



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



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 general | V |
| Índice de figuras | IX |
| 1 Introducción | 1 |
| 1.1 Motivación | 1 |
| 1.2 Objetivos | 1 |
| 1.3 Estructura de la memoria | 2 |
| 2 Los microservicios en el proceso de desarrollo | 3 |
| 2.1 ¿Qué son los microservicios? | 3 |
| 2.2 Los microservicios en la fase de requisitos | 4 |
| 2.2.1 Requisitos funcionales y no funcionales | 4 |
| 2.2.2 El teorema de CAP | 6 |
| 2.3 Los microservicios en la fase de diseño | 6 |
| 2.3.1 Librerías versus servicios | 6 |
| 2.3.2 Diseño guiado por el dominio (DDD) | 7 |
| 2.3.3 Descomposición en microservicios | 8 |
| 2.3.4 La tarea del arquitecto de software | 9 |
| 2.4 Los microservicios en la fase de implementación | 9 |
| 2.4.1 Integración de microservicios | 9 |
| 2.4.2 Programación y persistencia políglotas | 11 |
| 2.4.3 La ley de Conway | 11 |
| 2.5 Los microservicios en la fase de pruebas | 12 |
| 2.5.1 Pruebas unitarias | 12 |
| 2.5.2 Pruebas de servicios | 12 |
| 2.5.3 Pruebas de extremo a extremo | 12 |
| 2.5.4 Balance de pruebas a realizar | 13 |
| 2.6 Los microservicios en la fase de despliegue | 14 |
| 2.6.1 Integración y entrega continua | 14 |
| 2.6.2 Virtualización y tecnología de contenedores | 15 |
| 2.7 Los microservicios en la fase de mantenimiento | 15 |
| 2.7.1 Reemplazamiento | 16 |
| 2.7.2 You Build It, You Run It | 16 |
| 2.7.3 Documentación | 17 |
| 2.7.4 Monitorización | 17 |
| 3 Estado del arte de la tecnología de microservicios | 19 |
| 3.1 Contenedores | 19 |
| 3.1.1 Contenedores Linux | 19 |
| 3.1.2 Docker | 19 |
| 3.2 Orquestadores | 20 |
| 3.2.1 Kubernetes | 20 |
| 3.2.2 Docker Swarm | 21 |
| 3.3 Proveedores de servicios en la nube | 21 |
| 3.3.1 Amazon Web Services (AWS) | 21 |

| | | |
|----------|--|-----------|
| 3.3.2 | Microsoft Azure | 21 |
| 3.4 | Crítica al estado del arte | 22 |
| 3.5 | Propuesta | 22 |
| 4 | Especificación de requisitos del caso de estudio | 23 |
| 4.1 | Descripción del caso de estudio | 23 |
| 4.2 | Casos de uso y modelo de dominio | 23 |
| 5 | Proceso de desarrollo | 27 |
| 5.1 | Plan de trabajo | 27 |
| 5.2 | Organización del trabajo | 27 |
| 6 | Diseño e implementación de la solución monolítica | 31 |
| 6.1 | Diseño de la solución | 31 |
| 6.2 | Detalles de la implementación back-end | 34 |
| 6.2.1 | Operaciones CRUD | 34 |
| 6.2.2 | Seguridad | 35 |
| 6.2.3 | Persistencia | 38 |
| 6.2.4 | Informes empleando la librería Open XML PowerTools | 39 |
| 6.2.5 | Notificaciones con la librería MailKit | 40 |
| 6.2.6 | Inyección de dependencias | 41 |
| 6.2.7 | Documentando la API con Swagger UI | 42 |
| 6.2.8 | Logging | 43 |
| 6.2.9 | Generación de la capa de proxy | 43 |
| 6.2.10 | Calidad del código | 44 |
| 6.3 | Interfaz de usuario | 45 |
| 6.4 | Pruebas | 47 |
| 6.5 | Despliegue | 48 |
| 7 | Diseño e implementación de la solución basada en microservicios | 51 |
| 7.1 | Diseño de la solución | 51 |
| 7.1.1 | Arquitectura interna de los microservicios | 52 |
| 7.1.2 | Organización de los microservicios | 53 |
| 7.2 | Diferencias en la implementación respecto a la solución monolítica | 55 |
| 7.2.1 | Consumo de otros microservicios | 55 |
| 7.2.2 | Consistencia eventual | 55 |
| 7.2.3 | Microservicio de notificaciones | 57 |
| 7.2.4 | Persistencia en microservicio de incidencias | 59 |
| 7.3 | Versionado de servicios | 60 |
| 7.4 | Adaptación de la interfaz de usuario | 62 |
| 7.5 | Adaptación de las pruebas | 62 |
| 7.6 | Despliegue | 64 |
| 8 | Evaluación de las soluciones | 65 |
| 8.1 | Mantenimiento | 65 |
| 8.1.1 | Mantenimiento correctivo | 65 |
| 8.1.2 | Mantenimiento perfectivo | 66 |
| 8.1.3 | Mantenimiento adaptativo | 66 |
| 8.2 | Comparación de las soluciones ante RNFs | 67 |
| 8.2.1 | Disponibilidad | 67 |
| 8.2.2 | Tolerancia a fallos | 67 |
| 8.2.3 | Utilización de recursos | 68 |
| 8.2.4 | Capacidad de ser reemplazado | 69 |
| 8.3 | Otras posibles soluciones | 70 |
| 8.4 | Ventajas e inconvenientes de los microservicios | 71 |
| 9 | Conclusiones | 73 |

| | |
|---------------------|-----------|
| Bibliografía | 75 |
|---------------------|-----------|

Apéndices

| | |
|---|-----------|
| A Descripción del prototipo de IU desarrollado | 77 |
| B Despliegue del sistema basado en microservicios | 79 |
| B.1 Ejecución de un contenedor con Docker | 79 |
| B.1.1 Archivo Docker Compose | 79 |
| B.2 Introducción a Azure Kubernetes Service (AKS) | 79 |
| B.3 Depuración de contenedores a través de Azure | 79 |

Índice de figuras

| | | |
|------|--|----|
| 2.1 | Los microservicios escalan de acuerdo a su carga de trabajo para asegurar la disponibilidad de la funcionalidad que ofrecen. [17] | 4 |
| 2.2 | Características y subcaracterísticas definidas en el modelo de calidad del producto de la ISO/IEC 25010. [29] | 5 |
| 2.3 | Ejemplo de dos contextos delimitados dentro del mismo dominio que emplean el mismo nombre para un concepto, pero con significados diferentes. [14] | 7 |
| 2.4 | Primero, se dividen los grandes servicios en microservicios. Una vez hecho esto, se migran sus datos. [26] | 8 |
| 2.5 | Ejemplo de integración basada en eventos con un contenedor de RabbitMQ. [7] | 10 |
| 2.6 | Ejemplo de sistema políglota. | 11 |
| 2.7 | Diagrama de pruebas unitarias. | 12 |
| 2.8 | Diagrama de pruebas de servicios. | 13 |
| 2.9 | Diagrama de pruebas de extremo a extremo. | 13 |
| 2.10 | Pirámide de pruebas diseñada por Mike Cohn. [5] | 13 |
| 2.11 | Ejemplo de una pipeline. [21] | 15 |
| 2.12 | Comparación entre la virtualización y la contenerización. [21] | 16 |
| 3.1 | Proceso de despliegue con Docker. [19] | 20 |
| 3.2 | Un nodo de Kubernetes. [25] | 21 |
| 4.1 | Modelo de dominio del sistema. | 25 |
| 5.1 | Cronograma del proceso de desarrollo del caso de estudio. | 28 |
| 5.2 | Proyecto de GitHub asociado al desarrollo del front-end. | 28 |
| 5.3 | Número de commits realizados cada día en los repositorios del caso de estudio. | 29 |
| 6.1 | Capas del back-end monolítico. | 32 |
| 6.2 | Solución del sistema monolítico. | 33 |
| 6.3 | Diagrama de clases de dominio de la solución monolítica. | 33 |
| 6.4 | Interfaz interna para las operaciones CRUD. | 34 |
| 6.5 | Interfaz en la capa de contratos asociada a la entidad Pedido. | 35 |
| 6.6 | La clase User. | 35 |
| 6.7 | Tablas creadas por ASP.NET Core Identity. | 36 |
| 6.8 | Ejemplo de métodos de la clase de servicios. | 36 |
| 6.9 | Pieza de código para añadir la autenticación usando tokens JWT. | 37 |
| 6.10 | Ejemplo de petición HTTP donde se incluye un token de seguridad. | 37 |
| 6.11 | Creación de una base de datos SQL en Azure. | 38 |
| 6.12 | Configuración del contexto de la capa de persistencia para apuntar a la BD en Azure. | 38 |
| 6.13 | Clase ShopContext. | 39 |
| 6.14 | Interfaz genérica para los DAOs que exponen operaciones CRUD. | 39 |

| | | |
|------|---|----|
| 6.15 | Plantilla para la creación de facturas. | 40 |
| 6.16 | Archivo de recursos para las notificaciones. | 40 |
| 6.17 | Constructor de la clase ProductsManager donde se aplica DI. | 41 |
| 6.18 | Registro de interfaces en la capa de persistencia. | 41 |
| 6.19 | Metadatos de la API especificados en la clase Startup. | 42 |
| 6.20 | Documentación de la API generada con Swagger UI. | 42 |
| 6.21 | Añadir servicios para el logging. | 43 |
| 6.22 | Ejemplo del contenido de un log. | 43 |
| 6.23 | Fragmento del proxy de pedidos donde se adaptan los parámetros a tipos simples. | 44 |
| 6.24 | Fragmento de un proyecto donde se añade la opción TreatWarningsAsErrors. | 45 |
| 6.25 | Solución de la interfaz de usuario hecha con Xamarin.Forms. | 45 |
| 6.26 | Método para transformar de un DTO a un modelo de una vista. | 46 |
| 6.27 | Servicio concreto del proyecto de Android. | 47 |
| 6.28 | Pruebas realizadas en la solución monolítica. | 48 |
| 6.29 | Ejemplo de método Setup donde se emplea una base de datos en memoria. | 48 |
| 6.30 | Dockerfile de la solución monolítica. | 49 |
| 6.31 | Recurso App Service en el portal de Azure. | 49 |
| 6.32 | Despliegue a través de App Service. | 50 |
| 7.1 | División del modelo de dominio en contextos delimitados. | 52 |
| 7.2 | Diagrama de componentes de la solución basada en microservicios. | 53 |
| 7.3 | Organización de la solución basada en microservicios. | 54 |
| 7.4 | Dependencias del microservicio de pedidos en la capa de aplicación. | 55 |
| 7.5 | Código para resolver otros microservicios consumidos. | 55 |
| 7.6 | Clases de dominio de la entidad comentario en la solución monolítica (derecha) y la basada en microservicios (izquierda). | 56 |
| 7.7 | Parte del microservicio de notificaciones desarrollada en C#. | 57 |
| 7.8 | Parte del microservicio de notificaciones desarrollada en Java. | 57 |
| 7.9 | Proxy del microservicio de seguridad para su consumo en Java. | 58 |
| 7.10 | Dockerfile del microservicio de notificaciones. | 58 |
| 7.11 | Base de datos de incidencias en Firebase. | 59 |
| 7.12 | Fragmento de DAO que accede a una base de datos de Firebase. | 60 |
| 7.13 | Fragmento del proyecto (.csproj) del servicio de pedidos. | 61 |
| 7.14 | Archivo Versions.props con las versiones de los microservicios. | 61 |
| 7.15 | Cambios en la UI para emplear la solución basada en microservicios. | 62 |
| 7.16 | Fake de la interfaz de contratos del servicio de seguridad. | 63 |
| 7.17 | Fake de la interfaz de contratos del servicio de seguridad. | 63 |
| 7.18 | Método para eliminar registros escritos durante las pruebas del servicio de incidencias. | 64 |
| 8.1 | Estadísticas de uso de recursos de los contenedores de la solución monolítica y la basada en microservicios. | 69 |

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 las necesidades del negocio. De lo contrario, el futuro mantenimiento será más costoso y repercutirá en la confianza de los clientes y en la moral del equipo. [18]

Las arquitecturas basadas en microservicios son una tendencia actual donde diferentes funcionalidades se encapsulan en servicios pequeños y autónomos que cooperan entre ellos. Existe una gran cantidad de bibliografía que ilustra los beneficios de este tipo de arquitecturas. Este trabajo pretende aplicar un enfoque práctico a través de un caso de estudio para comparar de primera mano la realidad de este tipo de arquitecturas. Para ello, se desarrollará una aplicación de forma integral empleando tecnologías lo más cercanas posibles a las que se emplearían en el desarrollo profesional de un producto software.

El presente trabajo se ha desarrollado en el contexto de una colaboración con una PYME dedicada al desarrollo software. Esta empresa desarrolla un ERP para el sector socio-sanitario a la vez que apuesta por el uso de las tecnologías más punteras, como son el uso de microservicios. El autor de esta memoria ha colaborado en este espacio desarrollando algunos microservicios y herramientas que facilitan su construcción a través de la generación automática de código.

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.
- Examinar ambas arquitecturas respecto a los siguientes requisitos no funcionales: disponibilidad, tolerancia a fallos, utilización de recursos y capacidad para ser reemplazado.

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 su influencia en el proceso de desarrollo a lo largo del ciclo de vida del software.

El capítulo 3 presentará brevemente el estado del arte asociado a la tecnología de los microservicios, explicando brevemente algunas de las herramientas más empleadas.

En el capítulo 4 se analiza la aplicación a desarrollar mediante su especificación empleando casos de uso y un modelo de dominio.

El capítulo 5 contiene el proceso de desarrollo y la organización del trabajo que se seguirá durante la implementación de las dos soluciones propuestas.

El capítulo 6 explica el diseño e implementación del sistema siguiendo una arquitectura monolítica.

El capítulo 7 se centra en los mismos aspectos que el capítulo anterior pero para una arquitectura basada en microservicios.

En el capítulo 8 se realiza una comparación de ambas soluciones respecto a diferentes requisitos no funcionales 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 inconvenientes 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

Los microservicios en el proceso de desarrollo

2.1 ¿Qué son los microservicios?

Según Newman, los microservicios son servicios pequeños y autónomos que trabajan conjuntamente. [21] 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. [32] 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. [3] 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. [17] 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. [17]

2.2 Los microservicios en la fase de requisitos

La fase de requisitos del software es aquella en la que se elicitán, analizan, documentan, validan y mantienen los requisitos del sistema. Los requisitos del software expresan las necesidades y restricciones asociadas a un sistema. [10] 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 arquitectóni-

cos a seguir, las futuras estrategias de implementación del sistema o la plataforma sobre la que se desplegará el sistema. [2]



Figura 2.2: Características y subcaracterísticas definidas en el modelo de calidad del producto de la ISO/IEC 25010. [29]

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

- La **disponibilidad** se define como la capacidad del sistema de estar operativo y accesible para su uso cuando se requiere. [29] 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. [16] En la siguiente sección entraremos en más detalle sobre el balance que se debe realizar para elegir que propiedad se debe sacrificar.
- 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. [29] 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 resiliente que un sistema que puede continuar ofreciendo el resto de sus funcionalidades.
- La **utilización de recursos** se define en la ISO como la cantidad y tipos de recursos empleados durante el funcionamiento del software bajo unas condiciones determinadas. [29] Por un lado, los arquitecturas basadas en microservicios ofrecen beneficios ya que permiten que cada servicio se despliegue en un hardware diferente con unas prestaciones acordes a sus necesidades. También permiten que se escale cada servicio de acuerdo a su demanda de forma independiente, en lugar de tener que escalar el sistema monolítico al completo. [7] Por otro lado, los servicios se ejecutan en diferentes procesos y las llamadas interprocedurales a través de la red pueden suponer un mayor consumo de recursos. [15]
- La **capacidad para ser reemplazado** es la capacidad de un producto software para ser reemplazado por otro en el mismo entorno y con el mismo propósito. La ventaja de los microservicios es que, gracias a su diseño modular, las piezas de software se pueden reemplazar fácilmente. Cada microservicio es un componente de software

que puede ser completamente reescrito en poco tiempo. Autores como Jon Eaves estiman que el tiempo para hacerlo no debería superar las dos semanas. Esto hace que se pueda reaccionar a situaciones imprevistas más fácilmente. Por ejemplo, para que un sistema pueda operar en un entorno determinado puede ocurrir que se tenga que reescribir un servicio por otra persona a la que inicialmente lo desarrolló. Al reescribirlo se puede emplear otro lenguaje de programación; basta con mantener la misma interfaz del servicio para que otros puedan continuar operando con él. [9] Si en lugar de reescribir un servicio se tuviera que reescribir toda la aplicación, el proceso de hacerlo no se podría abordar de forma incremental si la aplicación fuera un monolito.

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. [21]

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. [27]

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. [17]

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. [31]

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 distinto. 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.

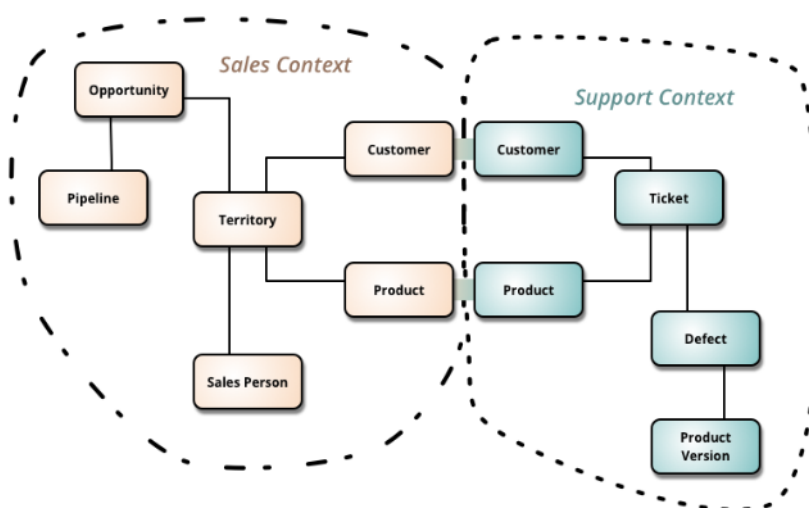


Figura 2.3: Ejemplo de dos contextos delimitados dentro del mismo dominio que emplean el mismo nombre para un concepto, pero con significados diferentes. [14]

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. [21]

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. Un mal diseño inicial puede desenvocar en que dos servicios se tengan que combinar porque exista un exceso de interacciones entre ellos.

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. [26]

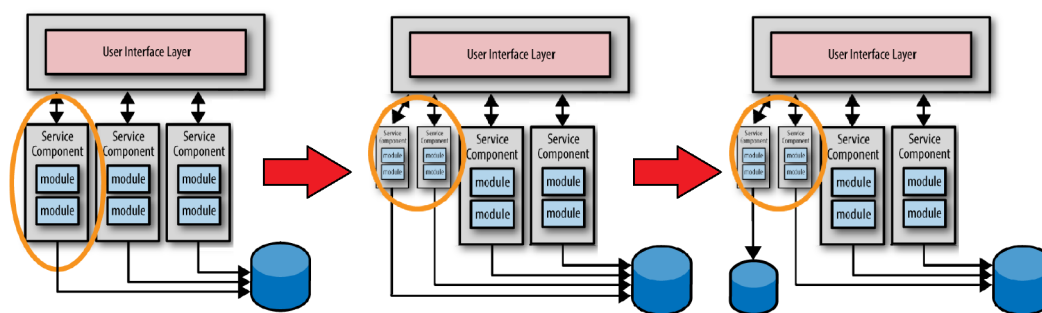


Figura 2.4: Primero, se dividen los grandes servicios en microservicios. Una vez hecho esto, se migran sus datos. [26]

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. [21]

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. [27]

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. [21] 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 demanda. 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.

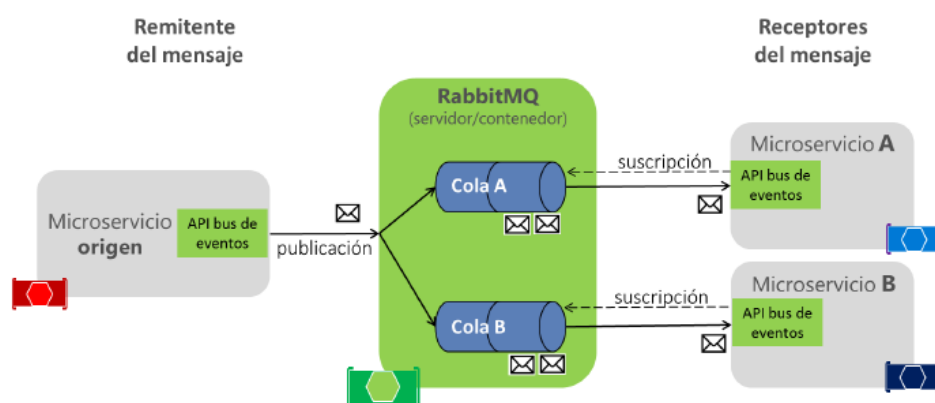


Figura 2.5: Ejemplo de integración basada en eventos con un contenedor de RabbitMQ. [7]

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. [20] 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 resiliencia 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. [21]

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. [7] 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.

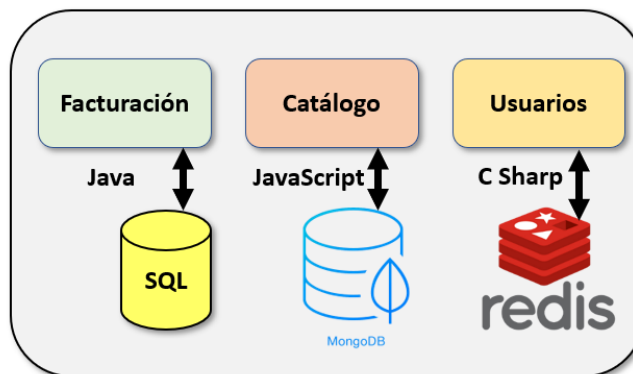


Figura 2.6: Ejemplo de sistema políglota.

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. [12]

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. La ley de Conway

Problemas asociados a bases de código inmensas donde colabora un gran equipo pueden ser evitados empleando microservicios. Si el código está repartido entre diferentes componentes, equipos pequeños pueden encargarse de mantener cada uno de ellos. De esta manera, la organización del trabajo adquiere un enfoque más ágil al estar formado por equipos independientes y auto-organizados. [21]

En las aplicaciones monolíticas, normalmente los equipos de trabajo que se forman están asociados la interfaz de usuario, la base de datos y la parte servidora. Cualquier cambio simple involucra la coordinación entre diferentes equipos. De acuerdo a la ley de Conway, una organización que diseña un sistema producirá un diseño para la aplicación que será una copia de su estructura organizacional.

La organización en los microservicios se realiza alrededor de las capacidades de negocio. Los equipos que se encargan de cada servicio son multifuncionales y se pueden autogestionar. En cada equipo están presentes, entre la totalidad de sus miembros, todas las habilidades necesarias para el desarrollo completo del servicio. [17]

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. [27]

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. [22] 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.

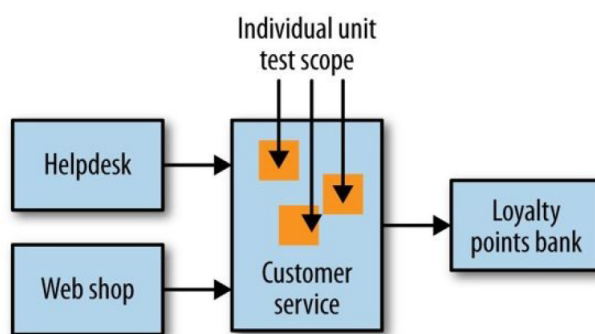


Figura 2.7: Diagrama de pruebas unitarias.

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. [22] 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.

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

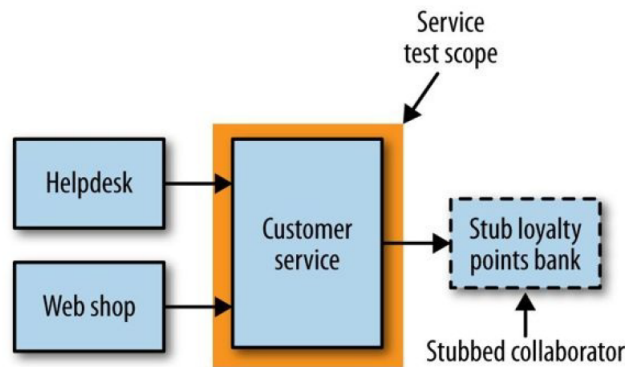


Figura 2.8: Diagrama de pruebas de servicios.

está situación. Su tiempo de ejecución es mayor y en consecuencia, más tarde se detecta si un cambio ha introducido un defecto. [21]

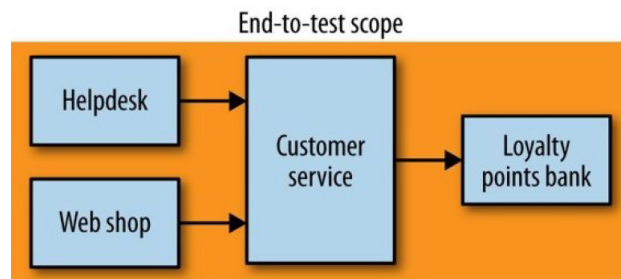


Figura 2.9: Diagrama de pruebas de extremo a extremo.

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. [5]

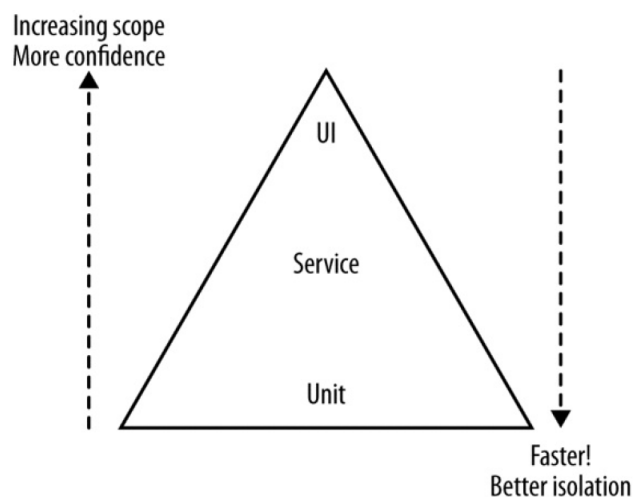


Figura 2.10: Pirámide de pruebas diseñada por Mike Cohn. [5]

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.

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. [23] 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. [11] 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. [11]
- **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. [21]



Figura 2.11: Ejemplo de una pipeline. [21]

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**. [13]

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. [21]

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. [8] Esto es muy importante en desarrollos que siguen las técnicas de integración continua donde se despliegan artefactos frecuentemente.

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. [8] Su uso de recursos 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. [21]

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.

2.7 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

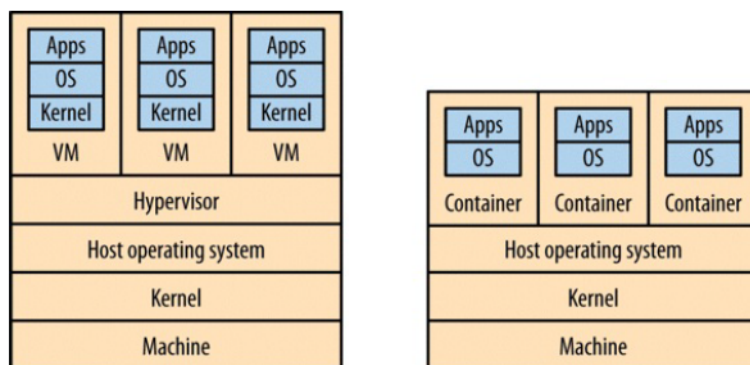


Figura 2.12: Comparación entre la virtualización y la contenerización. [21]

los usuarios se ven modificados. Es una fase que debe comenzar de forma temprana para garantizar la mantenibilidad del sistema a desarrollar. [27]

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 sí mismo. [23]

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. [17]

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. [21]

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: **zou build, you run it**". [17]

No hay necesidad de distinguir entre quien construye un sistema y quien lo ejecuta y posteriormente mantiene. Esta filosofía aproxima a desarrolladores 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 di-

ferentes equipos y añade un sobreesfuerzo de coordinación y comunicación cuando un problema aparece. [30]

2.7.3. Documentación

Una de las ventajas de los microservicios es que acelera 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**. [15]

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 información 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 excepciones en el servicio. [15]

CAPÍTULO 3

Estado del arte de la tecnología de microservicios

3.1 Contenedores

3.1.1. Contenedores Linux

Los contenedores Linux (LXC) permite crear un espacio de procesos separado en el que puede ejecutarse de forma aislada un servicio. Cada contenedor es un subárbol del principal y puede tener asociado recursos físicos gestionados por el kernel. [21].

Estos contenedores se fundamentan en los espacios de nombre de Linux y los grupos de control (cgroups). Los espacios de nombre aísla el conjunto del sistema de archivos que un grupo de procesos puede ver. Los grupos de control organizan los procesos en un árbol de forma jerárquica, donde se pueden definir límites y políticas para el uso de los recursos. [1]

La principal ventaja de los LXC es que son una implementación muy ligera que se ejecuta a velocidades muy similares a las de la máquina donde se ejecuta. Sin embargo, está limitado al uso de Linux como base del entorno, estando muy acoplado a su kernel, y presenta problemas en cuanto a la seguridad y aislamiento de los contenedores. [8] Una solución pasa por usar contenedores Linux contenidos en máquinas virtuales para establecer los límites de seguridad a través de la MV y los límites de los recursos empleados por la aplicación a través de los contenedores. [6]

3.1.2. 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. [19] 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. [7]

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. [19]

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.

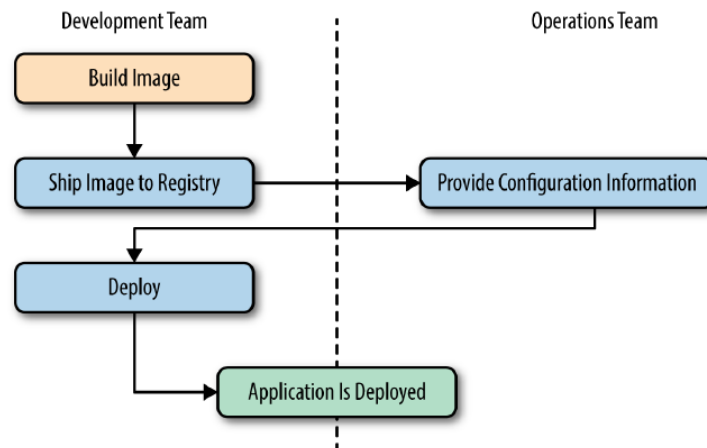


Figura 3.1: Proceso de despliegue con Docker. [19]

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.

3.2 Orquestadores

3.2.1. Kubernetes

Kubernetes es una herramienta diseñada por Google para el despliegue y orquestación de aplicaciones en contenedores Docker de forma resiliente. Cada una de las máquinas que aloja uno o más contenedores Docker se denomina **nodo**. La unidad más pequeña con la que trabaja Kubernetes no son los contenedores, sino los pods. Un **pod** es una colección de contenedores 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. [25]

Kubernetes se emplea principalmente para especificar el número de replicas que se desea tener simultáneamente de un pod. La herramienta es buena para asegurar la disponibilidad de un servicio, pero no garantizan la escalabilidad de esta, 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 balanceadores redirigirían cada petición al pod correspondiente y se encargarán de ajustar el número de replicas de un pod según una serie de reglas.

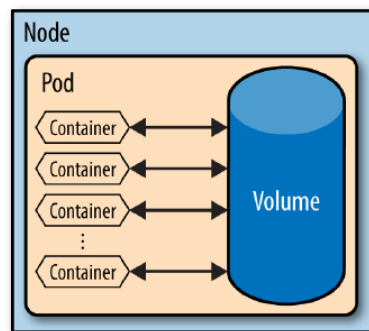


Figura 3.2: Un nodo de Kubernetes. [25]

3.2.2. Docker Swarm

Docker Swarm es la aproximación nativa propuesta por Docker para la creación de clústeres, donde un conjunto de contenedores Docker aparentan formar uno único de forma virtual. En Docker Swarm, los nodos ejecutan un agente llamado **Swarm** y uno de ellos ejecuta un gestor para su coordinación.

Cuando un contenedor es creado, este se despliega en cualquiera de los nodos gestionados por el Swarm. El conjunto de contenedores son gestionados por el Swarm como una única entidad.

Docker Swarm se puede ejecutar para ofrecer servicios con alta disponibilidad a través de herramientas como etcd, Consul o ZooKeeper para la gestión y recuperación de los fallos. Además, está complementamente integrado dentro de la interfaz de línea de comandos de Docker. [6]

3.3 Proveedores de servicios en la nube

3.3.1. Amazon Web Services (AWS)

Amazon es el líder en el mercado de computación en la nube con su producto Amazon Web Services (AWS), lanzado en 2006. Fue el primer proveedor en ofrecer infraestructura como servicio (IaaS), permitiendo el alquiler de máquinas virtuales.

Amazon Elastic Compute Cloud (EC2) permite a sus usuarios la creación de las máquinas necesarias para la ejecución de una aplicación. Este servicio es capaz de escalar las aplicaciones de acuerdo a las necesidades de sus usuarios. La creación, ejecución y destrucción de máquinas virtuales o instancias puede ser controlada a demanda del usuario de AWS. El pago de esta plataforma se realiza por horas, a diferencia de Azure, donde se paga por minutos y el pago es más exacto con el uso realizado. [24]

3.3.2. Microsoft Azure

Microsoft Azure es una plataforma que ofrece un conjunto de herramientas y servicios en la nube. Es el segundo mayor proveedor de servicios en la nube y fue lanzado en 2010. Ofrece tanto funcionalidades de plataforma como servicio (PaaS) como IaaS.

Los usuarios pueden crear, desplegar y gestionar aplicaciones servicios a través de la red global de los centros de datos de Microsoft. Aunque el pago de sus servicios sea más cercano a la realidad, los modos de pago son manejados de forma poco transparente. [24]

3.4 Crítica al estado del arte

3.5 Propuesta

En el caso de estudio, en el desarrollo de la solución basada en microservicios se van a emplear contenedores Docker. Para su orquestación, en producción se hará uso de Kubernetes. En desarrollo, se empleará Docker Compose para agilizar la creación de los diferentes contenedores.

En cuanto a la comunicación entre los servicios y con la interfaz de usuario, en la sección 2.4.1 **Integración de microservicios** hemos presentado tres posibles alternativas. Hemos optado por la que la literatura considera más sencilla, la integración REST a través de HTTP.

En cuanto a aspectos más técnicos, el lenguaje de programación que se va a emplear principalmente 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

Para el desarrollo de la interfaz de usuario se va a utilizar Xamarin.Forms. Xamarin es una plataforma que permite desarrollar aplicaciones móviles para dispositivos Universal Windows Platform (UWP), Android y iOS empleando código C#. Xamarin.Forms es un conjunto de herramientas para Xamarin centrado en el desarrollo multiplataforma. Su propósito que se pueda compartir la mayor cantidad de código posible en el desarrollo de una aplicación para las distintas plataformas que hemos mencionado.³

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). Hemos decidido emplear esta tecnología porque es la mejor soportada por el IDE en el que vamos a trabajar, ofreciendo funcionalidades para desplegar una nueva versión simplemente pulsando un botón. Tanto App Service como AKS son parte de los servicios de Azure. 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.⁴

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.⁵

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 Introducción a Xamarin.Forms : <https://docs.microsoft.com/es-es/xamarin/xamarin-forms/get-started/introduction-to-xamarin-forms>

⁴ Página oficial de Microsoft Azure: <https://azure.microsoft.com/es-es/>

⁵ Página oficial de Postman: <https://www.getpostman.com/>

CAPÍTULO 4

Especificación de requisitos del caso de estudio

4.1 Descripción del caso de estudio

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.

4.2 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:

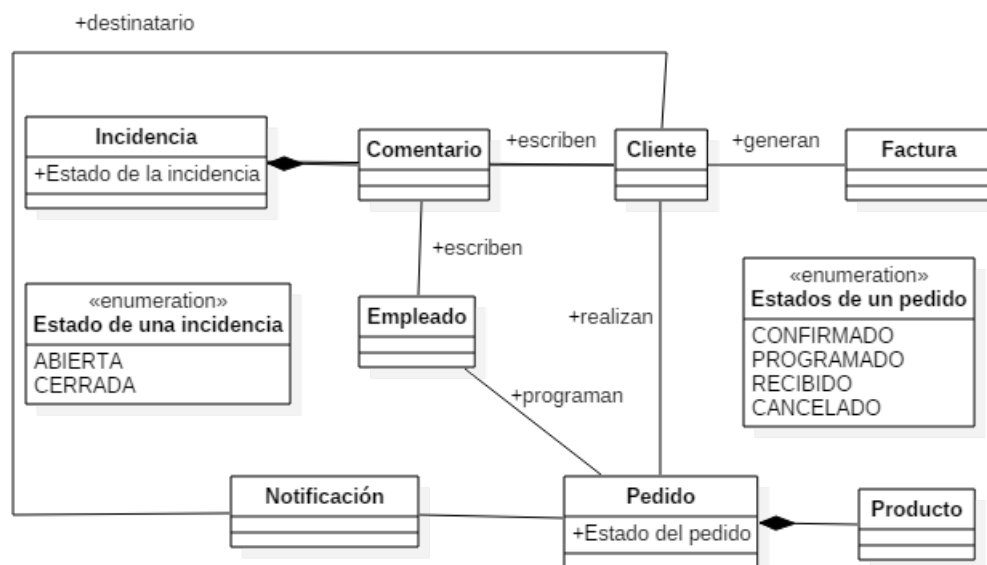


Figura 4.1: Modelo de dominio del sistema.

CAPÍTULO 5

Proceso de desarrollo

5.1 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 para obtener la solución basada en microservicios, siguiendo los principios que en la sección **2.3 Los microservicios en la fase de diseño** se han mencionado.

Una vez se implemente la solución monolítica, se comenzará la implementación de la aplicación front-end. Las llamadas a la parte servidora van a realizarse a través de invocaciones HTTP. En consecuencia, el front-end va a estar desacoplado de la implementación que elijamos para el back-end y va a poder emplearse la misma solución, con muy pequeñas modificaciones, para comunicarse con ambas partes servidoras.

La organización de la memoria en los siguientes capítulos es acorde a este proceso. Primero, presentaremos la solución monolítica centrándonos sobretodo en aspectos de implementación, que son comunes a los de la solución basada en servicios. Después, explicaremos la solución basada en microservicios, centrándonos sobretodo en aspectos de diseño y modificaciones que se han realizado frente a la monolítica.

5.2 Organización del trabajo

El proceso de desarrollo se ha desglosado en grandes tareas, muy ligadas a las entidades del dominio que existen en el caso de estudio. En el siguiente cronograma se pueden observar estas tareas y su evolución a lo largo del tiempo que ha durado el desarrollo de las soluciones. En azul se han representado las tareas asociadas a la solución monolítica, en verde las asociadas al front-end y en naranja las asociadas a la solución basada en microservicios.

Se puede observar como de las 3 fases del desarrollo donde más esfuerzo se ha dedicado es en la solución monolítica. Esto es porque es aquí donde se ha implementado casi la totalidad del código del sistema y donde se han evaluado muchas de las tecnologías empleadas. La fase de desarrollo de la solución basada en microservicios se prolonga menos porque únicamente ha involucrado refactorizar el diseño monolítico y a penas se ha modificado la implementación del sistema. En un sistema real, esta tarea sería mucho más costosa que en nuestro caso de estudio.

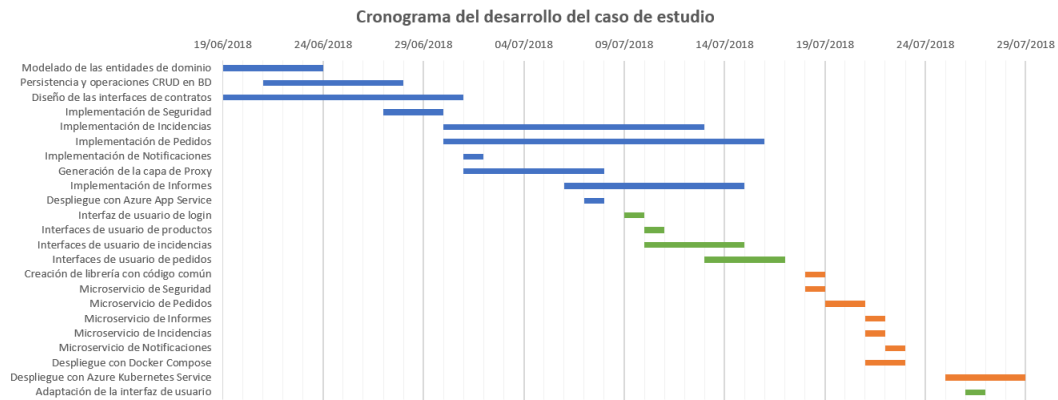


Figura 5.1: Cronograma del proceso de desarrollo del caso de estudio.

Todo el código desarrollado se va a almacenar en un repositorio de código de GitHub. Se han creado tres repositorios: uno para el back-end monolítico, otro para la interfaz de usuario y otro para el back-end basado en servicios. Adicionalmente, se ha creado un repositorio para el desarrollo de la memoria de este TFG.

En los repositorios de GitHub vamos a emplear un proyecto. Un proyecto de GitHub es similar a un Trello. Aquí podemos crear columnas, que representan un estado del trabajo, y tarjetas, que representan tareas a realizar y que van transitando de una columna a otra conforme se van completando. En nuestro caso, las tareas que vamos a incluir en el proyecto son de un detalle más bajo a las que hemos incluido en el cronograma, más cercanas a las tareas que debe llevar a cabo el programador. El proyecto lo hemos dividido en 4 columnas: TODO, que contiene las tareas pendientes, TODO - Priority, que contiene también tareas pendientes pero que urge resolver, DONE, para las tareas ya realizadas, y DISCARD, para las tareas que se han descartado y ya no se van a implementar.

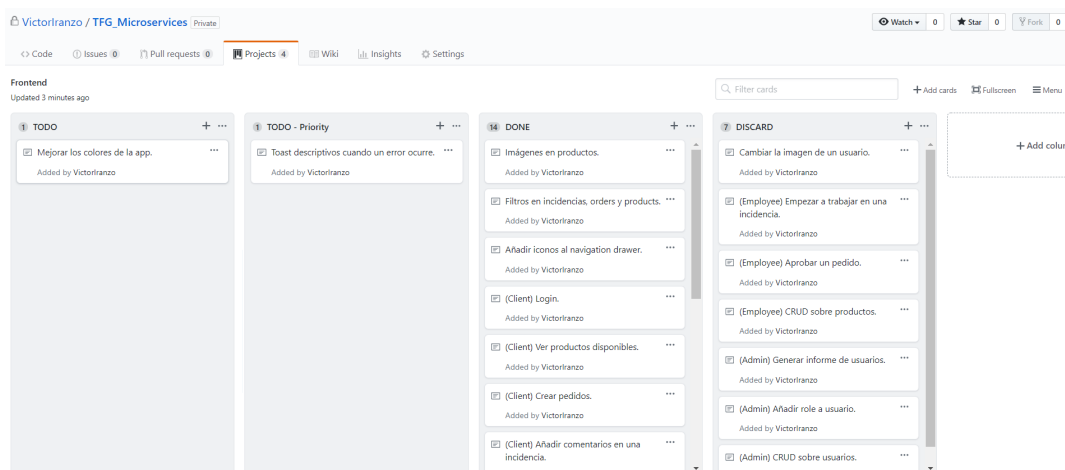


Figura 5.2: Proyecto de GitHub asociado al desarrollo del front-end.

El proceso de desarrollo se puede visualizar en los diferentes repositorios a través de los commits que en ellos se hacen. En el siguiente gráfico se muestra el número de commits realizados cada día sobre los tres repositorios mencionados. Como en el cronograma, se puede ver como las fases de front-end e implementación de la solución monolítica se superponen en la mitad del proceso de desarrollo. También se puede ver como la fase de la solución monolítica es la que más commits ha requerido porque ha sido la más inestable respecto a defectos corregidos y detalles de implementación.



Figura 5.3: Número de commits realizados cada día en los repositorios del caso de estudio.

CAPÍTULO 6

Diseño e implementación de la solución monolítica

6.1 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 encargada de lanzar la excepción oportuna. También aquí se situarán los conversores para transformar una entidad 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.

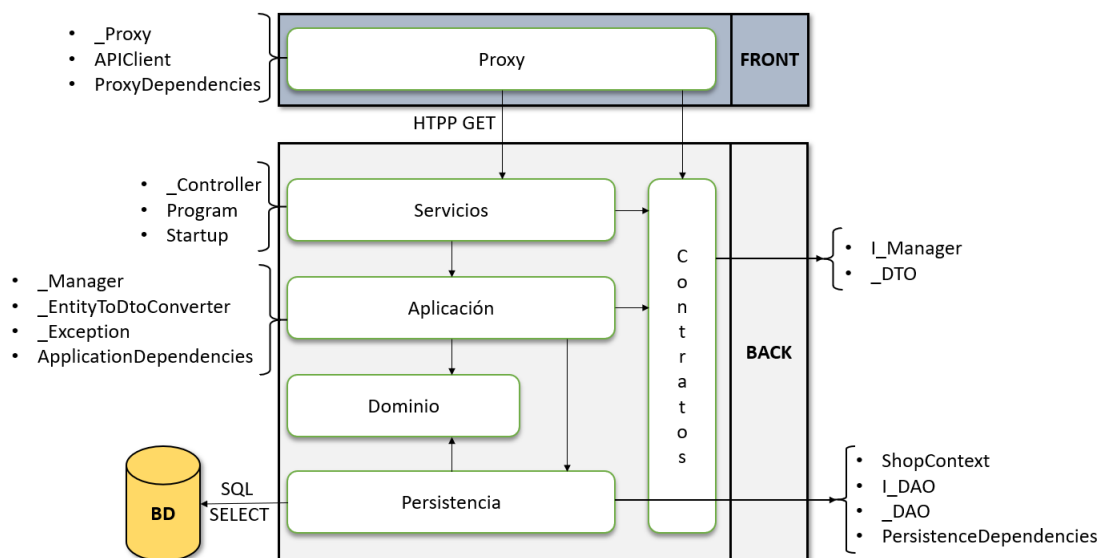


Figura 6.1: Capas del back-end monolítico.

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 la 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>

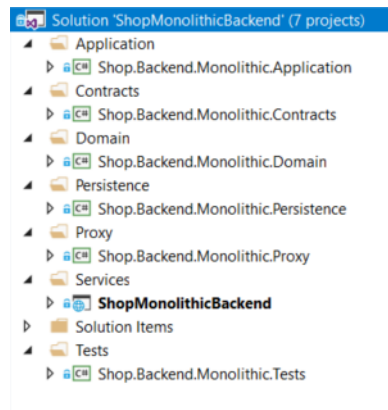


Figura 6.2: Solución del sistema monolítico.

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:

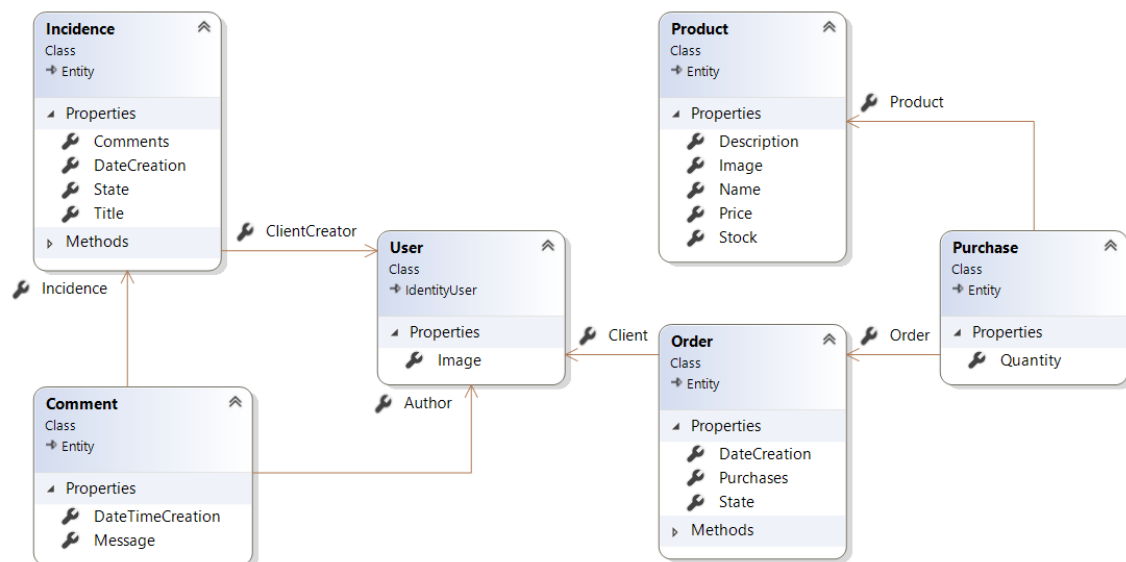


Figura 6.3: Diagrama de clases de dominio de la solución monolítica.

6.2 Detalles de la implementación back-end

6.2.1. 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 forma 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. [21]

```

/// <summary> The internal interface for the managers that offer CRUD operations ...
3 references | VictorIrranzo, 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 | VictorIrranzo, 17 days ago | 1 author, 4 changes | 0 exceptions
    Task<Guid> CreateAsync(TCreateDTO createDTO);

    /// <summary> Creates the specified entities in the database.
    4 references | VictorIrranzo, 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 | VictorIrranzo, 21 days ago | 1 author, 3 changes | 0 exceptions
    Task<TEntityDTO> GetAsync(Guid entityId);

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

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

```

Figura 6.4: 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.

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 sobrescrita 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

```

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

    /// <summary> Generates the report of and order.
    4 references | VictorIrranzo, 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 | VictorIrranzo, 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 6.5: Interfaz en la capa de contratos asociada a la entidad Pedido.

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.

6.2.2. Seguridad

La mayoría de métodos expuestos en la API del backend requieren de mecanismos de autorización para establecer si la persona que realiza una petición puede acceder o no a los datos que solicita. Con este propósito, vamos a hacer uso de **ASP.NET Core Identity** y de tokens de seguridad JWT (JSON Web Token).

Identity es el sistema de ASP.NET Core para administrar los usuarios registrados de una aplicación a través de proveedores externos como Google y Facebook o a través de una base de datos de usuarios propia.⁴ Atendiendo a los requisitos especificados, no necesitamos que nuestros usuarios se puedan autenticar a través de proveedores externos, por lo que vamos a optar por gestionar los usuarios de la aplicación nosotros mismos. Los pasos para conseguirlo se resumen a continuación:

- En la capa de dominio, se debe crear una entidad que represente al usuario de la aplicación y que herede de la clase `IdentityUser`. La clase `IdentityUser` provee numerosos atributos, como el nombre o el correo electrónico. Se puede añadir cualquier atributo más asociado al usuario de la aplicación dentro de la clase `User`.

```

/// <summary> The class that represents an identified user.
33 references | VictorIrranzo, 14 days ago | 1 author, 5 changes
public class User : IdentityUser
{
    /// <summary> Gets or sets the image of the user.
    3 references | VictorIrranzo, 14 days ago | 1 author, 1 change | 0 exceptions
    public byte[] Image { get; set; }
}

```

Figura 6.6: La clase User.

- En la capa de persistencia, para asegurar la creación de las tablas asociadas a Identity, el contexto de la aplicación que representa a la base de datos debe extender

⁴ Introducción a la identidad en ASP.NET Core: <https://docs.microsoft.com/es-es/aspnet/core/security/authentication/identity>

la clase `IdentityDbContext<User>`. En la base de datos, las tablas de datos creadas automáticamente por Identity se identifican por el prefijo `AspNet`.

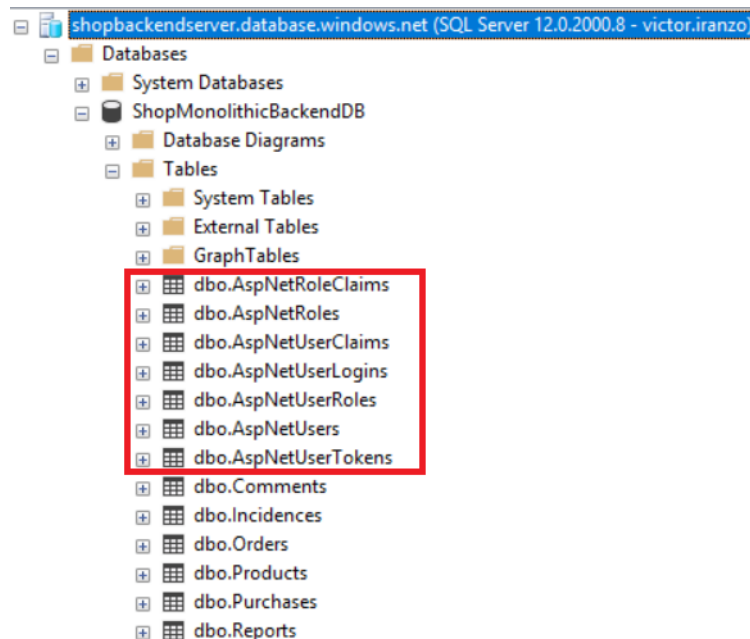


Figura 6.7: Tablas creadas por ASP.NET Core Identity.

- En la capa de servicios, a los métodos de la API que requieran de autorización se les ha de añadir el atributo `Authorize`. Con esto, cuando se realice una petición sin proveer un token de autorización se devolverá automáticamente un error 401: `Unauthorized`. En el atributo `Authorize` se puede especificar si el método puede ser ejecutado solo por los usuarios que tengan cierto rol.⁵ Por ejemplo, podemos establecer que las operaciones agregadas de tipo CRUD solo puedan ser invocadas por los usuarios con el rol `Administrador`.

```
/// <summary> Creates the specified entity in the database.
[HttpPost("Create")]
[Authorize]
[ProducesResponseType(typeof(Guid), 200)]
0 references | VictorIranzo, 9 days ago | 1 author, 1 change | 0 exceptions
public virtual async Task<Guid> CreateAsync([FromBody]TCreatedTO createdTO)...

/// <summary> Creates the specified entities in the database.
[HttpPost("CreateAggregated")]
[Authorize(Roles = "Administrator")]
[ProducesResponseType(typeof(bool), 200)]
0 references | VictorIranzo, 9 days ago | 1 author, 1 change | 0 exceptions
public virtual async Task<bool> CreateAsync([FromBody]IList<TCreatedTO> createdTOs)...
```

Figura 6.8: Ejemplo de métodos de la clase de servicios.

- Por último, en la capa de servicios se debe especificar que el mecanismo por el cual el usuario proveerá su identidad será a través de tokens JWT. Para ello, en la clase `Startup` se debe añadir a la colección de servicios el servicio de autenticación con este tipo de tokens, indicando que los únicos tokens válidos son los firmados con la clave provista en el archivo de configuración.⁶

⁵ Autorización basada en roles en ASP.NET Core: <https://docs.microsoft.com/es-es/aspnet/core/security/authorization/roles?view=aspnetcore-2.1>

⁶ Securing ASP.NET Core 2.0 Applications with JWTs: <https://auth0.com/blog/securing-asp-dot-net-core-2-applications-with-jwts/>

```
// ===== Adds Jwt Authentication. =====
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear(); // Removes default claims.
services
    .AddAuthentication(options =>
    {
        // Indicates that the authentication is done using JWT Tokens.
        options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(cfg =>
    {
        // In development environments it's not mandatory to use HTTPS.
        cfg.RequireHttpsMetadata = false;

        // The token is stored in the application and it will be added
        // to every request automatically while the session is open.
        cfg.SaveToken = true;

        // Specifies the valid issuer and key that signs the JWT token from
        // the configuration files.
        cfg.TokenValidationParameters = new TokenValidationParameters
        {
            ValidIssuer = this.Configuration["JwtIssuer"],
            ValidAudience = this.Configuration["JwtIssuer"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(this.Configuration["JwtKey"])),
        };
    });
```

Figura 6.9: Pieza de código para añadir la autenticación usando tokens JWT.

El usuario podrá obtener el token que le identifica a través del método Login del manager de Seguridad. Para hacerlo, deberá proveer su correo electrónico y contraseña. En la implementación de este método en la capa de aplicación se comprueba que el correo electrónico y contraseña coinciden con los almacenados en las tablas de Identity. Si coinciden, el método devuelve un token JWT donde se incluirán los datos del usuario y la fecha en la que el token expira. En cada petición HTTP que haga el usuario a la parte servidora deberá proveer este en la cabecera.

```
GET /Shop/Microservices/v1/Incidentes/GetById HTTP/1.1
Host: localhost:8080
entityId: f2644dac-42ed-46df-aeb0-99d4c7cd514a
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
    .eyJzdWIiOiJ2aXJhbnpvOTZAZ21haWwY29tIiwianRpIjoieYzMDA0N2Q0NDcyLWE3OTEtYjQ2ZTBhOT
    NjMDUyIiwiaHR0cDovL3NjaGVtYXMuG1sc29hcC5vcmlld3MvMjAwNS8wNS9pZGVudG10eS9jbGFpbXMvbmFtZW1kZ
    W50aWZpZXIiOiJmOTk2ODUwNy1jY2RlLTQwOGQtYTI0MS05MmFjNDVlY2Q0YmQlLCJodHRwOi8vc2NoZW1hcy54bWxz
    b2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy91bWVpY29kZ3MvMjAwNS8wNS9pZGVudG10eS9jbGFpbXMvbmFtZW1kZ
    tIiwiaHR0cDovL3NjaGVtYXMuG1sc29hcC5vcmlld3MvMjAwNS8wNS9pZGVudG10eS9jbGFpbXMvbmFtZW1kZpY3
    RvciIsImh0dHA6Ly9zY2h1bWVzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudG10eS9jbGFpbXMvbmFtZW1kZ
    kFkbWluaXN0cmF0b3IiLCJleHAiOiE1MzI0NTE5NDksIm1zcyI6IiIzPy3Rvci5JcmFuZC5lc3Rvci5JcmFuZC5lc3Rvci5J
    SXJhbnpvIn0.zo6Yx9ABDuu61D7KjSNEXR81maTYOXswJyBh3XZ-bhc
Cache-Control: no-cache
```

Figura 6.10: Ejemplo de petición HTTP donde se incluye un token de seguridad.

6.2.3. Persistencia

Para la persistencia se va a emplear **Entity Framework Core** para el mapeo objeto-relacional y una base de datos SQL en la nube de **Microsoft Azure**. Entity Framework Core es una versión multiplataforma del ORM Entity Framework (EF) para trabajar con una base de datos mediante objetos .NET.⁷

Vamos a emplear una aproximación Code-First para el diseño de la base de datos. Esta aproximación nos permite diseñar primero las entidades de nuestro dominio a través de las clases en la capa de dominio y luego trasladar sus atributos y relaciones a un esquema de base de datos gracias a EF.⁸

Para la creación de la base de datos se debe ir al portal de Azure⁹ y navegar hasta la página SQL Databases. Para la creación de la base de datos hay que proveer principalmente su nombre, la suscripción a la que se redirigirán los costes asociados, el grupo de recursos en el que se incluirá y el servidor donde se emplazará.

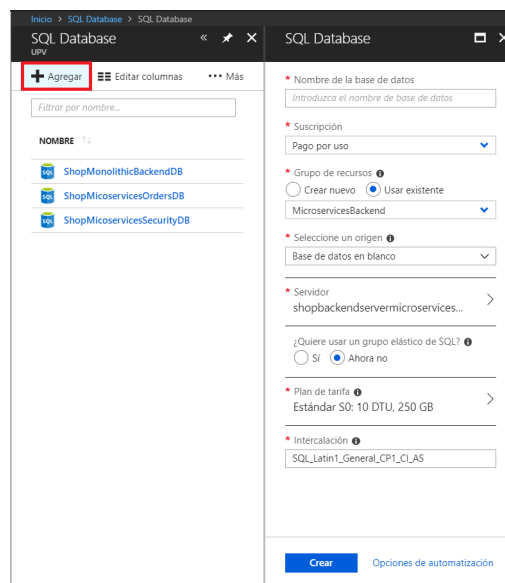


Figura 6.11: Creación de una base de datos SQL en Azure.

Una vez creada, debemos ir al recurso y en la pestaña de cadenas de conexión copiar la asociada a .NET. La cadena de conexión la pegaremos en el archivo de configuración de la capa de servicios. En la clase Startup del mismo proyecto leeremos la cadena de conexión para configurar la clase ShopContext.

```
// Sets the configuration of the context to link it to the SQL database
// paced in Azure.
services.AddDbContext<ShopContext>(options =>
{
    options.UseSqlServer(
        this.Configuration["ConnectionString"],
        sqlServerOptionsAction: sqlOptions =>{...});
});
```

Figura 6.12: Configuración del contexto de la capa de persistencia para apuntar a la BD en Azure.

En la clase ShopContext debemos indicar cuáles son las entidades del dominio que se añadirán como tablas a la BD. Para cada entidad del dominio se creará un DbSet distinto.

⁷Descripción general de Entity Framework Core: <https://docs.microsoft.com/es-es/ef/core/>

⁸What is Code-First?: <http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx>

⁹ Portal de Azure: <https://portal.azure.com/>


```

/// <summary> The Shop database context.
12 references | VictorIrranzo, 31 days ago | 1 author, 4 changes
public partial class ShopContext : IdentityDbContext<User>
{
    /// <summary> Initializes a new instance of the ShopContext class.
    0 references | VictorIrranzo, 37 days ago | 1 author, 1 change | 0 exceptions
    public ShopContext(DbContextOptions<ShopContext> options) {...}

    /// <summary> Gets or sets the comments collection.
    1 reference | VictorIrranzo, 31 days ago | 1 author, 1 change | 0 exceptions
    internal DbSet<Comment> Comments { get; set; }

    /// <summary> Gets or sets the incidences collection.
    2 references | VictorIrranzo, 31 days ago | 1 author, 1 change | 0 exceptions
    internal DbSet<Incidence> Incidences { get; set; }

    /// <summary> Gets or sets the orders collection.
    2 references | VictorIrranzo, 37 days ago | 1 author, 1 change | 0 exceptions
    internal DbSet<Order> Orders { get; set; }
}

```

Figura 6.13: Clase ShopContext.

Por último, para que desde otras capas accedan a los datos emplearemos el **patrón DAO**. Exponer el contexto entero fuera de la capa de persistencia puede ser peligroso porque permitiría modificar desde otras capas aspectos que solo deben conocerse en esta capa. Por ello, se definirá para cada entidad del dominio una interfaz para acceder a sus datos donde las operaciones permitidas están acotadas. De nuevo, tanto estas interfaces como su implementación se pueden definir de forma genérica y si se desea se puede extender o sobrescribir su comportamiento.

```

/// <summary> Interface for the Data Access Object's that implement CRUD operati ...
23 references | VictorIrranzo, 29 days ago | 1 author, 4 changes
public interface ICrudDAO<TEntity>
    where TEntity : Entity
{
    /// <summary> Creates the entity in the persistence database.
    3 references | VictorIrranzo, 29 days ago | 1 author, 1 change | 0 exceptions
    int Create(TEntity entity);

    /// <summary> Creates the entity in the persistence database.
    2 references | VictorIrranzo, 29 days ago | 1 author, 1 change | 0 exceptions
    int Create(IEnumerable<TEntity> entities);

    /// <summary> Deletes the entity in the persistence layer.
    3 references | VictorIrranzo, 29 days ago | 1 author, 1 change | 0 exceptions
    int Delete(TEntity entity);
}

```

Figura 6.14: Interfaz genérica para los DAOs que exponen operaciones CRUD.

6.2.4. Informes empleando la librería Open XML PowerTools

La generación de informes es una de las funcionalidades más empleadas en los software de gestión. En los requisitos de nuestro sistema solo se ha establecido un informe: la factura de un pedido. Esto no implica que debamos dejar de seguir el principio de responsabilidad única. La lógica para generar un informe debe ser genérica para que no dependa del tipo de informe que se va a generar y así pueda ser invocada desde diferentes sitios. Vamos a hacer uso de una librería para la generación de informes con estas librerías: **Open XML PowerTools**.

Open XML PowerTools provee funcionalidades para la combinación de documentos, la conversión de estos a diferentes formatos y la creación de informes a partir de plantillas.¹⁰ Para generar un informe se separan explícitamente los datos que lo originan y

¹⁰ Página de GitHub de Open XML PowerTools: <https://github.com/OfficeDev/Open-Xml-PowerTools>

la plantilla del documento. En las plantillas se define cómo se van a renderizar los datos mediante un lenguaje basado en anotaciones. Este lenguaje nos permite renderizar contenido en base a condiciones, iterar sobre las colecciones en los datos, crear tablas, etc.

Tienda Online

Factura

Factura

Cliente: <# <Content Select= "/Client" /> #>

State of the order: <# <Content Select= "/State" /> #>

Artículos:

Número de artículos: <# <Content Select= "/NumberOfArticles" /> #>

<# <Repeat Select= "/Purchases/PurchaseReportDTO" /> #>

| | | | |
|----------------------|---|------------------|---|
| Nombre del producto: | <# <Content Select= "/ProductName" /> #> | | |
| Descripción: | <# <Content Select= "/ProductDescription" /> #> | | |
| Nº de unidades: | <# <Content Select= "/Quantity" /> #> | Precio unitario: | <# <Content Select= "/Price" /> #> € |
| | | Precio total: | <# <Content Select= "/TotalProduct" /> #> € |

<# <EndRepeat/> #>

Figura 6.15: Plantilla para la creación de facturas.

Las plantillas se almacenarán en la base de datos y cuando se quiera generar un informe se recuperará esta para ensamblar el informe. Los métodos para generar un informe concreto estarán expuestos en los managers de la entidad asociada al informe. Por ejemplo, el método para generar la factura de un pedido se situará en el OrdersManager. En este método se recuperarán los datos para generar la factura y se enviarán estos al manager de informes para que los combine con la plantilla de la factura.

6.2.5. Notificaciones con la librería MailKit

Con las notificaciones ocurre lo mismo que con los informes: solo existe un caso de uso que envíe una notificación al cliente, pero para segregar responsabilidades vamos a crear un manager específico para este propósito. Para enviar notificaciones se va emplear la librería **MailKit**, una librería multiplataforma con utilidades sobre los protocolos IMAP, POP3 y SMTP.

Para desacoplar los mensajes a enviar y el código para enviarlo se hará uso de un DTO que contenga el usuario destinatario, el asunto del correo y el cuerpo del mismo. Tanto el cuerpo como el asunto de la notificación se definirán en archivos de recursos en la capa de aplicación.

| Name | Value |
|------------------------------|--|
| MessageOrderToDeliverBody | Hola {0}, Tu pedido con identificador {1} está preparado para ser entregado. En los próximos días lo tendrás disponible en tu hogar. |
| MessageOrderToDeliverSubject | El pedido {0} está preparado para ser entregado. |

Figura 6.16: Archivo de recursos para las notificaciones.

6.2.6. Inyección de dependencias

La **inyección de dependencias** (DI) es un mecanismo para desacoplar un objeto de sus colaboradores. En lugar de instanciar o obtener una referencia a los objetos de los que depende una clase, estos se declaran en el constructor de la clase y es el sistema quien automáticamente los resuelve e instancia.¹¹

```
/// <summary>
/// Initializes a new instance of the <see cref="ProductsManager"/> class.
/// </summary>
/// <param name="productCrudDAO"> The product DAO. </param>
/// <param name="productEntityDTOConverter"> The product entity to DTO converter. </param>
0 references | VictorIrranzo, 36 days ago | 1 author, 1 change | 0 exceptions
public ProductsManager(
    ICrudDAO<Product> productCrudDAO,
    EntityDTOTBaseConverter<Product, ProductDTO> productEntityDTOConverter)
    : base(productCrudDAO, productEntityDTOConverter)
{
}
```

Figura 6.17: Constructor de la clase ProductsManager donde se aplica DI.

El uso de interfaces reduce el acoplamiento entre la definición de las operaciones de una clase y las implementaciones concretas que esta definición puede tener. Sin embargo, es necesario indicar con qué implementación se resuelve una interfaz cuando esta se inyecta en el constructor de una clase. Para ellos, se emplean una serie de métodos sobre la colección de servicios donde el sistema busca sus dependencias para indicar cómo resolver cada interfaz. El método más empleado es AddScoped. Este método instancia el objeto de la dependencia una vez por cada petición que el servidor recibe y comparte este mismo objeto entre todas las clases que dependan de él.

En nuestra solución, cada capa es responsable de registrar las diferentes interfaces que contiene junto con la clase que la implementa. De esta forma, desde otras capas se podrán emplear las interfaces resultas.

```
/// <summary> Adds the Persistence service dependencies.
4 references | VictorIrranzo, 14 days ago | 1 author, 14 changes | 0 exceptions
public static IServiceCollection AddPersistenceDependencies(
    this IServiceCollection services,
    IConfiguration configuration = null)
{
    services.AddScoped<IUserDAO, UserDAO>();
    services.AddScoped<IReportsDAO, ReportsDAO>();
    services.AddScoped<ICrudDAO<Product>, ProductsDAO>();
    services.AddScoped<ICrudDAO<Comment>, CommentsDAO>();
    services.AddScoped<ICrudDAO<Order>, OrdersDAO>();
    services.AddScoped<ICrudDAO<Purchase>, PurchasesDAO>();
    services.AddScoped<ICrudDAO<Incidence>, IncidencesDAO>();

    return services;
}
```

Figura 6.18: Registro de interfaces en la capa de persistencia.

¹¹ Inserción de dependencias en ASP.NET Core: <https://docs.microsoft.com/es-es/aspnet/core/fundamentals/dependency-injection>

6.2.7. Documentando la API con Swagger UI

Swagger es un conjunto de herramientas de código abierto para describir la estructura de una API, crear clientes para consumirla en diferentes lenguajes (Swagger CodeGen) y documentarla para que los usuarios puedan emplearla de forma interactiva (Swagger UI).¹²

En este apartado nos centraremos en la construcción de la API interactiva. Para hacerlo, basta con añadir la siguiente pieza de código en la clase Startup de la capa de servicios. Aparte de proveer algunos metadatos como la versión de la API o su mantenedor, se debe indicar que los controladores para generar la documentación se encuentran en el ensamblado actual.

```
app.UseSwaggerUi(typeof(Startup).GetTypeInfo().Assembly, swaggerSettings =>
{
    swaggerSettings.PostProcess = document =>
    {
        document.Info.Version = "1.2.2";
        document.Info.Title = "Shop Monolithic Backend API";
        document.Info.Description = "The API for the Shop Monolithic Backend";
        document.Info.Contact = new NSwag.SwaggerContact
        {
            Name = "Víctor Iranzo",
            Email = "viranzo96@gmail.com",
            Url = "https://github.com/VictorIranzo",
        };
    };
});
```

Figura 6.19: Metadatos de la API especificados en la clase Startup.

La documentación de la API se genera en la dirección formada por la unión de la URL del servidor y el recurso con nombre swagger. Para generar la documentación de los métodos, Swagger hace uso de la documentación del método del controlador de la capa de servicios que la origina. En la documentación de cada método se incluye información como la descripción de los parámetros o el tipo de respuesta que da el método.

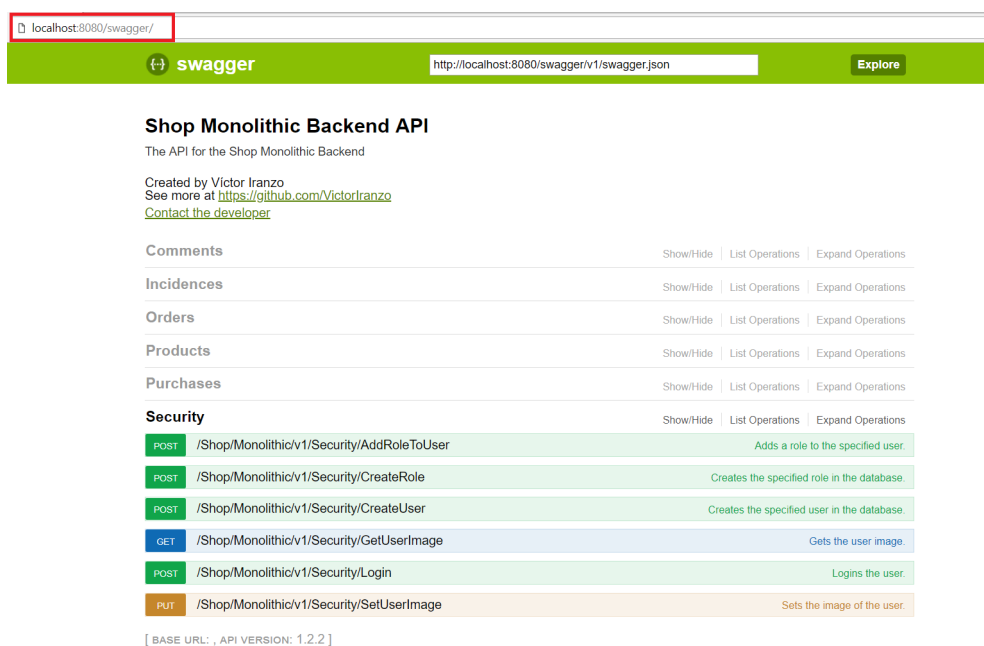


Figura 6.20: Documentación de la API generada con Swagger UI.

¹² Documentación oficial de Swagger: <https://swagger.io/docs/specification/2-0/what-is-swagger/>

6.2.8. Logging

En el entorno de desarrollo puede ser útil emplear logs que capturen todo lo que sucede en el proceso del servidor. Para hacerlo, simplemente se deben añadir las siguientes líneas de código en la clase Startup de la capa de servicios.

```

/// <summary> This method gets called by the runtime. Use this method to configu ...
0 references | VictorIrranzo, 9 days ago | 1 author, 1 change | 0 exceptions
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();

        loggerFactory.AddConsole(this.Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();
    }

    // Other invocations
}

/// <summary> This method gets called by the runtime. Use this method to add ser ...
1 reference | VictorIrranzo, 9 days ago | 1 author, 1 change | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging();

    // Other invocations.
}

```

Figura 6.21: Añadir servicios para el logging.

Información como el entorno donde se ejecuta el servidor, las peticiones HTTP que se realicen al servidor o las consultas a base de datos quedarán registrados en la consola del proceso.

```

hosting environment: Development
Content root path: D:\TFG\TFG_Backend_Monolithic\Services
Now listening on: http://localhost:8080
Application started. Press Ctrl+C to shut down.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
Request starting HTTP/1.1 GET http://localhost:8080/swagger
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
Request finished in 33.702ms 302
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
Request starting HTTP/1.1 GET http://localhost:8080/swagger/index.html?url=/swagger/v1/swagger.js
son
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
Request finished in 91.4806ms 200 text/html; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
Request starting HTTP/1.1 GET http://localhost:8080/swagger/v1/swagger.json
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
Request finished in 1285.7071ms 200 application/json; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
Request starting HTTP/1.1 POST http://localhost:8080/Shop/Monolithic/v1/Security/Login applicati
on/json 69
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
Route matched with {action = "LoginAsync", controller = "Security"}. Executing action Shop.Back
end.Monolithic.ShopMonolithicBackend.Controllers.SecurityController.LoginAsync (ShopMonolithicBackend)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
Executing action method Shop.Backend.Monolithic.ShopMonolithicBackend.Controllers.SecurityContro
ller.LoginAsync (ShopMonolithicBackend) with arguments (Shop.Backend.Monolithic.Contracts.DTOs.Securit
y.EmailAndPasswordDTO)
Validation state: Valid
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 2.1.1-rtm-30846 initialized 'ShopContext' using provider 'Microsoft.Entity
FrameworkCore.SqlServer' with options: MigrationsAssembly=ShopMonolithicBackend
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (1,253ms) [Parameters=[@_normalizedEmail_0='?' (Size = 256)], CommandType='
ext', CommandTimeout='30']
SELECT TOP(1) [u].[Id], [u].[AccessFailedCount], [u].[ConcurrencyStamp], [u].[Email], [u].[Email
Confirmed], [u].[Image], [u].[LockoutEnabled], [u].[LockoutEnd], [u].[NormalizedEmail], [u].[Normaliz
edEmail], [u].[PasswordHash], [u].[PhoneNumber], [u].[PhoneNumberConfirmed], [u].[SecurityStamp], [u
].[TwoFactorEnabled], [u].[UserName]
FROM [AspNetUsers] AS [u]
WHERE [u].[NormalizedEmail] = @_normalizedEmail_0

```

Figura 6.22: Ejemplo del contenido de un log.

6.2.9. Generación de la capa de proxy

Como hemos comentado en la sección **6.1 Diseño de la solución**, la capa de proxy se emplea para invocar a la API de la parte servidora a través de llamadas HTTP. Para

cada una de las interfaces de contratos vamos a crear un proxy, que implementará sus métodos delegando en un cliente HTTP generado automáticamente con **NSwag**. NSwag es un conjunto de herramientas para varias plataformas, entre ellas .NET Core, para la generación de especificaciones Swagger a partir de los controladores definidos en C# y la generación de clientes HTTP para el consumo de estos controladores.^{13 14}

El cliente HTTP autogenerado es inyectado en el proxy a través del constructor. El proxy también es el encargado de adaptar los parámetros de la interfaz de contratos a los parámetros que espera el cliente HTTP. Esta adaptación es necesaria porque, por ejemplo, en las peticiones HTTP para los métodos GET no se puede especificar un Body y todos los parámetros se deben pasar a través de la cabecera de la petición. Como consecuencia, los parámetros solo pueden ser de tipos simples como una cadena (string) y objetos muy empleados como los identificadores (Guid) han de ser transformados a tipos más simples.

```
/// <summary> The orders proxy. </summary>
3 references | VictorIrranzo, 11 days ago | 1 author, 1 change
public class OrdersProxy : IOrdersManager
{
    private OrdersClient ordersClient;

    /// <summary> Initializes a new instance of the OrdersProxy class.
    0 references | VictorIrranzo, 11 days ago | 1 author, 1 change | 0 exceptions
    public OrdersProxy(OrdersClient ordersClient)...

    /// <inheritdoc>
    4 references | VictorIrranzo, 11 days ago | 1 author, 1 change | 0 exceptions
    public async Task<bool> AddPurchasesAndConfirmOrder(AddPurchasesDTO addPurchasesDTO)
    {
        return await this.ordersClient.AddPurchasesAndConfirmOrderAsync(addPurchasesDTO);
    }

    /// <inheritdoc>
    4 references | VictorIrranzo, 11 days ago | 1 author, 1 change | 0 exceptions
    public async Task<byte[]> GenerateOrderReportAsync(Guid orderId, bool pdfFormat = true)
    {
        return await this.ordersClient.GenerateOrderReportAsync(orderId.ToString(), pdfFormat.ToString());
    }
}
```

Figura 6.23: Fragmento del proxy de pedidos donde se adaptan los parámetros a tipos simples.

6.2.10. Calidad del código

Para asegurar la calidad del código se ha hecho uso de dos herramientas: CodeMaid y StyleCop.

- **CodeMaid**: es una extensión de Visual Studio para la limpieza automática de código C# y otros lenguajes. En el proceso de limpieza, CodeMaid ordena los métodos y propiedades de acuerdo a los estándares, revisa el formato de los comentarios, elimina las referencias a espacios de nombres que no se emplean, etc.¹⁵
- **StyleCop**: es un analizador estático de código C# desarrollado por Microsoft. Se basa en reglas centradas en aspectos como la documentación, la ordenación de los elementos o el estilo de código. Además, permite configurar las reglas que se aplican, definir excepciones a una de ellas o la creación de reglas propias.¹⁶

En la mayoría de proyectos se ha configurado para que las alertas que devuelve StyleCop cuando no se cumple una regla se trate como un error y no como una

¹³ Página de GitHub de NSwag: <https://github.com/RSuter/NSwag>

¹⁴ NSwag Tutorial: How to integrate NSwag into your ASP.NET Core Web API project: <https://www.youtube.com/watch?v=1F9ZZ8p2Ciw>

¹⁵ Página oficial de CodeMaid: <http://www.codemaid.net/>

¹⁶ Página de la Wikipedia de StyleCop: <https://en.wikipedia.org/wiki/StyleCop>

alerta. Esto se puede conseguir añadiendo la opción `TreatWarningsAsErrors` en los archivos `.csproj`.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <TreatWarningsAsErrors>True</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <AdditionalFiles Include="..\stylecop.json" Link="stylecop.json" />
  </ItemGroup>
</Project>
```

Figura 6.24: Fragmento de un proyecto donde se añade la opción `TreatWarningsAsErrors`.

6.3 Interfaz de usuario

Como hemos comentado en la sección 5.1 **Plan de trabajo**, vamos a implementar una interfaz de usuario que cumpla con los casos de uso y se pueda emplear para comunicar tanto con la solución monolítica como con la basada en microservicios. Para desacoplar el desarrollo del back-end y el front-end, la aplicación móvil va a desarrollarse en un repositorio de código distinto.

La solución de la UI está compuesta por dos proyectos. El primero es en el que se localiza la mayoría del código que se ha implementado, un proyecto .NET Standard con el código compartido por todas las plataformas. El segundo es el proyecto específico para la plataforma Android. Los proyectos asociados a las plataformas UWP y iOS han sido eliminados para hacer más simple su desarrollo.

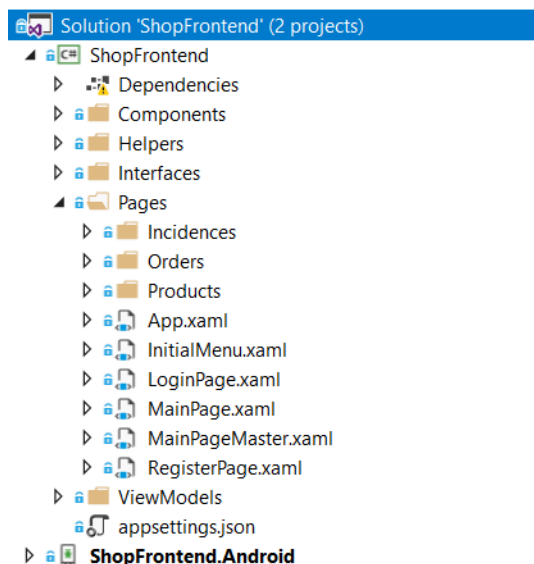


Figura 6.25: Solución de la interfaz de usuario hecha con Xamarin.Forms.

Podemos citar las siguientes características que se desean poner en valor de la implementación de la interfaz de usuario:

- **Presentación de resultados de forma paginada:** en el back-end se han implementado diferentes métodos para obtener resultados (como la lista de incidencias de un

usuario o la lista de productos) de forma paginada, ordenada y a través de filtros. La paginación en las invocaciones a una API son una forma de mejorar el rendimiento de una aplicación porque evita traer más resultados de los que luego se consultan. El filtrado y la ordenación de los datos están más centrados en mejorar la experiencia del usuario (UX).¹⁷

A nivel de UI se ha implementado de tal forma que se soliciten al back-end una cantidad de datos aproximada a la que se puede mostrar en la pantalla de un dispositivo. Cuando el usuario haga scroll para mostrar más resultados, se realizará una nueva petición en segundo plano a la parte servidora solicitando el siguiente bloque de datos. Mientras se cargan los datos, el usuario visualizará un elemento que indica que el sistema está trabajando en segundo plano para traer más datos.

- **Uso de ViewModels:** la separación de responsabilidades no es un principio que se deba aplicar solo en el back-end. El patrón arquitectónico **Model-View-ViewModel** (MVVM) divide la interfaz de usuario en 3 capas: la vista, empleando páginas XAML, los datos tal como se obtienen en su origen, también llamados el modelo, y el modelo de la vista, que conecta ambos y se emplea para rellenar la vista.¹⁸

La diferencia entre este patrón y otros como el Model-View-Controller(MVC) es que en MVC los datos con los que se llena la vista son los que se obtienen en el origen y es el controlador quien ha de adaptarlos a cómo se visualizan en la vista. En cambio, en MVVM la vista y el modelo de la vista suelen relacionarse a través de enlaces (bindings) en el propio XAML y se notifican mutuamente cuando alguna de sus propiedades cambia.¹⁹

Sin embargo, usar este patrón puede ser excesivo para UIs muy sencillas.²⁰ Por este motivo, hemos empleado objetos ViewModel en solo aquellos casos que era necesario adaptar los datos del modelo a la vista. Lo que aquí definimos como modelo es un DTO que devuelve la parte servidora y ya está diseñado para que contenga solo la información estrictamente necesaria. Un ejemplo en el que se emplea un ViewModel es el ProductViewModel, donde se tiene que transformar la imagen que se recibe del ProductDTO de byte[] a un ImageSource.

```
/// <summary> Creates a <see cref="ProductViewModel"/> from a <see cref="ProductDTO"/>. </summary>
/// <param name="productDTO"> The product DTO. </param>
/// <returns> The product view model obtained from the producto DTO. </returns>
1 reference | 0 changes | 0 authors, 0 changes
public static ProductViewModel CreateProductViewModelFromProductDTO(ProductDTO productDTO)
{
    return new ProductViewModel()
    {
        Id = productDTO.Id,
        Description = productDTO.Description,
        Name = productDTO.Name,
        Price = productDTO.Price,
        Stock = productDTO.Stock,
        Image = ImageSource.FromStream(() => new MemoryStream(productDTO.Image)),
    };
}
```

Figura 6.26: Método para transformar de un DTO a un modelo de una vista.

¹⁷ REST API Design: Filtering, Sorting, and Pagination: <https://www.moesif.com/blog/technical/api-design/REST-API-Design-Filtering-Sorting-and-Pagination/>

¹⁸ From Data Bindings to MVVM: <https://docs.microsoft.com/es-es/xamarin/xamarin-forms/xaml/xaml-basics/data-bindings-to-mvvm>

¹⁹ Patterns - WPF Apps With The Model-View-ViewModel Design Pattern: <https://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

²⁰ Advantages and disadvantages of M-V-VM: <https://blogs.msdn.microsoft.com/johngossman/2006/03/04/advantages-and-disadvantages-of-m-v-vm/>

- **Código específico de plataforma:** existen algunos servicios que no se pueden implementar en el proyecto compartido por todas las plataformas. Estos servicios están relacionados con aspectos muy ligados a cada plataforma, como puede ser el sistema de archivos. Para este propósito se define una interfaz en el proyecto compartido. Cada una de las plataformas debe dar una implementación concreta a esta interfaz. Desde el proyecto genérico, se hace uso de la interfaz y será el sistema quien se encargará de dirigir la operación que se invoca sobre la interfaz a la implementación de la plataforma.²¹

```

/// <summary>
///     The implementation of the <see cref="IFileManager"/> interface
///     for the Android platform.
/// </summary>
public class AndroidFileManager : IFileManager
{
    /// <inheritdoc/>
    public string GetPathFile()
    {
        return Path.Combine(
            Android.OS.Environment.ExternalStorageDirectory.AbsolutePath,
            Android.OS.Environment.DirectoryDownloads);
    }

    /// <inheritdoc/>
    public void OpenPdfFile(string path)
    {
        Android.Net.Uri uri = Android.Net.Uri.Parse("content://" + path);
        Intent intent = new Intent(Intent.ActionView);
        intent.SetDataAndType(uri, "application/pdf");
        intent.SetFlags(ActivityFlags.ClearWhenTaskReset | ActivityFlags.NewTask);
    }
}

```

Figura 6.27: Servicio concreto del proyecto de Android.

Como material adicional, en el apéndice **A Descripción del prototipo de IU desarrollado** se incluyen capturas de la aplicación desarrollada y la navegación que existe entre las diferentes pantallas.

6.4 Pruebas

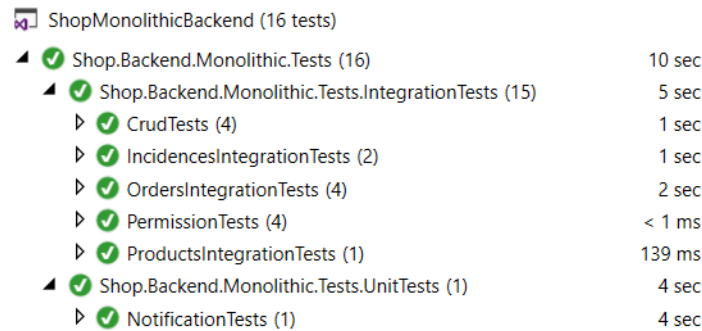
Se han realizado un total de 16 pruebas automatizadas mediante el framework **NUnit**. NUnit es una librería para la implementación de pruebas unitarias en todos los lenguajes .NET al igual que JUnit lo es para el lenguaje Java.²² La mayoría de las pruebas realizadas se clasifican como pruebas de integración porque involucran a más de una clase.

No se han podido probar todos los métodos expuestos en la interfaz del back-end, pero gracias a que existe mucho código genérico para la realización de operaciones CRUD y la persistencia de datos, podemos considerar el sistema como fiable. Algunas de las pruebas que si se han hecho son las asociadas a los casos de uso CU003, CU004, CU005, CU007, CU009 y CU013 que se encuentran en la sección **4.2 Casos de uso y modelo de dominio**.

Para la realización de las pruebas se ha empleado una base de datos en memoria. Así, las pruebas son más cercanas a una situación real. A la vez, no añade un sobrecoste asociado al acceso a datos de una base de datos real, que haría más lenta la ejecución de las

²¹Native Services with Xamarin.Forms? It's DependencyService to the Rescue: <https://visualstudiomagazine.com/articles/2015/09/01/native-services-with-xamarinforms.aspx>

²² Página de GitHub de NUnit: <https://github.com/nunit/nunit>



| | |
|---|--------|
| ShopMonolithicBackend (16 tests) | |
| Shop.Backend.Monolithic.Tests (16) | 10 sec |
| Shop.Backend.Monolithic.Tests.IntegrationTests (15) | 5 sec |
| CrudTests (4) | 1 sec |
| IncidencesIntegrationTests (2) | 1 sec |
| OrdersIntegrationTests (4) | 2 sec |
| PermissionTests (4) | < 1 ms |
| ProductsIntegrationTests (1) | 139 ms |
| Shop.Backend.Monolithic.Tests.UnitTests (1) | 4 sec |
| NotificationTests (1) | 4 sec |

Figura 6.28: Pruebas realizadas en la solución monolítica.

pruebas.²³ Además, en el método Setup que se invoca antes de cada prueba es donde se resuelven las dependencias declaradas en los constructores de las clases invocando a los métodos AddDependencies() de las capas de aplicación y persistencia. Por último, para algunos servicios provistos por las herramientas utilizadas se han tenido que implementar fakes. Es el caso del manager de Identity para hacer login en la aplicación, que se debe sobrescribir para devolver que el usuario ha hecho login correctamente sin acceder a la base de datos de usuarios.

```

/// <summary> Setups every test executed. </summary>
[Setup]
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
public void Setup()
{
    IServiceCollection services = new ServiceCollection();

    services.AddDbContext<ShopContext>(options =>
    {
        options.UseInMemoryDatabase(databaseName: "ShopContext");
    });

    services.AddPersistenceDependencies();
    services.AddApplicationDependencies();

    services.AddScoped<SignInManager<User>, FakeSignInManager>();

    IServiceProvider serviceProvider = services.BuildServiceProvider();

    this.productsManager = serviceProvider.GetService<ProductsManager>();
}

```

Figura 6.29: Ejemplo de método Setup donde se emplea una base de datos en memoria.

6.5 Despliegue

Como hemos mencionado en la sección 3.5 Propuesta, para el despliegue de la aplicación se va a emplear Azure App Service. No se va a implementar ninguna pipeline para desplegar automáticamente y todos los pasos que se detallan se deben hacer manualmente, aunque sean muy sencillos:

1. **Creación de un Dockerfile:** en la capa de servicios se debe crear un Dockerfile. En la mayoría de ejemplos encontrados se ha visto que la imagen de Docker se genera básicamente mediante dos pasos del Dockerfile: se copian todos los archivos

²³ Pruebas con InMemory: <https://docs.microsoft.com/es-es/ef/core/miscellaneous/testing/in-memory>

de la solución a la imagen y se compila la solución completo. Cuando se crea y arranca un contenedor, se inicia el proceso del servicio a través de la DLL que se ha generado tras la compilación.²⁴ Este Dockerfile es más fiable en cuanto a que los ensamblados necesarios se generan dentro de la propia imagen. Sin embargo, la creación de la imagen es más lenta. Por ello, para entornos de desarrollo vamos a simplificar el Dockerfile para que simplemente copie los ensamblados, que se han generado previamente en el entorno de desarrollo, de aquí a la imagen. Los ensamblados se publican dentro de una carpeta que hemos llamado PublishOutput. En el Dockerfile tampoco hace falta exponer ningún puerto porque esto ya lo hace la imagen base.

```
FROM microsoft/dotnet:2.1-sdk AS base

WORKDIR /app

COPY PublishOutput .

ENTRYPOINT ["dotnet", "ShopMonolithicBackend.dll"]
```

Figura 6.30: Dockerfile de la solución monolítica.

2. **Crear un App Service:** en el portal de Azure, seleccionar la pestaña de nuevo recurso y marcar Web App. Se debe proveer el nombre de la aplicación, que será la URL donde luego encontraremos nuestro servicio, la suscripción y el grupo de recursos. También se debe crear un plan de App Service, donde se establecerán las prestaciones del servidor donde se desplegará y las tarifas asociadas.
3. **Obtener perfil de publicación:** una vez creado el recurso nos podemos descargar el perfil de publicación. Desde el panel del recurso podemos iniciar y detener el servicio.

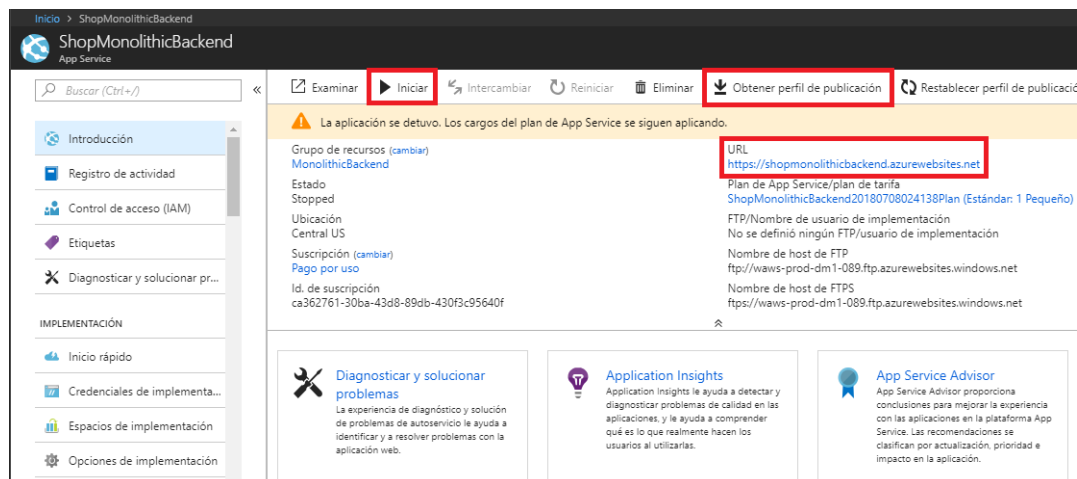


Figura 6.31: Recurso App Service en el portal de Azure.

4. **Importar perfil de publicación:** desde la solución de Visual Studio, abrir el menú contextual del proyecto de la capa de servicios y seleccionar la opción "Publish". En la ventana que se abre seleccionar "New profilez luego "Import profile", donde seleccionaremos el archivo que nos hemos descargado.

²⁴ Dockerize a .NET Core application : <https://docs.docker.com/engine/examples/dotnetcore/#create-a-dockerfile-for-an-aspnet-core-application>

5. **Desplegar el servicio:** cada vez que queramos desplegar a producción una nueva versión de la API nos iremos al proyecto de servicios, abriremos el menú de publicación a través del menú contextual y seleccionaremos el perfil de AppService. Si accedemos a la URL que antes hemos señalado, veremos la documentación de la API generada por Swagger UI.

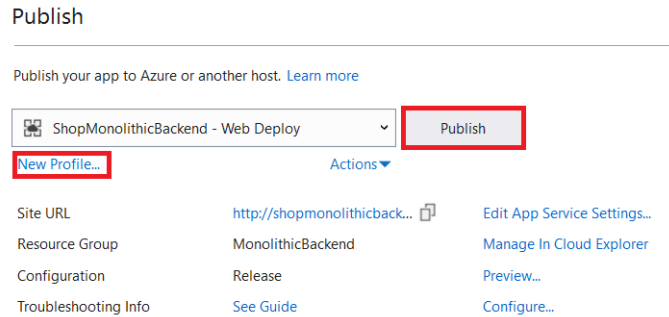


Figura 6.32: Despliegue a través de App Service.

CAPÍTULO 7

Diseño e implementación de la solución basada en microservicios

A lo largo del capítulo anterior nos hemos centrado sobretodo en aspectos de implementación relacionados con diferentes herramientas como Identity o Swagger. En este capítulo nos vamos a centrar más en el diseño ya que, como hemos dicho en el apartado [5.1 Plan de trabajo](#), vamos a refactorizar la solución monolítica y a nivel de código apenas va a verse modificada.

7.1 Diseño de la solución

Para la descomposición de la solución en microservicios vamos a aplicar los principios que hemos explicado en el apartado [2.3 Los microservicios en la fase de diseño](#). Se tienen que extraer contextos bien delimitados sin importar en un primer momento el tamaño de estos. Aunque contemos con la experiencia del desarrollo de la solución monolítica, el tamaño de los microservicios se puede ajustar más adelante evaluando si vale la pena dividir un microservicio.

Concretamente, para representar visualmente la división en contextos delimitados se va emplear el modelo de dominio del caso de estudio de forma similar a como se ha hecho en la figura [2.3](#). A continuación, detallamos cada uno de ellos, centrándonos en las capacidades que ofrece al negocio y no tanto en las entidades que maneja:

- **Incidencias:** permite la creación de incidencias, obtener las incidencias de un cliente y añadir comentarios dentro de una incidencia.
- **Seguridad:** ofrece las funcionalidades de registro de nuevos clientes y login. Además, persiste los datos de los clientes, por lo que algunos de los microservicios tendrán una referencia a este, por ejemplo, para obtener el cliente o empleado que escribió un comentario.
- **Informes:** es un motor que almacena las plantillas de los informes que se pueden generar y simplemente combina esta con los datos que recibe para generar un documento de salida. No ofrece directamente ninguna funcionalidad al cliente: se trata de un **microservicio interno** que otros servicios emplean.
- **Notificaciones:** al igual que el microservicio de informes, no está ligado a ningún tipo de dato. Simplemente, ofrece la funcionalidad de enviar una notificación con el contenido que recibe como parámetro al destinatario especificado. También es un microservicio interno.

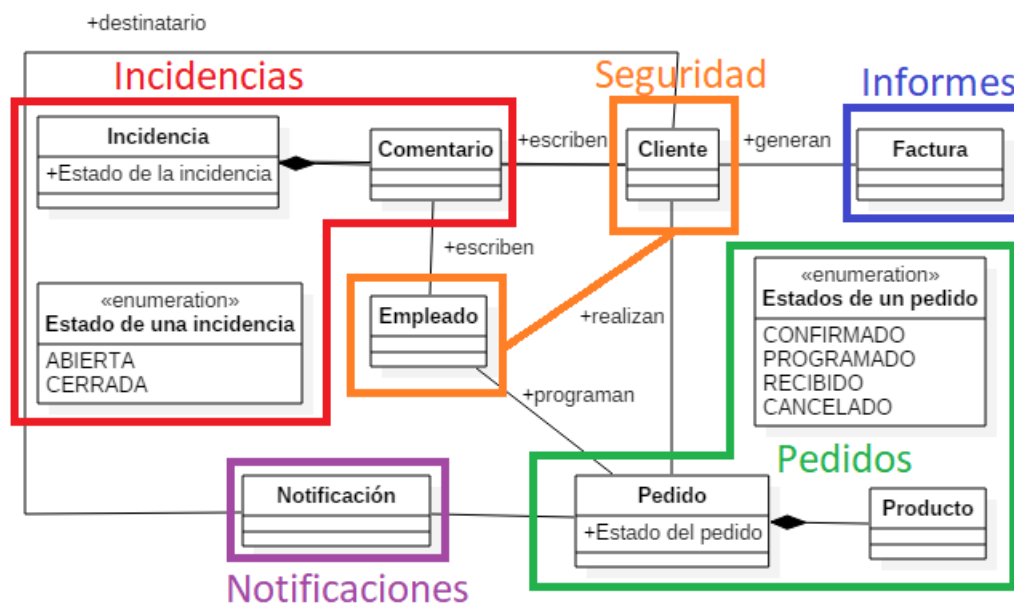


Figura 7.1: División del modelo de dominio en contextos delimitados.

- **Pedidos:** permite obtener los pedidos de un cliente, generar una factura de un pedido, añadir productos...En este servicio se ha incluido la entidad Producto. Podríamos considerar un microservicio de catálogo al que solo tuvieran acceso los empleados del comercio para gestionar el número de unidades en stock o la creación de nuevos productos. Sin embargo, ninguna funcionalidad fuera del contexto de los pedidos se relaciona con los productos, por lo que la incluiremos en este servicio y más adelante evaluaremos si vale la pena representarla como un microservicio independiente.

7.1.1. Arquitectura interna de los microservicios

En el apartado **2.4.2 Programación y persistencia políglotas** se ha reflexionado sobre que una arquitectura basada en microservicios permite emplear diferentes tecnologías, arquitecturas y bases de datos en cada microservicio. Para validar lo que en este apartado se dice, vamos a desarrollar algunos microservicios con diferentes combinaciones de estas 3 características.

En la mayoría de microservicios vamos a seguir la arquitectura de 6 capas que se describe en la sección **6.1 Diseño de la solución**. Esto hará que su refactorización sea más sencilla ya que sólo debemos copiar las clases relacionadas al microservicio en cada una de las capas.

Cada microservicio tendrá su propia base de datos, aunque estas pueden localizarse dentro del mismo servidor. Para evaluar otra tecnología de base de datos, la BD del servicio de incidencias la vamos a implementar con Firebase.

Las necesidades de cada microservicio de nuestro caso de estudio son diferentes: algunos como el de informes o notificaciones no requieren persistir datos, por lo que aplicar una arquitectura de 6 capas en ellos no es necesario.

En el microservicio de informes vamos a seguir una arquitectura más sencilla. Su lógica se situará directamente en la capa de servicios, eliminando la capa de aplicación. Las capas de dominio y persistencia también pueden ser eliminadas porque las plantillas de los informes se van a almacenar como un recurso y no en una base de datos.

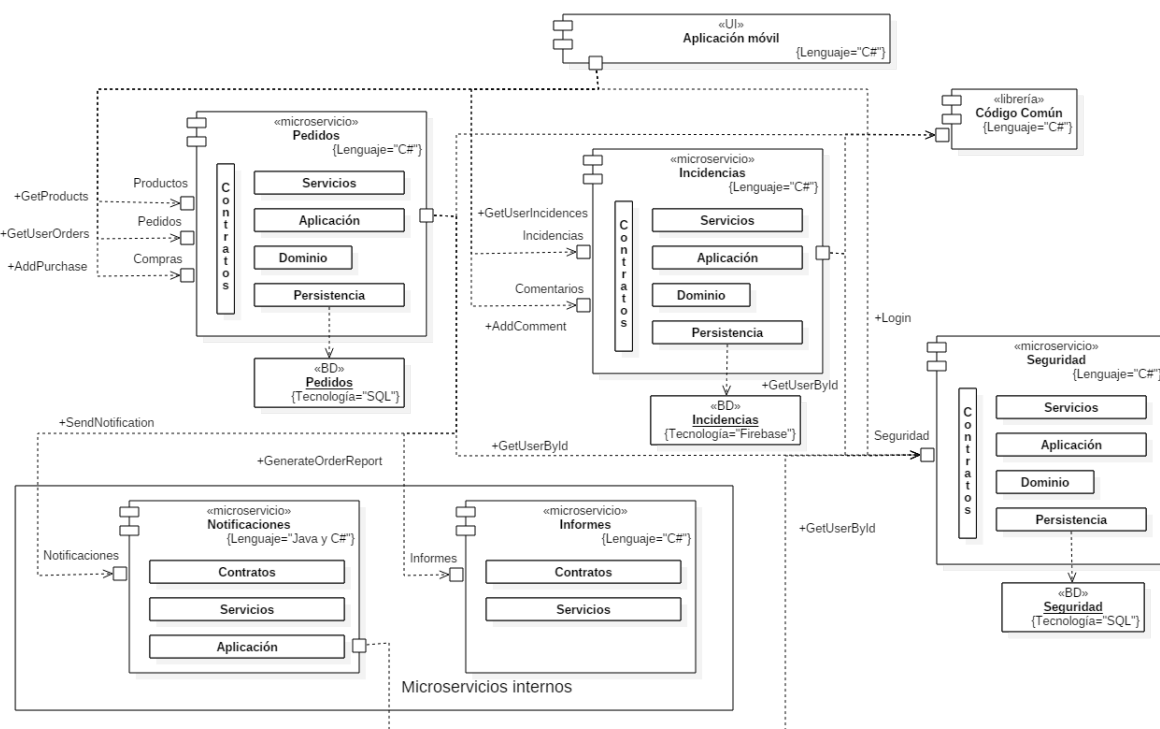


Figura 7.2: Diagrama de componentes de la solución basada en microservicios.

En cuanto al servicio de notificaciones, vamos a implementarlo en el lenguaje Java. Sin embargo, como debe ser consumido por microservicios en lenguaje C#, construiremos un proxy en este lenguaje para hacer más fácil su consumo.

En el capítulo anterior, en las secciones **6.2.1 Operaciones CRUD** y **6.2.3 Persistencia** hemos nombrado algunas interfaces e implementaciones genéricas relacionadas con las entidades de dominio. Los microservicios de pedidos e incidencias deben seguir haciendo uso de estas implementaciones. En el apartado **2.3.1 Librerías versus servicios** hemos reflexionado sobre las formas de realizar componentes software. Hacer de este código común un servicio no tiene sentido porque está relacionado con el acceso a datos y cada microservicio es dueño de sus propios datos. En este caso, este código genérico lo referenciarán como una librería a través de paquetes NuGet.

Por último, indicar que el consumo de cualquier microservicio, ya sea desde la interfaz de usuario o desde otros servicios, se realizará a través de las capas de proxy. En el diagrama de componentes no se ha incluido esta capa en la arquitectura interna de ninguno de los microservicios porque esta capa no se ejecuta en el proceso del servicio.

7.1.2. Organización de los microservicios

En cuanto a la organización de los microservicios, vamos a emplear un único repositorio de código que contenga el código de todos los microservicios. Como en la solución monolítica, cada capa se representará a través de un proyecto .NET y cada microservicio tendrá una solución de VS distinta.

En "Building Microservices", Newman reflexiona sobre esto en relación a las compilaciones de integración continua. Según él, la mejor opción es tener un repositorio y una compilación diferente para cada microservicio. De esta forma, existe una mayor correspondencia entre los servicios y los equipos encargados de cada uno porque cada equipo

realiza cambios en un único repositorio. Además, se evita que un simple cambio en un microservicio lance la compilación de toda la solución. [?] En nuestro caso, no vamos a aplicar las prácticas de integración continua y tampoco tenemos equipos de trabajo distintos, por lo que será más cómodo emplear un único repositorio. Como consecuencia, esto implica que en un mismo commit (la unidad de los repositorios de código git que representa los cambios que se guardan en el repositorio) se puedan incluir cambios de diferentes microservicios.

En cuanto a la organización en soluciones en Visual Studio, en "Microservicios .NET: Arquitectura para Aplicaciones .NET Contenerizadas" se organiza el código de la aplicación desarrollada en un único repositorio con una única solución. [7] Creemos que esta aproximación no es la mejor. Si consideramos una solución de Visual Studio como una vista de un repositorio de código, no tiene sentido que un desarrollador que trabaja en un único repositorio vea en su solución otros microservicios distintos al suyo. Además, incluir tantos archivos en la solución aumenta su tiempo de compilación, cuando jamás se van a realizar cambios o ejecutar algunos de ellos. Por ello, hemos optado por una solución diferente para cada servicio.

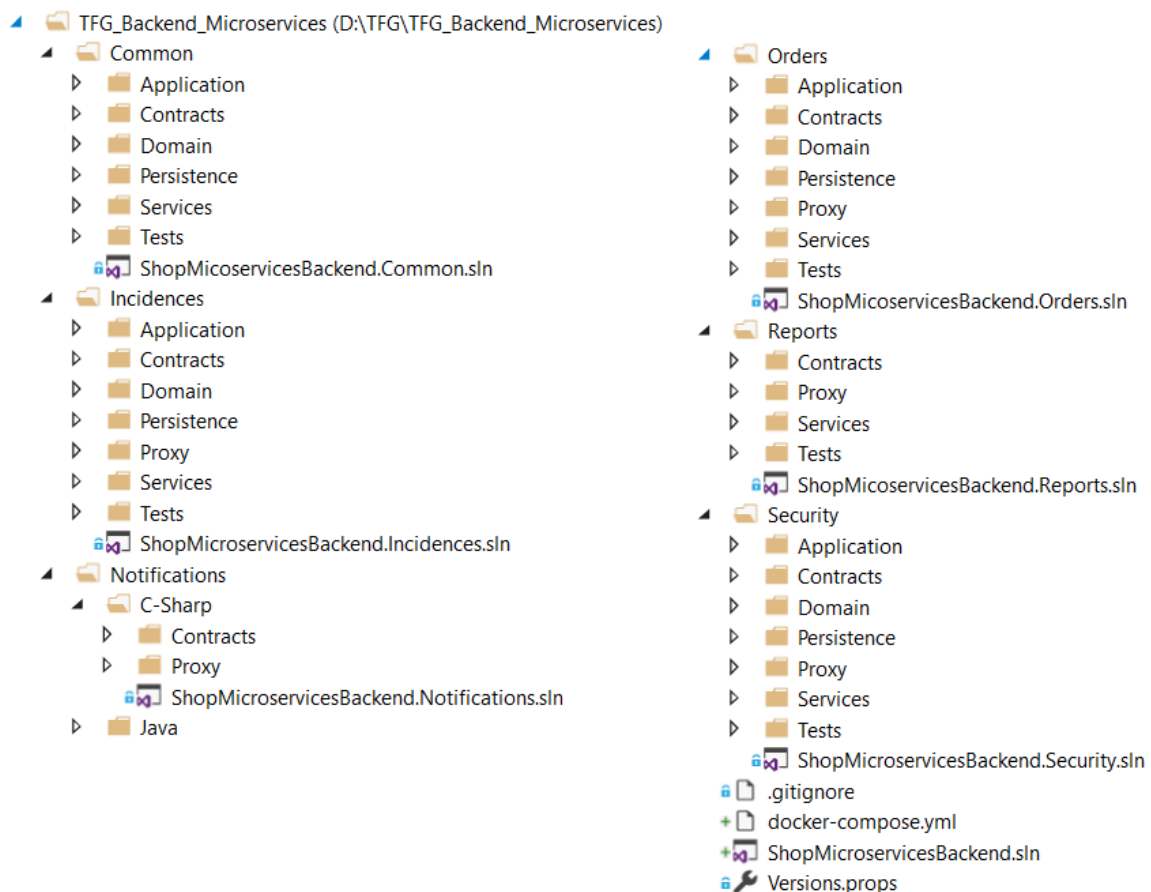


Figura 7.3: Organización de la solución basada en microservicios.

7.2 Diferencias en la implementación respecto a la solución monolítica

7.2.1. Consumo de otros microservicios

Para que un microservicio pueda hacer peticiones a otro se deben seguir los siguientes pasos:

1. Instalar el paquete NuGet del microservicio a consumir en la capa de aplicación.

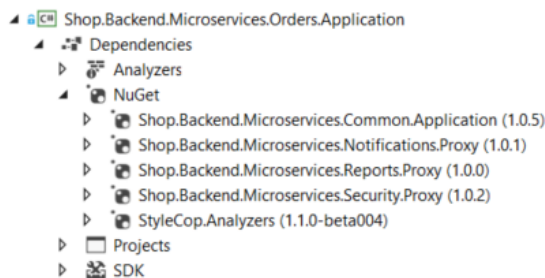


Figura 7.4: Dependencias del microservicio de pedidos en la capa de aplicación.

2. Al registrar las dependencias de la capa de aplicación, invocar al código de la capa de proxy donde se registran las interfaces del microservicio a consumir. En la capa de proxy, las interfaces de la capa de contratos se resuelven con un proxy. Esto significa que en un microservicio cuando hagamos una petición a otro servicio a través de la interfaz se realizará una llamada HTTP a través de su proxy.

```
if (addProxies)
{
    services.AddSecurityProxy(configuration);
    services.RegisterSecurityClient(new HttpClient(), configuration);

    services.AddReportsProxy(configuration);
    services.RegisterReportsClients(new HttpClient(), configuration);

    services.AddNotificationsProxy(configuration);
    services.RegisterNotificationsClients(new HttpClient(), configuration);
}

return services;
```

Figura 7.5: Código para resolver otros microservicios consumidos.

3. En el microservicio, inyectar en el constructor la interfaz de contratos del servicio que se desea consumir.

7.2.2. Consistencia eventual

Cada servicio es soberano de sus datos y la mejor manera de conseguir esto es separar los datos de cada uno en diferentes bases de datos. Una única base de datos relacional para toda la aplicación, tal y como se usa en la solución monolítica, tiene dos ventajas principalmente: se pueden emplear transacciones atómicas y restricciones de integridad referencial.

No se puede realizar una transacción única que involucre a diferentes microservicios, sobretodo porque cada uno puede utilizar una tecnología de BD diferente. Las transacciones se deben implementar en la capa de aplicación y en caso de que una operación falle,

se deben establecer mecanismos de compensación que reviertan los cambios hechos hasta ese punto. [7]

Por suerte, las únicas referencias en los datos que hay entre diferentes servicios son las que hay desde incidencias y pedidos al servicio de seguridad. Tanto los pedidos, las incidencias y los comentarios están asociados a un cliente. En caso de que se borre un cliente, en cascada se deben borrar las instancias de estas entidades que tenga asociadas como si de una transacción se tratara. La operación de eliminar un cliente no está soportada por la API, por lo que no ha sido necesario implementar ninguna transacción entre microservicios.

En caso de que estuviera soportada, no se puede realizar de forma atómica y la consistencia sería eventual. Un usuario podría ver un estado inconsistente del sistema si se ha eliminado un cliente en el microservicio de seguridad pero sus pedidos todavía no en el microservicio de pedidos. Se denomina eventual porque en algún punto próximo del tiempo la transacción que se ha iniciado entre diferentes microservicios para eliminar a un usuario se completará en todo el sistema.

En cuanto a las restricciones de integridad referencial, en una base de datos relacional se modela a través de claves externas (foreign key, FK). Dentro de la base de datos de un servicio podemos emplear FKs, como ocurre en las relaciones entre pedido y compra. Sin embargo, en las relaciones que hemos comentado con el módulo de seguridad, no se pueden emplear como tal FKs porque se referenciaría a otra base de datos. En su lugar, se almacena únicamente el identificador de la entidad referenciada y cuando se quiere acceder a ella se debe hacer una petición al servicio que la gestiona. Esto se refleja en las entidades de la capa de dominio, como se ve en la figura 7.6.

| | |
|---|---|
| <pre> /// <summary> The comment entity. </summary> 28 references VictorIrranzo, 40 days ago 1 author, 1 change public class Comment : Entity { /// <summary> Gets or sets the author. 7 references VictorIrranzo, 40 days ago 1 author, 1 change 0 exceptions public User Author { get; set; } /// <summary> Gets or sets the date time. 4 references VictorIrranzo, 40 days ago 1 author, 1 change 0 exceptions public DateTime DateTime { get; set; } /// <summary> Gets or sets the incidence. 3 references VictorIrranzo, 40 days ago 1 author, 1 change 0 exceptions public Incidence Incidence { get; set; } </pre> | <pre> /// <summary> The comment entity. </summary> 42 references VictorIrranzo, 18 days ago 1 author, 1 change public class Comment : Entity { /// <summary> Gets or sets the author. 6 references VictorIrranzo, 18 days ago 1 author, 1 change 0 exceptions public Guid Author { get; set; } /// <summary> Gets or sets the date time. 3 references VictorIrranzo, 18 days ago 1 author, 1 change 0 exceptions public DateTime DateTime { get; set; } /// <summary> Gets or sets the incidence. 7 references VictorIrranzo, 18 days ago 1 author, 1 change 0 exceptions public Incidence Incidence { get; set; } </pre> |
|---|---|

Figura 7.6: Clases de dominio de la entidad comentario en la solución monolítica (derecha) y la basada en microservicios (izquierda).

7.2.3. Microservicio de notificaciones

El microservicio de notificaciones ahora está desarrollado en dos lenguajes de programación distintos: Java y C#:

En C# está desarrollado todo lo necesario para el consumo del servicio. Esto incluye la capa de contratos (donde se define su interfaz y el DTO que se empleará en la parte C#) y la de proxy (donde se indica que la interfaz de contratos se resuelve a través del proxy, que recurre al cliente HTTP para realizar peticiones al microservicio).

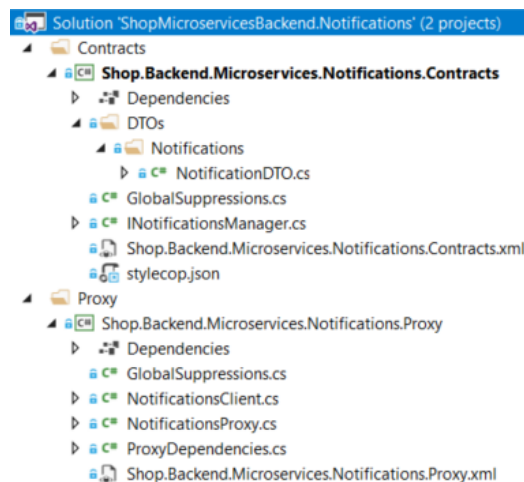


Figura 7.7: Parte del microservicio de notificaciones desarrollada en C#.

En Java se han desarrollado la capa de aplicación y servicios. La implementación de la capa de aplicación para mandar un correo electrónico es trivial. Para la capa de servicios, se crea e inicia un objeto `HttpServer` al que se le asigna un handler.

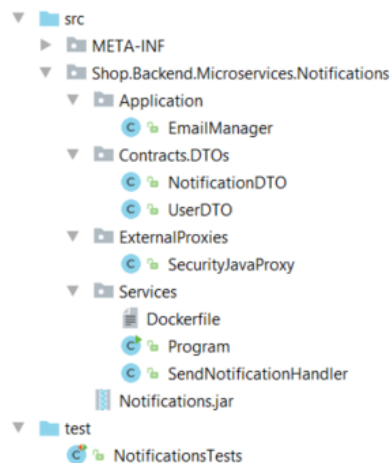


Figura 7.8: Parte del microservicio de notificaciones desarrollada en Java.

El handler es similar a lo que en .NET llamábamos controlador. En nuestro caso, deserializará el cuerpo (body) de la petición en un objeto empleando la librería `Gson` y delegará en la capa de aplicación la operación a realizar. Cuando se quiere consumir otro microservicio no podemos emplear los proxies que hemos creado para C# porque es otro lenguaje de programación. En nuestro ejemplo, queremos contactar con el servicio de seguridad para obtener el correo electrónico de un usuario. Debemos crear un objeto para conectar a través del protocolo HTTP con el microservicio de seguridad y leer la respuesta obtenida.

```

public class SecurityJavaProxy {
    public String getUserEmail(String userId, String authenticationToken) throws IOException {
        URL url = new URL( spec "http://localhost:8085/Shop/Microservices/v1/Security/GetUserById");

        HttpURLConnection connection = (HttpURLConnection)url.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setRequestProperty("Authorization", authenticationToken);
        connection.setRequestProperty("userId", userId);
        connection.connect();

        int code = connection.getResponseCode();

        if (code != 200) return null;

        Reader reader = new InputStreamReader(connection.getInputStream());
        Gson gson = new Gson();

        Type type = new UserDTO().getClass();

        UserDTO result = gson.fromJson(reader, type);

        return result.getEmail();
    }
}

```

Figura 7.9: Proxy del microservicio de seguridad para su consumo en Java.

Por último, el archivo Dockerfile para el despliegue del servicio también varía. Vamos a seguir el mismo criterio que en los contenedores de microservicios .NET de usar los ensamblados (en este caso, un archivo *.jar) para generar la imagen con el menor número de recursos posibles. Tanto el comando para iniciar el proceso del contenedor como la imagen base se verán afectados.

```

FROM java:8

WORKDIR /app

EXPOSE 9000

COPY Notifications.jar .

CMD java -jar Notifications.jar

```

Figura 7.10: Dockerfile del microservicio de notificaciones.

7.2.4. Persistencia en microservicio de incidencias

En el microservicio de incidencias vamos a emplear una base de datos en la plataforma Firebase. **Firebase** es una plataforma para el desarrollo de aplicaciones. Uno de los servicios que ofrece es una base de datos NoSQL en la nube en tiempo real. Los datos se almacenan en formato JSON y su acceso puede realizarse a través de peticiones HTTP.¹

Una de las principales ventajas de las bases de datos NoSQL es su escalabilidad horizontal. En lugar de tener un gran servidor para alojar una base de datos relacional, una base de datos NoSQL se puede distribuir entre diferentes máquinas. Esta funcionalidad en Firebase es de pago, por lo que no vamos a obtener grandes beneficios por emplear esta BD. Nuestro objetivo simplemente es validar que diferentes microservicios pueden emplear tecnologías de bases de datos distintas.



Figura 7.11: Base de datos de incidencias en Firebase.

Gracias a la arquitectura interna basada en capas, el uso de esta BD solo implica dejar de usar la implementación que se da por defecto al DAO en la librería común y programar una implementación propia que acceda a los datos a través de llamadas HTTP.

¹ Documentación de Firebase Realtime Database: <https://firebase.google.com/docs/database/>

```

/// <summary> The Incidences Data Access Object.
1 reference | VictorIrranzo, 14 days ago | 1 author, 2 changes
public class IncidencesDAO : ICrudDAO<Incidence>
{
    private readonly string baseUrl = "https://tfg-microservices-incidences.firebaseio.com/";

    /// <inheritdoc>
    1 reference | VictorIrranzo, 14 days ago | 1 author, 2 changes | 0 exceptions
    public Incidence GetById(Guid identifier)
    {
        FirebaseBaseContext firebaseContext = new FirebaseBaseContext(this.baseUrl);

        firebaseContext = firebaseContext.NodePath("incidences/" + identifier.ToString());

        FirebaseResponse getResponse = firebaseContext.Get();

        if (getResponse.JSONContent.Equals("null"))
        {
            return null;
        }

        return JsonConvert.DeserializeObject<Incidence>(getResponse.JSONContent);
    }
}

```

Figura 7.12: Fragmento de DAO que accede a una base de datos de Firebase.

7.3 Versionado de servicios

Cada uno de los microservicios se va a desplegar de forma independiente. El consumo de un microservicio se realiza a través de su proxy, que es una implementación de una interfaz de la capa de contratos. Cuando se realiza y se despliega un cambio en un servicio pueden ocurrir dos situaciones respecto a sus consumidores:

- La primera situación es que el cambio **no haya supuesto un cambio en la interfaz** de la capa de contratos. En este caso, se habrá realizado un cambio a nivel de la lógica del servicio o su persistencia, como puede ser la corrección de un defecto. El cambio realizado no necesita ser conocido por sus consumidores porque están desacoplados de la implementación específica del servicio.

Los consumidores pueden seguir consumiendo la misma interfaz del servicio. Solo en caso de que se haya modificado el comportamiento del servicio (su especificación) los consumidores deberán ser notificados para adaptarse a cambios en las respuestas del servidor. Por ejemplo, se puede haber cambiado la política de permisos de un método sin haber cambiado su interfaz y ahora devuelve un error 401:Unauthorized porque es necesario otro rol para invocarlo.

- Una segunda situación consiste en que el cambio **haya significado un cambio en la interfaz** de contratos. Por ejemplo, puede ocurrir que se haya añadido un nuevo parámetro a un método o se haya cambiado el DTO de respuesta de un método. En este caso, los consumidores deben actualizar el proxy que emplean para comunicar con la nueva versión del servicio. Para asegurar que ningún consumidor se rompa por la actualización se pueden seguir dos aproximaciones según la literatura:

- **Mantener la vieja interfaz y redireccionar internamente las peticiones que recibe a la nueva interfaz.** Ambas interfaces están disponibles hasta asegurarse de que ningún cliente está empleando la interfaz vieja, lo que requiere de la monitorización del servicio para saber cuántos clientes la emplean todavía.
- **Tener diferentes versiones del servicio en funcionamiento.** Las peticiones realizadas a la interfaz antigua se redireccionan a la versión vieja del servicio y al contrario con las nuevas peticiones.

Sin embargo, esta aproximación cuenta con algunas desventajas. En primer lugar, si se debe hacer una modificación como solucionar un defecto, posible-

mente se deba hacer y desplegar en ambas versiones del servicio. En segundo lugar, se necesita un middleware con la lógica para redireccionar las peticiones a una u otra versión. En conclusión, esta solución puede resultar apta para periodos cortos de tiempo, pero cuanto más tiempo permanezcan los clientes empleando la vieja versión del servicio más riesgo tendremos de que ocurran estos problemas. [21]

En nuestro caso de estudio, tenemos control de todos los consumidores ya que la interfaz de los servicios no va a ser publicada para su consumo por aplicaciones de terceros. Los consumidores de un microservicio serán o la interfaz de usuario u otros servicios. Como hemos comentado en la sección **7.2.1 Consumo de otros microservicios**, el proxy se añade como un paquete NuGet con una versión específica. Para asegurarnos de que siempre se consume la última versión de un servicio, vamos a hacer uso de variables para referir a la versión del paquete NuGet a instalar, como se puede ver en la figura 7.13.

```
<ItemGroup>
  <PackageReference Include="Shop.Backend.Microservices.Common.Application" Version="$(Common)" />
  <PackageReference Include="Shop.Backend.Microservices.Notifications.Proxy" Version="$(Notifications)" />
  <PackageReference Include="Shop.Backend.Microservices.Reports.Proxy" Version="$(Reports)" />
  <PackageReference Include="Shop.Backend.Microservices.Security.Proxy" Version="$(Security)" />
  <PackageReference Include="StyleCop.Analyzers" Version="1.1.0-beta004" />
</ItemGroup>
```

Figura 7.13: Fragmento del proyecto (.csproj) del servicio de pedidos.

A estas variables se les da valor en un archivo llamado Versions.props. Este archivo está centralizado y es común para todos los servicios para que todos ellos referencien a la misma versión de otro microservicio.² Cuando se realice un cambio que cambie la interfaz de un servicio, la versión de este se incrementará en una unidad. Los consumidores deberán ser modificados para adaptarse al cambio de la interfaz y deberán desplegarse también. Como se puede ver en el archivo, la versión de cada servicio evoluciona de forma separada.

```
<Project>
  <PropertyGroup>
    <Common>1.0.5</Common>
    <Security>1.0.2</Security>
    <Orders>1.0.0</Orders>
    <Reports>1.0.0</Reports>
    <Incidences>1.0.0</Incidences>
    <Notifications>1.0.1</Notifications>
  </PropertyGroup>
</Project>
```

Figura 7.14: Archivo Versions.props con las versiones de los microservicios.

² Common MSBuild properties and items with Directory.Build.props: <https://www.thomaslevesque.com/2017/09/18/common-msbuild-properties-and-items-with-directory-build-props/>

7.4 Adaptación de la interfaz de usuario

Para que la UI emplee los microservicios que se han creado se debe sustituir el proxy monolítico por los proxies de los microservicios de Seguridad, Incidencias y Pedidos. Esto se hace eliminando el paquete NuGet del proxy monolítico e instalando los tres paquetes nuevos.

Una vez hecho esto, habrá que corregir las referencias a diferentes espacios de nombres. Los espacios de nombre de la solución monolítica a la solución basada en microservicios se han modificado para distinguir clases de ambas soluciones que se llaman igual. Otro cambio que se debe realizar, como muestra la figura 7.15 es invocar el código para resolver las interfaces de contratos con los proxies.

```
using global::Xamarin.Forms;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
- using Shop.Backend.Monolithic.Proxy;
+ using Shop.Backend.Microservices.Incidences.Proxy;
+ using Shop.Backend.Microservices.Orders.Proxy;
+ using Shop.Backend.Microservices.Security.Proxy;
using Shop.Frontend.Xamarin.Components.Resources;
using Shop.Frontend.Xamarin.Helpers;
using Shop.Frontend.Xamarin.Interfaces;

@@ -27,8 +29,14 @@ public App()

    this.serviceCollection = new ServiceCollection();

-    this.serviceCollection.AddProxyDependencies(this.configuration);
-    this.serviceCollection.RegisterClients(new HttpClient(), this.configuration);
+    this.serviceCollection.AddSecurityProxy(this.configuration);
+    this.serviceCollection.RegisterSecurityClient(new HttpClient(), this.configuration);
+
+    this.serviceCollection.AddOrdersProxy(this.configuration);
+    this.serviceCollection.RegisterOrdersClients(new HttpClient(), this.configuration);
+
+    this.serviceCollection.AddIncidencesProxy(this.configuration);
+    this.serviceCollection.RegisterIncidencesClients(new HttpClient(), this.configuration);
```

Figura 7.15: Cambios en la UI para emplear la solución basada en microservicios.

7.5 Adaptación de las pruebas

De la misma forma que se han migrado las clases de cada una de las entidades a un microservicio específico, las pruebas también se situarán en el contexto de un servicio.

En la sección 6.4 **Pruebas** hemos visto que la mayoría de pruebas realizadas eran de integración porque representaban un caso de uso de la aplicación. Vamos a emplear la clasificación de pruebas hemos presentado en el apartado 2.5 **Los microservicios en la fase de pruebas**. Las pruebas de integración pasarán ahora a denominarse pruebas de servicio, si toda la lógica que necesita se encuentra en un único servicio, o de extremo a extremo, si involucra para su ejecución más de un servicio.

De las pruebas realizadas en la solución monolítica, la mayoría se transformarán en pruebas de servicio. Sin embargo, servicios como los de incidencias o pedidos dependen del de seguridad para realizar muchos de los casos de uso en los que se requiere obtener datos de un cliente. Para evitar tener que representar estas pruebas como de extremo a extremo, mucho más costosas, se empleará un fake del servicio de seguridad. Si recordamos, en la solución monolítica ya hemos tenido que emplear un fake porque algunas clases de Identity trataban de acceder a la base de datos de usuarios. Ahora, el fake será de la interfaz de contratos de seguridad y devolverá, por ejemplo, los datos de un usuario falso cuando se requiera uno por su identificador.

```

/// <summary> The fake security manager.
2 references | VictorIrranzo, 22 days ago | 1 author, 1 change
public class FakeSecurityManager : ISecurityManager
{
    /// <inheritdoc/>
    0 references | VictorIrranzo, 22 days ago | 1 author, 1 change | 0 exceptions
    public Task<UserDTO> GetUserByIdAsync(Guid userId)
    {
        return Task.FromResult(new UserDTO()
        {
            Name = "Paco Flores",
        });
    }
}

```

Figura 7.16: Fake de la interfaz de contratos del servicio de seguridad.

Algunas pruebas han tenido que implementarse obligatoriamente como pruebas de extremo a extremo. Es el caso de la prueba para la generación de una factura. En este tipo de prueba se hará uso de la clase de ASP .NET Core TestServer. TestServer permite la creación de un servidor en memoria para pruebas que redirige las peticiones que recibe hacia los controladores que representa.³ De esta forma, las dependencias que tiene un servicio como pedidos para las pruebas quedan suplidas con un servidor que responde tal y como lo haría el servicio de informes y no como un fake.

```

/// <summary> Set ups the reports microservices, creating the service in memory.
1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
private void SetupReports(IServiceCollection services)
{
    // Gets the Startup class of the reports microservice and runs it.
    IWebHostBuilder webHostBuilder = WebHost.CreateDefaultBuilder()
        .UseContentRoot(ReportsStartup.GetServicesPath())
        .UseEnvironment("Development")
        .UseStartup<ReportsStartup>();

    // Launches the microservice in the URL specified in the Development appsettings.
    TestServer testServer = new TestServer(webHostBuilder);
    IServiceProvider reportsServiceProvider = testServer.Host.Services
        .CreateScope().ServiceProvider;

    // Creates a proxy pointing to the URL where the test server has been created.
    IReportsManager reportsManager = new ReportsProxy(
        new ReportsClient("http://localhost:8095", testServer.CreateClient()));

    // Registers the contracts of reports with the created proxy.
    services.AddSingleton<IReportsManager>(reportsManager);
}

```

Figura 7.17: Fake de la interfaz de contratos del servicio de seguridad.

³ In-Memory ASP.NET Core Integration Tests with TestServer: <https://visualstudiomagazine.com/articles/2017/07/01/testserver>

Otras pruebas que se han modificado son las del microservicio de incidencias. En las pruebas de servicios que emplean SQL se crea una base de datos de este tipo en memoria. La base de datos de incidencias ahora es Firebase y en Entity Framework no existe la funcionalidad de hacer BD en memoria para este proveedor. Para que las pruebas del servicio no dejen de cubrir código de la capa de persistencia, las pruebas escribirán en la base de datos de Firebase y eliminarán el contenido que han escrito en un método `tearDown`. Esto no es necesario en las bases de datos en memoria porque son eliminadas al final de cada prueba.

```
/// <summary>
/// Tears down the test, reverting the changes done at the end of every test.
/// </summary>
[TearDown]
0 references | VictorIrranzo, 22 days ago | 1 author, 1 change | 0 exceptions
public void TearDown()
{
    IIncidentesManager incidencesManager = this.serviceProvider.GetService<IncidentesManager>();
    incidencesManager.DeleteAsync(this.incidenceGuid);
    incidencesManager.DeleteAsync(this.incidenceGuid2);
}
```

Figura 7.18: Método para eliminar registros escritos durante las pruebas del servicio de incidencias.

Otra posible opción habría sido crear una base de datos de Firebase distinta la de producción y a la que apuntaran las pruebas. Con esta solución se habría tenido que implementar igualmente un método para eliminar el contenido escrito durante la prueba o de otra forma se crearía contenido duplicado en esta BD.

7.6 Despliegue

CAPÍTULO 8

Evaluación de las soluciones

8.1 Mantenimiento

A continuación, revisaremos como una arquitectura monolítica y una basada en microservicios afrontan diferentes situaciones de mantenimiento. Para ello, recorreremos los diferentes tipos de mantenimiento según la ISO/IEC 14764 y plantearemos ejemplos dentro de nuestro caso de estudio.

8.1.1. Mantenimiento correctivo

El mantenimiento correctivo consiste en una modificación del producto software una vez se ha entregado para corregir un error detectado. [27] En piezas de código muy largas resulta difícil encontrar dónde se encuentra un defecto. Gracias al diseño modular de las arquitecturas orientadas a servicios, un defecto será más fácil de localizar. A nivel de código, clases que deben modificarse como un conjunto para corregir un defecto estarán en el mismo microservicio.

En el caso de estudio, un fallo, por ejemplo, en la creación de un pedido se localizará en el microservicio de pedidos. En la misma solución monolítica, si las responsabilidades de las clases no están bien delimitadas, se deberá buscar entre más código para encontrar el defecto.

Sin embargo, esto no siempre es así: dependerá de la interacción entre los servicios. Si para ofrecer una funcionalidad se encadenan muchas peticiones entre servicios diferentes, entender el flujo de invocaciones será más difícil. La interacción basada en eventos también disminuye la comprensión de las comunicaciones. En consecuencia, el proceso de depuración para detectar un defecto será más costoso. Los tokens de correlación son un mecanismo que pueden facilitar esta tarea. Con ellos, se pasa el mismo identificador en cada invocación que se realiza a otro servicio para obtener trazabilidad. [4]

Por ejemplo, si se observa un fallo de que no se genera correctamente una factura para un cliente, existirán tres microservicios implicados: los de seguridad, informes y pedidos. Para depurar la funcionalidad hace falta una instancia de cada uno de ellos en ejecución. En cambio, en la solución monolítica basta con instanciar un único servicio, el del monolito completo. Además, en él todas las llamadas se realizarán en el mismo proceso.

8.1.2. Mantenimiento perfectivo

Según la ISO/IEC 14764, el mantenimiento perfectivo es una modificación de un producto software una vez se ha entregado para proveer mejoras a sus usuarios, mejorar la documentación del programa o atributos del software como el rendimiento o la mantenibilidad. [27] Según Sommerville, este tipo de mantenimiento se origina cuando los requisitos del sistema cambian para responder a un cambio en el negocio. Además, el tamaño de la modificación a realizar es normalmente superior al de modificaciones asociadas al del resto de tipos de mantenimiento. [28]

Como hemos comentado en el apartado 2.3.3 *Descomposición en microservicios*, el hecho de que se divida la solución en contextos bien delimitados hará que los nuevos requisitos encajen mejor en uno de ellos.

Por ejemplo, se puede añadir una nueva funcionalidad que consista en generar un informe de una incidencia, mostrando datos como cuanto tiempo ha estado abierta o los empleados que han participado en su resolución. Como en la generación de una factura, hay más de un servicio implicado, pero está claro que es el de incidencias quien debe ofrecer este método en su interfaz.

En caso de que no se pueda situar en ninguno de los contextos delimitados, es probable que estemos ante un nuevo microservicio. Un ejemplo para nuestro caso de estudio sería una nueva funcionalidad para obtener la localización de la tienda más cercana asociada al comercio electrónico.

El problema al que nos podemos enfrentar es que un nuevo requisito nos haga plantearnos nuestro diseño. En la solución basada en microservicios, si se desea añadir una nueva funcionalidad tal que una incidencia tenga que tener siempre asociado un pedido, se estarán relacionando entidades de distintos contextos.

Esto no es un problema y puede gestionarse a través de mecanismos para la consistencia eventual. Con estos mecanismos, si se elimina por ejemplo un pedido, el cambio se propagaría hasta el microservicio de incidencias para que actuase en consecuencia borrando las incidencias asociadas. En definitiva, aunque se puedan implementar manteniendo los límites de los servicios, puede ser interesante replantearse estos límites de acuerdo a los nuevos requisitos funciones.

8.1.3. Mantenimiento adaptativo

El mantenimiento adaptativo se define como la modificación de un producto software tras su entrega para que sea utilizable en un nuevo entorno. [27] Según Sommerville, estos cambios pueden deberse, por ejemplo, a cambios en el hardware, la plataforma donde opera el sistema o cambios en productos software de los que se depende. [28] En este tipo de mantenimiento, una arquitectura basada en microservicios presenta grandes beneficios que vamos a explicar siguiendo los 3 ejemplos de Sommerville.

En primer lugar, al poderse desplegar de forma independiente, el alcance de un cambio en el hardware influirá será menor. Cada servicio se puede desplegar en un hardware diferente de acuerdo a sus necesidades. En consecuencia, el aprovechamiento de los recursos es mayor. Microservicios que requieren mayor rendimiento se pueden ubicar en servidores con mayores prestaciones. Si nos situamos en el contexto de nuestro caso de estudio, los microservicios de incidencias y pedidos pueden situarse en un servidor con mayor CPU y RAM. En caso de que se detecte que las prestaciones hardware comienzan a deteriorarse para albergar ambos servicios en el mismo servidor, podemos migrar uno de ellos a otro servidor.

En una arquitectura monolítica, al desplegarse todo el back-end como un conjunto las prestaciones hardware del servidor han de ser superiores para soportar todas las peticiones que recibe el monolito. En caso de que se requiera migrar a otro servidor, se ha de migrar todo el sistema como un conjunto.

En segundo lugar, en un sistema basado en microservicios, gracias a su integración interprocedural, la plataforma donde operan diferentes servicios puede ser distintos. Por ejemplo, el microservicio de incidencias podría ejecutarse en AWS mientras que el resto del sistema se encontrara en la plataforma de Azure. Obviamente, no podemos decir lo mismo de un sistema monolítico.

En tercer lugar, las dependencias del software se pueden gestionar a un nivel más granular en un sistema basado en microservicios. Por ejemplo, puede ocurrir que un sistema necesite ejecutarse en .NET Framework para emplear una librería que no sea multiplataforma. En una arquitectura basada en microservicios, esa librería solo se empleará en un contexto bien delimitado, por lo que solo hará falta tener un servicio en este framework. En cambio, en un sistema monolítico para poder emplearla todo el sistema debería estar en este framework, lo que resulta muy restrictivo si se desea hacer un sistema multiplataforma.

Lo mismo ocurre con la versión de una librería. Diferentes servicios pueden emplear versiones diferentes de ella. Así, si se tiene que actualizar la librería, solo hará falta hacerlo en un microservicio y no en todo el sistema, disminuyendo sus posibles efectos secundarios.

8.2 Comparación de las soluciones ante RNFs

Ahora, revisaremos las soluciones frente a diferentes requisitos no funcionales que hemos mencionado en la sección **2.2.1 Requisitos funcionales y no funcionales**. En el capítulo de **Conclusiones** sintetizaremos lo que aquí se comenta para ofrecer una respuesta al cuarto objetivo que proponemos en este memoria.

8.2.1. Disponibilidad

La disponibilidad de un servicio se puede definir como la habilidad de un consumidor para conectarse a él y enviarle peticiones. [26] Para que un sistema esté disponible se han de implementar mecanismos conformes a sus necesidades para reaccionar automáticamente al incremento de carga o a un fallo. [21]

En la solución basada en microservicios, se ha garantizado la disponibilidad frente a algunas situaciones gracias a Kubernetes. Para cada uno de los microservicios desplegados se puede establecer el número de replicas que en todo momento se desea tener de él. Esto nos garantiza que en caso de fallo el microservicio se recupere rápidamente.

Sin embargo, no se ha implementado ningún mecanismo para reaccionar automáticamente al incremento de peticiones. Para garantizar esto se tiene que modificar el tamaño de un clúster de forma dinámica en función de su demanda. Con este propósito, se debe situar un balanceador de carga antes del clúster que modifique el número de replicas para crear nuevas en función de la carga que observe. [25]

8.2.2. Tolerancia a fallos

Un sistema debe ser capaz de tolerar fallos y continuar trabajando. En las arquitecturas SOA, la integración entre servicios está muy ligada a como se gestionan los fallos:

se debe asumir que cualquier servicio puede estar inoperativo. Por ejemplo, cuando se realiza una petición, no se puede esperar indefinidamente a que un servicio responda. [21]

En nuestro caso de estudio, este problema está presente en las dos soluciones implementadas. El front-end comunica con el back-end a través de llamadas HTTP, al igual que ocurre entre los propios microservicios de la segunda solución. En las soluciones, para limitar el tiempo de espera se emplean timeouts, tal que si una petición no obtiene una respuesta esta se considera un fallo.

Sin embargo, el uso de timeouts no es una solución aconsejable debido al valor que se le debe dar a la espera. No se puede poner un límite demasiado bajo porque trataría como errores respuestas que han tardado más de lo debido. Tampoco puede ser muy alto, o se tardaría mucho en devolver un error, sobretodo cuando se encadenan diferentes llamadas.

Una solución pasa por usar un patrón de cortocircuitos. Los cortocircuitos monitorizan continuamente el estado de un servicio. Si detectan que un servicio no responde cierto número de peticiones, el cortocircuito pasa a un estado en el que rechaza cualquier petición para que estas fallen rápidamente. Cuando el servicio vuelve a estar operativo, el cortocircuito comienza a aceptar de nuevo peticiones. [26]

También se ha de proteger el sistema de los fallos de infraestructura. La mejor manera de hacer un sistema tolerante a este tipo de fallos es no situar todos sus servicios en una misma infraestructura, ya sea una máquina física, red, etc. Así, se consigue que no haya un único punto de fallo. [21] En la solución del caso de estudio basada en microservicios, esto se consigue gracias al uso de Kubernetes, como hemos comentado en el apartado anterior.

El uso de diferentes bases de datos y no una única como hay en la solución monolítica también hace que en los datos no haya un único punto de fallo. No obstante, se podrían haber empleado tecnologías de bases de datos distribuidas (BDD) como MongoDB para garantizar esta característica a nivel de cada microservicio.

Por último, organizaciones como Netflix prueban la tolerancia a fallos de sus servicios incitando al fallo de estos. Para ello, emplea un conjunto de programas que componen el “ejército de simios”, que apagan máquinas y centros de datos en producción de forma aleatoria. [17]

8.2.3. Utilización de recursos

Los microservicios son una solución para afrontar problemas inherentes a productos de software de gran tamaño. Estos sistemas pueden llegar a descomponerse en cientos de pequeños servicios. Mientras que, según la teoría, en estos sistemas si que se observa un mejor aprovechamiento de los recursos por emplear microservicios, [7] en nuestro caso de estudio no. Por ejemplo, aunque hayamos hecho una solución basada en microservicios no hemos desplegado el servicio de pedidos en un hardware con mejores prestaciones, al igual que tampoco hemos hecho la situación inversa con los servicios que reciben menos peticiones.

En el siguiente estudio [1], realizado en la Universidad Politécnica de Cataluña junto con la organización IBM, se evalúa el uso de recursos de un servicio desplegado en diferentes configuraciones de contenedores y en una máquina virtual. En su primer experimento se evalúa el uso que hacen de CPU, que es similar tanto en la MV como en los contenedores. También se evalúa el tiempo que consume la creación del servicio, donde en todos los escenarios el tiempo de desplegar el servicio en un MV es al menos es el

doble que en un contenedor. En los últimos experimentos se evalúa el uso que hacen de la red. En este aspecto se puede concluir que, para las invocaciones que se hacen dentro del mismo host, los contenedores ofrecen un mejor rendimiento, mientras que para las llamadas fuera del host, el rendimiento de ambas tecnologías es similar.

Durante el desarrollo, tanto el sistema basado en microservicios como el monolítico se han desplegado en contenedores. Con el comando **docker stats** podemos obtener estadísticas sobre el uso de recursos que hacen los contenedores en ejecución. El resultado de aplicarlo en el entorno de desarrollo y con los servicios sin recibir peticiones se puede ver en la figura ?? . No se aprecia una gran diferencia en el uso de recursos del contenedor monolítico (shopmonolithic) y un contenedor de un microservicio. Los contenedores basados en una imagen .NET hacen un mayor uso de memoria. Estos también tienen un tamaño de la imagen Docker superior: alrededor de 1,8 GB frente a los 644 MB que emplea el microservicio de notificaciones, basado en una imagen java.

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O |
|--------------|---|-------|---------------------|-------|-------------|
| d484c7f61ff1 | shopmonolithic | 0.12% | 28.19MiB / 1.934GiB | 1.42% | 1.72kB / 0B |
| ccebf7c97678 | tfg_backend_microservices_notifications_1 | 0.08% | 11.86MiB / 1.934GiB | 0.60% | 1.21kB / 0B |
| 87e9606d80ed | tfg_backend_microservices_incidentes_1 | 0.05% | 25.77MiB / 1.934GiB | 1.30% | 1.39kB / 0B |
| 0fa1c6be2acb | tfg_backend_microservices_security_1 | 0.05% | 26.82MiB / 1.934GiB | 1.35% | 1.14kB / 0B |
| 5aaa398692ae | tfg_backend_microservices_reports_1 | 0.06% | 25.5MiB / 1.934GiB | 1.29% | 1.65kB / 0B |
| cbf658304185 | tfg_backend_microservices_orders_1 | 0.05% | 26.04MiB / 1.934GiB | 1.31% | 1.14kB / 0B |

Figura 8.1: Estadísticas de uso de recursos de los contenedores de la solución monolítica y la basada en microservicios.

En resumen, en el caso de estudio no se han apreciado grandes diferencias en el uso de recursos entre la solución monolítica y la basada en microservicios. Esto se debe principalmente al pequeño tamaño del sistema desarrollado. Sin embargo, tal y como dice la literatura, los servicios se pueden desplegar de forma independiente de acuerdo a las prestaciones que necesiten.

8.2.4. Capacidad de ser reemplazado

Los microservicios incrementan la facilidad de cambio de un producto software. Un cambio en un único servicio se despliegue de manera independiente y llega al cliente final de forma más rápida si se siguen prácticas como las de entrega continua.

La capacidad para ser reemplazado está estrechamente relacionada con la facilidad de cambio. El uso de interfaces en la capa de contratos, por ejemplo, hace que un microservicio pueda ser modificado y reemplazado fácilmente. El componente sustituto puede estar implementada con otra tecnología si respeta la vieja interfaz.

Si recordamos el cronograma que hemos explicado en el apartado **5.1 Plan de trabajo**, la refactorización de la solución monolítica para generar cualquiera de los servicios apenas se ha prolongado por más de un día (las tareas que empiezan su nombre con la palabra “microservicio”). A esto habría que incluirle el tiempo que se ha tardado en desarrollar la lógica del servicio durante el desarrollo de la solución monolítica (las tareas que empiezan su nombre con la palabra “implementación”). En comparación, el desarrollo de la solución monolítica en su totalidad se ha prolongado durante casi un mes (desde el día 19 de Junio hasta el día 16 de Julio).

El desarrollo de cada microservicio apenas ha costado dos semanas, tal y como dice la regla de Jon Eaves que hemos explicado en el apartado **2.2.1 Requisitos funcionales y no funcionales**. Si un componente del software debe ser reemplazado, puede ocurrir que en la solución basada en microservicios solo se tenga que reemplazar un microservicio mientras que en la solución monolítica se tenga que adaptar todo el monolito. Si consideramos que el tiempo para desarrollar un componente software que reemplaza a otro

es similar al tiempo que costó desarrollar el segundo, reemplazar la solución monolítica será más costoso que reemplazar un único microservicio. Si nos situamos en un caso más extremo en el que se tuviera que reemplazar todo el sistema basado en microservicios, este problema podría abordarse de forma incremental por la modularidad de la solución. De nuevo, no se puede afirmar lo mismo de la solución monolítica.

8.3 Otras posibles soluciones

Tanto el back-end monolítico como el basado en microservicios representan uno de los posibles diseños que se pueden emplear para implementar el sistema especificado. Otras soluciones pueden ofrecer los mismos requisitos funcionales pero con distintos atributos de calidad.

Respecto a la solución monolítica, para un caso de estudio como el que hemos presentado es posible que no sea necesario una arquitectura de 6 capas. Se podía haber seguido un diseño más simple, como el que se presenta para el microservicio de informes en la solución basada en servicios.

Las capas de aplicación y servicios se podían haber combinado en una única siempre que se mantenga el principio de responsabilidad única de una clase. Los controladores de la capa de servicios definían el método HTTP expuesto en la API del back-end, mientras que los managers de aplicación contenían la lógica de la operación. Aunque se combinaran ambas capas, se debería seguir manteniendo este criterio y mantener ambos tipos de clases.

Las capas de persistencia y dominio también podrían combinarse en una única. La capa de aplicación depende de ambas y la capa de persistencia depende de la de dominio. En nuestro caso de estudio, la capa de dominio solo contiene las entidades que hemos identificado en el modelo de dominio. Almacenarlas en la capa de persistencia, que es la capa que principalmente las emplea para implementar las operaciones CRUD a través de Entity Framework, no traería efectos adversos.

En cuanto a la solución basada en microservicios, se puede ver claramente que algunos de los servicios desarrollados solo son invocados por un consumidor. Concretamente, los servicios de informes y notificaciones son solo invocados por el servicio de pedidos. Esto debería conducirnos a razonar sobre si es beneficioso combinar los tres en un único servicio.

Gracias a su implementación, tanto el servicio de informes como el de notificaciones están desacoplados de los datos que requieren para realizar su función porque estos son transferidos como parámetros del método. Esto se ha hecho así para facilitar su invocación por cualquier servicio. Por ejemplo, el servicio de informes se podría haber implementado para que expusiera él directamente todos los informes que puede generar. Sin embargo, esto habría supuesto que tuviera una dependencia con todos los servicios de los que necesitara obtener datos. Como se desea mantener que puedan seguir siendo invocados desde cualquier otro servicio, no vamos a combinarlos con el servicio de pedidos.

No obstante, podemos plantearnos si realmente vale la pena representar estos componentes de software como microservicios. Por ejemplo, si observamos que apenas se modifican una vez implementados, sería buena idea representarlos como una librería y no como un servicio. Así, el sobrecoste asociado a las invocaciones a través de la red se reduciría a cambio de no poder desplegarlos de forma independiente.

Otra consideración a tener en cuenta a la hora de transformar un servicio en una librería es que las librerías están ligadas a una única plataforma. Por ejemplo, el servicio de

notificaciones está desarrollado en Java y gracias a que se ejecuta en otro proceso puede ser invocado por cualquier servicio, da igual su lenguaje de programación, a través de llamadas HTTP. La implementación del servicio tal como está en Java no puede desplegarse como una librería y emplearse en los servicios desarrollados en .NET porque no son plataformas compatibles. Por tanto, para cada servicio que desea consumir este componente se debería dar una implementación compatible con la plataforma del consumidor.

8.4 Ventajas e inconvenientes de los microservicios

CAPÍTULO 9

Conclusiones

En el apartado **8.1 Mantenimiento** se han planteado diferentes situaciones que acreditan que, durante esta fase del ciclo del software, los beneficios de una arquitectura basada en microservicios son superiores a los de una arquitectura monolítica.

Bibliografía

- [1] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015*, pages 27–34, 2016.
- [2] David Ameller, Claudia Ayala, Jordi Cabot, and Xavier Franch. Non-functional Requirements in Architectural Decision Making. *IEEE SOFTWARE*, 2013.
- [3] 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.
- [4] Vitaly Baum. Practical Microservices: Correlation Tokens – Microservices Practitioner Articles, 2016.
- [5] Mike Cohn. *Succeeding with agile : software development using Scrum*. Addison-Wesley, 2010.
- [6] Carlos de Alfonso, Amanda Calatrava, and Germán Moltó. Container-based virtual elastic clusters. *Journal of Systems and Software*, 127:1–11, 2017.
- [7] Cesar De la Torre, Bill Wagner, and Mike Rousos. *Microservicios .NET: Arquitectura para Aplicaciones .NET Contenerizadas*. Microsoft Corporation, 2 edition, 2018.
- [8] 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.
- [9] Jon Eaves. Micro services, what even are they?, 2014.
- [10] João M. Fernandes and Ricardo J. Machado. *Requirements in Engineeng Projects*. Springer, 2016.
- [11] Martin Fowler. Continuous Integration, 2006.
- [12] Martin Fowler. PolyglotPersistence, 2011.
- [13] Martin Fowler. ContinuousDelivery, 2013.
- [14] Martin Fowler. BoundedContext, 2014.
- [15] Susan J. Fowler. *Microservices in Production : Standard Principles and Requirements*. O'REILLY, 2017.
- [16] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, feb 2012.

- [17] James Lewis and Martin Fowler. *Microservices*, 2014.
- [18] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.
- [19] Karl Matthias and Sean P Kane. *Docker Up & Running*. O'REILLY, 2015.
- [20] Gavin Mulligan and Denis Gračanin. A COMPARISON OF SOAP AND REST IMPLEMENTATIONS OF A SERVICE BASED INTERACTION INDEPENDENCE MIDDLEWARE FRAMEWORK. Technical report, 2009.
- [21] Sam Newman. *Building Microservices*. O'Reilly, 2015.
- [22] Roy. Osherove. *The art of unit testing : with examples in C Sharp*. Manning Publications, 2014.
- [23] Roger S. Pressman. *Software engineering : a practitioner's approach*.
- [24] Laila M Qaisi and Ibrahim Aljarah. A Twitter Sentiment Analysis for Cloud Providers: A Case Study of Azure vs. AWS. pages 1–6, 2016.
- [25] David Rensin. *Kubernetes*. O'REILLY, 2015.
- [26] Mark Richards. *Microservices AntiPatterns and Pitfalls*. O'REILLY, 2016.
- [27] IEEE COMPUTER SOCIETY. *Guide to the Software Engineering - Body of Knowledge*. 2014.
- [28] Ian Sommerville. *Software Engineering*. 2010.
- [29] International Standard. ISO 25010. 2010, 2010.
- [30] Jurg van. Vliet and Flavia. Paganelli. *Programming Amazon EC2*. O'Reilly Media, 2011.
- [31] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.
- [32] Wikipedia. Service (systems architecture).

APÉNDICE A

Descripción del prototipo de IU
desarrollado

APÉNDICE B

Despliegue del sistema basado en microservicios

B.1 Ejecución de un contenedor con Docker

B.1.1. Archivo Docker Compose

B.2 Introducción a Azure Kubernetes Service (AKS)

B.3 Depuración de contenedores a través de Azure
