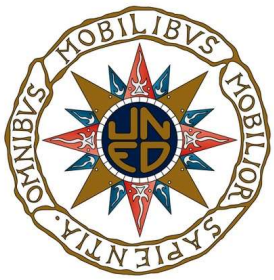
TRABAJO FIN DE MÁSTER

Master Universitario de Investigración en Ingeniería de Software y Sistemas Informáticos

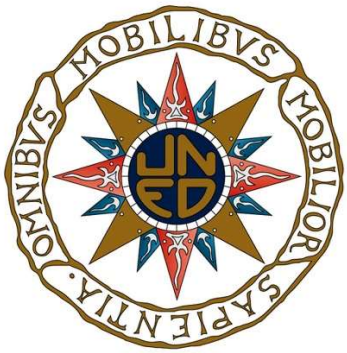
Itinerario de Ingeniería de Software
(Cod. 31105151)

Curso 2016/17
(Convocatoria Septiembre del 2017)

Gestor de transacciones distribuidas asíncronas en arquitecturas de microservicios



Autor: Francisco Cilleruelo Trotter
Director: José Félix Estívariz López



Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Itinerario de Ingeniería de Software

(Cod. 31105151)

Gestor de transacciones distribuidas asíncronas en arquitecturas de microservicios

Autor: Francisco Cilleruelo Trotter

Director: José Félix Estívariz López

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 04/09/2017

Quién suscribe:

Autor(a): D. Francisco Cilleruelo Trotter
D.N.I/N.I.E/Pasaporte.: 50877251-D

Hace constar que es el autor(a) del trabajo:

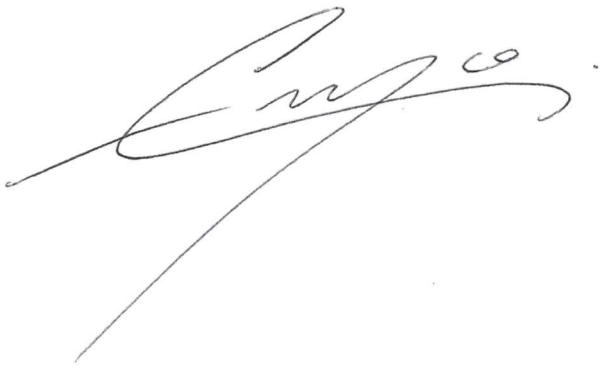
Gestor de transacciones distribuidas asíncronas en arquitecturas de microservicios

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.



Impreso TFdM05_Autor. Autorización de publicación

y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09

www.issi.uned.es

Resumen

El presente trabajo de fin de Máster, como parte del plan de estudios del Máster en Ingeniería del Software y Sistemas Informáticos impartido por la Universidad Nacional de Educación a Distancia, se ha desarrollado con el objetivo fundamental de implementar un gestor de transacciones que solucione los principales problemas planteados en la comunicación que se establece entre los servicios que componen una arquitectura de microservicios, que por su propia naturaleza son distribuidos, y que además se comunican de manera asíncrona a través de mensajes mediante AMQP (Advanced Messaging Queuing Protocol). Como resultado se ha conseguido desarrollar una librería en Java reutilizable, extensible y fácilmente integrable, tanto en el componente emisor como en el componente receptor que intervienen en este tipo de comunicación asíncrona, dentro de un marco tecnológico software que responda a este tipo de arquitectura de servicios distribuidos. A esta librería la hemos nombrado por sus siglas en inglés: ADTM (Asynchronous Distributed Transaction Manager).

Palabras clave

Arquitecturas software, Microservicios, Comunicación asíncrona, Mensajería, AMQP, Sistemas distribuidos, Servicios, Transacciones, ACID, Atomicidad, IPC, Arquitectura dirigida por eventos

Introducción

En los últimos años, y como una evolución natural del desarrollo software, ha surgido con fuerza un nuevo estilo de arquitectura software, la arquitectura de microservicios. Una arquitectura en constante evolución y con un futuro ciertamente muy prometedor, en la que hoy en día muchas empresas y profesionales ya están apostando como una nueva manera de concebir y desarrollar software de calidad.

Esta nueva arquitectura, aunque está lejos de ser la solución definitiva al desarrollo software, tal y como pueden llegar a pensar muchas personas erróneamente, sí es cierto que ofrece numerosas ventajas y virtudes frente al desarrollo más tradicional de aplicaciones monolíticas, que la convierten en una grande candidata para el desarrollo de nuevas aplicaciones o migrar de las existentes. Algunas de estas ventajas son la facilidad de escalabilidad, el desacoplamiento entre los servicios, el desarrollo y despliegue de los microservicios de manera independiente, mayor resiliencia, desarrollo dirigido por dominio, convivencia de distintas tecnologías, mayor facilidad de mantenimiento…, entre otras muchas.

A la hora de implementar una arquitectura de este tipo, en la medida que es un modelo de arquitectura conceptual definido por un conjunto de características propias, son muchos los patrones específicos que se pueden seguir de acuerdo con los requisitos que se tengan que cumplir. Estos patrones abarcan perspectivas tan variadas como la forma de despliegue, la descomposición en servicios, la gestión de datos, la seguridad o la manera de implementar el interfaz de usuario. Una de estas perspectivas que se debe tener en cuenta es la forma en la que los microservicios se comunican entre sí. Básicamente, si lo hacen de manera síncrona, por ejemplo, mediante servicios REST, o de manera asíncrona, por ejemplo, mediante mensajes a través de algún gestor de mensajería AMQP.

En el contexto de una arquitectura de microservicios en el que los servicios se comunican de manera asíncrona surgen, tal y como veremos más adelante, una serie de problemas e inconvenientes. Como son, por ejemplo, la transaccionalidad o la atomicidad en el envío de los mensajes al gestor de mensajería. Partiendo de esta situación, vamos a intentar desarrollar una solución software que pueda ser integrada de manera sencilla en los microservicios involucrados en la comunicación y, de esa manera, dar una respuesta eficiente a la problemática planteada.

Para alcanzar este objetivo, vamos a distinguir tres partes fundamentales en nuestro proyecto. En primer lugar, y como contexto tecnológico inicial necesario para entender la problemática y elaborar la solución, realizar un estudio lo suficientemente amplio y profundo de la arquitectura software de microservicios, como una arquitectura en auge relativamente reciente y, presumiblemente, con un futuro muy prometedor. En segundo lugar, dentro del marco tecnológico de este tipo de arquitectura software, exponer mediante un ejemplo práctico en Java los problemas e inconvenientes planteados en una implementación donde la comunicación entre los microservicios no sólo es distribuida por la propia naturaleza de la arquitectura, sino que además también es asíncrona. Y, por último, desde este punto de partida, analizar, desarrollar e implementar una solución software en Java que pueda integrarse de una manera sencilla con cualquier arquitectura de microservicios que responda a este modelo de comunicación, y resuelva de una manera eficiente los problemas expuestos.

Microservicios

Los microservicios son una arquitectura y una propuesta para el desarrollo de software con la finalidad de satisfacer las necesidades de los negocios actuales. Los microservicios no son un invento; son más bien una evolución de estilos de arquitectura software previos y más tradicionales.

La arquitectura de microservicios es uno de los patrones de arquitectura software actuales con mayor auge y popularidad, complementado con DevOps y la Cloud. La evolución de los microservicios se ha visto enormemente influenciada por las tendencias de innovación tecnológica en los negocios modernos y la evolución de las tecnologías de los últimos años.

En esta época de transformación digital, las empresas adoptan de manera creciente las tecnologías como una de las claves fundamentales para incrementar considerablemente sus clientes y sus beneficios. Las empresas utilizan las redes sociales, las aplicaciones móviles, la Cloud, Big Data o Internet de las cosas (IoT) como una manera de alcanzar la innovación. Mediante el uso de estas tecnologías las empresas encuentran nuevas maneras de entrar rápidamente en el mercado, que de manera constante plantea retos a los mecanismos tecnológicos tradicionales.

En contraposición, frente a este nuevo tipo de arquitectura software distribuida, coexiste de manera paralela el modelo de arquitectura tradicional monolítica. En el que, los componentes que integran el sistema, independientemente de la organización o interacción que exista entre ellos, se presentan encapsulados dentro del propio sistema, delimitados por éste y como parte de una misma entidad funcional. Y, por lo tanto, su comportamiento se circunscribe únicamente al contexto de la aplicación que los contiene. Tal y como veremos, existen grandes diferencias entre estos dos tipos de arquitecturas, y con ello numerosas ventajas e inconvenientes que se deben tener en cuenta según los requisitos particulares del ámbito en el que se quieran aplicar.

El siguiente gráfico muestra el estado de los microservicios y el desarrollo tradicional frente a los nuevos retos de las empresas como agilidad, velocidad de entrega y escalado.

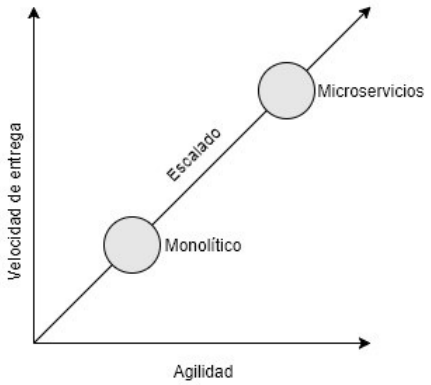


Ilustración 1 - Comparativa de sistemas monolíticos y de microservicios respecto de la velocidad de entrega y agilidad de desarrollo

Ya han pasado aquellos días en los que las empresas invertían en el desarrollo de grandes aplicaciones con una duración de años. Las empresas ya no están interesadas en desarrollar aplicaciones consolidadas para gestionar sus funciones de negocio como hacían hace años.

El siguiente gráfico muestra el estado de las aplicaciones monolíticas tradicionales y los microservicios en comparación con el tiempo y el coste exigidos.

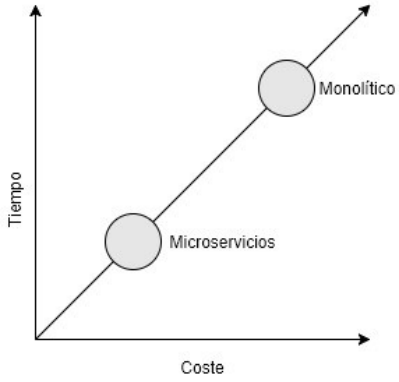


Ilustración 2 - Comparativa de sistemas monolíticos y de microservicios respecto del tiempo y coste

Hoy, por ejemplo, las compañías aéreas o las instituciones financieras no invierten en reconstruir sus sistemas principales como tradicionales y enormes aplicaciones monolíticas. El foco se ha desplazado en construir soluciones rápidas y eficientes con las que alcanzar sus objetivos para conseguir sus necesidades inmediatas de la forma más ágil posible.

Las tecnologías emergentes también nos han hecho repensar la manera en la que construimos los sistemas software. Por ejemplo, hace unas pocas décadas, no podíamos ni tan siquiera imaginar una aplicación distribuida sin un commit en dos fases. Mas tardes, las bases de datos NoSQL nos han hecho pensar de una manera muy diferente.

Similarmente, este tipo de cambios de paradigma en la tecnología han transformado todas las capas de la arquitectura software.

La aparición de HTML5 y CSS3 y el avance de las aplicaciones móviles cambiaron las interfaces gráficas de usuario. Los frameworks de JavaScript en el lado del cliente como Angular, Ember, React, Backbone y muchos más, se han hecho enormemente populares debido a sus diseños adaptables y ajustables al cliente.

Con la introducción de la Cloud como parte de las principales tecnologías, las proveedoras de plataformas como servicio (PaaS) como, por ejemplo, Pivotal CF, AWS, Salesforce, IBMs Bluemix, RedHat OpenShift y muchas otras, nos han hecho replantearnos la manera de construir componentes middleware. La revolución de los contenedores creada por Docker influyó radicalmente en la infraestructura. En la actualidad, una infraestructura es vista como un servicio de producto.

Por otro lado, las bases de datos NoSQL han revolucionado el ámbito de las bases de datos. Hace unos pocos años, disponíamos sólo de unos pocas bases de datos populares, todas ellas basadas en los principios de los modelos de datos relacionales. Mientras que hoy en día disponemos de una larga lista de bases de datos: Hadoop, Cassandra, CouchDB y Neo 4j, son sólo unas pocas. Donde cada una de estas bases de datos están enfocadas a resolver un determinado problema.

La arquitectura de aplicaciones siempre ha evolucionado de acuerdo con los exigentes requisitos de los negocios y la evolución de las tecnologías. Las arquitecturas han ido a través de la evolución desde antiguos y desfasados sistemas, hasta servicios en la nube completamente abstractos, como AWS Lambda.

Diferentes propuestas y estilos de arquitectura como la arquitectura cliente-servidor, la arquitectura en N capas o la arquitectura orientada a servicios, han sido muy populares en distintos momentos de la historia del desarrollo software. Pero, independientemente de la elección del estilo de arquitectura, siempre la aplicábamos a arquitecturas monolíticas. La arquitectura de microservicios evolucionó como resultado de las demandas del mercado, como son la agilidad y velocidad de respuesta, tecnologías emergentes, y el aprendizaje de arquitecturas previas.

Los microservicios nos ayudan a romper las barreras de las aplicaciones monolíticas y construir un sistema de sistemas más pequeños lógicamente independientes. Si consideramos las aplicaciones monolíticas como un conjunto de subsistemas lógicos coordinados con una barrera física, los microservicios son un conjunto de subsistemas independientes sin ninguna barrera física.

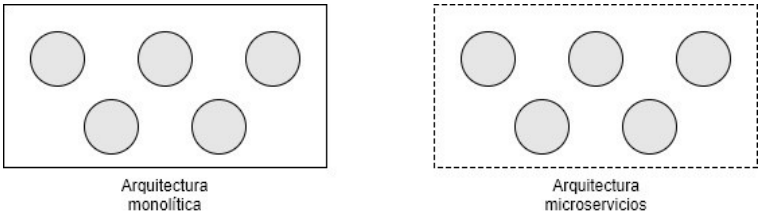


Ilustración 3 - Delimitación de las arquitecturas monolíticas frente a las arquitecturas de microservicios

Los microservicios son un estilo de arquitectura usado en la actualidad por muchas compañías como una manera de alcanzar un alto grado de agilidad, velocidad de respuesta y escalado. Los microservicios nos ofrecen la posibilidad de desarrollar aplicaciones modulares más separadas físicamente.

Pero los microservicios no son un invento. Muchas compañías como Netflix, Amazon y Ebay ya utilizaron con éxito la técnica de "divide y vencerás" para separar funcionalmente sus aplicaciones monolíticas en unidades atómicas más pequeñas, donde cada una de ellas se hace cargo de una funcionalidad particular. De esta manera, estas empresas resolvieron un gran número de problemas que sufrían con sus aplicaciones monolíticas. Siguiendo el éxito de estas organizaciones, muchas otras organizaciones empezaron a adoptar este patrón común para refactorizar sus aplicaciones monolíticas. Más tarde, este patrón derivo en lo que es hoy la arquitectura de microservicios. Los microservicios surgieron de la idea de la arquitectura hexagonal acuñada por Alistair Cockburn. Esta arquitectura hexagonal también es conocida como el patrón de los puertos y los adaptadores. Son un estilo de arquitectura o una propuesta para construir sistemas IT como un conjunto de responsabilidades o capacidades del negocio autónomos, auto contenidos y débilmente acoplados.

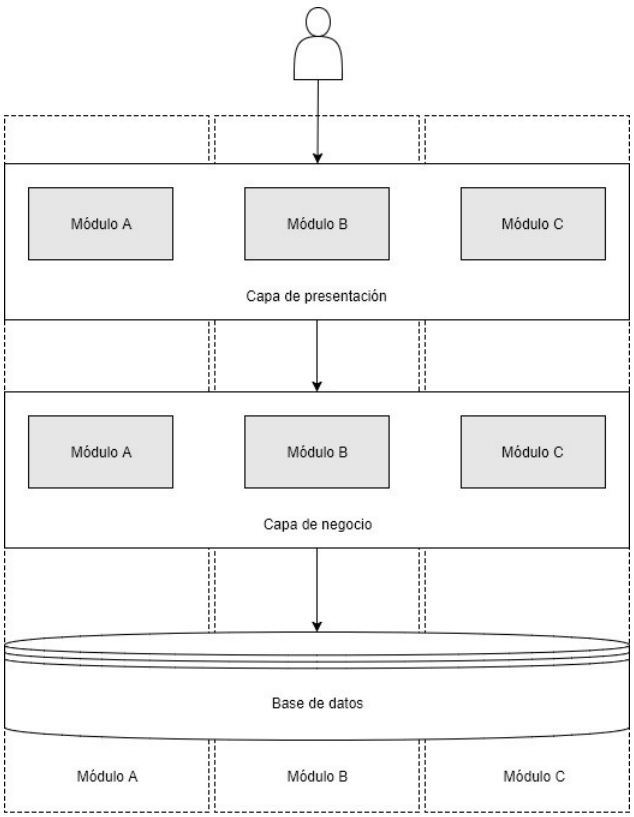


Ilustración 4 - Arquitectura tradicional en N capas

La imagen anterior muestra una arquitectura de aplicación tradicional en N capas con una capa de presentación, una capa de lógica de negocio y una capa de base de datos. Los módulos A, B y C representan los tres ámbitos del negocio. Las capas del diagrama representan una separación de los distintos aspectos o intereses de la arquitectura. Y cada capa contiene los tres ámbitos del negocio pertenecientes a esa capa. La capa de presentación tiene los componentes web de los tres módulos, la capa de lógica de negocio tiene los componentes de negocio de los tres módulos, y la capa de la base de datos aloja la estructura de base de datos de esos mismos tres módulos. En la mayoría de los casos, las capas son físicamente disgregables, mientras que los módulos dentro de la misma capa están fuertemente acoplados. Examinemos esa misma arquitectura como una arquitectura basada en microservicios.

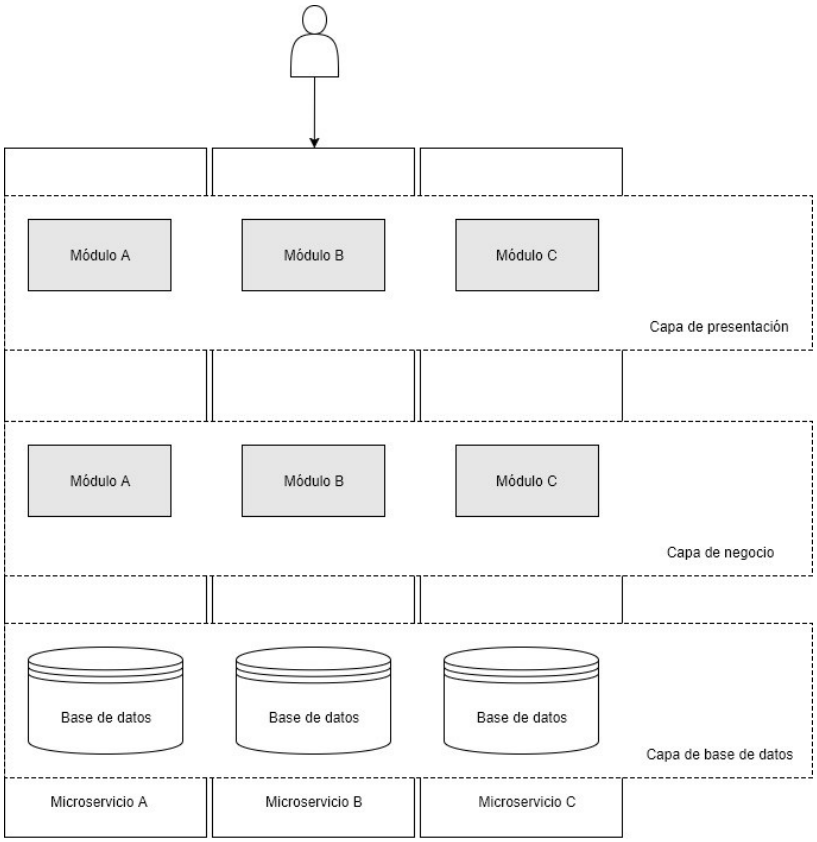


Ilustración 5 - Arquitectura basada en microservicios

Como podemos observar en esta imagen, los límites en la arquitectura de microservicios se han invertido. Cada bloque vertical representa un microservicio. Cada microservicio tiene su propia capa de presentación, capa de lógica de negocio y capa de base de datos. De esta manera, los microservicios están alineados hacía las capacidades del negocio. Haciendo esto, las modificaciones en un microservicio no afectan a los demás.

No hay un estándar en los mecanismos de comunicación o transporte para los microservicios. En general, los microservicios se comunican entre ellos utilizando protocolos ligeros ampliamente conocidos, como HTTP y REST, o protocolos de mensajería, como JMS o AMQP. En ciertos casos específicos uno podría optar por protocolos de comunicación más optimizados, como Thrift, ZeroMQ, Protocol Buffers o Avro.

Puesto que los microservicios están enfocados a las capacidades del negocio y tienen ciclos de vida gestionables de manera independiente, son una elección ideal para empresas que opten por DevOps y la Cloud, que no dejan de ser dos aspectos de los microservicios.

El panal es una analogía ideal para representar el desarrollo de la arquitectura de microservicios.

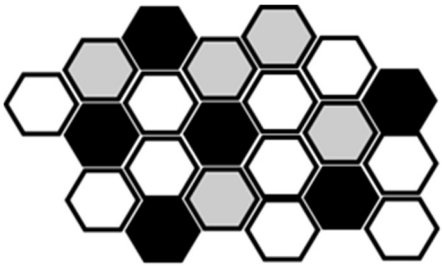


Ilustración 6 - Analogía de la arquitectura de microservicios como un panal de abejas

En el mundo real, las abejas construyen un panal mediante la composición de celdas de cera hexagonales. Empiezan por unas pocas de manera reducida, utilizando diferentes materiales para construir las celdas. La construcción se basa en lo que está disponible en el momento en el que se construye el panal. La repetición de celdas forma un patrón y conforman una estructura fuerte. Cada celda del panal es independiente, pero al mismo tiempo está integrada con el resto de celdas. Añadiendo nuevas celdas, el panal crece progresivamente hasta llegar a ser una gran estructura muy sólida. Y, el contenido de cada celda no es visible desde fuera, ni el daño en una celda daña otras celdas, y en el caso de que pase, las abejas pueden reconstruirlo sin impacto en el conjunto del panal

Principios de los microservicios

Los principios que vamos a exponer son los principios que se deben cumplir en el diseño y desarrollo de microservicios

Una única responsabilidad por microservicio

El principio de responsabilidad única es uno de los principios definidos como parte del patrón de diseño SOLID, que mantiene que una unidad debería tener una sola responsabilidad. En concreto, corresponde con la primera de las siglas del acrónimo.

- S (SRP, Single Responsability Principle). Principio de Responsabilidad Único
- O (OCP, Open/Closed Principle). Principio de Abierto/Cerrado
- L (LSP, Liskov Substitution Principle). Principio de substitución de Liskov
- I (ISP, Interface Segregation Principle). Principio de segregación de la interfaz
- D (DIP, Dependency Inversion Principle). Principio de Inversión de dependencia

Esto implica que una unidad, ya sea una clase, una función, o un servicio, debería tener una sola responsabilidad. Por lo tanto, en ningún caso dos unidades deberían compartir la misma responsabilidad, o una unidad tener más de una responsabilidad.

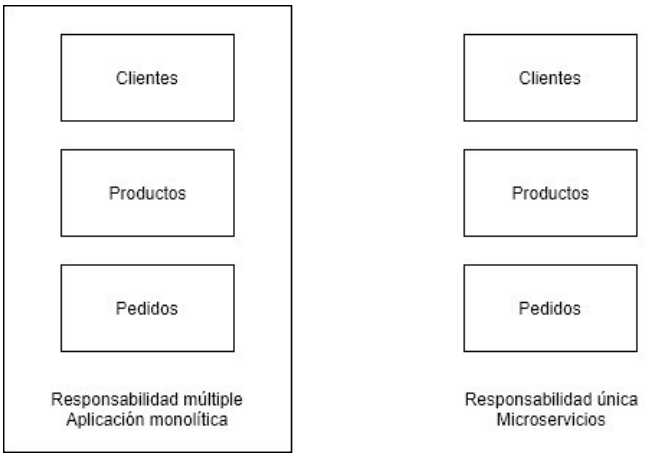


Ilustración 7 - Comparativa de desacoplamiento de responsabilidades entre aplicaciones monolíticas y arquitecturas de microservicios

Tal y como se puede ver en esta imagen, Cliente, Producto y Pedido son diferentes funciones de una aplicación de comercio por Internet. En vez de construir todas ellas dentro de una misma aplicación, es mejor tener tres servicios diferentes, donde cada uno se hace responsable de una función del negocio. De esa manera los cambios en la responsabilidad de uno de ellos no deben afectar a los demás. Cada uno de estos servicios (Cliente, Producto y Pedido) será tratado como un microservicio independiente.

Los microservicios son autónomos

Los microservicios son servicios auto contenidos, desplegables de manera independiente y autónomos, que toman la responsabilidad de un área del negocio y su ejecución. También incluyen todas las dependencias necesarias, incluyendo librerías y entornos de ejecución, como un servidor web y contenedores o máquinas virtuales que abstraen de los recursos físicos.

Una de las mayores diferencias entre microservicios y SOA está en su nivel de autonomía. Mientras que la mayoría de las implementaciones SOA proveen abstracción en el nivel de servicio, los microservicios van más allá y abstraen también del entorno de ejecución.

En el desarrollo de aplicaciones tradicionales, construimos un WAR o un EAR, y lo desplegamos en un servidor de aplicaciones JEE, como JBoss, WebLogic, WebSphere, entre otros. Podemos desplegar varias aplicaciones en el mismo contenedor JEE. En la estrategia seguida por los microservicios, cada microservicio será construido como un JAR que embebe todas las dependencias y se ejecuta como un proceso Java aislado, sin necesidad de ningún contenedor JEE adicional.

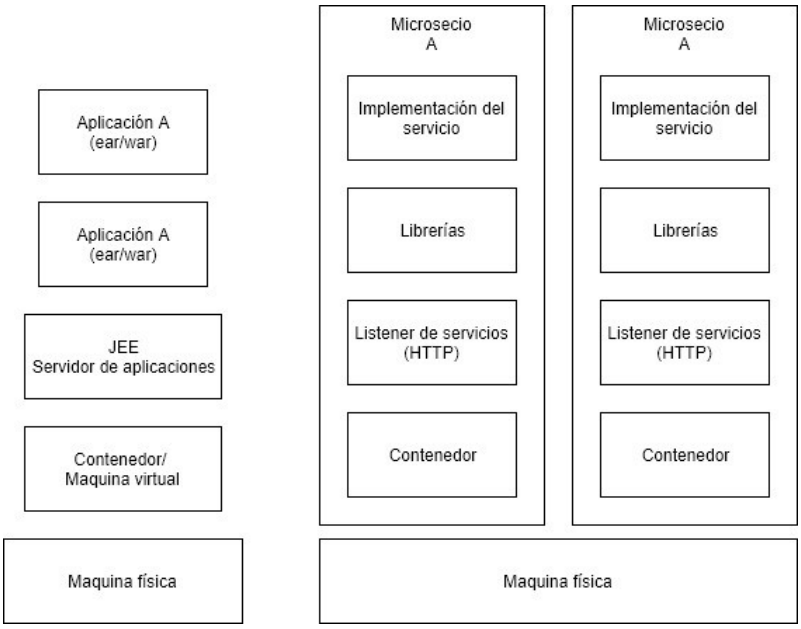


Ilustración 8 - Comparativa de arquitectura física entre las aplicaciones tradicionales y la arquitectura de microservicios

Los microservicios pueden también tener sus propios contenedores donde ser ejecutados, como se muestra en la imagen anterior. Los contenedores son portables, gestionables de manera independiente y con entornos de ejecución ligeros. Tecnologías de contenedores, como Docker, son una elección ideal para el despliegue de microservicios.

Características de los microservicios

No hay una definición única, concreta y universalmente aceptada para los microservicios. Sin embargo, todas las implementaciones de microservicios exitosas presentan una serie de características comunes que son importantes entender.

Componetización a través de servicios

Uno de los principales deseos de la industria del software ha sido poder generar bloques de construcción o "componentes" que permitan ser reutilizados y combinados entre sí para crear sistemas más complejos, de manera similar a lo que hacemos con en ámbitos como la arquitectura.

En este contexto, una buena definición de "componente" sería una unidad de software que es independientemente reemplazable y actualizable.

Así, en una arquitectura de microservicios, la principal manera de ensamblar el software aplicativo es mediante la descomposición en servicios, en lugar de bibliotecas o librerías.

Definimos bibliotecas como componentes que están vinculados a un programa y son llamados mediante llamadas de función en memoria; mientras que los servicios son componentes separados, que se comunican mediante un mecanismo tal como una solicitud de servicio web (web service request), o una llamada a procedimiento remoto (RCP, Remote Procedure Call).

Una de las principales razones del uso de servicios como componentes, en lugar de bibliotecas, es que los servicios son desplegados de manera independientemente. Esto implica que, si una aplicación se descompone en múltiples servicios, ante varios cambios en un servicio, será necesario redesplegar exclusivamente el servicio modificado.

Organizada en torno a las capacidades del negocio

Normalmente, cuando se busca dividir una gran aplicación en partes, las organizaciones se focalizan en hacerlo por capas tecnológicas, o silos, según la especialidad de sus recursos, esto es, algunos equipos se dedican al desarrollo de interfaces de usuario (UI), otros equipos a la lógica aplicativa de servidor (App), y otros equipos a la Base de datos (DB).

Cuando estos equipos están además separados físicamente, incluso cambios simples pueden conducir a cambios que afecten a todos los equipos, y que, por lo tanto, impliquen mayores tiempos y presupuesto. Está de más mencionar que, cuando surge un problema con la solución que se está construyendo, las idas y vueltas entre los equipos, suelen llevar no sólo a la no resolución temprana del problema, sino a distanciar aún más a los equipos.

Esto es un ejemplo claro de la ley de Conway, definida de la siguiente manera:

"Cualquier organización que diseñe un sistema (definido ampliamente), producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización."

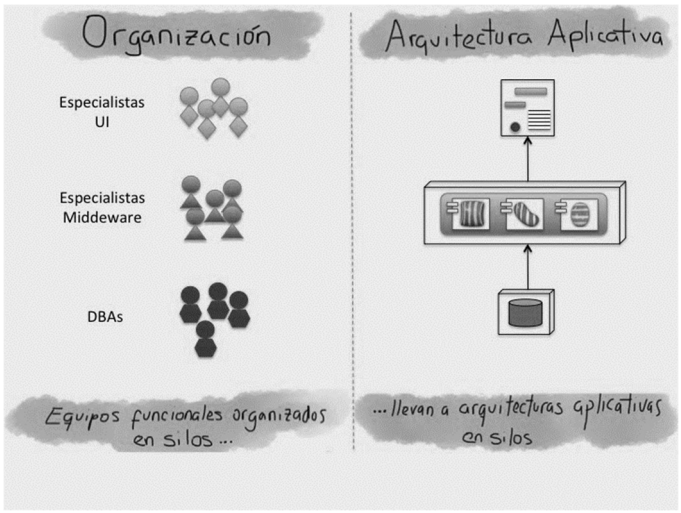


Ilustración 9 - Ley de Conway

Lo que hace referencia este enunciado, tiene más que ver con gente que con tecnología. Es decir, el componente principal de diseño de software es la gente, por lo tanto, el tipo de gente que uno tiene, los procesos que adopte y el tipo de comunicación entre ellos, es lo que determinará el tipo de resultado que se desee alcanzar.

Otra consecuencia de la ley de Conway es que un sistema realizado para una organización difícilmente encajará de manera correcta en una organización diferente, a menos claro, que la organización cambie y se adapte al nuevo sistema, lo que también tiene sus costos, un ejemplo de ello son algunos productos, como por ejemplo SAP.

Sin embargo, en el estilo de arquitectura de Microservicios, el enfoque de división de la aplicación es diferente; la misma se divide en servicios que están organizados en torno a las capacidades del negocio.

Estos microservicios implementan la funcionalidad completa del software para el área de negocio, incluyendo la interface a usuario, la persistencia en el almacenamiento, y cualquiera de las colaboraciones externas. De esta manera, los equipos poseen funciones cruzadas, incluyendo toda la gama de habilidades necesarias para el desarrollo: la experiencia de usuario, la base de datos, y la gestión de proyectos.

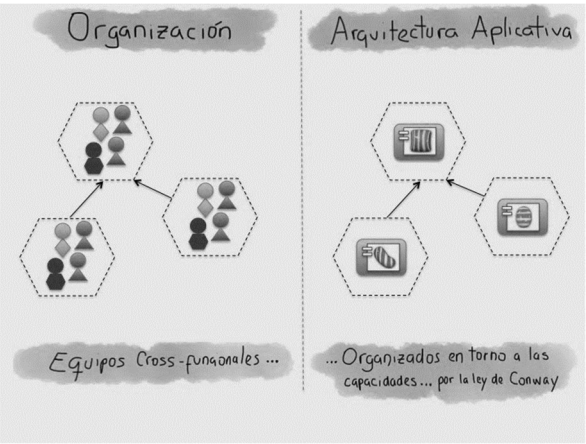


Ilustración 10 - Enfoque de los servicios organizados en torno a las capacidades de negocio

Productos, no proyectos

La mayoría de los desarrollos aplicativos que conocemos, se realizan bajo el esquema de proyectos, es decir, un equipo de desarrollo (Dev) define un alcance, y al cabo de un tiempo entrega a producción una pieza de software (producto o servicio) que se considera terminado. Pasado un cierto tiempo de estabilización, otro equipo, normalmente el de Operaciones (Ops), toma el control del producto, y el equipo de desarrollo que formó parte del proyecto se disuelve para hacerse cargo de otros proyectos.

Quienes utilizan el estilo Microservicios, tratan de evitar este modelo.

En su lugar, prefieren la idea de que un equipo debe estar a cargo del producto durante toda la vida del mismo. Esto ha estado inspirado en uno de los principios de Amazon que dice: "lo construyes, lo ejecutas" ("you build it, you run it"). Donde un equipo de desarrollo asume completa responsabilidad de ese software, no sólo en la etapa de diseño y construcción, sino también en toda la fase de producción y mantenimiento.

Esto lleva a los desarrolladores a estar en contacto día a día con la operación de su software. También les pone en contacto diario con el cliente. Este bucle de retroalimentación constante con el cliente es esencial para la mejora de la calidad de servicio.

Esta mentalidad de producto se relaciona también con la vinculación a las capacidades de negocio. En lugar de ver al software como un conjunto de funcionalidades terminadas, hay una

relación continua, donde la pregunta es cómo puede el software ayudar a sus usuarios a mejorar las capacidades de negocio.

La principal corriente que apoya esta forma de gestionar los productos es la llamada DevOps (Desarrollo - Operaciones), la cual ya está presente en la mayoría de las start-ups y empresas de internet.

Si bien este enfoque se podría llevar a las aplicaciones monolíticas, el nivel de granularidad que poseen los microservicios permite que sea más fácil crear relaciones personales entre los desarrolladores de servicios y sus usuarios.

Gestión de datos descentralizada

La descentralización de la gestión de datos se presenta de diferentes maneras. En el nivel más abstracto, significa que el modelo conceptual del mundo será diferente entre los sistemas. Este es un problema común de integración a través de una gran empresa, por ejemplo, la vista de las ventas de un cliente difiere de la vista del área de soporte. Lo que llamamos "cliente" en la vista de ventas, a veces no existe en la vista de soporte. Aquellos que coinciden, pueden incluso tener atributos distintos, o lo que es peor, los mismos atributos, pero con semántica sutilmente diferente.

Realmente el gobierno de datos en una gran empresa es complejo, más aún cuando dicha organización ha tenido múltiples evoluciones en sus aplicaciones a lo largo de los años, lo que deja como resultado lo que solemos llamar "capas geológicas aplicativas". Cada una de estas aplicaciones posee una parte de las entidades principales (clientes, productos, ordenes, recursos…), y generalmente sus atributos no coinciden.

Este problema es común entre las aplicaciones, pero también suele suceder dentro de las aplicaciones, en particular, cuando dicha aplicación se divide en componentes separados.

Una manera útil de trabajar esto es mediante las técnicas del diseño dirigido por el dominio (DDD, Domain-Driven Design).

Es muy importante cuando se construye un nuevo sistema, dedicarle tiempo al diseño, y más aún si el sistema es complejo. Uno puede sentarse y comenzar a codificar, y muchas veces para desarrollos simples esto funciona. Pero, para sistemas complejos, necesitamos dedicarle un tiempo al diseño, y para esta etapa es clave que el desarrollador trabaje en conjunto con aquellos recursos que más conocen del dominio del sistema que estamos intentando construir. Estos son los expertos de dominio, de manera similar a lo que sucede con la construcción de un avión, donde intervienen los expertos de propulsión, estructura, aviónica, etc.

Domain-Driven Design es un enfoque de desarrollo de software que ayuda a estos casos de proyectos complejos. DDD divide un dominio complejo en múltiples contextos y traza las relaciones entre ellos, haciendo hincapié en la colaboración entre los expertos de dominio, desarrolladores y otros, para lograr un entendimiento común sobre un dominio en particular, y para acotar el contexto de forma iterativa hasta llegar al corazón conceptual del problema.

El proceso es útil para ambos estilos: monolítico y microservicios, pero existe una correlación natural entre los límites del servicio y del contexto que ayuda a clarificar y reforzar las separaciones entre ellos.

Adicionalmente, cuando se trabaja con DDD, en especial, sobre el diseño de modelos de datos que requieren una compleja interacción o mucha lógica de negocio durante su manipulación, suele ser conveniente la aplicación de un patrón de diseño llamado CQRS (Command Query Responsibility Segregation).

Básicamente, CQRS consiste en la separación del modelo de datos para comandos de escritura (Command) del modelo de datos de lectura (Query).

Para aplicaciones triviales de una empresa que incluyen las operaciones básicas de interacción llamadas CRUD (Create, Read, Update, Delete) sobre los almacenes de datos, donde no existe un comportamiento complejo de operaciones, este patrón realmente no es aplicable.

En cambio, cuando las operaciones, ya sean del tipo comandos o consultas se tornan más complejas, por ejemplo, consultas que requieren hacer cálculos complejos sobre grandes conjuntos de datos, o comandos que requieren múltiples validaciones o complejas reglas de negocio, es decir, en aquellas áreas donde la complejidad es particular del domino en cuestión, es donde CQRS comienza a tener sentido.

Esto está relacionado con la naturaleza distinta de ambos mundos, que surgen del diseño obtenido con los expertos en DDD, donde los comandos, generalmente, requieren niveles de consistencia elevados, mientras las lecturas o consultas requieren de velocidad ante el manejo de grandes conjuntos de datos.

Este enfoque también tiene que ver con el principio de responsabilidad única (SRP, Single Responsibility Principle), donde en lugar de tener un objeto que tiene distintas responsabilidades, es conveniente separarlo en distintos objetos, cada uno con su correspondiente responsabilidad.

Esta separación de responsabilidades habilita a diseñar distintos tipos de repositorios y técnicas apropiadas para cada tipo de operación, y esto es, básicamente lo que trata CQRS. Esto está representado en la siguiente figura:

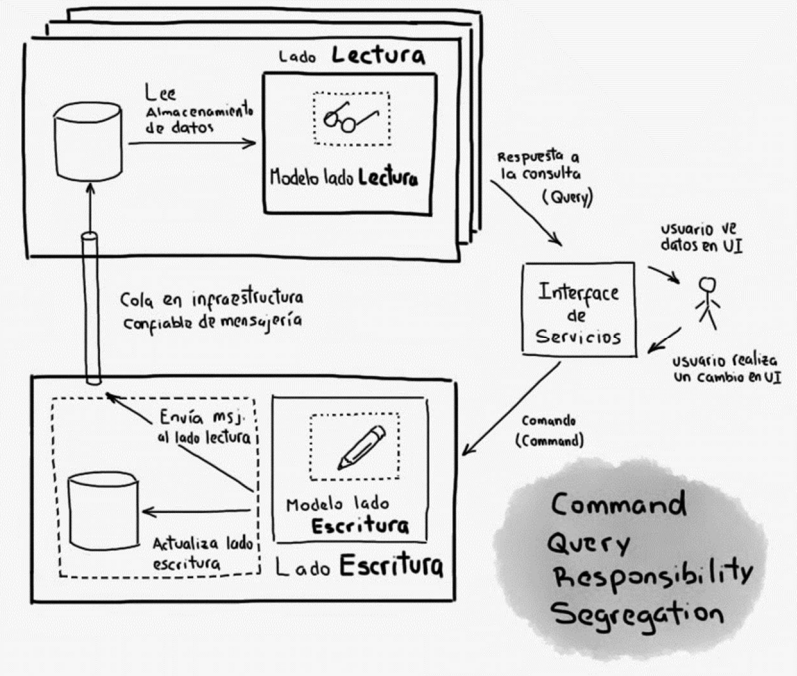


Ilustración 11 - Command Query Responsability Segregation (CQRS)

En relación con lo anterior, así como se produce la descentralización de las decisiones sobre los modelos conceptuales como producto del DDD, los sistemas basados en microservicios también descentralizan las decisiones sobre el tipo de almacenamiento de datos.

Mientras que las aplicaciones monolíticas prefieren una única base de datos lógica para los datos persistentes, a su vez, las empresas prefieren una única base de datos física para varias aplicaciones, por considerar esto, en teoría, más eficiente.

En cambio, los Microservicios prefieren dejar a cada servicio que gestione su propia base de datos, sean estas diferentes instancias de la misma tecnología de base de datos, o sistemas de base de datos completamente diferentes, enfoque que se conoce como Persistencia Políglota.

Así, por ejemplo, una aplicación Web de tienda minorista, en lugar de usar una única base de datos (modelo monolítico), podría usar múltiples tipos de repositorios (modelo de microservicios) dependiendo del tipo de datos, por ejemplo:

- Redis (Key-value cache store): para las sesiones de usuarios
- MySQL (RDBMS SQL): para los datos de transacciones de pago
- MongoDB (Document store): para el catálogo de productos
- Neo4J (Graph store): para las recomendaciones
- Cassandra (Key-Value store): para el análisis de logs y analytics

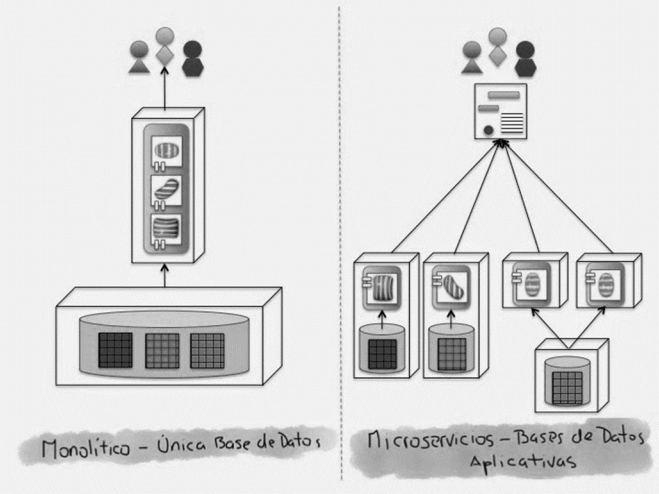


Ilustración 12 - Persistencia políglota en arquitecturas de microservicios

Si bien podemos usar persistencia políglota en arquitecturas monolíticas, ésta aparece más frecuentemente en microservicios.

Existen numerosos ejemplos de implementaciones de persistencia políglota, entre los cuales, se destacan los grandes de internet, fundamentalmente: Google, Twitter, Amazon, Facebook… etc., por haber sido, en parte, los creadores de varios de los nuevos repositorios. Pero también existen otros casos menos conocidos, como, por ejemplo, Klout con su aplicación de análisis de impacto social que utiliza HBase, MySQL, ElasticSearch, MongoDB y HDFS; o Imgur, un sitio para compartir y almacenar fotos, al que se suben más de 1.500.000 fotos diarias, a pesar de sus relativas necesidades básicas, usa cinco tipos de repositorios distintos, incluyendo Redis, ElasticSearch y MySQL (vía Amazon RDS).

Ahora bien, en la arquitectura de microservicios, la descentralización de la responsabilidad de los datos tiene implicancias en el manejo de las actualizaciones. Una forma de tratar las actualizaciones ha sido la de utilizar transacciones para garantizar la consistencia en la actualización de múltiples recursos. Enfoque utilizado también ampliamente con aplicaciones monolíticas.

Si bien, el uso de transacciones ayuda con la consistencia, impone un acoplamiento temporal significativo, lo que se torna problemático a través de múltiples servicios.

Debido a que las transacciones distribuidas son mucho más difíciles de implementar, las arquitecturas de microservicios promueven la coordinación no transaccional entre servicios, con el reconocimiento explícito que la consistencia puede ser solamente una consistencia eventual, y los problemas son compensados operativamente.

El negocio suele manejar un grado de inconsistencia para responder rápidamente a la demanda, mientras dispone de algún tipo de proceso de contingencia para manejar los errores. El trade-off vale la pena, siempre y cuando, el costo de solucionar los errores sea menor que el costo de perder negocios por una mayor consistencia.

Los servicios son prioritarios

En el mundo de los microservicios, los servicios son ciudadanos de primera clase. Los microservicios exponen el extremo de conexión de un servicio como APIs, y abstraen de los detalles de implementación. La lógica de implementación interna, la arquitectura, y las tecnologías utilizadas (incluyendo el lenguaje de programación, bases de datos … etc.) están completamente ocultos detrás de la API del servicio.

Además, en la arquitectura de microservicios, no hay más desarrollo de aplicaciones. En su lugar, las organizaciones se centran en el desarrollo de servicios. En la mayoría de las empresas, esto exige un cambio cultural mayor en la forma de construir aplicaciones.

Por ejemplo, en un microservicio de perfil de cliente, los detalles internos como la estructura de datos, tecnologías, lógica de negocio… etc., están ocultos. Estos detalles no son visibles desde las entidades externas. Y el acceso está restringido a través de extremos de conexión de servicios o APIs.

Características de los servicios en un microservicio

Como los microservicios son muy parecidos a SOA, muchas de las características definidas en SOA son aplicables a los microservicios también

Las siguientes son algunas de las características de los servicios que son aplicables a los microservicios también:

- **Contrato de servicio**. De una manera similar a SOA, los microservicios están descritos a través de contratos de servicio bien definidos. En el mundo de los microservicios, JSON y REST son el estándar de comunicación universalmente aceptado. En el caso de JSON/REST, hay muchas técnicas utilizadas para definir estos contratos, como JSON Schema, WADL, Swagger, y RAML, por citar algunos ejemplos.
- **Bajo acoplamiento**. Los microservicios son independientes y están débilmente acoplados. En la mayoría de los casos, los microservicios aceptan un evento como entrada y responden con otro evento. Los mensajes, HTTP y REST son utilizados con mucha frecuencia para la interacción entre los microservicios. Los extremos de conexión basados en mensajería ofrecen altos niveles de desacoplamiento.
- **Abstracción del servicio**. En los microservicios, la abstracción del servicio no es sólo una abstracción de la ejecución del servicio, también ofrece una completa abstracción de todas las librerías y detalles del entorno.
- **Reutilización del servicio**. Los microservicios son servicios de negocio reutilizables. Son utilizados por dispositivos móviles, aplicaciones de escritorio, otros microservicios, o incluso otros sistemas.
- **No mantienen el estado**. Los microservicios bien diseñados no mantienen el estado y no comparten nada con ningún estado compartido o estado conversacional mantenido por los servicios. Y en el caso que hiciese falta, se mantendría en una base de datos, quizás en memoria
- **Pueden ser descubiertos**. En un entorno de microservicios típico, los microservicios exponen su existencia y se hacen accesibles para poder ser encontrados. Cuando el servicio muere, ellos mismos salen del ecosistema de microservicios.
- **Interoperabilidad entre servicios**. Los servicios son interoperables en la medida que utilizan protocolos conocidos y estándares para el intercambio de mensajes. La mensajería, HTTP, otras tantos, son utilizados como mecanismos de transporte. REST/JSON es el método más popular para desarrollar servicios interoperables en el mundo de los microservicios. En los casos en los que se hace necesaria una optimización adicional en las comunicaciones, se pueden utilizar otros protocolos, como Thrift, Avro o Zero MQ. Sin embargo, el uso de estos protocolos puede limitar la interoperabilidad entre los servicios.

- **Composición de servicios**. Se debería poder componer nuevos microservicios a partir de los microservicios existentes, ya sea por medio de un servicio de orquestación o un servicio de coreografía.

Los microservicios son ligeros

Los microservicios bien diseñados están enfocados hacía una única capacidad del negocio, por lo tanto, realizan una única función. Como resultado, una de las características más comunes que vemos en la mayoría de las implementaciones son microservicios con un alcance menor.

Cuando tenemos que elegir las tecnologías en las que nos vamos a apoyar, como los contenedores web, tendremos que asegurarnos que son ligeras para que los microservicios sean manejables. Por ejemplo, Jetty o Tomcat son las mejores elecciones como contenedores de aplicaciones para microservicios, en comparación con servidores de aplicaciones más complejos como WebLogic o WebSphere.

Las tecnologías de contenedores como Docker también nos ayudan a mantener el impacto de la infraestructura lo más pequeña posible en comparación con tecnologías como VMWare o Hyper-V

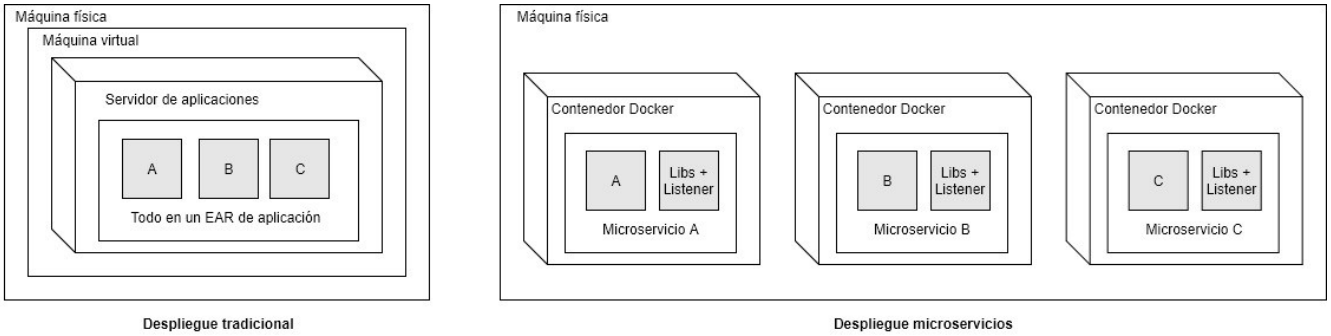


Ilustración 13 - Comparativa entre un despliegue tradicional y un despliegue de microservicios

Tal y como se muestra en la imagen, los microservicios se despliegan por lo general en contenedores Docker, los cuales encapsulan la lógica de negocio y las librerías necesarias. Esto nos ayuda a replicar rápidamente la configuración entera en una maquina nueva o en un entorno completamente diferente, o incluso trasladarlo a través de distintos proveedores de la Cloud. Como no hay dependencia con la infraestructura física, los microservicios en contenedores son portables con mucha facilidad.

Microservicios con arquitectura poliglota

Puesto que los microservicios son autónomos y abstraen todo lo que hay detrás de las APIs de los servicios, es posible tener diferentes arquitecturas para distintos microservicios. Unas pocas características comunes que vemos en las implementaciones de los microservicios son:

- Diferentes servicios utilizan diferentes versiones de las mismas tecnologías. Un microservicio puede estar desarrollado con Java 1.7, y otra podría estarlo con Java 1.8
- Se utilizan diferentes lenguajes para desarrollar diferentes microservicios, de manera que un microservicio podría estar implementado en Java y otro en Scala.
- Se utilizan arquitecturas diferentes. De esta manera, mientras que un microservicio utiliza la cache de Redis como servidor de datos, otro podría utilizar MySQL como base de datos persistente.

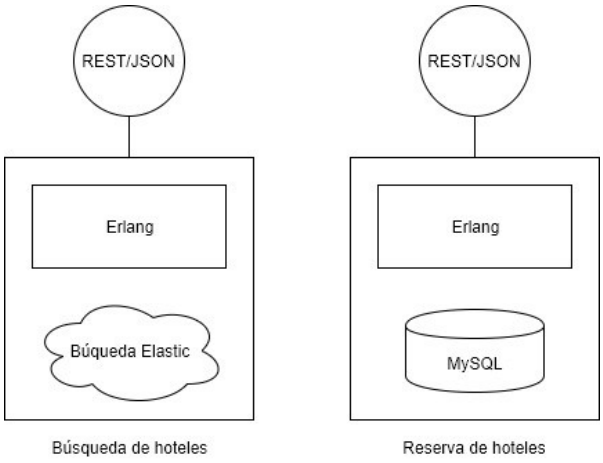


Ilustración 14 - Microservicios con arquitectura políglota

Una de las principales consecuencias de un gobierno centralizado es la tendencia en estandarizar en plataformas tecnológicas individuales.

Esto es lo que sucede con el enfoque monolítico, donde es común utilizar las mismas tecnologías para cada una de las capas de un sistema, por ejemplo, para la capa de lógica de negocio, normalmente se utiliza el mismo lenguaje de programación (Java, .Net…); para la base de datos, el mismo tipo de tecnología de base de datos (MySql, Oracle…), independientemente del tipo de entidades y el uso específico que se realice con los datos.

En cambio, con un sistema compuesto de múltiples servicios colaborativos, como es del enfoque de microservicios, podemos decidir utilizar diferentes lenguajes de programación y tecnologías dentro de cada servicio. Esto nos permite elegir la herramienta adecuada para cada tipo de trabajo, en lugar de tener que elegir una estandarizada, es decir, una solución única para todo.

Adicionalmente, si una parte del sistema necesita mejorar su rendimiento, es posible utilizar una solución tecnológica que sea capaz de alcanzar los niveles de rendimiento requeridos.

Por otro lado, también podemos decidir que la forma en la que almacenamos los datos sea diferente para diferentes partes de nuestro sistema. Por ejemplo, para una red social, podríamos almacenar interacciones de nuestros usuarios en una base de datos orientada a grafos (Graph-DB) para reflejar la naturaleza altamente interconectada de un gráfico social, pero tal vez, los mensajes que los usuarios postean podrían estar almacenados en un almacén de datos orientada a documentos (NoSQL).:

Además, con los microservicios podemos adoptar nuevas tecnologías más rápido y entender cómo los nuevos avances nos pueden ayudar. Una de las principales barreras para la adopción de nuevas tecnologías es el riesgo asociado que conlleva el cambio. En una aplicación monolítica, si queremos probar un nuevo lenguaje de programación, base de datos o framework, cualquier cambio impactará en una gran porción del sistema. Con un sistema que consiste en múltiples servicios, tal vez puedo tomar un servicio de menor riesgo para probar cierta tecnología en particular. Varias organizaciones encuentran esta capacidad de absorber nuevas tecnologías más rápidamente como una verdadera ventaja competitiva.

Automatización en un entorno de microservicios

La mayoría de las implementaciones de los microservicios están automatizadas al máximo desde el entorno de desarrollo al de producción.

Dado que, los microservicios descomponen las aplicaciones monolíticas en un número más pequeño de servicios, las grandes empresas pueden ver una gran proliferación de microservicios. Y un número elevado de microservicios es muy difícil de gestionar a menos que se utilice alguna

técnica de automatización. Además, el tamaño reducido de los microservicios nos ayuda a automatizar su ciclo de vida de despliegue. En general, los microservicios están automatizados de principio a fin, por ejemplo, la compilación, empaquetado, pruebas o escalado.

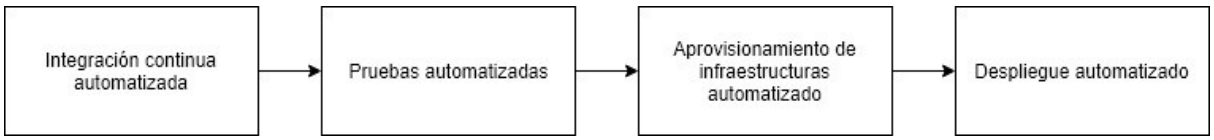


Ilustración 15 - Automatización en un entorno de microservicios

Tal y como se puede observar, las automatizaciones se aplican por lo general durante las fases de desarrollo, pruebas, versionados y despliegue:

- La fase de despliegue esta automatizada utilizando herramientas de versión de control como Git, junto con herramientas de integración continua como Jenkins, Travis CI, entre otras. Esta fase puede incluir también pruebas de calidad del código y automatizaciones de pruebas unitarias.
- La fase de pruebas se automatizará utilizando herramientas de comprobación como Selenium o Cucumber, y otras estrategias de pruebas AB. Puesto que los microservicios están enfocados a las capacidades del negocio, el número de casos de prueba a automatizar es mucho menor que en el caso de aplicaciones monolíticas, por lo que, las pruebas de regresión en cada compilación resultan posibles.
- El aprovisionamiento de infraestructuras se realiza a través de tecnologías de contenedor como Docker, junto con herramientas de gestión de versionado como Chef o Puppet, y herramientas de gestión de la configuración como Ansible. Los despliegues automatizados se pueden llevar a cabo mediante herramientas como Spring Cloud, Kubernetes, Mesos y Marathon.

Los microservicios con un ecosistema de soporte

La mayoría de las implementaciones de microservicios a gran escala tienen un ecosistema de soporte en su lugar. Las capacidades de este ecosistema incluyen procesos DevOps, gestión de trazas centralizado, servicio de registros, pasarelas API, monitorización extensiva, servicio de enrutamiento y mecanismos de control de flujo.

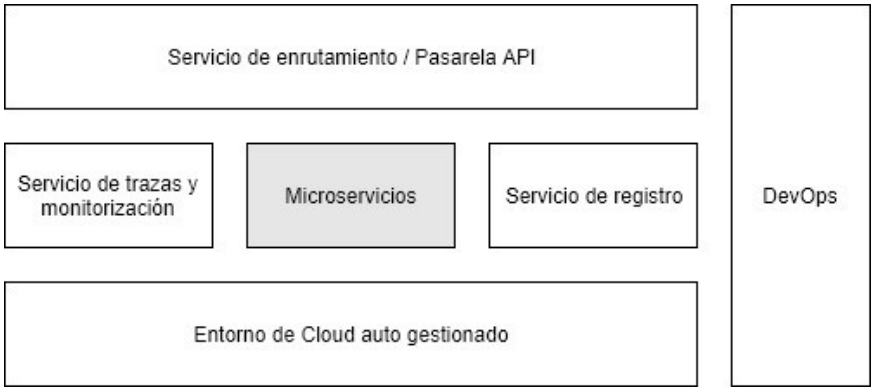


Ilustración 16 - Ecosistema de soporte en una arquitectura de microservicios

Los microservicios son distribuidos y dinámicos

Las implementaciones exitosas de microservicios encapsulan la lógica y los datos dentro de los servicios. Esto da lugar a dos situaciones poco convencionales: lógica y datos distribuidos y una gestión descentralizada.

En comparación con las aplicaciones tradicionales, que engloban toda la lógica y los datos dentro de los límites de la aplicación, los microservicios descentralizan los datos y la lógica. Cada

servicio, enfocado en una capacidad específica del negocio, tiene sus propios datos y su propia lógica.

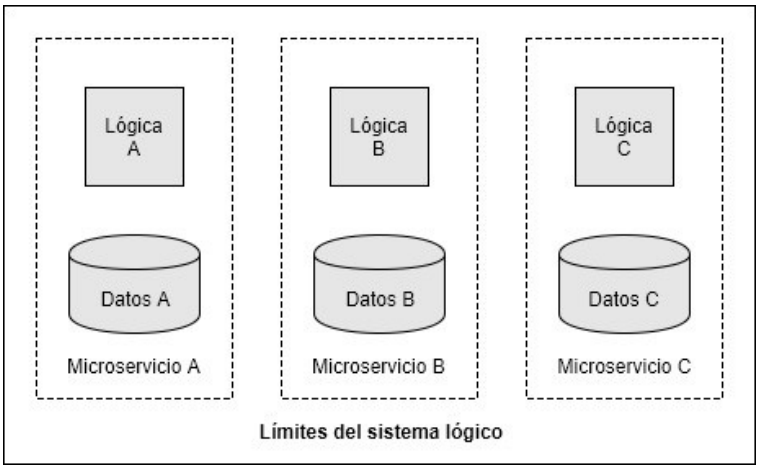


Ilustración 17 - Desacoplamiento de la lógica y los datos en una arquitectura de microservicios

Los microservicios no utilizan por lo general mecanismos de gobierno o gestión centralizados, como lo hace SOA. Una de las características comunes en las implementaciones de los microservicios es que no dependen de pesados productos de nivel empresarial como Enterprise Service Bus (ESB). En vez de eso, la lógica de negocio y la inteligencia forman parte del propio servicio.

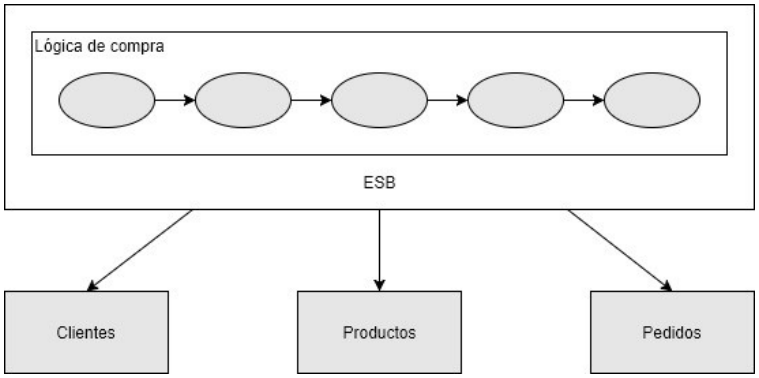


Ilustración 18 - Implementación SOA con ESB (Enterprise Service Bus)

La imagen anterior presenta una implementación típica de SOA. La lógica para hacer una compra está implementada por completo en ESA mediante la orquestación de diferentes servicios. Sin embargo, en el planteamiento de los microservicios, está lógica se ejecutará como un microservicio separado, que interactuará con otros microservicios de una manera desacoplada.

Las implementaciones de SOA dependen de configuraciones de repositorios y registros estáticos para gestionar los servicios y otros artefactos. Los microservicios, por el contrario, trabajan de una manera más dinámica y natural. Por lo tanto, un enfoque de gestión estática se ve como un esfuerzo para mantener su información actualizada. Este es la razón por la que la mayoría de las implementaciones de microservicios utilizan mecanismos automatizados para construir información de registros dinámicamente desde las propias topologías en tiempo de ejecución.

Los beneficios de los microservicios

Los microservicios ofrecen múltiples beneficios frente a las arquitecturas tradicionales multicapa.

Soporta arquitectura poliglota

Con microservicios, arquitectos y desarrolladores pueden elegir las arquitecturas y tecnologías que mejor encajan para cada microservicio. Esto da la flexibilidad para diseñar las soluciones que mejor se adapten con el mejor coste posible.

Como los microservicios son autónomos e independientes, cada servicio puede ejecutar su propia arquitectura o tecnología, o incluso, diferentes versiones de la misma tecnología.

La siguiente imagen muestra un ejemplo práctico y simple de una arquitectura políglota con microservicios.

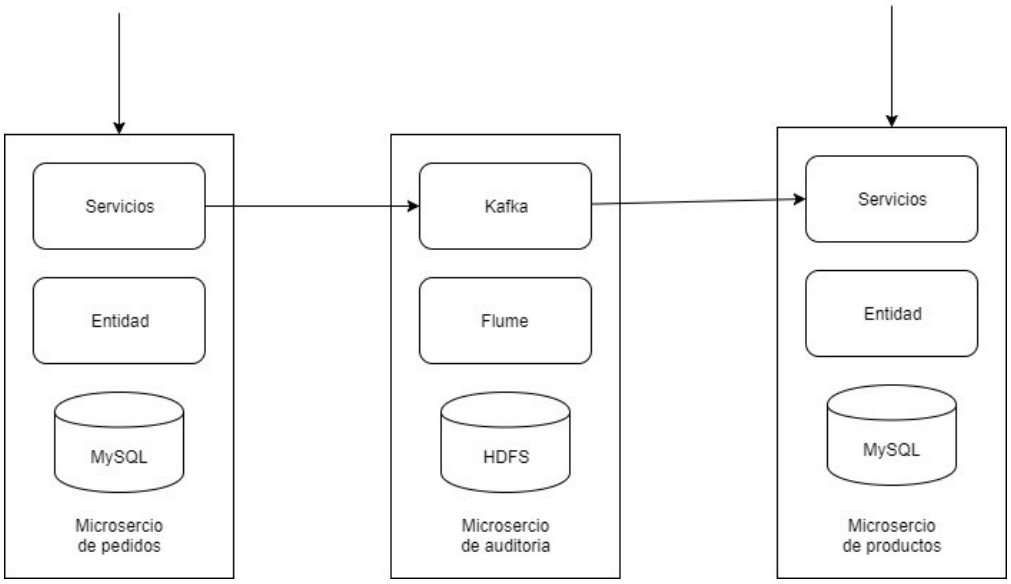


Ilustración 19 - Ejemplo de arquitectura políglota con microservicios

Tal y como se muestra en la imagen, mientras que los servicios principales como los microservicios de Pedidos y Productos usan un repositorio de datos relacional, el microservicio de Auditoria mantiene la información en un sistema de ficheros Hadoop (HDFS). En un planteamiento de aplicación monolítica, la aplicación generalmente utiliza una base de datos única y compartida que almacena Pedidos, Productos y datos de auditoria.

En este ejemplo, el servicio de auditoria es un microservicio técnico que utiliza una arquitectura diferente. De la misma manera, diferentes servicios funcionales podrían utilizar distintas arquitecturas.

En otro ejemplo, podría haber un microservicio de reservas funcionando en Java 7, mientras que un microservicio de búsquedas podría funcionar en Java 8. Igualmente, un microservicio de pedidos podría estar escrito en Erlang, mientras que un microservicio de entregas podría estar desarrollado en el lenguaje Go. Nada de esto es posible dentro del contexto de una aplicación monolítica.

Permite la experimentación y la innovación

Las empresas modernas están progresando hacia las ganancias rápidas. Los microservicios son una de las clases que permiten las empresas llevar a cabo una gran innovación mediante la posibilidad de experimentar y fallar de una manera muy rápida.

Puesto que los microservicios son bastante simples y mucho más pequeños en tamaño, las empresas pueden esforzarse en experimentar nuevos procesos, algoritmos, lógicas de negocio… etc. En el caso de grandes aplicaciones monolíticas, la experimentación no resulta sencilla, no es rápido ni asequible económicamente. Las empresas tienen que gastar una gran cantidad de dinero

en construir o cambiar una aplicación para intentar algo nuevo. Con los microservicios, sin embargo, es posible escribir un pequeño microservicio para alcanzar la funcionalidad buscada y acoplarlo en un sistema de una manera reactiva. Entonces uno puede experimentar con la nueva funcionalidad por unos meses, y si el microservicio no funciona como se esperaba, se puede modificar o remplazar por otro. En este caso, el coste del cambio será muchísimo mejor que para una aplicación tradicional monolítica.

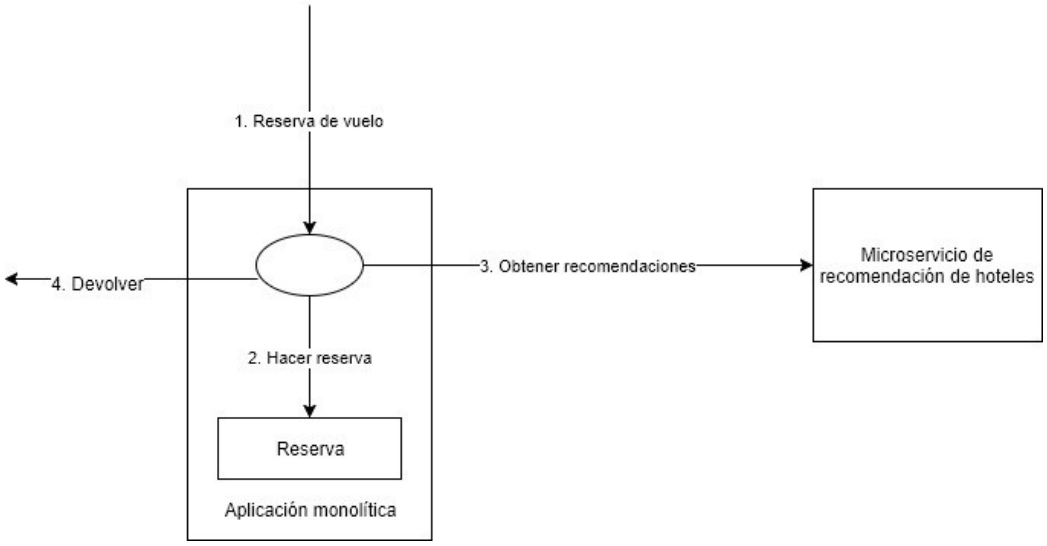


Ilustración 20 - Ejemplo de integración de microservicio en una aplicación monolítica

Pongamos el caso de un sitio web de una compañía de reserva de vuelos, la compañía quiere mostrar recomendaciones de hotel personalizadas en su página de reservas. Y las recomendaciones se deben mostrar en la página de confirmación de la reserva.

Tal y como se muestra en la imagen anterior, resulta conveniente desarrollar un microservicio que puede ser acoplada en la aplicación de reservas, en vez de incorporarlo como parte integrada en la propia aplicación monolítica. La compañía en ese caso puede elegir comenzar con un simple servicio de recomendaciones e ir remplazando con nuevas versiones hasta dar con el resultado buscado.

Elasticidad y escalado selectivo

Puesto que los microservicios son pequeñas unidades de trabajo, nos permiten implementar escalabilidad selectiva.

Los requisitos de escalabilidad pueden ser diferentes para distintas funcionalidades dentro de la misma aplicación. Una aplicación monolítica, empaquetada como un único WAR o EAR, sólo puede ser escalado como una unidad de manera completa.

Sin embargo, en el caso de los microservicios, cada microservicio podría ser escalado independientemente, ya sea para ampliarlo o para reducirlo. Como la escalabilidad puede ser aplicada a cada servicio de manera selectiva, el coste derivado del escalado es comparativamente mucho menor en una arquitectura de microservicios.

En la práctica existen muchas formas diferentes de escalar una aplicación y está íntimamente ligado a la arquitectura y el comportamiento de la aplicación. El cubo del escalado define tres estrategias para escalar una aplicación:

- Escalando el eje X, clonando horizontalmente la aplicación
- Escalando el eje Y, separando diferentes funcionalidades
- Escalando el eje Z, particionando o fragmentando los datos

Cuando el escalado en el eje Y se aplica a aplicaciones monolíticas, ésta se divide en pequeñas unidades alineadas con las funciones del negocio. Muchas organizaciones aplican satisfactoriamente esta técnica para alejarse de la tradicional visión de aplicación monolítica. En principio, las unidades resultantes están en línea con las características de los microservicios.

Por ejemplo, en una página web típica de una compañía aérea, las estadísticas indican que el radio de búsqueda de vuelos para realizar una reserva podría llegar a 500:1. Esto quiere decir, que se realiza una transacción de reserva por cada 500 transacciones de búsqueda. En este escenario, la búsqueda necesita 500 veces más escalabilidad que la función de reserva. Este sería un caso ideal para el escalado selectivo.

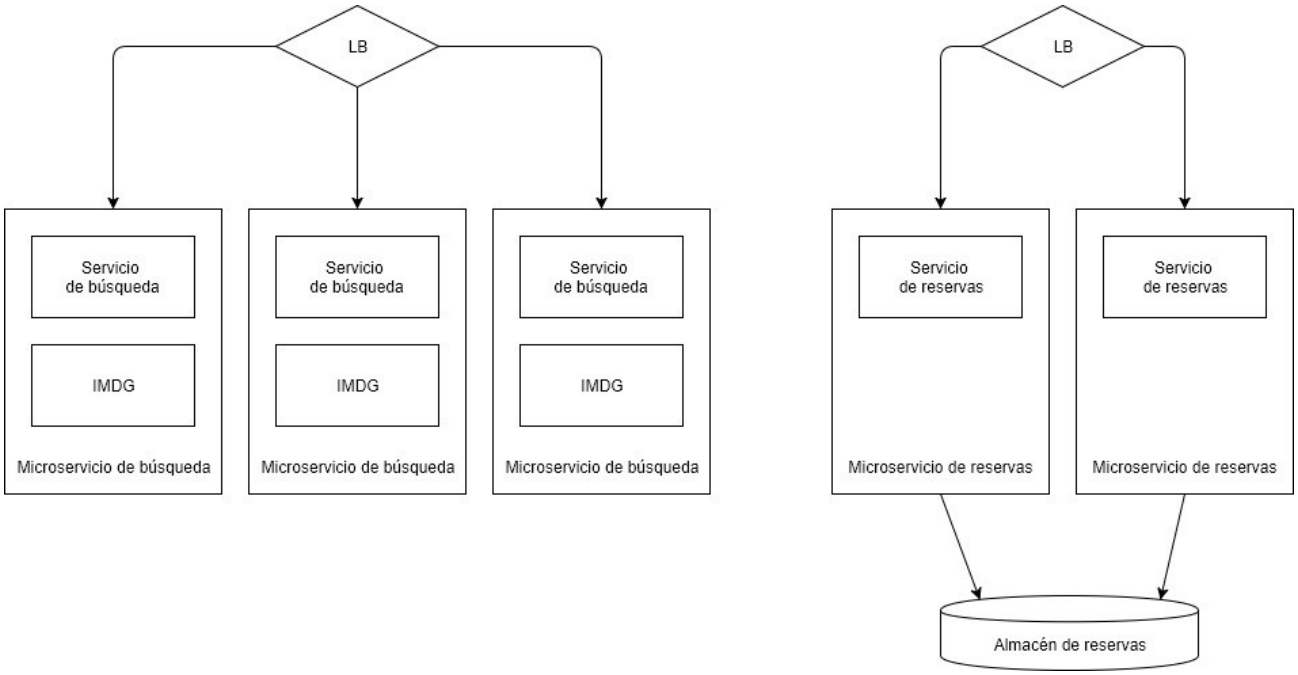


Ilustración 21 - Escalabilidad en una arquitectura de microservicios

La solución es tratar las solicitudes de búsqueda y reserva de manera diferente. Con una arquitectura monolítica, esto sería sólo posible con un escalado en el eje Z. En el que se replica la aplicación en su totalidad, con un coste mucho mayor, obviamente.

Tal y como se puede ver en la imagen, los microservicios de búsqueda y reservas están diseñados como microservicios separados, de manera que el servicio de búsqueda puede ser escalado de manera independiente al de reservas. Mientras que el microservicio de búsqueda tiene tres instancias, el de reservas tiene dos. La escalabilidad selectiva no está limitada al número de instancias, tal y como se muestra en la imagen, también en la manera en la que los microservicios son diseñados. En el caso del microservicio de búsqueda, una tabla de datos en memoria (in-memory data grid, IMDG) como Hazelcast se puede utilizar para almacenar la información. Esto incrementará aún más el rendimiento y la escalabilidad de este servicio. Cuando un nuevo microservicio de búsqueda es instanciado, un nodo IMDG adicional es añadido al cluster existente IMDG. Sin embargo, el servicio de reservas no requiere la misma escalabilidad. En este caso, las instancias de este servicio comparten la misma instancia de base de datos.

Permite la sustitución

Los microservicios son módulos desplegables autocontenidos e independientes, lo que permite la sustitución de un microservicio con otro microservicio similar.

Muchas grandes empresas siguen la política de comprar y construir para implementar sus sistemas software. Un escenario común es construir la mayoría de las funcionalidades dentro de la propia empresa y comprar ciertas funcionalidades adicionales a otras empresas. Esto plantea retos en las aplicaciones monolíticas tradicionales, ya que son muy cohesivas. Intentar acoplar

soluciones de terceras partes a aplicaciones monolíticas da como resultado integraciones muy complejas. Con microservicios esto no es un problema. Desde el punto de vista de la arquitectura, un microservicio puede ser fácilmente remplazado por otro microservicio desarrollado por la propia u otra empresa.

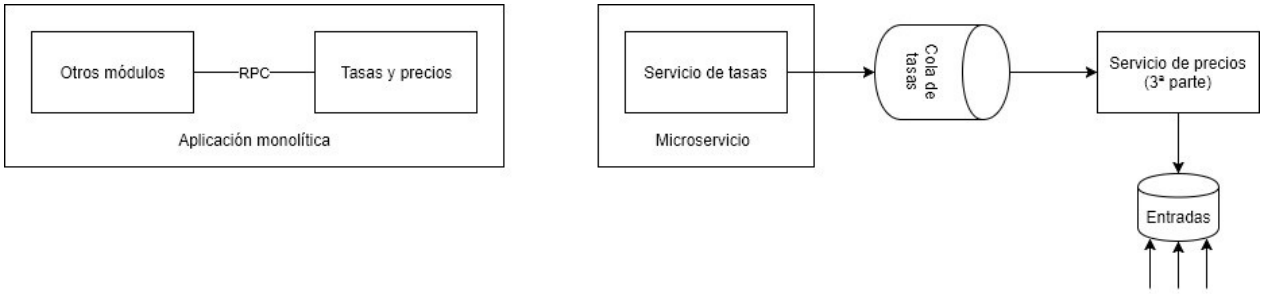


Ilustración 22 - Sustitución de microservicios

El mecanismo de elección de precios en una compañía aérea es complejo. Las tasas para diferentes rutas se calculan usando complejas fórmulas matemáticas conocidas como la lógica de precios. Las compañías aéreas pueden decidir comprar esta lógica a un tercero, en vez de construirlo ellos mismos. En una arquitectura monolítica, la funcionalidad de precios es una función de tasas y reservas. Y en la mayoría de los casos los precios, las tasas y las reservas están altamente acopladas, resultando prácticamente imposible desacoplarlas.

En un sistema de microservicios bien diseñado, las reservas, tasas y precios sería microservicios independientes. Remplazar el microservicio de precios tendría un impacto mínimo en los otros servicios, en la medida que son independientes y están muy débilmente acoplados. Hoy, podría ser un servicio comprado a otra empresa; mañana, podría sustituirse por otro desarrollado por otra empresa diferente.

Permitir construir sistemas orgánicos

Los microservicios nos ayudan a construir sistemas que son orgánicos en su naturaleza. Esto es significativamente importante cuando migramos sistemas monolíticos gradualmente a microservicios.

Los sistemas orgánicos son sistemas que crecer lateralmente en un periodo de tiempo añadiendo gradualmente más y más funcionalidades. En la práctica, una aplicación crece indefinidamente durante su tiempo de vida, y, en la mayoría de los casos, la capacidad de gestionar la aplicación se reduce drásticamente de manera directamente proporcional a lo largo de este tiempo.

En la medida que los microservicios son servicios gestionables de manera independiente. Esto nos permite seguir añadiendo progresivamente más y más servicios de acuerdo con las necesidades de cada momento con un impacto mínimo en los servicios existentes. Construir este tipo de sistemas no requiere una inversión enorme de capital. Por lo tanto, los negocios pueden mantener la construcción de estas nuevas funcionalidades como parte de su gasto operacional.

Pongamos como ejemplo un sistema de fidelidad que fue construido por una compañía aérea hace años, enfocado a pasajeros individuales. Esto valió hasta que la compañía aérea empezó a ofrecer beneficios de fidelidad a sus clientes corporativos, es decir, clientes agrupados bajo determinada corporación. Puesto que el modelo de datos del sistema actual es plano (individuos particulares), la empresa necesita un cambio fundamental en el modelo de datos, y, por lo tanto, un enorme trabajo de reconstrucción para incorporar este requisito.

Ilustración 23 - Sistemas orgánicos en una arquitectura de microservicios

Tal y como se puede comprobar en la imagen, en una arquitectura basada en microservicios, la información del cliente sería gestionada por el microservicio de clientes y el de fidelidad por el microservicio de puntos de fidelidad.

En esta situación resulta fácil añadir un nuevo microservicio de clientes corporativos para gestionar este tipo de clientes. Cuando se registra una corporación, los miembros de ésta serán enviados al microservicio de clientes para gestionarlos de la manera habitual. El microservicio de clientes corporativos ofrece una vista corporativa añadiendo datos desde el microservicio de cliente. E igualmente, ofrecerá servicios para dar soporte a las reglas de negocio específicas de corporaciones. Con este planteamiento, añadir nuevos servicios tendrá un impacto mínimo en los servicios que ya existen.

Ayudar a reducir el desfase tecnológico

Puesto que los microservicios tienen un tamaño más reducido y dependencias mínimas entre ellos, es posible la migración de servicios que usan tecnologías desfasadas o en proceso de desaparecer con un coste mínimo.

Los cambios tecnológicos son una de las grandes barreras en el desarrollo software. En muchas aplicaciones monolíticas tradicionales, debido a los rápidos cambios tecnológicos, las aplicaciones de la próxima generación de la actualidad podrían llegar a estar fácilmente desfasadas incluso antes de ponerse en producción. Los arquitectos y desarrolladores tienden a ofrecer gran resistencia a los cambios tecnológicos añadiendo capas de abstracción. Sin embargo, en realidad, esta estrategia no sólo no soluciona el problema, sino que termine dando lugar a sistemas con una arquitectura exagerada. Como las actualizaciones tecnológicas suelen ser arriesgadas y caras, y, además no suponen un beneficio directo en el negocio, las empresas suelen ser bastante reticentes a hacer este tipo de inversión

Con los microservicios, es posible cambiar o actualizar la tecnología para cada microservicio de manera individual en vez de actualiza el conjunto de la aplicación en su totalidad.

Actualizar una aplicación, con, por ejemplo, cinco millones de líneas escritas en EJB 1.1 y Hibernate a Spring JPA y servicios REST, es casi equivalente a rehacer la aplicación por completo. En un entorno de microservicios esto se podría hacer de manera incremental.

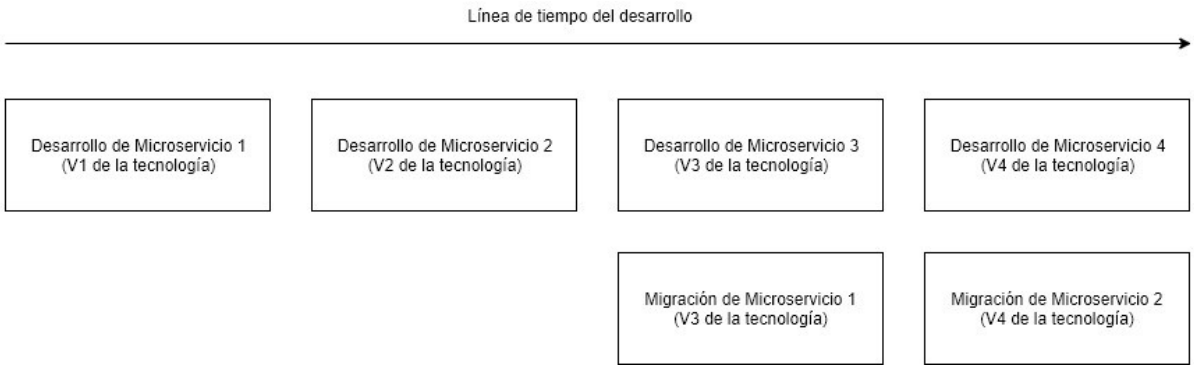


Ilustración 24 - Desarrollo paralelo e independiente de los microservicios

Tal y como se puede ver en la imagen, mientras que las versiones más antiguas de los servicios están funcionando con antiguas tecnologías, los nuevos servicios pueden implementarse con las tecnologías más actuales. El coste de la migración de microservicios con tecnologías obsoletas es considerablemente mucho menor en este tipo de arquitecturas que en arquitecturas monolíticas.

Permitir la coexistencia de diferentes versiones

Ya que el microservicio empaqueta el entorno de ejecución del servicio como parte de sí mismo, esto permite que múltiples versiones del mismo servicio coexistan en el mismo entorno.

Habrá situaciones en las que habrá que ejecutar múltiples versiones del mismo servicio al mismo tiempo. Una arquitectura en la que no puede haber tiempo de inactividad y, por lo tanto, se debe cambiar de una versión a otra sin interrumpir el servicio, es un ejemplo de escenario de este tipo en el que habrá un periodo de tiempo en el que ambos servicios, cada uno con su correspondiente versión, tendrán que estar funcionando al mismo tiempo. Con aplicaciones monolíticas, es un procedimiento muy complejo porque actualizar nuevos servicios en un nodo del cluster es muy difícil de manejar, y podría dar lugar, por ejemplo, a problemas de cargas de clases. Una liberación de versión, en la que la nueva versión se libera para unos pocos usuarios para que validen el nuevo servicio, es otro ejemplo donde múltiples versiones de los servicios deben coexistir.

Con los microservicios, ambos escenarios se pueden gestionar de una manera relativamente sencilla. Puesto que cada microservicio utiliza un entorno diferente, incluyendo un listener de servicios como Tomcat o Jetty embebidos, múltiples versiones pueden ser liberadas sin mayor complicación. Cuando un consumidor busca un servicio, buscará una versión específica.

Aun así, hay que prestar especial atención a nivel de base de datos para asegurarse que el diseño de la base de datos es siempre compatible hacía atrás para evitar romper con los cambios.

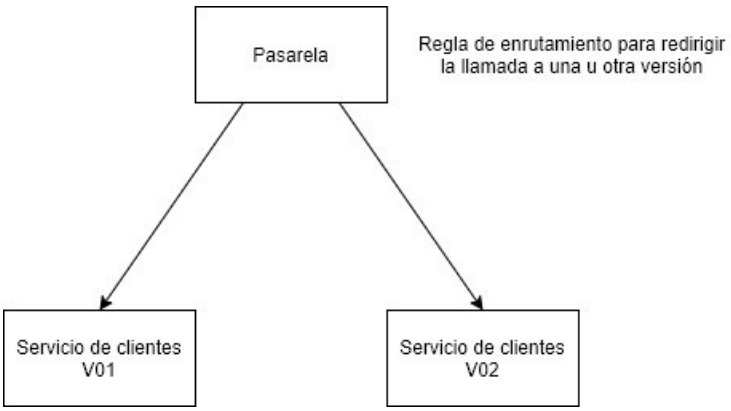


Ilustración 25 - Coexistencia de microservicios de distintas versiones

Tal y como se muestra en la imagen, las versiones 1 y 2 de servicio de cliente pueden coexistir y no interfieren entre ellos, ya que cada uno tiene su propio entorno de ejecución. Las reglas de enrutamiento se pueden establecer en la pasarela para redirigir el tráfico a la instancia particular. Alternativamente, los clientes pueden solicitar versiones específicas como parte de la propia solicitud. En la imagen, la pasarela selecciona la versión en base a la región donde se ha originado la solicitud.

Permitir la construcción de sistemas auto organizados

Los microservicios nos ayudan a construir sistemas auto organizados. Un sistema auto organizado nos ayudará a automatizar los despliegues, ser resiliente, y exponer las capacidades para auto recuperarse y aprender.

En un sistema de microservicios bien construido, un servicio no es consciente del resto de los microservicios. Acepta un mensaje de una cola determinada y lo procesa. Al final del proceso puede ser que envíe otro mensaje que desemboque en la ejecución de otro microservicio. Esto nos permite eliminar cualquier servicio del ecosistema sin analizar el impacto en el conjunto de éste. En base a los datos de entrada y salida, los servicios se auto organizarán dentro del ecosistema. De esta manera, no se requiere ningún cambio adicional de código o servicio de orquestación. No hay un elemento central para controlar y coordinar los procesos.

Imaginemos un servicio de notificaciones que permanece a la escucha en una cola de entrada y envía notificaciones a un servidor de correo SMTP, tal y como muestra la siguiente figura:

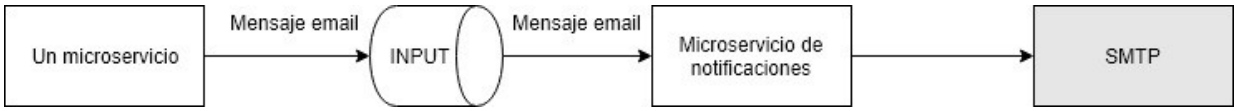


Ilustración 26 - Flujo de llamadas entre microservicios para el envío de correos electrónicos

Supongamos que más adelante, un sistema de personalización, responsable de cambiar el lenguaje de los mensajes al lenguaje nativo del cliente, necesita ser introducido para personalizar los mensajes antes de enviarlos al cliente. En este escenario, el sistema de personalización es el responsable de cambiar el lenguaje de los mensajes al lenguaje nativo del cliente:

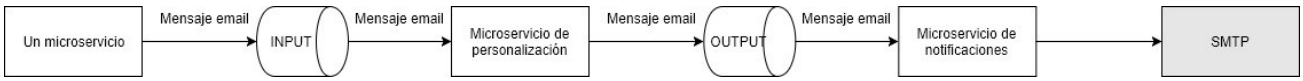


Ilustración 27 - Flujo de llamadas entre microservicios para el envío de correos electrónicos con un microservicio de personalización intermedio

En una arquitectura de microservicios, un nuevo microservicio de personalización se creará para hacer este trabajo. La cola de entrada será configurada como entrada de un servidor de configuración externo, y el servicio de personalización recogerá los mensajes de la cola de entrada (la misma que anteriormente utilizaba el servicio de notificaciones) y enviará los mensajes a la cola de salida después de completar el proceso. La cola de entrada del servicio de notificación la enviará entonces a la salida. Y desde este momento, el sistema adopta automáticamente este nuevo flujo para los mensajes.

Soporta la arquitectura dirigida por eventos

Los microservicios nos permiten desarrollar sistemas software transparentes. Los sistemas tradicionales se comunican entre sí por medio de protocolos nativos, y, por lo tanto, se comportan como una aplicación de caja negra. Los eventos de negocio y los eventos de sistema, a menos que estén publicados explícitamente, son muy difíciles de comprender y analizar. Las aplicaciones modernas requieren datos para el análisis de negocio, para comprender el comportamiento de sistemas dinámicos, y analizar las tendencias de los mercados, y, por otro lado, también necesitan

responder a eventos en tiempo real. Los eventos es un mecanismo muy útil para la extracción de datos.

Un microservicio bien diseñado siempre trabaja con eventos, tanto para la entrada como para la salida. Y estos eventos pueden ser interceptados por cualquier servicio. Una vez capturados pueden ser utilizados en una gran variedad de casos de uso.

Por ejemplo, una empresa quiere ver la velocidad de los pedidos categorizados por el tipo de producto en tiempo real. En un sistema monolítico, necesitamos pensar como extraer estos eventos. Esto puede exigir cambios en el sistema.

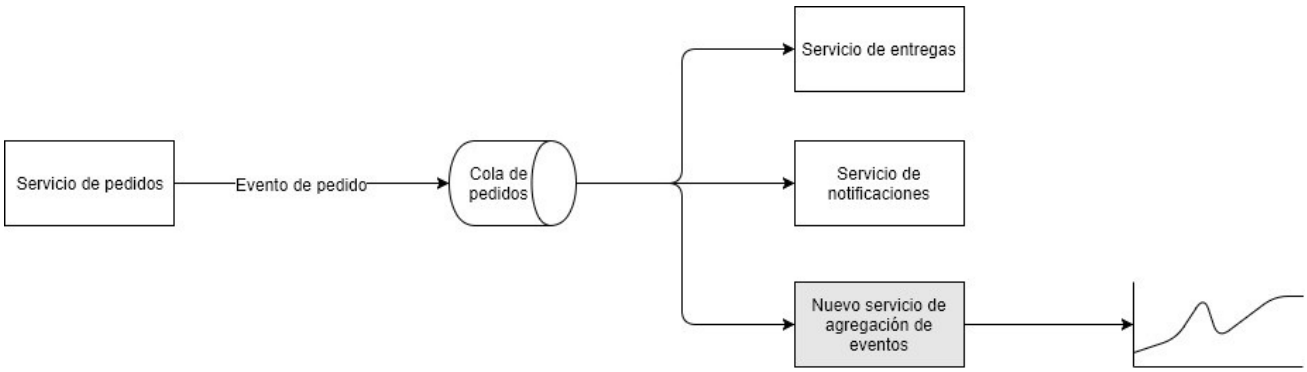


Ilustración 28 - Ejemplo de arquitectura dirigida por eventos

En el mundo de los microservicios, el evento de pedido es generado y publicado siempre que se crea un pedido. Esto significa que es sólo cuestión de añadir un nuevo servicio para suscribirse al tópico, extraer el evento, llevar a cabo las agregaciones solicitadas, y lanzar otro evento para que sea consumido por el panel de estadísticas.

Permite DevOps

Los microservicios son una de las claves que hace posible DevOps. DevOps está ampliamente aceptado como una práctica en muchas empresas, fundamentalmente para incrementar la velocidad y agilidad de entrega. Una adopción satisfactoria de DevOps requiere cambios culturales, cambios de procesos, además de cambios en la arquitectura. DevOps propone tener desarrollo ágil, ciclos de entrega de alta velocidad, pruebas automáticas, aprovisionamiento de infraestructura automática, y despliegues automáticos.

Automatizar todos estos procesos es tremendamente duro de conseguir con aplicaciones tradicionales monolíticas. Los microservicios no son una de las últimas respuestas, sin embargo, están en el centro de muchas implementaciones DevOps. Muchas herramientas DevOps y técnicas están también asociadas con el uso de microservicios.

Condiremos una aplicación monolítica que tarda horas en completar un “build” completo y de 20 a 30 minutos en arrancar la aplicación. Uno pude darse cuenta rápidamente que este tipo de aplicación no es ideal para la automatización de DevOps. Es muy difícil automatizar la integración continua en cada actualización del código. Debido a su tamaño, las aplicaciones monolíticas no son ideales para la automatización, y las pruebas y despliegues en un entorno de integración continua son también muy difíciles de lograr.

Por otro lado, los microservicios pequeños son mucho más fáciles de automatizar y, por lo tanto, pueden permitir estos requisitos con mucha más facilidad. Los microservicios también permiten ser más pequeños, enfocados en equipos ágiles para su desarrollo. Los equipos pueden estar organizados de acuerdo con los límites establecidos por los propios microservicios.

Desafíos

Desafíos técnicos

Los microservicios son sistemas distribuidos con llamadas entre los microservicios a través de la red. Esto puede impactar negativamente tanto en el tiempo de respuesta como en la latencia de los microservicios.

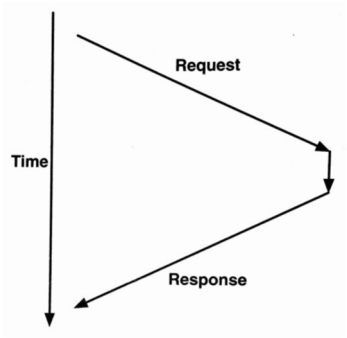


Ilustración 29 - Latencia entre microservicios a través de la red

La razón para esto queda justificada en la figura previa. Una llamada tiene que ir a través de la red hasta alcanzar los servicios, una vez allí la solicitud es procesada, y seguidamente devolver la respuesta al emisor de la llamada. La latencia sólo por la comunicación a través de la red puede ser alrededor de 0,5 milisegundos en un centro de computación. Dentro de este periodo de tiempo un procesador funcionando a 3Ghz puede llegar a procesar alrededor de 1,5 millones de instrucciones. Cuando la computación esta redistribuida a otro nodo, se debería comprobar si el procesamiento local de la solicitud podría llegar a ser más rápida. La latencia incluso se puede incrementar aún más debido a la transformación de los parámetros de entrada de la solicitud y los de respuesta. La optimización o conectar los nodos a la misma red puede mejorar esta situación.

La primera regla para objetos distribuidos y la precaución de ser conscientes de la latencia dentro de una red viene ya de tiempo atrás, cuando CORBA (Common Object Request Broker Architecture) y EJB (Enterprise JavaBeans) se utilizaron a comienzos de los 2000. Estas tecnologías se utilizaron como bastante frecuencia en arquitecturas distribuidas de tres capas. Por cada solicitud del cliente la capa web sólo devuelve el contenido HTML para construir la página. Por otro lado, la lógica reside en otro servidor, que es invocado a través de la red. Y los datos se almacenan en una base de datos, que generalmente también se sitúa en un servidor distinto. Cuando son sólo los datos los que tienen que ser mostrados, en la capa intermedia (EJB, CORBA) los datos no se procesan y simplemente se devuelven. Por razones de rendimiento y latencia, sería mucho mejor guardar la lógica en el mismo servidor que la capa web. Aunque, separar las capas entre distintos servidores permite el escalado de la capa intermedia, hacer esto en situaciones en las que la capa intermedia no tiene mucho que hacer, no hace que el sistema vaya más rápido.

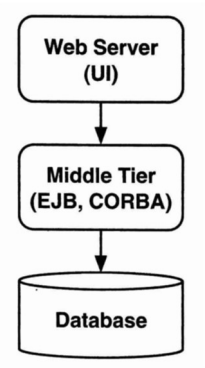


Ilustración 30 - Arquitectura en tres capas

Para los microservicios la situación es diferente, ya que el interfaz de usuario (UI) está en el propio microservicio. Las llamadas entre los microservicios solo tienen lugar cuando los microservicios necesitan la funcionalidad de otros microservicios. Si esta situación es frecuente, esto podría indicarnos que hay problemas en la arquitectura, ya que los microservicios deberían ser independientes unos de otros en la mayor medida posible.

En la realidad, las arquitecturas basadas en microservicios funcionan a pesar de los problemas relacionados con la distribución. Sin embargo, para mejorar el rendimiento y reducir la latencia, los microservicios no se deberían comunicar entre ellos demasiado

Dependencias de código

Una importante ventaja de las arquitecturas basadas en microservicios es la posibilidad de desplegar los servicios de manera individual. Sin embargo, esta ventaja puede desaparecer por las dependencias de código. Si una librería es utilizada por varios microservicios y aparece una nueva versión de esta librería que debería utilizarse, se podría necesitar un despliegue coordinado de varios microservicios; algo que, como principio básico, se debería evitar por todos los medios. Este escenario puede ocurrir con relativa facilidad por dependencias binarias donde diferentes versiones no son nunca más compatibles. El despliegue tiene que ser planificado de tal manera que todos los microservicios deberían ser desplegados en un cierto intervalo de tiempo y en un orden predefinido. La dependencia de código se tiene que cambiar también en todos los microservicios, un proceso que tiene que ser priorizado y coordinado en todos los equipos implicados. Una dependencia a nivel binario es una técnica de acoplamiento muy ceñida, que lleva a acoplamiento organizacional muy intenso.

Por lo tanto, los microservicios deberían adoptar la estrategia de no compartir nada, por la que los microservicios no tienen código compartido. Así pues, los microservicios deberían aceptar la redundancia de código y resistir la tentación de reutilizar código para evitar de esa manera un alto acoplamiento.

Las dependencias de código pueden ser aceptable en ciertas situaciones. Por ejemplo, cuando un microservicio ofrece una librería de cliente que ayuda los que los invocan a utilizarlos, y esto no tiene necesariamente consecuencias negativas. La librería dependerá del interfaz del microservicio. Si el interfaz es modificado de una manera compatible con la implementación, un cliente con una versión antigua de dicha librería podría seguir utilizando el microservicio. Los despliegues permanecen desacoplados. Sin embargo, la librería del cliente puede ser un punto de partida de una dependencia de código. Por ejemplo, si la librería del cliente contiene los mismos objetos de dominio, esto podría entrañar un problema. De hecho, si la librería del cliente contiene el mismo código para los objetos del dominio que el utilizado internamente, entonces los cambios al modelo interno afectarán a los clientes. Esto podría implicar tener que ser desplegado de nuevo. Si el objeto de dominio contiene lógica, esta lógica puede ser modificada cuando todos los clientes están desplegados igualmente de la misma manera. Esto también infringe el principio del despliegue independiente de los microservicios.

Comunicación poco confiable

La comunicación entre los microservicios tiene lugar en la red y es, por lo tanto, de poca confianza. Además, los microservicios individuales pueden fallar. Para asegurar que el fallo de un microservicio no da lugar al fallo del sistema entero, los microservicios que quedan en funcionamiento deben compensar el fallo y permitir que el sistema siga funcionando. Sin embargo, para conseguir este objetivo, la calidad de los servicios puede verse perjudicada, por ejemplo, utilizando valores por defecto o cacheados o limitar la funcionalidad utilizable.

Este problema no se puede resolver por completo en un nivel técnica. Por ejemplo, la disponibilidad de un microservicio puede ser mejorada utilizando hardware con alta disponibilidad. Sin embargo, este incremento en los costes no es una solución completa; en cierto modo, esto puede incrementar el riesgo. Si el microservicio falla a pesar de la alta disponibilidad del hardware y el fallo se propaga a través de todo el sistema, ocurriría un fallo completo. Por

lo tanto, los microservicios deberían compensar aun así el fallo de la alta disponibilidad de un microservicio.

Además, se sobrepasa el límite entre un problema técnico y un problema de dominio. Tomemos una ATM (Automated Teller Machine) como ejemplo. Cuando el ATM no puede recuperar el balance de la cuenta de un cliente, hay dos formas de enfrentarse a la situación. El ATM podría rechazar la retirada de dinero; que, aunque es una opción segura, provocaría molestias al usuario y una pérdida de beneficios. De manera alternativa, el ATM podría dar el dinero solicitado (quizás hasta un cierto límite). En cualquier caso, debería implementarse como una decisión de negocio. Alguien tiene que decidir si es preferible actuar de una manera segura, aunque esto supongas una molestia para el usuario, o correr ciertos riesgos y devolver demasiado dinero.

Pluralismo de tecnologías

La libertad de elección y uso de tecnologías en una arquitectura de microservicios puede dar lugar a que en un proyecto se utilicen distintas y muy variadas tecnologías. Los microservicios no necesitan tener tecnología compartida; sin embargo, la falta de tecnología común puede dar lugar a incrementar la complejidad del sistema en su conjunto. Cada equipo decide las tecnologías que se van a utilizar en su propio microservicio. Sin embargo, el gran número de tecnologías y estrategias utilizadas pueden provocar que el sistema puede alcanzar un nivel de complejidad que ningún miembro desarrollador ni equipo pueden llegar a entenderlo en su conjunto. Aunque por lo general un conocimiento del proyecto como un todo no necesario en la medida que cada equipo sólo entiende su propio microservicio. No obstante, cuando se hace necesario analizar y entender el sistema en su conjunto, esta complejidad puede llegar a ser un problema. En esta situación, la unificación pueden ser una medida razonable. Esto no significa que todas las tecnologías utilizadas tengan que ser exactamente las mismas, sino más bien que ciertas partes deberían ser uniformes o que ciertos microservicios de manera individual se deberían comportar de la misma manera.

Islas de datos

Los microservicios abstraen su propio almacenamiento local transaccional, que es utilizado para sus propios propósitos transaccionales. El tipo de almacenamiento y la estructura de datos será optimizado para los servicios ofrecidos por el microservicio.

Por ejemplo, si queremos desarrollar un gráfico de relaciones entre clientes, podemos utilizar una base de datos gráfica como Neo4j, OrientDB, o alguna otro por el estilo. Una búsqueda de texto predictiva para encontrar un cliente en base a cualquier información suya asociada (dirección, correo electrónico, teléfono…) se podría llevar a cabo utilizando una base de datos de búsqueda indexada como Elasticsearch o Solr.

Esto dará lugar a situación única de información fragmentada en islas heterogéneas de datos. Por ejemplo, cliente, puntos de fidelidad, reservas, y otros microservicios diferentes, utilizarán distintas bases de datos. El problema surge cuando queremos hacer análisis en tiempo real de datos que provienen de distintas fuentes de datos. Este escenario con una aplicación monolítica resultaba sencillo porque todos los datos estaban concentrados en una única base de datos.

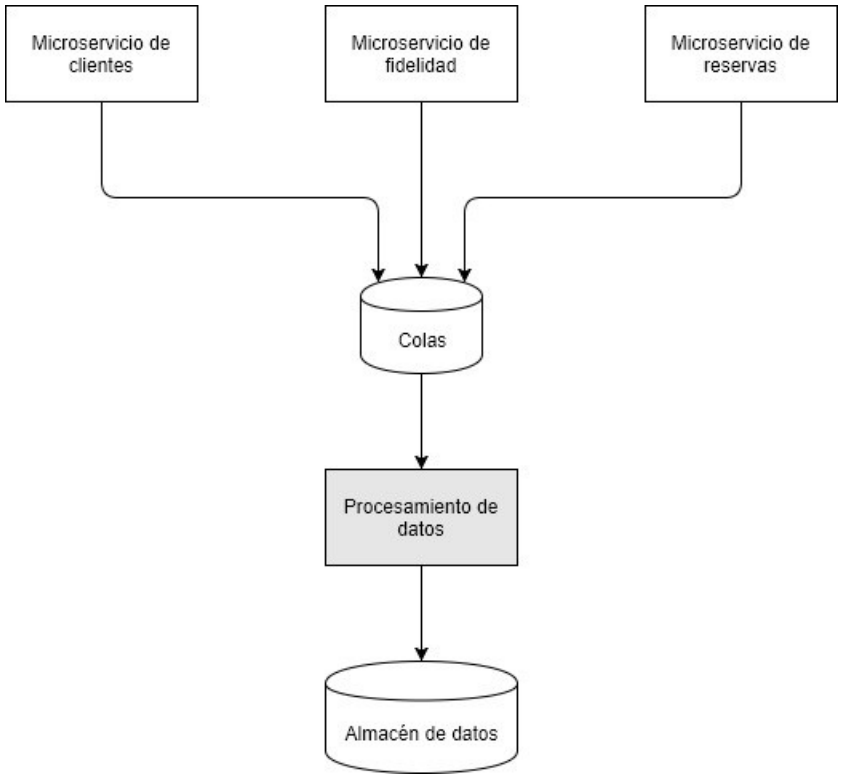


Ilustración 31 - Estructura de almacén y procesamiento de datos distribuidos en una arquitectura de microservicios

Para satisfacer este requisito, se requiere un almacén o un lago de datos. Almacenes tradicionales de datos como Oracle, Teradata, entre otros, se utilizan principalmente para los informes en lote. Pero con bases de datos NoSQL (como Hadoop) y técnica de microlotes, las analíticas casi en tiempo real resultan posibles con el concepto de lagos de datos. Al contrario que los tradicionales almacenes de datos que están construidos expresamente para el reporte de lotes, los lagos de datos almacenan los datos en estado puro sin asumir como se van a utilizar. Ahora la cuestión es cómo llevar los datos de los microservicios a estos lagos de datos.

Llevar los datos de los microservicios a los lagos o los almacenes de datos se puede hacer de muchas maneras. La tradicional ETL podría ser una de las opciones. Pero, en la medida que permitimos la entrada con ETL, y se rompe la abstracción, esta no se considera una forma efectiva de trasladar los datos. Una mejor forma de hacerlo sería enviar eventos desde los microservicios cuando estos ocurren, por ejemplo, el registro de un cliente, la actualización de los datos de un cliente… Y las herramientas de captura de datos consumen estos eventos y propagan el cambio de estado al lago de datos de manera apropiada. Las herramientas de captura de datos son plataformas altamente escalables, como Spring Cloud Data Flow, Kafka, Flume…etc.

Arquitectura

La arquitectura de un sistema basado en microservicios divide las partes de la funcionalidad basadas en el dominio entre los microservicios. Para entender la arquitectura a este nivel, se tienen que conocer las relaciones de dependencias y comunicación entre los microservicios. Analizar las relaciones de comunicación es difícil. Para grandes aplicaciones monolíticas hay herramientas que leen el código fuente o incluso ejecutables que pueden generar diagramas para visualizar los módulos y sus relaciones. Esto permite verificar la arquitectura implementada, y ajustarla hacía la arquitectura planeada, y seguir la evolución de dicha arquitectura a lo largo del tiempo. Este tipo de visión es fundamental para trabajar sobre la arquitectura; sin embargo, es muy difícil de conseguir cuando se utilizan microservicios por la falta de este tipo de herramientas.

Arquitectura = Organización

Un concepto clave para entender en lo que se basan los microservicios, es que la organización y la arquitectura son los mismo. Los microservicios se aprovechan de esta situación para implementar la arquitectura. La organización está estructurada de una manera que hace fácil la implementación de la arquitectura. No obstante, el inconveniente de esto es que la refactorización de una arquitectura puede exigir cambios en la organización. Esto provoca que los cambios en la arquitectura resulten mucho más difíciles. Esto no es sólo un problema exclusivo de los microservicios; la ley de Conway es aplicable a todos los proyectos. Por lo tanto, no utilizan la ley de una manera productiva y no pueden estimar los problemas organizacionales provocados por los cambios en la arquitectura.

Arquitectura y requisitos

La arquitectura también influye en el despliegue independiente de cada uno de los microservicios de manera individual y sus desarrollos independientes. Cuando la distribución de microservicios basada en el dominio no es óptima, los requisitos pueden afectar a más de un microservicio, y, por lo tanto, también a más de un equipo. Esto incrementa la coordinación exigida entre los diferentes equipos y microservicios. Esto, obviamente, influye negativamente en la productividad y tira por tierra una de las razones fundamentales para la introducción de microservicios.

Con los microservicios la arquitectura no sólo influye en la calidad del software, también en la de la organización y la habilidad de los equipos para trabajar de manera independiente de una manera productiva. Diseñar una arquitectura optima es incluso más importante en la medida que los posibles fallos tienen un alcance mayor.

Muchos proyectos no prestan la atención suficiente a la arquitectura de dominio, con mucha menos frecuencia incluso que a la parte técnica de la arquitectura. La mayoría de los arquitectos no tienen tanta experiencia con la arquitectura de dominio como con el diseño técnico de la arquitectura. Esta situación puede provocar problemas importantes en la implementación de propuestas basadas en microservicios. La separación de la funcionalidad en diferentes microservicios y, por lo tanto, en diferentes áreas de responsabilidad para diferentes equipos tiene que ser llevado a cabo de acuerdo con criterios de dominio.

Refactorización

Refactorizar un solo microservicio es una tarea directa en la medida que los microservicios son pequeños. Y, por este mismo motivo, pueden ser reemplazados o reimplementados con relativa facilidad.

Entre los microservicios la situación es diferente. Transferir funcionalidad de un microservicio a otro es difícil. La funcionalidad tiene que ser traslada a una unidad de despliegue diferente. Y esto es siempre más difícil que mover la funcionalidad dentro de la misma unidad. Las tecnologías pueden ser diferentes entre los microservicios. Los microservicios pueden utilizar distintas librerías e incluso diferentes lenguajes de programación. En estos casos, la funcionalidad debe ser implementada de nuevo en la tecnología del otro microservicio e incluida en éste. Esto, obviamente, es mucho más complejo que mover código dentro del mismo microservicio.

Arquitectura ágil

Los microservicios permiten que nuevos requisitos del producto se pueden entregar rápidamente al usuario final y que los equipos de desarrollo alcancen una velocidad de despliegue sostenible. Este es una gran ventaja cuando hay un gran número de requisitos y son muy difíciles de predecir. Este es el contexto perfecto para los microservicios. Los cambios a un microservicio son también muy simples. Sin embargo, ajustar la arquitectura del sistema, por ejemplo, trasladando funcionalidades, no es tan sencillo.

A menudo el primer intento de la arquitectura de un sistema no es la mejor solución. Durante la implementación los equipos aprenden mucho sobre el dominio. En un segundo intento, será mucho más viable llegar al diseño de una arquitectura apropiada. La mayoría de los proyectos que sufren una mala arquitectura tienen una buena idea de arquitectura en mente en base al estado del conocimiento del dominio en ese momento. Sin embargo, en la medida que el proyecto va avanzando, los requisitos que no se entendían bien llegan a estar más claros, y aparecen nuevos requisitos para los que la arquitectura inicial no encaja. El problema llega ser aún mayor cuando esto no lleva a un cambio de la arquitectura. Si el proyecto continúa con una arquitectura inapropiada, en algún momento esta arquitectura no encajará en absoluto y será completamente inviable. Esto se puede evitar ajustando la arquitectura poco a poco, adaptándola a los requisitos modificados en base al actual estado del conocimiento. La habilidad para cambiar y ajustar la arquitectura en línea con los nuevos requisitos es fundamental. Sin embargo, la habilidad para cambiar la arquitectura del sistema entero es una debilidad de los microservicios, que por separado resultan fácilmente modificables.

Infraestructura y operaciones

Se supone que los microservicios deberían llevarse a un entorno de producción de manera independiente entre ellos y deberían ser capaces de utilizar sus propias tecnologías. Por esta razón cada microservicio normalmente se aloja en su propio servidor. Esta es la única manera de asegurarse la independencia tecnológica completa. Con esta estrategia no es posible mantener el número de servidores requeridos usando servidores hardware. Incluso con virtualización la gestión de este tipo de entorno resulta muy difícil. Cuando hay cientos de microservicios, se requieren también cientos de máquinas virtuales, y para algunas de ellas, un balanceo de carga para distribuir el trabajo entre las distintas instancias. Esto requiere de automatización y la infraestructura apropiada que sea capaz de generar un gran número de máquinas virtuales.

Líneas de entrega continua

Más allá de lo requisitos de un entorno de producción, cada microservicio requiere infraestructura adicional; necesita su propia línea de entrega continua de manera que pueda llevarse a producción de una manera independiente a otros microservicios. Esto implica que son necesarios entornos de pruebas y scripts de automatización. El gran número de estas líneas de entrega conlleva desafíos adicionales. Cada una de estas distintas líneas tiene que ser construida y mantenida. Y para reducir costes, también se hace necesaria una amplia estandarización.

Monitorización

Cada microservicio necesita ser monitorizado de manera independiente. Esta es la única forma de diagnosticar posibles problemas con los microservicios en tiempo de ejecución. En una aplicación tradicional monolítica la monitorización del sistema es una cuestión relativamente sencilla. Cuando los problemas aumentan, el administrador puede logarse en el sistema y utilizar herramientas específicas para analizar los errores. Los sistemas basados en microservicios contienen tantos sistemas que esta forma de trabajar no es viable. Por lo tanto, debe haber algún sistema monitorización que recoja y centralice la información de monitorización de todos los servicios juntos. Esta información debería incluir, no sólo la típica información del sistema operativo y las entradas/salidas al disco duro y la red, también una visión de la aplicación en base a distintas métricas. Esta es la única manera para los desarrolladores de averiguar dónde están ocurriendo los problemas en un momento dado y en que punto la aplicación debe ser optimizada.

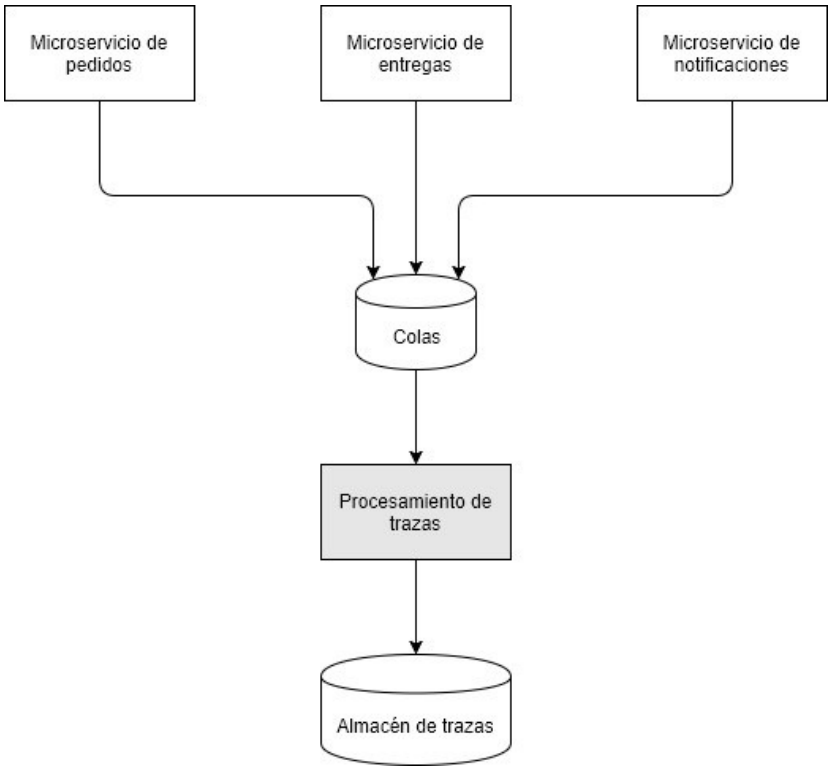


Ilustración 32 - Estructura de almacén y procesamiento de trazas distribuidas en una arquitectura de microservicios

Control de versiones

Cada microservicio tiene que ser almacenado bajo con un control de versiones independiente de otros microservicios. Sólo el software que es versionado de manera separada puede ser llevado a un entorno de producción de manera individual. Cuando dos módulos software son versionados conjuntamente, deberían igualmente ser llevados a producción de manera conjunta. Si no se hace esta manera, entonces un cambio podría afectar a ambos módulos, lo que implica que tendrían que ser entregados de nuevo. Además, si una versión antigua de uno de los servicios está en producción, no está claro si hace falta una actualización o si la nueva versión no contiene cambios; después de todo, la nueva versión podría tener cambios únicamente en el otro microservicio.

En el caso de despliegues de aplicaciones monolíticas, haría falta un número reducido de servidores, entornos y proyectos bajo control de versiones. Y esto reduce obviamente la complejidad. Los requisitos de operación e infraestructura en un entorno de microservicios son mucho mayores. Lidiar con esta complejidad es uno de los mayores retos cuando se trabaja con microservicios.

Probar los microservicios

Los microservicios presentan una dificultad añadida, la de probar los servicios. Para lograr una funcionalidad completa, un servicio puede ser que tengan que confiar en otro servicio, y este a su vez en otro servicio, ya sea síncrona o asíncronamente. El problema es como problema es como probar que una funcionalidad funciona desde el principio hasta el final, cuando ciertos servicios de los que depende pueden no estar disponibles en el momento de las pruebas.

La virtualización de servicios o simular los servicios mediante mocking es una de las técnicas utilizadas para probar servicios sin las dependencias reales. En las pruebas de entornos, cuando los servicios no están disponibles, los servicios mock simulan el comportamiento del servicio real. Los ecosistemas de microservicios necesitan la capacidad de virtualizar los servicios. Sin embargo, esto puede ser que no nos dé plena seguridad, ya que hay muchos casos particulares que la

simulación de servicios mediante mocks no llegan a cubrir, especialmente cuando hay profundas dependencias.

Otra estrategia sería utilizar un contrato dirigido por consumidor. Los casos de prueba de integración trasladados pueden llegar a cubrir en mayor o menor medida todos los casos de la invocación del servicio

La automatización de pruebas, las pruebas de rendimiento apropiadas, y las estrategias de entrega continua como las pruebas A/B, desarrollos azul-verde y rojo-negro, nos ayudan a reducir el riesgo de las puestas en marcha en un entorno de producción.

Contexto del problema

Comunicación asíncrona basada en mensajes

Cuando se utilizan sistemas de mensajería, los procesos se comunican mediante intercambio asíncrono de mensajes. Un cliente hace una solicitud a un servicio enviándole un mensaje. Si se espera que el servicio conteste, lo hace enviando un mensaje adicional de vuelta al cliente. Puesto que la comunicación es asíncrona, el cliente en vez de quedarse boqueado a la espera de una respuesta asume que la respuesta no se recibirá inmediatamente.

Un mensaje consiste en una cabecera (con información de metadatos como el remitente) y un cuerpo principal del mensaje. Los mensajes son intercambiados sobre canales. Cualquier número de productores pueden enviar mensajes a un canal. De la misma manera, cualquier número de consumidores puede recibir mensajes de un canal. Existen dos tipos de canales:

- Canal punto a punto, que envía un mensaje a un único consumidor que está leyendo del canal. Los servicios usan canales punto a punto para los estilos de iteración uno a uno.
- Canal de publicación/subscripción, que entrega cada mensaje a todos los consumidores suscritos al canal. Los servicios utilizan este tipo de canal para los estilos de iteración uno a muchos.

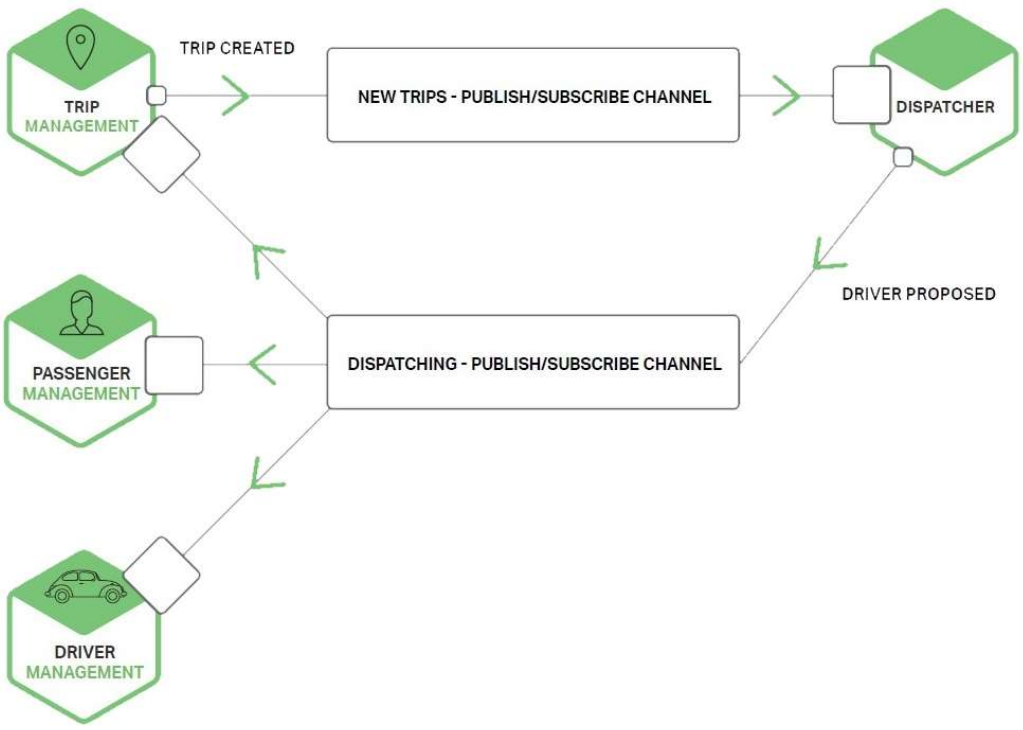


Ilustración 33 - Comunicación asíncrona en una arquitectura de microservicios

Tal y como se muestra en la imagen anterior, el servicio de gestión de viajes notifica a los servicios interesados, como el Dispatcher, un nuevo viaje escribiendo un mensaje de viaje creado a un canal de publicación/subscripción. El Dispatcher por su parte encuentra un conductor disponible y lo notifica a otros servicios, como el de gestión de conductores, escribiendo otro mensaje en un canal del mismo tipo.

Hay muchos sistemas de mensajería de los que poder elegir. Se debería elegir uno que soporte una gran variedad de lenguajes de programación. Algunos de estos sistemas de mensajería soportan protocolos estándar como AMQP y STOMP. Y los que no, están debidamente documentados.

Hay un gran número de sistemas de mensajería de libre distribución, como RabbitMQ, Apache Kafka, Apache ActiveMQ y NSQ. Todos ellos intentan dentro de sus posibilidades ser fiables, con un alto rendimiento y escalables.

Hay muchas ventajas para utilizar sistemas de mensajería:

- Desacopla el cliente del servicio. Un cliente hace una petición simplemente enviando un mensaje al canal apropiado. EL cliente no conoce las instancias del servicio que recibe dicha petición. Por lo tanto, no es necesario un mecanismo de descubrimiento de servicios para determinar la localización exacta de un servicio
- Almacenamiento de mensajes. Con un protocolo de petición/respuesta síncrona, como HTTP, tanto el cliente como el servicio deben estar disponibles durante la duración de la comunicación. Sin embargo, en una comunicación asíncrona el gestor de mensajes pone en cola los mensajes escritos en un canal hasta que el consumidor puede procesarlos. Esto significa, por ejemplo, que un almacén online puede aceptar peticiones de los clientes incluso cuando el sistema de gestión de pedidos sea lento o no esté disponible. Los mensajes de los pedidos simplemente se ponen en cola.
- Iteraciones cliente-servicio flexibles. Los sistemas de mensajería soportan todos los estilos de iteración posibles.
- Comunicación explicita entre procesos. Los mecanismos basados en RPC intentan funcionar de manera que se invoque un servicio remoto como si se tratase de un servicio local. Sin embargo, por motivos obvios y la posibilidad de fallos parciales, hay bastantes diferencias. Los sistemas de mensajes hacen que estas diferencias sean muy explicitas, por lo tanto, los desarrolladores no tienen una falsa sensación de seguridad.

Hay también, sin embargo, algunos inconvenientes al utilizar este tipo de comunicación por mensajes:

- Complejidad operacional adicional. El sistema de mensajes es otro componente adicional que debe ser instalado, configurado y gestionado. Es esencial que el gestor de mensajes tenga una alta disponibilidad, en caso contrario la fiabilidad del sistema se verá afectada.
- Complejidad de la implementación de las iteraciones basadas en petición/respuesta. El estilo de iteración petición/respuesta requiere un trabajo adicional para poder ser implementado. Cada mensaje de petición debe contener un identificador del canal de respuesta y un identificador de correlación. El servicio escribe un mensaje de respuesta con el identificador de correlación al canal de respuesta. El cliente utiliza el identificador de correlación para asociar la respuesta con la petición. Para esto, suele ser mucho más fácil utilizar un mecanismo IPC que directamente soporte este tipo de comunicación petición/respuesta.

Los microservicios y el problema de la gestión de datos distribuidos

Una aplicación monolítica por lo general tiene una única base de datos relacional. Un beneficio clave de usar una base de datos relacional es que la aplicación puede utilizar transacciones ACID, las cuales ofrecen importantes garantías:

- **Atomicidad (Atomicity)**. Los cambios se realizan de manera atómica
- **Consistencia (Consistency)**. El estado de la base de datos es siempre consistente
- **Aislamiento (Isolation)**. Aunque las transacciones se ejecuten concurrentemente, parece que son ejecutadas de manera secuencial
- **Duradero (Durable)**. Una vez que la transacción se ha terminado, no se deshace.

Como resultado, la aplicación puede simplemente empezar una transacción, realizar el cambio (insertar, actualizar o borrar) varias filas, y finalizar la transacción.

Otro gran beneficio de utilizar una base de datos relacional es que ofrece SQL, que es un rico lenguaje de consultas declarativo y estandarizado. Puedes fácilmente escribir una consulta que combina datos de múltiples tablas. El sistema de gestión de la base de datos relacional (RDBMS, Relational Database Management System) determina la manera óptima de ejecutar la consulta. No tienes que preocuparte sobre detalles de bajo nivel como la manera de acceder a la base de datos. Y, puesto que todos los datos de la aplicación están en una única base de datos, resulta fácil realizar una consulta.

Desafortunadamente, el acceso a los datos se hace más complejo cuando lo hacemos desde una arquitectura de microservicios. Ya que cada microservicio tiene sus propios datos, a los que sólo se puede acceder a través de su API. La encapsulación de los datos asegura que los microservicios están débilmente acoplados y pueden evolucionar independientemente unos de otros. Si varios servicios acceden a los mismos datos, las actualizaciones sobre el esquema no sólo consumen un tiempo, además exigen que estas actualizaciones se lleven a cabo de una manera coordinada entre los servicios.

Para hacer más complicada la situación, por lo general diferentes microservicios utilizan distintos tipos de bases de datos. Las aplicaciones actuales almacenan y procesan diversos tipos de información, y una base de datos relacional no es siempre la mejor elección. Para algunos casos de uso, una base de datos NoSQL podría ofrecer un modelo de datos más conveniente y ofrecer mejor rendimiento y escalabilidad. Por ejemplo, para un servicio que almacena y realiza consultas sobre texto tendría sentido utilizar un motor de búsqueda de texto como Elasticsearch. De la misma manera, un servicio que almacena datos de gráficos sociales debería utilizar probablemente una base de datos gráfica, como Neo4j. Por esta razón, las aplicaciones basadas en microservicios suelen utilizar una mezcla entre bases de datos SQL y NoSQL, también llamada estrategia de persistencia políglota.

Una arquitectura de persistencia políglota y particionada para almacenar datos tiene muchas ventajas, incluyendo el bajo acoplamiento entre los servicios y un mejor rendimiento y escalabilidad. Sin embargo, introduce algunos retos de gestión de datos distribuidos.

El primer reto es como implementar transacciones de negocio que mantengan la consistencia entre varios microservicios. Para ver de una manera más clara este problema, echemos un vistazo a un ejemplo de un almacén B2B online. El servicio de clientes mantiene información sobre los clientes, incluyendo sus líneas de crédito. El servicio de pedidos gestiona pedidos y debe verificar que un nuevo pedido no incumple el límite de crédito del cliente. En una versión monolítica de esta aplicación, el servicio de pedidos puede utilizar simplemente una transacción ACID para comprobar el crédito disponible y crear el pedido.

Por el contrario, en una arquitectura de microservicios las tablas con la información relativa a los pedidos y los clientes son privadas y propiedad de cada uno de sus correspondientes microservicios.

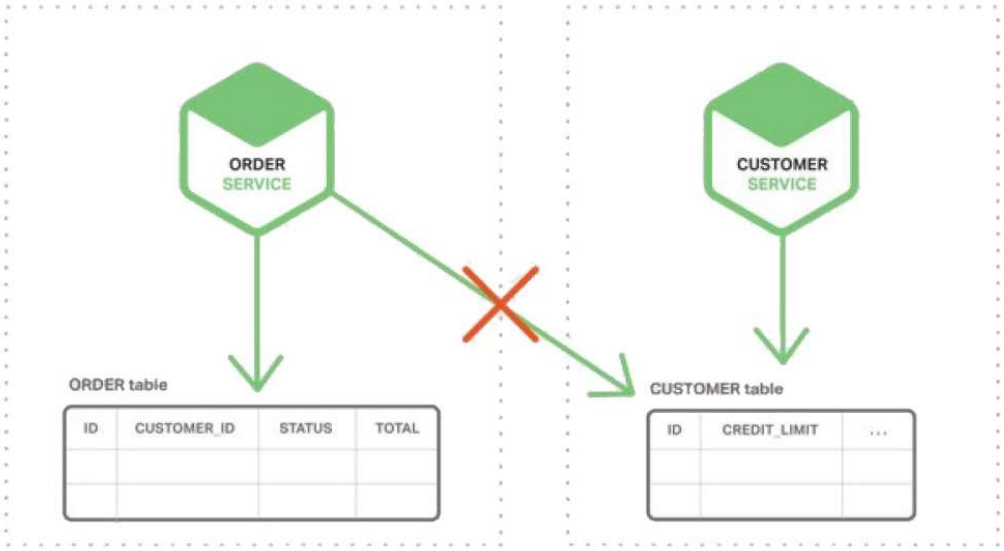


Ilustración 34 - Aislamiento de datos entre microservicios

El servicio de pedidos (Order Service) no puede acceder a la tabla con la información de los clientes directamente. Sólo puede utilizar el API expuesto por el servicio de clientes (Customer Service). El servicio de pedidos podría utilizar transacciones distribuidas, también conocidas como Two-phase commit (2PC). Sin embargo, normalmente 2PC no es una opción viable en las aplicaciones actuales. El teorema de CAP requiere que elijas entre disponibilidad y consistencia al estilo de ACID, y por lo general la disponibilidad es la mejor opción. Además, muchas tecnologías modernas, como la mayoría de bases de datos NoSQL, no soportan 2PC. Mantener la consistencia de los datos a través de los servicios y las bases de datos es fundamental, por lo tanto, necesitamos otra solución.

El segundo reto es cómo implementar las consultas que recuperan información de varios servicios. Por ejemplo, imaginemos que la aplicación necesita mostrar un cliente junto con sus pedidos recientes. Si el servicio de pedidos ofrece un API para recuperar los pedidos de un cliente entonces puedes recuperar esta información utilizando una join del lado de la aplicación. La aplicación recupera el cliente del servicio de clientes y los pedidos del cliente del servicio de pedidos. Sin embargo, supongamos que el servicio de pedidos sólo permite la búsqueda de pedidos por su clave primaria (quizás utiliza una base de datos NoSQL que sólo ofrecer recuperar datos en base a su clave primaria). En esta situación, no hay una manera obvia de recuperar la información necesaria.

Arquitectura dirigida por eventos

Para muchas aplicaciones, la solución es utilizar una arquitectura dirigida por eventos. En esta arquitectura, un microservicio publica un evento cuando ocurre algo importante, como, por ejemplo, cuando actualiza una entidad de negocio. Otro microservicio se subscribe a estos eventos. Cuando un microservicio recibe un evento puede actualizar sus propias entidades de negocio, lo cual podría llevar a que se publiquen más eventos.

Puedes utilizar eventos para implementar las transacciones de negocio que abarcan varios servicios. Una transacción consiste en una serie de pasos. Cada paso consiste en un microservicio que actualiza una entidad de negocio y publica un evento que lleva al siguiente paso. La siguiente secuencia de diagramas muestra cómo se puede utilizar una estrategia dirigida por eventos para comprobar el crédito disponible cuando se crea un nuevo pedido.

Los microservicios intercambiar eventos a través de un gestor de mensajes (Message Broker):

- El servicio de pedidos (Order Service) crea un nuevo pedido con el estado NEW y publica un evento de pedido creado

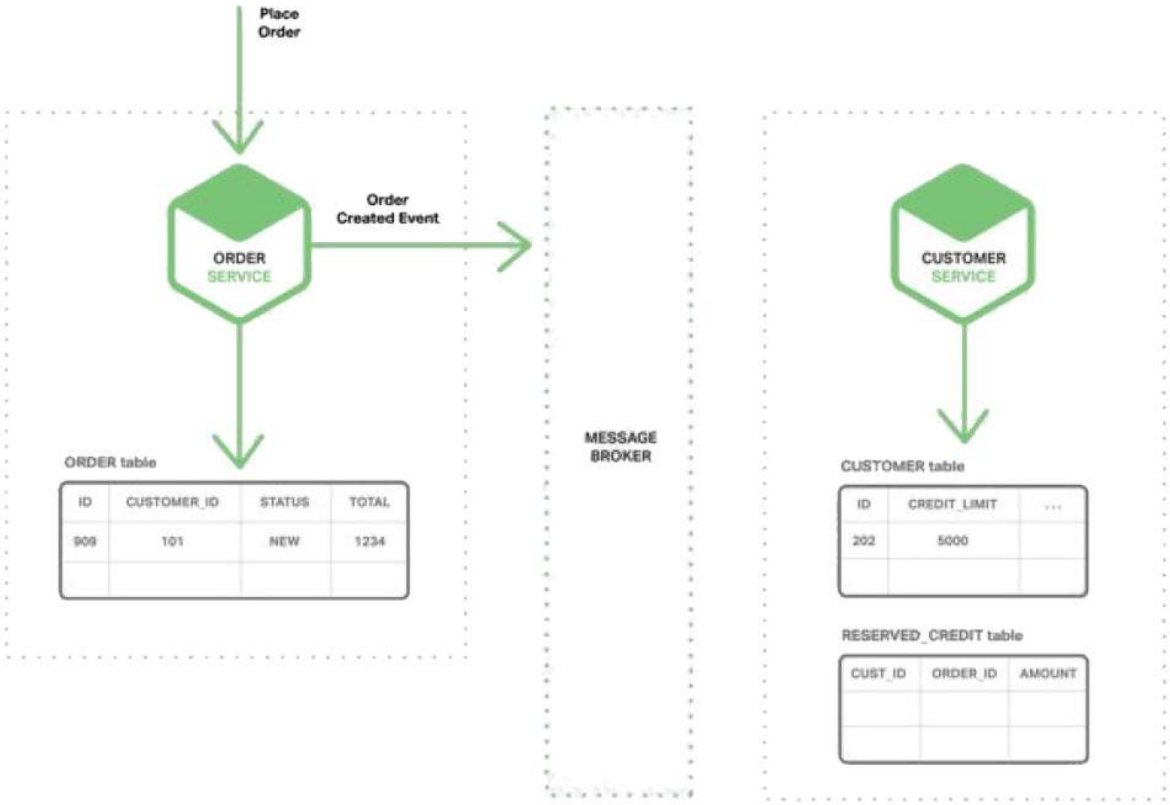


Ilustración 35 - Primer paso del proceso ejemplo de una arquitectura dirigida por eventos

- El servicio de cliente (Customer Service) consume el evento de pedido creado, reserva crédito para el pedido, y publica un evento de crédito reservado

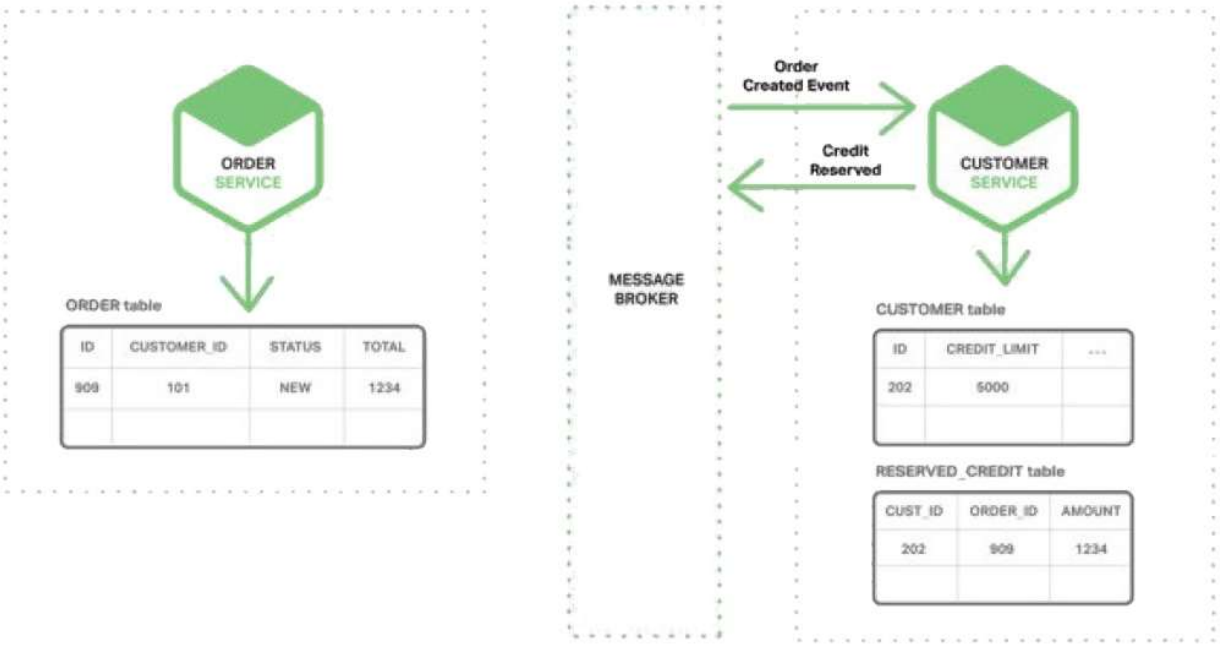


Ilustración 36 - Segundo paso del proceso ejemplo de una arquitectura dirigida por eventos

- El servicio de pedidos (Order Service) consume el evento de crédito reservado y cambia el estado del pedido a OPEN

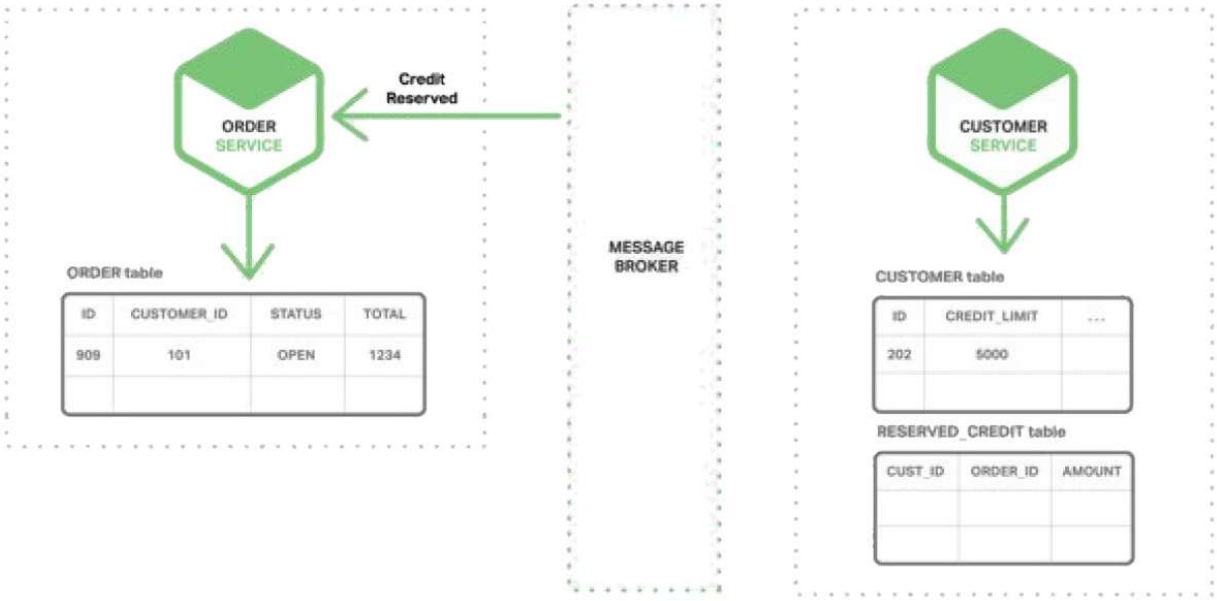


Ilustración 37 - Tercer paso del proceso ejemplo de una arquitectura dirigida por eventos

Un escenario más complejo podría incluir pasos adicionales, como un inventario de reservas al mismo tiempo que se comprueba el crédito del cliente.

Así pues, si por un lado cada microservicio actualiza su base de datos y publica un evento, y por otro el gestor de mensajes (Message Broker) garantiza que un evento es entregado al menos una vez, entonces podemos implementar transacciones de negocio que abarcan varios servicios. Es importante destacar que estas no son transacciones ACID. Ofrecen muchas menos garantías, como una eventual consistencia. Este modelo de transacción se conoce como el modelo BASE.

También se pueden utilizar eventos para mantener vistas materializadas que engloban datos procedentes de varios microservicios. El servicio que mantiene la vista se subscribe a los eventos relevantes y actualiza la vista. La siguiente figura muestra un servicio de actualización de la vista de los pedidos del cliente (Customer Order View Updater Service) que actualiza la vista de pedidos del cliente en base a los eventos publicados por el servicio de clientes y el servicio de pedidos.

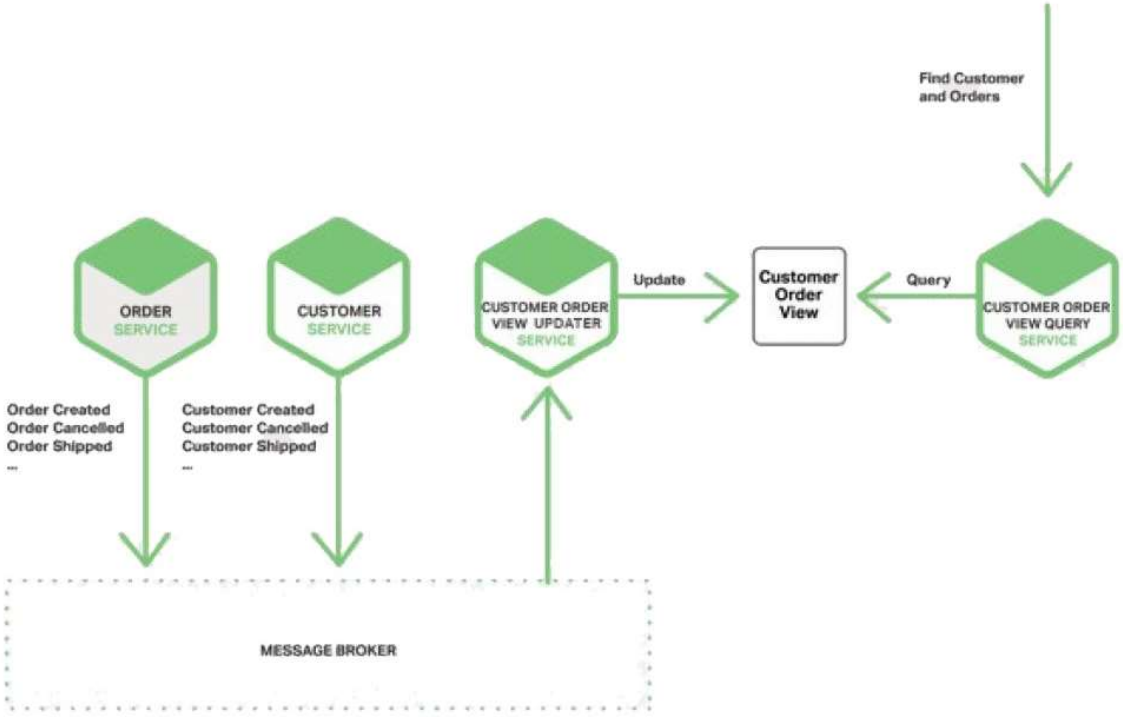


Ilustración 38 - Arquitectura de microservicios dirigida por eventos para mantener vistas compuestas por datos provenientes de distintos servicios

Cuando el servicio de actualización de la vista de pedidos del cliente recibe un evento de cliente o de pedido, entonces actualiza el almacén de la vista de pedidos del cliente. Se podría implementar la vista de pedidos del cliente mediante una base de datos documental como MongoDB y almacenar un documento por cliente. El servicio de consulta de la vista de pedidos del cliente (Customer Order View Query Service) atiende las peticiones sobre un cliente y sus pedidos recientes realizando una consulta al almacén de la vista de pedidos del cliente.

Una arquitectura dirigida por eventos tiene varias ventajas e inconvenientes. Permite la implementación de transacciones que abarcan varios servicios y aportan consistencia eventual. Otro beneficio es que también permite mantener vistas materializadas.

Una desventaja es que el modelo de programación es más complejo que cuando se utilizan transacciones ACID. Por lo general se deben implementar transacciones de compensación para recuperarse de fallos a nivel de aplicación; por ejemplo, se debe cancelar un pedido si la comprobación del crédito falla. También las aplicaciones deben tratar las inconsistencias de los datos. Esto es debido a que los cambios realizados por transacciones al vuelo son visibles. La aplicación también puede obtener inconsistencias si lee de una vista materializada que todavía no se ha actualizado. Otra desventaja es que los subscriptores deben detectar e ignorar eventos duplicados.

Logrando la atomicidad

En una arquitectura dirigida por eventos también existe el problema de actualizarla base de datos y publicar un evento de manera atómica. Por ejemplo, el servicio de pedidos debe insertar una fila en la tabla de pedidos y publicar un evento de pedido creado. Es esencial que

estas dos operaciones se realicen de manera atómica. Si el servicio deja de funcionar después de actualizar la base de datos y antes de publicar el evento, el sistema queda inconsistente. La manera estándar de asegurar la atomicidad es utilizar una transacción distribuida que involucre tanto la base de datos como el gestor de mensajes. Sin embargo, por las razones explicadas anteriormente, como el teorema de CAP, esto es exactamente lo que no queremos hacer.

Publicando transacciones locales utilizando eventos

Una manera de conseguir la atomicidad es publicar eventos utilizando un proceso de múltiples pasos involucrando sólo transacciones locales. La manera de hacerlo es tener una tabla de eventos, la cual funciona como una cola de mensajes, en la base de datos que almacena el estado de las entidades de negocio. La aplicación comienza como una transacción de base de datos (local), actualiza el estado de las entidades de negocio, inserta un evento en la tabla de eventos, y finaliza la transacción. Un proceso o un hilo de ejecución distinto consulta la tabla de eventos, publica el evento del gestor de mensajes (Message Broker), y entonces utiliza una transacción local para marcar el evento como publicado.

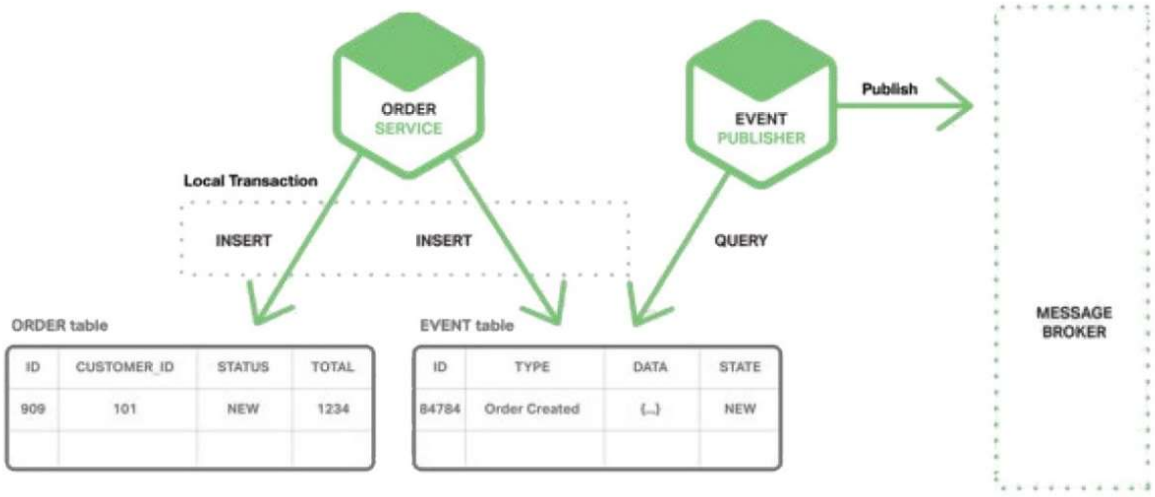


Ilustración 39 - Propuesta de solución al problema de atomicidad en comunicaciones asíncronas mediante transacciones locales

El servicio de pedidos inserta una fila en la tabla de pedidos e inserta un evento de pedido creado en la tabla de eventos. El publicador de eventos (Event Publisher) consulta en la tabla de eventos los eventos no publicados, publica los eventos, y entonces actualiza la tabla de eventos para marcar los eventos como publicados.

Esta estrategia tiene varios beneficios e inconvenientes. Un beneficio es que garantiza que un evento es publicado por cada actualización sin recurrir a 2PC. También, la aplicación publica eventos a nivel de negocio, que elimina la necesidad de inferirlos. Una desventaja de esta estrategia es que tiene cierta tendencia al error ya que el desarrollador debe recordar que tiene que publicar los eventos. Una limitación de este enfoque es su dificultad para ser implementada cuando se utilizan bases de datos NoSQL por sus capacidades de consulta y transacciones limitadas.

Este enfoque elimina la necesidad de 2PC al hacer que la aplicación use transacciones locales para publicar eventos y actualizar su estado.

Examinando el log de transacciones de la base de datos

Otra forma de lograr la atomicidad sin 2PC es que los eventos sean publicados por un hilo de ejecución o proceso que extraiga la información del log de transacciones de la base de datos. La aplicación actualiza la base de datos, y estos cambios son registrados en el log de transacciones de la base de datos. El proceso de del examinador del log de transacciones

(Transaction Log Miner) lee el log de transacciones y publica los eventos al gestor de mensajes (Message Broker).

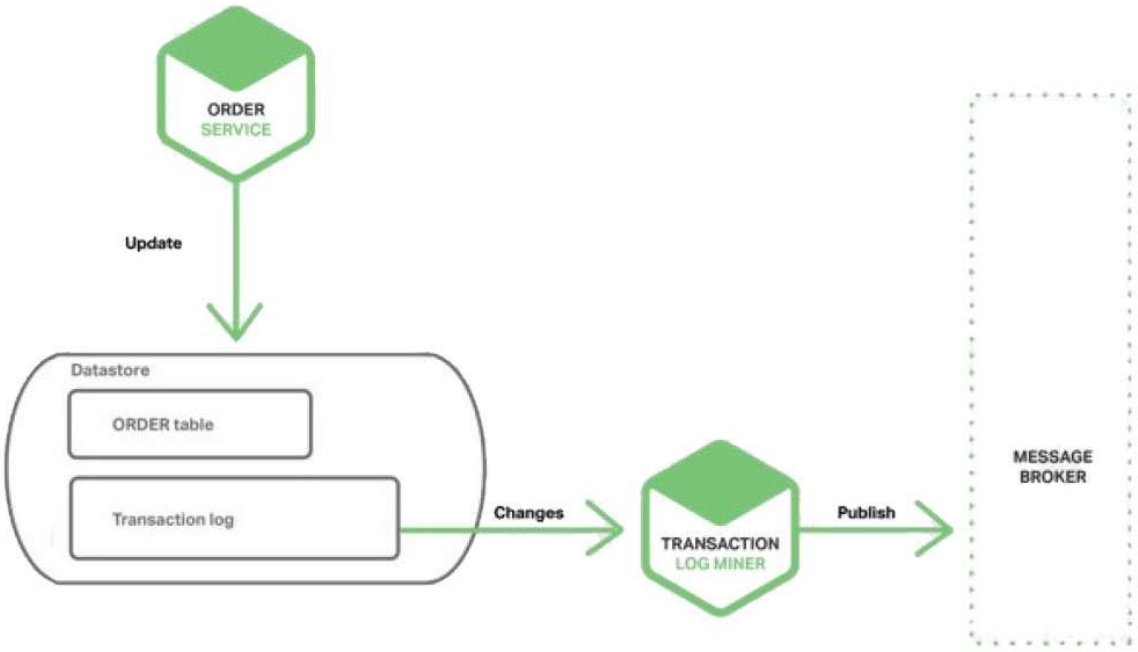


Ilustración 40 - Propuesta de solución al problema de atomicidad en comunicaciones asíncronas examinando el log de transacciones de la base de datos

Un ejemplo de este enfoque es el proyecto de código abierto LinkedIn Databus. Databus examina el log de transacciones de Oracle y publica los eventos correspondientes a los cambios. LinkedIn utiliza Databus para mantener varios almacenes de datos consistentes con el registro del sistema.

Otro ejemplo es el mecanismo de streams en AWS DynamoDB, que es una base de datos NoSQL gestionada. Un stream en DynamoDB contiene la secuencia de cambios ordenada en el tiempo (operaciones de creación, actualización y borrado) realizados en los ítems de una tabla de DynamoDB en las últimas 24 horas. Una aplicación puede leer los cambios del stream y, por ejemplo, publicarlos como eventos.

La extracción de información del log de transacciones tiene varios beneficios e inconvenientes. Uno de los beneficios es que garantiza que un evento es publicado para cada actualización sin utilizar 2PC. Este mecanismo también puede simplificar la aplicación separando la publicación de eventos de la lógica de negocio. Uno de los mayores inconvenientes es que el formato del log de transacciones viene definido por cada base de datos, e incluso puede cambiar entre versiones de la misma base de datos. Igualmente, puede resultar difícil hacer ingeniería inversa de los eventos de negocio de alto nivel desde las actualizaciones de bajo nivel registradas en el log de transacciones.

Este enfoque elimina la necesidad de 2PC, de manera que la aplicación sólo tiene que preocuparse por actualizar la base de datos.

Utilizando una fuente de eventos

Una fuente de eventos consigue la atomicidad sin 2PC utilizando un enfoque radicalmente diferente centrado en los eventos para las entidades de negocio persistentes. En vez de almacenar el estado actual de una entidad, la aplicación almacena una secuencia de eventos de cambio de estado. La aplicación reconstruye el estado actual de una entidad recorriendo estos eventos. Tan pronto como el estado de una entidad de negocio cambia, un nuevo evento es añadido a la lista

de eventos. En la medida que almacenar un evento es una operación única, podemos asegurar que es atómica.

Para ver cómo funciona esta estrategia supongamos la entidad pedido (Order) como ejemplo. En un planteamiento tradicional cada pedido corresponde con una fila en la tabla de pedidos.

Pero cuando utilizamos una fuente de evento, el servicio de pedidos almacena un pedido en la forma de eventos de cambio de estado: creado, aprobado, enviado, cancelado. Cada evento contiene suficiente información para reconstruir el estado del pedido.

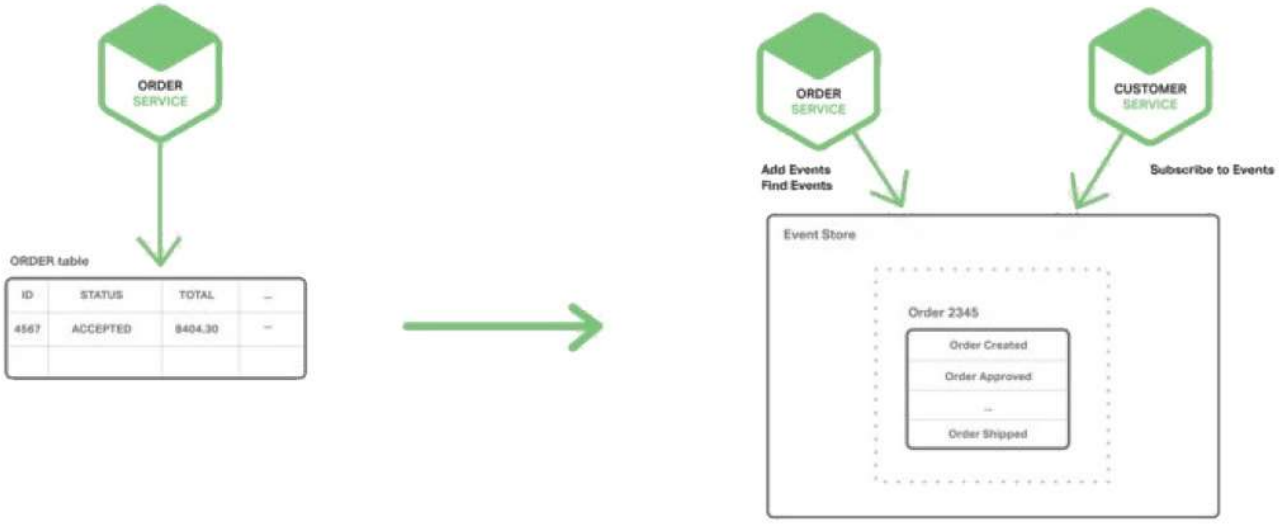


Ilustración 41 - Propuesta de solución al problema de atomicidad en comunicaciones asíncronas a través de una fuente de eventos

Los eventos se almacenan en un almacén de eventos, es decir, una base de datos de eventos. El almacén tiene una API para añadir y recuperar el evento de una entidad. El almacén de eventos también se comporta como un gestor de mensajes. Expone una API que permite a los servicios subscribirse a los eventos. De manera que este almacén de eventos entrega todos los eventos a los subscriptores interesados. Este componente es la pieza clave de una arquitectura de servicios dirigida por eventos.

Este enfoque tiene numerosos beneficios. Resuelve uno de los problemas clave en la implementación de una arquitectura dirigida por eventos y hace posible publicar eventos cuando el estado de una entidad cambia de manera fiable. Como resultado, se soluciona los problemas de consistencia de datos en una arquitectura de microservicios. Al mismo tiempo, porque persiste los eventos en vez de los objetos del dominio, evita sobre todo el problema de desajuste de impedancia de los objetos relacionales. Igualmente, permite un log de auditoria de los cambios realizados sobre las entidades de negocio totalmente confiable, y permite implementar consultas temporales para determinar el estado de una entidad en un momento determinado. Otro de los grandes beneficios es que la lógica de negocio está compuesta de entidades de negocio débilmente acopladas que intercambian eventos. Esto facilita la migración de una aplicación monolítica a una arquitectura de microservicios.

La estrategia planteada también presenta ciertos inconvenientes. Es un estilo de programación diferente y muy poco familiar, y por lo tanto tiene una curva de aprendizaje. Por otro lado, el almacén de eventos sólo permite la búsqueda de entidades de negocio a través de su clave primaria. Se debe utilizar CQRS (Command Query Responsability Separation) para implementar las consultas. Como resultado, la aplicación debe manejar datos consistentes de manera eventual.

Ejemplo de arquitectura de microservicios con transacciones distribuidas asíncronas

Con el objetivo de comprobar la problemática que entraña una comunicación asíncrona distribuida en el contexto de una arquitectura de microservicios, y poder a partir de ese punto desarrollar un componente que le dé solución, veamos la implementación y funcionamiento de un ejemplo concreto. Antes de nada, aclarar que, de cara a simplificar el ejemplo y no desviar la atención a detalles que escapan del alcance de este trabajo, el desarrollo planteado se centra fundamentalmente en la implementación y el mecanismo de comunicación asíncrona en un entorno distribuido dentro de una arquitectura de microservicios. Y en ningún caso de la propia lógica de negocio que se lleva a cabo, que por su sencillez resulta difícilmente aplicable a un caso empresarial real con procesos de negocio mucho más complejos.

Las tecnologías utilizadas son:

- Framework Spring Boot 1.5.6. Con los siguientes módulos adicionales:
  - Spring Boot Starter AMQP
  - Spring Boot Starter JPA
  - Spring Boot Starter Data Rest
  - Spring Data Rest Hal Browser
  - Spring Boot Starter HATEOAS
  - Spring Boot Starter Web
- Base de datos H2 1.4.196
- Apache Maven 3.5.0
- Java JDK 1.8.0_73
- RabbitMQ 3.6.10
- Erlang 20.0

Siguiendo con el modelo de ejemplo planteado en el apartado anterior, el sistema estará formado por dos microservicios, uno para la gestión de pedidos y otro para la gestión de clientes. Por definición estos microservicios estarán totalmente desacoplados, desarrollados de manera independiente y desplegables por separado. La principal y única funcionalidad que llevarán a cabo estos servicios será la creación de un pedido por parte del microservicio de pedidos, y su posterior validación o rechazo por parte del microservicio de gestión de clientes, de acuerdo con el crédito del que disponga el cliente que desea hacer el pedido. La manera de comunicarse entre ellos será por medio del middleware RabbitMQ como gestor de mensajería basado en el protocolo AMQP (Advanced Message Queuing Protocol). La arquitectura de este sistema quedaría definida de la siguiente manera:

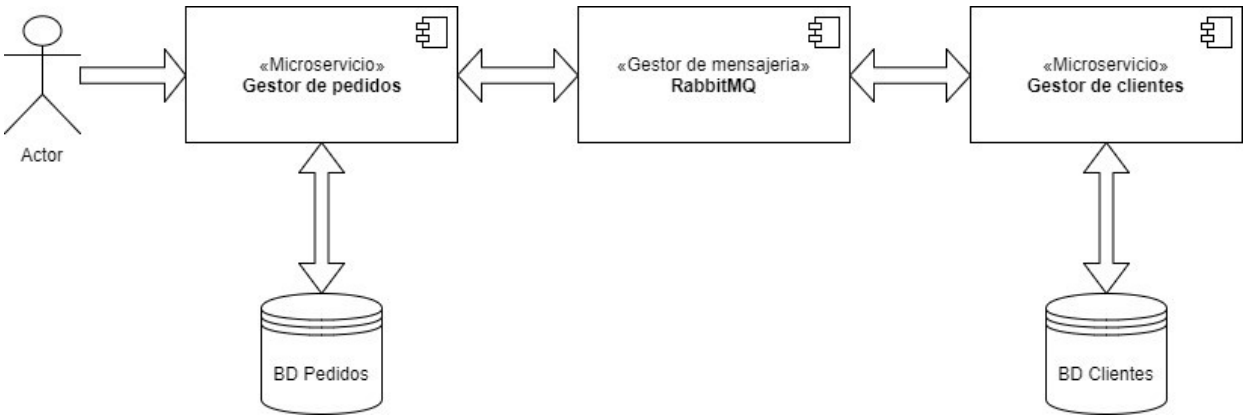


Ilustración 42 - Diagrama de componentes de la arquitectura de microservicios ejemplo

De una manera más detallada el proceso llevado a cabo entre los distintos componentes es el siguiente:

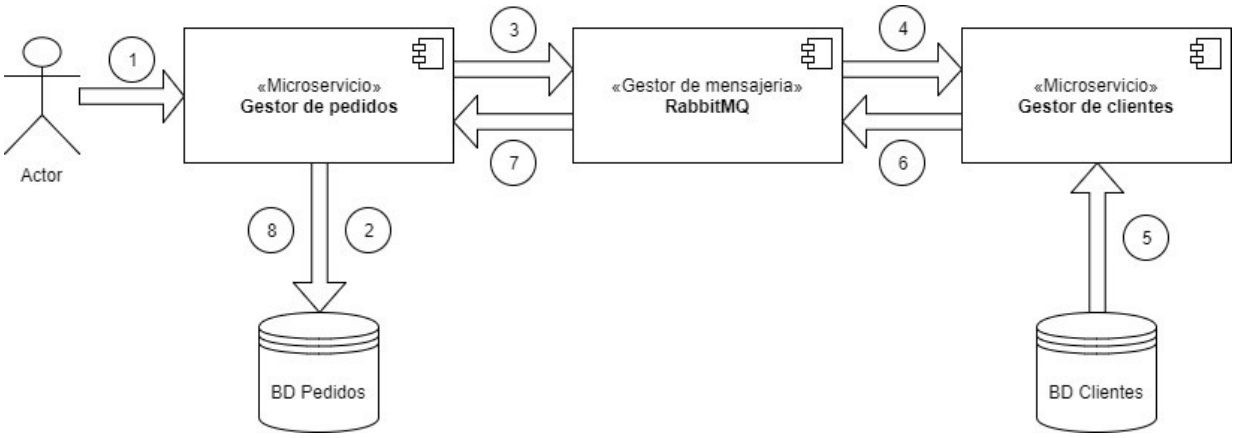


Ilustración 43 - Diagrama de funcionamiento de la arquitectura de microservicios ejemplo

1. El cliente, a través del interfaz (Web o REST) expuesta por el microservicio de pedidos, solicita la creación de un nuevo pedido con un importe asociado
2. El microservicio de pedidos guarda el pedido como nuevo (estado presumiblemente temporal), a la espera de su aceptación o rechazo por parte del microservicio de clientes
3. El microservicio de pedidos envía un mensaje de manera asíncrona a la cola del gestor de mensajes RabbitMQ donde está escuchando el microservicio de clientes, para comunicarle que se ha creado un nuevo pedido y que pueda realizar las validaciones correspondientes.
4. El microservicio de clientes recupera el mensaje enviado por el microservicio de pedidos
5. El microservicio de clientes recupera la información relativa al crédito máximo permitido para el cliente que ha solicitado el pedido, junto con el crédito que ya ha reservado para otros pedidos. Y con esta información hace las comprobaciones necesarias para aceptar o no el pedido.
6. El microservicio de clientes envía un mensaje de manera asíncrona a la cola del gestor de mensajes RabbitMQ donde ya estará escuchando el microservicio de pedidos, para comunicarle si el pedido ha sido aceptado o no mediante un nuevo estado.
7. El microservicio de pedidos recupera el mensaje enviado por el microservicio de clientes
8. El microservicio de pedidos actualiza el pedido en base de datos con el nuevo estado de aceptación o rechazo.

Toda la implementación completa de conjunto del sistema, tanto del microservicio de pedidos como el de clientes, está disponible para quien lo quiera consultar, o incluso mejorar, en el repositorio público de GitHub: https://github.com/franciscocilleruelo/ServicesTFM.git

Así mismo, el código fuente de las principales clases Java involucradas en la implementación de este capítulo se puede encontrar en el Anexo 1 – Código fuente del ejemplo de arquitectura de microservicios con transacciones distribuidas asíncronas

Veamos ahora por separado los detalles de cada uno de los componentes implicados en el sistema.

RabbitMQ/AMQP

Desde los primeros intentos de compañías como Sun, Oracle o IBM con JMS (Java Message Service) y Microsoft con MSMQ (Microsoft Message Queuing), todos los protocolos utilizados eran propietario. Más tarde, y gracias al equipo de JPMorgan, se creó el protocolo AMQP (Advanced Message Queuing Protocol). Es una capa de aplicación estándar y abierto para MOM (Message Oriented Middleware). En otras palabras, AMQP se puede utilizar con cualquier tecnología o lenguaje de programación.

Los gestores de mensajería compiten entre ellos para probar cual es más robusto, confiable y escalable, y, sobre todo, lo más importante, como son de rápidos. Y entre todos ellos, por su facilidad de uso, escalabilidad y velocidad, destaca por encima de todos ellos RabbitMQ que implementa el protocolo AMQP.

El protocolo AMQP define tres conceptos que son algo diferentes del contexto de mensajería JMS (Java Message Service); pero relativamente sencillos de entender. AMQP define exchanges, que son las entidades donde son enviados los mensajes. Cada una de estas entidades Exchange toma un mensaje y lo enruta hacia ninguna o muchas colas. Este enrutamiento implica un algoritmo que está basado en el tipo de intercambio y algunas reglas, llamadas bindings.

Este protocolo AMQP define cuatro tipos de exchanges: Direct, Fanout, Topic y Headers. La siguiente figura muestra estos tipos y su funcionamiento.

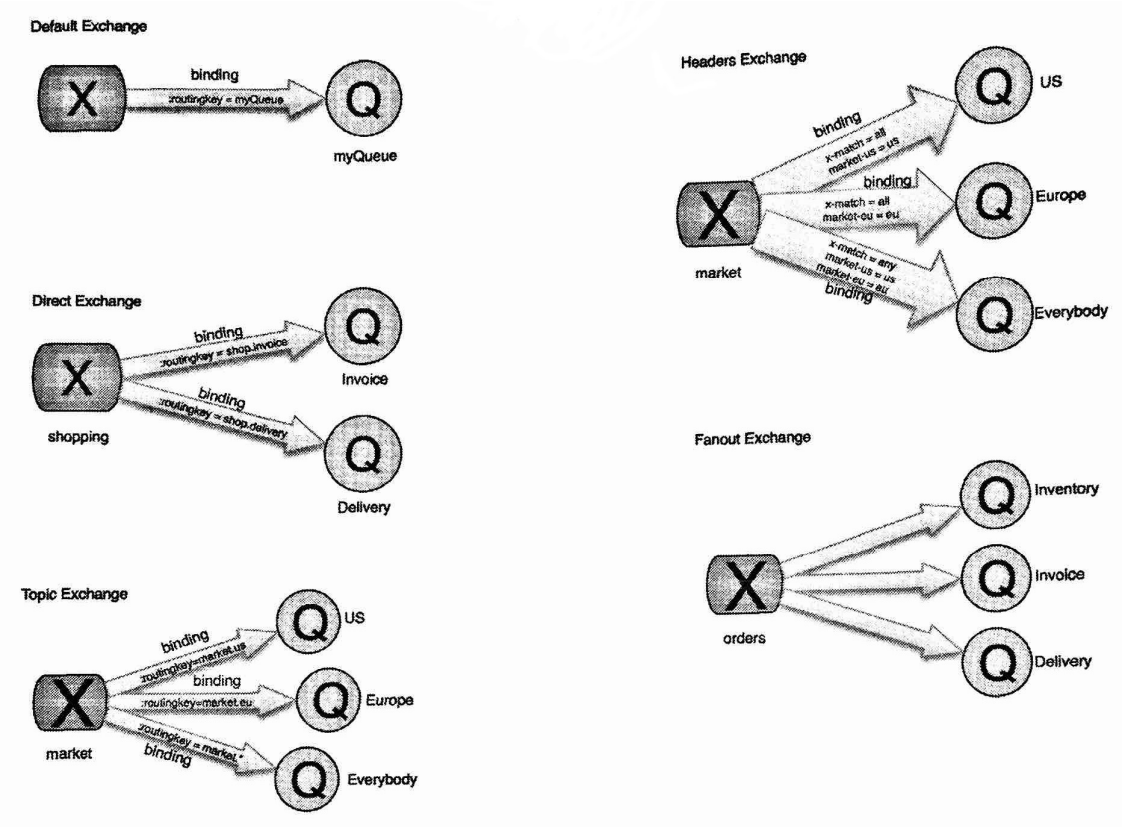


Ilustración 44 - Modos de intercambio de mensajes definidos por AMQP

Esta figura muestra los posibles tipos de exchange. Por lo tanto, la idea principal es mandar un mensaje a una entidad exchange, y entonces ésta mandará el mensaje a su cola correspondiente de acuerdo con su tipo. En el caso del tipo por defecto (default exchange) el envío se hará a todas las colas creadas, puesto que todas estarán enlazadas a él. El tipo directo (direct exchange) envía el mensaje a la cola asociada la clave de enrutamiento, se puede ver este tipo de envío como uno a uno. El topic exchange es similar al direct exchange, la única diferencia es que se puede añadir un comodín a clave de enrutamiento. El headers exchange es similar al topic exchange, la única diferencia es que la asociación entre la cola y el elemento Exchange se basa en las cabeceras del mensaje. Y, por último, el fanout exchange enviará el mensaje a todas las colas asociadas a él, se puede ver como un envío del mensaje broadcast.

Al instalar y arrancar el servidor de RabbitMQ, esta sería la apariencia de su página principal una vez autenticado en la URL (localhost:15672) y credenciales (guest/guest) por defecto:

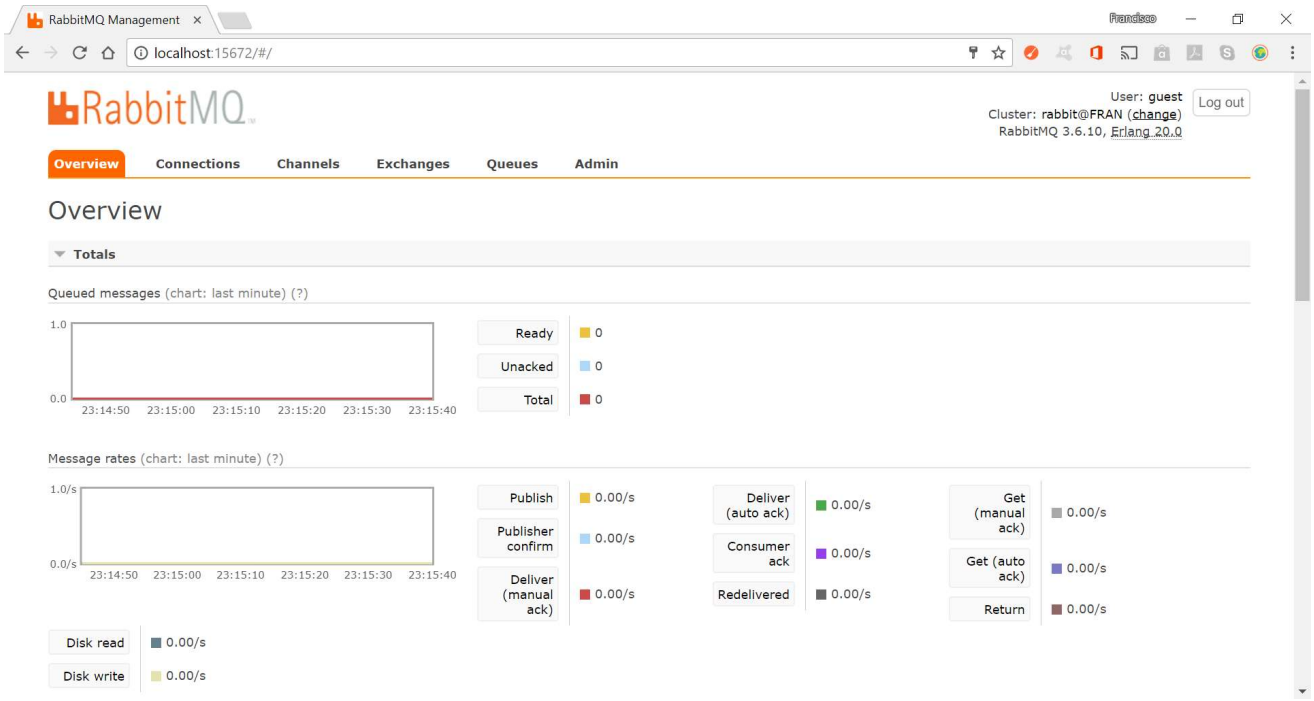


Ilustración 45 - Pantalla principal del interfaz web de RabbitMQ

Microservicio de pedidos

Por un lado, y como emisor y desencadenante de la comunicación tenemos el microservicio encargado de gestionar los pedidos, es decir, para este sencillo ejemplo, crear el pedido indicado por el usuario, comunicárselo al microservicio de clientes para comprobar su viabilidad y, finalmente, actualizar su estado. La arquitectura de este componente presentará la siguiente estructura.

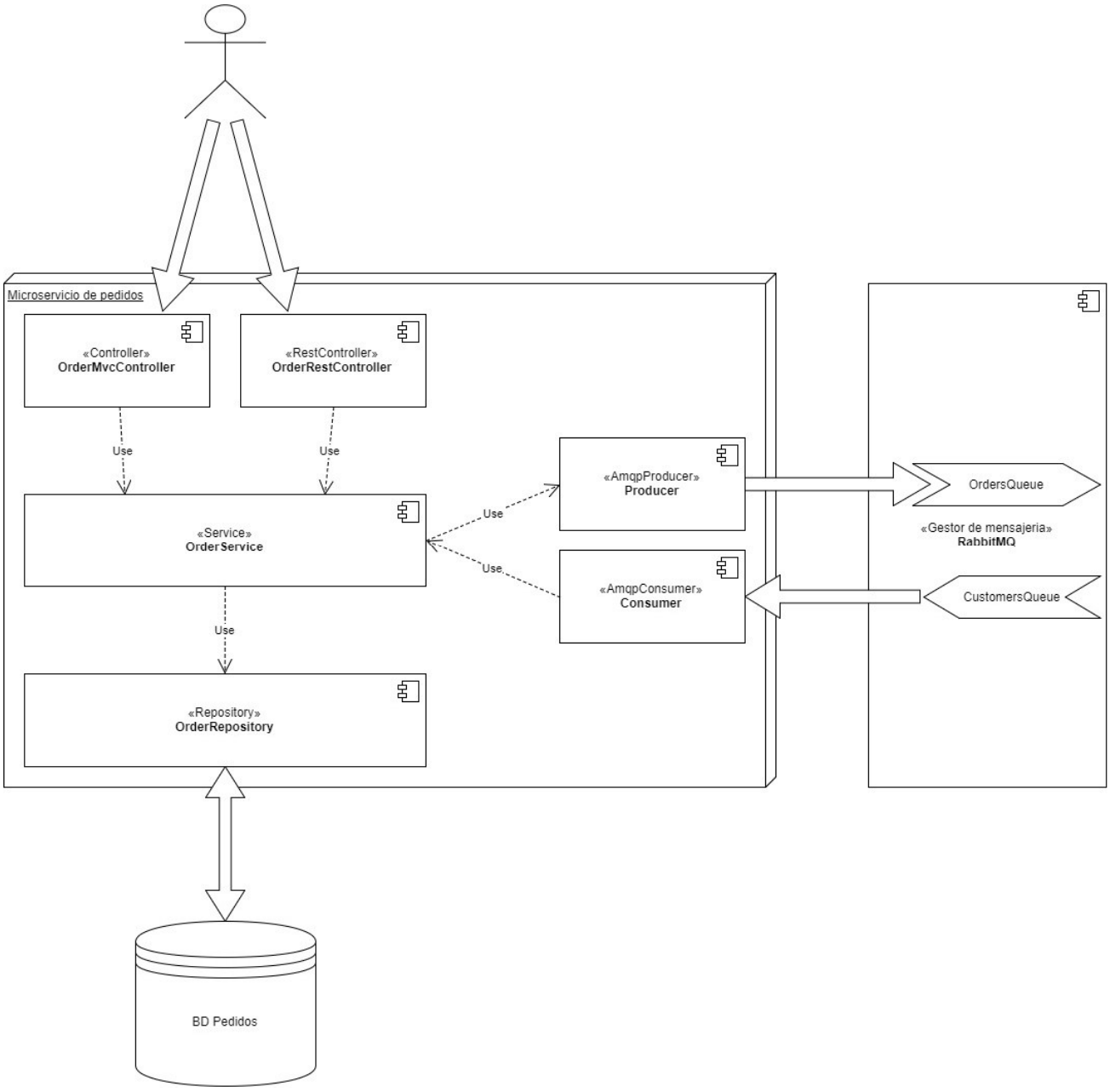


Ilustración 46 - Diagrama de componentes del microservicio de pedidos

El siguiente diagrama muestra la interacción entre estos componentes para poder llevar a cabo el proceso deseado.

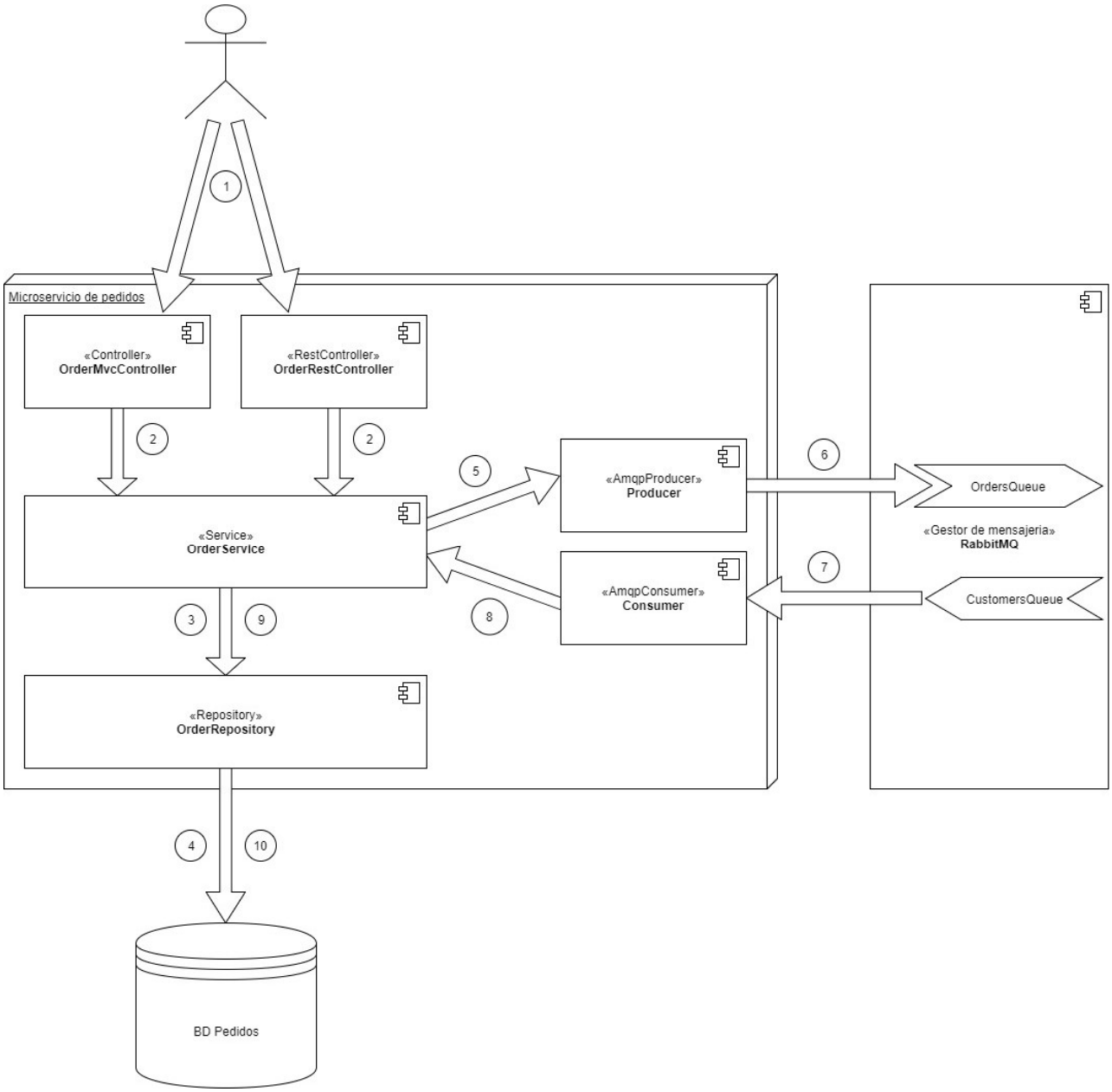


Ilustración 47 - Diagrama de funcionamiento del microservicio de pedidos

1. El usuario/cliente envía una solicitud para crear un pedido nuevo. Esta solicitud puede ser bien a través de un servicio REST expuesto a través del controlador REST (OrderRestContoller), o bien a través de una URL invocada desde formulario web HTML a través del controlador MVC genérico (OrderMvcController)
2. Independientemente del punto de entrada de la solicitud, el controlador correspondiente llama al componente de servicios (OrderService) encargado de la lógica de negocio. Que en este caso simplemente pondrá el pedido con estado nuevo ("NEW")
3. El servicio invoca a la componente de repositorio JPA (OrderRespository), encargado de interactuar con la base de datos, para persistir el pedido con ese estado inicial
4. El repositorio guarda el pedido en base de datos
5. El servicio llama al productor AMQP (Producer) asociado con el gestor de mensajería (RabbitQ) para enviar el mensaje correspondiente. En este caso el propio objeto pedido con sus atributos.
6. El productor de mensajes envía el objeto en formato JSON para que pueda ser debidamente convertido a la cola correspondiente (OrdersQueue) donde espera recibirlo el microservicio de clientes

7. Pasado un tiempo, presumiblemente, y en el caso de que todo haya ido bien en el lado del receptor, el consumidor AMQP (Consumer) de la cola de recepción (ConsumerQueue) recibe el mensaje/objeto de respuesta como una cadena JSON.
8. El consumidor invoca al componente de servicio (OrderService) para actualizar el estado del pedido de acuerdo con la respuesta recibida
9. El servicio llama al repositorio (OrderRepository) como mediador y gestor de las operaciones con la base de datos para que guarde el pedido
10. Finalmente, y si todo ha ido bien, el repositorio persiste el pedido con el estado indicado en la respuesta recibida.

El código fuente de las principales clases Java involucradas en la implementación de este microservicio se puede encontrar en el apartado Microservicio de pedidos como parte del Anexo 1 – Código fuente del ejemplo de arquitectura de microservicios con transacciones distribuidas asíncronas

Microservicio de clientes

Por otro lado, y como receptor y componente necesario para completar el sistema, nos encontramos con el microservicio de clientes, cuya función en este sencillo ejemplo será simplemente la de recibir el pedido creado desde el microservicio de pedidos y comprobar si es o no viable de acuerdo con el crédito del que dispone el usuario que lo ha creado. Este componente presentará la siguiente arquitectura.

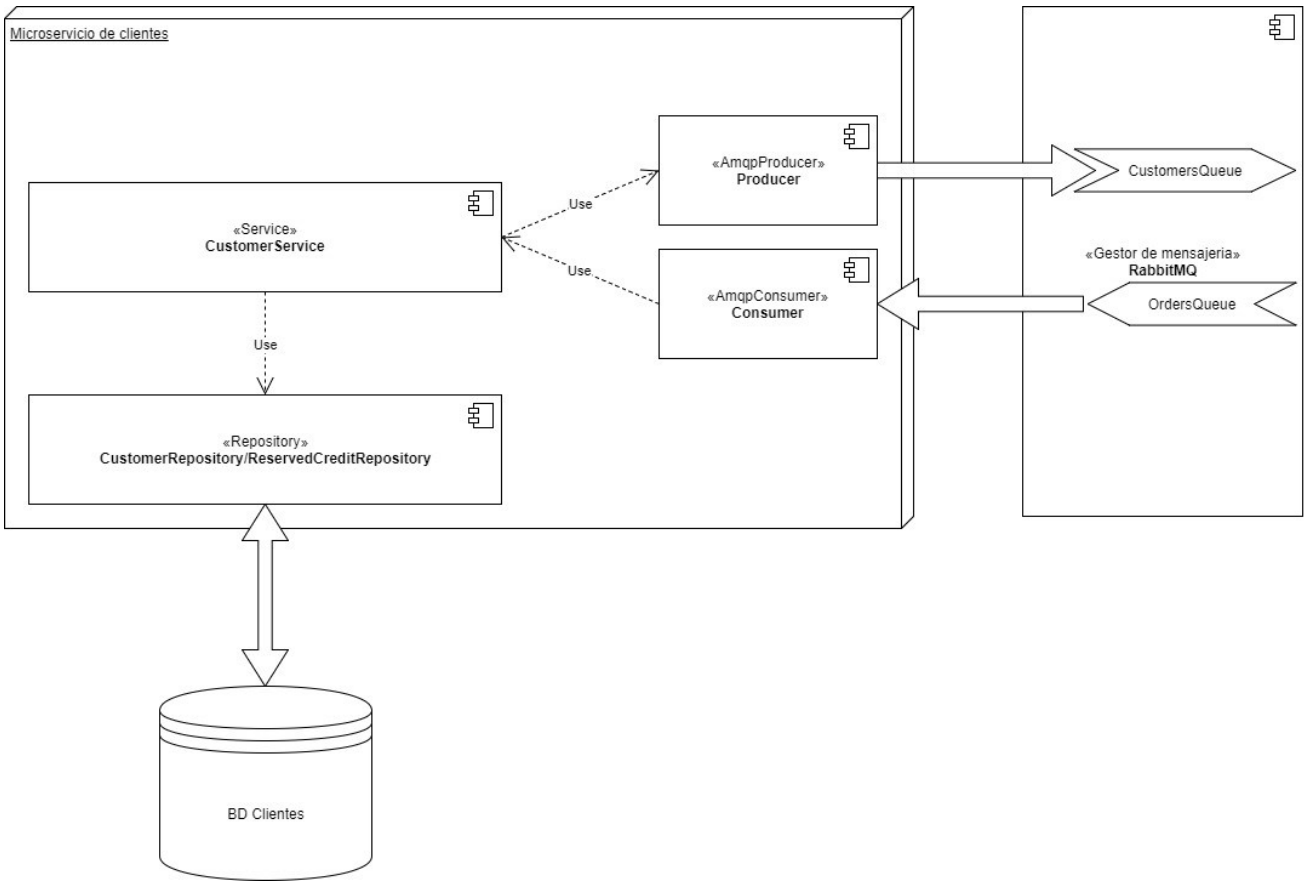


Ilustración 48 - Diagrama de componentes del microservicio de clientes

El siguiente diagrama muestra la interacción entre estos componentes para poder realizar el proceso deseado.

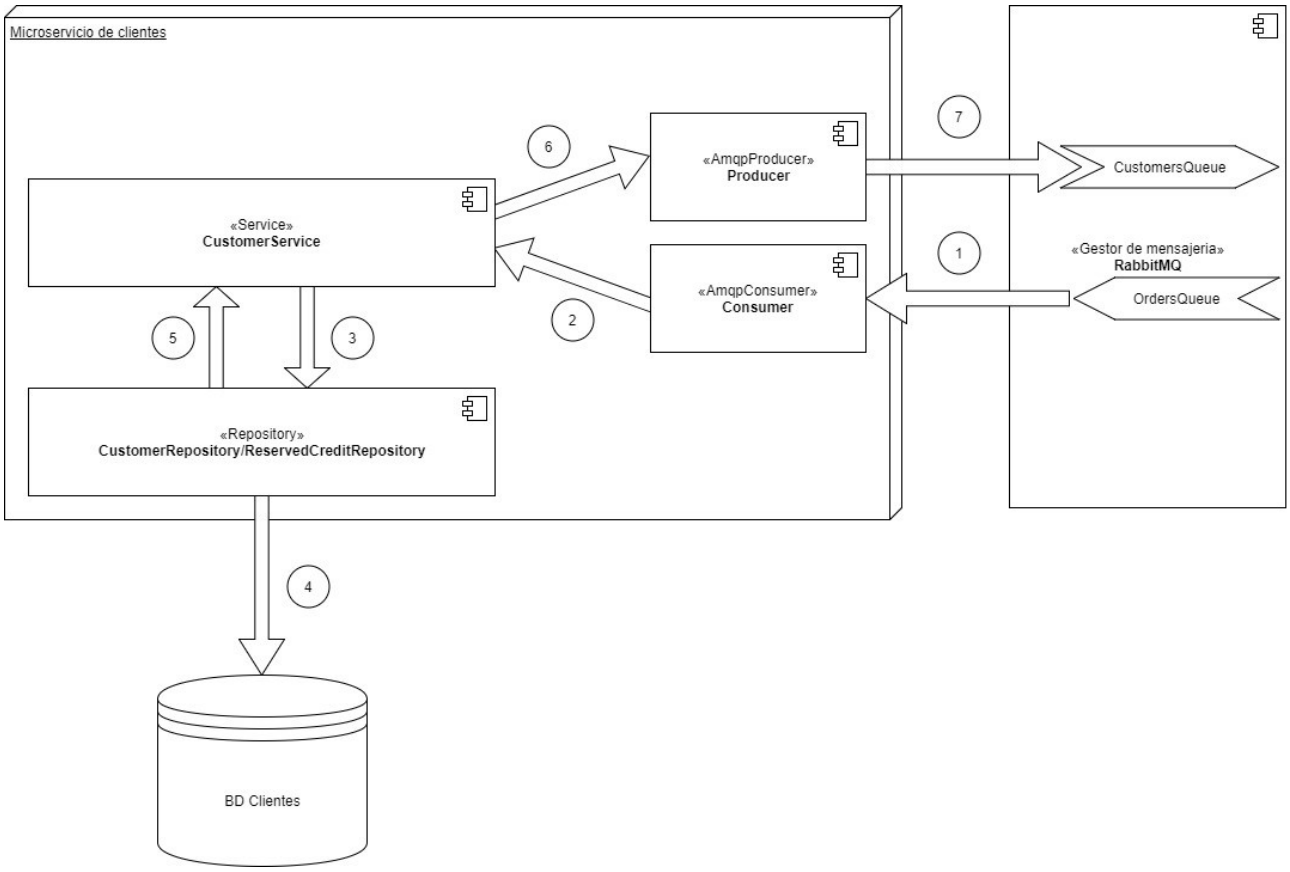


Ilustración 49 - Diagrama de funcionamiento del microservicio de clientes

1. El consumidor AMQP (Consumer) encargado de escuchar en la cola donde presumiblemente recibirá los mensajes enviados por el microservicio de pedidos, recibe un nuevo mensaje de éste como un objeto en formato JSON
2. El consumidor llama al servicio encargado de la lógica de negocio (CustomerService) para que compruebe si el pedido puede ser tramitado de acuerdo con la situación de crédito del cliente que lo solicita.
3. El servicio invoca a los repositorios (CustomerRepository y ReservedCreditRepository) para recuperar la situación de crédito del cliente que ha creado el pedido, es decir, su límite de crédito y la suma de créditos que ya ha solicitado previamente para otros pedidos
4. Los repositorios JPA solicitan la información requerida a la base de datos de clientes y ésta se la devuelve
5. El repositorio le devuelve la información solicitada al servicio
6. El servicio, después de comprobar si el pedido puede o no tramitarse y cambiar su estado a abierto ("OPEN") o rechazado ("REJECTED"), lo envía al productor AMQP (Producer) encargado de transmitir el mensaje a la cola del gestor de mensajes (RabbitMQ) correspondiente
7. El productor AMQP envía el objeto actualizado con su nuevo estado en formato JSON a la cola (OrdersQueue) donde espera recibirlo el microservicio de pedidos

El código fuente de las principales clases Java involucradas en la implementación de este microservicio se puede encontrar en el apartado Microservicio de clientes como parte del Anexo 1 – Código fuente del ejemplo de arquitectura de microservicios con transacciones distribuidas asíncronas

Funcionamiento conjunto

Una vez vista la estructura, implementación y modo de funcionamiento del sistema que utilizaremos como ejemplo de arquitectura de microservicios con comunicación asíncrona distribuida, pasemos a verlo en funcionamiento.

Debido a la propia naturaleza de los microservicios y su implementación con Spring Boot, cada uno de ellos se ejecutará y desplegará de manera independiente. Para este caso, el microservicio de pedidos en el puerto 8080 y el de clientes en el puerto 9090. Y de manera paralela se arrancará el servidor del gestor de mensajes RabbitMQ.

En el momento inicial del arranque, y antes de cualquier interacción por parte del usuario. En el gestor de mensajería quedan registradas las colas que utilizaremos para la comunicación (OrdersQueue y CustomersQueue), tal y como se puede comprobar en la consola de administración de RabbitMQ.

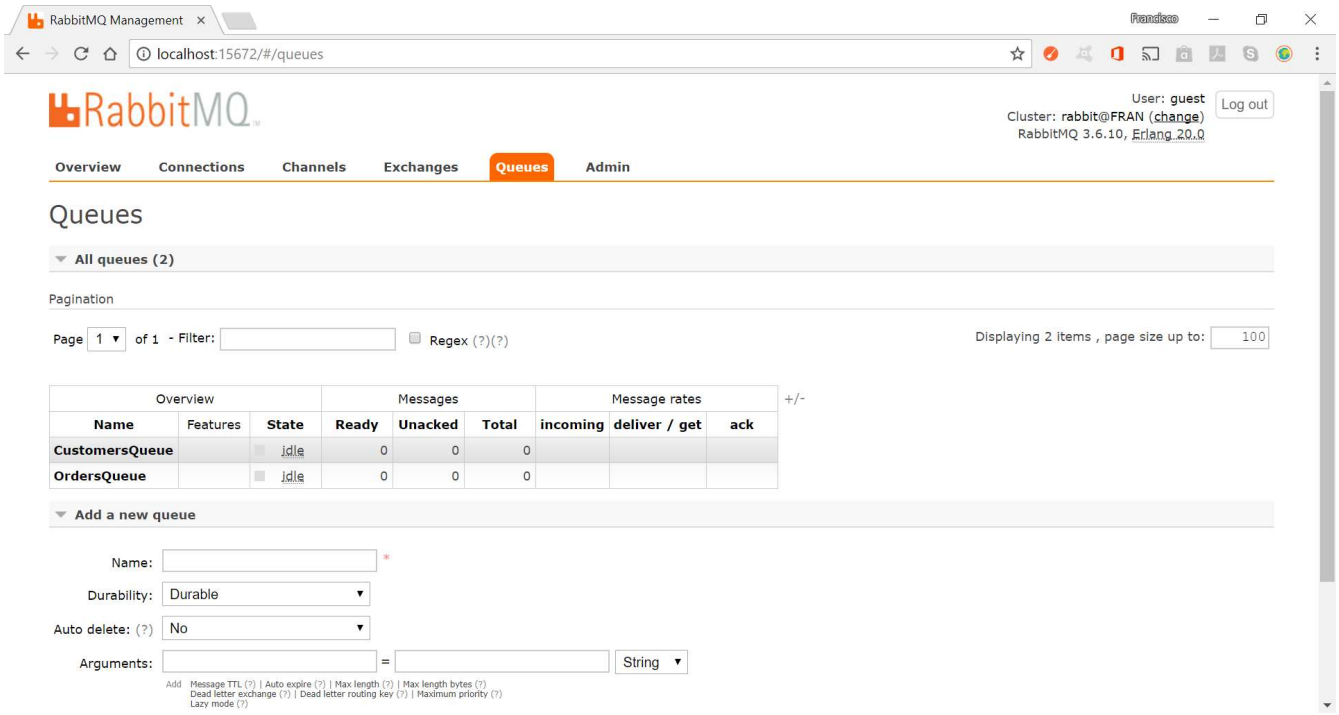


Ilustración 50 - Estado inicial de las colas de RabbitMQ en el momento del arranque de los microservicios, previo a la transmisión de mensajes

Así como los canales de comunicación establecidos por cada microservicio.

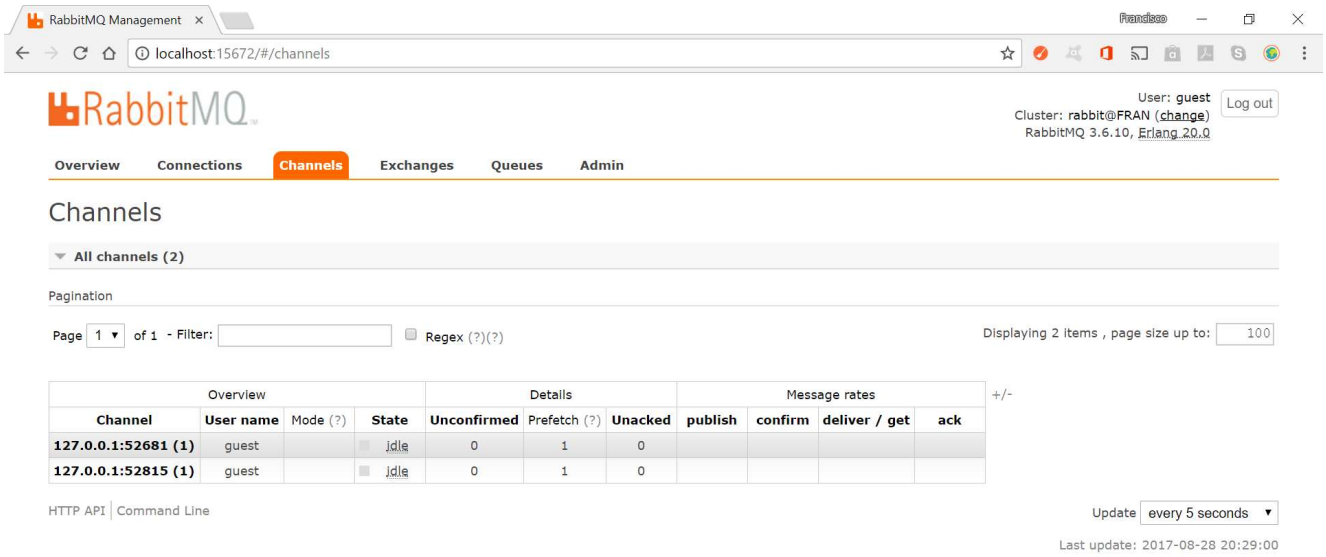


Ilustración 51 - Creación de los canales de comunicación en RabbitMQ en el momento del arranque de los microservicios

Por su parte, el microservicio de pedidos expone una interfaz web que permitirá al usuario poder interactuar con él, ya sea para crear un nuevo pedido o para listar los existentes. Aunque esta interacción se podría haber hecho igualmente a través de un cliente REST de los muchos que hay disponibles en el mercado, como por ejemplo Insomnia o Boomerang.

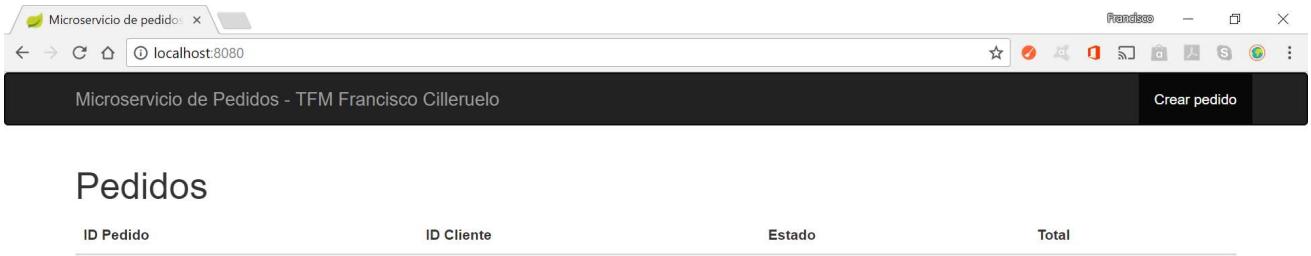


Ilustración 52 - Interfaz web por defecto del microservicio de pedidos con el listado de pedidos registrados

Por otro lado, puesto que el usuario no va a interactuar directamente con el microservicio de clientes, este expone un navegador web basado en HAL (Hypertext Application Languge) para poder consultar su API y de esa manera consultar sus recursos (los clientes y su crédito acumulado) de acuerdo con el patrón de servicios REST HATEOAS (Hypertext As The Engine Of Application State), que permite navegar entre recursos asociados por medio de enlaces (URLs) con relativa facilidad.

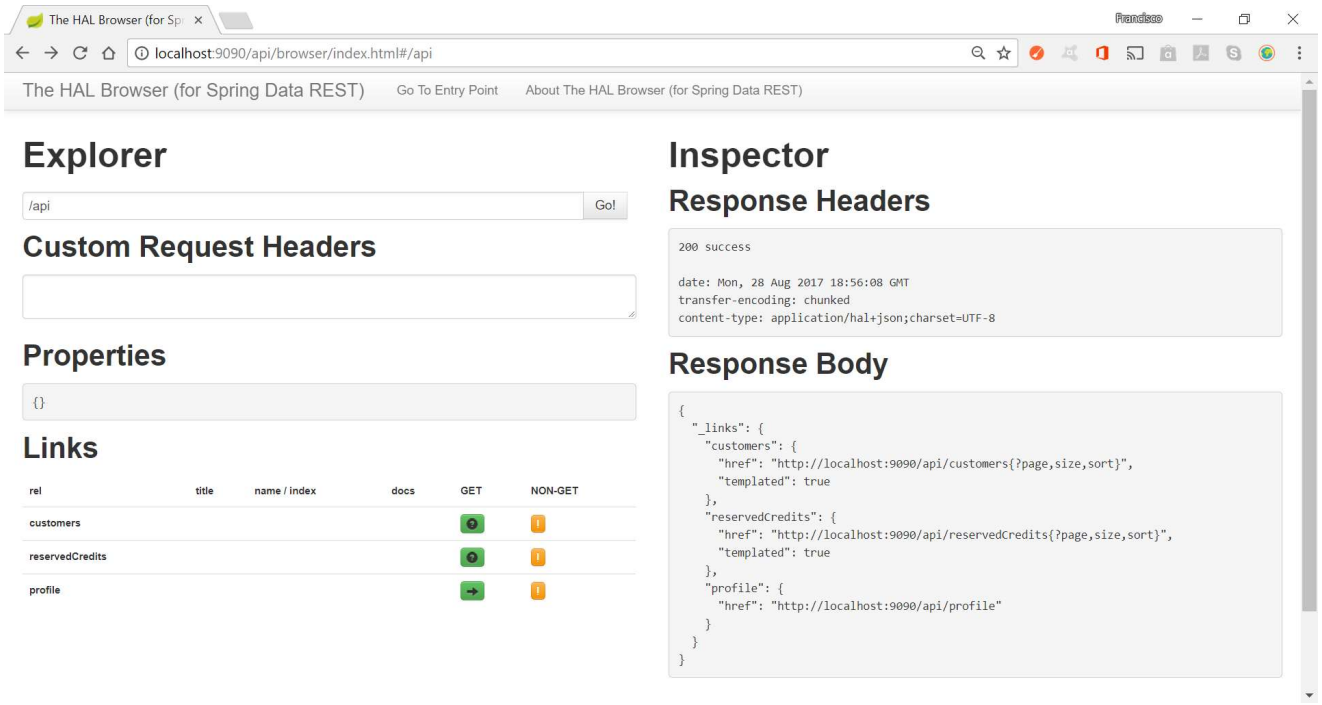


Ilustración 53 - Navegador web del API del microservicio de clientes basado en HAL

Así, por ejemplo, por medio de este navegador podemos consultar fácilmente los clientes de ejemplo instanciados y guardados inicialmente al arrancar el servicio.

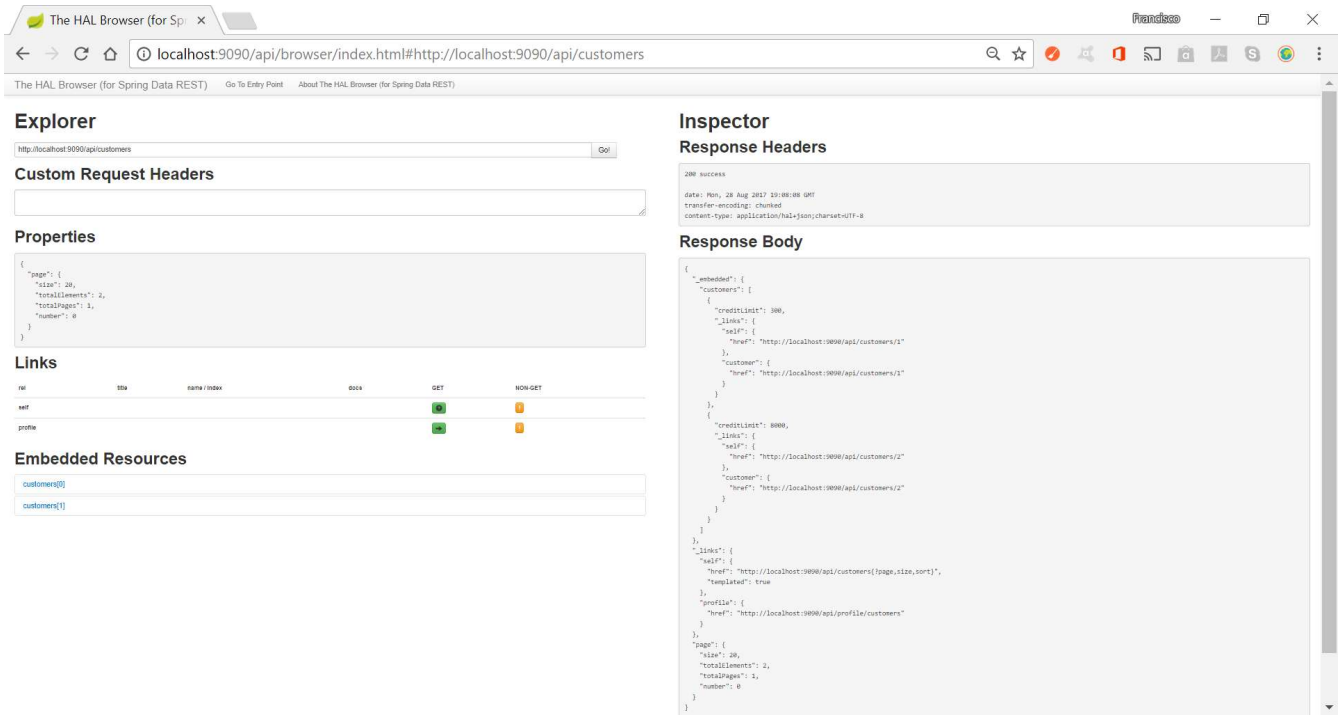


Ilustración 54 - Consulta de clientes registrados en el microservicio de clientes a través del navegador HAL de su API

El proceso será muy simple, creamos el pedido a través del interfaz web del microservicio de pedidos

Ilustración 55 - Interfaz web para la creación de un nuevo pedido

Al crearlo, por un lado, se notifica que el pedido ha sido creado, y comprobamos en la lista de pedidos existentes que ha sido guardado en un estado inicial "NEW", a la espera de recibir la aceptación o rechazo por parte del microservicio de clientes.

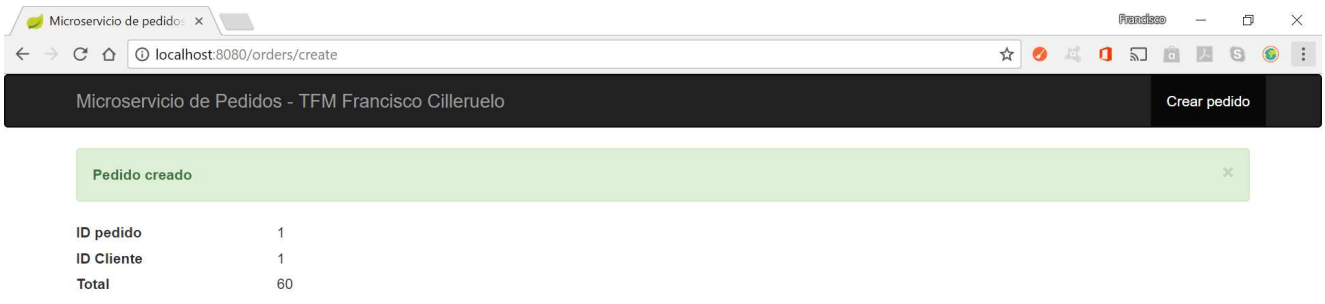


Ilustración 56 - Pantalla de confirmación del registro de un nuevo pedido

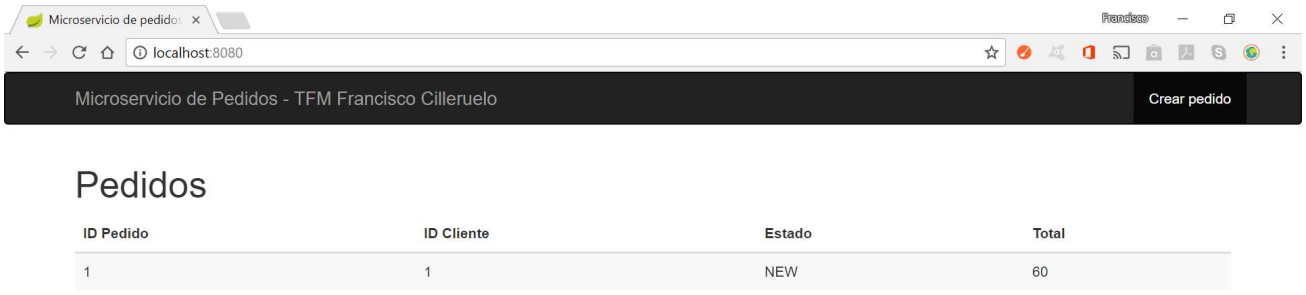


Ilustración 57 - Listado de pedidos registrados con el nuevo pedido recién creado

Y, por otro lado, comprobamos que el mensaje ha sido creado y enviado a la cola OrdersQueue de donde lo recogerá el microservicio de clientes (Para poder comprobarlo, hemos tenido que detener el microservicio de clientes temporalmente para que no lo consuma inmediatamente).

Desde un punto de vista más gráfico, y tomando como referencia los diagramas de funcionamiento que hemos visto, este proceso inicial correspondería con el flujo de ejecución que se realiza de los pasos 1° a 3° del digrama planteado en la "Ilustración 43 - Diagrama de funcionamiento de la arquitectura de microservicios ejemplo", o con más detalle, desde la perspectiva del componente emisor, el proceso que abarca de los pasos 1° al 6° del diagrama de la "Ilustración 47 - Diagrama de funcionamiento del microservicio de pedidos"

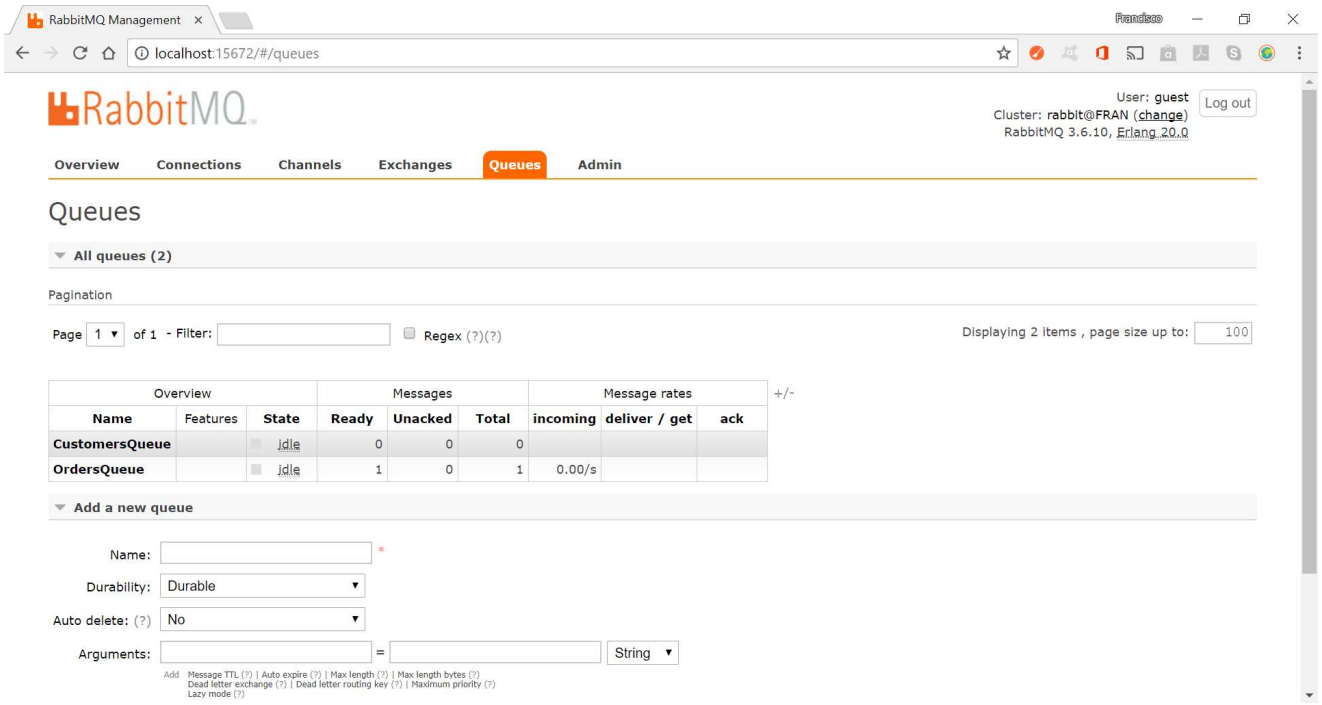


Ilustración 58 - Interfaz web de RabbitMQ con el estado de las colas en el momento del envío del mensaje por parte del microservicio de pedidos

Una vez que el microservicio de clientes recibe el mensaje, y hace las comprobaciones pertinentes, envía el mensaje de vuelta a la cola CustomersQueue de donde lo recuperará el microservicio de pedidos (Igual que en el caso anterior, para poder comprobarlo hemos tenido que detener el microservicio de pedidos temporalmente para que no lo consuma al momento).

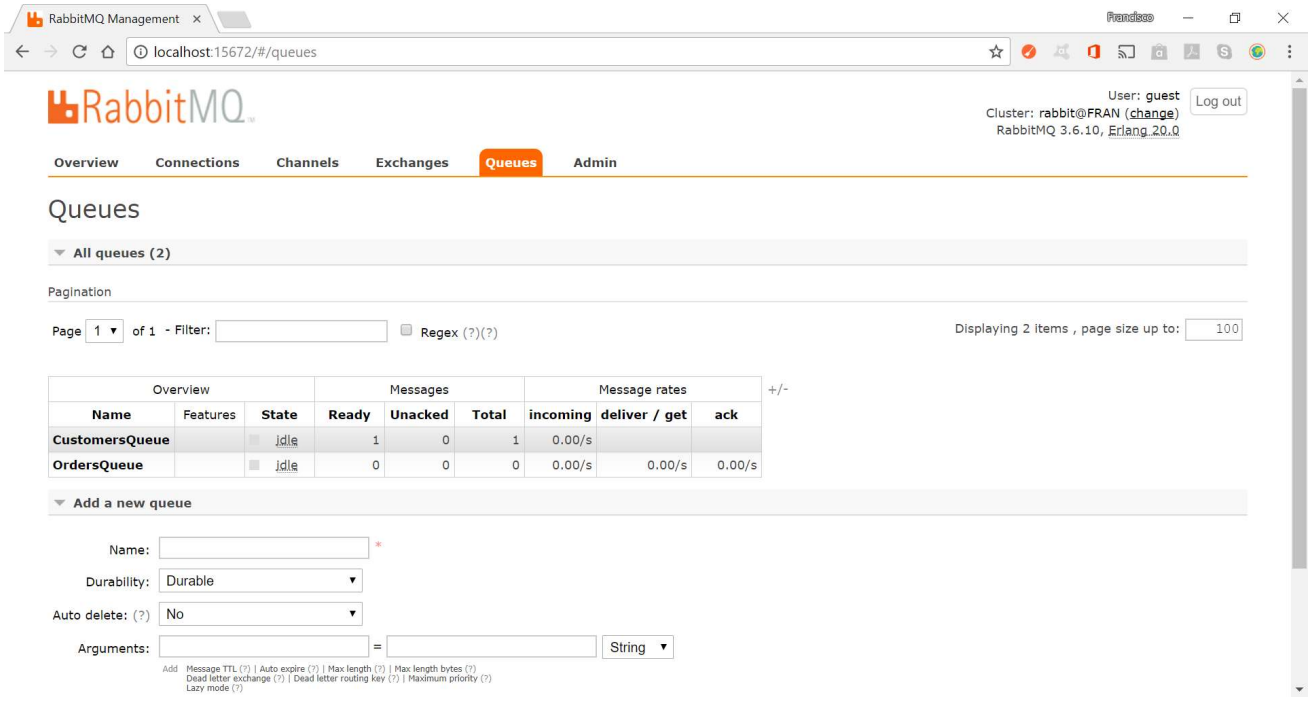


Ilustración 59 - Interfaz web de RabbitMQ con el estado de las colas en el momento del envío del mensaje por parte del microservicio de clientes

Desde la perspectiva de los diagramas de funcionamiento que hemos expuesto, este proceso de recepción del mensaje y respuesta al emisor correspondería, desde una perspectiva general, con los pasos 4º al 6º del diagrama desarrollado en la "Ilustración 43 - Diagrama de

funcionamiento de la arquitectura de microservicios ejemplo"; o bien, desde la perspectiva concreta del componente receptor, abarcaría el procesamiento completo que se lleva a cabo de los pasos 1° al paso 7° del diagrama planteado en la "Ilustración 49 - Diagrama de funcionamiento del microservicio de clientes"

Y una vez que lo recibe el microservicio de pedidos, y comprueba el resultado, actualiza el estado del pedido; en este caso a "OPEN" ya que el total solicitado no supera el crédito permitido y acumulado para ese usuario. En caso contrario, se mostraría con estado "REJECTED".

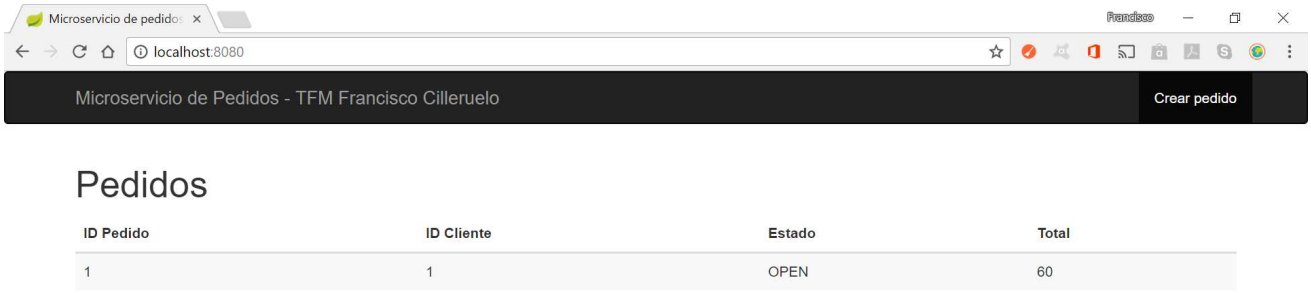


Ilustración 60 - Listado de pedidos con el nuevo pedido una vez ha sido aceptado por el microservicio de clientes

Y con esto, el proceso de creación de un pedido de manera asíncrona y distribuida se daría por terminado.

En los diagramas que hemos visto a lo largo de este capítulo, esta última fase del procesamiento de la transacción correspondería, desde la perspectiva del conjunto, con los pasos 7° y 8° del diagrama planteado en la "Ilustración 43 - Diagrama de funcionamiento de la arquitectura de microservicios ejemplo", y, desde una perspectiva más detallada del servicio emisor, con el proceso que abarca del paso 7° al paso 10° de la "Ilustración 47 - Diagrama de funcionamiento del microservicio de pedidos"

Problemática

En base a este ejemplo podemos ver como este tipo de procesamiento asíncrono entre microservicios distribuidos presenta una serie de problemas importantes a los que se debería dar solución:

El objeto de negocio del emisor permanece en un estado inicial inconsistente por la falta de atomicidad en el proceso de inicio de la transacción y el envío del mensaje al agente de mensajería

En el supuesto de que el gestor de mensajería (RabbitMQ en este caso) no esté disponible, o estándolo no sea capaz de aceptar y procesar el mensaje enviado por el servicio emisor, el proceso quedaría incompleto, y con ello el objeto de negocio (en nuestro caso el pedido) en un estado inconsistente. En otras palabras, no se asegura la atomicidad y, por lo tanto, la transaccionalidad, del proceso formado, por un lado, de las operaciones de persistencia en el microservicio origen y, por otro, de la operación de envío y recepción del mensaje por parte del gestor de mensajería. Lo que es claramente una fuente de inconsistencias. De un modo más gráfico, correspondería con una interrupción del proceso de comunicación en el paso 3° del diagrama presentado en la "Ilustración 43 - Diagrama de funcionamiento de la arquitectura de microservicios ejemplo", o de una manera más detallada, desde la perspectiva del componente emisor, en el paso 6° de la "Ilustración 47 - Diagrama de funcionamiento del microservicio de pedidos".

El objeto de negocio del emisor permanece en un estado inicial inconsistenete por la falta de respuesta del receptor

Aún en el caso de que el mensaje sea enviado y recibido correctamente por el gestor de mensajería, el objeto de negocio queda en un estado inconsistente inicial (en nuestro caso "NEW"), a la espera de recibir la correspondiente respuesta del servicio receptor que dé por terminado el proceso y, con ello, deje dicho objeto en un estado final consistente (en nuestro caso "OPEN" o "REJECTED"). En el caso de que este servicio al que va dirigida le mensaje, por el motivo que sea, no reciba, o recibiéndolo no lo procese y envíe una respuesta de manera correcta, el objeto de negocio quedará en un estado inconsistente de manera indefinida. Desde el punto de vista de los diagramas de funcionamiento presentados en este capítulo, correspondería con una interrupción del proceso en el paso 6° de la "Ilustración 43 - Diagrama de funcionamiento de la arquitectura de microservicios ejemplo", o de una manera más detallada, desde la perspectiva del microservicio receptor, en el paso 7° de la "Ilustración 49 - Diagrama de funcionamiento del microservicio de clientes".

La falta de encapsulamiento del dominio de negocio del emisor

De acuerdo con las características descritas para una arquitectura de microservicios, no sólo la implementación o particularidades tecnológicas de los propios microservicios deben ser opacas y desconocidas para el resto de los microservicios, también debe serlo la lógica de negocio que llevan a cabo, específica de su dominio y alcance de acción. Sin embargo, tal y como se puede comprobar, mediante este proceso de comunicación delegamos en el microservicio receptor parte de la lógica de negocio del microservicio emisor (en nuestro caso el microservicio de clientes es el encargado de determinar el estado final del pedido, cuando ese es un aspecto propio del dominio de microservicio de pedidos). De acuerdo con los diagramas de funcionamiento que hemos visto, y tomando como referencia la perspectiva del microservicio que recibe la llamada, correspondería concretamente con el paso 6° del proceso expuesto en la "Ilustración 49 - Diagrama de funcionamiento del microservicio de clientes".

Implementación de la solución: Gestor de transacciones distribuidas asíncronas. Asynchronous Distributed Transaction Manager (ADTM)

Con el objetivo de poder ofrecer la mejor solución posible a los problemas y retos que, tal y como hemos comprobado, presenta una comunicación asíncrona en una arquitectura distribuida como es la de microservicios. Vamos a implementar una solución software que, no sólo resuelva el problema que se ajusta a esta tipología; sino que además responda a unos estándares de calidad, es decir, que sea versátil y reutilizable para poder integrarse con estas u otras arquitecturas distribuidas que presenten este modo de comunicación, de fácil manejo para el desarrollador y con la posibilidad, además, de que pueda evolucionar hacia versiones mejoradas para ajustarse a nuevos requisitos puedan surgir dentro de este mismo contexto.

Igual que hicimos en el desarrollo de la arquitectura de microservicios utilizada a modo de ejemplo, y por seguir la misma línea de tecnologías que faciliten la compatibilidad, vamos a utilizar las siguientes tecnologías:

- Framework Spring Boot 1.5.6. Con los siguientes módulos adicionales:
  - Spring Boot Starter AMQP
  - Spring Boot Starter JPA
- Base de datos H2 1.4.196
- Apache Maven 3.5.0
- Java JDK 1.8.0_73
- RabbitMQ 3.6.10
- Erlang 20.0

Este módulo o librería, a la que llamaremos ADTM, por sus siglas en inglés Asynchronous Distributed Transaction Manager (Gestor de transacciones distribuidas asíncronas), va a interactuar, por un lado, con los componentes distribuidos que lo utilizarán como herramienta integrada, es decir, tanto el microservicio emisor iniciador de la transacción, como el microservicio receptor de ésta. Y, por otro lado, con el gestor de mensajería AMQP (en nuestro caso RabbitMQ) al que cualquiera de ellos enviará o recibirá mensajes utilizando como intermediario middleware la solución propuesta. Así pues, nuestra implementación desde un punto de vista de la arquitectura de componentes presentará la siguiente estructura.

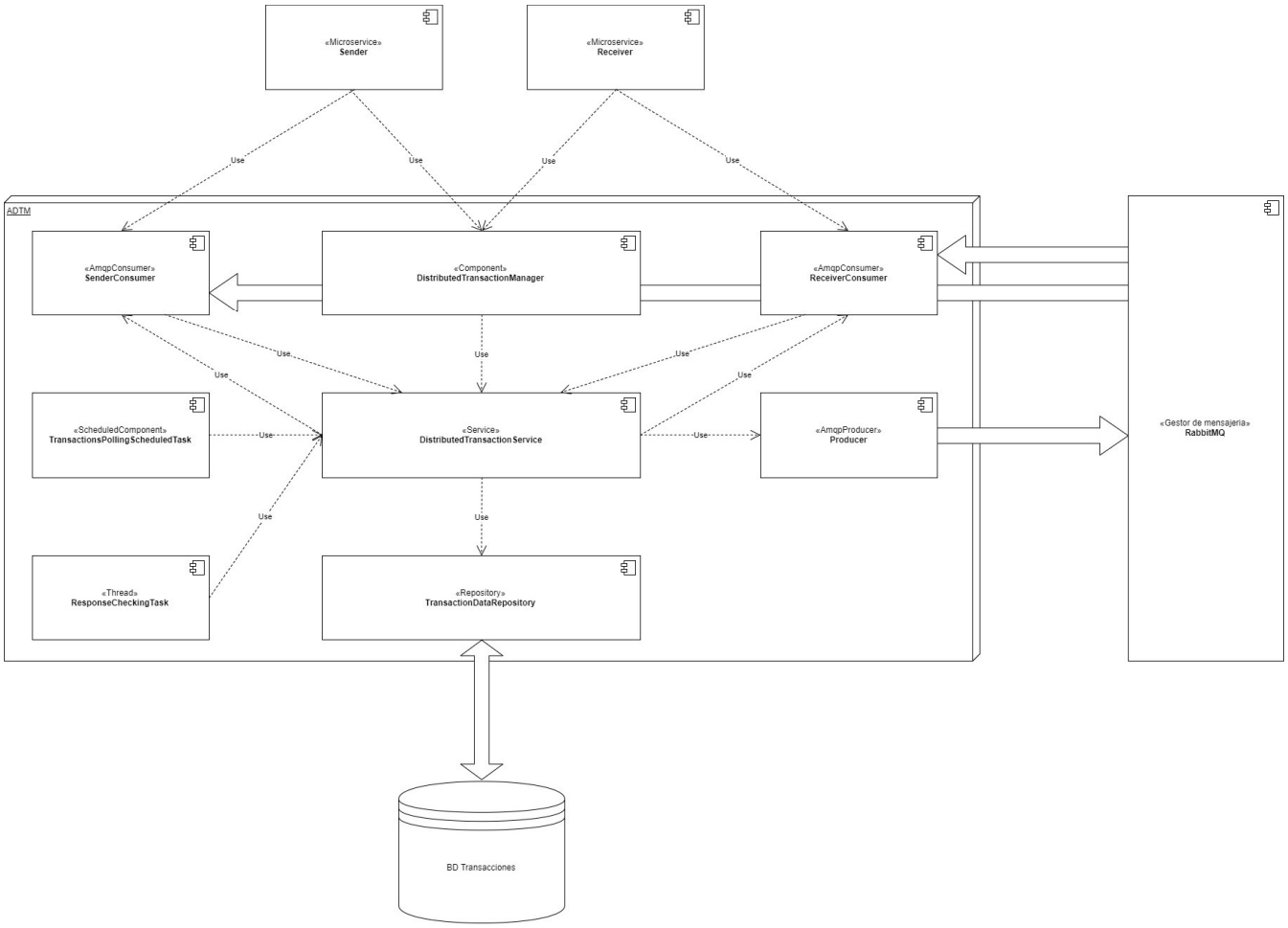


Ilustración 61 - Diagrama de componentes de ADTM

Ahora bien, la interacción y comunicación entre estos componentes será diferente dependiendo de si el que hace uso de la solución ADTM es el microservicio emisor o el microservicio receptor, puesto que cada uno de ello implica una lógica de tratamiento y resolución distinta. Veamos este aspecto en más detalle.

En el caso de que el rol del microservicio que utiliza nuestra herramienta ADTM es el de emisor, o, dicho de otra forma, el que inicia la transacción, el proceso será como se muestra en el diagrama siguiente

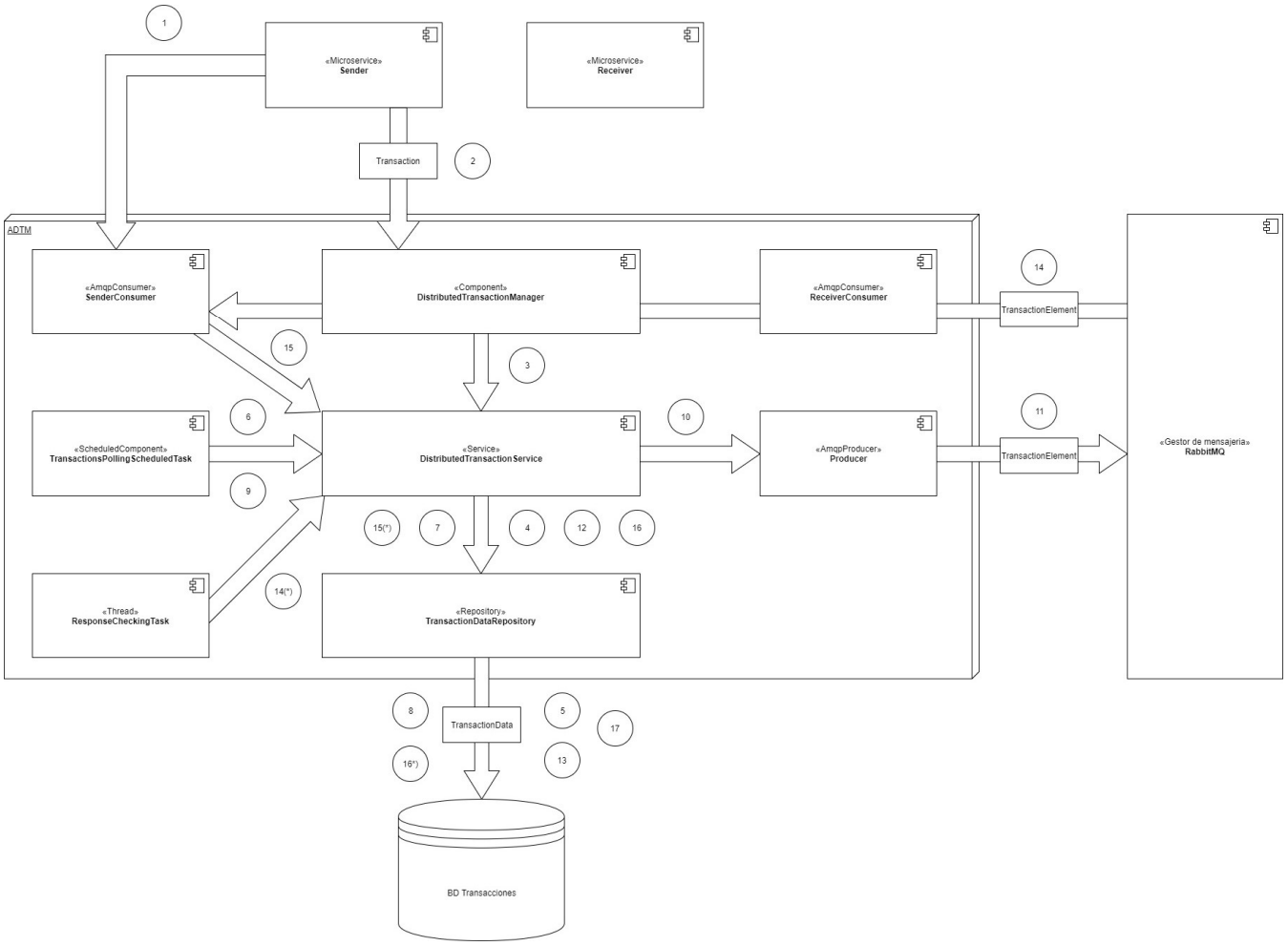


Ilustración 62 - Diagrama de funcionamiento de ADTM para el emisor de la transacción

1. El microservicio emisor, antes de iniciar la transacción como tal, implementa e instancia como componente propio, una clase que hereda de la clase abstracta SenderConsumer definida en ADTM, para luego ser transmitida como parte de la transacción. Este componente será el encargado de recibir, en el caso de que llegue, la respuesta transmitida por parte del receptor. La parte implementable por el emisor, sus métodos abstractos (commit() y rollback()), se hará de acuerdo con sus requisitos de negocio particulares. De manera que se ejecutará uno u otro según la respuesta recibida se considere correcta (commit) o, por el contrario, incorrecta (rollback)
2. En el inicio de la transacción, el emisor llama al método sendTransaction() del componente DistributedTransactionManager, inyectado automáticamente en modo Singleton al importar la solución ADTM como parte del microservicio. Al que se pasará únicamente el objeto que representa la propia transacción con todas sus propiedades asociadas: el objeto SenderConsumer implementado e instanciado en el paso anterior que consumirá la respuesta del receptor, el objeto de negocio transmitido, los nombres de las colas de envío y recepción, y, por último, si lo hubiese, el tiempo máximo de respuesta permitido (tiempo tras el cual la transacción no se dará por correcta y se ejecutará la acción de rollback definida para el objeto SenderConsumer ya mencionado)
3. El componente DistributedTransactionManager llama al componente donde reside la lógica de negocio, DistributedTransactionService. Éste, además de crear el objeto de persistencia TransactionData que se guardará en base de datos posteriormente, registra el SenderConsumer para asociarlo más tarde, en el momento en el que se hace efectivo el envío del mensaje al gestor de mensajería, a la cola donde se espera recibir la respuesta por parte del receptor.
4. El servicio llama al componente JPA (TransactionDataRepository) para que persista el objeto TransactionData recién creado en base de datos, con información relativa a la

transacción: objeto de negocio a transmitir, fecha de inicio de la transacción, estado inicial (TO_BE_SENT) …etc.

5. El objeto TransactionData es persistido en base de datos
6. En paralelo y de manera constante, un hilo de ejecución (TransactionsPollingScheduledTask) lanzado en el momento del arranque, comprueba de manera periódica si hay alguna transacción pendiente de ser enviada. Esta frecuencia está establecida por defecto a 60000 milisegundos, es decir, un minuto, pero puede ser modificada por el propio emisor a través de la propiedad "*adtm.pollingfrecuency*". Para ello, solicita al servicio que le devuelva todas las transacciones pendientes de ser enviadas, ya sea porque se han creado después de la última comprobación, o porque en intentos anteriores no se han podido enviar, quizás porque el gestor de mensajería no estaba disponible en ese momento.
7. El componente de servicio DistributedTransactionService solicita el listado de transacciones pendientes de enviar a la capa de repositorio
8. El componte JPA que interactúa con la base de datos recupera el listado de transacciones pendientes de ser enviadas, si las hubiese, es decir, sin fecha de envío y que además no se han descartado por haber superado el límite de tiempo de espera establecido por el emisor
9. En el caso de que haya alguna transacción pendiente de ser enviada, el componente TransactionsPollingScheduledTask llama al servicio DistributedTransactionService. Este, por su parte, además de las tareas que se describen en los siguientes pasos, recupera el objeto SenderConsumer asociado a la transacción que debe recibir la respuesta, y lo asocia a la cola donde esta se espera recibir. Y, además, en el caso de que el emisor indicase que para la transacción tratada hay un tiempo máximo de respuesta permitido, se crea un hilo de ejecución paralelo (ResponseCheckingTask), que pasado ese tiempo y en el caso de que no se haya recibido respuesta alguna por parte del receptor toma las acciones pertinentes para no dejar la transacción en un estado inconsistente, tal y como veremos más adelante.
10. El componente de servicio envía la transacción al componente Producer encargado de enviarla a la cola de envío establecida por el emisor. La información relativa a la transacción se envía como un objeto del tipo TransactionElement, con información como: el identificador de referencia de la transacción, el estado de estado, el objeto de negocio transmitido y el nombre de la cola donde el receptor debe enviar su respuesta
11. El componente Producer envía la transacción como un mensaje en formato JSON al gestor de mensajería
12. El servicio actualiza el objeto persistido TransactionData que representa la transacción en curso de acuerdo con el proceso de envío. Ya sea, en el caso de que no haya habido ningún problema en el envío, para establecer como nuevo estado de transacción el estado SENT y reflejar la hora en la que se ha hecho el envío, o, en el caso contrario de que haya habido algún tipo de excepción, para incrementar el número de intentos de envío.
13. El componente de servicio invoca a la capa de persistencia JPA para guardar el objeto TransactionData actualizado
14. En el caso de que el receptor conteste correctamente, el componente SenderConsumer asociado a la transacción y registrado por el emisor, recibe la respuesta a la transacción como un objeto TransactionElement. En el caso de que el receptor en su respuesta indique que la transacción ha sido procesada correctamente ("OK"), se ejecutará el proceso implementado por el emisor en el método commit(), en caso contrario ("NOK" o cualquier otro estado no reconocido), se ejecutará el proceso implementado en el método rollback() para ese tipo de situaciones
15. El componente SenderConsumer indica en cualquier caso al servicio DistributedTransactionService que se ha recibido la respuesta. Éste, por su parte, actualiza el objeto del tipo TransactionData guardado en base de datos y que representa la transacción en curso, con la fecha de recepción, el estado devuelto por el receptor y cualquier otra información adicional que haya podido indicar
16. El servicio llama a la capa de repositorio JPA TransactionDataRepository para actualizar el objeto de la transacción con la nueva información
17. El componente de repositorio persiste el objeto en base de datos

A este flujo del procesamiento de la transacción desde la perspectiva del emisor, en el caso presumiblemente más normal y habitual, habría que añadir un flujo alternativo en el caso del emisor hubiese establecido un tiempo máximo de respuesta permitido y éste se hubiese superado sin recibir respuesta.

14. (*) El hilo de ejecución creado previamente a partir del componente ResponseCheckingTask como parte del proceso de envío (paso 9), encargado de comprobar si la transacción ha recibido respuesta pasado el tiempo límite establecido, después de verificar que no se ha recibido respuesta alguna, llama al componente de servicio DistributedTransactionService para que tome las acciones pertinentes. El servicio, por su parte, actualiza el objeto persistido del tipo TransactionData que representa la transacción en curso de acuerdo con esta situación, establece su estado como no respondido (NOT_RECEIVED) y la fecha cuando se ha hecho esta comprobación. Además, elimina del contexto, puesto que ya no tiene sentido, el componente SenderConsumer que esperaba recibir la respuesta
15. (*) El servicio llama a la capa de persistencia JPA para que actualice el objeto de la transacción con esta nueva información
16. (*) La base de datos persiste y con ello actualiza el objeto en base de datos

Por su parte, en el caso de que el microservicio que utiliza nuestra herramienta ADTM sea el receptor, el proceso será como se muestra en el diagrama siguiente

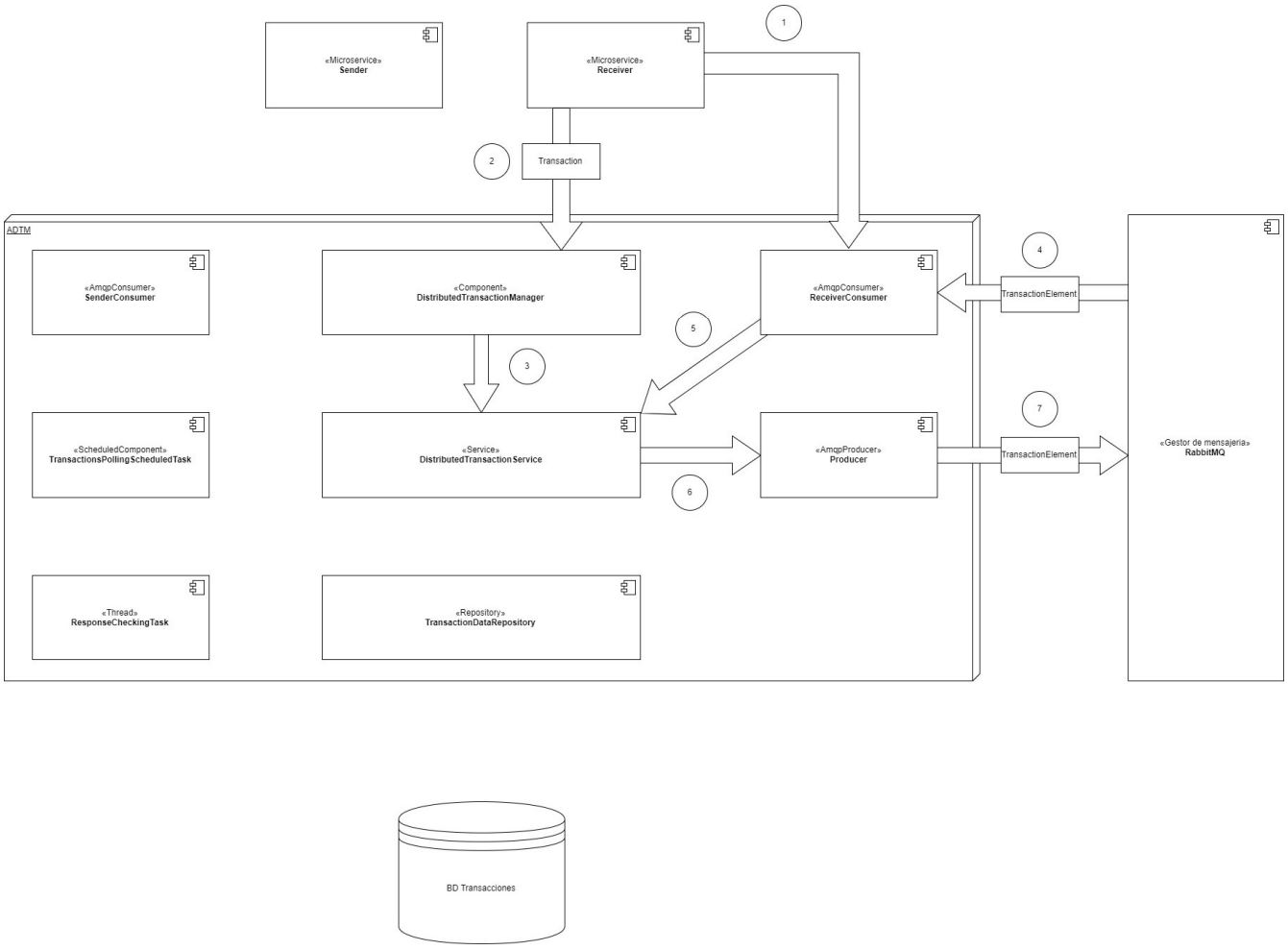


Ilustración 63 - Diagrama de funcionamiento de ADTM para el receptor de la transacción

1. El microservicio receptor, en previsión de recibir una transacción, implementa e instancia como un componente propio, una clase que hereda de la clase abstracta

ReceiverConsumer definida en ADTM, para ser pasado posteriormente como parámetro en la llamada al componente DistributedTransactionManager. La parte implementable por el emisor, su método abstracto (processRequest()), se hará de acuerdo con sus requisitos de negocio particulares. Este componente recibirá, en el caso de que llegue, la transacción enviada por el emisor, y ejecutará el comportamiento definido por el receptor en el método processRequest() para determinar si la transacción se puede dar por correcta (true) o no (false), además de cualquier otra operación adicional que pueda querer hacer el microservicio, como, por ejemplo, guardar información en su propia base de datos.

2. El receptor llama al método receiveTransaction() del componente DistributedTransactionManager, inyectado automáticamente en modo Singleton al importar la solución ADTM como parte del microservicio. Al que se pasará, el nombre de la cola donde se espera recibir la transacción por parte del microservicio emisor, y el objeto ReceiverConsumer implementado e instanciado en el apartado
3. El componente DistributedTransacionManager llama al componente de servicio DistributedTransactionService, para que éste enganche el componente indicado ReceiverConsumer a la cola establecida donde se espera recibir la transacción del emisor. Y que, de esa manera, el componente pueda consumir los mensajes transmitidos por el emisor
4. Presumiblemente, en un momento dado, el gestor de mensajería enviará el objeto TransactionElement, que representa la transacción enviada por el emisor con toda su información asociada, a través de la cola donde el componente ReceiverConsumer espera recibirla y está escuchando. Por su parte, este componente ejecuta la lógica de negocio definida por el receptor en el método processRequest() para establecer si la transacción se puede dar por correcta o no, además de ejecutar cualquier otra acción adicional que el receptor haya establecido.
5. En cualquier caso, tanto si el receptor da la transacción por correcta como si no, el componente ReceiverConsumer llama al componente de servicio para enviar la respuesta al emisor
6. El componente DistributedTransactionService llama al componente Producer encargado de enviar la transacción a la cola de envío indicada como parte de la información de la transacción, donde espera recibir la respuesta el emisor
7. Finalmente, el componente Producer envía la transacción como un objeto TransactionElement con toda su información actualizada de acuerdo con el procesamiento realizado por el receptor, indicando, como es obvio, si ha sido procesado correctamente (OK) o no (NOK), para que el emisor tome las medidas que considere necesarias dentro de su propio dominio de negocio.

Toda la implementación del gestor de transacciones distribuidas asíncronas (ADTM), está disponible para quien lo quiera consultar, o incluso mejorar, en el repositorio público de GitHub: https://github.com/franciscocilleruelo/ADTM.git

Así mismo, el código fuente de las principales clases Java involucradas en la implementación de este capítulo se puede encontrar en el Anexo 2- Código fuente de la implementación de la solución: Gestor de transacciones distribuidas asíncronas (ADTM)

Veamos ahora por separado los detalles de cada uno de los componentes implicados en la solución propuesta.

Problemática resuelta

Mediante la solución planteada, ADTM (Asynchronous Distributed Transaction Manager), que en el siguiente apartado veremos cómo integrarla dentro de una arquitectura de microservicios, quedan resueltos, tal y como pretendíamos, los problemas e inconvenientes que habíamos comprobado que surgen en una implementación de un sistema con una arquitectura distribuida, como es la de microservicios, con un estilo de comunicación asíncrono. Así pues, los aspectos resueltos quedan resumidos de la siguiente manera:

El objeto de negocio del emisor permanece en un estado inicial inconsistente por la falta de atomicidad en el proceso de inicio de la transacción y el envío del mensaje al agente de mensajería

Con esta librería, en el supuesto de que el gestor de mensajería no esté disponible, o estándolo no sea capaz de aceptar y procesar el mensaje enviado por el servicio emisor, el proceso no quedaría incompleto, y con ello el objeto de negocio en un estado inconsistente. Esto lo conseguimos asegurando la atomicidad de la transacción en el proceso de envío, cosa que antes no sucedía, ya que después de la lógica de negocio propia de emisor, y previa al envío al componente de mensajería, como parte de la misma transacción local sobre base de datos, guardamos la transacción en un estado pendiente de envío. De manera que, el hilo de ejecución establecido por el componente TransactionsPollingScheduledTask comprueba de manera periódica que transacciones están pendientes de envío, y las envía a su cola correspondiente tan pronto como el gestor de mensajería esté disponible. Desde un punto de vista más gráfico, tomando como referencia los diagramas de funcionamiento expuestos con anterioridad, está solución correspondería con el proceso llevado a cabo por el componente emisor del paso 3º al 9º del diagrama presentado en la "Ilustración 62 - Diagrama de funcionamiento de ADTM para el emisor de la transacción".

El objeto de negocio del emisor permanece en un estado inicial inconsistente por la falta de respuesta del receptor

Tal y como apuntábamos, aún en el caso de que el mensaje sea enviado y recibido correctamente por el gestor de mensajería, el objeto de negocio queda en un estado inconsistente inicial a la espera de recibir la correspondiente respuesta del servicio receptor que dé por terminado el proceso y deje dicho objeto en un estado final consistente. En el caso de que este servicio receptor, por el motivo que sea, no reciba el mensaje, o recibiéndolo no lo procese y envíe una respuesta de manera correcta, el objeto de negocio ya no quedaría en un estado inconsistente de manera indefinida. Puesto que, el propio emisor puede establecer un tiempo máximo de respuesta permitido después de haber realizado el envío, tiempo tras el cual, se ejecuta un hilo de ejecución establecido por el componente ResponseCheckingTask, que, al comprobar que no se ha recibido respuesta, ejecuta las acciones indicada por el emisor en la implementación del método rollback() del componente SenderConsumer asociado a dicha transacción. Desde el punto de vista de los diagramas de funcionamiento presentados en este capítulo, esta solución correspondería con el proceso que lleva a cabo por el servicio emisor en el paso 9º, seguido del flujo alternativo seguido desde el paso 14º(*) al 16º(*), del diagrama expuesto en la "Ilustración 62 - Diagrama de funcionamiento de ADTM para el emisor de la transacción".

La falta de encapsulamiento del dominio de negocio del emisor

Así mismo, de acuerdo con las características descritas para una arquitectura de microservicios, no sólo la implementación o particularidades tecnológicas de los propios microservicios deben ser opacas y desconocidas para el resto de los microservicios, también debe serlo la lógica de negocio que llevan a cabo, específica de su dominio y alcance de acción. Y, con esta solución, la lógica del microservicio que inicia la transacción ya no se reparte entre éste y el receptor, sino que recae íntegramente en él. De manera totalmente transparente para el microservicio que recibe la transacción, que se limita exclusivamente a indicar si por su parte se puede dar la transacción por correcta o no, para que el emisor tome las acciones que considere oportunas dentro de su propio dominio de negocio. De acuerdo con los diagramas de funcionamiento que hemos visto, esta solución quedaría implementada, por un lado, por parte del

servicio receptor en el paso 4° del diagrama presentado en la "Ilustración 63 - Diagrama de funcionamiento de ADTM para el receptor de la transacción", y por otro, por parte del microservicio que inicia la transacción y, por lo tanto, encargado de recibido la respuesta del receptor, en el paso 14° del diagrama mostrado en la "Ilustración 62 - Diagrama de funcionamiento de ADTM para el emisor de la transacción".

Integración de la solución en el contexto del problema

Veamos a continuación como integrar este gestor de transacciones distribuidas asíncronas dentro del contexto de una arquitectura de microservicios. Para ello, tomaremos como referencia el sistema de microservicios implementado inicialmente a modo de ejemplo.

Toda la implementación correspondiente al sistema inicial adaptado e integrado con la herramienta ADTM, está disponible para quien lo quiera consultar, o incluso mejorar, en el repositorio público de GitHub: https://github.com/franciscocilleruelo/ServicesTFM_ADTM.git

Así mismo, el código fuente de las principales clases Java involucradas en la implementación de este capítulo se puede encontrar en el Anexo 3 – Código fuente de la solución integrada en el contexto del problema

Pasemos a ver como quedan cada uno de los componentes implicados en la comunicación asíncrona, tanto el emisor, es decir, el microservicio de pedidos, como el receptor, es decir, el microservicio de clientes.

Servicio emisor (Microservicio de pedidos)

Por su parte, el emisor, en nuestro caso el microservicio de pedidos, presentará la siguiente estructura de componentes con la librería ADTM una vez integrada:

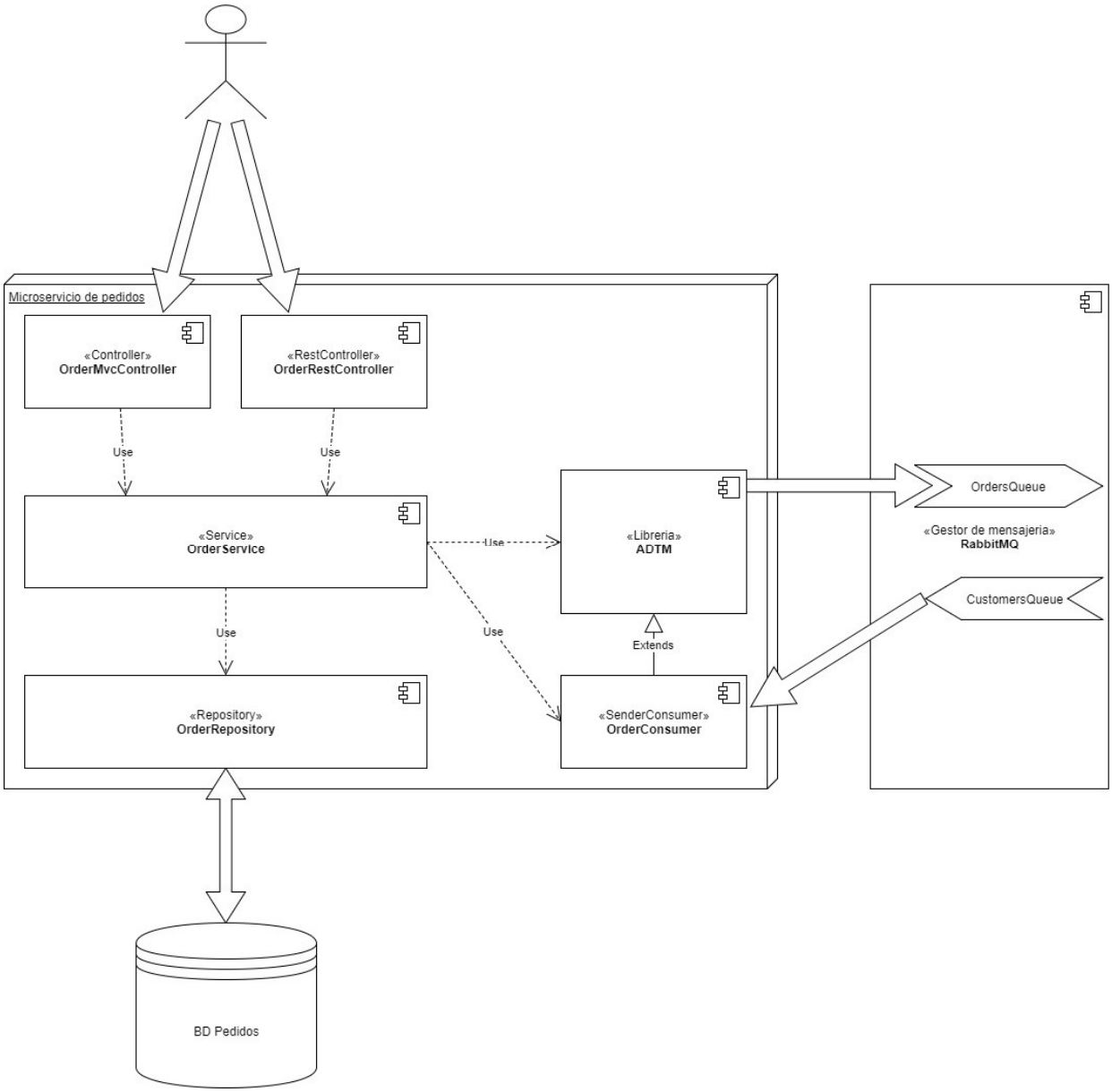


Ilustración 64 - Diagrama de componentes de ADTM integrado en el microservicio de pedidos

Y llevará a cabo el proceso de la siguiente manera:

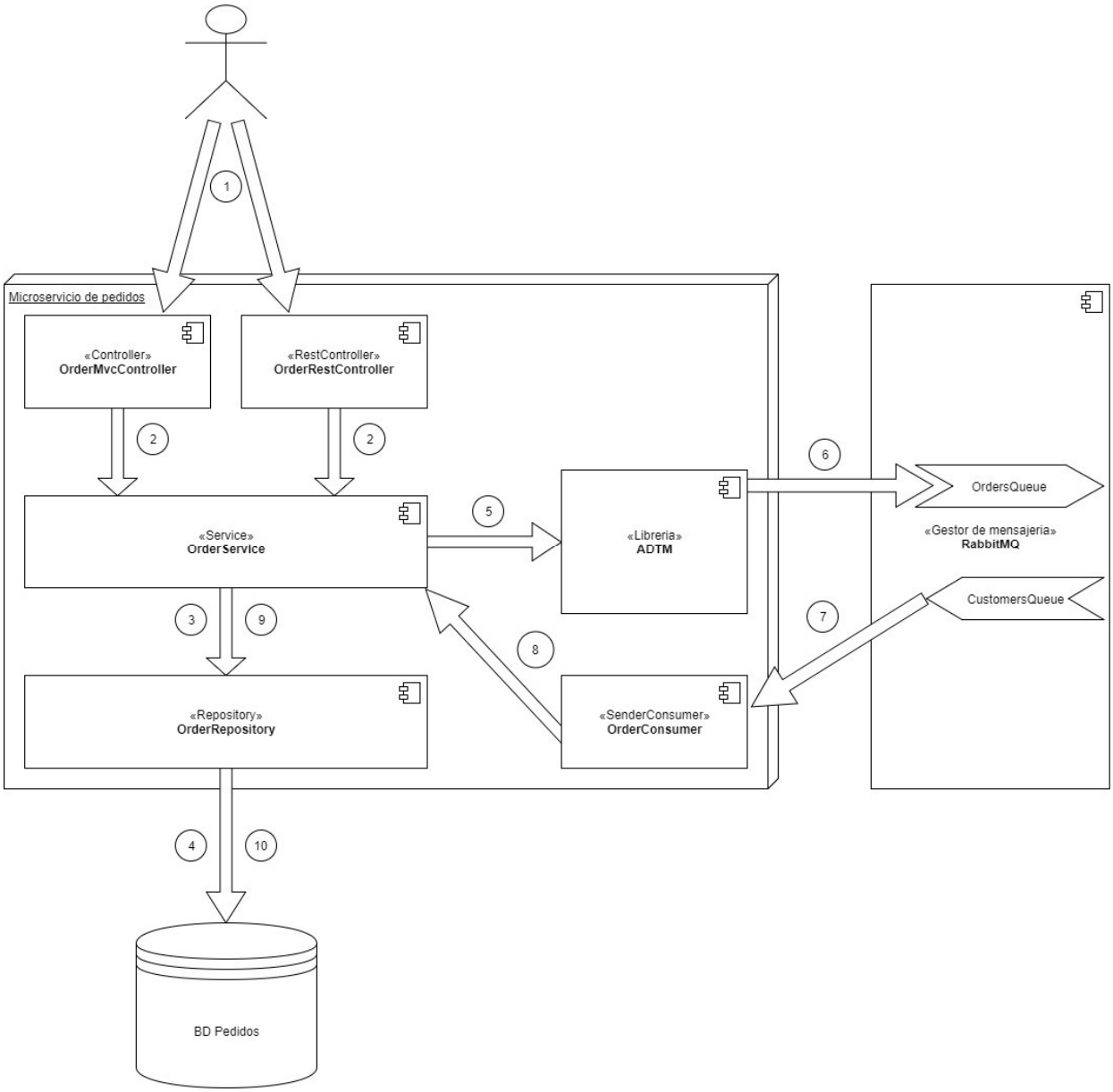


Ilustración 65 - Diagrama de funcionamiento de ADTM integrado en el microservicio de pedidos

1. El usuario/cliente envía una solicitud para crear un pedido nuevo. Esta solicitud puede ser bien a través de un servicio REST expuesto a través del controlador REST (OrderRestContoller), o bien a través de una URL invocada desde formulario web HTML a través del controlador genérico (OrderController)
2. Independientemente del punto de entrada de la solicitud, el controlador correspondiente llama al componente de servicios (OrderService) encargado de la lógica de negocio. Que en este caso simplemente podrá el pedido con estado nuevo ("NEW")
3. El servicio invoca a la componente de repositorio JPA (OrderRespository), encargado de interactuar con la base de datos, para persistir el pedido con ese estado inicial
4. El repositorio guarda el pedido en base de datos
5. El servicio llama al componente ADTM para que comience la transacción. Y le pasa el objeto que representa la transacción con todos los datos asociados necesarios (objeto de negocio, nombre de las colas de envío y recepción…), incluyendo el objeto OrderConsumer (que extiende de SenderConsumer) he implementa la lógica de negocio particular que se debe ejecutar al recibir la respuesta del microservicio al que va dirigida la transacción.

6. El componente ADTM realiza toda la lógica propia explicada anteriormente para poder enviar el mensaje de la transacción a la cola indicada (OrdersQueue) donde espera recibirlo el microservicio de clientes
7. Pasado un tiempo, presumiblemente, y en el caso de que todo haya ido bien en el lado del receptor, el componente SenderConsumer (OrderConsumer) recibe el mensaje/objeto de respuesta para ejecutar la lógica de negocio correspondiente de acuerdo con la respuesta recibida. Ya sea esta correcta, en cuyo caso ejecutará el método commit(), o no, en cuyo caso ejecutará el método rollback().
8. El componente SenderConsumer invoca al componente de servicio (OrderService) para actualizar el estado del pedido de acuerdo con la acción a realizar.
9. El servicio llama al repositorio (OrderRepository) como mediador y gestor de las operaciones con la base de datos para que actualice el pedido.
10. Finalmente, y si todo ha ido bien, el repositorio persiste el pedido.

Desde el punto de vista de la implementación, y de una manera más genérica, para poder ser aplicado a otros sistemas que igualmente se ajusten a una arquitectura de microservicios, el microservicio encargado de iniciar la transacción simplemente tendrá que:

- Importar la librería ADTM mediante la anotación @Import en su clase de arranque e inicialización (marcada mediante la anotación @SpringBootApplication)
- Extender la clase abstracta SenderConsumer de ADTM para implementar la lógica de negocio a realizar en el caso de que la transacción se dé por correcta (método commit()), o, por el contrario, y por el motivo que sea, se considere incorrecta (método rollback())
- En el punto que se considere oportuno, en principio en la clase de servicio donde reside la lógica de negocio, se inicia una transacción determinada mediante el objeto del tipo DistributedTransactionManager, propio de ADTM e inyectado automáticamente a través de Spring. Para ello, se llama a su método sendTransaction() junto con el objeto del tipo Transaction que representa la transacción con todos sus datos asociados (objeto de negocio, cola de envío, cola de respuesta, tiempo máximo de respuesta permitido y el objeto del tipo SenderConsumer encargado de recibir y procesar la respuesta del receptor)

El código fuente de las principales clases Java involucradas en la implementación de este microservicio se puede encontrar en el apartado Servicio emisor (Microservicio de pedidos) como parte del Anexo 3 – Código fuente de la solución integrada en el contexto del problema

Servicio receptor (Microservicio de clientes)

Por su parte, el emisor, en nuestro caso el microservicio de clientes, presentará la siguiente estructura de componentes con la librería ADTM una vez integrada:

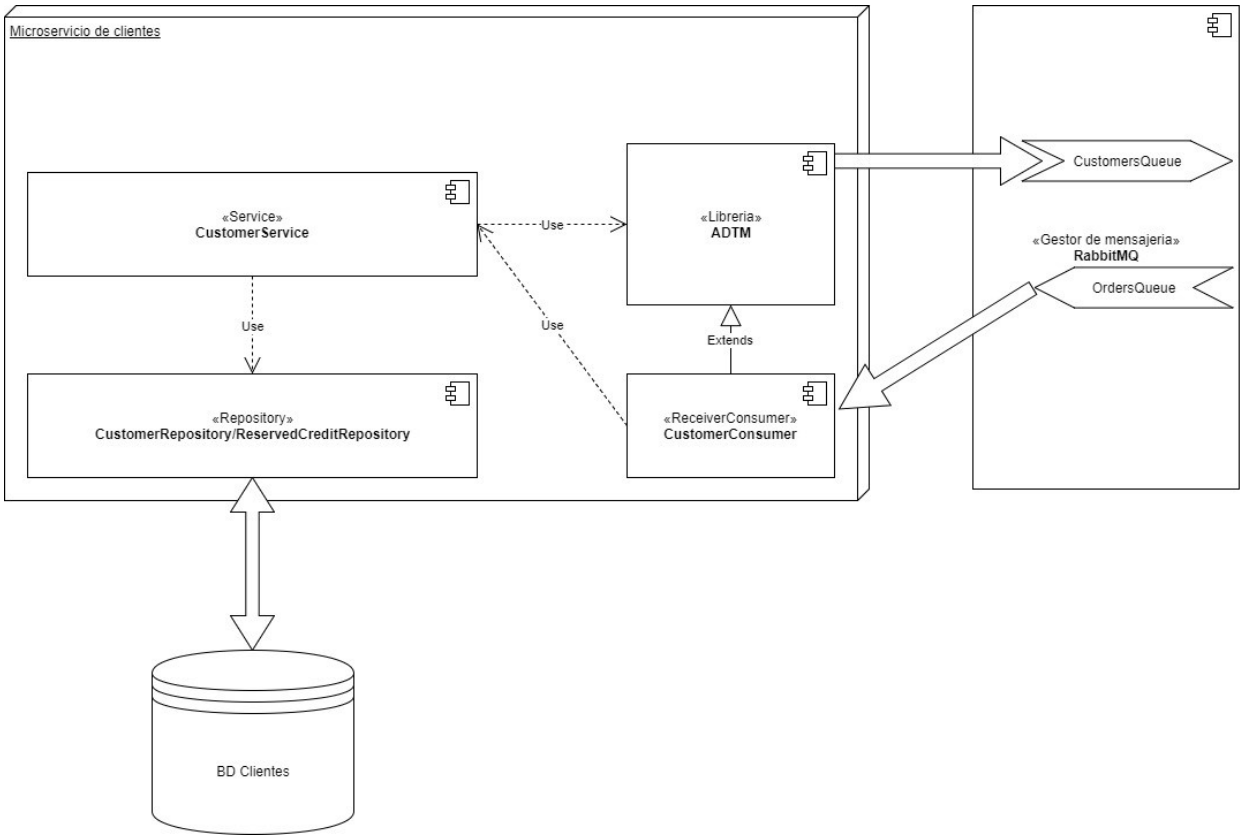


Ilustración 66 - Diagrama de componentes de ADTM integrado en el microservicio de clientes

Y llevará a cabo el proceso de la siguiente manera:

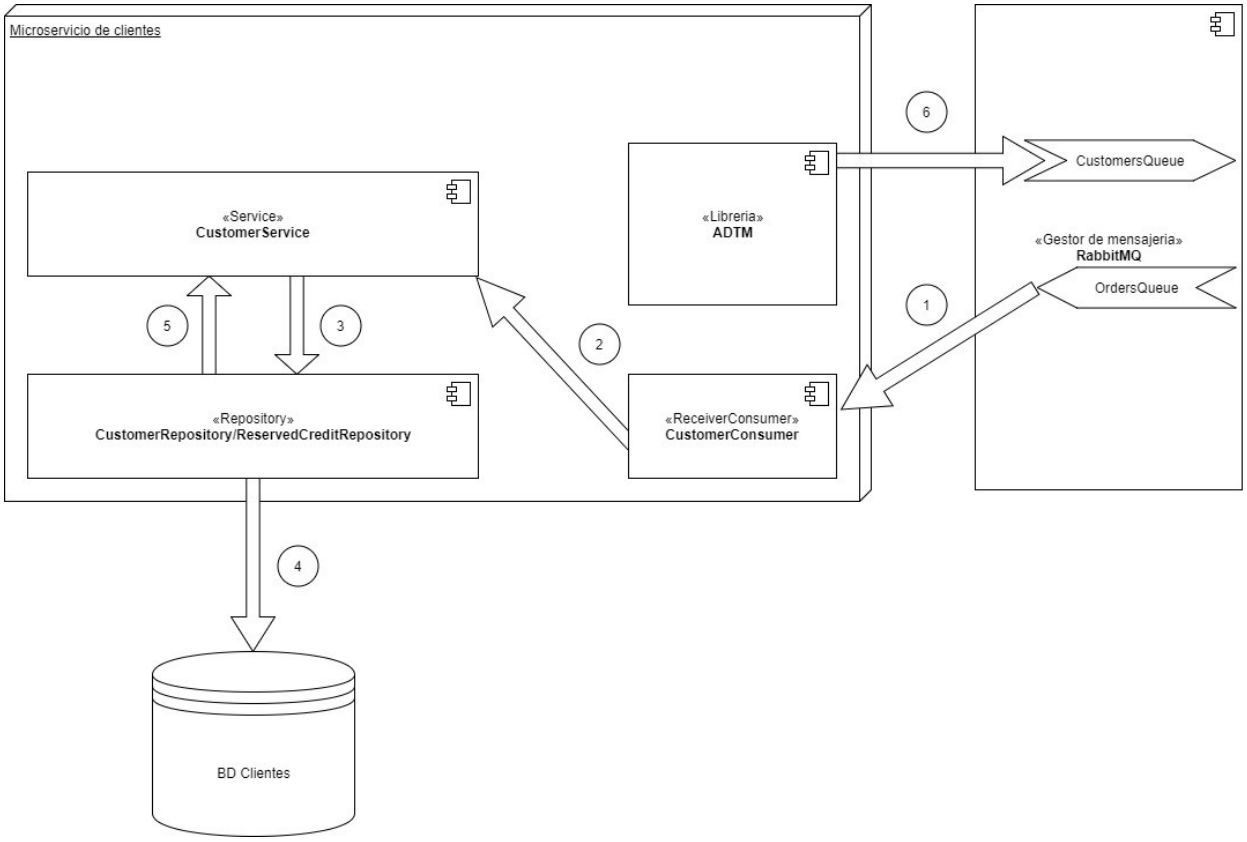


Ilustración 67 - Diagrama de funcionamiento de ADTM integrado en el microservicio de clientes

1. El componente ReceiverConsumer, que en nuestro caso es CustomerConsumer, y que implementa por esta relación de herencia la lógica a realizar para determinar si la respuesta a devolver al emisor es de conformidad o no, recibe una transacción por parte de este a través de la cola establecida
2. El consumidor llama al servicio encargado de la lógica de negocio (CustomerService) para que compruebe si el pedido puede ser tramitado de acuerdo con la situación de crédito del cliente que lo solicita.
3. El servicio invoca a los repositorios (CustomerRepository y ReservedCreditRepository) para recuperar la situación de crédito del cliente que ha creado el pedido, es decir, su límite de crédito y la suma de créditos que ya ha solicitado previamente para otros pedidos
4. Los repositorios JPA solicitan la información requerida a la base de datos de clientes y ésta se la devuelve
5. El repositorio le devuelve la información solicitada al servicio
6. El componente integrado ADTM, y de una manera transparente para el microservicio receptor, envía la respuesta de la transacción al emisor a través de la cola establecida por ambas partes para este fin

Desde el punto de vista de la implementación, y de una manera más genérica, para poder ser aplicado a otros sistemas que igualmente se ajusten a una arquitectura de microservicios, el microservicio encargado de recibir la transacción y enviar una respuesta al emisor simplemente tendrá que:

- Igual que en el caso del emisor, importar la librería ADTM mediante la anotación @Import en su clase de arranque e inicialización (marcada mediante la anotación @SpringBootApplication)
- Extender la clase abstracta ReceiverConsumer de ADTM para implementar la lógica de negocio a realizar cuando se reciba la transacción por parte del emisor, e indicarle en la respuesta si la transacción se puede dar por correcta o no
- Indicar al componente TransactionManager de ADTM que se espera recibir una transacción en una cola determinada. Para ello, aunque podría haber otros enfoques en la manera y momento de hacer la llamada, se declara un método anotado mediante @Bean en la clase de inicialización con el objetivo de que se ejecute en el momento del arranque de la aplicación, que recibe el objeto TransactionManager inyectado automáticamente por Spring, y se llama a su método receiveTransaction() al que se le pasarán como parámetros el nombre de la cola donde se espera recibir la transacción del emisor y el objeto del tipo RecevierConsumer encargado de recibir y procesar la transacción iniciada por el emisor.

El código fuente de las principales clases Java involucradas en la implementación de este microservicio se puede encontrar en el apartado Servicio receptor (Microservicio de clientes) como parte del Anexo 3 – Código fuente de la solución integrada en el contexto del problema

Conclusiones

La arquitectura de microservicios se presenta, con todas sus ventajas y grandes beneficios, como una arquitectura software capaz de hacer frente a los retos y problemas de muchos de los desarrollos software que, con un nivel de exigencia y complejidad cada vez mayor, surgen en la actualidad. Pero, lejos de ser la respuesta inmediata y la panacea al desarrollo de cualquier sistema software, presenta, como es lógico, en mayor o menor medida respecto de otros estilos de arquitectura, una serie de inconvenientes y desventajas que hay que tener en muy en cuenta a la hora de descantarse por elegir este modelo de implementación.

Tal y como hemos podido comprobar, uno de estos problemas, que es en el que nos hemos centrado para el desarrollo de este trabajo, es el que surge cuando los microservicios en su interacción optan por un modo de comunicación asíncrono. En este contexto, no sólo surgen importantes retos, tanto desde la perspectiva de la propia implementación, ya que hay que desarrollar toda la lógica necesaria en los microservicios involucrados en la comunicación, como desde la perspectiva de los componentes presentes en el sistema, ya que hay que añadir el gestor de mensajería AMQP como un nuevo componente a tener en cuenta. Además, tal y como ha quedado reflejado, aparecen problemas e inconvenientes adicionales en tiempo de ejecución que lo convierten en una fuente de errores e inconsistencias.

Algunos de estos problemas, que a nuestro parecer han sido los más importantes y, por lo tanto, los más acuciantes de resolver son:

- Falta de atomicidad en el proceso de inicio de la transacción por parte del microservicio emisor (con sus correspondientes operaciones de lógica de negocio) y el envío del mensaje pertinente al agente de mensajería encargado de transmitirlo al servicio receptor. Lo que puede dar lugar a que los objetos de negocio involucrados en la transacción permanezcan de manera indefinida en un estado inconsistente
- Falta de previsión y control ante la posibilidad de que el componente emisor no reciba respuesta alguna del microservicio al que va dirigido la transacción. Lo que puede provocar, igual que en el caso anterior, que los objetos de negocio involucrados en la transacción permanezcan en un estado inconsistente de manera indefinida.
- Falta de encapsulamiento e independencia del dominio y lógica de negocio de los microservicios involucrados en la transacción. Lo que implica por si solo romper uno de los principios básicos que caracterizan la arquitectura de microservicios.

Así, partiendo de este contexto, con el objetivo de dar una solución a esta problemática y utilizando las herramientas y conocimientos a nuestro alcance, hemos conseguido desarrollar una librería Java para la gestión de transacciones distribuidas (ADTM); que, integrada de una manera sencilla en los microservicios implicados en la comunicación, resuelve de una manera eficiente los problemas expuestos. Dando lugar a los siguientes ventajas o logros respecto de la situación inicial:

- Atomicidad en el proceso que abarca desde el inicio de la transacción por parte del servicio emisor hasta el envío definitivo del mensaje al agente de mensajería encargado de su transmisión. Esta característica, aunque no evita que el objeto de negocio involucrado en la transacción permanezca durante un tiempo en un estado inconsistente inicial, sí evita que sea de manera indefinida; ya que el mensaje será enviado tan pronto como el agente de mensajería esté disponible.
- Se contempla como parte de la propia lógica del proceso de comunicación la posibilidad de que el microservicio que recibe la transacción, por el motivo que sea, no mande en ningún momento una respuesta al servicio emisor. Y, en ese caso, se ejecutan de manera automática las acciones oportunas que éste haya establecido previamente. De esa manera, evitamos el acoplamiento, y con ello la dependencia excesiva, entre el componente emisor y el componente receptor, y

devolvemos el flujo de funcionamiento al emisor en el caso de que la transacción no se pueda terminar con normalidad.

- Aislamos el alcance del dominio de negocio de cada uno de los microservicios involucrados en la comunicación a su propio contexto de ejecución. O, dicho de otra forma, descargamos al servicio receptor de la responsabilidad de la lógica de negocio de la transacción que no le corresponde, y la trasladamos al servicio emisor donde sí debería estar. De esa forma, mantenemos el principio de encapsulación ya mencionado, y evitamos segregar entre varios microservicios la lógica de negocio de un dominio/servicio particular.

Además, apuntar que, en la medida que el código fuente de esta solución está disponible públicamente y de manera gratuita en el repositorio GitHub: https://github.com/franciscocilleruelo/ADTM.git, cualquier profesional del software puede contribuir libremente para mejorarla o adaptarla a sus propias necesidades.

Bibliografía

- Gutierrez, F. (2016). Pro Spring Boot [recurso electrónico]. 1st ed. Berkeley, CA: Apress.
- Keogh, J. (2003). J2EE. Mc Graw Hill.
- Richardson, C. and Smith, F. (2016). Microservices. From design to deployment. 1st ed. NGINX.
- Rv, R. (2016). Spring microservices. 1st ed. Packt Publishing.
- Walls, C. (2011). Spring. 3rd ed. Anaya.
- Williams, N. (2014). Professional Java for Web Applications. 1st ed. Wrox.
- Wolff, E. (2017). Microservices. 1st ed. Addison-Wesley.
- Yang, D. (2013). Java Persistence with JPA 2.1. Outskirts Press.
- Docs.microsoft.com. (2017). Asynchronous message-based communication. [online] Available at: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/asynchronous-message-based-communication [Accessed 31 Aug. 2017].
- dzone.com. (2017). Building Microservices: Inter-Process Communication in a Microservices Architecture - DZone Cloud. [online] Available at: https://dzone.com/articles/building-microservices-inter-process-communication-1 [Accessed 31 Aug. 2017].
- dzone.com. (2017). Event Driven Microservices Patterns - DZone Integration. [online] Available at: https://dzone.com/articles/event-driven-microservices-patterns [Accessed 31 Aug. 2017].
- dzone.com. (2017). Microservices in Practice: From Architecture to Deployment - DZone Cloud. [online] Available at: https://dzone.com/articles/microservices-in-practice-1 [Accessed 31 Aug. 2017].
- dzone.com. (2017). Microservices with Spring - DZone Integration. [online] Available at: https://dzone.com/articles/microservices-with-spring [Accessed 31 Aug. 2017].
- Lewis, J. and Fowler, M. (2017). Microservices. [online] martinfowler.com. Available at: https://www.martinfowler.com/articles/microservices.html [Accessed 31 Aug. 2017].
- microservices.io. (2017). Microservices Pattern: Event-driven architecture. [online] Available at: http://microservices.io/patterns/data/event-driven-architecture.html [Accessed 31 Aug. 2017].
- microservices.io. (2017). Microservices Pattern: Messaging. [online] Available at: http://microservices.io/patterns/communication-style/messaging.html [Accessed 31 Aug. 2017].
- microservices.io. (2017). Microservices Pattern: Sagas. [online] Available at: http://microservices.io/patterns/data/saga.html [Accessed 31 Aug. 2017].
- Nadareishvili, I., Mitra, R., McLarty, M. and Amundsen, M. (2016). Microservice Architecture. 1st ed. O'Reilly Media, Inc.
- Programmatic Ponderings. (2017). Eventual Consistency: Decoupling Microservices with Spring AMQP and RabbitMQ. [online] Available at: https://programmaticponderings.com/2017/05/15/eventual-consistency-decoupling-microservices-with-spring-amqp-and-rabbitmq/ [Accessed 31 Aug. 2017].
- Vilas, F. (2017). Asynchronous APIs in Choreographed Microservices | Nordic APIs |. [online] Nordic APIs. Available at: https://nordicapis.com/asynchronous-apis-in-choreographed-microservices/ [Accessed 31 Aug. 2017].

Anexo 1 – Código fuente del ejemplo de arquitectura de microservicios con transacciones distribuidas asíncronas

Microservicio de pedidos

Controlador web MVC: OrderMvcController

```
package es.uned.master.software.tfm.microservice.order.controller;

import java.util.Map;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import es.uned.master.software.tfm.microservice.order.jpa.entity.Order;
import es.uned.master.software.tfm.microservice.order.service.OrderService;

@Controller
public class OrderMvcController {

	private static final Logger log =
LoggerFactory.getLogger(OrderMvcController.class);

	@Autowired
	private OrderService orderService;

	@RequestMapping(path={"/", "/orders/list"}, method=RequestMethod.GET)
	public ModelAndView listOrders(){
		log.info("Se solicita la vista de listado de pedidos");
		ModelAndView mav = new ModelAndView("listOrders");
		mav.addObject("orders", orderService.findAll());
		return mav;
	}

	@RequestMapping(path= "/orders/create", method=RequestMethod.GET)
	public ModelAndView addOrder(Map<String, Object> model) {
		log.info("Se solicita la vista para crear un nuevo pedido");
		ModelAndView mav = new ModelAndView("createOrder");
		mav.addObject("order", new Order());
		return mav;
	}

	@RequestMapping(path="/orders/create", method=RequestMethod.POST)
	public ModelAndView createOrder(@ModelAttribute("order") Order order){
		log.info("Se realiza la llamada desde el cliente para crear un
nuevo pedido");
		Order orderCreated = orderService.createOrder(order);
		ModelAndView mav = new ModelAndView("orderCreated");
		mav.addObject("order", orderCreated);
		return mav;
	}

}
```

Clase Java 1 - OrderMvcController. Controlador web MVC del microservicio de pedidos

Controlador REST: OrderRestController

```
package es.uned.master.software.tfm.microservice.order.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import es.uned.master.software.tfm.microservice.order.jpa.entity.Order;
import es.uned.master.software.tfm.microservice.order.service.OrderService;

@RestController
public class OrderRestController {

	private static final Logger log =
LoggerFactory.getLogger(OrderRestController.class);

	@Autowired
	private OrderService orderService;

	@RequestMapping(path="/orders", method=RequestMethod.GET)
	public List<Order> listOrders(@ModelAttribute("order") Order order){
		log.info("Se llama al servicio REST para listar los pedidos");
		return orderService.findAll();
	}

	@RequestMapping(path="/orders", method=RequestMethod.POST)
	public Order createOrder(@ModelAttribute("order") Order order){
		log.info("Se llama al servicio REST para crear un nuevo pedido");
		return orderService.createOrder(order);
	}


}
```

Clase Java 2 - OrderRestController. Controlador REST del microservicio de pedidos

Servicio con la lógica de negocio: OrderService

```
package es.uned.master.software.tfm.microservice.order.service;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import es.uned.master.software.tfm.microservice.order.amqp.Producer;
import es.uned.master.software.tfm.microservice.order.jpa.entity.Order;
import
es.uned.master.software.tfm.microservice.order.jpa.repository.OrderRepository;

@Service
```

```
@Transactional
public class OrderService {

	private static final Logger log =
LoggerFactory.getLogger(OrderService.class);

	@Autowired
	private OrderRepository orderRepository;

	@Autowired
	private Producer producer;

	@Value("${queue.orders.name}")
	private String ordersQueueName;

	public List<Order> findAll(){
		log.info("Busqueda de todos los pedidos");
		return orderRepository.findAll();
	}

	public Order createOrder(Order order){
		log.info("Se inicializa el estado del pedido a crear como nuevo
(NEW)");
		order.setStatus("NEW");
		log.info("Se guarda el nuevo pedido");
		orderRepository.save(order);
		log.info("Se envia el pedido a la cola {} para ser procesado por
el servicio de clientes", ordersQueueName);
		producer.sendTo(ordersQueueName, order);
		return order;
	}

	public void udpate(Order order){
		log.info("Se actualiza el pedido");
		orderRepository.save(order);
	}

}
```

Clase Java 3 - OrderService. Servicio con la lógica de negocio del microservicio de pedidos

Repositorio JPA: OrderRepository

```
package es.uned.master.software.tfm.microservice.order.jpa.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import es.uned.master.software.tfm.microservice.order.jpa.entity.Order;

@RepositoryRestResource
public interface OrderRepository extends JpaRepository<Order, Long> {

}
```

Clase Java 4 - OrderRepository. Repositorio JPA del microservicio de pedidos

Objeto de negocio: Order

```
package es.uned.master.software.tfm.microservice.order.jpa.entity;

import java.io.Serializable;
```

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "ORDERS")
public class Order implements Serializable {

	private static final long serialVersionUID = 5171845146709995115L;

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long orderId;
	private Long customerId;
	private String status;
	private int total;

	public Order() {
		super();
	}

	public Order(String status, int total) {
		super();
		this.status = status;
		this.total = total;
	}

	public Long getOrderId() {
		return orderId;
	}

	public void setOrderId(Long orderId) {
		this.orderId = orderId;
	}

	public Long getCustomerId() {
		return customerId;
	}

	public void setCustomerId(Long customerId) {
		this.customerId = customerId;
	}

	public String getStatus() {
		return status;
	}

	public void setStatus(String status) {
		this.status = status;
	}

	public int getTotal() {
		return total;
	}

	public void setTotal(int total) {
		this.total = total;
	}
```

```
}
```

Clase Java 5 - Order. Objeto de negocio del microservicio de pedidos

Productor AMQP: Producer

```
package es.uned.master.software.tfm.microservice.order.amqp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

@Component
public class Producer {

	private static final Logger log =
LoggerFactory.getLogger(Producer.class);

	@Autowired
	private RabbitTemplate rabbitTemplate;

	public void sendTo(String routingKey, String message){
		this.rabbitTemplate.convertAndSend(routingKey, message);
	}

	public void sendTo(String routingKey, Object object){
		ObjectMapper mapper = new ObjectMapper();
		try {
			String message = mapper.writeValueAsString(object);
			this.rabbitTemplate.convertAndSend(routingKey, message);
			log.info("Mensaje {} enviado satisfactoriamente a la cola
{}", message, routingKey);
		} catch (JsonProcessingException jsonEx){
			log.error("Error al intentar convertir un objeto a JSON para
ser transmitido a la cola {}", routingKey);
		} catch (Exception ex){
			log.error("Error en el envio del mensaje a la cola {}",
routingKey);
		}
	}


}
```

Clase Java 6 - Producer. Productor AMQP del microservicio de pedidos

Consumidor AMQP: Consumer

```
package es.uned.master.software.tfm.microservice.order.amqp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.databind.ObjectMapper;
```

```
import es.uned.master.software.tfm.microservice.order.jpa.entity.Order;
import es.uned.master.software.tfm.microservice.order.service.OrderService;

@Component
public class Consumer {

	private static final Logger log =
LoggerFactory.getLogger(Consumer.class);

	@Autowired
	private OrderService orderService;

	@RabbitListener(queues="${queue.customers.name}")
	public void handleMessage(String message){
		log.info("Recibido mensaje {}", message);
		ObjectMapper mapper = new ObjectMapper();
		try {
			Order order = mapper.readValue(message, Order.class);
			orderService.udpate(order);
		} catch (Exception ex){
			log.error("Error en la conversión del mensaje JSON {} a
objeto partir del objecto del tipo", message, Order.class);
		}

	}
}
```

Clase Java 7 - Consumer. Consumidor AMQP del microservicio de pedidos

Clase de arranque e inicialización: OrderServiceApplication

Esta clase servirá para arrancar el microservicio como una aplicación Spring con su propio contenedor JEE Tomcat embebido, y hacer las inicializaciones adicionales oportunas (instanciación de colas AMQP).

```
package es.uned.master.software.tfm.microservice.order;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class OrderServiceApplication {

	private static final Logger log =
LoggerFactory.getLogger(OrderServiceApplication.class);

	public static void main(String[] args) {
		log.info("Arrancado microservicio de pedidos");
		SpringApplication.run(OrderServiceApplication.class, args);
	}

	@Value("${queue.orders.name}")
	private String ordersQueueName;

	@Bean
	public Queue ordersQueue(){
		return new Queue(ordersQueueName, false, false, false);
```

```
	}

	@Value("${queue.customers.name}")
	private String customersQueueName;

	@Bean
	public Queue customersQueue(){
		return new Queue(customersQueueName, false, false, false);
	}

}
```

Clase Java 8 - OrderServiceApplication. Clase de arranque e inicialización del microservicio de pedidos

Microservicio de clientes

Servicio con la lógica de negocio: CustomerService

```
package es.uned.master.software.tfm.microservice.customer.service;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;

import es.uned.master.software.tfm.microservice.customer.amqp.Producer;
import es.uned.master.software.tfm.microservice.customer.jpa.entity.Customer;
import es.uned.master.software.tfm.microservice.customer.jpa.entity.Order;
import
es.uned.master.software.tfm.microservice.customer.jpa.entity.ReservedCredit;
import
es.uned.master.software.tfm.microservice.customer.jpa.entity.ReservedCreditId;
import
es.uned.master.software.tfm.microservice.customer.jpa.repository.CustomerRepos
itory;
import
es.uned.master.software.tfm.microservice.customer.jpa.repository.ReservedCredi
tRepository;

@Service
@Transactional
public class CustomerService {

	private static final Logger log =
LoggerFactory.getLogger(CustomerService.class);

	@Autowired
	private CustomerRepository customerRepository;

	@Autowired
	private ReservedCreditRepository reservedCreditRepository;

	@Value("${queue.customers.name}")
	private String customersQueueName;

	@Autowired
	private Producer producer;
```

```
	public void insertExampleData(){
		customerRepository.save(new Customer(300));
		customerRepository.save(new Customer(8000));
		log.info("Inicializado repositorio de clientes con datos de
ejemplo");
	}

	public List<Customer> findAll(){
		log.info("Busqueda de todos los clientes");
		return customerRepository.findAll();
	}

	public void checkLimit(Order order){
		log.info("Recuperamos el cliente {} asociado al pedido",
order.getCustomerId());
		Customer customer =
customerRepository.findOne(order.getCustomerId());
		int reservedCreditNow = 0;
		String reservedCreditNowS =
reservedCreditRepository.sumReserverCreditByCustomerId(order.getCustomerId());
		if (StringUtils.hasText(reservedCreditNowS)){
			reservedCreditNow = Integer.valueOf(reservedCreditNowS);
		}
		log.info("El credito reservado del cliente {} para otros pedidos
es de {}", order.getCustomerId(), reservedCreditNow);
		if (customer != null && customer.getCreditLimit() >=
order.getTotal() + reservedCreditNow){
			log.info("El limite de credito para el cliente {} es
superior a la suma de la cantidad solicitada para el pedido ({}) mas el
credito reservado para otros pedidos ({})"
				, order.getCustomerId(), order.getOrderId(),
reservedCreditNow);
			log.info("Se establece el pedido como abierto (OPEN)");
			order.setStatus("OPEN");
			ReservedCreditId reservedCreditId = new
ReservedCreditId(order.getOrderId(), order.getCustomerId());
			ReservedCredit reservedCredit = new
ReservedCredit(reservedCreditId, order.getTotal());
			log.info("Se reserva el credito {} para el pedido {} del
cliente {}", reservedCredit.getTotalReserved(),

	reservedCredit.getReservedCreditId().getOrderId(),
reservedCredit.getReservedCreditId().getCustomerId());
			reservedCreditRepository.save(reservedCredit);
		} else { // No existe el cliente o la cantidad del pedido supera
el credito
			log.info("El limite de credito es inferior a la cantidad
solicitada por el pedido");
			log.info("Se establece el pedido como rechazado
(REJECTED)");
			order.setStatus("REJECTED");
		}
		log.info("Se envia el pedido con su estado modificado a la cola {}
para ser procesador por el servicio de pedidos", customersQueueName);
		producer.sendTo(customersQueueName, order);
	}

}
```

Clase Java 9 - CustomerService. Servicio con la lógica de negocio del microservicio de clientes

Repositorios JPA: CustomerRepository y ReservedCreditRepository

```
package es.uned.master.software.tfm.microservice.customer.jpa.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import es.uned.master.software.tfm.microservice.customer.jpa.entity.Customer;

@RepositoryRestResource
public interface CustomerRepository extends JpaRepository<Customer, Long> {

}
```

Clase Java 10 - CustomerRepository. Repositorio JPA del microservicio de clientes

```
package es.uned.master.software.tfm.microservice.customer.jpa.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import
es.uned.master.software.tfm.microservice.customer.jpa.entity.ReservedCredit;
import
es.uned.master.software.tfm.microservice.customer.jpa.entity.ReservedCreditId;

@RepositoryRestResource
public interface ReservedCreditRepository extends
JpaRepository<ReservedCredit, ReservedCreditId> {

	@Query("SELECT sum(rc.totalReserved) FROM ReservedCredit rc WHERE
rc.reservedCreditId.customerId=?1")
	public String sumReserverCreditByCustomerId(Long customerId);

}
```

Clase Java 11 - ReservedCreditRepository. Repositorio JPA del microservicio de clientes

Objetos de negocio: Customer y ReservedCredit

```
package es.uned.master.software.tfm.microservice.customer.jpa.entity;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable {

	private static final long serialVersionUID = -6341360796725637534L;

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long customerId;
	private int creditLimit;

	public Customer() {
```

```
            super();
      }

      public Customer(int creditLimit) {
            super();
            this.creditLimit = creditLimit;
      }

      public Long getCustomerId() {
            return customerId;
      }

      public void setCustomerId(Long customerId) {
            this.customerId = customerId;
      }

      public int getCreditLimit() {
            return creditLimit;
      }

      public void setCreditLimit(int creditLimit) {
            this.creditLimit = creditLimit;
      }

}
```

Clase Java 12 - Customer. Objeto de negocio del microservicio de clientes

```
package es.uned.master.software.tfm.microservice.customer.jpa.entity;

import java.io.Serializable;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "RESERVED_CREDIT")
public class ReservedCredit implements Serializable {

      private static final long serialVersionUID = -5895756852084946427L;

      @EmbeddedId
      private ReservedCreditId reservedCreditId;
      private int totalReserved;

      public ReservedCredit() {
            super();
      }

      public ReservedCredit(ReservedCreditId reservedCreditId, int
totalReserved) {
            super();
            this.reservedCreditId = reservedCreditId;
            this.totalReserved = totalReserved;
      }

      public ReservedCreditId getReservedCreditId() {
            return reservedCreditId;
      }

      public void setReservedCreditId(ReservedCreditId reservedCreditId) {
```

```
            this.reservedCreditId = reservedCreditId;
      }

      public int getTotalReserved() {
            return totalReserved;
      }

      public void setTotalReserved(int totalReserved) {
            this.totalReserved = totalReserved;
      }

}
```

```
package es.uned.master.software.tfm.microservice.customer.jpa.entity;

import java.io.Serializable;

import javax.persistence.Embeddable;

@Embeddable
public class ReservedCreditId implements Serializable {

      private static final long serialVersionUID = 2234618454466060795L;

      private Long orderId;
      private Long customerId;

      public ReservedCreditId() {
            super();
      }

      public ReservedCreditId(Long orderId, Long customerId) {
            super();
            this.orderId = orderId;
            this.customerId = customerId;
      }

      public Long getOrderId() {
            return orderId;
      }

      public Long getCustomerId() {
            return customerId;
      }

}
```

Clase Java 13 - ReservedCredit (y ReservedCreditId). Objeto de negocio del microservicio de clientes

Productor AMQP: Producer

```
package es.uned.master.software.tfm.microservice.customer.amqp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

@Component
```

```
public class Producer {

	private static final Logger log =
LoggerFactory.getLogger(Producer.class);

	@Autowired
	private RabbitTemplate rabbitTemplate;

	public void sendTo(String routingKey, String message){
		this.rabbitTemplate.convertAndSend(routingKey, message);
	}

	public void sendTo(String routingKey, Object object){
		ObjectMapper mapper = new ObjectMapper();
		try {
			String message = mapper.writeValueAsString(object);
			this.rabbitTemplate.convertAndSend(routingKey, message);
			log.info("Mensaje {} enviado satisfactoriamente a la cola
{}", message, routingKey);
		} catch (JsonProcessingException jsonEx){
			log.error("Error al intentar convertir un objeto a JSON para
ser transmitido a la cola {}", routingKey);
		} catch (Exception ex){
			log.error("Error en el envio del mensaje a la cola {}",
routingKey);
		}
	}


}
```

Clase Java 14 - Producer. Productor AMQP del microservicio de clientes

Consumidor AMQP: Consumer

```
package es.uned.master.software.tfm.microservice.customer.amqp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.databind.ObjectMapper;

import es.uned.master.software.tfm.microservice.customer.jpa.entity.Order;
import
es.uned.master.software.tfm.microservice.customer.service.CustomerService;

@Component
public class Consumer {

	private static final Logger log =
LoggerFactory.getLogger(Consumer.class);

	@Autowired
	private CustomerService customerService;

	@RabbitListener(queues="${queue.orders.name}")
	public void handleMessage(String message){
		log.info("Recibido mensaje {}", message);
		ObjectMapper mapper = new ObjectMapper();
```

```
            try {
                  Order order = mapper.readValue(message, Order.class);
                  customerService.checkLimit(order);
            } catch (Exception ex){
                  log.error("Error en la conversión del mensaje JSON {} a
objeto partir del objecto del tipo", message, Order.class);
            }

      }
}
```

Clase Java 15 - Consumer. Consumidor AMQP del microservio de clientes

Clase de arranque e incialización: CustomerServiceApplication

Esta clase servirá para arrancar el microservicio como una aplicación Spring con su propio contenedor JEE Tomcat embebido, y hacer las inicializaciones adicionales oportunas (carga de datos de ejemplo e instanciación de colas AMQP).

```
package es.uned.master.software.tfm.microservice.customer;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import
es.uned.master.software.tfm.microservice.customer.service.CustomerService;

@SpringBootApplication
public class CustomerServiceApplication {

      private static final Logger log =
LoggerFactory.getLogger(CustomerServiceApplication.class);

      public static void main(String[] args) {
            log.info("Arrancado microservicio de clientes");
            SpringApplication.run(CustomerServiceApplication.class, args);
      }

      @Bean
      CommandLineRunner init(CustomerService customerService){
            return args -> {
                  customerService.insertExampleData();
                  customerService.findAll().forEach(entry ->
log.info(entry.toString()));
            };
      }

      @Value("${queue.orders.name}")
      private String ordersQueueName;

      @Bean
      public Queue ordersQueue(){
            return new Queue(ordersQueueName, false, false, false);
      }

      @Value("${queue.customers.name}")
```

```
	private String customersQueueName;

	@Bean
	public Queue customersQueue(){
		return new Queue(customersQueueName, false, false, false);
	}

}
```

Clase Java 16 - CustomerServiceApplication. Clase de arranque e inicialización del microservicio de clientes

Anexo 2- Código fuente de la implementación de la solución: Gestor de transacciones distribuidas asíncronas (ADTM)

Componente principal de la librería: DistributedTransactionManager

```
package es.uned.master.software.tfm.adtm.manager;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import es.uned.master.software.tfm.adtm.amqp.receiver.ReceiverConsumer;
import es.uned.master.software.tfm.adtm.entity.Transaction;
import es.uned.master.software.tfm.adtm.service.DistributedTransactionService;

/**
 * Clase principal para la gestion de transacciones distribuidas asincronas
 *
 * @author Francisco Cilleruelo
 */
@Component
public class DistributedTransactionManager {

	@Autowired
	private DistributedTransactionService distributedTransactionService;

	/**
	 * Metodo invocado por el emisor de una transaccion para comenzarla
	 *
	 * @param transaction Transaccion a enviar
	 */
	public void sendTransaction(Transaction<?> transaction){
		distributedTransactionService.startTransaction(transaction);
	}

	/**
	 * Metodo invocado por el receptor de una transaccion para recibirla
	 *
	 * @param requestQueueName Nombre de la cola donde se espera recibir la
transaccion enviada por el emisor
	 * @param receiverConsumer Componente encargado de recibir y procesar la
transaccion recibida
	 */
	public void receiveTransaction(String requestQueueName,
ReceiverConsumer<?> receiverConsumer){
		distributedTransactionService.receiveTransaction(requestQueueName,
receiverConsumer);
	}

}
```

Clase Java 17 - DistributedTransactionManager. Componente principal de la librería ADTM

Servicio con la lógica de negocio: DistributedTransactionService

```
package es.uned.master.software.tfm.adtm.service;

import java.util.Date;
import java.util.List;
import java.util.Map;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import
org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import es.uned.master.software.tfm.adtm.amqp.Producer;
import
es.uned.master.software.tfm.adtm.amqp.receiver.ReceiverConsumer;
import es.uned.master.software.tfm.adtm.amqp.sender.SenderConsumer;
import es.uned.master.software.tfm.adtm.amqp.util.AmpqUtil;
import es.uned.master.software.tfm.adtm.entity.Transaction;
import es.uned.master.software.tfm.adtm.entity.TransactionElement;
import es.uned.master.software.tfm.adtm.entity.TransactionStatus;
import es.uned.master.software.tfm.adtm.exception.SendingException;
import es.uned.master.software.tfm.adtm.jpa.entity.TransactionData;
import
es.uned.master.software.tfm.adtm.jpa.repository.TransactionDataRepos
itory;
import es.uned.master.software.tfm.adtm.task.ResponseCheckingTask;

/**
 * Servicio (transaccional) encargado de la logica de negocio del
gestor de transacciones
 *
 * @author Francisco Cilleruelo
 */
@Service
@Transactional
public class DistributedTransactionService {

	private static final Logger log =
LoggerFactory.getLogger(DistributedTransactionService.class);

	@Autowired
	private TransactionDataRepository transactionDataRepository;

	@Autowired
	private ThreadPoolTaskScheduler taskScheduler;

	@Autowired
	private Producer producer;

	@Autowired
	private AmpqUtil ampqUtil;

	@Autowired
	private Map<Long, SenderConsumer> senderConsumerRepository;

	@Autowired
	private AutowireCapableBeanFactory beanFactory;

	/**
	 * Metodo invocado cuando el emisor no ha recibido respuesta
del receptor pasado el tiempo limite establecido para la transaccion
	 *
	 * @param transactionData Transacction sin respuesta a
actualizar
	 * @return Transaccion actualizada
```

```
        */
       public TransactionData
transactionResponseNotReceived(TransactionData transactionData){
             log.info("Indicamos que la transacción {} no ha recibido
respuesta", transactionData.getTransactionDataId());

       transactionData.setStatus(TransactionStatus.NOT_RECEIVED.toStri
ng());
             transactionData.setResponseCheckedDate(new Date());
             log.info("Ejecutamos el rollback definido en la
transaccion");
             if
(senderConsumerRepository.containsKey(transactionData.getTransaction
DataId())){
                   log.info("Ejecutamos el rollback asociado a la
transacción {}", transactionData.getTransactionDataId());

       senderConsumerRepository.get(transactionData.getTransactionData
Id()).rollback(transactionData.getObjectTransmited());
                   log.info("Eliminamos el componente que debería
recibir la respuesta de la transaccion {}",
transactionData.getTransactionDataId());

       senderConsumerRepository.remove(transactionData.getTransactionD
ataId());
             }
             return transactionDataRepository.save(transactionData);
       }

       /**
        * Metodo invocado cuando el emisor ha recibido una respuesta
por parte del receptor
        *
        * @param transaction Transaccion con respuesta a actualizar
        * @return Transaccion actualizada
        */
       public TransactionData
transactionResponseReceived(TransactionElement<?> transaction){
             log.info("Se obtiene la transacccion {} persistida para
actualizar su estado", transaction.getTransactionReference());
             TransactionData transactionData =
transactionDataRepository.findOne(transaction.getTransactionReferenc
e());
             transactionData.setReceivedDate(new Date());

       transactionData.setStatus(transaction.getStatus().toString());

       transactionData.setAdditionalInfo(transaction.getAdditionalInfo
());
             log.info("Eliminamos el componente ya ha recibido la
respuesta de la transaccion {}",
transactionData.getTransactionDataId());

       senderConsumerRepository.remove(transactionData.getTransactionD
ataId());
             return transactionDataRepository.save(transactionData);
       }

       /**
        * Metodo invocado para empezar una transaccion
        *
```

```
         * @param transaction Transaccion a empezar
         * @return Transaccion persistida
         */
        public TransactionData startTransaction(Transaction<?>
transaction){
                TransactionData transactionData = new
TransactionData(transaction);
                log.info("Se empieza una nueva transacción");
                transactionData.setStartDate(new Date());
                log.info("Se guarda para ser enviado de acuerdo con el
proceso de envio de transacciones establecido periodicamente");

        transactionData.setStatus(TransactionStatus.TO_BE_SENT.toString
());
                TransactionData transactionDataSaved =
transactionDataRepository.save(transactionData);
                log.info("Asociamos para la transaccion recien creada {}
el listener de la cola donde espera recibir la respuesta",
transactionDataSaved.getTransactionDataId());

        beanFactory.autowireBean(transaction.getSenderConsumer());

        senderConsumerRepository.put(transactionData.getTransactionData
Id(), transaction.getSenderConsumer());
                return transactionDataSaved;
        }

        /**
         * @return Transaccion pendientes de ser enviadas
         */
        public List<TransactionData> getTransactionsToBeSent(){
                log.info("Se obtienen las transacciones pendientes de ser
enviadas");
                return
transactionDataRepository.getTransactionToBeSent();
        }

        /**
         * Metodo para recuperar una transaccion persistida por su
identificador
         *
         * @param transactionId Identificador de la transaccion
         * @return La transaccion persistida para el identificador
indicado
         */
        public TransactionData getTransactionDataById(Long
transactionId){
                log.info("Recuperamos los datos asociados a la
transacción, incluyendo el Executor");
                return transactionDataRepository.findOne(transactionId);
        }

        /**
         * Metodo invocado para enviar una transaccion
         *
         * @param transactionData Transaccion a enviar
         * @throws SendingException Excepcion por error de envio
         */
        public void sendTransaction(TransactionData transactionData)
throws SendingException{
                try {
```

```
                    log.info("Se procede a enviar la transacción {} a
su cola de envio {}", transactionData.getTransactionDataId(),
                                transactionData.getRequestQueueName());
                    TransactionElement<?> transactionElement = new
TransactionElement(transactionData);

      producer.sendTo(transactionData.getRequestQueueName(),
transactionElement);
                    log.info("Se ha enviado la transacción {} a su cola
de envio {}", transactionData.getTransactionDataId(),
                                transactionData.getRequestQueueName());
                    transactionData.setSentDate(new Date());

      transactionData.setStatus(TransactionStatus.SENT.toString());
                    log.info("Se guarda la transacción {} como
enviada", transactionData.getTransactionDataId());
                    transactionDataRepository.save(transactionData);
                    if (transactionData.getMaxResponseTime()>0){ // Se
ha establecido un tiempo máximo para recibir la respuesta
                        log.info("Se ha establecido un tiempo maximo
de respuesta de {} msg", transactionData.getMaxResponseTime());
                        ResponseCheckingTask checkerTask = new
ResponseCheckingTask(this, transactionData);
                        log.info("Lanzamos el hilo de ejecución para
comprobar si se ha recibido la respuesta para la transaccion {} en
un tiempo maximo de {} msg",

      transactionData.getTransactionDataId(),
transactionData.getMaxResponseTime());
                        taskScheduler.execute(checkerTask,
transactionData.getMaxResponseTime());
                    }
                    log.info("Recuperamos el consumidor de la respuesta
para la transaccion {}", transactionData.getTransactionDataId());
                    SenderConsumer<?> senderConsumer =
senderConsumerRepository.get(transactionData.getTransactionDataId())
;
                    String responseQueueName =
transactionData.getResponseQueueName();
                    log.info("Creamos el listener (y la cola si fuese
necesario) de la cola {} donde la transaccion {} espera recibir la
respuesta", responseQueueName,
transactionData.getTransactionDataId());
                    ampqUtil.createRabbitListener(responseQueueName,
senderConsumer);
                } catch (Exception ex){
                    log.error("Ha habido un error en el envio de la
transacción {} a su cola de envio {}:",
transactionData.getTransactionDataId(),
                                transactionData.getRequestQueueName(),
ex);
                    int sentTries = transactionData.getSentTries();
                    transactionData.setSentDate(null);
                    transactionData.setSentTries(sentTries+1);
                    log.info("Se guarda la transacción {} como NO
enviada con un numero de intentos alcanzado de {}",
transactionData.getTransactionDataId(),
                                sentTries);
                    transactionDataRepository.save(transactionData);
                    throw new SendingException();
                }
```

```
	}

	/**
	 * Metodo invocado para recibir una transaccion por parte del receptor
	 *
	 * @param requestQueueName Nombre de la cola donde el receptor espera recibir la transaccion
	 * @param receiverConsumer Componente encargado de recibir y procesar la transaccion
	 */
	public void receiveTransaction(String requestQueueName, ReceiverConsumer<?> receiverConsumer){
		try {
			log.info("Se procede a crear el listener correspondiente para la cola {} donde se espera recibir una transaccion", requestQueueName);
			beanFactory.autowireBean(receiverConsumer);
			ampqUtil.createRabbitListener(requestQueueName, receiverConsumer);
		} catch (Exception ex){
			log.error("Error al crear el listener para la cola {} donde se espera recibir la respuesta de una transaccion", requestQueueName);
		}
	}

	/**
	 * Metodo invocado por el receptor para devolver al emisor la transaccion una vez que la ha recibido y procesado
	 *
	 * @param transaction Elemento transmitido entre emisor y receptor y que representa la transaccion
	 */
	public void sendResponse(TransactionElement<?> transaction){
		producer.sendTo(transaction.getResponseQueueName(), transaction);
	}

}
```

Clase Java 18 - DistributedTransactionService. Servicio con la lógica de negocio de ADTM

Repositorio JPA: TransactionDataRepository

```
package es.uned.master.software.tfm.adtm.jpa.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import es.uned.master.software.tfm.adtm.jpa.entity.TransactionData;

/**
 * Repositorio de transacciones
 *
 * @author Francisco Cilleruelo
 */
@Repository
public interface TransactionDataRepository extends JpaRepository<TransactionData, Long> {
```

```
	/**
	 * @return Lista de transacciones que todavia no se han enviado
	 */
	@Query("SELECT TD FROM TransactionData TD WHERE sentDate IS NULL and
responseCheckedDate IS NULL")
	public List<TransactionData> getTransactionToBeSent();

}
```

Clase Java 19 - TransactionDataRepository. Repositorio JPA de ADTM

Objetos de negocio: TransactionData, Transaction, TransactionElement

```
package es.uned.master.software.tfm.adtm.jpa.entity;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;

import es.uned.master.software.tfm.adtm.entity.Transaction;

/**
 * Objeto perisistido en BD que representa la transaccion
 *
 * @author Francisco Cilleruelo
 */
@Entity
@Table(name = "TRANSACTIONS_DATA")
public class TransactionData implements Serializable{

	private static final long serialVersionUID = 8238622515676354812L;

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long transactionDataId;
	/**
	 * Objeto de negocio transmitido en la transaccion
	 */
	@Lob
	private Serializable objectTransmited;
	/**
	 * Fecha de comienzo de la transaccion
	 * Cuando el emisor la comienza, no cuando se envia a la cola del
receptor de la transaccion
	 */
	private Date startDate;
	/**
	 * Fecha de envio a la cola del receptor de la transaccion
	 */
	private Date sentDate;
	/**
	 * Fecha en la que el emisor se recibe la respuesta por parte del
receptor
	 */
	private Date receivedDate;
	/**
```

```
	 * Fecha en la que se comprueba si se ha recibido una respuesta por parte del receptor
	 * de acuerdo con el valor establecido en la propiedad maxResponseTime
	 */
	private Date responseCheckedDate;
	/**
	 * Estado de la transacción (TO_BE_SENT, SENT, RECEIVED_OK, RECEIVED_NOK, NOT_RECEIVED)
	 */
	private String status;
	/**
	 * Informacion adicional
	 */
	private String additionalInfo;
	/**
	 * Nombre de la cola donde el emisor envia la transaccion
	 */
	private String requestQueueName;
	/**
	 * Nombre de la cola donde se espera recibir la respuesta del receptor de la transaccion
	 */
	private String responseQueueName;
	/**
	 * Numero de intentos de envio
	 */
	private int sentTries;
	/**
	 * Tiempo maximo de respuesta permitido
	 * Superado este tiempo si no se ha recibido respuesta, la transaccion se dara por invalida
	 * Un valor menor o igual que cero indicara que no hay limite para recibir la respuesta
	 */
	private int maxResponseTime;

	public TransactionData() {
		super();
	}

	public TransactionData(Transaction transaction) {
		super();
		this.objectTransmited = transaction.getObjectTransmited();
		this.requestQueueName = transaction.getRequestQueueName();
		this.responseQueueName = transaction.getResponseQueueName();
		this.maxResponseTime = transaction.getMaxResponseTime();
	}

	public Long getTransactionDataId() {
		return transactionDataId;
	}

	public void setTransactionDataId(Long transactionDataId) {
		this.transactionDataId = transactionDataId;
	}

	public Serializable getObjectTransmited() {
		return objectTransmited;
	}

	public void setObjectTransmited(Serializable objectTransmited) {
```

```
            this.objectTransmited = objectTransmited;
      }

      public String getAdditionalInfo() {
            return additionalInfo;
      }

      public void setAdditionalInfo(String additionalInfo) {
            this.additionalInfo = additionalInfo;
      }

      public Date getStartDate() {
            return startDate;
      }

      public void setStartDate(Date startDate) {
            this.startDate = startDate;
      }

      public Date getSentDate() {
            return sentDate;
      }

      public void setSentDate(Date sentDate) {
            this.sentDate = sentDate;
      }

      public Date getReceivedDate() {
            return receivedDate;
      }

      public void setReceivedDate(Date receivedDate) {
            this.receivedDate = receivedDate;
      }

      public Date getResponseCheckedDate() {
            return responseCheckedDate;
      }

      public void setResponseCheckedDate(Date responseCheckedDate) {
            this.responseCheckedDate = responseCheckedDate;
      }

      public String getStatus() {
            return status;
      }

      public void setStatus(String status) {
            this.status = status;
      }

      public String getRequestQueueName() {
            return requestQueueName;
      }

      public void setRequestQueueName(String requestQueueName) {
            this.requestQueueName = requestQueueName;
      }

      public String getResponseQueueName() {
            return responseQueueName;
```

```
	}

	public void setResponseQueueName(String responseQueueName) {
		this.responseQueueName = responseQueueName;
	}

	public int getSentTries() {
		return sentTries;
	}

	public void setSentTries(int sentTries) {
		this.sentTries = sentTries;
	}

	public int getMaxResponseTime() {
		return maxResponseTime;
	}

	public void setMaxResponseTime(int maxResponseTime) {
		this.maxResponseTime = maxResponseTime;
	}

}
```

Clase Java 20 - TransactionData. Objeto de negocio (persistido) de ADTM

```
package es.uned.master.software.tfm.adtm.entity;

import java.io.Serializable;

import es.uned.master.software.tfm.adtm.amqp.sender.SenderConsumer;
import es.uned.master.software.tfm.adtm.jpa.entity.TransactionData;

/**
 * Objeto que representa la transaccion para el emisor que desencadena la comunicacion
 *
 * @author Francisco Cilleruelo
 *
 * @param <T> Tipo de objeto de negocio (serializable) enviado como parte de la transaccion
 */
public class Transaction<T extends Serializable> implements Serializable {

	private static final long serialVersionUID = 6768349709404274424L;

	/**
	 * Objeto enviado como parte de la transaccion
	 */
	private T objectTransmited;
	/**
	 * Componente encarga de recibir y procesar la respuesta recibida del receptor de la transaccion
	 */
	private SenderConsumer<T> senderConsumer;
	/**
	 * Nombre de la cola donde se enviara la transaccion
	 */
	private String requestQueueName;
	/**
	 * Nombre de la cola donde se espera recibir la respuesta del receptor de la transaccion
```

```
	 */
	private String responseQueueName;
	/**
	 * Tiempo maximo de respuesta permitido
	 * Superado este tiempo si no se ha recibido respuesta, la transaccion se
dara por invalida
	 * Un valor menor o igual que cero indicara que no hay limite para
recibir la respuesta
	 */
	private int maxResponseTime;

	private Transaction() {
		super();
	}

	public Transaction(TransactionData transactionData) {
		super();
		this.requestQueueName = transactionData.getRequestQueueName();
		this.responseQueueName = transactionData.getResponseQueueName();
		this.objectTransmited = (T)transactionData.getObjectTransmited();
	}

	public Transaction(T objectTransmited, SenderConsumer<T> senderConsumer,
String requestQueueName, String responseQueueName,
			int maxResponseTime) {
		super();
		this.objectTransmited = objectTransmited;
		this.senderConsumer = senderConsumer;
		this.requestQueueName = requestQueueName;
		this.responseQueueName = responseQueueName;
		this.maxResponseTime = maxResponseTime;
	}

	public T getObjectTransmited() {
		return objectTransmited;
	}

	public void setObjectTransmited(T objectTransmited) {
		this.objectTransmited = objectTransmited;
	}

	public SenderConsumer<T> getSenderConsumer() {
		return senderConsumer;
	}

	public void setSenderConsumer(SenderConsumer<T> senderConsumer) {
		this.senderConsumer = senderConsumer;
	}

	public String getRequestQueueName() {
		return requestQueueName;
	}

	public void setRequestQueueName(String requestQueueName) {
		this.requestQueueName = requestQueueName;
	}

	public String getResponseQueueName() {
		return responseQueueName;
	}
```

```
	public void setResponseQueueName(String responseQueueName) {
		this.responseQueueName = responseQueueName;
	}

	public int getMaxResponseTime() {
		return maxResponseTime;
	}

	public void setMaxResponseTime(int maxResponseTime) {
		this.maxResponseTime = maxResponseTime;
	}

}
```

Clase Java 21 - Transaction. Objeto de negocio de ADTM

```
package es.uned.master.software.tfm.adtm.entity;

import java.io.Serializable;

import es.uned.master.software.tfm.adtm.jpa.entity.TransactionData;

/**
 * Objeto compartido entre emisor y receptor como elemento transmitido en la transaccion
 *
 * @author Francisco Cilleruelo
 *
 * @param <T> Tipo de objeto de negocio (serializable) enviado como parte de la transaccion
 */
public class TransactionElement<T extends Serializable> implements Serializable {

	private static final long serialVersionUID = 2432265495175652943L;

	/**
	 * Identificador unico de la transaccion
	 */
	private Long transactionReference;
	/**
	 * Informacion adicional por parte del emisor o del receptor hacia la otra parte
	 */
	private String additionalInfo;
	/**
	 * Objeto de negocio transmitido
	 */
	private T objectTransmited;
	/**
	 * Estado de la transaccion (TO_BE_SENT, SENT, RECEIVED_OK, RECEIVED_NOK, NOT_RECEIVED)
	 */
	private TransactionStatus status;
	/**
	 * Nombre de la cola donde el emisor espera recibir la respuesta del receptor
	 */
	private String responseQueueName;

	public TransactionElement() {
		super();
```

```
	}

	public TransactionElement(TransactionData transactionData) {
		super();
		this.transactionReference = transactionData.getTransactionDataId();
		this.additionalInfo = transactionData.getAdditionalInfo();
		this.objectTransmited = (T)transactionData.getObjectTransmited();
		this.status =
TransactionStatus.valueOf(transactionData.getStatus());
		this.responseQueueName = transactionData.getResponseQueueName();
	}

	public TransactionElement(Long transactionReference, String
additionalInfo, T objectTransmited,
			TransactionStatus status) {
		super();
		this.transactionReference = transactionReference;
		this.additionalInfo = additionalInfo;
		this.objectTransmited = objectTransmited;
		this.setStatus(status);
	}

	public Long getTransactionReference() {
		return transactionReference;
	}

	public void setTransactionReference(Long transactionReference) {
		this.transactionReference = transactionReference;
	}

	public String getAdditionalInfo() {
		return additionalInfo;
	}

	public void setAdditionalInfo(String additionalInfo) {
		this.additionalInfo = additionalInfo;
	}

	public T getObjectTransmited() {
		return objectTransmited;
	}

	public void setObjectTransmited(T objectTransmited) {
		this.objectTransmited = objectTransmited;
	}

	public TransactionStatus getStatus() {
		return status;
	}

	public void setStatus(TransactionStatus status) {
		this.status = status;
	}

	public String getResponseQueueName() {
		return responseQueueName;
	}

	public void setResponseQueueName(String responseQueueName) {
		this.responseQueueName = responseQueueName;
	}
```

```
}
```

```
package es.uned.master.software.tfm.adtm.entity;

/**
 * Estado de la transaccion
 *
 * @author Francisco Cilleruelo
 */
public enum TransactionStatus {

	TO_BE_SENT, SENT, RECEIVED_OK, RECEIVED_NOK, NOT_RECEIVED

}
```

Clase Java 22 - TransactionElement (y TranactionStatus). Objeto de negocio de ADTM

Productor AMQP: Producer

```
package es.uned.master.software.tfm.adtm.amqp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import es.uned.master.software.tfm.adtm.entity.TransactionElement;

/**
 * Componente para enviar a una cola determinada una transaccion
 *
 * @author Francisco Cilleruelo
 */
@Component
public class Producer {

	private static final Logger log =
LoggerFactory.getLogger(Producer.class);

	@Autowired
	private RabbitTemplate rabbitTemplate;

	/**
	 * Envia la transaccion a la cola indicada
	 *
	 * @param routingKey La cola destino a la que sera enviada la transaccion
	 * @param transaction Transaccion a enviar
	 */
	public void sendTo(String routingKey, TransactionElement<?> transaction){
		try {
			ObjectMapper mapper = new ObjectMapper();
			String message = mapper.writeValueAsString(transaction);
			this.rabbitTemplate.convertAndSend(routingKey, message);
			log.info("Mensaje {} enviado satisfactoriamente a la cola
{}", message, routingKey);
		} catch (JsonProcessingException jsonEx){
			log.error("Error al intentar convertir un objeto a JSON para
ser transmitido a la cola {}", routingKey);
		} catch (Exception ex){
```

```
                  log.error("Error en el envio del mensaje a la cola {}",
routingKey);
            }
      }


}
```

Clase Java 23 - Producer. Productor AMQP de ADTM

Consumidores AMQP: ReceiverConsumer y SenderConsumer

```
package es.uned.master.software.tfm.adtm.amqp.receiver;

import java.io.Serializable;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;

import com.fasterxml.jackson.databind.JavaType;
import com.fasterxml.jackson.databind.ObjectMapper;

import es.uned.master.software.tfm.adtm.entity.TransactionElement;
import es.uned.master.software.tfm.adtm.entity.TransactionStatus;
import es.uned.master.software.tfm.adtm.service.DistributedTransactionService;

/**
 * Componente para recibir transacciones por parte del receptor de estas
 *
 * @author Francisco Cilleruelo
 *
 * @param <T> Tipo de objeto de negocio (serializable) que espera recibir el
receptor de la transaccion
 */
public abstract class ReceiverConsumer<T extends Serializable> implements
Serializable{

      private static final long serialVersionUID = 7278191552064450096L;

      private static final Logger log =
LoggerFactory.getLogger(ReceiverConsumer.class);

      private Class<T> classType;

      @Autowired
      private DistributedTransactionService distributedTransactionService;

      public ReceiverConsumer(Class<T> classType){
            this.classType = classType;
      }

      /**
       * Metodo encargado de recibir la transaccion como un String JSON
       *
       * @param message Mensaje recibido
       */
      public void handleMessage(String message){
            log.info("Recibido mensaje {}", message);
            try {
                  ObjectMapper mapper = new ObjectMapper();
```

```
                    JavaType javaType =
mapper.getTypeFactory().constructParametricType(TransactionElement.class,
classType);
                    TransactionElement<T> transaction = mapper.readValue(message,
javaType);
                    try {
                          log.info("Se procesa la transaccion recibida");
                          boolean resultado =
processRequest((T)transaction.getObjectTransmited());
                          if (resultado){
                                log.info("La respuesta se ha procesado
correctamente");

      transaction.setStatus(TransactionStatus.RECEIVED_OK);
                          } else {
                                log.info("La respuesta NO se ha procesado
correctamente");

      transaction.setStatus(TransactionStatus.RECEIVED_NOK);
                          }
                    } catch (Exception ex){
                          log.info("La respuesta NO se ha procesado
correctamente");
                          transaction.setStatus(TransactionStatus.RECEIVED_NOK);
                          transaction.setAdditionalInfo("La respuesta NO se ha
procesado correctamente" + ex.getMessage());
                    }
                    log.info("Procedemos a comunicar al emisor el resultado del
procesamiento de la transaccion a traves de la cola {}",
transaction.getResponseQueueName());
                    distributedTransactionService.sendResponse(transaction);
              } catch (Exception ex){
                    log.error("Error en la conversión del mensaje JSON {}",
message);
              }
      }

      /**
       * Metodo a implementar por el receptor de la transaccion para definir el
procesamiento
       * a realizar con la transaccion. Para indicar posteriormente si se puede
dar la transaccion
       * como correcta o no
       *
       * @param requestObject Objeto de negocio a procesar
       * @return Resultado del procesamiento: Correcto (true) o Incorrecto
(false)
       */
      public abstract boolean processRequest(T requestObject);
}
```

Clase Java 24 - ReceiverConsumer. Consumidor AMQP para el receptor de ADTM

```
package es.uned.master.software.tfm.adtm.amqp.sender;

import java.io.Serializable;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;

import com.fasterxml.jackson.databind.JavaType;
```

```java
import com.fasterxml.jackson.databind.ObjectMapper;

import es.uned.master.software.tfm.adtm.entity.TransactionElement;
import es.uned.master.software.tfm.adtm.entity.TransactionStatus;
import es.uned.master.software.tfm.adtm.service.DistributedTransactionService;

/**
 * Componente para recibir por parte del remitente la respuesta del receptor
 *
 * @author Francisco Cilleruelo
 *
 * @param <T> Tipo de objeto de negocio (serializable) que espera recibir el
emisor de la transaccion
 */
public abstract class SenderConsumer<T extends Serializable> implements
Serializable {

	private static final long serialVersionUID = 8480348200882129004L;

	private static final Logger log =
LoggerFactory.getLogger(SenderConsumer.class);

	private Class<T> classType;

	@Autowired
	private DistributedTransactionService distributedTransactionService;

	public SenderConsumer(Class<T> classType) {
		super();
		this.classType = classType;
	}

	/**
	 * Metodo encargado de recibir la transaccion como un String JSON
	 *
	 * @param message Mensaje recibido
	 */
	public void handleMessage(String message){
		log.info("Recibido mensaje {}", message);
		ObjectMapper mapper = new ObjectMapper();
		try {
			JavaType javaType =
mapper.getTypeFactory().constructParametricType(TransactionElement.class,
classType);
			TransactionElement<T> transaction = mapper.readValue(message,
javaType);
			if
(transaction.getStatus().equals(TransactionStatus.RECEIVED_OK)){
				log.info("La transaccion ha terminado correctamente,
hacemos el commit");
				commit((T)transaction.getObjectTransmited());
			} else { // Cualquier otro estado, preferiblemente
RECEIVED_NOK
				log.info("La transaccion no ha terminado correctamente,
hacemos el rollback");
				rollback((T)transaction.getObjectTransmited());
			}

	distributedTransactionService.transactionResponseReceived(transaction);
		} catch (Exception ex){
```

```
                log.error("Error en la recepción de la respuesta del
destinatario por parte del emisor: ", ex);
            }
        }

        /**
         * Metodo implementado por el emisor para definir el proceso a realizar
         * en el caso de que la transaccion haya terminado correctamente
(TransactionStatus.RECEIVED_OK)
         *
         * @param requestObject Objeto de negocio a procesar
         */
        public abstract void commit(T requestObject);

        /**
         * Metodo implementado por el emisor para definir el proceso a realizar
         * en el caso de que la transaccion NO haya terminado correctamente
         * (TransactionStatus.RECEIVED_NOK o TransactionStatus.NOT_RECEIVED)
         *
         * @param requestObject Objeto de negocio a procesar
         */
        public abstract void rollback(T requestObject);
}
```

Clase Java 25 - SenderConsumer. Consumidor AMQP para el emisor de ADTM

Clase de utilidades AMQP: AmpqUtil

```
package es.uned.master.software.tfm.adtm.amqp.util;

import java.util.ArrayList;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitAdmin;
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 * Componente de utilidades relativas al gestor de mensajes AMPQ
 *
 * @author Francisco Cilleruelo
 */
@Component
public class AmpqUtil {

        private static final Logger log =
LoggerFactory.getLogger(AmpqUtil.class);

        @Autowired
        private ConnectionFactory connectionFactory;

        @Autowired
        private RabbitAdmin rabbitAdmin;

        private List<String> queuesCreated = new ArrayList<>();

        /**
```

```
	 * Metodo para crear el listener para los mensajes recibidos por la cola indicada
	 *
	 * @param queueName Nombre de la cola
	 * @param consumer Componente encargado de recibir y procesar los mensajes recibidos en esa cola
	 */
	public void createRabbitListener(String queueName, Object consumer){
		log.info("Comprobacion de que no se haya creado ya la cola con el nombre {}", queueName);
		if (!queuesCreated.contains(queueName)){
			log.info("La cola con el nombre {} no existia, procedemos a crearla", queueName);
			Queue queue = new Queue(queueName, false, false, false);
			rabbitAdmin.declareQueue(queue);
		}
		log.info("Procedemos a crear el listener para la cola {}", queueName);
		MessageListenerAdapter adapter = new MessageListenerAdapter(consumer);
		SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
		container.setMessageListener(adapter);
		container.setQueueNames(queueName);
		log.info("Arrancamos el listener para la cola {}", queueName);
		container.start();
	}

}
```

Clase Java 26 - AmpqUtil. Clase de utilidades AMQP de ADTM

Clase de propiedades para la parametrización: AdtmProperties

```
package es.uned.master.software.tfm.adtm.configuration;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

/**
 * Componente de propiedades para el gestor de transacciones ADTM
 *
 * @author Francisco Cilleruelo
 */
@Component
@ConfigurationProperties(prefix="adtm")
public class AdtmProperties {

	/**
	 * Frecuencia de escaneo de transacciones no enviadas para ser enviadas a su cola correspondiente
	 */
	private long pollingfrecuency;

	public long getPollingfrecuency() {
		return pollingfrecuency;
	}

	public void setPollingfrecuency(long pollingfrecuency) {
		this.pollingfrecuency = pollingfrecuency;
	}

}
```

Clase Java 27 - AdtmProperties. Clase de propiedades para la parametrización de ADTM

Clase de inicialización: ADTM

```
package es;


import java.util.HashMap;
import java.util.Map;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitAdmin;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;

import es.uned.master.software.tfm.adtm.amqp.sender.SenderConsumer;
import es.uned.master.software.tfm.adtm.manager.DistributedTransactionManager;

/**
 * Libreria de gestion de transacciones distribuidas asincronas
 * Asynchronous Distributed Transactions Manager (ADTM)
 *
 * @author Francisco Cilleruelo
 */
@Configuration
@EnableAsync
@EnableScheduling
@ComponentScan
@EnableJpaRepositories
@EntityScan
public class ADTM {

	@Bean
	public ThreadPoolTaskScheduler taskScheduler(){
		ThreadPoolTaskScheduler taskScheduler = new
ThreadPoolTaskScheduler();
		taskScheduler.setThreadNamePrefix("ADTM_ScheduledTasks-");
		taskScheduler.setPoolSize(60);
		return taskScheduler;
	}

	@Autowired
	private ConnectionFactory connectionFactory;

	@Bean
	public RabbitAdmin getRabbitAdmin(){
		return new RabbitAdmin(connectionFactory);
	}

	@Bean
	public DistributedTransactionManager distributedTransactionManager(){
		return new DistributedTransactionManager();
	}

	/**
```

```
         * @return Repositorio de listeners para los mensajes de respuesta
recibidos de los receptores de las transacciones
         */
        @Bean
        public Map<Long, SenderConsumer> buildSenderConsumerRepository(){
              return new HashMap<>();
        }

}
```

Clase Java 28 - ADTM. Clase principal de inicialización de ADTM

Anexo 3 – Código fuente de la solución integrada en el contexto del problema

Servicio emisor (Microservicio de pedidos)

Servicio con la lógica de negocio: OrderService

```
package es.uned.master.software.tfm.adtm.microservice.order.service;

import java.io.Serializable;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import es.uned.master.software.tfm.adtm.entity.Transaction;
import es.uned.master.software.tfm.adtm.manager.DistributedTransactionManager;
import
es.uned.master.software.tfm.adtm.microservice.order.consumer.OrderConsumer;
import es.uned.master.software.tfm.adtm.microservice.order.jpa.entity.Order;
import es.uned.master.software.tfm.adtm.microservice.order.jpa.repository
.OrderRepository;

@Service
@Transactional
public class OrderService implements Serializable{

	private static final long serialVersionUID = -8757659633801168330L;

	private static final Logger log =
LoggerFactory.getLogger(OrderService.class);

	@Autowired
	private OrderRepository orderRepository;

	@Autowired
	private DistributedTransactionManager transactionManager;

	@Autowired
	private OrderConsumer orderConsumer;

	@Value("${queue.orders.name}")
	private String ordersQueueName;

	@Value("${queue.customers.name}")
	private String customersQueueName;

	public List<Order> findAll(){
		log.info("Busqueda de todos los pedidos");
		return orderRepository.findAll();
	}

	public Order createOrder(Order order){
		log.info("Se inicializa el estado del pedido a crear como nuevo
(NEW)");
		order.setStatus("NEW");
		log.info("Se guarda el nuevo pedido");
```

```
            orderRepository.save(order);
            log.info("Se envia el pedido a la cola {} para ser procesado por
el servicio de clientes", ordersQueueName);
            Transaction<Order> transaction = new Transaction<Order>(order,
orderConsumer, ordersQueueName, customersQueueName, 0);
            transactionManager.sendTransaction(transaction);
            return order;
      }

      public void update(Order order){
            log.info("Se actualiza el pedido");
            orderRepository.save(order);
      }

}
```

Clase Java 29 - OrderService. Servicio con la lógica de negocio del microservicio de pedidos con ADTM integrado

Consumidor AMQP: OrderConsumer

```
package es.uned.master.software.tfm.adtm.microservice.order.consumer;

import java.io.Serializable;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import es.uned.master.software.tfm.adtm.amqp.sender.SenderConsumer;
import es.uned.master.software.tfm.adtm.microservice.order.jpa.entity.Order;
import
es.uned.master.software.tfm.adtm.microservice.order.service.OrderService;

@Component
public class OrderConsumer extends SenderConsumer<Order> implements
Serializable{

      private static final long serialVersionUID = -6922109684903455774L;

      private static final Logger log =
LoggerFactory.getLogger(OrderConsumer.class);

      @Autowired
      private OrderService orderService;

      public OrderConsumer() {
            super(Order.class);
      }

      @Override
      public void commit(Order order) {
            log.info("Se actualiza el pedido a OPEN");
            order.setStatus("OPEN");
            orderService.update(order);
      }

      @Override
      public void rollback(Order order) {
            log.info("Se actualiza el pedido a OPEN");
            order.setStatus("REJECTED");
```

```
            orderService.update(order);
      }

}
```

Clase Java 30 - OrderConsumer. Consumidor AMQP del microservicio de pedidos con ADTM integrado

Clase de arranque e inicialización: OrderServiceAdtmApplication

```
package es.uned.master.software.tfm.adtm.microservice.order;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;

import es.ADTM;

@SpringBootApplication
@Import(ADTM.class)
public class OrderServiceAdtmApplication {

      public static void main(String[] args) {
            SpringApplication.run(OrderServiceAdtmApplication.class, args);
      }

}
```

Clase Java 31 - OrderServiceAdtmApplication. Clase de arranque e inicialización del microservicio de pedidos con ADTM integrado

Servicio receptor (Microservicio de clientes)

Servicio con la lógica de negocio: CustomerService

```
package
es.uned.master.software.tfm.adtm.microservice.customer.service;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;

import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.entity.Cu
stomer;
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.entity.Or
der;
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.entity.Re
servedCredit;
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.entity.Re
servedCreditId;
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.repositor
y.CustomerRepository;
```

```
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.repositor
y.ReservedCreditRepository;


@Service
@Transactional
public class CustomerService {

      private static final Logger log =
LoggerFactory.getLogger(CustomerService.class);

      @Autowired
      private CustomerRepository customerRepository;

      @Autowired
      private ReservedCreditRepository reservedCreditRepository;

      public void insertExampleData(){
            customerRepository.save(new Customer(300));
            customerRepository.save(new Customer(8000));
            log.info("Inicializado repositorio de clientes con datos
de ejemplo");
      }

      public List<Customer> findAll(){
            log.info("Busqueda de todos los clientes");
            return customerRepository.findAll();
      }

      public boolean checkCredit(Order order){
            log.info("Recuperamos el cliente {} asociado al pedido",
order.getCustomerId());
            Customer customer =
customerRepository.findOne(order.getCustomerId());
            int reservedCreditNow = 0;
            String reservedCreditNowS =
reservedCreditRepository.sumReserverCreditByCustomerId(order.getCust
omerId());
            if (StringUtils.hasText(reservedCreditNowS)){
                  reservedCreditNow =
Integer.valueOf(reservedCreditNowS);
            }
            log.info("El credito reservado del cliente {} para otros
pedidos es de {}", order.getCustomerId(), reservedCreditNow);
            if (customer != null && customer.getCreditLimit() >=
order.getTotal() + reservedCreditNow){
                  log.info("El limite de credito para el cliente {}
es superior a la suma de la cantidad solicitada para el pedido ({})
mas el credito reservado para otros pedidos ({})"
                              , order.getCustomerId(),
order.getOrderId(), reservedCreditNow);
                  ReservedCreditId reservedCreditId = new
ReservedCreditId(order.getOrderId(), order.getCustomerId());
                  ReservedCredit reservedCredit = new
ReservedCredit(reservedCreditId, order.getTotal());
                  log.info("Se reserva el credito {} para el pedido
{} del cliente {}", reservedCredit.getTotalReserved(),

      reservedCredit.getReservedCreditId().getOrderId(),
reservedCredit.getReservedCreditId().getCustomerId());
```

```
                    reservedCreditRepository.save(reservedCredit);
                    return true;
              } else { // No existe el cliente o la cantidad del pedido
supera el credito
                    log.info("El limite de credito es inferior a la
cantidad solicitada por el pedido");
                    return false;
              }
        }

}
```

Clase Java 32 - CustomerService. Servicio con la lógica de negocio del microservicio de clientes con ADTM integrado

Consumidor AMQP: CustomerConsumer

```
package es.uned.master.software.tfm.adtm.microservice.customer.consumer;

import java.io.Serializable;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import es.uned.master.software.tfm.adtm.amqp.receiver.ReceiverConsumer;
import
es.uned.master.software.tfm.adtm.microservice.customer.jpa.entity.Order;
import
es.uned.master.software.tfm.adtm.microservice.customer.service.CustomerService
;

@Component
public class CustomerConsumer extends ReceiverConsumer<Order> implements
Serializable{

      private static final long serialVersionUID = 1729892484009760697L;

      private static final Logger log =
LoggerFactory.getLogger(CustomerConsumer.class);

      @Autowired
      private CustomerService customerService;

      public CustomerConsumer() {
            super(Order.class);
      }

      @Override
      public boolean processRequest(Order order) {
            log.info("Comprobamos la disponibilidad de credito para el cliente
{}", order.getCustomerId());
            return customerService.checkCredit(order);
      }

}
```

Clase Java 33 - CustomerConsumer. Consumidor AMQP del microservicio de clientes con ADTM integrado

Clase de arranque e inicialización: CustomerServiceAdtmApplication

```
package es.uned.master.software.tfm.adtm.microservice.customer;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;

import es.ADTM;
import es.uned.master.software.tfm.adtm.manager.DistributedTransactionManager;
import
es.uned.master.software.tfm.adtm.microservice.customer.consumer.CustomerConsum
er;
import
es.uned.master.software.tfm.adtm.microservice.customer.service.CustomerService
;

@SpringBootApplication
@Import(ADTM.class)
public class CustomerServiceAdtmApplication {

	private static final Logger log =
LoggerFactory.getLogger(CustomerServiceAdtmApplication.class);

	public static void main(String[] args) {
		SpringApplication.run(CustomerServiceAdtmApplication.class, args);
	}

	@Bean
	CommandLineRunner init(CustomerService customerService){
		return args -> {
			customerService.insertExampleData();
			customerService.findAll().forEach(entry ->
log.info(entry.toString()));
		};
	}

	@Value("${queue.orders.name}")
	private String ordersQueueName;

	@Autowired
	private CustomerConsumer customerConsumer;

	@Bean
	public DistributedTransactionManager
buildTransactionManager(DistributedTransactionManager transactionManager){
		transactionManager.receiveTransaction(ordersQueueName,
customerConsumer);
		return transactionManager;
	}
}
```

Clase Java 34 - CustomerServiceAdtmApplication. Clase de arranque e inicialización del microservicio de clientes con ADTM integrado