuesta.
- *ms-paysheet*: cuando se solicita la nómina de un profesor, se requiere el DTO de *Professor*.

Cada microservicio se desplegará de manera individual. Esto se puede llevar a cabo de distintas maneras: desplegando cada uno en un servidor distinto; ejecutarlos todos o varios en un mismo servidor (al ser aplicaciones distintas cada una estaría siendo

ejecutada en un proceso distinto); o podrían ser ejecutados como contenedores de Docker en un mismo servidor o en múltiples servidores.

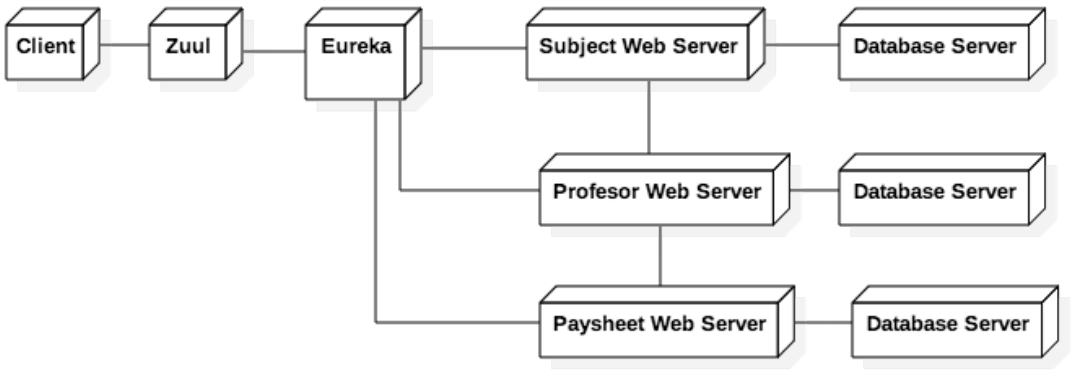


Figura 16
Diagrama de despliegue para esta división en Microservicios
Fuente: Elaboración propia

Como se puede observar en la Figura 16 cada microservicio es una aplicación independiente donde cada uno posee su propia base de datos, así se garantiza que cada uno sea dueño de su información y la pueda tratar de la manera en que sea necesaria. Además se desplegó un servidor de Zuul que se encargará del redireccionamiento de las peticiones de parte del cliente y, un servidor de Eureka que se encargará de la localización de los servicios para realizar un balanceo de carga e intercomunicar las diferentes instancias de microservicios.

Ventajas de este enfoque respecto al monolítico

Esta solución solventa los problemas que fueron encontrados en la propuesta anterior:

- La escalabilidad vertical ya no es un inconveniente puesto que se puede asignar recursos según convenga al microservicio que posea una mayor demanda por parte de los usuarios. Por otro lado, también se solventa el problema de la escalabilidad horizontal ya que si es necesario crear nuevas instancias, en lugar de incrementar las capacidades del servidor, se puede hacer sin problema, como se puede observar en la Figura 17.

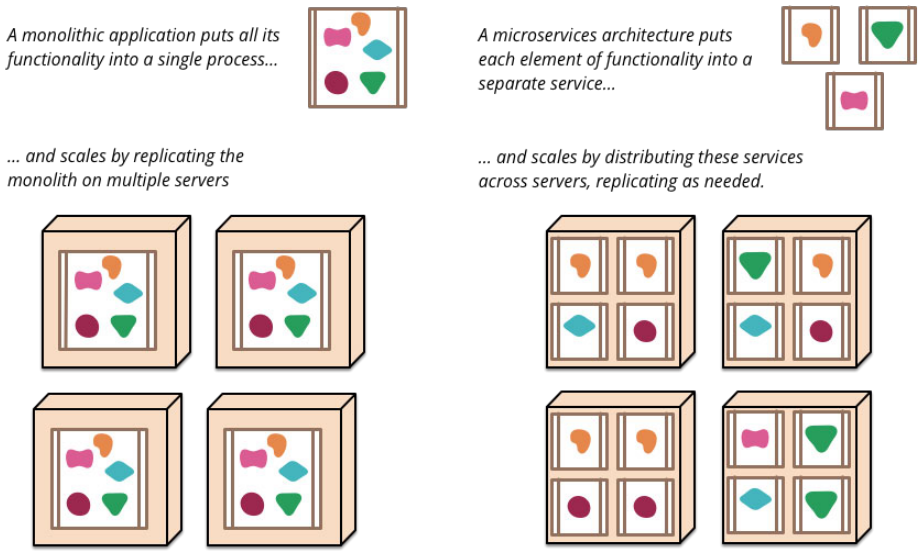


Figura 17
Comparación de escalabilidad horizontal
Fuente: https://martinfowler.com/articles/microservices.html

- Los riesgos que se corrían al momento de realizar Despliegue Continuo se reducen en gran medida, ya que no es necesario hacer un redespliegue completo de la aplicación por cualquier cambio. En este caso de estudio, si *ms-professor* fue modificado, solamente es necesario construir y desplegar este microservicio mientras que los demás no tendrían que pasar por estos procesos.
- La mantenibilidad por microservicio se ve mejorada, ya que un equipo sólo se debe preocupar por mantener y mejorar un microservicio, en vez de tener que conocer y manipular una aplicación entera.
- Además se debe agregar que una de las ventajas de tener una arquitectura de este estilo es que se puede seleccionar la tecnología que mejor se adapte a las necesidades de cada microservicio. Por ejemplo, si para *ms-paysheet* fuese necesario generar reportes sumamente pesados de procesar, se puede investigar cuál tecnología se adapta mejor a este requerimiento e implementar este servicio de esta manera. Mientras que para *ms-subject* capaz no es necesario algo tan elaborado y se puede utilizar un *framework* más sencillo, ésta es una gran ventaja que nos provee utilizar Microservicios.

Problemas encontrados a resolver

- División de los microservicios: Uno de los mayores problemas al dividir un monolito o comenzar un sistema utilizando Microservicios, es identificar que tan granular debe ser cada servicio, es decir, que tan pequeño debe ser un Microservicio. Cada uno debe encargarse de un conjunto no muy grande de

responsabilidades, lo suficientemente grande para no ser un simple servicio de CRUD, pero sin perder la alta cohesión entre sus responsabilidades.

En este caso se ha hecho una división excesiva, ya que es totalmente innecesario crear un microservicio por entidad en el sistema, si se plantea esto a un sistema que pueda tener 40 entidades, se estaría hablando de una arquitectura de 40 microservicios sin mucha lógica de negocio en cada uno.

- Comunicación entre Microservicios: es realizada a través de llamadas HTTP, esto trae 2 consecuencias principales:
    - Si se está realizando una petición que requiere información de más de un Microservicio bien sea al momento de realizar la solicitud o al momento de obtener la respuesta, se agrega una pequeña latencia que en la solución anterior no existía ya que toda la información necesaria se encontraba en la misma aplicación, pero ahora es necesario solicitar la data faltante al microservicio responsable. Para este tipo de problemas se puede implementar la comunicación entre servicios vía un enrutador de mensajes como se mencionó anteriormente.
    - Como tanto *ms-subject* como el de *ms-paysheet* tienen el DTO de *Professor*, cualquier cambio en este "contrato" en el *ms-professor* tendría tendría que ser replicado en los antes mencionados, ya que cada microservicio tendría que actualizar esta clase para mantener el contrato, lo cual también implicaría realizar un nuevo despliegue. Esto rompe con el concepto de bajo acoplamiento, alta cohesión.

      Aunado a esto, esta actualización, además de no debería ser responsabilidad de estos microservicios por estar fuera de su dominio, en caso de que no haya una comunicación efectiva entre los equipos de desarrollo y no se lleven a cabo los cambios pertinentes, ocurrirán fallos innecesarios.

      Este problema también lo encontramos con el DTO de *Subject* que se encuentra en *ms-professor*.
- Por último, sin duda alguna hay un incremento en el uso de recursos, aunque mejor aprovechados esto no quita el hecho que en vez de correr por ejemplo una máquina virtual de Java (JVM) se estén utilizando N según la cantidad de microservicios que estén implementados en Java.

Iteración 2: Refactorización de los Microservicios

En esta iteración se buscó solventar los problemas que se consiguieron luego de hacer el primer intento de separación de la aplicación en Microservicios.

El primer problema atacado fue el de la extrema granularidad de cada microservicio. La idea al separar por Dominio es agrupar los conceptos similares o relacionados para que cada microservicio tenga una alta cohesión, sin embargo esto no implica separar cada tabla de la base de datos en un microservicio específico como se hizo en la primera iteración. Por esto se procedió a unir las funcionalidades de los microservicios *ms-professor* y *ms-paysheet*, puesto que ambos conceptos están muy relacionados entre si (una nómina no puede existir sin un profesor al que esté asignada), en un nuevo microservicio denominado *ms-employee.*

En la Figura 18 se puede observar la estructura resultante al unir los funcionalidades de los *ms-professor* y ms-*paysheet* en el microservicio *ms-employee.*

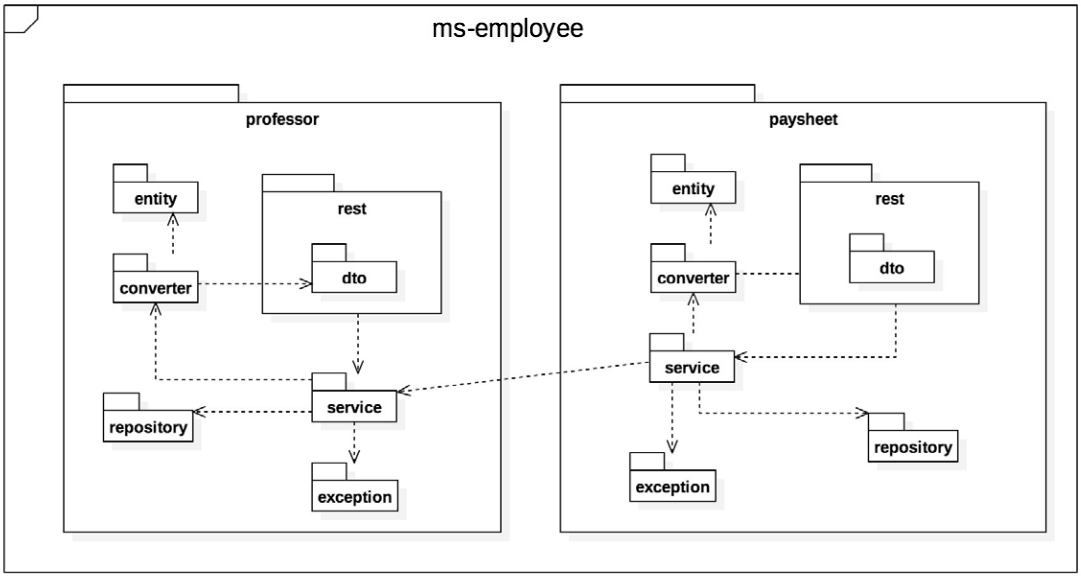


Figura 18
Diagrama de paquetes del microservicio *ms-employee*
Fuente: Elaboración propia

Un efecto colateral de unir *ms-professor* y *ms-paysheet* es que ya no se tendrá el problema del acoplamiento innecesario que se tenía al tener el DTO de *Professor* replicado en *ms-paysheet*.

Así mismo, el diagrama ER de la base de datos resultante se puede observar en la Figura 19.

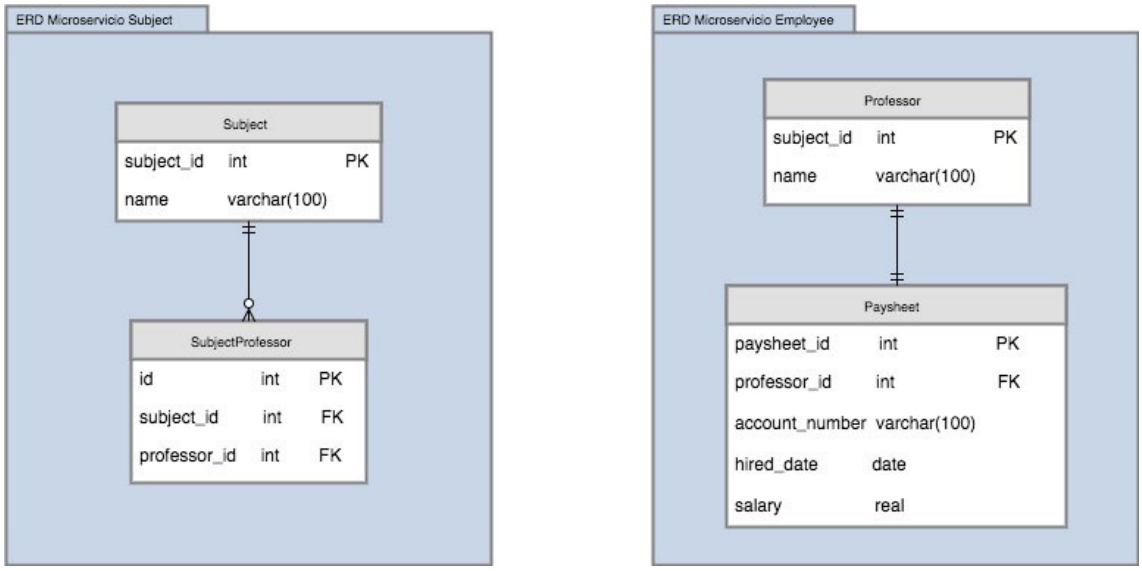


Figura 19
ERD final
Fuente: Elaboración propia

El diagrama de despliegue (a falta de la última modificación que se expone más adelante) quedaría como está expuesto en la Figura 20.

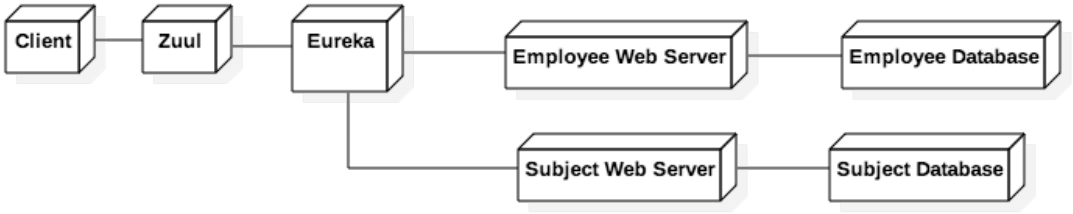


Figura 20
Diagrama de despliegue de los microservicios
Fuente: Elaboración propia

El siguiente problema resuelto fue el del alto acoplamiento entre los microservicios *ms-subject* y *ms-employee* derivado de la decisión de tener el DTO de *Professor* en *ms-subject* para poder devolver la lista de materias con sus profesores, así como la decisión de que ms-e*mployee* (*ms-professor*, en la iteración anterior) tuviera el DTO de la clase *Subject* para devolver lista de materias para un profesor.

En primer lugar se procedió a remover el DTO de *Professor* de *ms-subject* y el DTO de *Subject* del microservicio *ms-employee*, quedando la distribución de los DTOs de la siguiente forma:

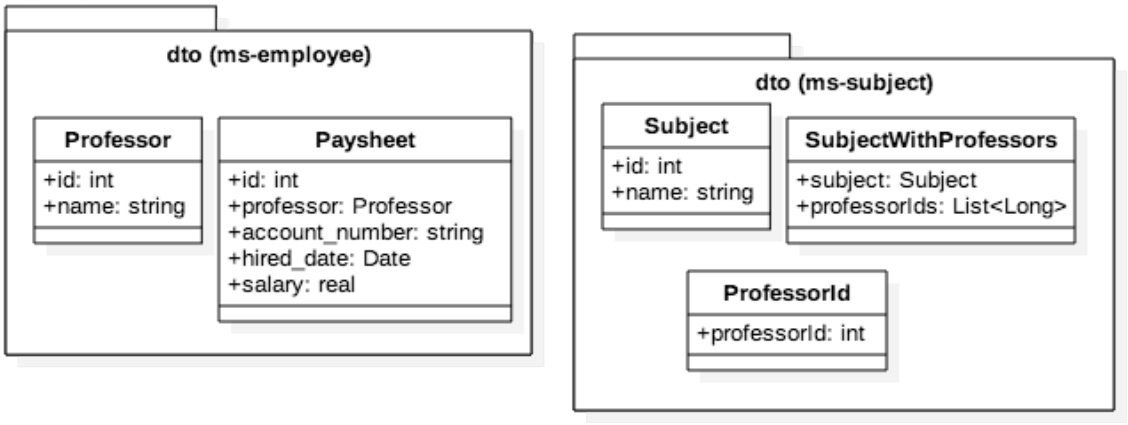


Figura 21
Especificación de clases pertenecientes a cada paquete DTO de cada microservicio
Fuente: Elaboración propia

Luego, se siguió el patrón de *Backends For Frontends* y se creó un tercer microservicio denominado *ms-composite*, que tendrá una responsabilidad:

- Componer los *json* solicitados en las llamadas *GET* que requieren obtener objetos de distintos microservicios y transformarlos para devolver la respuesta esperada por el cliente.

El diagrama de despliegue resultante es el expuesto en la Figura 22.

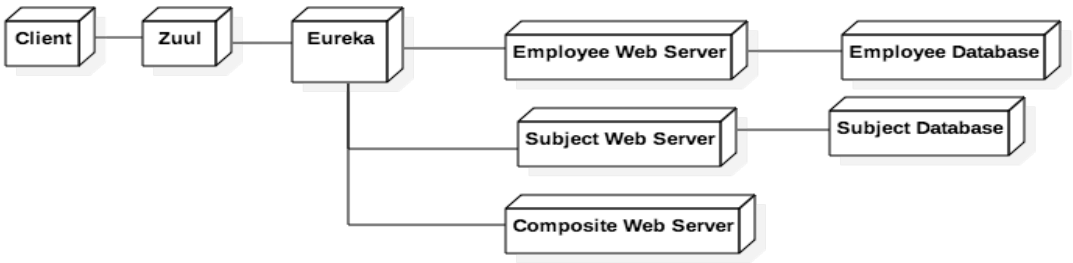


Figura 22
Diagrama de despliegue final de la arquitectura
Fuente: Elaboración propia

Sin embargo, este nuevo microservicio no fue implementado utilizando el mismo *framework* que en los anteriores puesto que, a pesar de que Spring permite el manejo de objetos del tipo *JsonNode* con lo cual se podrían manipular las respuestas o solicitudes sin necesidad de tener un DTO definido, existen herramientas con las que se puede hacer esta composición de manera más sencilla.

Por esto se decidió implementar este microservicio en *Node.Js* ya que no es necesario conocer detalladamente la estructura interna de los *json* devueltos por cada microservicio, el *ms-composite* sólo debe obtenerlos y armar un nuevo *json* que será retornado al cliente. El *ms-composite* es capaz de recibir un objeto de clase A del

microservicio X y un objeto de clase B del microservicio Y sin importar la estructura interna de cada objeto, para luego componer un nuevo *json* con la información deseada de cada uno, ya sea transformando parte de la información, obteniendo atributos específicos o combinando los dos objetos directamente.

Para mostrar más claramente el punto anterior se puede observar el procedimiento que realiza el *ms-composite* para obtener una materia con los profesores que la imparten:

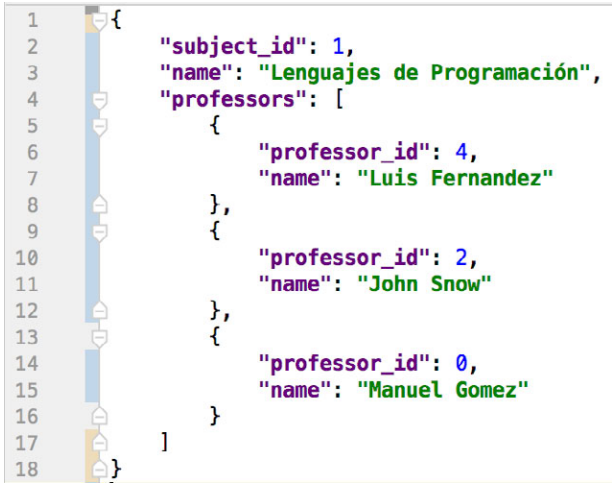
1. El cliente solicita la materia con sus respectivos profesores al *ms-composite*.
2. *ms-composite* realiza una llamada al microservicio *Subject* para obtener la información de le información completa da la materia más la lista de *ids* de los profesores asociados a ella (Figura 23).

Figura 23
Json obtenido del microservicio *Subject* con información de la materia y lista de profesores
Fuente: Elaboración propia

3. El *ms-composite* solicita al microservicio *ms-employee* la lista de profesores correspondientes a los *ids* obtenidos en la llamada anterior (Figura 24).

Figura 24
Json con la lista con la información de los profesores solicitados
Fuente: Elaboración propia

4. Por último, *ms-composite* crea un nuevo objeto con la información de la materia y la información completa de todos los profesores el cual será devuelto al cliente (Figura 25).



```
{
    "subject_id": 1,
    "name": "Lenguajes de Programación",
    "professors": [
        {
            "professor_id": 4,
            "name": "Luis Fernandez"
        },
        {
            "professor_id": 2,
            "name": "John Snow"
        },
        {
            "professor_id": 0,
            "name": "Manuel Gomez"
        }
    ]
}
```

Figura 25

Json construido a partir del *json* obtenido del microservicio *ms-subject* y de los *jsons* con la información de los profesores obtenidos del microservicio *ms-employee*

Fuente: Elaboración propia

Como se puede observar, de esta manera se rompe con el acoplamiento que existía entre los dos microservicios lo cual nos permite evitar hacer un *build* nuevo del ms-s*ubject* en caso de que se modifique el DTO de *Professor* en el *ms-employee,* de la misma forma que también se evita hacer un *build* nuevo de *ms-employee* cuando cambie el DTO de *Subject*.

Problemas encontrados

A pesar de que en esta iteración se lograron solventar varios de los problemas identificados en la iteración pasada, aún se presentan inconvenientes inherentes a la arquitectura basada en microservicios. Para empezar, los problemas de la latencia entre la comunicación de microservicios y del uso de recursos (ambos explicados en la iteración anterior). Sumado a esto se presentan se evidenciaron los siguientes problemas:

- **Comunicación entre los Microservicios**

  A diferencia de una aplicación hecha en una Arquitectura Monolítica en la que todo el sistema esa interconectado, los distintos microservicios no tienen una conexión directa entre si, por lo que los desarrolladores tienen que invertir tiempo extra de desarrollo diseñando e implementando la vía de comunicación entre estos. Estas vías de comunicación pueden ser realizada a través de llamadas RESTful o de colas de mensaje.
- **Comunicación entre los equipos de desarrollo**

  Los desarrolladores deben invertir una cantidad de tiempo considerable en establecer los contratos de comunicación entre los microservicios. Además, a pesar de que en el presente proyecto no se presentó dicho problema por ser de un alcance tan corto, si el proyecto crece y existen

numerosos microservicios desarrollados por varios equipos de desarrolladores, se deben crear mecanismos de comunicación lo suficientemente efectivos y eficientes para transmitir todos los cambios realizados a los diversos grupos.

BIBLIOGRAFÍA

1. *Monolithic Application.* Recurso Web. Disponible en: https://www.wikiwand.com/en/Monolithic_application

2. *Monolithic Architecture*. Recurso Web. Disponible en: http://whatis.techtarget.com/definition/monolithic-architecture

3. *Monolithic vs. Microservices Architecture*. Recurso Web. Disponible en: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

4. *Pattern: Monolithic Architecture.* Recurso Web. Disponible en: http://microservices.io/patterns/monolithic.html

5. W*hat is Domain Driven Design?*. Recurso Web. Disponible en: http://dddcommunity.org/learning-ddd/what_is_ddd/

6. *Microservices*. Recurso Web. Disponible en: https://martinfowler.com/articles/microservices.html

7. Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95

8. *Pattern: Decompose by business capability*. Recurso Web. Disponible en: http://microservices.io/patterns/decomposition/decompose-by-business-capability.html

9. *Pattern: Decompose by subdomain*. Recurso Web. Disponible en: http://microservices.io/patterns/decomposition/decompose-by-subdomain.html

10. Ubiquitous Language. Recurso Web. Disponible en: https://martinfowler.com/bliki/UbiquitousLanguage.html

11. *Microservice Trade-Offs*. Recurso Web. Disponible en: https://martinfowler.com/articles/microservice-trade-offs.html

12. *Pattern: Microservice Architecture*. Recurso Web. Disponible en: http://microservices.io/patterns/microservices.html

13. Newman, Sam (2015). Building Microservices. O'Reilly. p.71-72

14. *Eureka at a Glance*. Recurso Web. Disponible en: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

15. Microservicios Spring Cloud: Arquitectura. Recurso Web. Disponible en: https://www.paradigmadigital.com/dev/quien-es-quien-en-la-arquitectura-de-microservicios-spring-cloud-12/

16. *Zuul.* Recurso Web. Disponible en: https://github.com/Netflix/zuul/wiki

17. *Ribbon*. Recurso Web. Disponible en: https://github.com/Netflix/ribbon/wiki

18. *Web Services Architecture*. Recurso Web. Disponible en: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest

19. *Oracle MySQL*. Recurso Web. Disponible en: https://www.oracle.com/mysql/index.html

20. Jesús Bernal. Transparencias utilizadas en el curso Back-end con Tecnologías de Código Abierto del Máster de Ingeniería Web. Universidad Politécnica de Madrid, Madrid, España(2017).

21. About Node.Js. Recurso Web. Disponible en: https://nodejs.org/en/about/

CONCLUSIONES Y POSIBLES AMPLIACIONES

En el presente informe se realizó un trabajo de investigación sobre la Arquitectura Monolítica y la Arquitectura Basada en Microservicios. Inicialmente se presentaron las características de cada una, sus ventajas y desventajas teóricas, y estudiaron las distintas formas de dividir un monolito en microservicios. Posteriormente se procedió poner en práctica dicha teoría implementando en primera instancia una aplicación para la gestión de materias, profesores, y nóminas de la universidad basándose en la Arquitectura Monolítica, para luego pasar a realizar la división de esta en tres microservicios (*ms-subject*, *ms-paysheet*, y *ms-professor*). Por último, luego de analizar los problemas que se encontraron al realizar la división anterior, se procedió a refinarla, dando como resultado dos microservicios (*ms-subject* y *ms-employee)*.

Al finalizar la implementación se pudieron observar más claramente las ventajas y desventajas de la Arquitectura Basada en Microservicios. Entre las ventajas observadas podemos mencionar que existen fronteras más firmes entre cada módulo por el hecho de estar en aplicaciones separadas; se pueden realizar despliegues independientes por cada microservicio; cada microservicio, al ser dueño de su propio dominio, puede ser implementarse de la manera más conveniente para satisfacer sus necesidades, cada uno tiene total control sobre las decisiones de implementación propias (como en qué lenguaje o *framework* será implementado o que manejador de base de datos se adapta mejor a sus necesidades); así como existe mayor facilidad para escalar tanto horizontal como verticalmente cada microservicio. Por otro lado, entre las desventajas se evidenció que hay una mayor latencia por la comunicación entre los microservicios, existe el problema de la consistencia eventual, hay un incremento en la complejidad operacional causado por el incremento de aplicaciones que se tienen que desplegar y monitorear a diferencia de la arquitectura monolítica en la que únicamente se debe manejar una aplicación, lo cual también conlleva a un incremento en el uso de recursos.

En base a todo esto se puede decir que no existe una regla de oro al momento de tomar la decisión de qué tipo de arquitectura implementar (monolítica o basada en microservicios), sino que se deben tomar ciertas consideraciones respecto a los pro y los contra de dicha arquitectura. Tomando esto en consideración, se puede recomendar el uso de microservicios en los siguientes casos:

- Cuando hay partes de la aplicación que sean más propensas a tener un alto consumo de recursos por lo que sería necesario escalarlas individualmente, ya sea horizontal o verticalmente.
- Cuando el dominio es muy extenso o es una aplicación muy grande por lo que se tendrán muchos equipos de trabajo enfocados en desarrollar los módulos.

- Cuando el proyecto es de largo recorrido ya que se asegura que la productividad perdida al inicio del desarrollo, se verá compensada por el aumento de productividad al tener a cada equipo de desarrollo trabajando en una aplicación independiente.

Posibles Ampliaciones

Respecto a la teoría expuesta en el presente informe se podría ahondar en cómo realizar las pruebas de las funcionalidades que impliquen comunicación entre microservicios, así como cómo puede ser el manejo de temas de seguridad.

Respecto a la aplicación final se recomiendan las siguiente ampliaciones:

- Ampliar el dominio del sistema.
- Incluir pruebas.
- Incluir seguridad como el manejo de sesiones y permisos.
- Mejorar el manejo de excepciones.