UNIVERSIDAD
DE GRANADA

Caso práctico:
Servicios y microservicios en banca

INGENIERÍA DE SISTEMAS DE INFORMACIÓN
Grado en Ingeniería Informática
CURSO 2019-20

Alonso Bueno Herrero
alonsobueno13@correo.ugr.es

Escuela Técnica Superior de
Ingenierías Informática y de Telecomunicación

Índice

Índice de figuras

Resumen

En este documento se expone una descripción sobre el enfoque orientado a servicios y microservicios de los modernos sistemas de banca para el «Caso práctico» de la asignatura *Ingeniería de Sistemas de Información*, del Grado en Ingeniería Informática de la Universidad de Granada.

Alonso Bueno Herrero.

En Granada, a 4 de agosto de 2020.

1. Motivación

Gran parte de los grandes bancos del mundo están optando por un cambio en la arquitectura de sus sistemas informáticos. Todos ellos son sistemas que se han ido desarrollando, básicamente, a través de arquitecturas monolíticas. El modelo monolítico parte de los años 60-70. En este tiempo, a diferencia del actual, el volumen de datos con los que los sistemas tenían que trabajar era bastante menor que con el que tienen que enfrentarse los sistemas actuales.

Dichos sistemas han de diseñarse precisamente con esa intención (en general), sobre todo si estamos hablando de grandes entidades financieras donde, como veremos con algunos datos estadísticos reales, la cantidad de transacciones que pueden llegar a registrarse alcanzan los cientos de millones al día.

Pero también los sistemas tienen que enfrentarse al nuevo "paradigma": la web. Una web donde cada día supone casi un paso de gigante en eficiencia, en escalabilidad, en accesibilidad... Y quedarse atrás en estos aspectos (y muchos otros) no pueden permitírselo las grandes entidades bancarias.

Precisamente, estas entidades han llegado al nuevo paradigma "de la web" con la pesada carga de una arquitectura que aporta, en general, las siguientes limitaciones (intento resumirlas en base a las experiencias reseñadas en la bibliografía):

- Complejidad en la mantenibilidad de las funcionalidades, debido, esencialmente, al alto acoplamiento entre módulos (o subsistemas). Dicho de otra forma, el equipo y la gestión de proyectos son tareas demasiado amplias y complejas.
- Dificultad en la corrección de defectos o fallos software.
- Incremento excesivo de los módulos de funcionalidades, con lo cual el alcanzar la escalabilidad y el rendimiento adecuados son tareas demasiado caras.
- Complejidad de integración de tecnologías diversas (o lenguajes) dentro del sistema
- Asignación (y posible reemplazo) de recursos hardware (equipos físicos de unidades de almacenamiento y procesamiento) en los módulos de funcionalidades.
- A más bajo nivel, no se pueden crear pipelines de entrega (de información) continuamente.

Las *soluciones* que se proponen van en la línea de ese paradigma que comentábamos antes: la web. Se opta por arquitecturas orientadas a servicios (SOA, la más "genérica"), y sus posibles derivados, de entre los que destacan los microservicios. Parece ser que los diseñadores de estos sistemas han visto en los ***patrones de diseño orientados a servicios*** una suerte de "grial" para dar solución, sino a todos, a gran parte de las trabas que ponía la arquitectura monolítica.

Pero también cabe destacar que no todo en las arquitecturas monolíticas son desventajas. Su propuesta se asocia al momento en que se planteó, y los puntos fuertes que en ese momento se apreciaron, y que hoy por hoy todavía podríamos identificar son: su alto grado de estabilidad y seguridad, debido esencialmente a su solidez, debido a la simplicidad de su estructura, suelen ser menos costosos de desarrollar en general. Sin embargo, aporta grandes inconvenientes, como veremos más adelante, en términos de escalabilidad, mantenibilidad, actualización, detección de errores, etc. En la referencia [5], se realiza un estudio comparativo entre las prestaciones de estas dos arquitecturas, estresándolas con diversos modelos de carga.

2. Requisitos del sistema

La idiosincrasia de los sistemas de banca hacen que tengan unos requisitos muy particulares en ciertos aspectos. Se enumeran y describen a continuación :

1. ***Eficiencia***: debe soportar una carga de cálculo muy intensa, y deberá resolverlo con una elección de procedimientos y recursos (software y hardware) adecuados. Por ejemplo, con el cálculo del nivel de riesgo agregado (un cálculo económico que mide el nivel de riesgo que afronta la entidad).

2. ***Escalabilidad***: en este tipo de sistemas, donde se trabaja con operaciones complejas, con cantidades de datos muy grandes, y con situaciones donde muchos usuarios pueden estar realizando transacciones al mismo tiempo, disponer de un mecanismo que permita escalar el sistema es esencial. La idea clave radica en localizar rápidamente la funcionalidad que supone ese cuello de botella del sistema y proveerlo de los recursos necesarios para que pueda seguir respondiendo a las peticiones, evitando una situación catastrófica.

3. ***Rendimiento***: se trata de sistemas distribuidos de gran complejidad, donde hay muchas aplicaciones y módulos que necesitan comunicarse, desempeñando una gran cantidad de funcionalidades.

4. ***Tiempo de respuesta***: los sistemas de banca deben responder lo más rápido posible a las peticiones realizadas en general, pero es posible que haya ciertos módulos dedicados a datos financieros, por ejemplo, que se actualizan constantemente y por tanto no se puede permitir retrasos en la obtención de la información calculada por dichos módulos.

5. ***Seguridad***: se debe estudiar desde dos perspectivas, la protección del *sistema* (accesos, privilegios,...) y también de los *datos* que circulan por el mismo, sobre todo a nivel de transacción (cifrado, autenticación, etc.).

6. ***Fiabilidad***: el sistema debe ser tolerante a fallos, identificando el lugar donde ha ocurrido el fallo si es a nivel software o el equipo problemático si es un problema hardware. Las arquitecturas orientadas a servicios ayudarán mucho en esta tarea, por motivos que se mencionarán.

7. ***Extensibilidad***: ¿qué ocurre si hay que añadir nuevas funcionalidades al sistema, como por ejemplo, un nuevo modelo de análisis de riesgos, o un nuevo servicio que ofrece la entidad? ¿Cuál sería el coste de añadirlas? La arquitectura escogida deberá minimizar el esfuerzo en añadir esas nuevas características.

8. ***Mantenibilidad***: en línea con el requisito anterior, la arquitectura debe escogerse de forma que propicie una simplificación (dentro de lo posible) de las tareas de mantenimiento del sistema, frente a posibles fallos a nivel hardware o software.

9. ***Usabilidad***: se busca un acceso lo más ubicuo posible, en tanto que los sistemas de banca son accedidos, en general, por muchos tipos de personales y organizaciones, y desde sitios muy heterogéneos: comerciantes, clientes, inversores, etc... Además, las interfaces de acceso deben ser lo más "*amigables*" posible, de cara a garantizar el éxito de implantación del sistema.

3. Diseño arquitectónico

El diseño arquitectónico sigue una tónica general en las diversas referencias estudiadas: un **modelo orientado a servicios**. Se trata de tener un componente front-end (interfaces de acceso al sistema) y una serie de servicios, mecanismos de interconexión y base/s de datos.

A continuación se exponen varios modelos de arquitecturas que han sido planteadas en los diversos artículos analizados. He intentado organizarlas de forma que se vaya viendo la arquitectura más genérica e ir llegando a los enfoques más "especiales", es decir, de una arquitectura puramente orientada a servicios a otras con ligeras diferencias sobre ella.

3.1. Aproximación inicial: diseño orientado íntegramente a servicios/microservicios

El patrón de diseño es orientado a servicios. Este es el enfoque más "sencillo" y se organiza precisamente en los mismos subsistemas ya comentados al principio de la sección: front-end, back-end y mecanismos de interconexión.

En cuanto a las **interfaces de usuario** (front-end), se habla de adaptarse a las nuevas tecnologías de la web, desarrollando así interfaces web rápidas, que sean capaces de comunicarse oportunamente con los servicios adecuados para responder a la petición.

Como ya referíamos, en el modelo SOA no suele haber una interfaz de usuario para cada servicio, sino que ésta se desarrolla como un módulo aparte y permite acceder a las funcionalidades de los servicios, que están en el back-end. Ello contribuye de alguna forma a construir una nueva barrera de seguridad de acceso al núcleo del sistema, donde están las posibles bases de datos y los servicios.

Además, se habla de que el modelo SOA evita al usuario tener que acceder a través de terminales a las funcionalidades del *mainframe* (sistema monolítico), sustituyéndolas por interfaces web (API REST, normalmente), contribuyendo con ello a una mayor ubicuidad del sistema, pues serán mucho más accesibles que los programas instalables nativamente tradicionales.

En la zona del **back-end**, se configuran una serie de servicios, que, por definición, son independientes entre sí, aunque con algunas diferencias dependiendo si se adopta un modelo de arquitectura:

- ***SOA***, en la cual los servicios pueden "compartir", por ejemplo, el modelo de datos, que, como veremos en la arquitectura propuesta por el Credit Suisse, es común para todo el sistema.
- ***microservicios***, los cuales están más orientados al despliegue y al desarrollo de componentes.

Hace falta también una serie de herramientas que de alguna manera administren esos servicios: orquestación, actualizaciones, control del ciclo de vida, etc. Hay dos grandes alternativas: o bien se desarrolla una herramienta propia de gestión, con toda la complejidad y coste que ello supone; o bien pueden usarse tecnologías ya elaboradas, ya sean libres o propietarias (Docker, Podman, ...).

Los servicios acceden a una base de datos que suele ser común, compartiendo el modelo de datos de la misma todos ellos. Esto no ocurrirá en los microservicios, por su propia filosofía.

Es precisamente el Credit Suisse el que adopta la primera filosofía: tal y como los autores indican en el artículo, en el momento en que quisieron implementar esta tecnología de gestión de los servicios no encontraron ninguna que les convenciese, con lo que decidieron elaborar una propia para el sistema: **IFMS (Interface Management System)**. IFMS era una base de datos que incluía todos los metadatos relevantes, de forma independiente de la implementación, básicamente información semántica (elementos comerciales afectados por el servicio y las pos y pre-condiciones), información no funcional (como el rendimiento del servicio) e información relacionada con la implementación (por ejemplo, las plataformas y tecnologías en las que el servicio está disponible).

Más adelante expongo un caso de diseño basado en ***microservicios***, que aplica la filosofía arquitectónica que estamos explicando, y que usa las herramientas de **Docker** para desplegar los microservicios en contenedores y poder gestionarlos cómodamente y con la garantía que proporciona un software de esta categoría.

El **mecanismo de interconexión** es bastante "subjetivo" y cada cual lo llega a implementar como cree más conveniente. Existen diversas estrategias:

- Se puede idear un "***canal de comunicación global***", que conecte todo el sistema: front-end y back-end, y las interfaces entre los servicios web. Esta fue la elección que hicieron los

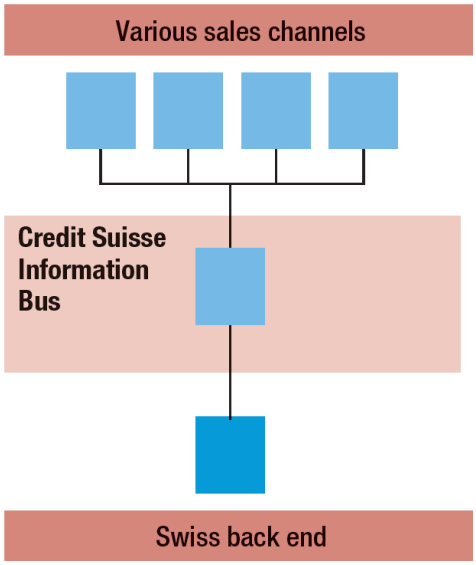

Figura 1: Arquitectura simplificada del Credit Suisse Bank de Suiza. Fuente: [1].

equipos de desarrollo del Credit Suisse Bank (cfr. [1]), al cual llamaron "***Credit Suisse Information Bus***", y que interconectaba los subsistemas principales de la arquitectura descrita. La arquitectura (simplificada) del Credit Suisse se puede ver en la figura 1.

- La otra opción suele ser usar mecanismos de paso de mensajes entre servicios (*RabbitMQ*, RPC, etc.) y comunicación con el front-end a través de API REST, como el caso del Dankse Bank (cfr. [2]).

En cuanto a la **seguridad**, como ya se indicó en los requisitos no funcionales, hay dos aspectos clave:

- la seguridad del propio *sistema*: para el acceso se usan mecanismos basados en canales seguros como VPN,
- la seguridad a nivel de *negocio*, con todo lo que ello implica, se intenta cubrir con mecanismos de autenticación de usuarios, jerarquía de autoridad para gestión, e incluso auditorías de servicios de seguridad en algunos casos.

3.2. Un caso concreto: microservicios sobre contenedores (a partir de [2]).

La alternativa planteada para el Dankse Bank se sustenta sobre un idea básica: ***alojar microservicios en contenedores***.

La arquitectura se puede visualizar en la Figura 2.

Se utilizan contenedores para almacenar todos los microservicios que conforman el sistema al completo. Estos contenedores son gestionados por el software de orquestación **Docker**, y en concreto con la herramienta *API Docker Swarm Cluster*[1].

Además, se usa la tecnología de **Docker Registry** para almacenar y gestionar así de forma centralizada (y por tanto más cómoda y eficaz) las *imágenes* de los contenedores. Ello garantiza la correcta gestión de las mismas bajo la tecnología oficial: Docker.

En cuanto a las tareas de **automatización**, toda la funcionalidad asociada al despliegue de las imágenes de los contenedores se puede realizar ahora usando únicamente un comando de Docker.

[1]Esta herramienta permite agrupar una serie de hosts de Docker en un clúster y gestionar los clústeres de servidores de forma centralizada, así como orquestar los contenedores.

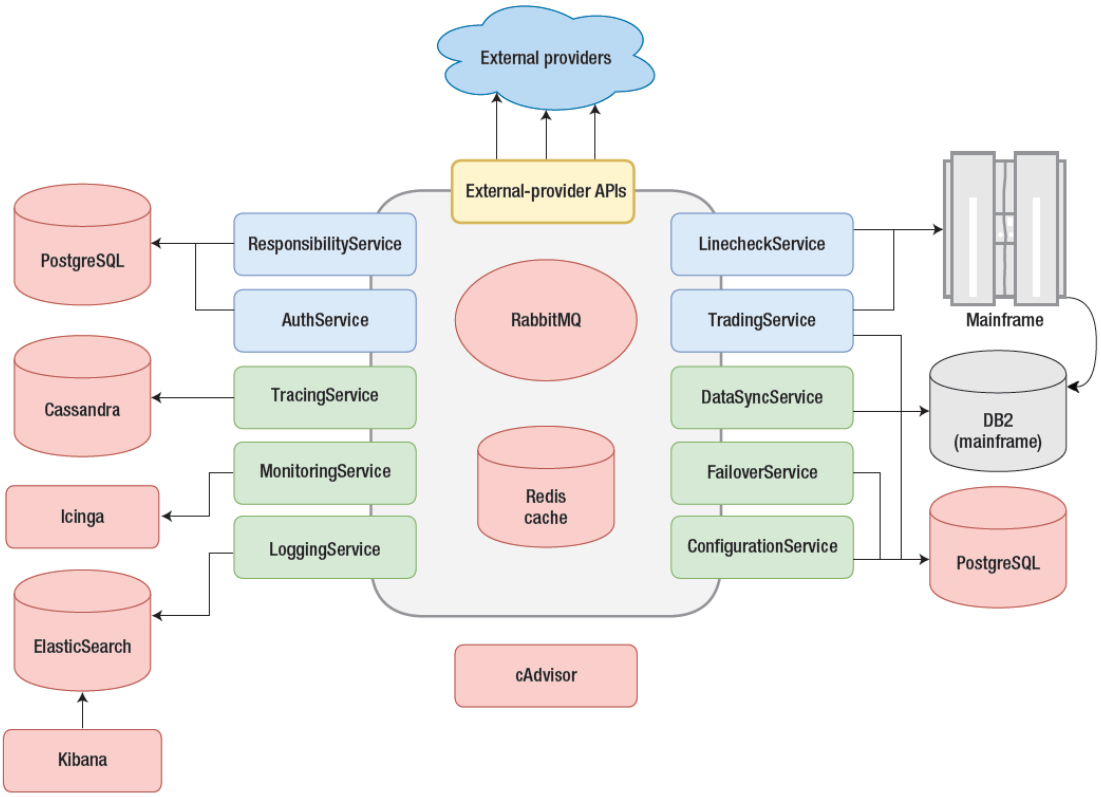

Figura 2: Arquitectura orientada a microservicios propuesta por el equipo de IT del Dankse Bank para su nuevo núcleo *FX Core*. Fuente: [2].

Las nuevas imágenes se **despliegan** sobre el *Docker Registry* una vez que han pasado una fase de construcción y *testeo* satisfactorias siguiendo una estrategia de *Continuous Integration*(integración continua).

Todos los servicios se rigen por una metodología de cauce **CICD**[2] (*continuous integration and continuous deployment*).

En cuanto a la **orquestación** de los servicios, se usa la herramienta ***Docker Swarm***, la cual permite la restauración total de servicios que han fallado. Además, provee otras herramientas como el balanceo de carga.

Por su parte, la **integración de los microservicios** se logra con *mecanismos de paso de mensajes* entre los mismos. Concretamente, se propone ***RabbitMQ***.

Para el **almacenamiento** de los datos, se propone el uso de la tecnología ***Red Hat OpenShift***, que proporciona el servicio *cloud* **PaaS** (*Platform as a Service*), en tanto que ofrece una infraestructura para la programación de las aplicaciones/servicios con soporte para múltiples lenguajes de programación. La decisión de escoger esta herramienta no es en absoluto baladí: *Openshift* ofrece grandes ventajas en cuanto a la escalabilidad de los microservicios mediante la replicación de contenedores que les sirvan de forma individual, en la cantidad necesaria. Además, aísla al desarrollador de los detalles concretos de su tecnología.

A pesar de todo, en una etapa previa al desarrollo del sistema desde el enfoque de los contenedores, esta entidad ha optado por usar máquinas virtuales gestionadas y lanzadas manualmente por el grupo de trabajo del FX Core (núcleo del sistema).

Algo muy importante, y que ya refería más arriba, es el uso de una ***caché*** temporal para

[2]Surge de la combinación de dos técnicas de ingeniería del software: integración continua (los cambios del código nuevo en una aplicación se diseñan, se prueban y se combinan periódicamente en un repositorio compartido) y entrega continua (cambios que implementa un desarrollador en una aplicación, a los que se les realizan pruebas de errores automáticas y que se cargan en un repositorio (como GitHub o un registro de contenedor), para que luego el equipo de operaciones pueda implementarlos en un entorno de producción en vivo). Fuente: [7].

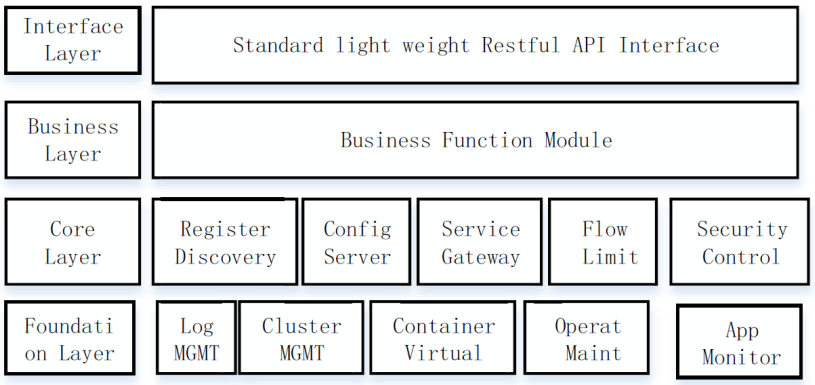

Figura 3: Arquitectura orientada a microservicios propuesta en [4]. Fuente: [4].

poder gestionar adecuadamente y que el sistema siga funcionando completamente hasta que éste se componga únicamente de microservicios. La caché constituye el puente entre las dos partes del sistema: la nueva y la monolítica, de la cual se siguen extrayendo datos e incluso procesando alguna tarea.

3.3. Un paso más en el diseño: microservicios organizados en capas

Damos un paso más, y presentamos ahora una visión más "estructurada" de la arquitectura de microservicios: modelo en capas. Este diseño fue propuesto para la *Plataforma del Banco Regional de China* (cfr. [4]) y es bastante común, en base a las referencias usadas para este proyecto, de cara a agrupar de una mejor forma los subsistemas de los que hemos hablado al empezar a describir la arquitectura. Estas cuatro capas, genéricamente, son:

1. La capa de interfaz: ***Interface Layer***,
2. La capa de la lógica de negocio: ***Business Layer***,
3. La capa de gestión de los microservicios: ***Core Layer***,
4. Y en el nivel inferior, la capa de gestión a más bajo nivel: ***Basic Function Layer***.

La arquitectura se puede ver en la Figura 3.

En cuanto a la Capa de Interfaz, o **Interface Layer**, ésta implementa la comunicación con el exterior, es el conjunto-interfaz del sistema. La tecnología más frecuente usada en esta capa es REST, desarrollando API REST para acceso a los servicios.

La capa de lógica de negocio, o **Business Layer**, se compone de microservicios que implementan todas las funcionalidades propias de la lógica de negocio necesaria. Estos microservicios procesan transacciones, líneas de créditos, autorizaciones sobre determinadas operaciones en el sistema, y otras tareas similares.

La capa de gestión de microservicios (**Core Layer**) se ocupa de la gestión integral de microservicios. Abarca las tareas de control del ciclo de vida, del descubrimiento y registro de los microservicios, la configuración integral (centralizada) de los mismos, así como de los mecanismos de comunicación "inter-process", los cuales son básicos, como ya se ha comentado, para la comunicación entre microservicios.

Esta capa se ocupa también de otras tareas como el control de acceso al sistema y los mecanismos de seguridad y el balanceo de carga.

La última capa, **Basic Function Layer**, está dedicada a la gestión del sistema a más bajo nivel, en tanto que se encarga de las siguientes tareas:

1. Funcionalidades para la gestión de recursos hardware.

2. Gestión y mantenimiento de clústers, con herramientas como Kubernetes, Docker Swarm o CoreOS.

3. Monitorización y visualización de datos; destacan los monitores para servidores Zabix y Nagios, entre otros.

4. Recopilación de bitácora (log) para posibles labores de auditoría; se usan herramientas como Kafaka, Dapper, Zipkin, y HTrace,

5. Manejo de errores y excepciones.

6. Tecnologías de contenedores y de virtualización (según el caso, lo que se use), con herramientas como Docker, CoreOS, RancherOS o VMWare.

7. Otras funcionalidades secundarias.

3.4. Otro enfoque basado en capas: separar modelo de negocio y el núcleo con los servicios.

El Banco Comercial de China (cfr. [3]) propuso, en base a una serie de estudios realizados en el país, un modelo rompedor de arquitecturas para sistemas de banca que se basase en: la implementación de servicios para el núcleo del sistema (la capa *Service Layer* ya comentada) por un lado, y además una capa que se encargase de todas las cuestiones puramente de negocio ("no informáticas"), y una tercera capa (la intermedia) que sería precisamente eso, una *moderadora* entre ambas.

La arquitectura basada en 3 capas propuesta se especifica en la Figura 4 y se compone de:

1. ***Business Layer***. Tiene que trabajar con las peticiones de los usuarios. Incluye "módulos" asociados a la asistencia a los clientes, innovación de los productos e información multidimensional de gestión (para elaborar informes, básicamente). El objetivo de esta capa es, por un lado (y es lo más evidente) proveer de una interfaz al sistema pero a la vez captar correcta y eficientemente la "petición" que el usuario ha solicitado.

2. ***Semantic Layer***. Es el "puente" entre las dos capas. Su función es tan vital aquí como lo eran las API REST en las otras arquitecturas vistas. Sin esta capa, los servicios, es decir, el núcleo, no pueden saber qué hay que hacer, porque no se les ha comunicado. Pues bien, esta capa, como ya adelantábamos se compone de un protocolo de comunicación propio llamado FL7 y que usa una descripción en lenguaje de primer orden para descomponer la descripción de la petición en una proposición simple y a continuación deduce las formas correctas y las reglas de inferencia, es decir, elabora una descripción lógica de primer orden a partir de esa proposición simple.

3. ***Service Layer***. Compone el núcleo del sistema, las funcionalidades elaboradas a partir de servicios, para cuya construcción se tenía en cuenta el modelado de negocio (tal y como se explicará en la sección *Estrategias de desarrollo*).

4. Estrategias de desarrollo

En líneas generales:

1. Se opta por ir ***reemplazando poco a poco*** una arquitectura plenamente monolítica en un modelo orientado a servicios (o microservicios). Se evita un cambio brusco para poder así controlar posibles problemas que puedan surgir. Surge aquí la necesidad de usar una caché, como ya se ha comentado en la Sección 3.

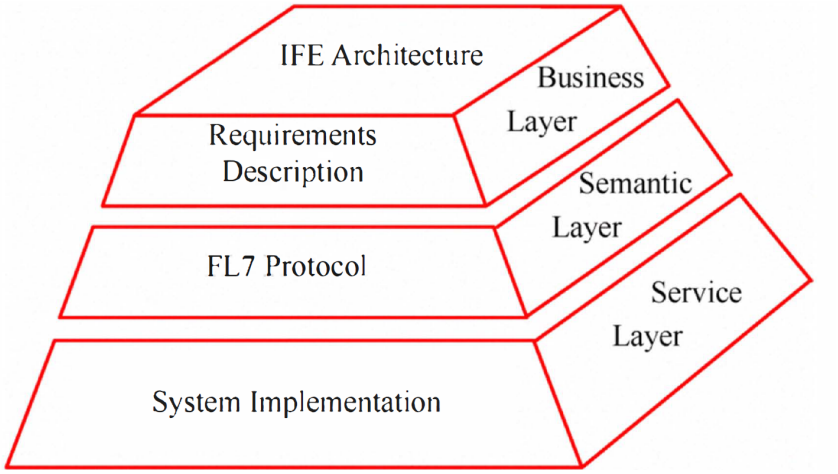

Figura 4: Arquitectura orientada a microservicios propuesta en [3]. Fuente: [3].

2. Una de las estrategias planteadas (por el Credit Suisse, en concreto) fue intentar lograr un equilibrio entre el control de calidad de la arquitectura (expuesto más abajo) y la flexibilidad que caracteriza a los equipos de desarrollo, poniendo el foco en las decisiones de diseño que afectaban más globalmente al sistema: *controlar que no haya servicios redundantes, más que fijarse en el formato de los datos*, por ejemplo.

3. El modelo de organización de los equipos de desarrollo es muy diferente con respecto al tradicional: los grupos de trabajo van a ser *más reducidos*, para fomentar una muy buena comunicación interna, y se van a encargar, en general, de elaborar **un servicio cada uno de los grupos** (y algún otro que pueda surgir a partir de éste, si fuera necesario). Es decir, se abandona el enfoque de "áreas de trabajo" por grupo de desarrollo y se centra en funcionalidades independientes (lo máximo posible).

Se plantean varias estrategias de desarrollo, de entre las cuales destaca una de ellas, quizás bastante adecuada para este tipo de sistemas, donde se pone el foco en el "refinamiento" y la secuencialidad necesaria para el desarrollo de los servicios y de su posterior orquestación.

Dicha estrategia ha sido referida por el Credit Suisse Bank (cfr. [1]) y en ella se parte de que los proyectos van a ir solicitando la información que necesiten para desarrollar su propia funcionalidad (servicio). El proceso es:

Fase 1. En una primera fase, los arquitectos encargados de la integración deberán dar solución a esa petición, ofreciendo uno de estos tres caminos:

- hay disponible un servicio que proporciona esa información, luego puede ser usado;
- no hay un servicio que ofrezca esa información o similar, entonces el servicio demandante podrá construir un servicio privado que le nutra de esa información;
- o bien es posible que haya un servicio que aporte parte de esa información, y el resto se puede obtener, luego puede contemplarse la ampliación de este servicio.

Fase 2. En una segunda fase, un grupo de diseñadores revisaría y corregiría al completo la arquitectura planteada para dejarla lista para su implementación.

Fase 3. En una última fase, un grupo de especialistas comprueba que la arquitectura planteada cubre todos los requisitos no funcionales definidos.

Una ***segunda estrategia*** destacable es la propuesta por el Banco Comercial de China (cfr. [3]), orientada a diferenciar entre el análisis del componente puramente de negocio y del diseño de los servicios software. Entre ellos, una "capa puente" que rescata qué se ha pedido y qué

servicios, en consecuencia, se demandan. Realmente, en el artículo hacen referencia al proceso de *modelado*, pero me ha parecido muy interesante y por eso se incluye aquí muy brevemente. La estrategia contempla:

Fase 1. Modelado de negocio: consiste en establecer un modelo de conocimiento del dominio comercial y dar una clara definición del negocio y el proceso. Se empezó a trabajar, dentro de esta parte, sobre los procesos de negocio de la entidad, procesos organizacionales y procesos de gestión e ideas culturales. Otros aspectos tratados fueron: los requisitos de supervisión y gestión, y los conceptos avanzados del sistema central extranjero.

Fase 2. Modelo de servicios. Está dedicado a los servicios SOA, y se desarrolla en dos pasos:

1. *Definición del servicio.* Define dos partes: la parte abstracta y la parte concreta. La primera de ellas decribe las funcionalidades de los servicios, y contiene una serie de componentes independientes: interfaz, operaciones (y su semántica), definiciones de las estructuras de datos y la segunda parte decribe el acceso a los servicios.

2. *Implementación del servicio.* Es la fase de codificación del servicio.

Fase 3. Protocolo FL7. Es un protocolo propio diseñado para ser compatible con los estándares de protocolos actuales. FL7 convierte los requisitos de negocio al servicio que se encargará de la reestructuración del requisito y de la llamada a los servicios adecuados.

4.1. Desarrollo Dirigido por Modelos (MDD) aplicado a las tecnologías de microservicios (MSA)

Debido al escaso número de referencias encontradas por mi parte sobre estrategias de desarrollo acerca de las arquitecturas de servicios/microservicios en banca, expongo brevemente en este apartado una propuesta más genérica: un modelo de desarrollo muy conocido, aplicado a arquitecturas orientadas a microservicios, propuesto en [6].

La motivación básica de ese artículo es justificar por qué un Desarrollo Dirigido por Modelos (MDD) es una muy buena alternativa como estrategia de desarrollo para arquitecturas orientadas a microservicios (y también a servicios).

Me pareció una reflexión muy interesante, debido al tema que estamos tratando en el presente Caso Práctico.

> Un **modelo** es un medio para describir aspectos concretos de un sistema en un nivel apropiado de abstracción, por ejemplo, omitiendo detalles técnicos.
>
> En esta misma línea, el **Desarrollo Dirigido por Modelos (MDD)** es, tal y como se refirió en el Tema 1, un enfoque de diseño de software donde se usan modelos para describir la arquitectura del sistema.

Las ideas básicas entorno a las que gira ese por qué de usar MDD en microservicios son (en base a lo expuesto en [6]):

- **Identificación de servicios basada en modelos.** Esto es, la estretegia MDD permite que los desarrolladores de servicios y los expertos en el ámbito de la entidad (bancaria, en nuestro caso) conformen un modelo de dominio común, de donde surgirán los microservicios y su interfaz.

- **Heterogeneidad de las tecnologías usadas:** pues si, dado el caso, un nuevo desarrollador tiene que incorporarse al grupo de trabajo de un determinado servicio, tener su modelo **abstracto** le servirá para poder entender mejor dicho servicio, evitando tener que repasar «de golpe» las tecnologías con las que se está desarrollando el servicio, y que puede que él no conozca lo suficiente.

- El modelo de desarrollo que se propone en [6] está, según se indica expresamente, orientado a equipos de desarrollo de DevOps.[3] En esta línea, otra ventaja que aportaría el modelo MDD sería que dentro del mismo equipo de un microservicio se pueden desarrollar diferentes *modelos de puntos de vista* de acuerdo al rol de cada DevOps.

En líneas generales, la propuesta que se hace en [6] plantea lo siguiente:

- Si el equipo de trabajo usa MDD para desarrollar el/los microservicio/s, entonces: (1) se desarrolla un *modelo de dominio* común (DM), (2) el desarrollador del servicio deduce de él los posibles microservicios, junto con sus estructuras de datos y las interfaces agrupándolo en un *modelo de servicio* (SM); (3) el administrador de sistemas (*operator*, de Dev**Ops**) desarrolla un modelo *operacional* (OM) para especificar el despliegue de esos microservicios.

- A partir de los modelos de *dominio* (DM) y *servicio* (SM), se generará un *modelo de servicio*, que se subirá a un ***repositorio*** compartido entre todos los equipos de desarrollo del proyecto, y que hace las veces de sistema de control de versiones. La herramienta que permite esta transformación es SMT (*Service Model Transformation*)

- Igual que se hace con los *modelos de servicio*, se puede hacer para extraer y compartir el *modelo operacional*, mediante la correspondiente herramienta OMT (*Operational Model Transformation*.

- Esta forma colaborativa hace que un servicio llamado B que utiliza alguna vez el servicio A puede importar algunos datos (como la descripción de la interfaz) para poder acceder a ese servicio. De la misma forma, un servicio C podrá usar esos modelos de servicio y operación (despliegue, desarrollo, ...) para poder generarse a sí mismo, incluso si pretende usar una tecnología distinta (en este caso necesitará descargar del repositorio común algún modelo existente asociado a esa tecnología). Como es lógico, C también *publicará* en el repositorio todos esos modelos.

Como valoración, este enfoque da al proceso de desarrollo un espíritu colaborativo, sencillo, intuitivo, que cubre mayoritariamente las necesidades de cada uno de los equipos de desarrollo, a través del uso de modelos y puntos de vista para confluir a los diferentes DevOps (cada día más necesarios) en una estrategia común, coordinada dentro del equipo y coordinar igualmente a los mismos equipos, pero manteniendo esas «distancias de seguridad» entre los microservicios que están desarrollando (esas distancias son parte del *encanto* de esta arquitectura, pues aportan ventajas que motivan su uso, como ya se ha ido refiriendo a lo largo de este documento).

[3] Combinación de administradores de sistemas e ingenieros software.

Referencias

[1] Stephan Murer, Claus Hagen: *Fifteen Years of Service-Oriented Architecture at Credit Suisse.* IEEE Software 31(6):9-15 (2014). DOI `http://dx.doi.org/10.1109/MS.2013.137`.

[2] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, Manuel Mazzara: *From Monolithic to Microservices: An Experience Report from the Banking Domain.* IEEE Software, vol. 35, no. 3, pp. 50-55, May/June 2018. DOI `http://dx.doi.org/10.1109/MS.2018.2141026`

[3] S. Zhang, Y. Wang and G. Zhang, "*The service architecture of commercial bank*,"4th International Conference on New Trends in Information Science and Service Science, Gyeongju, 2010, pp. 180-184.

[4] X. Wang, S. Wang, Z. Hao and X. Zhang, "*Research on the Construction of Regional Credit Bank Platform Based on Microservices*,"2018 10th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), Changsha, 2018, pp. 452-456, doi: 10.1109/ICMTMA.2018.00116.

[5] K. Gos and W. Zabierowski, "*The Comparison of Microservice and Monolithic Architecture*,"2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), Lviv, Ukraine, 2020, pp. 150-153, doi: 10.1109/MEMSTECH49584.2020.9109514.

[6] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. 2019. *A model-driven workflow for distributed microservice development.* In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19). Association for Computing Machinery, New York, NY, USA, 1260–1262. DOI:https://doi.org/10.1145/3297280.3300182

[7] Para el concepto de CICD: `https://www.redhat.com/es/topics/devops/what-is-ci-cd`