UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de software basado en microservicios: un caso de estudio para evaluar sus ventajas e inconvenientes

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Víctor Alberto Iranzo Jiménez

Tutor: Patricio Orlando Letelier Torres

Curso 2017-2018

Resum

????

Paraules clau: Microservices, Arquitecturas de software, ??????????????

Resumen

????

Palabras clave: Microservicios, Arquitecturas de software, ??????????????

Abstract

????

Key words: Microservices, Software Architecture, ?????????????

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

1.1 Motivación

En la actualidad, no es necesario un alto grado de conocimientos en ingeniería del software para desarrollar una aplicación o sistema. Personas que no tienen estudios relacionados con la informática pueden producir código que, sin ser limpio y elegante, funciona. Desarrollar sistemas de calidad requiere de grandes conocimientos, pero minimiza los costes y aumenta la productividad de una organización. Se debe poner el foco en emplear una arquitectura de software que se adapte a nuestras necesidades. De lo contrario, el futuro mantenimiento será más costoso y repercutirá en la confianza de los clientes y en la moral del equipo. [14]

Las arquitecturas basadas en microservicios son una tendencia actual que emerge asociada a conceptos clave como la integración continua, el desarrollo centrado en el dominio del problema o el despliegue en contenedores. En estas arquitecturas diferentes funcionalidades se encapsulan en servicios pequeños y autónomos que cooperan entre ellos. En términos de diseño, principios como el de Responsabilidad Única son más fáciles de conseguir y los desafíos de organización del código pueden abordarse de formas más diversas por la baja granularidad de la arquitectura.

1.2 Objetivos

El objetivo de este proyecto es validar con un caso de estudio las ventajas e inconvenientes de una arquitectura basada en microservicios frente a una arquitectura monolítica. Concretamente,los objetivos específicos son:

- Desarrollar una misma aplicación para la venta de productos y la gestión de pedidos siguiendo dos arquitecturas diferentes: una basada en microservicios y otra monolítica.
- Comparar el proceso de desarrollo de ambos sistemas a lo largo del ciclo de vida del software.
- Evaluar cómo se pueden llevar a cabo diferentes modificaciones durante el mantenimiento de ambas aplicaciones una vez se ha finalizado su implementación.
- Verificar que una arquitectura basada en microservicios facilita alcanzar los requisitos no funcionales de escalabilidad y tolerancia a fallos frente a una arquitectura monolítica.

1.3 Estructura de la memoria

A continuación, se presenta la estructura de la memoria:

El capítulo 1 introduce la memoria a través de la motivación del trabajo y los objetivos a cumplir.

En el capítulo 2 se presenta una definición para el término microservicio y se describe el estado del arte de estos a lo largo del ciclo de vida del software.

En el capítulo 3 se analiza la aplicación a desarrollar mediante su especificación empleando casos de uso y un modelo de dominio. Además, se presenta el plan de trabajo y las tecnologías empleadas.

El capítulo 4 explica el diseño e implementación del sistema siguiendo una arquitectura monolítica.

El capítulo 5 se centra en los mismos aspectos que el capítulo anterior pero para una arquitectura basada en microservicos.

En el capítulo 6 se realiza una comparación de ambas soluciones respecto a los requisitos no funcionales de escalabilidad y tolerancia a fallos y se evalúa como afrontaría cada solución diferentes situaciones de mantenimiento.

El último capítulo presenta las conclusiones del trabajo respecto a los objetivos planteados inicialmente.

Adicionalmente, al final de la memoria se adjuntan un conjunto de apéndices. En ellos se resumen las ventajas e inconvientes de los microservicios, se incluye un modelo de la aplicación desarrollada y se detallan una serie de prácticas de interés seguidas en su desarrollo.

CAPÍTULO 2
Estado del arte

2.1 ¿Qué son los microservicios?

Según Newman, los microservicios son servicios pequeños y autónomos que trabajan conjuntamente. [17] Podemos desglosar esta definición así:

- Un **servicio** es un conjunto de funcionalidades que se expone a los clientes para que las empleen con diferentes propósitos. [26] La programación orientada a servicios es un paradigma que se aplica en las arquitecturas orientadas a servicios (SOA). El objetivo principal de SOA es desarrollar servicios que aporten valor al negocio y se adapten a los cambios en las necesidades de los clientes, de forma ágil y con el menor coste posible. Las arquitecturas orientadas a servicios no están asociadas a ninguna tecnología específica. En líneas generales, dividen un sistema en componentes que cambian por el mismo motivo y promueven la flexibilidad y los servicios compartidos frente a implementaciones específicas y óptimas. Son muchos los beneficios de estas arquitecturas, sin embargo existe una falta de consenso sobre cómo debe llevarse a cabo este tipo de arquitecturas en aspectos como los protocolos de comunicación a emplear o la granularidad de los servicios. [2] Los microservicios pueden entenderse como una aproximación específica de las arquitecturas SOA.

- Diseñar microservicios con el menor **tamaño** posible no debe ser el foco principal. En todo momento han de cumplirse los principios de cohesión y coherencia: el código relacionado debe agruparse conjuntamente porque será modificado por el mismo motivo.

- Los servicios han de ser lo menos acoplados posibles para garantizar la **autonomía** de cada uno. Cada microservicio es una entidad separada: cambian de forma independiente al resto y al hacerlo sus consumidores no necesiten ser modificados, a menos que se modifique el contrato entre ambas partes. Para lograrlo, lo más habitual es que cada servicio exponga una interfaz (API) y todas las comunicaciones se realicen mediante llamadas a través de la red.

Otra definición interesante es la que aportan Lewis y Fowler. Según ellos, los microservicios son una aproximación para desarrollar una aplicación compuesta por pequeños servicios, cada uno ejecutándose en su propio proceso y comunicando a través de mecanismos ligeros. Estos servicios se construyen alrededor de las capacidades de negocio y se despliegan de forma independiente. [13] Cada funcionalidad se encapsula en un servicio que puede escalar de forma diferente de acuerdo a sus necesidades, a diferencia de las aplicaciones monolíticas donde se debe replicar el monolito al completo.

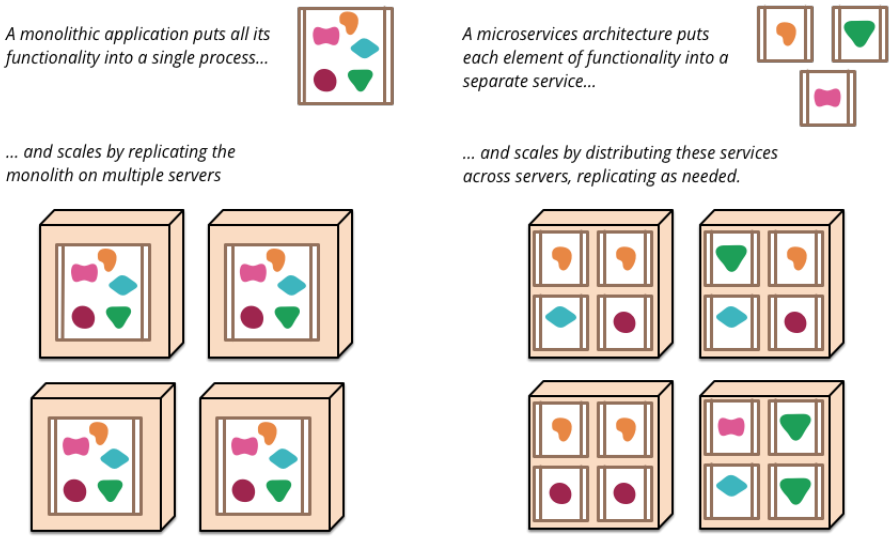

Figura 2.1: Los microservicios escalan de acuerdo a su carga de trabajo para asegurar la disponibilidad de la funcionalidad que ofrecen. [13]

2.2 Los microservicios en la fase de requisitos

La fase de requisitos del software es aquella en la que se elicitan, analizan, documentan, validan y mantienen los requisitos del sistema. Los requisitos del software expresan las necesidades y restricciones asociadas a un sistema. [6] El artefacto principal que se produce en esta fase es el documento con la especificación de requisitos software (ERS). Una vez validado dicho documento por los stakeholders se puede iniciar la fase de diseño la solución. Esto no significa el final de esta fase del proceso: la gestión de los requisitos continúa durante el resto del desarrollo del producto.

2.2.1. Requisitos funcionales y no funcionales

Los requisitos se pueden clasificar en funcionales y no funcionales. Los **requisitos funcionales** (RF) describen la funcionalidad que los usuarios esperan del sistema. Los **requisitos no funcionales** (RNF) son restricciones impuestas sobre el sistema a desarrollar, estableciendo por ejemplo como de rápido o fiable ha de ser. Mientras que los primeros no incluyen ninguna mención relacionada con la tecnología que emplea el sistema, los segundos sí pueden establecer restricciones de este tipo. Por ejemplo, un requisito no funcional puede consistir en desarrollar una aplicación en un lenguaje de programación específico o hacer que esté disponible para diferentes sistemas operativos móviles. Por este motivo, los requisitos deben ser tenidos en cuenta a lo largo de todo el desarrollo del sistema.

Los requisitos funcionales y no funcionales son ortogonales en el sentido de que diferentes diseños de software pueden ofrecer la misma funcionalidad (RF) pero con distintos atributos de calidad (RNF). Los arquitectos software están más centrados en los requisitos no funcionales porque estos son los que conducen hacia la elección de una u otra arquitectura. Los requisitos no funcionales pueden influir en los patrones arquitectónicos a seguir, las futuras estrategias de implementación del sistema o la plataforma sobre la que se desplegará el sistema. [1]

Figura 2.2: Características y subcaracterísticas definidas en el modelo de calidad del producto de la ISO/IEC 25010. [23]

Requisitos no funcionales asociados a atributos de calidad como la tolerancia a fallos o la disponibilidad pueden conducir al arquitecto hacia la elección de una arquitectura basada en microservicios:

- La **tolerancia a fallos** se define en la ISO/IEC 25010 como la capacidad del sistema para operar según lo previsto en presencia de fallos hardware o software. [23] Cuando se escala un sistema, la probabilidad de fallo es inevitable. Muchas organizaciones invierten mucho esfuerzo en evitar que un fallo se produzca, pero muy poco en mecanismos para recuperar el sistema una vez se ha producido. Un sistema que por culpa de un servicio caído deja de funcionar es menos resilente que un sistema que puede continuar ofreciendo el resto de sus funcionalidades.
- La **disponibilidad** se define como la capacidad del sistema de estar operativo y accesible para su uso cuando se requiere. [23] En los sistemas distribuidos existen 3 características sobre las que se debe hacer balance: la consistencia, que establece que vamos a obtener una respuesta correcta de cualquier nodo de un servicio de acuerdo a su especificación, la disponibilidad, que ya hemos definido, y la tolerancia a particiones, que es la habilidad de gestionar situaciones en las que la comunicación entre las partes de un sistema se interrumpe. El teorema de CAP (debe sus siglas a las características en inglés Consistency, Availability y Partition Tolerance) establece que en caso de fallo, solo dos de las tres características pueden prevalecer. [12]

2.2.2. El teorema de CAP

Pongamos un ejemplo en el que un microservicio está replicado y se produce un fallo por el cual la comunicación entre las replicas se interrumpe y los cambios en una replica no se pueden propagar al resto. Los **sistemas AP** son los sistemas que surgen fruto de sacrificar la consistencia cuando un fallo se produce, mientras que en los **sistemas CP** se pierde la disponibilidad.

En el primer tipo, las replicas continuarían operativas, pero como los datos entre las replicas no se sincronizan se pueden obtener datos incorrectos al hacer una consulta. Cuando la comunicación se restablece, los cambios entre las replicas se sincronizarán. En los sistemas CP, para mantener la consistencia entre las replicas se tiene que rechazar cualquier petición, con lo que el servicio deja de estar disponible.

Los sistemas AP escalan más fácilmente y son más sencillos de construir, pero nos obligan a una consistencia eventual de los datos. Los segundos son los únicos que nos aseguran una consistencia fuerte, pero son más difíciles de construir. A la hora de optar por una solución u otra se debe tener en cuenta la especificación de requisitos, donde se debe detallar de forma específica y cuantitativa cuánto tiempo puede nuestro servicio estar inoperativo o con un dato obsoleto. Si en las fases siguientes se opta por una arquitectura basada en microservicios, no será necesario implementar el sistema completo

de una u otra forma. Cada microservicio tendrá necesidades diferentes y podrá seguir la aproximación que mejor le convenga. [17]

2.3 Los microservicios en la fase de diseño

En la fase de diseño se definen la arquitectura, componentes e interfaces del sistema. La especificación de requisitos es analizada para producir una descripción de la estructura interna del sistema, con el suficiente nivel de detalle para que sirva como base en su construcción. En esta fase se plantean diferentes diseños como alternativas de las que se debe hacer balance. Los modelos que se generan se emplearán para validar que se cumplen los requisitos establecidos y para planificar la fase de implementación. [22]

2.3.1. Librerías versus servicios

Un componente es una unidad de software que se puede reemplazar y actualizar de forma independiente. Las **librerías** son componentes que están ligados a un programa y son invocadas a través de llamadas a funciones. En cambio, los **servicios** son componentes que se ejecutan como procesos externos y con los que se puede comunicar a través de mecanismos como llamadas a procedimientos remotos (RPC) o peticiones HTTP. [13]

Uno de los motivos por los que se recomienda emplear servicios frente a librerías es que los servicios se pueden desplegar de forma independiente. Solo algunos cambios en la interfaz o contrato del servicio requerirán un cambio en sus consumidores. Además, algunas librerías obligan al uso específico de una tecnología. Sin embargo, las llamadas remotas son más costosas que las invocaciones dentro del mismo proceso, por lo que la interfaz del servicio debe definirse de tal forma que sus consumidores no tengan que comunicarse con él continuamente.

2.3.2. Diseño guiado por el dominio (DDD)

El **diseño guiado por el dominio** (DDD) es un enfoque para el desarrollo de software que propone un modelado rico, expresivo y evolutivo basado en la realidad del negocio. El dominio representa lo que hace una organización y la forma en qué lo hace.

Con esta aproximación los expertos del dominio y los desarrolladores se sitúan en el mismo nivel empleando un lenguaje ubicuo. No hace falta ninguna traducción de términos entre ellos porque todos hablan un lenguaje común, que se plasma hasta en el código de programación. El lenguaje no tiene porque seguir los estándares de la industria que representa: emplea los términos y acciones que en el negocio se dan. [25]

El lenguaje ubicuo que se emplea crece y cambia con el paso del tiempo. Nadie es capaz de conocer el dominio de un negocio completo porque este forma parte de un proceso de descubrimiento continuo. Si la organización sigue una estrategia de desarrollo ágil, en cada iteración se refina el modelo de forma incremental y este plasma en todo momento el software en funcionamiento.

Para su tratamiento, las áreas independientes del dominio se transforman en contextos delimitados. Cada contexto delimitado es un límite explícito que tiene su propio lenguaje ubicuo. Un concepto tiene un significado dentro de un contexto delimitado, pero fuera de él puede tener un significado totalmente diferente. No se puede tratar de incluir todos los conceptos en un único contexto: se deben separar los conceptos en diferentes contextos y entender las diferencias que existen para un concepto llamado igual en uno y otro contexto.

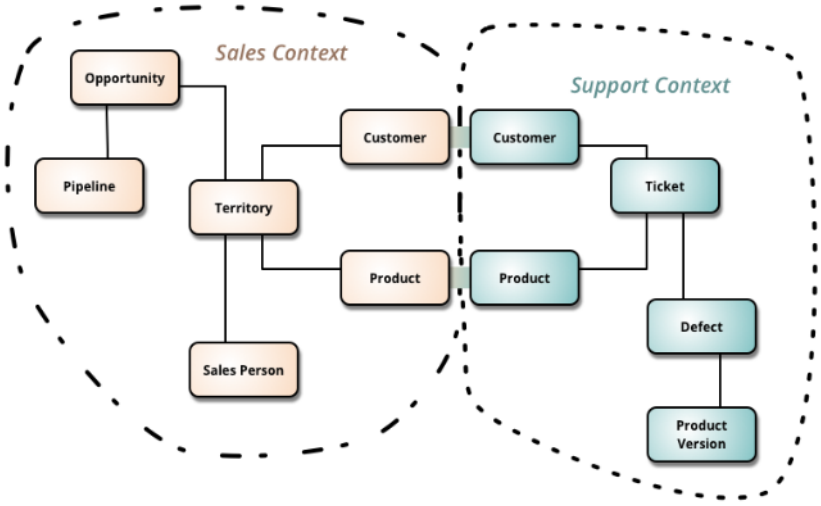

Figura 2.3: Ejemplo de dos contextos delimitados dentro del mismo dominio que emplean el mismo nombre para un concepto, pero con significados diferentes. [10]

Cada contexto está formado por modelos que no necesitan ser compartidos con otros a menos que se defina explícitamente una interfaz que los empleen. La interfaz es el punto de entrada para que otros contextos puedan comunicar con el nuestro, empleando los términos y entidades que en nuestros modelos se definan.

Esta perspectiva puede trasladarse fácilmente al modelado de microservicios. Los contextos delimitados que analicemos en nuestro sistema son firmes candidatos a transformarse en servicios. Así, los límites de un servicio quedan bien limitados porque todas las entidades que pueda requerir se encuentran dentro de sus fronteras, garantizándose su alta cohesión y bajo acoplamiento. [17]

2.3.3. Descomposición en microservicios

Cuando se razona sobre los límites de un servicio no se debe pensar en los datos que este almacena sino en las funcionalidades que ofrece. Pensar en los datos que almacena nos conduce a desarrollar únicamente servicios CRUD (en inglés, aquellos que nos permiten las operaciones de crear, leer, actualizar y eliminar datos), que ofrecen unas operaciones muy limitadas. Un servicio ofrece ciertas funcionalidades o capacidades que aportan **valor al negocio**.

Una descomposición temprana de un sistema en microservicios puede conllevar ciertos riesgos. Si el equipo a cargo del desarrollo tiene pocos conocimientos del dominio del problema a resolver, puede ser buena idea comenzar la implementación como si de un sistema monolítico se tratara.

Es aconsejable dividir la solución en grandes servicios que poco a poco se vayan dividiendo en más pequeños conforme se estudien las ventajas de hacer cada nueva extracción. Una vez sean conocidos los límites de cada servicio, se pueden refactorizar el código y los datos hacia una granularidad más fina. Como se puede ver en la figura, se puede migrar primero solo la funcionalidad del servicio sin preocuparse por sus límites en base de datos para no tener que realizar simultáneamente dos migraciones. Una vez nos aseguremos de que el servicio funciona correctamente, podemos migrar sus datos a una base de datos diferentes ya que cada servicio ha de ser dueño de sus propios datos. [21]

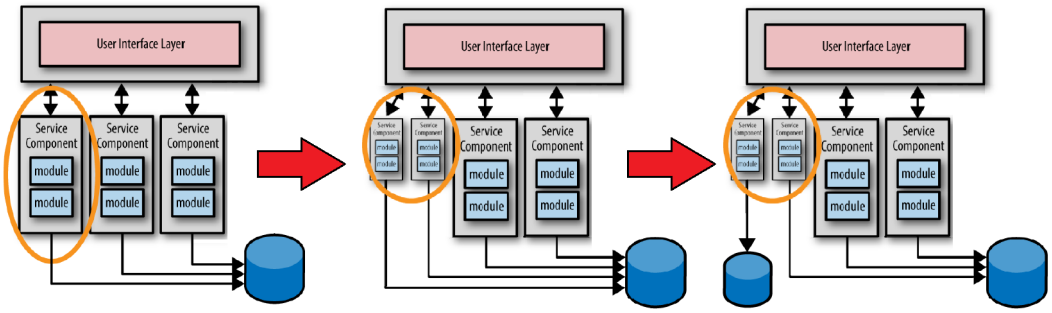

Figura 2.4: Primero, se dividen los grandes servicios en microservicios. Una vez hecho esto, se migran sus datos. [21]

Cuando sea necesario realizar un cambio por nuevos requisitos del negocio, estos se localizarán en un contexto bien delimitado porque existirá una correspondencia entre la estructura de la organización y los microservicios del sistema. Como consecuencia, el tiempo medio para realizar un cambio se verá reducido porque solo hará falta volver a desplegar una porción del sistema. Además, la comunicación entre microservicios se asemejará a la existente entre las entidades del negocio.

2.3.4. La tarea del arquitecto de software

El software ha de ser diseñado para ser flexible, adaptarse y evolucionar en función de los requisitos de los usuarios. En lugar de centrarse en diseñar un producto final perfecto, el arquitecto debe crear un entorno donde el sistema correcto pueda emerger creciendo progresivamente a medida que se descubren nuevos requisitos. Gracias a su modularidad, los microservicios son el entorno perfecto para que esto ocurra.

El arquitecto de software debe preocuparse más por como interaccionan los servicios entre ellos y no tanto en lo que ocurre dentro de sus límites. En organizaciones grandes, cada microservicio puede estar desarrollado por un equipo distinto y es el arquitecto quien debe hacer de puente entre ellos. [17]

Una de las ventajas de las arquitecturas basadas en microservicios es la **heterogeneidad tecnológica**: cada servicio puede ser desarrollado empleando una pila tecnológica distinta. No obstante, dejar plena libertad a cada equipo para elegir la tecnología del servicio que va a desarrollar puede traer problemas a la hora de integrarlo con el resto del sistema. El uso de contratos o establecer normas en aspectos clave como el protocolo de comunicación entre los servicios facilitará su consumo. Las decisiones de diseño de un servicio en particular pueden recaer en el equipo responsable. En este caso, el arquitecto solo juega un papel de supervisor y asesor para evitar que se pierda la imagen del sistema completo.

2.4 Los microservicios en la fase de implementación

La fase de implementación consiste en la creación de un sistema o componente software combinando técnicas de programación, verificación y depuración. Esta fase emplea la salida de la fase de diseño y sirve como entrada para la de pruebas. Los límites entre estas tres fases varían en función del proceso seguido. [22]

2.4.1. Integración de microservicios

La **integración** es la parte más relevante en los sistemas basados en microservicios. Hacerlo correctamente nos asegurará su autonomía y su despliegue de manera independiente.

Por un lado, la tecnología empleada para la comunicación entre servicios no debe restringir la tecnología empleada en estos. Por otro lado, los consumidores deberían de tener total libertad en la tecnología que emplean y consumir un servicio para ellos no debe ser complejo de implementar. Además, los consumidores no deben conocer detalles internos del servicio que consumen para garantizar que estén desacoplados. [17] Existen muchas técnicas para la integración entre las que destacan:

- **RPC** (Remote Procedure Call):la llamada a procedimiento remoto es una técnica que permite ejecutar una llamada a un servicio como si de una llamada local se tratara. No es necesario que cliente y servidor empleen la misma pila tecnológica, aunque algunas tecnologías como Java RMI (Remote Method Invocation) sí lo requieren.

  El formato de los mensajes también varía de una tecnología a otra. SOAP (proviene del inglés Simple Object Access Protocol) emplea XML (también del inglés eXtensible Markup Language) mientras que por ejemplo Java RMI transmite mensajes binarios.

- **REST** (Representational State Transfer): la transferencia de estado representacional es un estilo arquitectónico inspirado en la Web. Se basa en el concepto de recurso, un objeto que el servicio conoce y del que puede crear diferentes representaciones bajo demando. La representación del recurso está completamente desacoplada de como se almacena.

  La arquitectura REST es ampliamente usada junto con HTTP (término que proviene del inglés Hypertext Transfer Protocol). Los verbos que se definen en HTTP actúan sobre recursos: por ejemplo, con el verbo GET se puede obtener una representación del recurso y con el POST crear uno nuevo.

- **Integración basada en eventos**: en la integración basada en eventos, un servicio publica un evento cuando sucede algo relevante. Al evento se suscriben aquellos componentes que deben reaccionar al evento producido. Para hacer llegar los eventos a sus consumidores se debe mantener una nueva infraestructura, como puede ser una cola basada en mensajes.

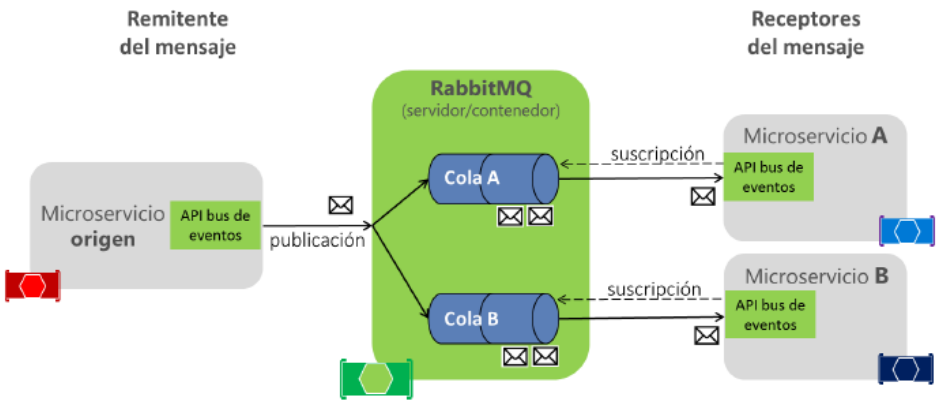

Figura 2.5: Ejemplo de integración basada en eventos con un contenedor de RabbitMQ. [4]

Un bróker de mensajería es un patrón arquitectónico para la validación, la transformación y el enrutamiento de mensajes. RabbitMQ es un ejemplo de este patrón. En esta herramienta, el productor de un evento publica este a través de una API, donde será tramitado por el bróker para hacerlo llegar a todos los consumidores suscritos.

Tanto SOAP como REST emplean HTTP, aunque solo el segundo emplea los verbos definidos en HTTP. Existen pocas librerías para trabajar con SOAP, mientras que prácticamente cualquier lenguaje de programación contemporáneo cuenta con un cliente HTTP, con soporte para más o menos verbos. En cuanto a su rendimiento, REST es más eficiente en términos de latencia y ancho de banda consumido. [16] REST es más recomendable que SOAP para enviar grandes volúmenes de datos por ser más compacto y permitir formatos como el JSON o el binario. Sin embargo, ni uno ni otro son recomendables para trabajar en redes con latencias bajas porque emplean HTTP. En este caso, es mejor emplear directamente protocolos como TCP o UDP.

En cuanto al tercer tipo de integración, la integración basada en eventos fomenta la escalabilidad y resilencia de los servicios y reduce el acoplamiento entre ellos. No obstante, requiere aprovisionar nuevas infraestructuras y añade complejidad a la hora de razonar sobre el sistema por ser una comunicación asíncrona. Algunas buenas prácticas para afrontar su complejidad van desde el uso efectivo de sistemas de monitorización hasta el uso de identificadores de correlación para trazar las llamadas entre servicios. [17]

2.4.2. Programación y persistencia políglotas

Como hemos comentado anteriormente, diferentes microservicios se pueden desarrollar empleando diferentes tecnologías. La base de datos que contiene los datos del servicio o la arquitectura interna del microservicio también pueden adaptarse a los requisitos del mismo. [4] La **programación políglota** se fundamenta en que diferentes lenguajes de programación son más aptos para tratar problemas específicos. Es más productivo escoger el lenguaje adecuado para un servicio concreto que tratar de buscar un lenguaje que se ajuste a los requisitos de todos.

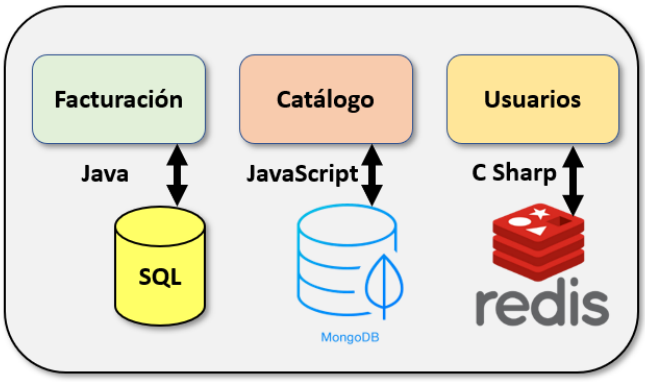

Figura 2.6: Ejemplo de sistema políglota.

El término también se puede extrapolar a la persistencia. La **persistencia políglota** emplea diferentes tecnologías para la persistencia en función de los datos a almacenar y de cómo estos se van a manipular. Cada microservicio es dueño de sus datos y puede emplear una tecnología diferente. Las bases de datos relacionales no son la única opción y se deben considerar otras que escalen mejor si el servicio recibe muchas peticiones o se quiere hacer una explotación eficiente de los datos. [8]

En ambos casos, se debe hacer balance entre los beneficios que puede aportar un diseño políglota y sus costes asociados, por ejemplo, de aprendizaje.

2.4.3. Microservicios en la interfaz de usuario

2.4.4. Ley de Conway

2.5 Los microservicios en la fase de pruebas

Las pruebas de software consisten en la verificación dinámica de que un programa produce las salidas esperadas para un conjunto finito de casos de prueba. Las pruebas son dinámicas porque se verifica el programa en ejecución. Los casos de prueba son finitos porque el número posible de pruebas es infinito y estos se seleccionan en función de la prioridad y riesgo del código bajo pruebas. Además, se debe comprobar la salida obtenida con el resultado esperado para comprobar si esta es o no aceptable. [22]

2.5.1. Pruebas unitarias

Una **prueba unitaria** es una pieza de código que invoca al método o clase bajo prueba y que comprueba ciertas asunciones sobre su lógica. Se ejecuta de forma rápida y sencilla, está automatizado y es fácilmente mantenible. [18] En general, se prefiere tener un gran número de este tipo de pruebas por su rapidez, porque pueden ayudar a la refactorización del código y porque es donde mayor cantidad de defectos se suele capturar.

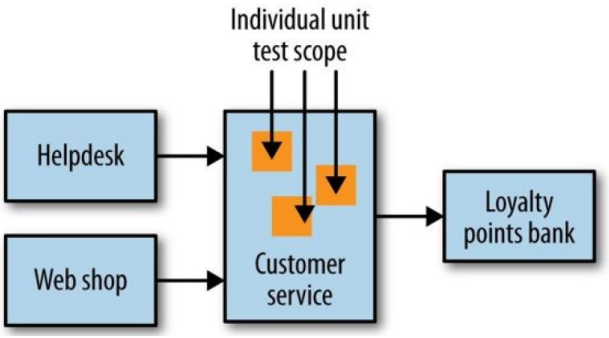

Figura 2.7: Diagrama de pruebas unitarias.

2.5.2. Pruebas de servicios

En las **pruebas de servicios** se verifica cada una de las funcionalidades que un servicio expone. Se pretende verificar el servicio de forma aislada y para ignorar las dependencias que el servicio bajo pruebas tiene sobre otros se reemplazan los servicios colaboradores por fakes.

Encajan dentro de las pruebas de integración, que se definen como la prueba como un conjunto de dos o más módulos de software que colaboran para evaluar un resultado esperado. [18] Este tipo de pruebas pueden ser igual de rápidas que las unitarias siempre que no se tengan que emplear un gran número de infraestructuras como bases de datos o colas.

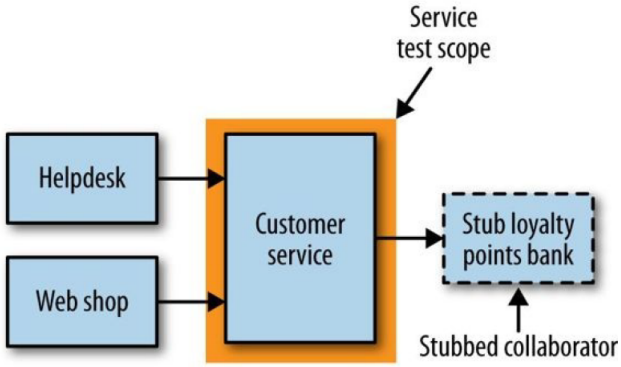

Figura 2.8: Diagrama de pruebas de servicios.

2.5.3. Pruebas de extremo a extremo

Las **pruebas de extremo** a extremo son pruebas que se ejecutan sobre todo el sistema. Cubren gran parte de código, por lo que su correcta ejecución dan mucho grado de confianza. En su ejecución se levantan varios servicios diferentes.

Son pruebas frágiles: conforme el alcance de la prueba aumenta más son las partes sobre las que no se puede tener control. Estas partes pueden introducir fallos que no demuestran que la funcionalidad tenga un defecto y hacen la prueba menos determinista. Si la prueba falla continuamente, el equipo encargado puede llegar a asumir como normal está situación. Su tiempo de ejecución es mayor y en consecuencia, más tarde se detecta si un cambio ha introducido un defecto. [17]

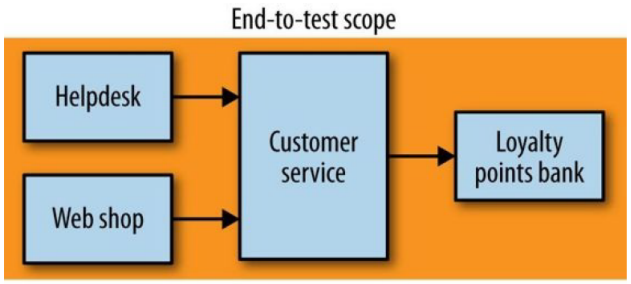

Figura 2.9: Diagrama de pruebas de extremo a extremo.

2.5.4. Balance de pruebas a realizar

A medida que aumenta el alcance de las pruebas lo hace el nivel de confianza que las pruebas dan sobre la ausencia de defectos. Por otro lado, cuanto más arriba en la pirámide más tiempo tardará una prueba en implementarse y ejecutarse. Además, determinar el motivo de fallo de una prueba será más costoso cuanto mayor sean las líneas de código probadas. [3]

El número de pruebas que se aconseja tener de cada tipo aumenta conforme descendemos por la pirámide. El número de servicios que participan para ofrecer una funcionalidad al usuario puede ser muy alto y en una prueba no se deberían de levantar más de 3 o 4 servicios para no potenciar las desventajas que hemos mencionado. Por este motivo, las pruebas de extremo a extremo deben ser las mínimas posibles y se deben refactorizar en pruebas de servicios que empleen fakes siempre que se pueda.

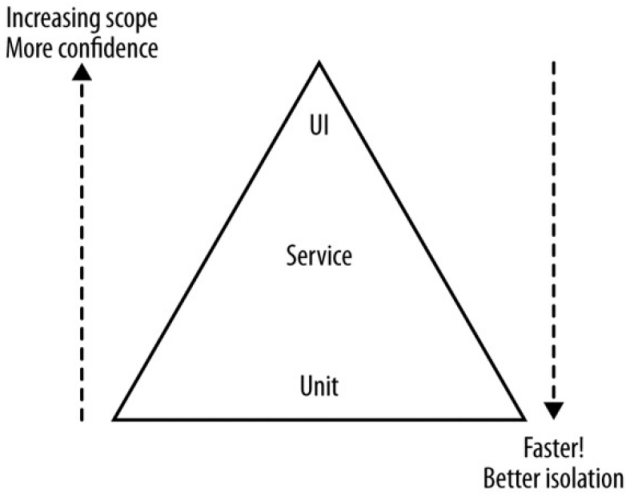

Figura 2.10: Pirámide de pruebas diseñada por Mike Cohn. [3]

2.6 Los microservicios en la fase de despliegue

El despliegue se define como la entrega de software (como un producto completo o como resultado de un incremento en un desarrollo incremental) al cliente para que este lo evalúe y devuelva retroalimentación al equipo de desarrollo. [19] Debido a la naturaleza incremental de la mayoría de procesos de desarrollo, esta es una actividad que se realiza numerosas ocasiones.

En cada despliegue, se debe proveer del soporte necesario para el empleo de las nuevas características. Además, la retroalimentación recibida guiará el proceso de desarrollo hacia las siguientes modificaciones y funcionalidades que se deben realizar.

Los problemas que aparecen en una nueva versión del producto deben ser atendidos. Para garantizar que estos no ocurran, se deben hacer pruebas en cuantos más entornos posibles mejor.

2.6.1. Integración y entrega continua

La **integración continua** es una práctica en el desarrollo de software donde los miembros del equipo integran su trabajo de forma frecuente, normalmente a diario. Cada integración se verifica mediante compilaciones automatizadas que incluyen la ejecución de pruebas para detectar errores y conflictos en la integración. [7] El término tiene su origen como una de las doce prácticas de la metodología Extreme Programming (XP).

Durante este proceso se crean artefactos sobre los que se ejecutarán validaciones. Los artefactos construidos deben ser lo más parecidos a los que más tarde se despliegan en la versión de producción para asegurar que es el mismo artefacto sobre el que se han hecho las pruebas. Entre los beneficios de la integración continua encontramos:

- **Rápida retroalimentación**: los desarrolladores obtienen más rápido una respuesta sobre la calidad de sus cambios. Con este fin, la compilación ha de ser lo más rápido posible. Se pueden obtener mayores beneficios de la integración continua a través de las compilaciones por fases o build pipelines. En este tipo de compilaciones se separa por pasos el proceso. Los pasos de la pipeline pueden ser manuales, como las pruebas de aceptación de un usuario, o estar automatizados, como la ejecución de pruebas unitarias. La ventaja de hacer esto es que se obtiene una retroalimentación

más rápida si se ejecutan de forma separada acciones rápidas de otras más pesadas. [7]

- **Equipo sincronizado**: los integrantes de un equipo obtienen más rápido los cambios que otros han realizado y conocen en todo momento el estado de la compilación. Si esta falla, arreglarla es la prioridad número uno.
- **Trazabilidad**: si el código está bajo control de versiones, en cualquier momento se puede volver a construir un artefacto de una versión concreta a partir del código que la origina. [17]

Figura 2.11: Ejemplo de una pipeline. [17]

La **entrega continua** extiende la idea de la integración continua en cuanto a que cada cambio que ha superado la compilación puede ser candidata para ser desplegada en producción en cualquier momento. Sobre cada cambio se puede decidir si se publica o no a producción y hacerlo no cuesta más que pulsar un botón porque está todo el proceso de despliegue automatizado. La práctica de que cada cambio que se realiza y supera la pipeline es publicado automáticamente a producción se denomina **despliegue continuo**. [9]

2.6.2. Virtualización y tecnología de contenedores

Un **host** representa una unidad genérica de aislamiento, un sistema operativo donde se pueden instalar y ejecutar servicios. Si desplegamos directamente sobre máquinas físicas, entonces un host será el equivalente a una de ellas. Sin embargo, tener muchos hosts es costoso si cada uno consiste en una máquina diferente. [17]

La **virtualización** nos permite dividir una máquina en hosts separados, donde pueden ejecutarse servicios distintos. En la virtualización tradicional, cada una de las máquinas virtuales (MV) puede ejecutar su propio sistema operativo. Un recurso adicional es añadido entre la máquina física y las virtuales: el hipervisor. El hipervisor reparte recursos de la máquina física como la CPU o la RAM entre los distintos hosts virtualizados y permite al usuario la gestión de las máquinas virtuales contenidas.

El mayor inconveniente de las máquinas virtuales es que se puede dividir una máquina en muchas piezas como queramos porque la separación entre los hosts supone un coste. El hipervisor también consume recursos y cuantas más son las máquinas que debe gestionar, mayor será su consumo. Además, el tiempo de puesta en funcionamiento de una máquina virtual es de la magnitud de minutos, mientras que el arranque de un contenedor ronda los pocos segundos. [5] Esto es muy importante en desarrollos que siguen las técnicas de integración continua donde se despliegan artefactos frecuentamente.

Los **contenedores** son sistemas operativos ligeros que se ejecutan sobre una máquina con la que comparten el kernel. El número de contenedores que una máquina puede alojar es mayor que el número de máquinas virtuales debido a la naturaleza ligera de estos. [5] Su uso de recursos es y tiempo de aprovisionamiento son menores al de una MV, lo que reduce el tiempo de retroalimentación sobre el funcionamiento de un servicio. Sin embargo, el grado de aislamiento de los contenedores no es perfecto ya que existen formas en las que se puede interferir en el funcionamiento de otro contenedor debido a problemas de diseño y bugs conocidos. [17]

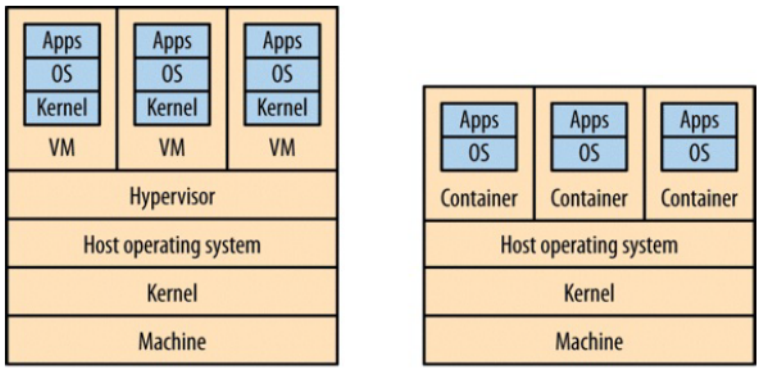

Figura 2.12: Comparación entre la virtualización y la contenerización. [17]

Ambas soluciones se pueden combinar para obtener las ventajas de ambas. Se puede obtener una máquina virtual de una plataforma como Amazon Web Services (AWS) que nos asegure características como la escalabilidad bajo demanda y sobre ella ejecutar diferentes servicios desplegados como contenedores.

2.6.3. Docker

Docker es una herramienta basada en contenedores Linux para la creación de artefactos que se pueden desplegar en cualquier entorno y se pueden distribuir y escalar bajo demanda. [15] Su funcionamiento es sencillo: en un fichero llamado **Dockerfile** se especifican las dependencias del servicio en distintos pasos y un comando que inicia el servicio. De la compilación del Dockerfile se origina una **imagen**, un paquete con todas las dependencias e información necesaria para crear un contenedor. Las imágenes permiten empaquetar un servicio para desplegarlo de forma confiable y reproducible tantas veces como se desee. [4]

La filosofía de Docker se centra en los contenedores desechables. Nada en el entorno donde se ejecuta la aplicación permanecerá ahí más allá de lo que viva la aplicación, por lo que no se delegará en entornos donde se encuentra casualmente una dependencia no especificada en la imagen. Docker promueve las arquitecturas sin estado (stateless) o que externalizan este a otras infraestructuras como bases de datos. Las aplicaciones así se hacen más portables, escalables y confiables. [15]

El proceso de desarrollo también se simplifica: en aproximaciones como la de microservicios, un equipo encargado de un servicio solo necesita la imagen de otro para integrarlo con el suyo y no necesita conocer detalles internos de este. Además, las tareas de construcción de la imagen, provisión de la configuración y despliegue pueden ser realizadas por diferentes personas, como se muestra en la figura.

Si se integra dentro del proceso de despliegue, cada cambio que pasa por una pipeline puede construir una nueva imagen Docker sobre la que se ejecutan pruebas de forma automatizada para después ser publicada a producción. De esta forma el equipo puede asegurarse de que el artefacto sobre el que se han ejecutado las pruebas es el mismo que se publica más tarde.

2.6.4. Kubernetes

Kubernetes es una herramienta diseñada por Google para el despliegue y orquestación de aplicaciones en contenedores Docker de forma resilente. Cada una de las máquinas que aloja uno o más contenedores Docker se denomina **nodo**. La unidad más

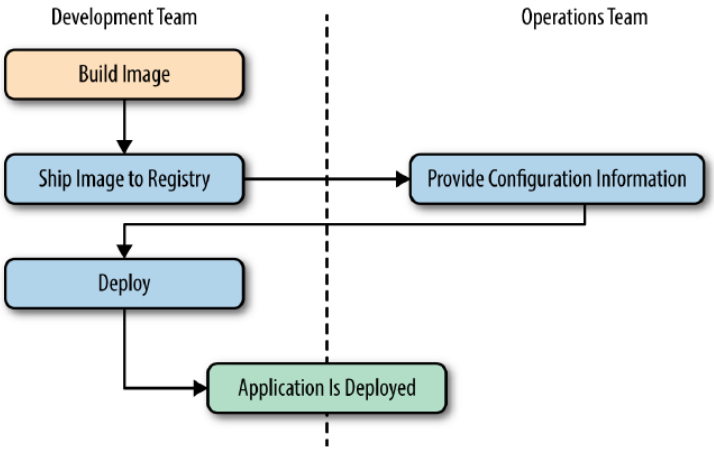

Figura 2.13: Proceso de despliegue con Docker. [15]

pequeña con la que trabaja Kubernetes no son los contenedores, sino los pods. Un **pod** es una colección de contendores y volúmenes agrupados juntos en un nodo. Los **volúmenes** son sistemas de archivos virtuales que se pueden emplear para comunicar entre ellos a los contenedores de un nodo. Kubernetes orquesta a nivel de los pods, por lo que dos contenedores en el mismo pod serán administrados igual. [20]

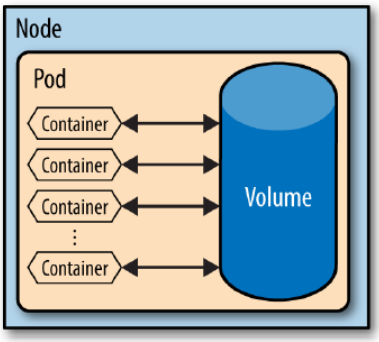

Figura 2.14: Un nodo de Kubernetes. [20]

Kubernetes se emplea principalmente para especificar el número de replicas que se desea tener simultánemanete de un pod. La herramienta es buena para asegurar la disponiblidad de un servicio, pero no garantizan la escalabilidad de este, que consistiría en modificar el número de replicas de un pod de forma dinámica en función de su demanda. Para este propósito se deben introducir balanceadores de carga como puntos de entrada al sistema. Estos balancenadores redirigirían cada petición al pod correspondiente y se encarguen de ajustar el número de replicas de un pod según una serie de reglas.

2.7 Los microservicios en la fase de mantenimiento

Los productos software cambian y evolucionan. Una vez están desplegados en el entorno donde operan se dan situaciones en las que se detectan errores o los requisitos de los usuarios se ven modificados. Es una fase que debe comenzar de forma temprana para garantizar la mantenibilidad del sistema a desarrollar. [22]

El mantenimiento de software es la fase del desarrollo que más recursos consume, alrededor del 60 % o 70 % del coste de un proyecto. La mayoría del software que hoy se emplea tiene entre 10 y 15 años, tiempo en el cual ha sufrido muchas modificaciones hasta

alcanzar un diseño pobre y difícil de mantener. Se puede definir el software mantenible como aquel que se diseña de forma modular, siguiendo patrones de diseño y estándares de calidad y que se documenta de tal forma que es explicativo por si mismo. [19]

Gracias a su diseño modular, los microservicios son una arquitectura que se puede seguir para hacer más sencillo el mantenimiento de un sistema. Sin embargo, las consecuencias de elegir un estilo arquitectónico no son evidente hasta años más tarde de haber sido tomadas, por lo que hasta que no se vean sistemas maduros que empleen microservicios no se debe afirmar a ciencia cierta que garantizan estas características. [13]

2.7.1. Reemplazamiento

En la mayoría de empresas abundan los sistemas legados que nadie desea mantener. La **refactorización** de estos sistemas es imposible por su tamaño y riesgo. Si en lugar de haber empleado una arquitectura monolítica para su diseño se emplearan microservicios se observaría como los barreras para la refactorización no existen al poderse reescribir el microservicio al completo en pocos días.

Una regla que puede ser aplicada es establecer un tamaño para un microservicio tal que pueda ser completamente reescrito en 2 semanas. [17]

2.7.2. You Build It, You Run It

En muchas organizaciones, el desarrollo de sistemas se hace a través de proyectos: piezas de software que una vez desarrolladas se consideran completadas. En una aproximación basada en microservicios, cada equipo es responsable del ciclo de vida completo de un producto o servicio. Esto sigue la filosofía de Amazon: **"you build, you run it"**. [13]

No hay necesidad de distinguir entre quien construye un sistema y quien lo ejecuta y posteriormente mantiene. Esta filosofía aproxima a desarralloradores y clientes: los desarrolladores están en contacto directo con los clientes día tras día, lo que les proporciona retroalimentación para mejorar la calidad de sus servicios. Tampoco hay necesidad de contar en la organización con un equipo centrado en la infraestructura (IT). Contar con un equipo así distribuye la responsabilidad de hacer un servicio funcionar entre diferentes equipos y añade un sobreesfuerzo de coordinación y comunicación cuando un problema aparece. [24]

2.7.3. Documentación

Una de las ventajas de los microservicios es que acelara el proceso de desarrollo. Sin embargo, cuanto más rápido tratamos de implementar un servicio más probable es que tomemos atajos para poder desplegar antes una iteración del producto, lo que se traduce en un aumento de la **deuda técnica**. [11]

En un equipo de desarrollo el conocimiento de cómo funciona un sistema se reparte entre los miembros que lo forman. Nadie es capaz de conocer completamente su funcionamiento y lo más probable es que cada desarrollador conozca mejor la parte donde más tiempo ha invertido. A la hora de realizar un cambio, el entendimiento de cómo funciona el sistema tiene que ser compartido por todos sus miembros para asegurar que el cambio a implementar es el correcto.

La documentación es una de las mejores maneras de solventar la deuda técnica y garantizar que el equipo de desarrollo conoce realmente cómo funciona un microservicio. Cabe recordar que cada microservicio puede seguir una arquitectura diferente o emplear

una tecnología distinta. Debido a esto, se debe documentar de forma exhaustiva para facilitar la integración entre ellos y facilitar el traslado de personas de un equipo a otro.

2.7.4. Monitorización

Durante el mantenimiento se debe asegurar la disponibilidad de los microservicios. Las herramientas de monitorización son clave para garantizar los **acuerdos de nivel de un servicio** (SLA) y el estudio de errores cuando estos ocurren.

Con este propósito, se debe registrar toda aquella información relevante que ocurra. Los microservicios colaboran entre ellos para ofrecer funcionalidades concretas y recrear un sistema donde ha ocurrido un error puede ser complejo debido a que cada uno se versiona de forma independiente. Por ello, lo mejor es contar con toda la información necesaria registrada para determinar la causa del problema.

La información más útil se debe mostrar de forma gráfica a través de dashboards que reflejen el estado de salud de los servicios. Así, la informacón más consultada se puede visualizar de forma rápida y fácil de entender para ahorrar tiempo. No obstante, cuando un problema ocurre se debe hacer uso de alertas para atenderlo de forma prioritaria. Estas alertas han de ser accionadas automáticamente por métricas que superen un límite establecido, como puede ser el uso de recursos, o por la ocurrencia de excepeciones en el servicio. [11]

2.8 Comparación con otras arquitecturas

2.9 Crítica al estado del arte

2.10 Propuesta

CAPÍTULO 3
Análisis del sistema

3.1 Descripción del sistema

Se desea desarrollar un sistema para la venta de productos electrónicos a través de dispositivos móviles. La aplicación móvil estará destinada solo a los clientes de una tienda, que deberán registrarse para usarla.

Los clientes podrán consultar los **productos** disponibles en la tienda junto con su precio, descripción y número de unidades en stock. Un **pedido** se define como un conjunto de productos que el cliente ha seleccionado junto con el número de unidades de cada uno. Mientras no se haya confirmado el pedido, se podrán añadir y eliminar productos al pedido en elaboración.

Todos los pedidos realizados por un cliente se mostrarán en un listado junto con el estado de los mismos. Para cada pedido, el cliente podrá generar una **factura** donde se listen los productos que lo componen y el precio total.

Una vez un cliente haya confirmado un pedido, este deberá ser aprobado por un empleado para ser entregado. Cuando esto suceda, se enviará una **notificación** al correo electrónico del cliente. Mientras que el empleado no lo apruebe, el pedido podrá ser cancelado. Además, cuando el pedido sea entregado físicamente al cliente, este deberá marcar el pedido como entregado.

Un cliente puede crear una **incidencia** si tiene algún problema con un pedido o quiere consultar una duda con los empleados de la tienda. Dentro de una incidencia, tanto el cliente como los empleados se comunicarán a través de **comentarios**. Cuando lo considere oportuno, el cliente podrá cerrar la incidencia creada, que podrá volver a consultar en cualquier momento.

3.2 Casos de uso y modelo de dominio

A continuación, vamos a listar los casos de uso de la aplicación móvil. En los casos de uso aumentaremos el nivel de detalle de la descripción para emplearlos como especificación del sistema. El único usuario de la aplicación móvil es el cliente de la tienda, por lo que es el actor de todos los casos citados. Para cada caso de uso se va a proveer un identificador, un título y una descripción breve:

CU001	Iniciar sesión en la aplicación
Para iniciar sesión, el usuario debe haberse registrado previamente en la aplicación. Para hacerlo debe proveer simplemente su correo electrónico y su contraseña.	

CU002	Registrarse en la aplicación
Cualquier persona que tenga la aplicación instalada en su dispositivo móvil puede registrarse como cliente. Para hacerlo, simplemente ha de proveer su nombre, su correo electrónico y una contraseña.	

CU003	Listar los productos de la tienda
El cliente puede consultar todos los productos disponibles en la tienda. Para cada producto se mostrará una imagen, el nombre del producto y su precio. El usuario podrá buscar un producto por su nombre y ordenar el listado por nombre y precio.	

CU004	Visualizar un producto
El cliente puede ver la descripción de un producto, su precio y número de unidades en stock. Cuando lo desee, el usuario puede seleccionarlo para comprarlo y crear así un nuevo pedido.	

CU005	Crear un pedido
El cliente puede crear un pedido y añadir en él productos. Para cada uno, debe especificar el número de unidades que desea. En cualquier momento puede eliminar cualquiera de los productos que componen el pedido. Cuando lo considere oportuno, el cliente ha de confirmar el pedido que está creando para que sea tramitado.	

CU006	Cancelar un pedido
Para los pedidos confirmados, mientras estos no hayan sido gestionados por un empleado para su entrega, el pedido se podrá cancelar.	

CU007	Generar factura de un pedido
El cliente podrá generar una factura para cualquiera de sus pedidos. La factura se generará en formato PDF e incluirá la siguiente información: los datos del cliente, el listado de todos los productos que componen el pedido junto con su precio unitario y el precio total del pedido antes y después de aplicar el IVA.	

CU008	Marcar un pedido como recibido
Cuando un pedido se entregue físicamente a un cliente, este deberá marcar a través de la aplicación el pedido como recibido.	

CU009	Listar los pedidos del cliente
Todos los pedidos que ha realizado un cliente se podrán listar en cualquier momento, mostrando la fecha en que se realizó y su estado actual. Los pedidos aparecen ordenados por la fecha en la que se realizaron. El listado se puede filtrar por el estado de los pedidos. Los posibles estados de un pedido son: CONFIRMADO, PROGRAMADO, RECIBIDO y CANCELADO.	

CU010	Crear una incidencia
Un cliente puede crear una incidencia si tiene algún problema con un pedido o quiere consultar una duda con los empleados de la tienda. Para hacerlo, ha de proveer un título que describa el problema o consulta.	

CU011	Añadir un comentario en una incidencia
En el contexto de una incidencia, los clientes podrán comunicarse a través de comentarios con los empleados de la tienda. En la incidencia se verán los comentarios tanto de los empleados como del cliente ordenados cronológicamente. Para cada comentario se mostrará el nombre del autor, la fecha en la que lo escribió y el contenido del mismo.	

CU012	Cerrar una incidencia
Solo el cliente puede cerrar una incidencia que haya abierto. Cuando lo haga, el cliente podrá seguir accediendo a ella pero ya no podrá añadir nuevos comentarios. Tampoco le estará permitido volver a abrirla.	

CU013	Listar las incidencias del cliente
Se pueden consultar todas las incidencias creadas por el cliente en un listado donde se mostrará tanto la fecha en la que se crearon como su estado actual. El listado aparecerá ordenado por la fecha de creación de las incidencias y se podrá filtrar por el estado de las mismas. Los posibles estados de las incidencias son: ABIERTA y CERRADA.	

A partir de la descripción del sistema y de los casos de uso podemos definir el siguiente modelo de dominio:

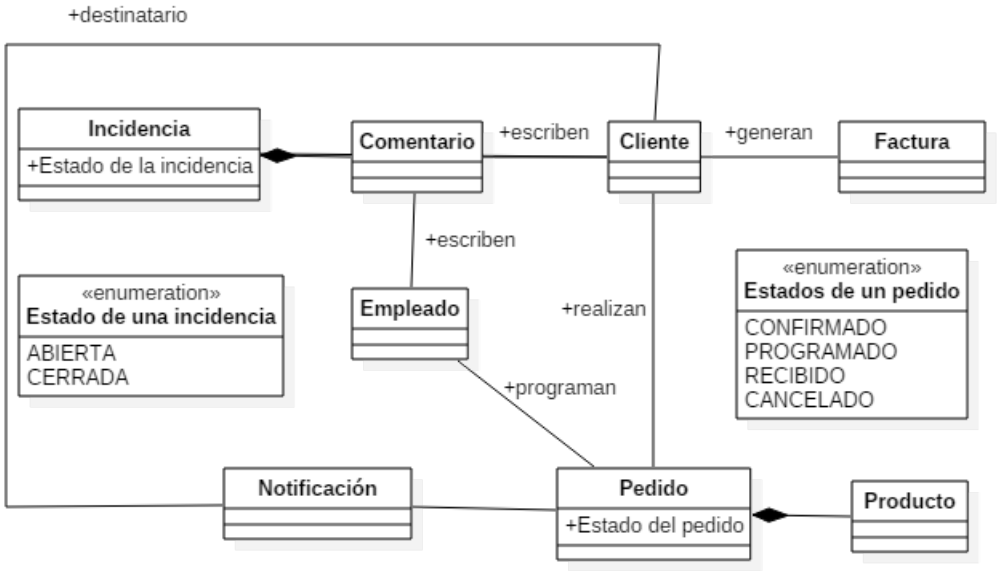

Figura 3.1: Modelo de dominio del sistema.

3.3 Plan de trabajo

El sistema a implementar se puede dividir en dos partes: front-end y back-end. En lo referente al back-end, este se va a implementar siguiendo dos arquitecturas distintas: una monolítica y otra basada en microservicios. Existirá una gran cantidad de código compartido entre ambas soluciones y las mayores diferencias entre ellas serán las relacionadas con la organización del código. Por este motivo, primero se va a implementar la solución monolítica y una vez implementada se creará una nueva solución donde se refactorizará el código siguiendo los principios que en el apartado de Diseño del Estado del Arte se han mencionado.

En cuanto al front-end, las llamadas a la parte servidora van a realizarse a través de invocaciones HTTP. Por tanto, va a estar desacoplado de la implementación que elijamos para el back-end y va a poder emplearse la misma solución para comunicarse con ambas soluciones.

3.4 Tecnología utilizada

El lenguaje de programación que se va a emplear es C# junto con el entorno de desarrollo (IDE) Visual Studio Enterprise 2017. Entre las plataformas de destino que ofrece la tecnología .NET vamos a utilizar tanto .NET Standard 2.0 como .NET Core en su versión 2.1, publicada en Mayo de 2018. [1] El uso de librerías distintas a las provistas por la plataforma se realizará a través de paquetes NuGet, un mecanismo sencillo que envuelve el código compilado de una librería y referencia a las dependencias de esta. [2]

A nivel de infraestructura se ha empleado Microsoft Azure para la persistencia de datos en la nube y para la exposición de servicios a través de Azure App Service y Azure Kubernetes Service (AKS). Microsoft Azure es una plataforma que ofrece un conjunto de herramientas y servicios en la nube. Tanto App Service como AKS son parte de sus servicios. El primero se emplea para crear aplicaciones en la nube de forma rápida sin necesidad de administrar la infraestructura sobre la que se hospeda. En cuanto al segundo, permite la orquestación de contenedores Docker a través de Kubernetes dentro de la infraestructura de Azure. [3]

Para terminar, para las pruebas a nivel de API se ha empleado Postman. Postman es un entorno para el desarrollo de APIs (ADE) que nos va a permitir crear y almacenar de forma sencilla llamadas HTTP hacia la API del back-end. [4]

[1] Versiones de .NET Core 2.1: https://www.microsoft.com/net/download/dotnet-core/2.1
[2] Una introducción a NuGet: https://docs.microsoft.com/es-es/nuget/what-is-nuget
[3] Página oficial de Microsoft Azure: https://azure.microsoft.com/es-es/
[4] Página oficial de Postman: https://www.getpostman.com/

CAPÍTULO 4
Diseño e implementación de la solución monolítica

4.1 Diseño de la solución

A nivel arquitectónico, la aplicación monolítica va a seguir una arquitectura de 6 capas, que detallamos a continuación:

- **Capa de contratos**: en esta capa se situarán las interfaces que contienen todas las acciones del back-end que pueden ser invocadas desde el exterior a través de la API. Estas interfaces se implementarán tanto en las capas de Aplicación, Servicios y Proxy.

  También se situarán en esta capa los objetos para la transferencia de datos (DTO). Los DTOs se emplean para desacoplar los servicios que ofrece una API de la representación interna que da el sistema a sus entidades. En lugar de devolver al cliente una entidad tal y como se almacena en base de datos, los DTOs representan una entidad ocultando aquellas propiedades que no necesitan ser transferidas o cambian su formato para que sea más cómodo para el cliente su procesamiento. [1]

- **Capa de aplicación**: en esta capa es donde se implementa la lógica de la parte servidora. Aquí se da implementación a las interfaces que se sitúan en la capa de contratos. Es en esta capa donde se validan los permisos de un usuario que ha realizado una petición al servidor sobre la acción que desea realizar. En caso de que se trate de hacer una operación no válida será esta capa la encarga de lanzar la excepción oportuna. También aquí se situarán los conversores para transformar una entididad en un DTO, que se emplearán principalmente en las operaciones CRUD.

- **Capa de servicios**: contiene el punto de entrada del proceso que representa al back-end (el método Main). En esta capa se encuentran los controladores, donde se definen todas las acciones de la API. Cada acción se define a través de un verbo HTTP, la URL donde se localiza y sus parámetros obtenidos a partir del cuerpo o las cabeceras de una petición. La capa de servicios delega en la capa de aplicación para devolver un resultado a cada una de las peticiones que atiende. [2]

[1] Crear objetos de transferencia de datos (DTO): https://docs.microsoft.com/es-es/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5

[2] Control de solicitudes con controladores en ASP.NET Core MVC: https://docs.microsoft.com/es-ES/aspnet/core/mvc/controllers/actions?view=aspnetcore-2.1

- **Capa de dominio**: contiene las entidades del dominio del sistema junto con sus propiedades y los estados asociados a estas. Esta capa es referenciada tanto por la capa de aplicación como por la capa de persistencia.
- **Capa de persistencia**: la capa de persistencia ofrece operaciones CRUD para cada una de las entidades que se almacenan en BD a través objetos para el acceso a datos (DAO). La capa de persistencia no trabaja con DTOs: cuando la capa de aplicación le solicita una entidad la devuelve completa y es la capa de aplicación quien a través de los conversores la transforma en un DTO para devolver una representación de la entidad al usuario.
- **Capa de proxy**: esta capa contiene clientes necesarios para invocar al back-end a través de llamadas HTTP realizadas por código C#. Esta capa será referenciada por el front-end a través de un paquete NuGet para comunicar con el back-end. El front-end es quien ejecutará el código contenido en esta capa: cuando se desee comunicar con la parte servidor, en el proceso del front-end se invocará al proxy para realizar una llamada HTTP al back-end. Como debe ofrecer todas las operaciones de la API, implementa las interfaces de la capa de contratos.

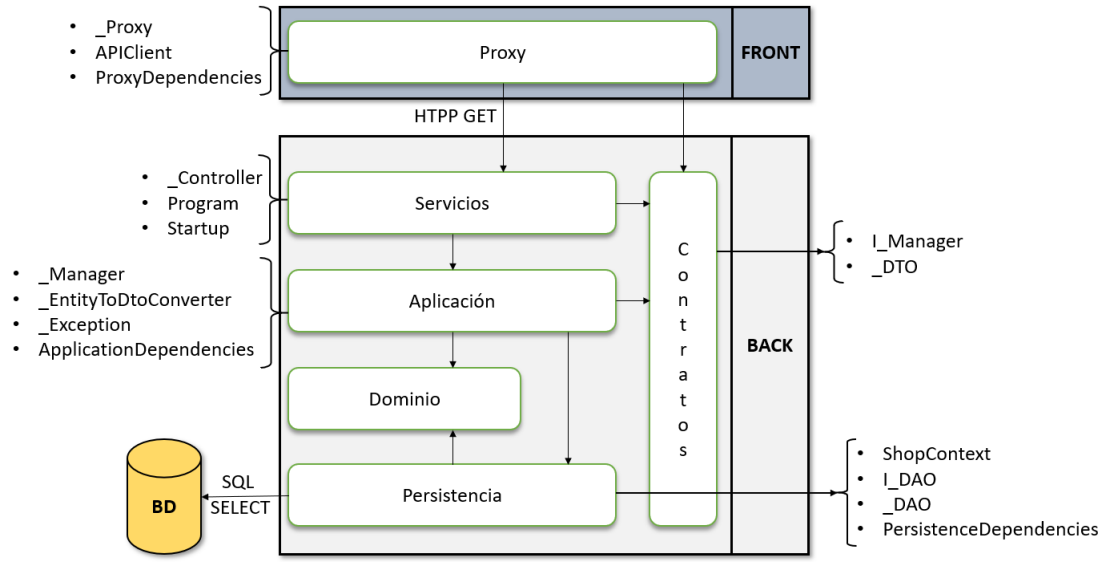

Figura 4.1: Capas del back-end monolítico.

A nivel de la solución en Visual Studio, cada capa consistirá en un proyecto distinto. Un proyecto consiste en un archivo XML con extensión *.csproj que tiene una plataforma y dependencias específicas y que se puede compilar de forma independiente. Cada proyecto es un contenedor que organiza sus clases y otros archivos de forma jerárquica. [3] Adicionalmente, existe un proyecto que contiene las pruebas del sistema. Todos los proyectos son .NET Standard 2.0 salvo los de las capa de servicios y pruebas, cuya plataforma es .NET Core 2.1 porque no son librerías y pueden ser ejecutados.

A nivel de diseño, se van a definir las siguientes entidades en la capa de dominio: pedido (Order), producto (Product), compra (Purchase), incidencia (Incidence) y comentario (Comment). Estas entidades se pueden obtener a partir del modelo de dominio. Sin embargo, algunos conceptos que aparecen en el modelo de dominio como los informes o las notificaciones no vamos a modelarlas como entidades. No son entidades como tal

[3] Soluciones y proyectos en Visual Studio: https://docs.microsoft.com/es-es/visualstudio/ide/solutions-and-projects-in-visual-studio

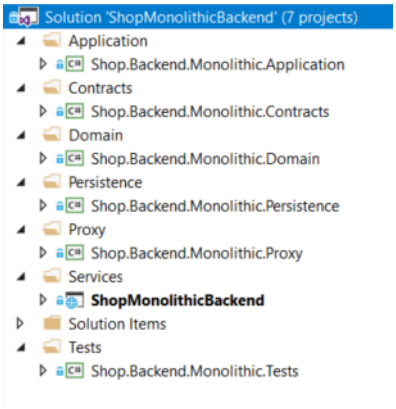

Figura 4.2: Solución del sistema monolítico.

porque no necesitamos almacenarlas o ofrecer operaciones CRUD sobre ellas, así que vamos a modelarlas como acciones. Simplemente ofreceremos métodos para generar un informe (GenerateReport) y enviar notificaciones (SendNotification).

La relación que existe entre un pedido y un producto es de muchos a muchos: un pedido está formado por múltiples productos y un producto puede estar incluido en diferentes pedidos. Además, la relación cuenta con una propiedad que es el número de unidades del producto en el pedido, que se puede modelar como una clase asociación. Si razonamos a nivel de base de datos, la clase asociación se implementará como una tabla intermedia entre las tablas de pedidos y productos. En nuestro código también lo modelaremos así para emplear la librería de Entity Framework para el mapeo objeto-relacional (ORM).

Con todo esto, el diagrama de clases que se encuentran en la capa de dominio será el siguiente:

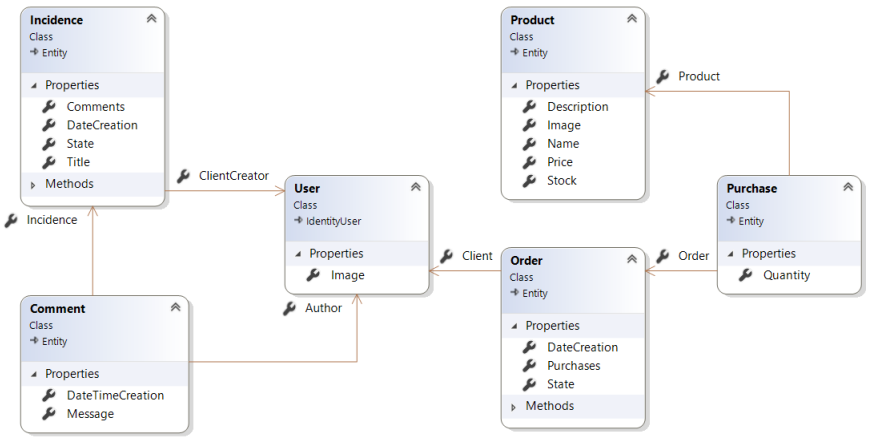

Figura 4.3: Diagrama de clases de dominio de la solución monolítica.

4.2 Detalles de la implementación back-end

4.2.1. Operaciones CRUD

Para la mayoría de entidades que hemos modelado vamos a ofrecer operaciones CRUD. Esto se hará a través de las interfaces que expone la parte servidora en la capa de contratos. Con este propósito vamos a definir una interfaz que exponga estas de for-

ma unitaria y agregada. En los servicios, las operaciones agregadas son un mecanismo para evitar que dos componentes tengan que comunicarse continuamente. Si se tienen que crear una colección de entidades, en lugar de crear una petición al servidor para cada entidad, se envía una única petición con toda la colección. [17]

```
/// <summary> The internal interface for the managers that offer CRUD operations ...
3 references | VictorIranzo, 17 days ago | 1 author, 4 changes
internal interface ICrudManager<TEntityDTO, TCreateDTO, TUpdateDTO>
    where TEntityDTO : EntityBaseDTO
    where TCreateDTO : CreateBaseDTO
    where TUpdateDTO : UpdateBaseDTO
{
    /// <summary> Creates the specified entity in the database.
    4 references | VictorIranzo, 17 days ago | 1 author, 4 changes | 0 exceptions
    Task<Guid> CreateAsync(TCreateDTO createDTO);

    /// <summary> Creates the specified entities in the database.
    4 references | VictorIranzo, 21 days ago | 1 author, 3 changes | 0 exceptions
    Task<bool> CreateAsync(IList<TCreateDTO> createDTOs);

    /// <summary> Gets the entity with the provided identifier.
    7 references | VictorIranzo, 21 days ago | 1 author, 3 changes | 0 exceptions
    Task<TEntityDTO> GetAsync(Guid entityId);

    /// <summary> Updates the entity in the database.
    5 references | VictorIranzo, 21 days ago | 1 author, 3 changes | 0 exceptions
    Task<bool> UpdateAsync(TUpdateDTO updateDTO);

    /// <summary> Updates the entities in the database.
    4 references | VictorIranzo, 21 days ago | 1 author, 3 changes | 0 exceptions
    Task<bool> UpdateAsync(IList<TUpdateDTO> updateDTOs);
}
```

Figura 4.4: Interfaz interna para las operaciones CRUD.

La interfaz es interna porque no debe ser visible fuera de la solución del back-end. Además, es genérica: se define en base a DTOs con diferentes propósitos. Para la creación de una entidad son necesarias solo algunas propiedades y otras como la fecha de creación son calculadas por el sistema y no deben ser provistas en el **CreateDTO** de la petición. Lo mismo ocurre con la lectura y la actualización de una entidad. Por ejemplo, de una entidad algunas propiedades no está permitido actualizarlas, por lo que no deben incluirse estas en el **UpdateDTO** de la entidad.

Sin embargo, no se van a definir únicamente operaciones CRUD para una entidad. Cada entidad tiene una serie de operaciones asociadas, como puede ser generar la factura de un pedido o obtener la lista de incidencias de un usuario. Tanto estas acciones como las CRUD se definen en la interfaz de contratos de la entidad. Si para una entidad se quieren exponer operaciones CRUD, se deben definir los DTOs específicos para las operaciones de lectura, escritura y actualización y se debe extender la interfaz genérica.

```
/// <summary> Contracts interface for managing the orders.
8 references | VictorIranzo, 15 days ago | 1 author, 7 changes
public interface IOrdersManager : ICrudManager<OrderDTO, OrderCreateDTO, OrderUpdateDTO>
{
    /// <summary> Adds the purchases and confirms the order.
    4 references | VictorIranzo, 17 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> AddPurchasesAndConfirmOrder(AddPurchasesDTO addPurchasesDTO);

    /// <summary> Generates the report of and order.
    4 references | VictorIranzo, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<byte[]> GenerateOrderReportAsync(Guid orderId, bool pdfFormat = true);

    /// <summary> Gets the list of user orders filtered.
    4 references | VictorIranzo, 16 days ago | 1 author, 1 change | 0 exceptions
    Task<IEnumerable<OrderDTO>> GetListOfUserOrdersFilteredAsync(
        Guid userId,
        string state = null,
        int fromElement = 0,
        int numberOfElements = 10);
}
```

Figura 4.5: Interfaz en la capa de contratos para la entidad Pedido.

La implementación de las operaciones CRUD en la capa de aplicación también se realizará de forma genérica. Si la implementación de una de las operaciones no se ajusta a la que una entidad necesita, esta puede ser sobreescrita en el manager de la entidad.

Por último, algunas entidades en lugar de ofrecer operaciones CRUD solo ofrecen **operaciones CUD** (crear, actualizar y eliminar). Estas entidades son las de Comentario y Compra porque podemos considerarlas como secundarias. No tiene sentido exponer un método en la interfaz de back-end para leer un único comentario o obtener el número de unidades de un producto en un pedido. No obstante, si que tiene sentido exponer un método para, por ejemplo, crear o eliminar un comentario dentro de una incidencia. Las operaciones de lectura de estas entidades se realizan a través de la entidad principal asociada. Por ejemplo, cuando solicitamos una incidencia obtendremos todos los comentarios que en la incidencia se han hecho aunque sea una entidad independiente.

4.2.2. Seguridad

4.2.3. Persistencia

4.2.4. Informes empleando la librería Open XML PowerTools

4.2.5. Notificaciones con la librería MailKit

4.2.6. Inyección de dependencias

El uso de interfaces promueve el desacoplamiento entre la definición de las operaciones de una clase y las implementaciones concretas que esta definición puede tener.

4.2.7. Documentando la API con Swagger

4.2.8. Calidad del código

4.3 Interfaz de usuario

4.4 Pruebas

CAPÍTULO 5

Diseño e implementación de la solución basada en microservicios

CAPÍTULO 6
Evaluación de las soluciones

6.1 Mantenimiento

6.1.1. Mantenimiento correctivo

6.1.2. Añadir una relación entre diferentes servicios

6.1.3. Migración de la infraestructura

6.2 Escalabilidad

6.3 Tolerancia a fallos

CAPÍTULO 7

Conclusiones

Bibliografía

[1] David Ameller, Claudia Ayala, Jordi Cabot, and Xavier Franch. Non-functional Requirements in Architectural Decision Making. *IEEE SOFTWARE*, 2013.

[2] Ali Arsanjani, Grady Booch, Toufic Boubez, Paul C. Brown, David Chappell, John DeVadoss, Thomas Erl, Nicolai Josuttis, Dirk Krafzig, Mark Little, Brian Loesgen, Anne Thomas Manes, Joe McKendrick, Steve Ross-Talbot, Stefan Tilkov, Clemens Utschig-Utschig, and Herbjörn Wilhelmsen. SOA Manifesto. *SOAManifesto*, 2009.

[3] Mike Cohn. *Succeeding with agile : software development using Scrum*. Addison-Wesley, 2010.

[4] Cesar De la Torre, Bill Wagner, and Mike Rousos. *Microservicios .NET: Arquitectura para Aplicaciones .NET Contenerizadas*. Microsoft Corporation, 2 edition, 2018.

[5] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support PaaS. *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pages 610–614, 2014.

[6] João M. Fernandes and Ricardo J. Machado. *Requirements in Engineeng Projects*. Springer, 2016.

[7] Martin Fowler. Continuous Integration, 2006.

[8] Martin Fowler. PolyglotPersistence, 2011.

[9] Martin Fowler. ContinuousDelivery, 2013.

[10] Martin Fowler. BoundedContext, 2014.

[11] Susan J. Fowler. *Microservices in Production : Standard Principles and Requirements*. O'REILLY, 2017.

[12] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, feb 2012.

[13] James Lewis and Martin Fowler. Microservices, 2014.

[14] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[15] Karl Matthias and Sean P Kane. *Docker Up & Running*. O'REILLY, 2015.

[16] Gavin Mulligan and Denis Gračanin. A COMPARISON OF SOAP AND REST IMPLEMENTATIONS OF A SERVICE BASED INTERACTION INDEPENDENCE MIDDLEWARE FRAMEWORK. Technical report, 2009.

[17] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[18] Roy. Osherove. *The art of unit testing : with examples in C Sharp*. Manning Publications, 2014.

[19] Roger S. Pressman. *Software engineering : a practitioner's approach*.

[20] David Rensin. *Kubernetes*. O'REILLY, 2015.

[21] Mark Richards. *Microservices AntiPatterns and Pitfalls*. O'REILLY, 2016.

[22] IEEE COMPUTER SOCIETY. *Guide to the Software Engineering - Body of Knowledge.* 2014.

[23] International Standard. ISO 25010. 2010, 2010.

[24] Jurg van. Vliet and Flavia. Paganelli. *Programming Amazon EC2*. O'Reilly Media, 2011.

[25] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

[26] Wikipedia. Service (systems architecture).

APÉNDICE A

Ventajas e inconvenientes del desarrollo basado en microservicios

APÉNDICE B

Modelo de navegación de la aplicación móvil

APÉNDICE C

Despliegue del sistema basado en microservicios

C.1 Ejecución de un contenedor con Docker

C.1.1. Archivo Docker Compose

C.2 Introducción a Azure Kubernetes Service (AKS)

C.3 Depuración de contenedores a través de Azure

APÉNDICE D

Creación de clientes con NSwag