



UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Técnica Superior  
Ingeniería de Sistemas Informáticos

TRABAJO FIN DE GRADO EN  
INGENIERÍA DEL SOFTWARE

*Arquitecturas de microservicios para aplicaciones  
desplegadas en contenedores*



Autor:  
Carlos Jiménez Aliaga

Tutora:  
Ana Isabel Lías Quintero

Diciembre 2018

*A mis padres,  
por haberme guiado a lo largo de los años.*

*A mi hermana,  
por estar siempre ahí y ayudarme a dar lo mejor de mí.*

*Y a mi abuela, donde quiera que esté,  
que siempre estuvo orgullosa de que su nieto estudiara álgebra.*

## RESUMEN

El estado actual de las redes de comunicaciones y la evolución de los dispositivos electrónicos, cada vez más potentes, ligeros y portátiles, ha cambiado la forma en la se consumen los servicios en línea. Los dispositivos están siempre conectados y los clientes necesitan acceder a los datos desde cualquier lugar y de forma instantánea. Esta necesidad implica una rápida adaptación de los servicios así como replicar los datos geográficamente para que puedan estar accesibles globalmente con una latencia mínima. La cuestión es, ¿cómo se consigue desarrollar servicios que cumplan las necesidades actuales de una forma eficiente a la vez que sean capaces de adaptarse rápidamente a nuevos cambios?

El propósito de este proyecto de fin de grado es presentar cómo definir arquitecturas basadas en microservicios, haciendo uso de buenas prácticas, para aplicaciones desplegadas en contenedores. Para ello se hace uso de tecnologías de código abierto y multiplataforma, en concreto .NET Core para el desarrollo y Docker como motor de los contenedores. Elegir dónde y cómo desplegar las aplicaciones es un factor determinante. Se presenta el uso los orquestadores para la gestión de los contenedores (Kubernetes específicamente, que es también de código abierto) y de las plataformas Cloud donde el orquestador, las aplicaciones y servicios adicionales (como las bases de datos) son desplegados y accesibles a nivel global. En este proyecto se ha elegido la nube de Microsoft, Microsoft Azure, como proveedor Cloud.

Con la arquitectura basada en microservicios desplegados en contenedores se consigue: alta disponibilidad, al tener varias instancias disponibles en distintas máquinas al mismo tiempo; escalabilidad, cada microservicio será capaz de escalar individualmente; y despliegue de nuevas versiones en un tiempo mínimo, gracias a la simbiosis de contenedores y orquestadores. Además, el escenario expuesto es igualmente aplicable a distintas tecnologías, tanto en lo relativo a plataformas como a los entornos.

## ABSTRACT

The way online services are accessed and consumed has changed thanks to the current status of communication networks and the evolution of electronic devices. Devices are now more powerful, lighter and portable. Also, they are always connected, so we need to access to our data everywhere and instantly. Services need to adapt and transform quickly and be able to replicate the data around the globe, so users can access them in a global zero-latency manner. Therefore, how can we develop services that flow with the changes in an efficient way?

The purpose of this project is to show the way of defining microservices architectures, applying good patterns and practices, for applications running on containers. This project proposes open source and multiplatform technologies such .NET Core for development and Docker as containers' engine. Deciding where and how deploy the applications is an important factor. To help making those decisions, orchestrators are introduced in order to manage the containers (Kubernetes, which is also open source, is the chosen one) and cloud platforms where the orchestrator, the applications and additional services (such databases) are deployed and accessible worldwide. Microsoft Azure, Microsoft's cloud platform, is the cloud chosen as cloud provider.

With microservices based architectures deployed on containers you get high availability, having several instances available in different machines at the same time; scalability, each microservice can scale individually based on its own needs; and deployment of new versions in a record time, thanks of containers and orchestrators working together. However, the solution exposed is equally valid to different technologies and applicable to several cloud platforms and on-premise environments.

## GLOSARIO

**PID:** Corresponde a *"Process ID"* o *"process identifier"* que en español lo traduciríamos como identificador de proceso. Es la forma en la que nos referimos a un proceso puesto que este identificador es único en el sistema.

**NuGet:** es un repositorio de paquetes y bibliotecas de código, libre y de código abierto, que se utiliza dentro del ecosistema de desarrollo de Microsoft.

**Framework:** "En el desarrollo de software, un entorno de trabajo es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto." [1]

**Runtime:** "Un entorno de ejecución (runtime environment en inglés) es un estado de máquina virtual que suministra servicios para los procesos de un programa de computadora que se está ejecutando. Puede pertenecer al mismo sistema operativo, o ser creado por el software del programa en ejecución. En la mayoría de los casos, el sistema operativo maneja la carga del programa con una parte del código llamada cargador, haciendo configuración básica de memoria y enlazando el programa con cualquier biblioteca de vínculos dinámicos a la cual haga referencia. En algunos casos un lenguaje o implementación hará esas tareas en lugar del runtime del lenguaje, a pesar de que es inusual en los lenguajes principales sobre los sistemas operativos de usuarios normales." [2]

**Git:** un sistema distribuido que permite a cada desarrollador tener una copia local del repositorio, crear ramas o bifurcaciones locales, ir guardando el progreso poco a poco en local o elegir sincronizarlo con el servidor y posteriormente mover estos cambios a la rama principal.

**GitHub:** es una plataforma de desarrollo colaborativo que se utiliza para alojar proyectos utilizando el control de versiones Git. Se utiliza principalmente para la creación de código fuente y es la plataforma de proyectos de código abierto y libre más grande del planeta.

**Pipeline:** Consiste en una cadena de elementos para ser procesados y ordenados de tal forma que la salida de un elemento es la entrada del siguiente. El nombre (se traduce como tubería) es debido a la analogía con una tubería física.

**IIS, Nginx, Apache:** son servidores web, disponibles en distintos sistemas operativos. Un servidor web es un programa informático que procesa una aplicación del lado del servidor, realizando conexiones bidireccionales o unidireccionales y sincrónicas o asíncronas con el cliente. Genera una respuesta en cualquier lenguaje o aplicación del lado del cliente. Generalmente se accede a él a través de navegadores web. Para la transmisión de datos se usan protocolos de comunicación.

**CLI:** Interfaz de línea de comandos. Es un método que permite dar instrucciones a algún programa informático por medio de una línea de texto simple.

**REST:** es un estilo de arquitectura software para sistemas distribuidos. Se fundamenta en un protocolo cliente/servidor sin estado. Cada mensaje HTTP enviado contiene toda la información necesaria para comprender la petición. [3]

**SDK:** kit de desarrollo de software o SDK (siglas en inglés de software development kit) es generalmente un conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto, por ejemplo ciertos paquetes de software, frameworks, sistemas operativos, etc.

**IDE:** : entorno de programación que ha sido empaquetado como un programa de aplicación. Consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica. Los IDEs pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes, y proveen un marco de trabajo amigable para la mayoría de los lenguajes de programación

**Endpoint:** punto de entrada a un servicio, proceso, cola o destino en una arquitectura orientada a servicios.

**Ensamblado:** en programación de sistemas informáticos o software de programación (en particular en .NET framework), un ensamblado o assembly, es una colección de uno o más archivos o ficheros, agrupados juntos para formar una unidad lógica.

**On-premise:** referido al mundo del software, instalación de un servicio sobre una máquina física en vez de instalarlo y utilizarlo en un sistema remoto o un servidor en un entorno Cloud.

**AMQP:** "El estándar AMQP (Advanced Message Queuing Protocol) es un protocolo de estándar abierto en la capa de aplicaciones de un sistema de comunicación. Las características que definen al protocolo AMQP son la orientación a mensajes, encolamiento ("queuing"), enrutamiento (tanto punto-a-punto como publicación-suscripción), exactitud y seguridad." [4]

Figura 1-1. Evolución de los servicios hasta la era del Cloud. <b>Fuente: Sequoia</b> .....	5
Figura 4-1. Creación de proyecto en Azure DevOps.....	16
Figura 4-2. Product Backlog en Azure Boards.....	18
Figura 4-3. Gestión de sprints en Azure Boards.....	19
Figura 4-4. Gestión de la capacidad del equipo en Azure Boards.....	20
Figura 4-5. Gestión backlog por sprint en Azure Boards.....	21
Figura 4-6. Tablero de tareas de un sprint en Azure Boards.....	22
Figura 4-7. Burndown de un sprint en Azure Boards.....	23
Figura 4-8. Creación de una rama asociada en Azure DevOps.....	24
Figura 4-9. Creación nueva Pull Request.....	25
Figura 4-10. Añadiendo un comentario en el código al revisar el pull request. ....	25
Figura 4-11. Completando pull request.....	26
Figura 5-1. Host de contenedores versus máquinas virtuales. <b>Fuente: Docker Docs</b> .....	27
Figura 5-2. Docker run para ejecutar imagen.....	31
Figura 5-3. Creación nuevo proyecto con el CLI de .NET Core.....	33
Figura 5-4. Publicación de proyecto con el CLI de .NET Core.....	33
Figura 5-5. Compilación imagen de Docker usando el CLI.....	34
Figura 5-6. Ejecución de la imagen compilada usando el CLI de Docker.....	35
Figura 5-7. Aplicación desplegada en el contenedor en ejecución.....	35
Figura 5-8. Habilitar soporte para Docker en VS2017.....	35
Figura 5-9. Añadir soporte para Docker en proyecto existente usando VS2017.....	36
Figura 5-10. Resultado ejecución docker ps.....	37
Figura 5-11. Resultado ejecución docker images.....	38
Figura 6-1. Arquitecturas monolíticas versus arquitecturas de microservicios. <b>Fuente: Elaboración propia.</b> .....	40
Figura 6-2. Pasos para identificar microservicios. <b>Fuente: Elaboración propia.</b> .....	42
Figura 7-1. Instalación Autofac desde el gestor de paquetes.....	53
Figura 7-2. Flujo de escritura en CQRS. <b>Fuente: Elaboración propia.</b> .....	69
Figura 7-3. Implementación CQRS en el microservicio de gestión de solicitudes. <b>Fuente: Elaboración propia.</b> .....	72
Figura 8-1. Arquitectura de la solución Titán. <b>Fuente: Elaboración propia.</b> .....	77
Figura 8-2. Capas DDD en el microservicio de gestión de solicitudes. <b>Fuente: Elaboración propia.</b> .....	79
Figura 9-1. Selección de origen en Azure Pipelines.....	85
Figura 9-2. Activación integración continua en build de Azure Pipelines.....	86
Figura 9-3. Definición tareas en build de Azure Pipelines.....	86
Figura 9-4. Configuración tarea docker-compose en build de Azure Pipelines.....	87
Figura 11-1. Costes mensuales de los servicios desplegados en la nube. ....	94
Figura 11-2. Presupuesto ejecución del proyecto.....	95

# ÍNDICE

RESUMEN .....	ii
ABSTRACT .....	iii
GLOSARIO .....	iv
1. INTRODUCCIÓN .....	5
1.1 Estado actual .....	5
1.1.1 Evolución de los servicios hasta la era actual .....	5
1.2 Metodologías ágiles y DevOps .....	6
1.3 Cloud computing .....	7
1.4 Y ahora, ¿qué? .....	7
1.5 Organización del proyecto.....	7
2. OBJETIVOS Y MOTIVACIÓN .....	9
2.1 Cómo surge .....	9
2.2 Objetivo general.....	9
2.3 Objetivos específicos .....	10
3. HERRAMIENTAS USADAS .....	11
3.1 El IDE: Visual Studio 2017 .....	11
3.2 Visual Studio Code.....	11
3.3 .NET Core.....	11
3.4 ASP.NET Core y ASP.NET Core MVC .....	12
3.5 Docker .....	13
3.6 Visual Studio Team Services (VSTS), ahora llamado Azure DevOps .....	13
3.6.1 Control de versiones .....	13
3.6.2 Gestión del proyecto y herramientas Ágiles.....	13
3.6.3 DevOps e integración continua.....	14
4. GESTIÓN DEL PROYECTO .....	15
4.1 Metodologías ágiles: Scrum .....	15
4.2 Azure DevOps: Creando el proyecto y equipo .....	16
4.3 Azure Boards: definiendo el Product Backlog.....	17



4.4 Azure Boards: definiendo los sprints (iteraciones).....	18
4.5 Azure Boards: Taskboard .....	21
4.6 Azure Boards: Gestión del código y tareas.....	23
5. CONTENEDORES Y DOCKER.....	27
5.1 Máquinas virtuales versus contenedores.....	27
5.2 Fundamentos en los que se basan los contenedores.....	28
5.3 Docker .....	29
5.3.1 Imágenes de Docker .....	29
5.3.2 Repositorio, Registro y Tags.....	30
5.3.3 Dockerfile y el formato de imágenes.....	30
5.4 Creando un Proyecto usando ASP.NET Core con soporte para Docker .....	32
5.4.1 .NET Core CLI y Dockerfile manual .....	32
5.4.2 Visual Studio 2017 con Dockerfile multi imagen.....	35
5.5 Comandos .....	37
5.5.1 Docker ps [19].....	37
5.5.2 Docker inspect [20] .....	38
5.5.3 Docker images [21].....	38
5.5.4 Docker start[23], stop[24] y restart[25].....	38
5.5.5 Docker push[26].....	38
6. SOLUCIÓN BASADA EN MICROSERVICIOS .....	39
6.1 Arquitecturas de microservicios .....	39
6.2 Definiendo el modelo de datos y el ámbito de cada microservicio .....	41
6.2.1 Analizando el dominio para identificar los microservicios .....	41
6.3 Comunicación entre los distintos microservicios.....	42
6.3.1 Protocolos y tipos de comunicación.....	43
6.3.2 Servicios basados en HTTP y REST .....	43
6.3.3 Comunicación asíncrona basada en mensajes .....	44
6.3.4 Registro de servicios.....	44
6.3.5 Exponiendo los microservicios directamente o utilizando una puerta de enlace .....	45
6.4 Ventajas de una solución basada en microservicios .....	46

6.5 Desventajas de una solución basada en microservicios.....	46
7. DISEÑO DE LA SOLUCIÓN .....	48
7.1 Inversión de control (IoC) e inyección de dependencias (DI).....	48
7.1.1 Contenedores de dependencias.....	50
7.1.2 Configurando el contenedor de dependencias integrado de <i>ASP.NET Core</i> .....	51
7.1.3 Configurando Autofac como contenedor de dependencias.....	53
7.2 Gestión del acceso a datos .....	54
7.2.1 Entidades de dominio .....	54
7.2.2 Enumeraciones usando clases en vez de los enumerados integrados en el lenguaje .....	56
7.2.3 Repositorios.....	57
7.2.4 Entity Framework Core.....	58
7.2.5 Eventos de dominio .....	61
7.2.6 Eventos de integración .....	65
7.3 Patrón-Arquitectura CQRS .....	68
7.3.1 Los patrones Command y Command Handler.....	69
7.3.2 Lanzar y procesar el comando .....	71
7.3.3 Patrón mediador.....	72
7.3.4 MediatR como implementación del patrón mediador .....	72
7.3.5 Registro de MediatR en el contenedor de dependencias .....	75
8. DESARROLLO DE LA SOLUCIÓN “TITÁN” .....	77
8.1 Visión general de la arquitectura.....	77
8.2 Microservicio de identidad.....	78
8.3 Microservicio de gestión de solicitudes .....	78
8.4 Microservicio de gestión de proyectos .....	79
9. DESPLIEGUE DE LA SOLUCIÓN .....	80
9.1 Orquestadores .....	80
9.1.1 Kubernetes .....	80
9.1.2 Azure Kubernetes Service (AKS).....	84
9.2 Del código al orquestador: desplegando la solución .....	84
9.2.1 Definir la compilación de varias imágenes a la vez con Docker-compose .....	84

9.2.2 Azure Pipelines: Definición de la Build .....	85
9.2.3 Azure Pipelines: Definición de la Release .....	88
10. TESTING.....	89
10.1 Pruebas unitarias .....	89
10.1.1 Mocks.....	90
10.1.2 Stubs.....	90
10.1.3 Implementando pruebas unitarias.....	90
10.2 Pruebas funcionales y de integración .....	91
10.2.1 Implementación de pruebas funcionales y de integración.....	91
11. CONCLUSIONES .....	93
11.1 Responsabilidad.....	93
11.2 Coste y planificación.....	94
12. LÍNEAS FUTURAS .....	96
13. REFERENCIAS.....	97
14. ANEXOS .....	100
14.1 Seedwork .....	100
14.1.1 La case base Entity.....	100
14.1.2 La clase base Enumeration .....	102

# 1. INTRODUCCIÓN

## 1.1 Estado actual

La tendencia natural que ha seguido la informática a nivel de hardware, tanto en el mundo del consumo personal como en el profesional, ha sido el conseguir dispositivos cada vez más potentes, de eso no tenemos duda, pero a la vez más ligeros y portátiles.

La evolución de las redes de comunicaciones ha facilitado que estos dispositivos estén siempre conectados haciendo que PCs, portátiles, móviles y tabletas evolucionen desde aplicaciones tradicionales y servicios *on-premise* a la proliferación de las aplicaciones móviles y servicios en el Cloud.

En los últimos años esta tendencia se ha consolidado y hemos pasado, no solo a ejecutar los servicios en la nube y acceder con nuestros teléfonos y dispositivos desde cualquier localización, sino a acceder a servicios globalizados, que necesitan que nuestros datos estén disponibles desde cualquier lugar por lo que tienen que replicarse geográficamente.

Con la llegada del internet de las cosas aparece un nuevo reto: la necesidad de escalar de forma rápida pues surgen picos de uso de manera impredecible. Millones de dispositivos conectados, enviando cantidades colosales de datos en tiempo real que necesitan ser distribuidos, procesados y replicados en cuestión de milésimas de segundo.

### 1.1.1 Evolución de los servicios hasta la era actual

La evolución de las aplicaciones en los últimos cincuenta años ha sido notoria. Como podemos ver en la siguiente figura, pasamos de aplicaciones ejecutándose completamente en el mismo ordenador, a la era del cliente servidor evolucionando hasta la era de la web.

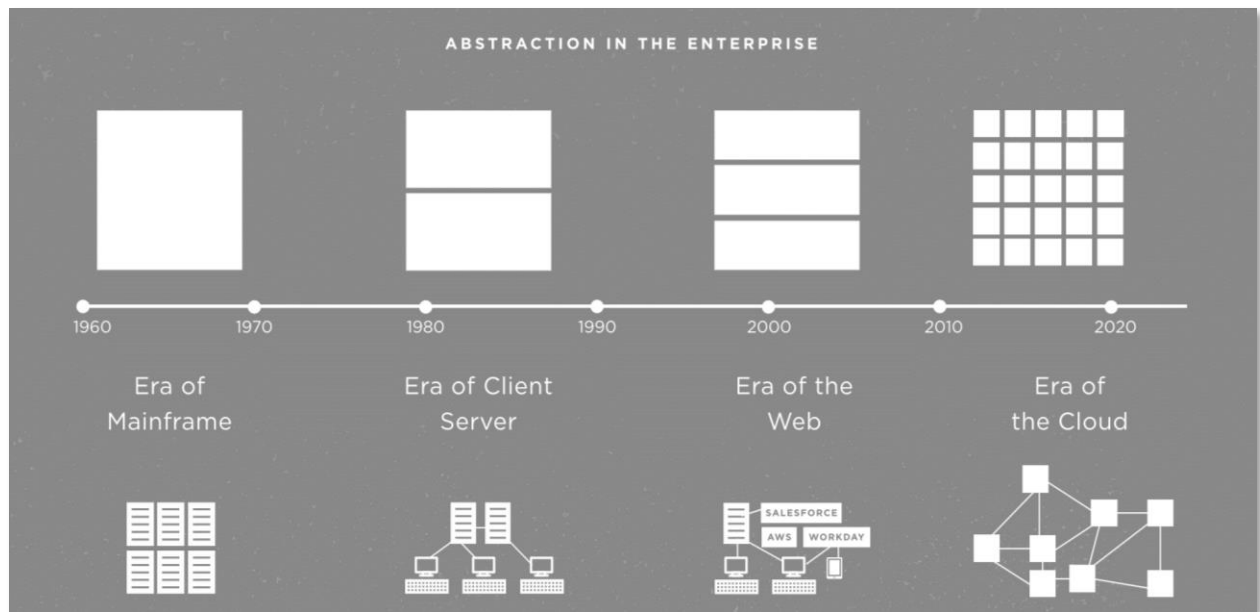


Figura 1-1. Evolución de los servicios hasta la era del Cloud. Fuente: Sequoia

Hasta este momento las arquitecturas de las aplicaciones eran consideradas *monolíticas*.

Con arquitectura monolítica no nos referimos a que las aplicaciones se desarrollen organizadas en un único módulo. Pueden, de hecho, estar compuestas y/o estructuradas como distintas bibliotecas, componentes o incluso capas (capa de aplicación, capa de dominio o negocio, capa de acceso a datos...). El término monolítica se aplica al despliegue, es decir, si se despliegan como un único proceso, una única aplicación web o un servicio único.

Este enfoque monolítico tiene las siguientes implicaciones:

- Las aplicaciones grandes son complejas.
- Cada vez que queremos realizar una actualización tenemos que redespargar la aplicación entera.
- La dificultad para escalar partes individuales obliga a que cada vez que queremos escalar nuestra aplicación tenemos que desplegar más versiones enteras de la misma aplicación.
- Un fallo en cualquier módulo puede tirar el proceso entero abajo.
- Se tienen mayores dificultades para adoptar nuevas tecnologías.

## 1.2 Metodologías ágiles y DevOps

Además de la evolución de la que hemos estado hablando, a lo largo de estos años también ha habido una evolución en la forma de desarrollar software. Se ha pasado de la metodología de desarrollo en cascada (Requisitos, Diseño, Codificación, Pruebas, Integración, Operación y Mantenimiento) a un proceso de desarrollo mucho más ágil, en el que todas las fases del proceso de cascada (que siguen teniendo sentido y son necesarias) están incluidas pero que no aparecen una única vez sino que se retroalimentan formando ciclos.

Con esta metodología, el desarrollo de una aplicación se divide en partes mucho más pequeñas, construyendo soluciones intermedias en ciclos de tiempo cortos. Con ello se consigue incorporar las modificaciones con mayor flexibilidad sin tener que esperar a tener toda la aplicación construida. Esto nos lleva a hablar de *mejora continua* y *entrega continua* siendo, por tanto, la retroalimentación recibida la parte primordial para tener en cuenta a la hora de decidir cómo continuar.

En estas metodologías ágiles encontramos el término *DevOps*, que es la unión de desarrollo y operaciones. Es una práctica que tiene como objetivo unificar el desarrollo y las operaciones, automatizando todas las fases e integrando como parte del desarrollo de una aplicación su despliegue en distintos entornos (como producción o pruebas) para conseguir mayor seguridad y rapidez.

Una de las fases de DevOps incluye la monitorización de los servicios desplegados y obtener *feedback* de cómo está actuando una versión de nuestro código para utilizarlo en el siguiente ciclo.

### 1.3 Cloud computing

Para poder adecuarnos a todas las necesidades de la sociedad actual, el concepto de *Cloud computing* o *computación en la nube* es básico. Inicialmente empezó como algo que permitía pasar de tener los servidores y centros de datos en el cuarto de servidores de nuestras oficinas y centros educativos a tenerlos alojados en Centros de Procesamientos de Datos (CPD) en cientos de máquinas virtuales, de una forma segura y con grandes mejoras en la disponibilidad y el mantenimiento.

Actualmente, además de seguir con la posibilidad de tener máquinas virtuales en estos centros de datos a lo largo del planeta, la cantidad de servicios que ofrecen los distintos proveedores Cloud ha experimentado un crecimiento exponencial y son cada vez más difíciles de enumerar. Tenemos servicios de infraestructura (*IaaS*), servicios de plataforma (*PaaS*) en los que no tenemos que administrar los servidores y nos encargamos simplemente de manejar y desplegar nuestras aplicaciones, o software como servicio (*SaaS*) que nos permiten, por ejemplo, enviar un correo o un SMS con un API.

Las listas de servicios *PaaS* y *SaaS* incluyen bases de datos, almacenamiento, análisis de datos, Inteligencia Artificial y *Machine Learning*, mensajería entre nuestras aplicaciones, servicios de identidad, servicios multimedia, gestión de redes virtuales... Como se ha comentado, esta lista es cada vez más difícil de enumerar y es aconsejable referirse a las páginas web de los distintos proveedores para tener una visión más clara de qué servicios ofrece cada uno: *Microsoft Azure* [5], *Amazon Web Services(AWS)* [6], *Google Cloud* [7].

### 1.4 Y ahora, ¿qué?

Como consecuencia y respuesta a las situaciones que hemos ido encontrando hasta este momento, han surgido dos nuevas tendencias que, aunque pueden funcionar muy bien por separado, juntas consiguen explotar al máximo potencial: **una es la arquitectura basada en microservicios y otra el despliegue de aplicaciones en contenedores.**

Ambas tendencias (de las que se hablará largo y tendido en este trabajo) permiten solventar los problemas que presentan las arquitecturas monolíticas, adoptar nuevas tecnologías y escalar, desplegar y poner en producción de una forma mucho más rápida y eficiente.

### 1.5 Organización del proyecto

Los capítulos de la memoria están organizados de la siguiente forma:

1. Introducción
2. Objetivos y motivación
3. Se expone la lista de las herramientas usadas a lo largo del proyecto, frameworks como ASP.NET Core, IDE como Visual Studio 2017 y Visual Studio Code, Docker para la gestión de los contenedores y Azure DevOps para todo lo que involucra gestión del proyecto.
4. Metodología: se expone la metodología ágil usada, scrum, y se expone cómo se usan las herramientas, en este caso Azure DevOps, para la gestión de tareas y del código.

5. Se explican los contenedores, en que están basados y se comparan con las máquinas virtuales. Se introduce como trabajar con los contenedores usando Docker y como crear los proyectos ASP.NET Core que se utilizan más adelante.
6. Se definen las arquitecturas de microservicios, cómo definir el dominio y el ámbito de cada microservicio, comunicaciones entre microservicios y el exterior y las ventajas y desventajas de usarlos.
7. Se presentan patrones de código para diseñar las arquitecturas de microservicios, como inyección de dependencias e inversión de control, CQRS o patrones que suelen ser usados junto a DDD, entre otros.
8. Se introduce la aplicación de ejemplo desarrollada, su arquitectura a alto nivel y los patrones aplicados en cada uno de los microservicios.
9. Dónde y cómo se ha desplegado la solución de ejemplo. Se definen los orquestadores, Kubernetes específicamente y la implementación en la nube usada, Azure Kubernetes Service. También se continua lo explicado en el capítulo 4 ampliando cómo se usa Azure DevOps para la compilación y despliegue del código.
10. Definición del tipo de pruebas que se aplican al código y ejemplos de como se han implementado en la aplicación de ejemplo desarrollada.
11. Conclusiones, incluyendo las responsabilidades y coste y planificación.
12. Líneas futuras, donde se exponen distintas formas de ampliar lo presentado en este trabajo.
13. Referencias, las cuales se han ido enlazando a lo largo de la memoria.
14. Anexos

## 2. OBJETIVOS Y MOTIVACIÓN

Tras haber desarrollado aplicaciones y servicios desplegados principalmente en entornos Cloud en mi entorno laboral y analizar y afrontar situaciones en las que la carga de usuarios en partes específicas de la aplicación colapsa el servicio entero, se identifican unas guías y patrones para garantizar la escalabilidad y desacoplamiento de las aplicaciones.

### 2.1 Cómo surge

Este proyecto surge debido a la actual necesidad de tener una alta disponibilidad, escalabilidad y posibilidad de despliegue de nuevas versiones en tiempo mínimo. Hoy en día es necesaria una mayor flexibilidad y una rápida adaptación al cambio. Es primordial abordar el desarrollo de aplicaciones y servicios de una forma sencilla y potente, nuevos patrones de carga y distintos usos en un periodo corto de tiempo. Por tanto, el objetivo principal de este trabajo es exponer la arquitectura en la que usando estos patrones y metodologías conseguimos esa alta disponibilidad y facilidad de escalado.

La curiosidad por el empleo de tecnologías en auge, como pueden ser los contenedores y los entornos Cloud, es también un factor motivador. La forma de desarrollo, despliegue y la necesidad de que la configuración de la infraestructura donde vamos a desplegar nuestra solución también sea definible mediante código y/o ficheros, ha hecho que la tecnología y las herramientas hayan mejorado para traernos a este punto. A lo largo de este trabajo veremos cómo somos capaces de definir explícitamente nuestro entorno de producción y cómo, a la hora de desarrollar, podemos tener un entorno exactamente igual de manera sencilla y potente.

Por último, pero no por ello menos importante, este proyecto incide en la necesidad de adaptarnos al mundo en el que vivimos actualmente. Nos movemos hacia un paradigma en el que estamos siempre conectados y queremos acceder a nuestros datos desde cualquier lugar o dispositivo, y lo queremos ya. En este trabajo se expone una manera de desarrollar soluciones en las que podremos estar disponibles globalmente para nuestros clientes y usuarios y de una forma en la que, en vez de adaptarnos a los cambios, simplemente fluiremos con ellos.

### 2.2 Objetivo general

El objetivo de este proyecto fin de grado (PFG) es presentar cómo definir arquitecturas basadas en microservicios para aplicaciones desplegadas en contenedores utilizando buenas prácticas. Con ello se consigue asegurar la alta disponibilidad, escalabilidad y despliegue de nuevas versiones en un tiempo mínimo.

En concreto, se realizará utilizando *Docker* como motor para nuestros contenedores y el ecosistema de código abierto multiplataforma de Microsoft: *.NET Core*. Todo lo aprendido con este proyecto es aplicable a diversos lenguajes y tecnologías con las que será posible desarrollar las aplicaciones y servicios.



El propósito de este proyecto por tanto es presentar una arquitectura para aplicaciones basadas en microservicios, desplegadas en contenedores y pensadas para entornos Cloud. La aplicación resultante será un soporte para mostrar el diseño de esta arquitectura más que un producto final en sí mismo.

## 2.3 Objetivos específicos

Los objetivos específicos son los siguientes:

1. Explicar qué es un contenedor y qué funciones del sistema operativo permiten su utilización, tanto en **Linux** como en **Windows**.
2. Explicar qué es una imagen y cómo las definen los distintos motores de contenedores (*Docker*, *CoreOS' rkt*...). El foco estará en *Docker* dado que es el motor más extendido y aceptado en los distintos orquestadores.
3. Explicar los beneficios de los contenedores, como son la rápida replicación, la capacidad de definir en código las necesidades y especificaciones de los entornos o eliminar la necesidad de un hipervisor para obtener entornos aislados.
4. Explicar arquitecturas basadas en microservicios, definiendo el ámbito de cada microservicio y los protocolos y tipos de comunicación entre ellos.
5. Presentar patrones de código y buenas prácticas a la hora de desarrollar microservicios. Estos patrones y buenas prácticas serán también aplicables a otras arquitecturas.
6. Desarrollar una aplicación, que sirva de ejemplo ilustrativo, basada en una arquitectura de microservicios y desplegada en contenedores, usando para ello los patrones y buenas prácticas explicados en el proyecto.
7. Desplegar la aplicación de ejemplo en un orquestador, definiendo el entorno a través de código y ejecutándola en un entorno en la nube.
8. Explicar cómo probar el código de una forma eficiente gracias al uso de patrones y buenas prácticas a la hora del desarrollo.

## 3. HERRAMIENTAS USADAS

En este capítulo hablaremos de las herramientas y software utilizados para la realización de este proyecto. Una de las razones principales por las que han sido escogidas es por el hecho de ser multiplataforma y la mayor parte de ellas de código abierto.

### 3.1 El IDE: Visual Studio 2017

Dado que para el desarrollo de nuestro proyecto utilizaremos tecnologías .NET (.NET Core en este caso) el IDE por excelencia en entornos Windows es Visual Studio. Un punto a su favor es que está perfectamente integrado con el resto de las herramientas que utilizaremos, como *Docker*, usado a la hora de compilar, desplegar y depurar nuestras aplicaciones en los contenedores.

Además, Visual Studio también se empleará, para gestionar nuestros repositorios de código fuente, asociar los cambios a nuestras tareas del proyecto para mantener una gestión de este más completa o incluso configurar los entornos en los que publicaremos nuestro proyecto, como por ejemplo la nube de Microsoft: Microsoft Azure.

En cuanto a la licencia, elegiremos la versión *Community* que es completamente gratuita para Estudiantes, Desarrolladores Individuales o equipos de no más de 5 personas y también, por supuesto, para contribuir a proyectos de código libre sin importar cómo de grande sea la organización.

A la hora de instalar Visual Studio 2017 tenemos que asegurarnos de que instalamos la funcionalidad de desarrollo con .NET Core multiplataforma: “.NET Core cross-platform development”.

### 3.2 Visual Studio Code

Es un editor de código fuente ligero y potente disponible en Windows, macOS y en Linux. Viene con soporte para JavaScript, TypeScript y NodeJS y tiene un rico ecosistema de extensiones con soporte para otros lenguajes como C++, C#, Java, Python, PHP, Go... y *runtimes* como .NET y Unity.

También cuenta con extensiones que permiten gestionar nuestros repositorios Git, trabajar con VSTS, publicar y depurar aplicaciones en los distintos Cloud como son Microsoft Azure, Amazon Web Services y Google Cloud entre otros.

### 3.3 .NET Core

.NET Core es una implementación de código abierto, multiplataforma y modular del .NET Framework. Contiene muchas de las APIs de .NET Framework e incluye el runtime, framework, el compilador y otras herramientas que soportan diversos sistemas operativos y arquitecturas.

Las principales características de .NET Core por las que ha sido elegido en este proyecto son:

- **Multiplataforma:** .NET Core proporciona funcionalidades clave para poder implementar nuestras aplicaciones y reutilizar el código independientemente de si vamos a ejecutarlas en

Windows, macOS o Linux. En nuestro proyecto nos permitirá desarrollar en Windows y posteriormente ejecutar nuestros servicios en contenedores bajo el sistema operativo Linux.

- **Código abierto:** .NET Core es uno de los proyectos disponibles en *GitHub*. Al tratarse de un proyecto de código abierto, .NET Core favorece que el proceso de desarrollo sea más transparente y que exista una comunidad activa y comprometida.
- **Modular:** .NET Core es modular y se distribuye a través de *NuGet* en paquetes más pequeños centrados en características más específicas en lugar de un ensamblado grande que contiene la mayor parte de la funcionalidad básica. Uno de los objetivos que queremos conseguir con la arquitectura de microservicios usando contenedores es desplegar justo y únicamente lo que necesitamos para ejecutar nuestros servicios. De esta forma obtenemos compilaciones optimizadas, con mejoras en seguridad, rendimiento y menores costes.

### 3.4 ASP.NET Core y ASP.NET Core MVC

ASP.NET Core es un framework multiplataforma, de alto rendimiento y de código libre para desarrollar aplicaciones y servicios modernos, basados en entornos Cloud y conectados a Internet.

Al estar construido sobre .NET Core tenemos beneficios como ser multiplataforma y de código abierto disponible en GitHub.

ASP.NET Core presenta, entre otras, las siguientes ventajas:

- Forma unificada para crear tanto interfaces de usuario Web como APIs.
- Inyección de dependencias directamente integrada en el framework (hablaremos de este importante patrón más adelante).
- Sistema de configuración extensible con distintas fuentes de datos basado en el entorno y preparado para entornos Cloud.
- Un pipeline HTTP ligero, de alto rendimiento y modular.
- Posibilidad de ejecutarlo por sí mismo en su propio proceso o junto a servidores web como IIS, Nginx, Apache en sistemas Windows, Linux o macOS.

ASP.NET Core MVC se configura sobre el pipeline de ASP.NET Core y ofrece las siguientes funcionalidades:

- El patrón Modelo-Vista-Controlador (MVC del que hablaremos más adelante) que permite hacer nuestras APIs y aplicaciones web testeables.
- Soporte para múltiples formatos de datos y negociación de contenido permitiendo a tus APIs trabajar con un rango amplio de clientes, incluyendo navegadores web y móviles.
- Mapea directamente los datos de la petición HTTP a parámetros en los métodos de nuestros controladores.
- Validación de los modelos tanto en el lado del cliente como en el lado del servidor.

## 3.5 Docker

Hablaremos de *Docker* largo y tendido en un capítulo posterior. En lo referente a su uso durante el proceso de desarrollo será necesario instalar *Docker Community Edition for Windows* [8]. Para su uso en las máquinas Linux y el despliegue en contenedores de servidores Linux necesitaremos *Docker Community Edition for Linux* [9].

## 3.6 Visual Studio Team Services (VSTS), ahora llamado Azure DevOps

*Visual Studio Team Services* es la evolución de *Team Foundation Service (TFS)* completamente basado en la nube. *VSTS* es un conjunto de herramientas de colaboración que sirve para planificar, desarrollar y administrar proyectos de software de cualquier tamaño, en cualquier lenguaje de programación.

Durante la realización de este proyecto, el producto fue renombrado a **Azure DevOps**, integrándolo aún más en la nube de *Microsoft Azure* [5] pero manteniendo las funcionalidades de este.

Está completamente integrado con Visual Studio, tanto la versión 2017 como Visual Studio Code, lo que permite la cobertura total del proceso de gestión de vida de las aplicaciones (ALM).

### 3.6.1 Control de versiones

El control de código fuente o control de versiones permite a los desarrolladores colaborar en el código y mantener un control de los cambios en el código fuente. El control de versiones es una herramienta esencial en proyectos con múltiples desarrolladores, aunque es cierto que también lo es en proyectos gestionados por un único desarrollador.

En Azure DevOps tenemos dos tipos de controles de código:

- **Git:** un sistema distribuido que permite a cada desarrollador tener una copia local del repositorio, crear ramas o bifurcaciones locales, ir guardando el progreso poco a poco en local o elegir sincronizarlo con el servidor y posteriormente mover estos cambios a nuestra rama principal.
- **Team Foundation Version Control (TFVC):** un sistema centralizado cliente servidor.

Ambos nos permiten guardar los cambios en los ficheros y organizar los ficheros en carpetas, ramas y repositorios.

### 3.6.2 Gestión del proyecto y herramientas Ágiles

Los proyectos de desarrollo de software necesitan formas sencillas para compartir información y gestionar el estado del trabajo, funcionalidades, tareas, problemas o defectos en el código.

Para poder planificar nuestros proyectos en Azure DevOps se dispone de distintos tipos de backlogs y tableros para trabajar con las principales metodologías ágiles: Scrum, Kanban o Scrumban:

- **Product backlog:** usado para crear y priorizar historias de usuario o requerimientos.

- Kanban: usado para visualizar y gestionar el flujo de trabajo desde que se crea hasta que se marca como completado.
- Spring backlog: usado para planificar el trabajo a realizar durante un ciclo, denominado sprint. Una cadencia de unas 2 a 4 semanas que los equipos siguen cuando implementan *Scrum* (metodología ágil que se explicará en el próximo capítulo).
- Tablero de tareas: usado durante las reuniones diarias del proceso de Scrum para revisar el trabajo completado, el pendiente o el que está bloqueado.

En el siguiente capítulo se verá cómo se han usado estas funcionalidades para la gestión de este proyecto.

### 3.6.3 DevOps e integración continua

El despliegue rápido y seguro de software proviene de automatizar al máximo los procesos necesarios para ello. Con Azure DevOps podemos automatizar la compilación, el testeo y el despliegue.

- Podremos definir compilaciones que se ejecuten por ejemplo cuando un miembro del equipo sube una nueva versión del código.
- Además de compilar el código, también podremos añadir instrucciones para que se ejecuten nuestras pruebas después de que se complete la compilación.
- También podremos definir procesos el despliegue de las compilaciones de nuestro código a distintos entornos, ya sean de pruebas o producción cuando se dan determinadas circunstancias.

Más adelante veremos la forma en la que utilizamos estas capacidades en el proyecto para compilar y desplegar nuestro código a los distintos entornos.

## 4. GESTIÓN DEL PROYECTO

En este capítulo hablaremos de cómo se ha gestionado el proyecto en términos de organización y planificación además del seguimiento de tareas, generación y análisis de reportes y otros datos necesarios para realizar un seguimiento del estado del proyecto y poder planificar siguientes pasos. La herramienta que se ha utilizado para la gestión es la introducida en el capítulo anterior: Azure DevOps. En este capítulo explicaremos el uso de la herramienta para gestionar proyectos de este tipo.

### 4.1 Metodologías ágiles: Scrum

La metodología que se ha utilizado para planificar y gestionar el proyecto ha sido Scrum. Scrum es la metodología ágil más usada en los proyectos de desarrollo software. A diferencia del modelo de desarrollo tradicional, cuando hablamos de Scrum se tiene en cuenta que:

- El producto se desarrolla de forma incremental, al contrario que en las metodologías tradicionales en cascada, donde se planifica y desarrolla el producto de forma completa.
- Las fases de análisis, diseño, desarrollo y pruebas se solapan en cada iteración, consiguiendo una parte del producto completa en cada una.

Cuando desarrollamos con el modelo tradicional, empezaríamos en una fase inicial de diseño del proyecto y análisis con la planificación del proyecto y, acto seguido, comenzaría el desarrollo. De esta forma la planificación sería estática y poco flexible, lo que la mayoría de las veces provoca que esta sea imprecisa al faltarnos información que iremos descubriendo a lo largo del proceso de desarrollo del proyecto.

Con el modelo ágil, dispondremos de una planificación a largo plazo y más flexible del proyecto en nuestro *product backlog* donde podremos visualizar el proyecto de forma global, las funcionalidades que ya han sido implementadas, las que aún nos faltan por implementar... y una planificación mucho más detallada, a corto plazo, una por iteración o *sprint*, donde tendremos cada una de las funcionalidades a implementar en esta iteración separadas en tareas concretas que serán asignadas a miembros del equipo.

Al planificar el proyecto usando esta metodología nos hacemos mucho más tolerantes a los cambios dado que no desarrollamos al detalle una funcionalidad hasta que llega el momento de ser implementada. Esto nos permite elegir, atendiendo a las necesidades, en qué iteración incorporaremos una parte de las que componen nuestro proyecto final, si introducimos modificaciones, si eliminamos un elemento, etc.

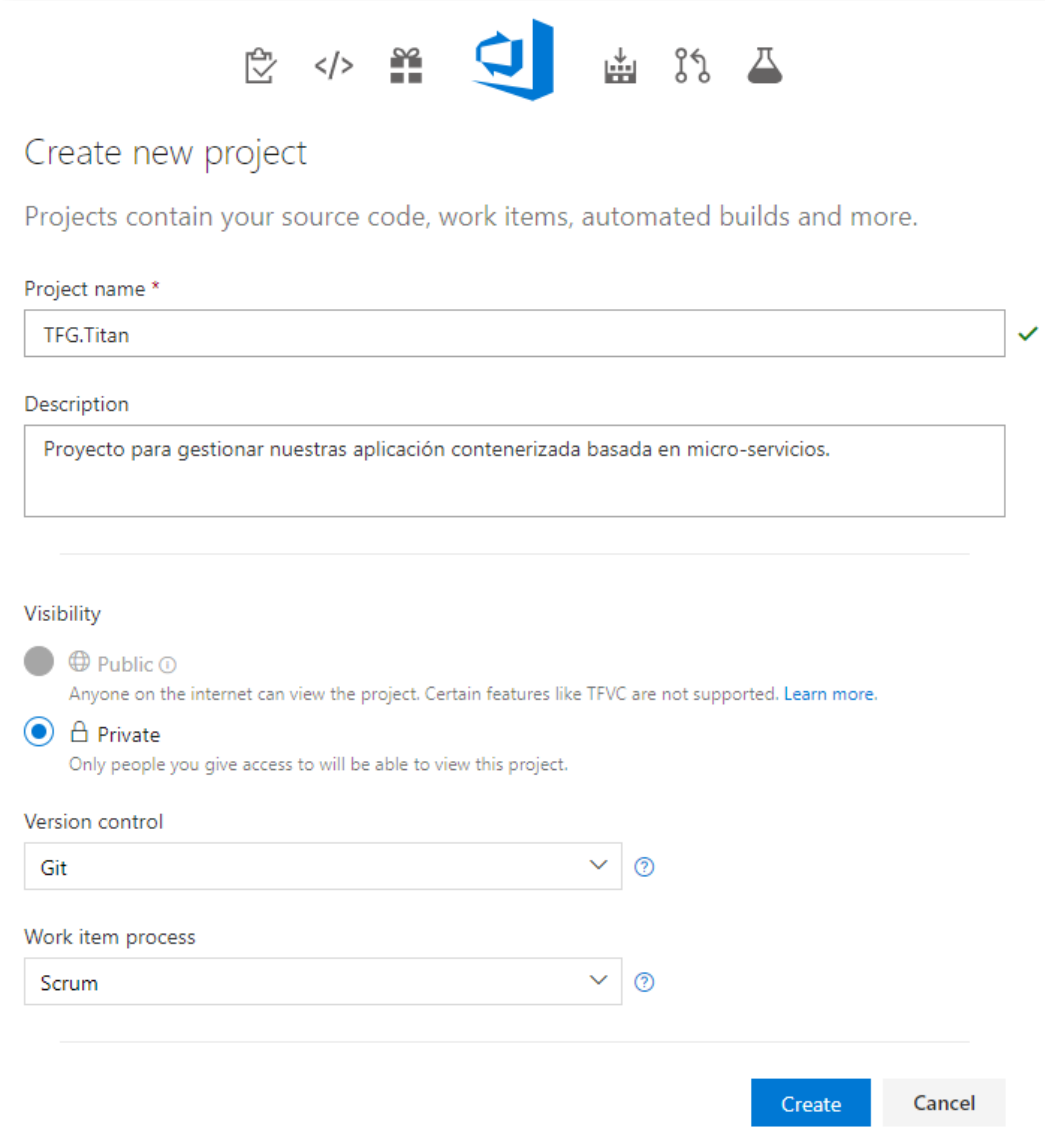
Esta metodología es adecuada para proyectos en los que podremos tener distintos equipos trabajando en tareas de gran envergadura, ya que son muy propensos a sufrir cambios a lo largo del desarrollo. Esto se debe a que haya muchos factores que pueden afectar al proyecto pero que no están presentes al inicio del mismo, el rumbo que tomará el proyecto se irá adaptando en función de la información que vayamos recopilando, los cambios del mercado y tendencias, impedimentos que vayamos

encontrando y nuevas tecnologías que iremos incorporando, etc. y, por supuesto, fruto del del conocimiento y dominio de nuestras aplicaciones,

Ahora veremos cómo gestionamos nuestro proyecto con la herramienta que se ha elegido, en este caso, *Azure DevOps*[10].

## 4.2 Azure DevOps: Creando el proyecto y equipo

Para empezar a gestionar nuestro proyecto lo primero es, como podríamos esperar, crearlo. Para ello accederemos a nuestra cuenta de Azure DevOps y crearemos un nuevo proyecto donde poder almacenar nuestros repositorios de código y empezar a gestionar y planificar:



The screenshot shows the 'Create new project' dialog in Azure DevOps. At the top, there is a navigation bar with icons for various DevOps functions. The main heading is 'Create new project', followed by a subtitle: 'Projects contain your source code, work items, automated builds and more.' Below this, there are several configuration options:

- Project name \***: A text input field containing 'TFG.Titan' with a green checkmark to its right.
- Description**: A text input field containing 'Proyecto para gestionar nuestras aplicación contenerizada basada en micro-servicios.'
- Visibility**: Two radio button options. The 'Public' option is selected, with a note: 'Anyone on the internet can view the project. Certain features like TFVC are not supported. [Learn more.](#)' The 'Private' option is also visible, with a note: 'Only people you give access to will be able to view this project.'
- Version control**: A dropdown menu showing 'Git' with a help icon to its right.
- Work item process**: A dropdown menu showing 'Scrum' with a help icon to its right.

At the bottom right, there are two buttons: 'Create' (highlighted in blue) and 'Cancel'.

Figura 4-1. Creación de proyecto en Azure DevOps

A la hora de crearlo se ha elegido Scrum como proceso.

Respecto al sistema utilizado como control de versiones (*version control*), hemos elegido el sistema distribuido por excelencia “*Git*”. Posteriormente detallaremos cómo utilizamos este sistema para publicar los cambios en producción.

*Azure DevOps*[10] permite que nuestro proyecto sea privado (nosotros elegimos los miembros del proyecto) o público (todo el mundo es capaz de acceder a nuestro proyecto). La visibilidad pública es comúnmente usada en proyectos de código abierto en los que todo el mundo puede ver el código y aportar.

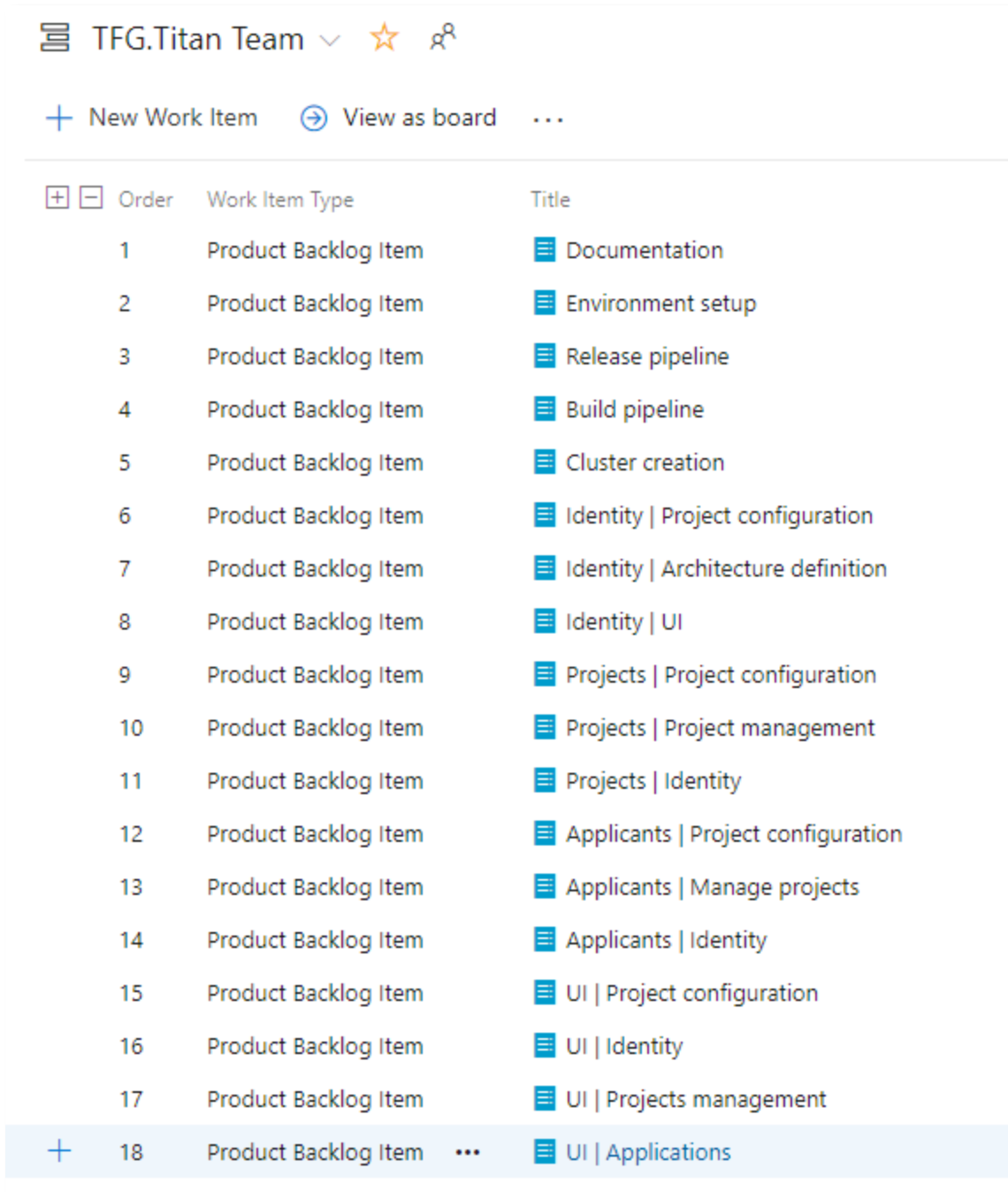
### 4.3 Azure Boards: definiendo el Product Backlog

Una de las cinco grandes partes de Azure DevOps [10] es Azure Boards [11]. En ella tenemos disponibles las herramientas ágiles para poder planificar, hacer seguimiento y discutir el progreso con diferentes equipos.

Como ya hemos dicho, una de las características que definen la metodología Scrum es el *Product Backlog*. En él tendremos los *Product Backlog Items* (elementos del *Product Backlog*), que definen las distintas funcionalidades o bloques de trabajo. Estos bloques serán separados posteriormente en tareas más específicas y concretas. Pueden corresponder a los casos de uso de nuestro proyecto, a elementos de infraestructura o, a elementos propios de gestión (entre otros), como puede ser la documentación (que en este caso, correspondería a esta memoria). En el *product backlog* los elementos se organizan por prioridad, colocando los de mayor prioridad arriba. A la hora de definir cada iteración los desarrolladores elegirán los elementos con mayor prioridad, los desgranarán en tareas más concretas y los irán completando a lo largo de la iteración.

En la siguiente figura podemos ver el *product backlog* definido para este proyecto.





TFG.Titan Team				
<a href="#">+ New Work Item</a> <a href="#">View as board</a> ...				
	Order	Work Item Type	Title	
	1	Product Backlog Item		Documentation
	2	Product Backlog Item		Environment setup
	3	Product Backlog Item		Release pipeline
	4	Product Backlog Item		Build pipeline
	5	Product Backlog Item		Cluster creation
	6	Product Backlog Item		Identity   Project configuration
	7	Product Backlog Item		Identity   Architecture definition
	8	Product Backlog Item		Identity   UI
	9	Product Backlog Item		Projects   Project configuration
	10	Product Backlog Item		Projects   Project management
	11	Product Backlog Item		Projects   Identity
	12	Product Backlog Item		Applicants   Project configuration
	13	Product Backlog Item		Applicants   Manage projects
	14	Product Backlog Item		Applicants   Identity
	15	Product Backlog Item		UI   Project configuration
	16	Product Backlog Item		UI   Identity
	17	Product Backlog Item		UI   Projects management
<a href="#">+</a>	18	Product Backlog Item	...	UI   Applications

Figura 4-2. Product Backlog en Azure Boards

#### 4.4 Azure Boards: definiendo los sprints (iteraciones)

Como hemos mencionado, la idea bajo la planificación ágil es la posibilidad de adaptarse al cambio de forma rápida y eficiente. Para ello se van definiendo iteraciones, que requieren cortos periodos de tiempo. El sprint, que así se llama cada iteración, se define como el periodo de tiempo entre una

entrega y la siguiente que comprenderá tareas de análisis, diseño, desarrollo, gestión, documentación, *testing* o pruebas y -muy importante- obtención de *feedback*.

Cuando un sprint está empezando se mantiene una reunión con los miembros del equipo, llamada *sprint planning*, en la que basándose en la capacidad del equipo se revisan y cogen los elementos del producto backlog de mayor prioridad y se descomponen en tareas específicas y concretas a las que se les asigna un valor de esfuerzo. Normalmente este esfuerzo se especifica basándose en las horas que llevará completar la tarea.

Cuando un elemento se añade al sprint es un compromiso. El equipo de desarrollo se está comprometiendo a desarrollar estas tareas en el periodo que dura el sprint y, al final de éste, se obtiene una versión funcional del producto con las características correspondientes a estas tareas, implementadas y completadas.

## Boards

### Iterations Areas

Create and manage the iterations for this project. These iterations will be used by teams for iteration planning (sprint planning). [Learn more about customizing areas and iterations](#)

To select iterations for the team, go to [the default team's settings](#).

New	New child		+	-
Iterations	Start Date	End Date		
▼ TFG.Titan				
Sprint 1	7/1/2018	7/31/2018		
Sprint 2	8/1/2018	8/31/2018		
Sprint 3	9/1/2018	9/30/2018		
Sprint 4	10/1/2018	10/26/2018		
Sprint 5	... 10/29/2018	11/23/2018		
Sprint 6				

Figura 4-3. Gestión de sprints en Azure Boards

Para configurar los sprints en *Azure Boards* tenemos que definir la duración de cada sprint. Desde el portal del proyecto se pueden definir las fechas de inicio y final de cada una de las iteraciones. Como podemos ver en la figura, las iteraciones en este proyecto se han definido con una duración de un mes.

A la hora de la planificación del sprint se debe tener en cuenta la capacidad de trabajo de los miembros del equipo puesto que podremos definir el número de tareas que podrán realizarse en el sprint en base al esfuerzo estimado (horas en nuestro caso) en completar cada tarea.

The screenshot shows the 'Capacity' tab for the 'TFG.Titan Team' in Azure Boards. The sprint duration is 'September 1 - September 30' with '4 work days remaining'. The 'Capacity' tab is active, showing a table with columns: User, Days off, and Capacity per day. Carlos Jiménez Aliaga is listed with 4 days off and a capacity of 2. Below this, it shows 'Team days off' as 0 days, applying to the whole team. A modal titled 'Days off for: Carlos Jiménez Aliaga' is open, showing a table of days off: 9/12/2018 to 9/13/2018 (2 days) and 9/10/2018 to 9/11/2018 (2 days), totaling 4 days. There is an 'Add additional days off' link and 'OK'/'Cancel' buttons.

User	Days off	Capacity per day
Carlos Jiménez Aliaga	4 days	2

Team days off: 0 days. These days off apply to the whole team.

Days off for: Carlos Jiménez Aliaga

Start	End	Days off
9/12/2018	9/13/2018	2
9/10/2018	9/11/2018	2

+ Add additional days off

Total 4

OK Cancel

Figura 4-4. Gestión de la capacidad del equipo en Azure Boards

En la anterior figura podemos ver que para Carlos (el autor de esta memoria) se ha establecido una capacidad de 2 horas al día así como los días que no estará en el proyecto, un total de 4. En este sprint obtendremos una capacidad de 32 horas (16 días de trabajo x 2 horas diarias), y podremos asignar tareas donde la estimación total esté dentro de este número.

Cuando tenemos la capacidad asignada podremos planificar la iteración y obtener datos del nivel de carga de trabajo de cada miembro del equipo, la capacidad restante y también lo asignado a cada miembro del equipo, entre otras métricas y valores.

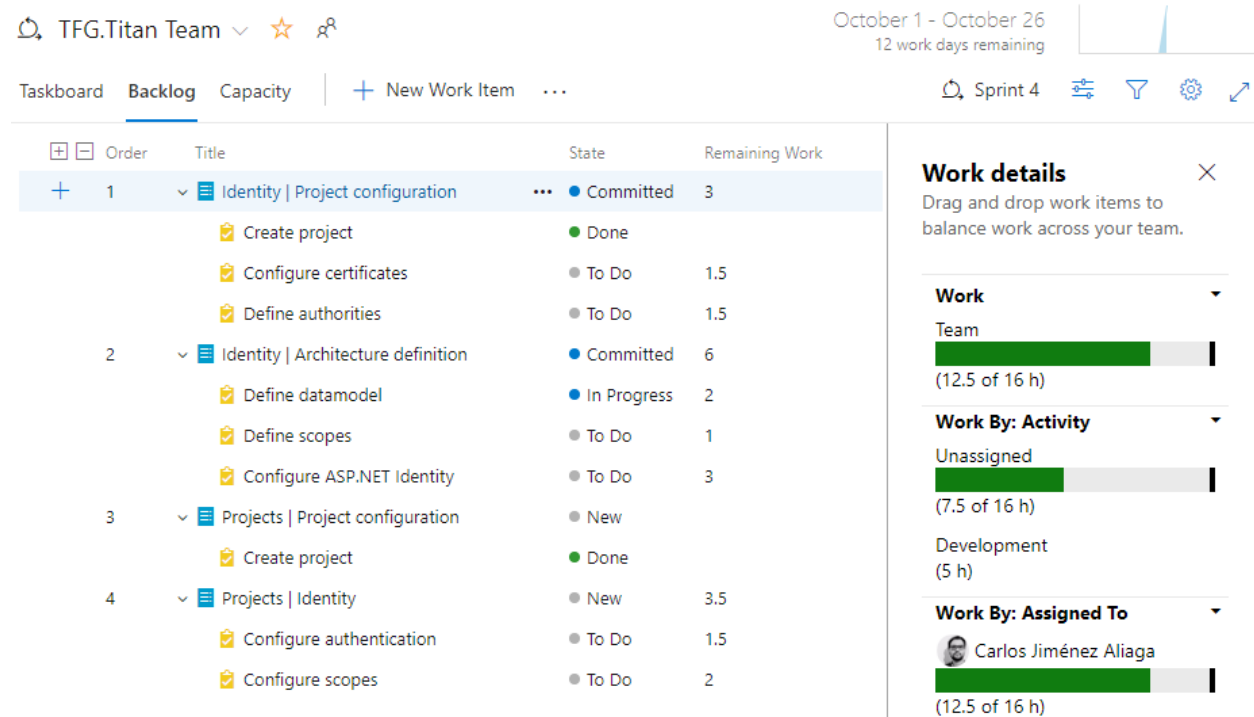


Figura 4-5. Gestión backlog por sprint en Azure Boards

## 4.5 Azure Boards: Taskboard

Taskboard, o tablero de tareas en español, es la funcionalidad que permite ver de forma rápida el estado en el que se encuentra cada una de las tareas de un sprint agrupadas por *Product Backlog Item*.

Los estados disponibles para cada tarea son los siguientes:

- To do: la tarea aún no se ha empezado , está **por hacer**.
- In progress: la tarea está **en progreso** y se está trabajando en ella.
- Done: la tarea ha sido **completada**.
- Removed: aunque la tarea estaba en el proyecto se ha procedido a **eliminarla**.

Desde el tablero de tareas tenemos una visión general del estado del proyecto en todo momento y permite pasar una tarea de un estado a otro, simplemente arrastrando como si fuera un *post-it*, en un tablero de verdad. Si nuestro entorno de trabajo lo permite, tener este tablero siempre abierto será muy útil durante el desarrollo del proyecto.

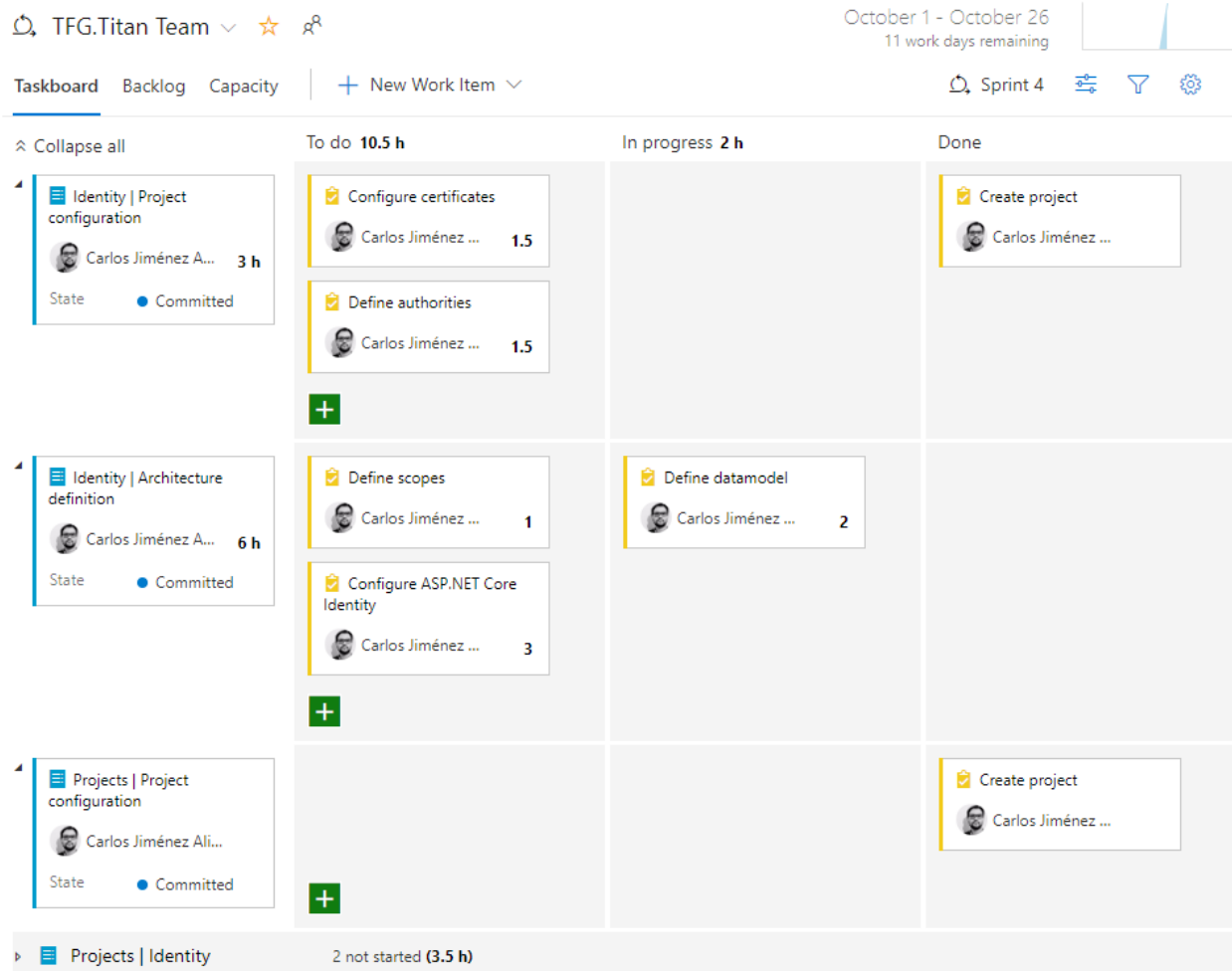


Figura 4-6. Tablero de tareas de un sprint en Azure Boards

En la metodología Scrum tenemos una gráfica que aporta los datos del estado del proyecto. Esta gráfica se llama *burndown chart* y en ella podremos ver la carga de trabajo actual y si se va cumpliendo con las estimaciones realizadas al principio del sprint.

En la siguiente imagen tenemos una gráfica con la fecha en el eje horizontal y las horas restantes en el eje vertical. La diagonal negra que atraviesa el gráfico nos indica el ritmo ideal de progreso y el área azul el tiempo estimado correspondiente a las tareas pendientes por completar. Por último la línea verde indica la capacidad ideal que deberíamos tener para que todo esté perfectamente planificado y ejecutado.

Burndown for: Sprint 4

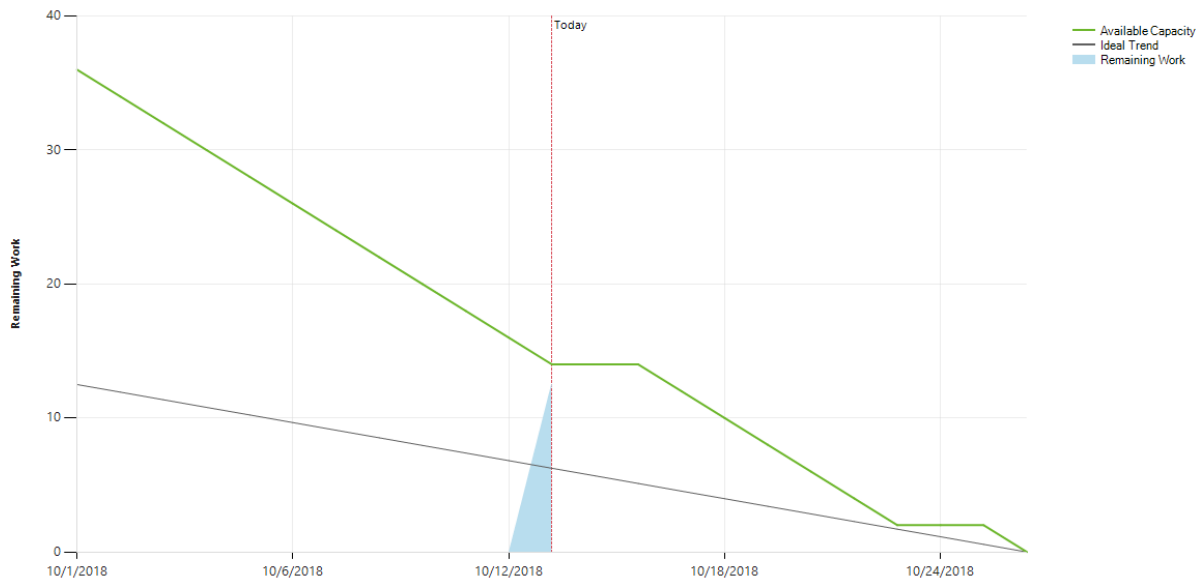


Figura 4-7. Burndown de un sprint en Azure Boards

Resumiendo, toda la información recopilada por las herramientas descritas anteriormente permite ajustar nuestras estimaciones, conocer en detalle cómo trabaja el equipo y, por tanto, hacer unas estimaciones más precisas dejando menos trabajo sin acabar al final de cada sprint.

#### 4.6 Azure Boards: Gestión del código y tareas

Azure Boards permite enlazar nuestras tareas y *product backlog items* a distintos elementos, tanto internos como externos, para tener un mejor seguimiento. Podemos, por ejemplo, enlazar nuestras tareas con cambios en el código, tanto desde el portal online como desde nuestro IDE: Visual Studio 2017.

A la hora de crear el proyecto, seleccionamos Git como sistema de repositorio de código. En este sistema al conjunto de cambios que confirmamos en el repositorio se le llama *"commit"*. Cuando queremos subir este conjunto de cambios podemos asociarlo a una o varias tareas e incluso modificar el estado de las tareas (cambiando, por ejemplo, a completada).

En Git también se pueden hacer bifurcaciones o ramas partiendo del punto que queramos. Este concepto se denomina *"branch"*. En este proyecto la organización de nuestro repositorio ha sido la siguiente:

- Hay una rama maestra o principal llamada *"master"*. Todo lo que esté en esta rama es considerado terminado y, lo que es más importante, desplegado en el entorno de producción.
- Cada vez que se empieza a trabajar en un Product Backlog ítem se crea una rama asociada al mismo. Esto podemos hacerlo desde la interfaz de Azure DevOps:

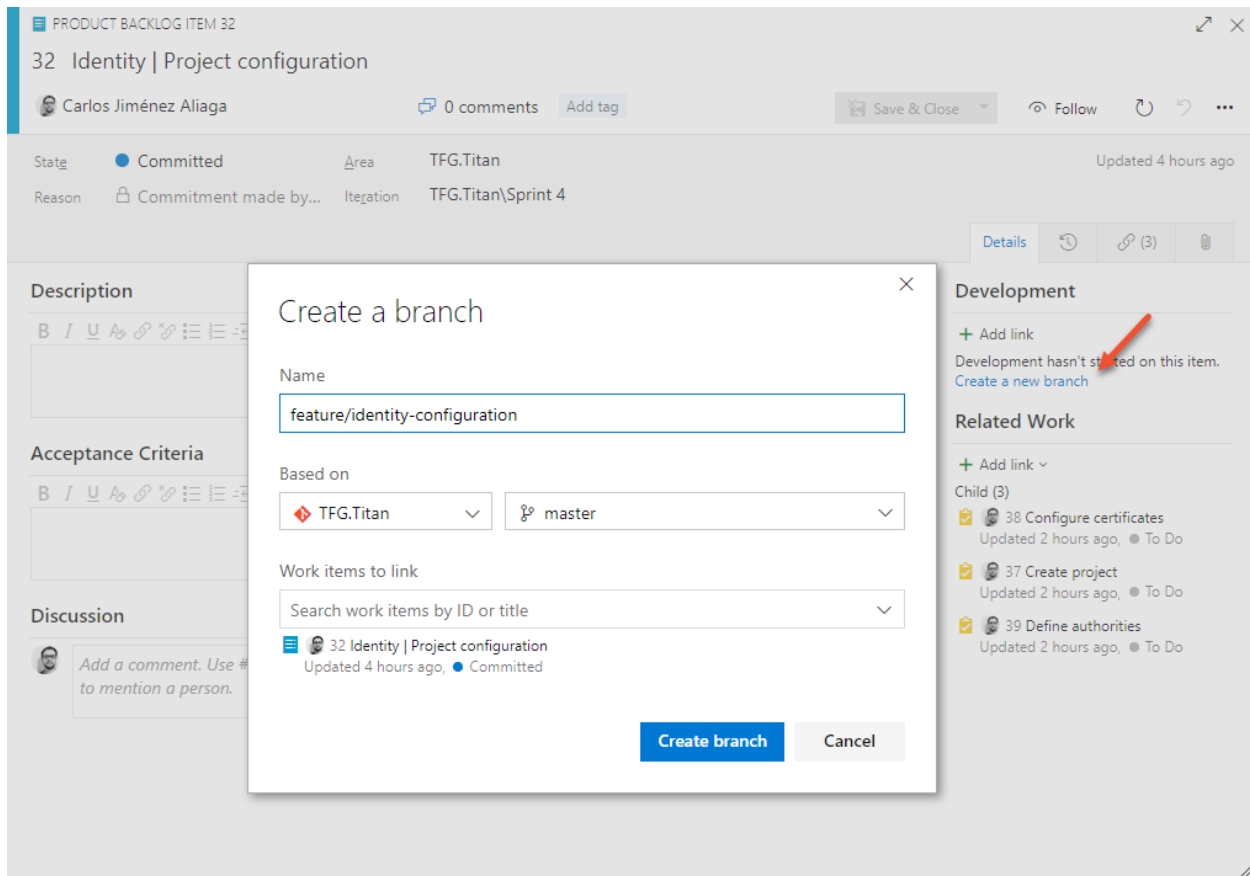



Figura 4-8. Creación de una rama asociada en Azure DevOps

Una vez creada esta rama, se irá trabajando en las tareas y subiendo los distintos cambios asociados a estas tareas a la rama. Cuando se considera que todas las tareas están completas se procederá a comprobar que todos los cambios cumplen los requisitos y que el código cumple con los estándares de calidad del equipo. En caso afirmativo, se procederá a introducir los cambios de esta rama en la rama de destino. A este concepto se le llama *"merge"*, al que en castellano podemos referirnos como fusionar para hablar sobre el proceso de introducir los cambios de una rama en otra.


Para ayudarnos a este proceso, en Azure DevOps [10] se puede crear una *"Pull request"*. En ella podremos ver qué rama de origen queremos fusionar con qué rama de destino, las tareas asociadas a la rama a fusionar y también, y más importante, los cambios en los ficheros.


Podemos definir, tanto en el momento en el que creamos la pull request como a nivel de proyecto o equipo, quiénes serán los encargados de revisar y aceptar los cambios.

## New Pull Request

 feature/projects-configuration

 into 

 master



Title \*









Projects micro-service configuration


Add label

Description

- Create initial projects for projects microservice.
- Adding Autofac and modules.
- Adding seedwork.



Markdown supported.


Aa        @ # 


 Add commit messages



- Create initial projects for projects microservice.
- Adding Autofac and modules.
- Adding seedwork.

Reviewers

 Carlos Jiménez Aliaga  Search users and groups to add as reviewers

Work Items 


Search work items by ID or title 

  25 Projects | Project configuration










Create

Figura 4-9. Creación nueva Pull Request

Una vez creada el pull request, las personas encargadas de revisarlo podrán hacer comentarios a nivel general o líneas específicas de código, proponer cambios, aprobarlo o rechazarlo.

 Instead of just adding MVC could be better to add just the minimum components needed to run the API.

Markdown supported. Drag & drop, paste, or select files to insert.

Aa        @ #  

Comment

Cancel

Instead of just adding MVC could be better to add just the minimum components needed to run the API.

Figura 4-10. Añadiendo un comentario en el código al revisar el pull request.



En el momento en el que se quiera aceptar los cambios, podemos elegir si queremos hacer un único commit con todos los cambios (*squash*) o introducir todos los commits individualmente.

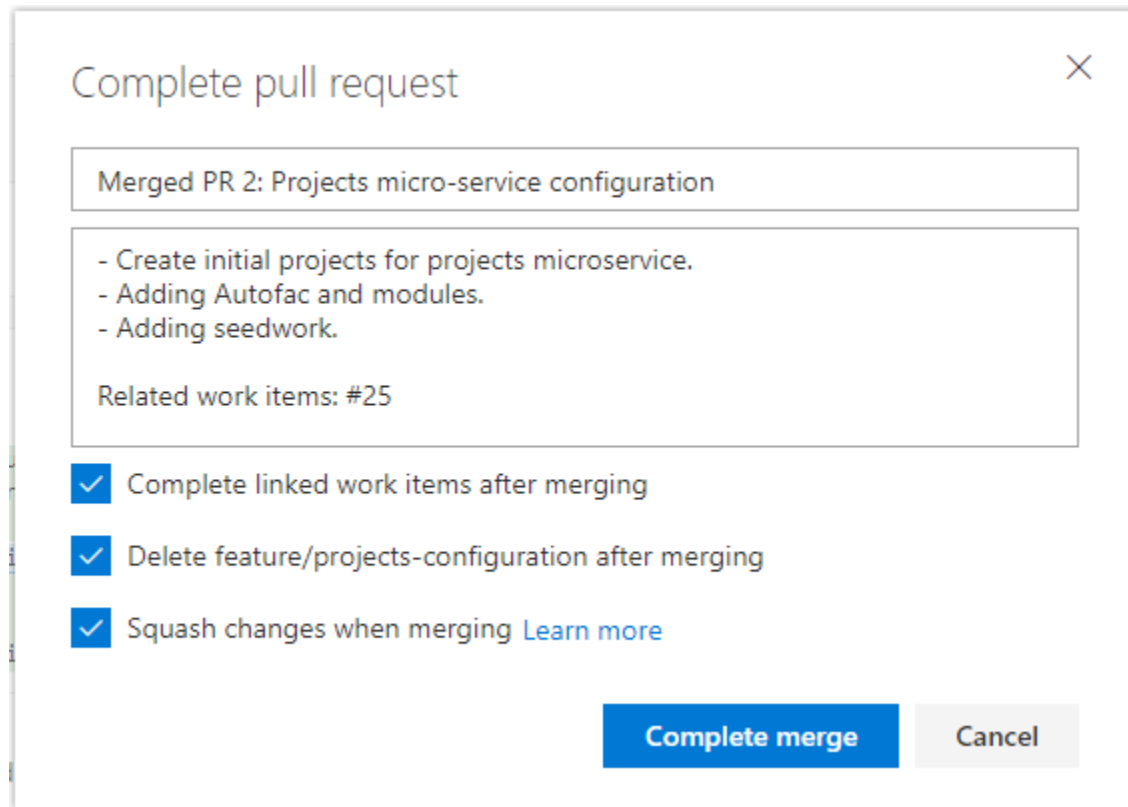


Figura 4-11. Completando pull request

Azure DevOps permite configurar acciones que ocurrirán como consecuencia de distintos eventos. Por ejemplo, podemos configurarlo para que se ejecute una compilación con cada cambio que se suba al servidor y se ejecuten las pruebas unitarias. También podremos hacer que con cada *pull-request* se ejecute una compilación y no permitir que se acepte hasta que no haya errores y la compilación sea correcta.

Veremos en próximos capítulos cómo utilizar estas acciones, no sólo para compilar la solución asegurando que no haya errores, sino además para desplegar el código en los distintos entornos.

## 5. CONTENEDORES Y DOCKER

En el ámbito del desarrollo de software existe un enfoque o aproximación que consiste en empaquetar una aplicación o un servicio junto a sus dependencias y configuraciones. Al elemento resultante de este proceso se le denomina **imagen**.

Posteriormente, para proceder a ejecutar el contenido de esta imagen (que deberá tener un formato específico), ésta se coloca en un contenedor. Será posible, por tanto, referirse a la imagen como a la receta que especificará cómo es la instancia en ejecución, a la que se denominará **contenedor**.

Al proceso completo de empaquetar un software en una imagen y ejecutarlo en un contenedor se le denomina en inglés *containerization*. En castellano se ha popularizado el anglicismo *contenerización* para referirnos a dicho proceso.

Al igual que los contenedores permiten ser transportados en tren, camión o barco independientemente de la carga, los contenedores de software actúan como una unidad estándar de software que puede contener distintas aplicaciones y dependencias.

Los contenedores permiten aislar cada aplicación de las demás que se estén ejecutando en el sistema operativo que comparten, ya sea Linux o Windows. Cuando se habla de aislamiento a nivel de aplicaciones, uno de los conceptos que puede venir a la mente son las máquinas virtuales. Por ello, en la siguiente sección compararemos contenedores con máquinas virtuales y se verán los beneficios de una forma clara.

### 5.1 Máquinas virtuales versus contenedores

Tanto los contenedores como las máquinas virtuales proporcionan aislamiento de los recursos desplegados, sin embargo, funcionan de manera distinta. En la siguiente figura podemos ver una comparación entre ambos:

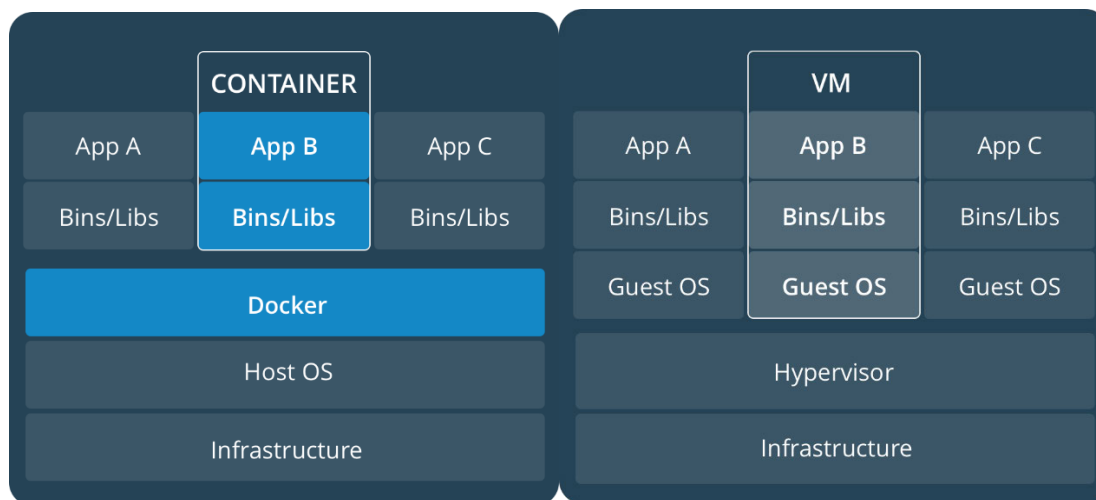


Figura 5-1. Host de contenedores versus máquinas virtuales. Fuente: Docker Docs

Lo primero que se observa es que los contenedores se ejecutan directamente en el sistema operativo, al contrario que las máquinas virtuales, que requieren un sistema operativo completo. Un beneficio directo de esto es que los contenedores se ejecutan aislados unos de otros pero el tamaño que ocupan es mucho menor. De hecho, como buena práctica, se intenta que cada imagen ocupe lo mínimo posible.

Lo siguiente que se ve es una capa denominada *Docker*. Docker es un ejemplo de motor o host (anfitrión en castellano) de contenedores. Esta es una pieza fundamental que se encargará de crear los contenedores desde las imágenes y ponerlos en ejecución.

Otro beneficio de la contenerización es la escalabilidad, beneficio que se nutre de lo ligeros que son los contenedores. Es posible escalar horizontalmente (crear nuevos contenedores de la misma imagen) para mejorar la disponibilidad de nuestra aplicación. Si la aplicación se ejecuta en un entorno en el que se tienen disponibles varios hosts (distintas máquinas físicas o virtuales), sería recomendable instanciar cada una de las imágenes que componen la aplicación en distintos hosts para asegurar que, si hay algún problema en alguno de ellos, la aplicación siga estando disponible.

Para que los hosts de contenedores puedan funcionar se requiere que el sistema operativo ofrezca los mecanismos necesarios para garantizar el aislamiento. A eso se debe la siguiente sección.

## 5.2 Fundamentos en los que se basan los contenedores

El sistema operativo provee de aislamiento del sistema de ficheros, de la red, registro y por ejemplo, la lista de los procesos que están en ejecución. En el kernel de Linux se posibilita este aislamiento mediante dos conceptos que fueron añadidos durante el año 2002 en la versión 2.4.19: *namespaces* y *cgroups* (Grupos de Control).

Los namespaces proveen a los procesos con una visión única del sistema aislándolos del resto. Hay distintos tipos como por ejemplo los siguientes:

- **PID:** En este namespace el proceso se convierte en el PID 1 y los hijos de éste serían el resto de los procesos. Los demás procesos del sistema “desaparecen” y ya no estarían visibles dentro de este namespace.
- **Networking:** Es el namespace que permite aislar nuestro proceso a nivel de red y permite, por ejemplo, que nuestro proceso utilice un puerto independientemente de si se está utilizando ya en el sistema.
- **Mount:** Es el namespace referente al sistema de ficheros. Permite montar y desmontar sistemas de ficheros sin afectar al sistema anfitrión.

Los cgroups permiten limitar el uso de CPU o memoria que nuestros procesos podrán usar.

En el kernel de Windows a partir de Windows Server 2016 existen dos posibilidades:

- “Windows Server Containers” proporciona el aislamiento basado en el mismo concepto que en Linux, aislamiento a nivel de procesos y namespaces. Al igual que en Linux, el kernel del host es compartido por todos los contenedores en ejecución.
- “Hyper-V Containers” funciona de la misma manera que Windows Servers Containers pero añadiendo un nivel mayor de aislamiento al ejecutar cada contenedor en una máquina virtual optimizada. En este caso, cada contenedor tiene un kernel único que no es compartido con el resto de los contenedores o con el host.

Aunque no vayamos a profundizar en cómo se gestiona a nivel del sistema operativo, dado que estaría fuera del ámbito de este trabajo podemos ver que no es una tarea trivial. Por tanto, para facilitar esta tarea y permitir que los contenedores se utilicen de manera global, existen distintos hosts de contenedores que podemos usar como CoreOS *RKT* (pronunciado “rocket”), *LXD*, *OpenVZ* o *Docker*. En este trabajo se utilizará *Docker* como motor (*engine* en inglés) o host para ejecutar nuestros contenedores.

## 5.3 Docker

Docker es un proyecto de código libre que permite a desarrolladores y administradores de sistemas desarrollar, desplegar y ejecutar aplicaciones con contenedores de una forma automatizada. Docker además es una compañía que promueve y continúa mejorando esta tecnología siempre teniendo en cuenta a la comunidad y sus usuarios.

Docker ofrece diversas herramientas y software dependiendo de si estamos en un entorno de desarrollo o en entornos de producción. Las versiones que utilizaremos a lo largo de este proyecto están detalladas en la sección Herramientas Usadas.

### 5.3.1 Imágenes de Docker

Las imágenes de Docker pueden ejecutarse nativamente en Linux y en Windows. Sin embargo, hay que tener en cuenta que las imágenes con base Linux pueden ejecutarse en hosts Linux y las imágenes con base Windows en hosts bajo un sistema Windows.

El propósito de una imagen es definir un entorno junto a sus dependencias de una forma explícita, asegurando que sea posible depurar una aplicación durante la fase de desarrollo en un entorno exactamente igual que el entorno de producción en el que se ejecutará.

Uno de los motivos por los que Docker y sus imágenes son tan populares es porque permiten definir imágenes extendiendo otras previamente definidas, esto es, se pueden definir **imágenes base** especialmente preparadas para ser extendidas, como pueden ser imágenes que contienen simplemente un sistema operativo.

Uno de los beneficios principales que se obtienen al usar Docker es eliminar el típico error que tanto ocurre al desarrollar software cuando algo funciona en nuestro equipo, pero no en otro entorno: “*it works on my machine!*”.

### 5.3.2 Repositorio, Registro y Tags

Antes de hablar sobre cómo se definen las imágenes de Docker, se presentan ciertos conceptos que ayudarán a entender el ecosistema de Docker y la forma en la que se gestionan dichas imágenes.

**Tag:** es una marca o etiqueta que se aplica a una imagen de Docker. Es la forma que tenemos para distinguir diferentes imágenes dispuestas en el mismo repositorio.

**Repositorio (repo):** un repositorio contiene un conjunto de imágenes de Docker que están relacionadas entre sí, marcadas con un tag que nos suele indicar la versión de la imagen. El tag se puede utilizar también para describir distintas variantes de una imagen, como por ejemplo imágenes que contienen SDKs (kits de desarrollo) o imágenes que, simplemente, contienen el runtime. Un repositorio puede contener imágenes para distintas plataformas, es decir, imágenes para Linux e imágenes para Windows.

**Registro:** un registro es un servicio que contiene y provee acceso a los repositorios. Docker ofrece uno llamado Docker Hub [12] que es el registro por defecto en el que se encuentran la mayoría de las imágenes públicas. Microsoft Azure [5] tiene disponible *Azure Container Registry* [13] para subir imágenes privadas.

Cabe destacar que en *Docker Hub* [12] hay disponibles, de forma pública, distintas imágenes base que han sido subidas a la plataforma por las compañías, organizaciones, equipos y desarrolladores independientes de cada una. Por ejemplo, podemos encontrar imágenes base con distintas versiones de Linux, o imágenes con distintas versiones de Java y/o Tomcat, NodeJS... Hablaremos de cómo hacer uso de estas imágenes base en la siguiente sección.

### 5.3.3 Dockerfile y el formato de imágenes

Para definir cómo van a ser las imágenes en Docker se usa un fichero de texto llamado "*Dockerfile*", sin extensión, en el que se definen los comandos, uno por línea.

Para saber cómo definir imágenes empezaremos por el ejemplo más sencillo, la imagen llamada hello-world, que está disponible en el registro público por excelencia: Docker Hub [12]: [https://hub.docker.com/\\_/hello-world/](https://hub.docker.com/_/hello-world/). Desde aquí se puede navegar hasta GitHub, donde podremos ver el fichero *Dockerfile* de esta imagen:

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

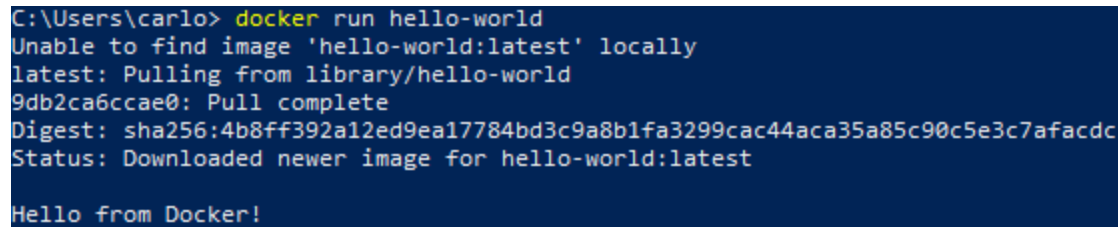
En este primer ejemplo tenemos simplemente 3 comandos:

- **FROM scratch:** referencia la imagen que se tomará como base. En este caso la imagen base "scratch" es una imagen vacía. Otras imágenes están basadas en distribuciones comunes como Ubuntu, Debian...
- **COPY hello /:** Este comando copia el binario "*hello*" al directorio raíz de la imagen.

- **CMD ["hello"]:** el último comando define el punto de entrada que se ejecutará cuando un contenedor es ejecutado basado en esta imagen. En este caso será el binario *"hello"* que copiamos en el paso anterior.

Dado que esta imagen está disponible en Docker Hub[12], es posible ejecutar un contenedor que use esta imagen abriendo una consola de comandos (PowerShell, CMD, Linux Bash) y ejecutando lo siguiente:

**docker run hello-world**



```
C:\Users\carlo> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

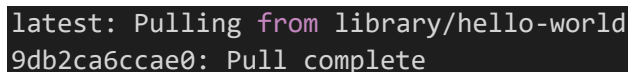
*Figura 5-2. Docker run para ejecutar imagen*

El CLI de Docker intentará ejecutar la imagen *hello-world*. Como vemos, dado que no ha sido capaz de encontrar la imagen localmente, el CLI de Docker se ha descargado la imagen de Docker Hub. Una vez descargada ha creado un contenedor basado en dicha imagen y lo ha ejecutado mostrándonos el texto *"Hello from Docker!"*.

Si nos fijamos bien en el resultado de la ejecución del comando anterior podemos ver lo siguiente:

Cuando nos referimos a una imagen, los parámetros necesarios son el repositorio y el tag que la identifica (el registro sería también necesario, pero aquí se está usando el dado por defecto: Docker Hub). El formato sería el siguiente: **registro/repositorio:tag**. Si no se especifica tag, el valor por defecto que se usará para intentar encontrar la imagen es **"latest"** que en castellano sería *"última versión"*.

Si ahora nos fijamos en el proceso de descarga de la imagen veremos un par de detalles:



```
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
```

El primero, como ya hemos comentado, indica que se está descargando el tag *"latest"* del repositorio *hello-world*. El segundo es un valor alfanumérico seguido de la confirmación de la descarga. Este valor indica la forma en la que funcionan las imágenes de Docker: las capas (*layers* en inglés) o imágenes intermedias.

Una capa es, en esencia, una serie de ficheros en disco que contienen el resultado de la ejecución del comando en cuestión. Las capas se generan cuando los comandos definidos en el *Dockerfile* se ejecutan, durante el proceso de compilación de la imagen. En nuestro primer ejemplo, vemos que solamente se descarga una capa, que es el resultado de la ejecución del segundo comando: **COPY**

**hello /**. Esto es debido a que el primer comando está utilizando como base una imagen vacía (*scratch*), designada específicamente por Docker para declarar que el siguiente comando será la primera capa y, por consiguiente, no aporta ninguna capa adicional. Esto es muy útil cuando se crean imágenes base.

Las capas son sólo de lectura y se almacenan sólo una vez en disco, lo que unido a que las imágenes de Docker normalmente derivan de múltiples imágenes base, permite reutilizar imágenes y/o capas ya descargadas, reduciendo significativamente los tiempos de compilación y despliegue de nuevas versiones de nuestras imágenes. Basado en este principio, se recomienda poner al final del fichero las capas que implican directamente a nuestra aplicación y no a sus dependencias.

Podemos concluir por tanto que un contenedor es una imagen con una capa de lectura/escritura encima de varias capas de "sólo lectura".

Vamos ahora a crear una aplicación sencilla de *ASP.NET Core* utilizando tanto Visual Studio Code y los CLI de Docker y .NET Core (que como ya hemos comentado son multiplataforma) como *Visual Studio 2017* añadiéndole soporte para Docker. Utilizaremos este ejemplo para definir un *Dockerfile* más completo y explicaremos cómo compilar y ejecutar un contenedor tanto desde el propio Visual Studio como desde la línea de comandos.

## 5.4 Creando un Proyecto usando ASP.NET Core con soporte para Docker

ASP.NET MVC Core junto a ASP.NET Core y .NET Core es la solución multiplataforma y de código abierto de Microsoft que permite implementar nuestros servicios y aplicaciones web de una forma rápida, centrándonos en nuestro código. Estos son los frameworks que se han usado en este proyecto. Para más información sobre ellos tenemos disponible el capítulo Herramientas Utilizadas.

Hay dos maneras de crear un proyecto y añadirle soporte para Docker. Una es utilizar la interfaz gráfica de Visual Studio 2017 y otra es utilizar la línea de comandos y el editor de código Visual Studio Code.

Vamos a empezar utilizando el CLI y generaremos un *Dockerfile* manualmente que permitirá ejecutar nuestro servicio en un contenedor. Posteriormente lo generaremos directamente con Visual Studio 2017 y veremos un *Dockerfile* muy interesante utilizando funciones avanzadas.

### 5.4.1 .NET Core CLI y Dockerfile manual

Una vez instalado el SDK de .NET Core podemos utilizar el ejecutable "dotnet" desde la línea de comandos y crear un nuevo proyecto usando dotnet new[14] a través del CLI.

El objetivo es crear un API utilizando la plantilla de webapi. Para ello simplemente nos vamos a un directorio vacío y ejecutamos ahí el comando:

```
dotnet new webapi
```

```
D:\TFG\TFG.SimpleAPI> dotnet new webapi
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on D:\TFG\TFG.SimpleAPI\TFG.SimpleAPI.csproj...
  Restoring packages for D:\TFG\TFG.SimpleAPI\TFG.SimpleAPI.csproj...
  Generating MSBuild file D:\TFG\TFG.SimpleAPI\obj\TFG.SimpleAPI.csproj.nuget.g.props.
  Generating MSBuild file D:\TFG\TFG.SimpleAPI\obj\TFG.SimpleAPI.csproj.nuget.g.targets.
  Restore completed in 991.98 ms for D:\TFG\TFG.SimpleAPI\TFG.SimpleAPI.csproj.

Restore succeeded.
```

Figura 5-3. Creación nuevo proyecto con el CLI de .NET Core

Este comando creará el proyecto y ejecutará `dotnet restore`[15], lo que restaurará las dependencias.

Acto seguido ejecutaremos `dotnet publish`[16] que compilará el proyecto y generará los binarios listos para ser ejecutados:

`dotnet publish -c Release`

```
D:\TFG\TFG.SimpleAPI> dotnet publish -c Release
Microsoft (R) Build Engine version 15.8.166+gd4e8d81a88 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 44.7 ms for D:\TFG\TFG.SimpleAPI\TFG.SimpleAPI.csproj.
TFG.SimpleAPI -> D:\TFG\TFG.SimpleAPI\bin\Release\netcoreapp2.1\TFG.SimpleAPI.dll
TFG.SimpleAPI -> D:\TFG\TFG.SimpleAPI\bin\Release\netcoreapp2.1\publish\
```

Figura 5-4. Publicación de proyecto con el CLI de .NET Core

Una vez que el proyecto es publicado, se añade el fichero *Dockerfile* a la carpeta del proyecto para meter nuestra aplicación en un contenedor con el siguiente contenido:

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime
WORKDIR /app
EXPOSE 80
COPY bin/Release/netcoreapp2.1/publish .
ENTRYPOINT [ "dotnet", "TFG.SimpleAPI.dll" ]
```

En este *Dockerfile* tenemos los siguientes comandos:

- **FROM:** Partimos de la imagen de dotnet que contiene el runtime de la versión 2.1 de aspnetcore el cual será necesario para ejecutar nuestra aplicación web. Esta imagen sólo contiene el runtime y no el SDK.
- **WORKDIR /app:** Este comando especifica que el directorio activo dentro de la imagen es /app.
- **EXPOSE 80:** Este es un comando muy importante. Dado que la aplicación que ejecutaremos dentro de nuestro contenedor es un servicio web, ésta debe ser accesible a través de la red. Con este comando se especifica que nuestro contenedor expondrá el puerto 80. Docker se encarga de la gestión de la red y permite crear redes virtuales en las que desplegar uno o



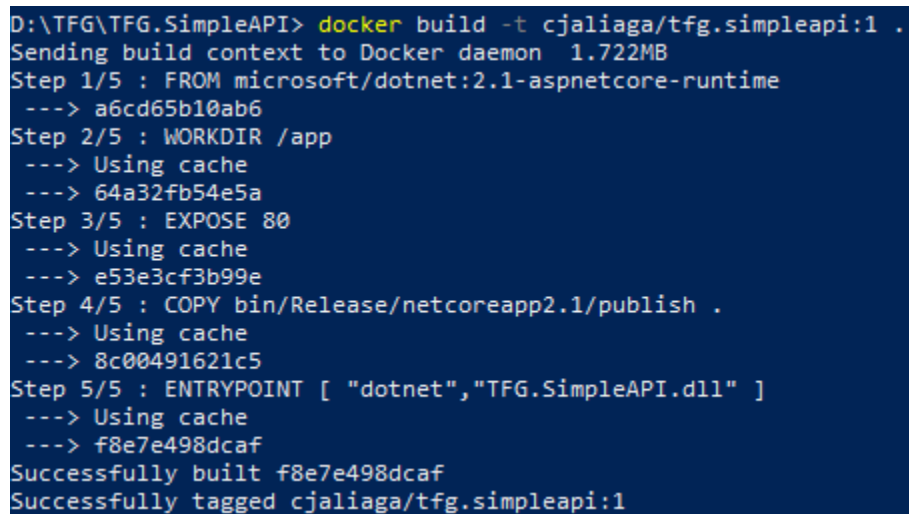
varios contenedores para que, por ejemplo, puedan comunicarse entre sí. Más adelante veremos cómo hacer que nuestra aplicación, desplegada en el puerto 80 en nuestro contenedor, sea accesible desde el exterior.

- **COPY:** Copia el resultado de la publicación de la aplicación dentro de la carpeta `app/` de nuestra imagen.
- **ENTRYPOINT:** Este comando define el punto de entrada de nuestra aplicación. En este caso ejecutará el CLI de .NET Core “dotnet” y como argumento se indica la DLL que contiene nuestra aplicación compilada.

Anteriormente hemos visto cómo ejecutar un contenedor de una imagen que ya estaba compilada y publicada en un registro, pero aún no hemos visto cómo es posible compilar y generar una imagen a partir de nuestro propio *Dockerfile*.

Para compilar la imagen se hará uso del comando del CLI de Docker llamado `docker build` [17], navegando con una consola de comandos hasta el directorio del proyecto donde está localizado el fichero *Dockerfile*:

```
docker build -t cjaliaga/tfg.simpleapi:1 .
```



```
D:\TFG\TFG.SimpleAPI> docker build -t cjaliaga/tfg.simpleapi:1 .
Sending build context to Docker daemon 1.722MB
Step 1/5 : FROM microsoft/dotnet:2.1-aspnetcore-runtime
--> a6cd65b10ab6
Step 2/5 : WORKDIR /app
--> Using cache
--> 64a32fb54e5a
Step 3/5 : EXPOSE 80
--> Using cache
--> e53e3cf3b99e
Step 4/5 : COPY bin/Release/netcoreapp2.1/publish .
--> Using cache
--> 8c00491621c5
Step 5/5 : ENTRYPOINT [ "dotnet", "Tfg.SimpleAPI.dll" ]
--> Using cache
--> f8e7e498dcdf
Successfully built f8e7e498dcdf
Successfully tagged cjaliaga/tfg.simpleapi:1
```

Figura 5-5. Compilación imagen de Docker usando el CLI

Para la ejecución del comando hemos utilizado el parámetro `-t` que indica el repositorio y el tag de esta imagen:

- **Repositorio:** `cjaliaga/tfg.simpleapi`
- **Tag:** `1`

El punto al final del comando es muy importante pues indica el directorio en el que se ejecutará el proceso de compilación. Por ejemplo, cómo podemos ver en el paso 4, al ejecutarse el comando **COPY**

el directorio especificado en el *Dockerfile* es relativo al directorio donde ejecutamos el proceso de compilación y que hemos especificado como parámetro en el comando.

Una vez generada la imagen se procederá a ejecutar un contenedor basado en ella, usando el comando `docker run`[18] como ya se ha hecho anteriormente:

```
docker run -p 5000:80 cjaliaga/tfg.simpleapi:1
```

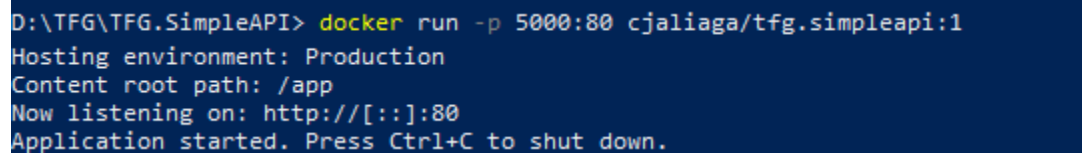


Figura 5-6. Ejecución de la imagen compilada usando el CLI de Docker

En este caso, dado que nuestra aplicación expone el puerto 80 (donde nuestra aplicación web está escuchando) utilizaremos el parámetro `-p` para mapear un puerto de nuestra máquina (el 5000 en este caso) al puerto 80 del contenedor de la siguiente forma: “puerto local:puerto del contenedor”.

En la aplicación desplegada en nuestro contenedor tenemos disponible una URL que mostrará una lista de valores en formato JSON. Para comprobar que todo ha funcionado como esperábamos simplemente tenemos que navegar a <http://localhost:5000/api/values> y veremos la respuesta:

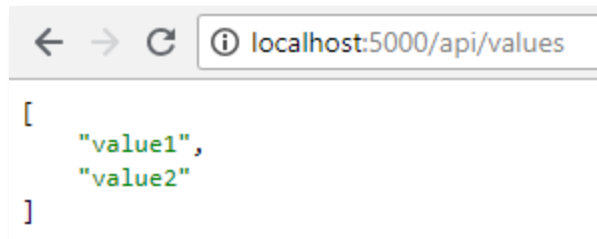


Figura 5-7. Aplicación desplegada en el contenedor en ejecución

#### 5.4.2 Visual Studio 2017 con Dockerfile multi imagen

Cuando creamos el proyecto con Visual Studio 2017 aparece automáticamente un checkbox para habilitar el soporte de Docker en nuestro proyecto e incluso seleccionar el sistema operativo bajo el que se ejecutará.

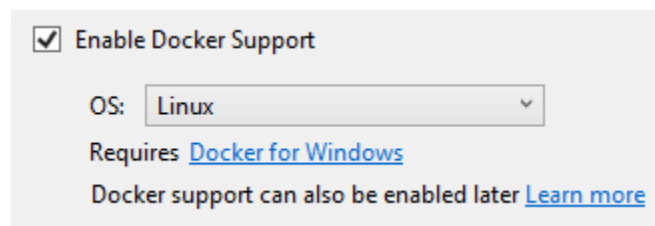


Figura 5-8. Habilitar soporte para Docker en VS2017

Si no lo seleccionamos a la hora de crearlo también tenemos disponible la opción más adelante simplemente haciendo clic derecho sobre nuestro proyecto, Add y posteriormente “Docker Support”

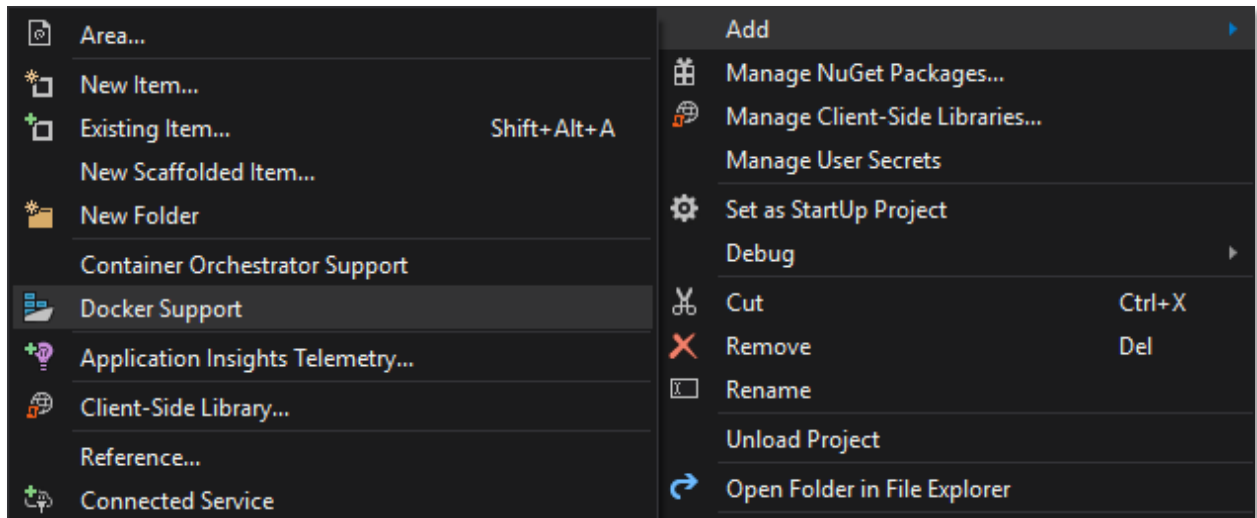


Figura 5-9. Añadir soporte para Docker en proyecto existente usando VS2017

Si comprobamos el *Dockerfile* que ha sido generado se ve que ahora hay 4 secciones separadas. Recordamos que siempre buscamos que nuestras imágenes sean lo más ligeras posibles, añadiendo únicamente lo necesario para ejecutar nuestras aplicaciones, pero también que los entornos de desarrollo y producción fueran exactamente iguales para evitar resultados inesperados.

Por tanto, sería muy interesante no sólo definir cómo ejecutar nuestra aplicación en nuestro *Dockerfile* sino además definir cómo la compilaremos y publicaremos.

Para poder cumplir estas dos premisas se utilizan imágenes intermedias como ayuda en el proceso de compilación y, posteriormente, se copia el resultado en la imagen final en la que únicamente tendremos las dependencias necesarias para ejecutar nuestra aplicación.

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /src
COPY ["TFG.SimpleAPI/TFG.SimpleAPI.csproj", "TFG.SimpleAPI/"]
RUN dotnet restore "TFG.SimpleAPI/TFG.SimpleAPI.csproj"
COPY . .
WORKDIR "/src/TFG.SimpleAPI"
RUN dotnet build "TFG.SimpleAPI.csproj" -c Release -o /app
```

```
FROM build AS publish
RUN dotnet publish "TFG.SimpleAPI.csproj" -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "TFG.SimpleAPI.dll"]
```

La mayoría de los comandos (tanto del CLI de .NET Core como de Docker) ya han sido explicados a lo largo del capítulo. Nos centraremos en las diferencias para este caso específico:

- **FROM:** Las dos primeras veces que aparece este comando son para definir las imágenes externas de las que se parte, la del runtime para ejecución y la del SDK para la compilación. La diferencia en estos casos es que se utiliza el comando **AS** para asignarle un alias a las imágenes y poder volver a referirnos a ellas a lo largo del proceso.
- **COPY:** El último COPY tiene un parámetro adicional *--from=publish* que está indicando de qué imagen anterior copiaremos el contenido de la carpeta /app.

Los pasos de este *Dockerfile* son los siguientes:

El runtime de ASP.NET Core como base y se expone el puerto 80 -> SDK de ASP.NET Core como Build, restauramos y compilamos -> Build como publish, publicamos el proyecto -> volvemos a la imagen base como final, copiamos todo el resultado de la compilación y definimos el punto de entrada.

Lo siguiente sería compilar la imagen y ejecutarla en un contenedor. En este caso usamos el parámetro *-f* para indicar la ruta de nuestro *Dockerfile*:

```
docker build -t cjaliaga/tfg.simpleapi:2 -f .\TFG.SimpleAPI\Dockerfile .
docker run -p 5001:80 cjaliaga/tfg.simpleapi:2
```

## 5.5 Comandos

Antes de pasar a hablar de microservicios, vamos a ver una serie de comandos del CLI de Docker que serán muy útiles.

### 5.5.1 Docker ps [19]

Este comando muestra la lista de contenedores que tenemos en la máquina, aunque por defecto sólo muestra los que están en ejecución. Si se quiere mostrar todos ellos simplemente hay que añadir el parámetro *-a* o *--all*. Podemos ver los dos contenedores ejecutando las imágenes generadas en las secciones anteriores junto a, por ejemplo, los puertos que cada uno tiene asignado:

```
D:\TFG\TFG.SimpleAPI> docker ps
IMAGE                COMMAND              PORTS
cjaliaga/tfg.simpleapi:2  "dotnet TFG.SimpleAP..."  0.0.0.0:5001->80/tcp
cjaliaga/tfg.simpleapi:1  "dotnet TFG.SimpleAP..."  0.0.0.0:5000->80/tcp
```

Figura 5-10. Resultado ejecución *docker ps*

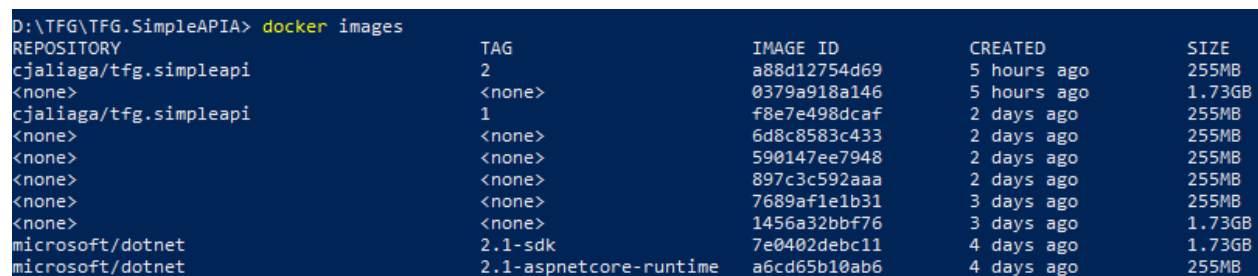
### 5.5.2 Docker inspect [20]

Inspect nos muestra toda la información sobre un contenedor, por defecto, en una salida en formato JSON. El uso sería *"docker inspect [Id del contenedor|nombre del contenedor]"*. Tanto el Id como el nombre lo podemos obtener usando el comando *"docker ps"* o asignándolos a la hora de ejecutarlos.

La salida de este comando es muy extensa e, incluso para un contenedor básico, podría ocupar varias páginas; por lo que no mostraremos ese resultado en esta memoria.

### 5.5.3 Docker images [21]

*Images* nos muestra un listado de las imágenes que tenemos en nuestro repositorio local, ya sea porque las hayamos generado nosotros, usadas para generar nuestras imágenes, directamente descargadas usando *"docker pull"*[22] o ejecutadas con el comando *"docker run"* [18].



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cjaliaga/tfg.simpleapi	2	a88d12754d69	5 hours ago	255MB
<none>	<none>	0379a918a146	5 hours ago	1.73GB
cjaliaga/tfg.simpleapi	1	f8e7e498dcac	2 days ago	255MB
<none>	<none>	6d8c8583c433	2 days ago	255MB
<none>	<none>	590147ee7948	2 days ago	255MB
<none>	<none>	897c3c592aaa	2 days ago	255MB
<none>	<none>	7689af1e1b31	3 days ago	255MB
<none>	<none>	1456a32bbf76	3 days ago	1.73GB
microsoft/dotnet	2.1-sdk	7e0402debc11	4 days ago	1.73GB
microsoft/dotnet	2.1-aspnetcore-runtime	a6cd65b10ab6	4 days ago	255MB

Figura 5-11. Resultado ejecución *docker images*

### 5.5.4 Docker start[23], stop[24] y restart[25]

Con estos tres comandos podremos arrancar, parar y reiniciar los contenedores. Al igual que con *docker inspect*[20] tendremos que recuperar el nombre y/o el id del contenedor usando *docker ps* [19].

### 5.5.5 Docker push[26]

Al igual que podemos bajarnos imágenes de registros externos, como *Docker Hub*[12] o registros privados como *Azure Container Registry* [13], también tenemos la posibilidad de subir nuestras imágenes a repositorios en estos registros. Para subirla simplemente tenemos que usar *docker push* TAG (después de haber iniciado sesión en el registro usando *docker login*[27]).

## 6. SOLUCIÓN BASADA EN MICROSERVICIOS

En el capítulo anterior hemos hablado de los contenedores, en lo que se basan, cómo podemos trabajar con ellos y sus beneficios. Una vez que los hemos definido vamos a pasar a hablar de una arquitectura que funciona muy bien desplegada en contenedores: los microservicios.

Los microservicios son un estilo de arquitectura de software en el que las aplicaciones están compuestas por módulos pequeños e independientes que se comunican entre ellos utilizando APIs bien definidas. Estos servicios son módulos altamente desacoplados y suficientemente pequeños para implementar una funcionalidad única.

El propósito de una arquitectura de microservicios es hacer más sencillo el desarrollo y escalado de aplicaciones. También mejoran la colaboración entre equipos completamente autónomos y permiten traer nuevas funcionalidades al mercado rápidamente.

Las arquitecturas de microservicios ofrecen grandes beneficios, pero también presentan grandes desafíos de los que se hablará a continuación.

Antes de meternos en materia, es muy importante destacar que, aunque los contenedores y sus beneficios encajan a la perfección con los microservicios amplificando su potencial enormemente, no es obligatorio el uso de los contenedores para hablar de microservicios y muchos patrones arquitectónicos de los que hablaremos a lo largo de esta memoria pueden ser aplicados sin ellos. También es importante aclarar que no es necesaria una arquitectura de microservicios para *contenerizar* una aplicación por lo que una aplicación monolítica podría ser desplegada dentro de un contenedor.

En este trabajo sin embargo nos centraremos en la simbiosis de ambos, dados los importantes beneficios que pueden conseguirse, de los cuales hemos hablado en capítulos anteriores.

### 6.1 Arquitecturas de microservicios

Como ya hemos adelantado, las arquitecturas de microservicios son un enfoque para desarrollar aplicaciones de servidor como un grupo de pequeños servicios. Cada servicio se ejecuta en su propio proceso y se comunica con otros procesos usando distintos protocolos, como HTTP/HTTPS, AMQP, WebSockets...

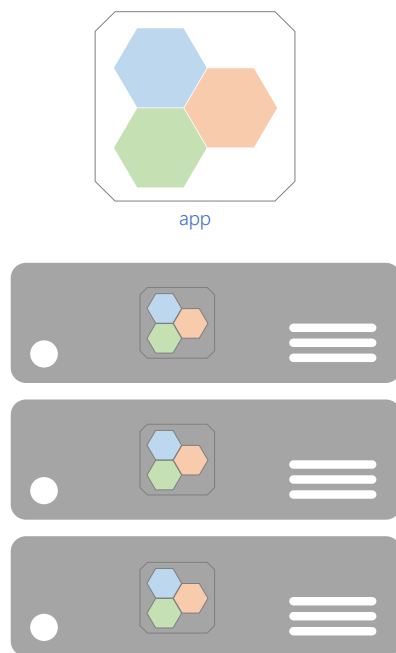
Cada microservicio es independiente de los demás y por tanto implementa una funcionalidad o lógica de negocio completa, de principio a fin. Debe ser desarrollado autónomamente y desplegado independientemente. Además, la lógica del dominio implementada en el microservicio será responsable de sus propios modelos de datos. Esto se hará de una forma descentralizada por lo que cada microservicio puede tener una base de datos distinta y/o podrán estar basados en distintas tecnologías de almacenamiento de datos (SQL, NoSQL) y distintos lenguajes de programación.

Aunque el propio nombre incluya la palabra micro, ello no implica que tengamos que hablar del tamaño de los microservicios como un punto importante. Lo importante debería ser el crear servicios altamente desacoplados ganando la autonomía en el desarrollo, despliegue y escalado de ellos. Por descontado queda que cuando estamos diseñando nuestros microservicios tenemos que tener en mente el hacerlos lo más pequeños posible siempre y cuando por ello no estemos creando demasiadas dependencias externas. Si un microservicio tiene que estar constantemente interconectado con otros para realizar cualquier operación puede ser un indicativo de que quizás deberían estar contenidos en él.

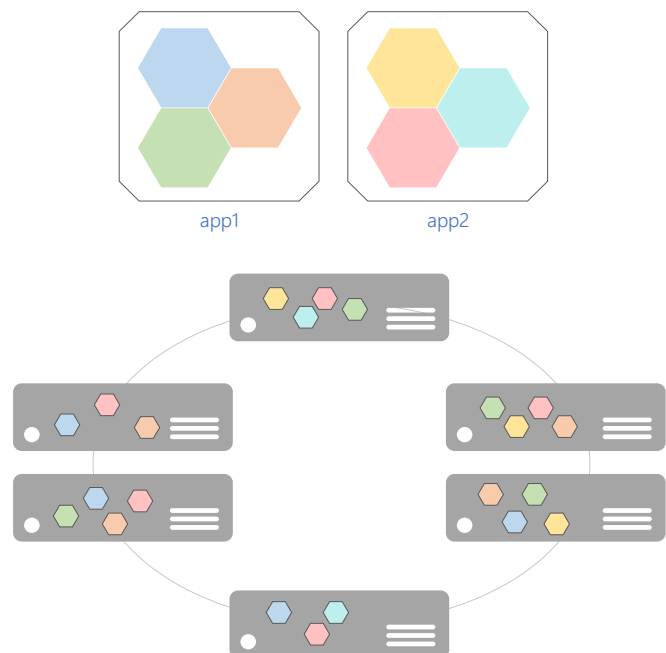
Esta arquitectura provee de mayor agilidad a largo plazo y un mejor mantenimiento en sistemas complejos altamente escalables dado que permite crear aplicaciones basadas en muchos servicios que pueden ser desplegados independientemente, por lo que sus ciclos de vida son autónomos.

Otro beneficio adicional es la capacidad de escalar cada microservicio de manera independiente, en contraposición a lo que ocurre en las aplicaciones monolíticas, sobre las que hablábamos al principio de esta memoria, que tienen que escalar como una unidad. Podemos escalar un microservicio específico bajo demanda otorgándole más capacidad de procesamiento, ancho de banda de red, memoria o desplegando más copias de este microservicio en la misma o distintas máquinas. Para gestionar todo el proceso de escalado de los microservicios, en el mundo de los contenedores en el que se ha orientado este proyecto, existen los **orquestadores** de los que hablaremos más adelante.

### Arquitectura monolítica



### Arquitectura de microservicios



*Figura 6-1. Arquitecturas monolíticas versus arquitecturas de microservicios. Fuente: Elaboración propia.*

Las aplicaciones basadas en arquitecturas de microservicios facilitan escenarios de integración y entrega continuas, elementos que son parte de la cultura DevOps. Aceleran la entrega de nuevas funcionalidades en la aplicación y permiten su evolución de forma autónoma siempre que los contratos que éstas exponen se mantengan claros. Si no los cambiamos podremos modificar la implementación de cualquier microservicio o añadir nuevas funcionalidades sin afectar a otros microservicios.

A continuación se presentan algunos factores a tener en cuenta en el diseño de nuestras soluciones de microservicios.

## 6.2 Definiendo el modelo de datos y el ámbito de cada microservicio

Una regla importante en las arquitecturas de microservicios es que cada microservicio debe ser el dueño de su propia lógica de negocio y de los datos del dominio que éste maneje. Una conclusión que podemos sacar de esto es que el modelo conceptual que manejará cada microservicio podrá ser completamente distinto del resto. Cada microservicio tendrá un *Bounded Context (BC)* que podríamos traducir al castellano como **contexto acotado**.

El concepto de microservicio deriva del patrón de diseño *Bounded Context (BC)* en la metodología *DDD (Domain-driven design)* que podemos traducir al castellano como diseño dirigido por el dominio. DDD se enfrenta a grandes modelos separándolos en distintos contextos acotados (BC) y siendo muy explícito sobre los límites o el ámbito de cada uno de ellos. Cada contexto debería tener su propio modelo y base de datos dado que cada uno maneja sus propios datos.

Uno de los términos más importantes en DDD es **Entidad** o **Entidad de Dominio**. Estas entidades pueden tener distintos nombres en diferentes BC, incluso cuando el identificador sea el mismo, es decir, el identificador único que usaremos para almacenarlo en la base de datos (o cualquier mecanismo que utilicemos para ello). Por ejemplo, en el contexto "Perfil de Usuario" la entidad Usuario podría compartir el identificador con la entidad Alumno en el contexto "Gestión de Asignaturas".

Un microservicio será por tanto como un *Bounded Context* pero también como un servicio distribuido. Se compilará como un proceso independiente (y en nuestro caso probablemente en un contenedor distinto) y deberá usar los protocolos distribuidos que comentamos anteriormente como HTTP/HTTPS, WebSockets o AMQP o incluso un protocolo binario como es TCP.

### 6.2.1 Analizando el dominio para identificar los microservicios

Uno de los desafíos más importantes de los microservicios es definir el ámbito de cada uno de los servicios. La regla general es que cada servicio debería hacer "una cosa", aplicar esta regla sin embargo requiere ser muy cuidadoso. No existe un proceso mecánico que determine el diseño correcto por lo que es necesario analizar en profundidad sobre el dominio de nuestro negocio, requerimientos y objetivos o, de lo contrario, podemos encontrarnos con situaciones indeseables como dependencias ocultas entre servicios o interfaces mal diseñadas.



DDD nos provee de un framework que facilita el diseño adecuado de microservicios. DDD tiene dos fases, una estratégica y otra táctica. En la estratégica se define la estructura del sistema a gran escala. Esta fase nos ayuda a asegurar que nuestra arquitectura siga centrada en las capacidades del negocio. La fase táctica nos provee de una serie de patrones de diseño que podemos usar para crear nuestro modelo de dominio. Estos patrones incluyen entidades, relaciones y servicios del dominio que nos ayudarán a diseñar microservicios altamente desacoplados y cohesivos.



*Figura 6-2. Pasos para identificar microservicios. Fuente: Elaboración propia.*

El diagrama de la figura anterior nos indica los pasos a seguir para identificar correctamente un microservicio: En concreto

1. Primero se analiza el dominio del negocio para entender los requerimientos funcionales de nuestra aplicación. La salida de este paso es una descripción informal del dominio que podrá ser refinada en conjuntos más formales de modelos de dominios.
2. Seguidamente se definen los contextos acotados de nuestro dominio (*bounded context* de los que hablábamos previamente en este capítulo). Cada BC contiene un modelo de dominio que representa un subdominio de la aplicación a gran escala.
3. Dentro del contexto aplicamos los patrones arquitectónicos de DDD para definir las entidades, relaciones y los servicios del dominio.
4. Finalmente usamos los resultados del paso anterior para identificar los microservicios en nuestra aplicación.

### 6.3 Comunicación entre los distintos microservicios

Si venimos del mundo de las aplicaciones monolíticas, donde toda la aplicación se está ejecutando en un único proceso, sabremos que unos componentes invocan a otros usando métodos a nivel del lenguaje o llamadas a funciones. Estas pueden estar fuertemente acopladas si los objetos se están creando directamente en el código (por ejemplo `new Service()`) o pueden ser invocadas en una forma desacoplada si se está usando inyección de dependencias referenciando abstracciones en vez de instancias concretas. Hablaremos de cómo integramos este patrón en el siguiente capítulo.

De cualquier manera en este caso los objetos están en el mismo proceso. Cuando se pasa de una aplicación monolítica a una basada en microservicios, uno de los mayores desafíos es cambiar los mecanismos de comunicación. Si hacemos una conversión directa desde llamadas *in-process* a llamadas RPC (llamada a procedimiento remoto) hacia los servicios, obtendremos comunicaciones nada eficientes que provocarán problemas de rendimiento en entornos distribuidos.

Los microservicios que componen una aplicación de inicio a fin suelen usar comunicaciones REST, en vez de por ejemplo protocolos complejos como SOAP, y comunicaciones flexibles basadas en eventos, en vez de usar orquestadores complejos centralizados, como los que se usan en la arquitectura SOA.

Los protocolos más usados son dos: peticiones y respuestas HTTP cuando estamos ejecutando consultas sobre las API que exponen los microservicios y mensajería asíncrona ligera cuando estamos ejecutando actualizaciones sobre muchos microservicios.

### 6.3.1 Protocolos y tipos de comunicación

El principal factor a tener en cuenta en la organización de las comunicaciones entre microservicios es si estas comunicaciones son síncronas (tenemos que esperar a que se complete la comunicación y por tanto quedarnos esperando al resultado) o asíncronas (lanzamos el mensaje y nos olvidamos).

En el escenario ideal los microservicios solamente deberían comunicarse propagando los datos asíncronamente pero no dependiendo de otros microservicios de forma síncrona como parte de una petición desde el cliente. De esa forma, los hacemos menos resistentes a fallos en caso de que el microservicio del que estemos dependiendo falle. Si necesitamos realizar alguna acción en otro microservicio como parte de una petición, es mejor hacerlo asíncronamente sin que afecte al tiempo de respuesta al microservicio cliente. En estas situaciones es mejor intentar replicar o propagar esos datos (únicamente los necesarios) en las bases de datos de los microservicios iniciales usando patrones como eventos de integración.

Cuando hablamos de comunicaciones síncronas podemos pensar en HTTP/HTTPS y arquitecturas REST que serían los más comunes mientras que en las asíncronas podemos pensar en AMQP (*Advanced Message Queuing Protocol*).

Entre los formatos de los mensajes más usados tenemos JSON o XML o incluso formatos binarios que suelen ser más eficientes. Sin embargo si nuestros servicios estuviesen disponibles públicamente lo más recomendable será usar formatos estándar.

### 6.3.2 Servicios basados en HTTP y REST

REST es un estilo arquitectónico muy popular para comunicaciones basadas en peticiones y respuestas. Cuando un cliente se conecta a un servicio usando este estilo de comunicación envía una petición al servicio, éste la procesa y envía una respuesta al cliente. Normalmente se asume que las peticiones serán procesadas en un corto periodo de tiempo, inferior al segundo o a lo sumo, unos pocos segundos.

REST está basado y altamente ligado al protocolo HTTP, usando sus verbos, como son GET, POST, PUT, DELETE, PATCH... Es el estilo arquitectónico de comunicación más usado con diferencia cuando desarrollamos servicios. En nuestro caso se ha usado para desarrollar Web API con ASP.NET Core.

Existe una iniciativa llamada *OpenAPI*[28] bajo la *Linux Foundation*[29] que nos provee de una estandarización en el formato usado para describir nuestras APIs. Este formato es una evolución de la

especificación de *Swagger*[30] que fue donada a la fundación. Facilita herramientas para crear clientes de forma automática que serán capaces de descubrir y consumir nuestros servicios.

### 6.3.3 Comunicación asíncrona basada en mensajes

La mensajería asíncrona y la comunicación basada en eventos son críticas cuando queremos propagar cambios en múltiples microservicios y sus modelos de dominio. Como comentamos anteriormente, la entidad Usuario podría significar distintas cosas en microservicios diferentes, por lo que cuando existan cambios, necesitamos alguna forma de sincronizarlos a lo largo de las distintas entidades. Una solución puede basarse en la comunicación asíncrona basada en mensajes.

En estas comunicaciones el cliente ejecuta un comando o una petición a otro servicio enviando un mensaje. El mensaje está compuesto por unas cabeceras y un cuerpo. Si el servicio necesita contestar mandará un mensaje distinto al cliente. Dado que es basada en mensajes, el cliente no esperará tener una respuesta inmediata o incluso podría no esperar recibir ninguna.

La infraestructura común es un bróker de mensajes ligero, el cual actúa simplemente distribuyendo los mensajes y, siendo los extremos, los microservicios, los encargados de toda la lógica de producir y consumir mensajes.

#### 6.3.3.1 Mensajes con un solo receptor

En estos casos hay una comunicación punto a punto que envía un mensaje a un único consumidor que está leyendo mensajes en ese canal y que será procesado una única vez.

En caso de que se produzca un fallo ciertos sistemas podrían reintentar el reenvío hasta que el mensaje sea procesado siendo el servidor el encargado de que sea procesado una única vez.

#### 6.3.3.2 Mensajes con varios receptores

También podemos usar el mecanismo de publicador/suscriptor de tal forma que la comunicación desde el publicador esté disponible para distintos microservicios o incluso aplicaciones externas. Nos ayudará además para seguir el principio abierto/cerrado de *SOLID*[31] en el servicio publicador. De esta forma podremos tener en el futuro suscriptores adicionales sin necesidad de modificar el servicio publicador.

### 6.3.4 Registro de servicios

Cada microservicio dentro de nuestra aplicación debe tener un nombre único (URL) que usaremos para descubrir cómo acceder a él. Podemos ver esto como una analogía a la resolución de DNS, que a través de un nombre (o una dirección), identifica a una dirección IP y por tanto a un servidor de destino. Este proceso se denomina resolución de nombres. Con los microservicios ocurre lo mismo, es necesaria una forma de acceder a ellos, usando un nombre, sin preocuparnos de la infraestructura en la que los estamos ejecutando.

Existe un patrón denominado Registro de servicios (*service registry* en inglés) que podemos ver como una base de datos que contiene las localizaciones de red de las instancias de nuestros servicios. Un

factor muy importante es que el registro debe actualizarse cada vez que se elimina una instancia o se añaden nuevas de nuestros servicios.

Cuando ejecutamos nuestros servicios en los orquestadores como *Kubernetes*, *DC/OS*, *Service Fabric*... veremos que el registro de servicios es una de las funciones que implementan por nosotros. Hablaremos de ellos en próximos capítulos.

### 6.3.5 Exponiendo los microservicios directamente o utilizando una puerta de enlace

En la arquitectura de microservicios cada microservicio expone una serie de *endpoints*. Los clientes que se conectan a nuestros servicios (ya sean aplicaciones móviles o aplicaciones web ejecutándose en el navegador de nuestros usuarios) pueden acceder directamente a ellos, por lo que estarían expuestos directamente a internet y tendrían que tener una dirección pública. En entornos de producción y en la nube la dirección pública que exponen podría ser un balanceador que distribuya la carga entre los microservicios.

La comunicación directa entre clientes y microservicios está bien cuando hablamos de una aplicación pequeña, sin embargo, cuando la aplicación es compleja este enfoque presenta una serie de problemas.

Si necesitamos realizar múltiples llamadas a distintos microservicios para cargar una página o pantalla en una aplicación móvil incrementaremos el número de viajes a lo largo de internet aumentando la latencia y la complejidad en la parte del cliente.

Usar protocolos como *AMQP* para comunicaciones orientadas a mensajes no están normalmente soportadas en las aplicaciones cliente por lo que tendríamos que hacer todo a través de peticiones HTTP/HTTPS.

#### 6.3.5.1 Utilización de un API Gateway

Para solucionar estos problemas podemos utilizar el patrón API Gateway. Este servicio nos expone un único punto de entrada para un grupo de microservicios. Este servicio podría implementarse como una Web API ejecutándose en un contenedor o podríamos usar otros servicios que ofrezcan los distintos proveedores Cloud donde estemos desplegando nuestros microservicios. En *Microsoft Azure*[5] tenemos disponible *API Management* [32].

No es una buena idea utilizar un único API Gateway que agregue todos los microservicios de nuestra aplicación dado que de esta forma estaríamos teniendo un único punto de fallo y estaríamos actuando como una aplicación monolítica

#### 6.3.5.2 Interfaces compuestas e interfaces monolíticas

De la misma forma y basándonos en la idea anterior hablaremos de interfaces monolíticas e interfaces compuestas. La interfaz monolítica sería la que se encargaría de conectarse directamente a todos los microservicios para generar la interfaz.

La interfaz compuesta, sin embargo, estaría formada por varios microservicios que generan la interfaz e, internamente, al resto de microservicios para obtener los datos necesarios, agregando por ejemplo unidades de negocio (similar al API Gateway).

## 6.4 Ventajas de una solución basada en microservicios

Lo anteriormente expuesto muestra que las soluciones basadas en microservicios tienen muchos beneficios: Resumidamente

**Cada microservicio es relativamente pequeño, sencillo de manejar y evolucionar:**

- Es fácil para el desarrollador entender la solución y empezar a trabajar, aumentando la productividad.
- Los contenedores arrancan rápido haciendo a los desarrolladores más productivos.
- Los IDE pueden cargar los proyectos mucho más rápido.
- Cada microservicio puede ser diseñado, desarrollado y desplegado independientemente del resto. Esto proporciona agilidad al ser capaces de desplegar nuevas versiones de los microservicios frecuentemente.

**Es posible escalar individualmente áreas de la aplicación.** el servicio de gestión de solicitudes puede ser escalado sin afectar al servicio de gestión de proyectos. Esto hace a la arquitectura de microservicios mucho más eficiente respecto a los recursos utilizados a la hora de escalar en comparación con las soluciones monolíticas tradicionales.

**Los problemas están más aislados.** Si hay un problema en uno de los servicios, el resto pueden continuar sirviendo peticiones (siempre que la solución esté bien diseñada y no haya dependencias directas entre microservicios) . Por el contrario, un componente que no esté funcionando de la forma esperada en una arquitectura monolítica puede tumbar un sistema de forma completa. El poder desplegar de forma sencilla nuevas versiones de cada microservicio permite solucionar el problema sin impactar al resto.

**Es posible hacer uso de las últimas tecnologías.** Dado que se pueden desarrollar y ejecutar los servicios de forma independiente, es posible incorporar nuevas tecnologías sin importar la forma en la que estén desarrollados los microservicios existentes.

**Se puede dividir el trabajo de desarrollo entre distintos equipos.** Cada microservicio puede ser desarrollado por un equipo de desarrollo distinto. Cada equipo puede desarrollar, desplegar, escalar y administrar su microservicio de forma independiente al resto de equipos.

## 6.5 Desventajas de una solución basada en microservicios

A pesar de las ventajas de las que se han hablado, también hay factores que se deben tener en cuenta cuando se desarrollan arquitecturas de este tipo:

**La aplicación pasa a ser distribuida.** La construcción de microservicios con esta arquitectura añade complejidad en el diseño pues los desarrolladores deben implementar comunicaciones externas al proceso, usando protocolos como pueden ser HTTP o AMPQ, que aumentan la latencia, la complejidad de las pruebas y la gestión de excepciones.

**Mayor complejidad a la hora de desplegar.** Una aplicación compuesta por docenas de microservicios que necesite alta escalabilidad (necesite ser capaz de crear muchas instancias de cada microservicio y balancearlas en distintos hosts) requiere un alto grado de complejidad a la hora de ser desplegada y gestionada. Si no se utiliza una infraestructura orientada a microservicios (como pueden ser los orquestadores de contenedores) se puede incurrir en unas labores de desarrollo incluso mayores que el desarrollo de la solución en sí misma.

**Transacciones atómicas.** Normalmente las transacciones atómicas entre microservicios no son posibles. Por ello será necesario mantener la consistencia entre múltiples microservicios a través de eventos.

**Incremento global de los recursos necesarios.** En muchos casos, cuando se reemplaza una aplicación monolítica por una con un enfoque basado en microservicios, la cantidad inicial de recursos globales necesarios por la nueva aplicación será mayor que la infraestructura necesaria para la monolítica. El beneficio se obtiene a largo plazo, con la posibilidad de escalar ciertas partes de la aplicación, pues los recursos empleados para escalar una aplicación monolítica son mucho mayores.

## 7. DISEÑO DE LA SOLUCIÓN

El diseño de la solución está basado en la metodología ágil que se ha empleado en este proyecto, empezando por patrones sobre los que iremos construyendo la solución, pero que serán también muy útiles individualmente.

### 7.1 Inversión de control (IoC) e inyección de dependencias (DI)

Hay ciertos patrones de desarrollo que son muy adecuados por sus múltiples beneficios. Uno de ellos es la inyección de dependencias que está fuertemente unida y basada en la inversión de control.

En el paradigma de la programación orientada a objetos [33] podemos referirnos a la inyección de dependencias como un patrón de diseño en el cual inyectaremos o proveeremos de objetos a una clase generalmente en su constructor a la hora de ser instanciada. También podemos inyectar estas dependencias por ejemplo en *propiedades*[34] en lenguajes como C# (que son básicamente lo que llamamos *getters* y *setters*[35]) aunque, generalmente, se usará el constructor de la clase.

Para que la inyección de dependencias sea efectiva tendremos que aplicar un patrón llamado inversión de control. Este patrón cambia la forma en la que diseñamos nuestras clases, de manera que el resto de las clases de las que ésta dependa serán parámetros definidos en el constructor. De esta forma se delega la responsabilidad de construir una clase y sus dependencias en un tercero.

Seguidamente ilustramos con un ejemplo cómo podemos aplicar estos patrones. Comenzamos con una clase *Coche* que no está haciendo uso de estos patrones:

```
public class Coche
{
    private Motor motor;

    public Coche()
    {
        this.motor = new Motor();
    }

    public bool Arrancar()
    {
        return motor.Arrancar();
    }
}
```

Esta clase *Coche* depende de otra clase *Motor* y tenemos que crear la instancia y usarla. Cambiando la clase *Coche* de la siguiente forma haremos que reciba una instancia de dicha dependencia ya creada:

```

public class Coche
{
    private Motor motor;

    public Coche(Motor motor)
    {
        this.motor = motor;
    }

    public bool Arrancar()
    {
        return motor.Arrancar();
    }
}

```

La responsabilidad ya no recae en la propia clase y se delega en el encargado de crear el coche. Para ello podemos usar el *patrón factoría* [36]

```

public class FactoriaVehiculos
{
    public Coche CrearCoche()
    {
        Motor motor = new Motor();
        Coche coche = new Coche(motor);
        return coche;
    }
}

```

Podríamos verlo como que la clase *Motor* está prestando un servicio a la clase *Coche*. Un beneficio directo de aplicar estos patrones sería que, dado que estamos delegando la creación, podríamos inyectar tanto una clase *Motor* como cualquier clase que derive de ésta como, por ejemplo, un *MotorElectrico*.

Y es, basándonos en esto último, como llegamos al tercer punto importante y del que sacaremos aún más beneficios. Si miramos los principios *SOLID*[31], el que corresponde a la D llamado "*Principio de inversión de dependencia*"[37] nos da la clave. Este principio indica que en vez de depender de tipos concretos es mejor depender de abstracciones o interfaces.

De esta forma cuando se define la interacción entre dos módulos lo hacemos estableciendo el contrato que esta interacción debe cumplir pero no la implementación a bajo nivel. Si seguimos con el ejemplo anterior podemos definir la interfaz *IMotor* que nos expondrá el contrato que deben cumplir los motores:



```
public interface IMotor
{
    bool Arrancar();
}
```

Nuestro *Coche* por tanto dependerá de esta abstracción y será a la hora de instanciarlo cuando se decidirá la implementación, siempre y cuando cumpla con dicho contrato:

```
public class Coche
{
    private IMotor motor;

    public Coche(IMotor motor)
    {
        this.motor = motor;
    }

    public bool Arrancar()
    {
        return motor.Arrancar();
    }
}
```

Con esto llegamos a uno de los mayores beneficios del uso de estos patrones y es el hacer nuestro código *testable*. Dado que nuestras clases no son responsables de crear las dependencias las pruebas unitarias serán muy sencillas, al ser capaces de inyectar implementaciones vacías que no incluirán lógica de negocio adicional. Hablaremos de ello con más detalle en un capítulo próximo.

### 7.1.1 Contenedores de dependencias

Para que sea sencillo gestionar la inyección de dependencias, ciclo de vida de nuestras instancias y la relación y dependencias entre ellas tenemos a nuestra disposición frameworks de inyección de dependencias. En el ecosistema de .NET tenemos varias soluciones cómo *Autofac* [38] o *Unity* [39].

Cómo ya se ha comentado, el framework sobre el que hemos construido nuestros servicios es *ASP.NET Core*. La gente del equipo de *ASP.NET* conoce la importancia de estos patrones y han diseñado el producto con la inyección de dependencias en mente, por lo que no hace falta ningún framework adicional para gestionarlo. Si el soporte que ofrece por defecto no cumple nuestros requisitos el utilizar alguno de terceros es trivial.

Antes de explicar cómo se configuran estas dependencias en nuestras aplicaciones *ASP.NET Core*, es importante conocer un factor que, si no lo tenemos en cuenta a la hora de diseñar nuestras

dependencias, puede causar problemas o situaciones inesperadas en tiempo de ejecución. Este factor lo llamamos en inglés *"lifetime"* que podemos traducir como tiempo de vida o duración de nuestras dependencias. Podemos referirnos a las instancias o dependencias como Servicios.

Cuando registramos nuestras dependencias podemos elegir el tiempo de vida que tendrán o cómo serán resueltas. Podemos elegir entre los siguientes:

- **Transient:** Cada vez que se requiere una instancia de esta clase se crea una nueva. Este *lifetime* es el más indicado tanto para servicios ligeros como para los que no requieren mantener el estado (*stateless*).
- **Scoped:** Con *Scoped* nos referimos a ámbito o contexto. Los servicios registrados como *scoped* serán resueltos una única vez por ámbito o contexto. En aplicaciones web como las que estamos desarrollando aquí, usando *ASP.NET Core*, se creará un contexto distinto con cada petición que procesemos en el servidor; por lo que, dentro de esa petición, si resolvemos una misma dependencia varias veces, estaremos usando siempre la misma instancia. También es posible crear contextos manualmente y resolver servicios dentro de ese contexto si lo queremos.
- **Singleton:** Es un patrón en sí mismo, en el que una instancia es creada una única vez y es usada durante todo el ciclo de vida de la aplicación. Dentro del ámbito del contenedor de dependencias se creará la instancia la primera vez que se requiera y, a partir de ahí, se usará esta instancia en lugar de resolver o crear nuevas. En caso de que nuestra aplicación necesite un servicio *Singleton* es recomendable dejar que el contenedor de dependencias lo inyecte por nosotros en lugar de implementar el patrón directamente en nuestra clase.

Debemos tener especial cuidado cuando mezclamos *Singleton* con *Scoped* para evitar situaciones inesperadas. Imaginemos que registramos el servicio **A** como *Singleton* y el servicio **B** como *Scoped*. En este ejemplo el servicio **A** necesita al servicio **B**: **A** -> **B**.

En la primera petición que procesáramos se resolvería **A** y se guardaría su instancia (a la que llamaremos **A<sub>1</sub>**) y se resolvería **B** (a la que llamaremos **B<sub>1</sub>**) como parte del contexto o ámbito de esta petición. Hasta aquí no hay ningún problema.

Ahora llega una segunda petición que requiere de **A**. Como **A** es un *Singleton* y ya lo tenemos resuelto simplemente inyectaríamos **A<sub>1</sub>**. Respecto a su dependencia con **B** esperaríamos que se generara una nueva instancia dado que lo tenemos registrado como *Scoped*: **B<sub>2</sub>**. Sin embargo, puesto que **A** es un *Singleton* y que **A<sub>1</sub>** es la instancia que usaremos, ésta irá de la mano de **B<sub>1</sub>** durante todo el ciclo de vida de la aplicación en lugar de obtener nuestras instancias de **B** en cada ámbito o contexto.

### 7.1.2 Configurando el contenedor de dependencias integrado de *ASP.NET Core*

En las aplicaciones desarrolladas usando *ASP.NET Core* la mayor parte de la configuración se hará en la clase **Startup**, la cual está disponible en el fichero *Startup.cs*. La configuración de las dependencias la haremos en el método *ConfigureServices*:

```

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        //Registramos la clase Dependency que será resuelta cuando se requiera un
        IDependency
        services.AddScoped<IDependency, Dependency>();
        services.AddTransient<IDependency,Dependency>();
        services.AddSingleton<IDependency,Dependency>();
        //Registramos IDependency como Singleton, esta vez creando manualmente la
        instancia
        services.AddSingleton<IDependency>(new DedicatedDependency("Manual"));
    }
}

//Definición de las clases usadas en el ejemplo anterior.
public interface IDependency
{
}

public class Dependency : IDependency
{
}

public class DedicatedDependency : IDependency
{
    private string name;
    public DedicatedDependency(string name)
    {
        this.name = name;
    }
}

```

### 7.1.3 Configurando Autofac como contenedor de dependencias

Autofac [38] es un contenedor de inyección de dependencias disponible para .NET. Al igual que el resto de las soluciones y framework usados en este proyecto, es de código libre y disponible en GitHub. Generalmente cuando se usa Autofac como contenedor de dependencias, el registro de los tipos se hará a través del uso de módulos. Esto permite separar el registro agrupando tipos en módulos independientes que podrán, incluso, estar en distintos proyectos.

Autofac es capaz de escanear diferentes ensamblados y registrar los tipos en el contenedor de dependencias a través de convenciones. Esto permite registrar todos los tipos que extiendan una determinada interfaz de una forma específica. O registrar todas las clases cuyo nombre acabe de una manera específica. La necesidad de registrar tipos a través de convenciones es el motivo por el cual Autofac es el IoC elegido en el microservicio de Proyectos.

Para instalar Autofac en *ASP.NET Core* simplemente hay que ir al gestor de paquetes de Visual Studio, o usar el CLI de dotnet, e instalar el paquete: `Autofac.Extensions.DependencyInjection`.

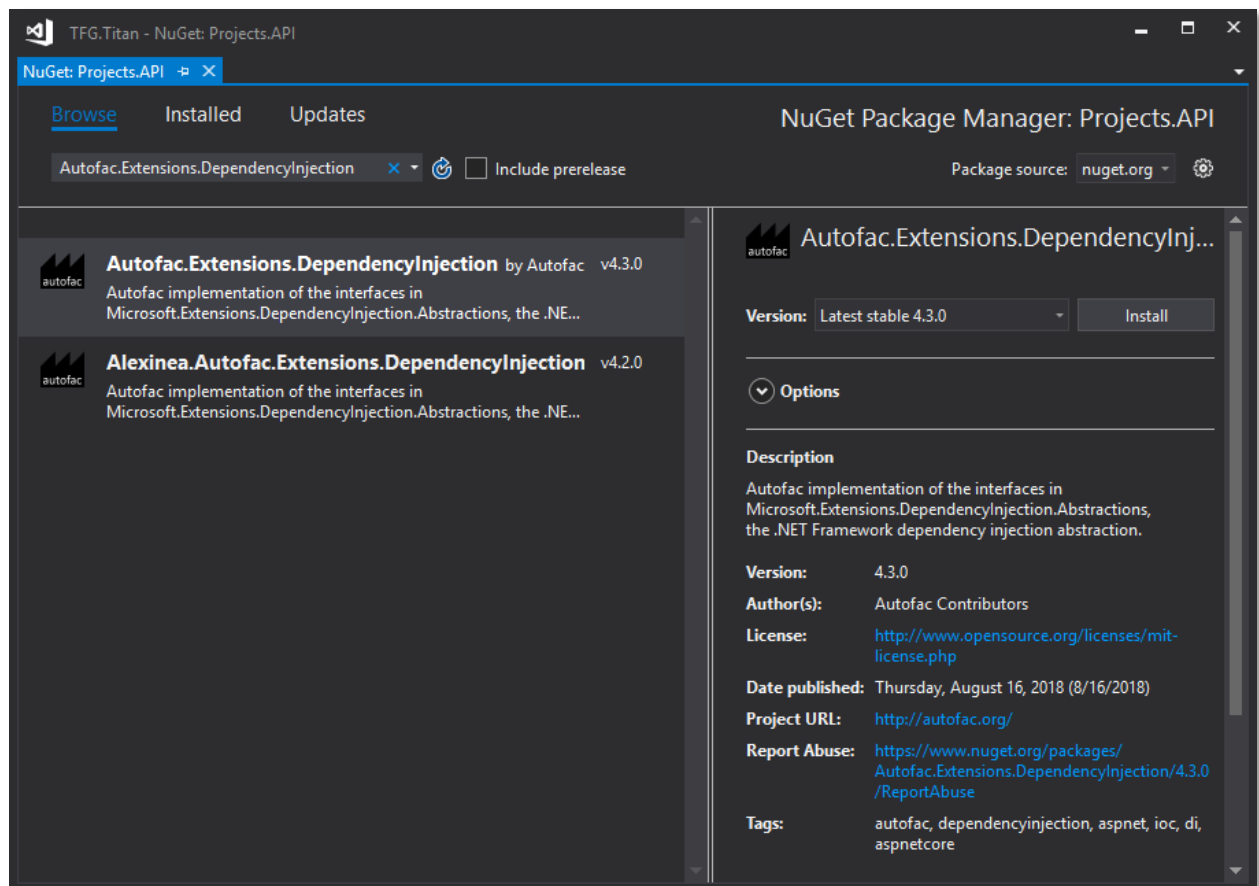


Figura 7-1. Instalación Autofac desde el gestor de paquetes

Con las dependencias instaladas en el proyecto hay que configurarlo para que use Autofac en lugar de usar el contenedor de dependencias integrado. Esto se hará en la clase `Startup`, cambiando la firma del método `ConfigureServices` de la siguiente forma:

```
public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //Hacemos la configuración igual que con el integrado
        services.AddMvc();

        //Creamos el contenedor de Autofac
        var container = new ContainerBuilder();
        //Añadimos los servicios ya registrados
        container.Populate(services);

        //Registramos los distintos módulos que componen la aplicación
        container.RegisterModule(new MediatorModule());
        container.RegisterModule(new ApplicationModule());

        //Devolvemos el contenedor configurado para ser usado
        return new AutofacServiceProvider(container.Build());
    }
}
```

En Autofac el ciclo de vida de las dependencias es similar al integrado en *ASP.NET Core*:

- **InstancePerLifetimeScope**: es el correspondiente a **Scoped**.
- **InstancePerDependency**: el correspondiente a **Transient**.
- **Singleton**: el correspondiente a **Singleton**.

Con la implementación del patrón CQRS en el microservicio de Solicitudes se verá la definición de los módulos de Autofac y el escaneado de los ensamblados para el registro automático de tipos.

## 7.2 Gestión del acceso a datos

A la hora de gestionar los modelos de datos hay varios factores que hay que tener en cuenta, diversos patrones y conceptos de los que se hablará en esta sección, empezando por las entidades.

### 7.2.1 Entidades de dominio

Las Entidades que representan los objetos de dominio son definidas principalmente por su identidad y su persistencia a lo largo del tiempo y no únicamente por los atributos que las componen. Eric Evans [40] define las entidades de la siguiente forma: *“un objeto primariamente definido por su identidad (clave*

*única por la que nos referimos a él) es llamado entidad".* Las entidades son muy importantes en los modelos de dominio dado que son la base de ellos. Es por ello por lo que deben ser identificados y diseñados con cautela.

La misma identidad (misma clave identificadora) puede aparecer en distintos contextos acotados (o BC, de lo que se habla en el capítulo 6) pero ello no implica que la misma entidad, con la misma lógica y atributos, sea implementada de la misma forma en los diferentes contextos. Una entidad tendrá los atributos y comportamientos establecidos dentro del contexto al que pertenezca.

Dentro de DDD una entidad del dominio debe implementar la lógica de dominio y comportamiento de la que es responsable y no sólo almacenar los datos referentes a la misma. Cuando la lógica referente a una entidad está implementada en clases de la lógica de dominio y no en las entidades podemos caer en el modelo de dominio anémico, definido así por Martin Fowler [41], y que está considerado como un anti patrón dentro de DDD.

#### *7.2.1.1 El patrón Value Object*

Eric Evans [40] expone *"Muchos objetos no tienen una identidad conceptualmente hablando. Estos objetos describen ciertas características de algo"*. A estos objetos los denomina *value objects* y son un tipo inmutable que se distingue simplemente por el estado de sus propiedades. Al contrario que una entidad que tiene un identificador único, dos objetos de este tipo con las mismas propiedades pueden ser considerados iguales.

#### *7.2.1.2 El patrón Agregado y la raíz del agregado*

Un modelo de dominio contiene un conjunto de distintas entidades y procesos que pueden encargarse de un área de funcionalidad amplia. El agregado se define como un grupo de entidades y comportamientos que pueden ser tratados como una unidad cohesiva.

Normalmente los agregados son definidos basándose en las transacciones necesarias en la aplicación desarrollada. Un ejemplo clásico es una orden de compra que también contiene la lista de los productos. Cada elemento de la lista sería una entidad por sí misma que, a su vez, será una entidad hija dentro del agregado orden de compra. Este agregado contendrá la entidad Orden como su entidad raíz, típicamente llamada agregado raíz.

El propósito de la raíz del agregado es asegurar la consistencia dentro de éste: Debe ser el único punto de entrada para actualizaciones, a través de métodos y operaciones en la raíz, del agregado. Los cambios por consiguiente (tanto para la entidad raíz como para las que dependen de este) deben ser realizados únicamente a través de la raíz del agregado. Si se cambiara alguna entidad hija independientemente (o algún *value object* hijo), la raíz no sería capaz de asegurar que el estado del agregado continúe siendo válido.

Para mantener la separación entre agregados y los límites de cada uno, es una buena práctica en DDD el no tener propiedades directas de navegación entre agregados y tener únicamente la clave que lo identifica:

```
public class Submission : Entity, IAggregateRoot
{
    private DateTime _submissionDate;
    private int? _applicantId; //Identificador del agregado de Applicants
    public SubmissionStatus SubmissionStatus { get; private set; }
    private int? _projectId;
    public string Title { get; private set; }
    public int? GetApplicantId => _applicantId;
    public string Comment { get; private set; }
}
```

La clase base Entity se puede consultar en los anexos.

### 7.2.2 Enumeraciones usando clases en vez de los enumerados integrados en el lenguaje

Los enumerados son una característica de los lenguajes de programación que están montados sobre el tipo entero. Se usan cuando se quiere almacenar un valor de una lista cerrada de valores, por ejemplo, para clasificar tallas (S,M,L,XL). Sin embargo, usar enumerados para el control de flujos complejos puede ser considerado una mala práctica.

Dado que son básicamente tipos enteros, es muy fácil especificar un valor que esté fuera del rango del enumerado pasándolo como entero y provocar resultados inesperados. Por ello una posibilidad es implementar los enumerados como clase aprovechando elementos de la orientación a objetos. La clase base *Enumeration* se puede consultar en los anexos.

```
public class SubStatus : Enumeration
{
    public static SubStatus Approved = new SubStatus(1, "Approved");
    public static SubStatus Pending = new SubStatus(2, "Review");
    public static SubStatus Cancelled = new SubStatus(3, "Cancelled");

    protected SubStatus(int id, string name) : base(id,name)
    {
    }

    public static IEnumerable<SubStatus> List() => GetAll<SubStatus>();

    public static SubStatus FromName(string name){...}
}
```

### 7.2.3 Repositorios

Los repositorios son clases o componentes que encapsulan la lógica requerida para acceder a las distintas fuentes de datos. Centralizan la funcionalidad común de acceso a los datos, mejor mantenibilidad y desacoplando la infraestructura o tecnología usada para acceder a las bases de datos de la capa de dominio. El patrón repositorio es una forma de trabajar con una fuente de datos muy bien documentada .

Debería crearse una clase repositorio por cada agregado o raíz del agregado. En un microservicio que esté basado en patrones DDD, el único canal por el que actualizar la base de datos deben ser los repositorios. Esto es debido a que mantienen una relación uno a uno con la raíz del agregado, el cual mantiene la integridad y consistencia.

Básicamente, un repositorio permite obtener datos en memoria que vienen de la base de datos en forma de entidades del dominio. Una vez que las entidades están en memoria pueden ser actualizadas y, posteriormente, almacenadas en la base de datos.

#### 7.2.3.1 Contratos e implementaciones

Los contratos de los repositorios son simplemente interfaces que expresan los requerimientos de los repositorios de cada agregado. Los contratos se definen en la capa de dominio cuando el microservicio esté basado en patrones DDD. La implementación de los repositorios, dado que implicará especificar qué medio se usará para acceder a las bases de datos, se hará en la capa de infraestructura de la aplicación.

Un patrón relacionado con esta práctica es el *Separated Interface pattern* [42]. Martin Fowler lo explica de la siguiente forma: “Consiste en definir la interfaz en un paquete y la implementación en otro distinto. De esta forma un cliente que es dependiente de la interfaz puede ser completamente agnóstico de la implementación”.

Los contratos definidos en la capa de dominio serían así:

```
public interface ISubmissionRepository : IRepository<Submission>
{
    Submission Add(Submission submission);
    Submission Update(Submission submission);
    Task<Submission> GetAsync(int submissionId);
}

//Definido en IRepository.cs como parte del SeedWork (Anexo 1)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```



Este patrón posibilita tener dependencia en los requerimientos definidos en la capa de dominio y no una dependencia directa a la capa de infraestructura. Además, gracias a la inyección de dependencias, es posible aislar la implementación entre las distintas capas.

Como se ha comentado, lo ideal es tener un repositorio por cada agregado raíz. Para asegurar que el código cumple este requerimiento, las interfaces que definen cada uno de los repositorios extenderán de una interfaz *IRepositorio* con un tipo genérico. El genérico especificará que serán del tipo agregado raíz.

```
public interface IRepository<T> where T : IAggregateRoot
```

## 7.2.4 Entity Framework Core

Entity Framework (EF) Core es un ORM (*Object Relational Mapping*) o mapeo objeto-relacional en castellano. ORM es una técnica de programación para convertir datos entre el sistema de tipos de un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.

EF Core es ligero, de alto rendimiento, multiplataforma y se integra perfectamente con *ASP.NET Core*, razón por la que se ha utilizado en este proyecto. El uso de EF se hará en la capa de infraestructura dado que es la implementación directa de acceso a la base de datos.

Una capacidad muy importante de EF es la de trabajar con las entidades de dominio y almacenarlas en la base de datos de nuestra elección. EF puede trabajar con distintos proveedores como lo son SQL Server, SQL Lite, PostgreSQL, MySQL, etc.

### 7.2.4.1 Configuración del mapeo de las entidades

EF ofrece un modo de mapear un modelo de dominio a la base de datos sin alterar las clases que definen las entidades, y por tanto, sin modificar la capa de dominio. Este mapeo, que se hará en la capa de persistencia (infraestructura), consiste en especificar la tabla a la que corresponderá una entidad, la relación entre entidades, el tipo y el nombre de cada columna, entre otros.

```
class SubmissionEntityTypeConfiguration : IEntityTypeConfiguration<Submission>
{
    public void Configure(EntityTypeBuilder<Submission> builder)
    {
        builder.ToTable("submissions");
        builder.Ignore(b => b.DomainEvents);
        builder.HasKey(s => s.Id);
        builder.HasOne(s => s.SubmissionStatus)
            .WithMany()
            .HasForeignKey("SubmissionStatusId");
    }
}
```

#### 7.2.4.2 Contexto de Entity Framework Core

Para usar Entity Framework se empieza generando un contexto de datos. EF Core ofrece una clase llamada *DbContext* con la funcionalidad necesaria para acceder a las entidades a través de consultas o hacer nuevas inserciones.

Para crear un contexto hay que partir de la clase *DbContext* y definir en ella una propiedad de tipo *DbSet<Entity>* por cada una de las entidades con las que se quiera trabajar.

Si se define un constructor que reciba las opciones, como el que se ve en el siguiente fragmento de código, se permitirá configurar elementos como la cadena de conexión desde la aplicación que use el contexto.

También es posible sobrescribir el método *OnModelCreating* para aplicar la configuración de las entidades (las clases implementando *IEntityTypeConfiguration<Entity>*).

```
public class ApplicantsContext : DbContext
{
    public DbSet<Submission> Submissions { get; set; }
    public DbSet<SubmissionStatus> SubmissionStatus { get; set; }
    public DbSet<Applicant> Applicants { get; set; }

    public ApplicantsContext(DbContextOptions<ApplicantsContext> options) :
        base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new SubmissionEntityTypeConfiguration());
        modelBuilder.ApplyConfiguration(new ApplicantEntityTypeConfiguration());
    }
}
```

#### 7.2.4.3 Repositorios con Entity Framework Core

Es posible usar Entity Framework Core directamente desde el código de aplicación (por ejemplo desde los controladores de las APIs). De esta forma el código creado es más simple, accediendo directamente al contexto sin añadir ninguna capa de abstracción adicional.

Sin embargo, crear repositorios añade beneficios cuando se desarrollan aplicaciones o microservicios más complejos. El patrón repositorio permite encapsular la capa de persistencia desacoplándola de las capas de aplicación y de modelo del dominio. Un gran beneficio de todo esto es la posibilidad de hacer pruebas unitarias simulando el acceso a la base de datos de una forma sencilla.

En el microservicio de gestión de solicitudes se ha implementado un repositorio de una forma similar a la siguiente:

```
public class SubmissionRepository : ISubmissionRepository
{
    private readonly ApplicantsContext _context;
    public IUnitOfWork UnitOfWork => _context;

    public SubmissionRepository(ApplicantsContext context)
    {
        _context = context ?? throw new ArgumentNullException(nameof(context));
    }

    public Submission Add(Submission submission)
    {
        return _context.Submissions.Add(submission).Entity;
    }

    public Submission Update(Submission submission)
    {
        return _context.Submissions.Update(submission).Entity;
    }

    public async Task<Submission> GetAsync(int submissionId)
    {
        var submission = await _context.Submissions
            .Include(s => s.SubmissionStatus)
            .Where(s => s.Id == submissionId)
            .SingleOrDefaultAsync();

        return submission;
    }
}
```

La misma instancia del contexto de EF debe ser compartida por todos los repositorios dentro de la misma petición para mantener la consistencia y ser una unidad. A este patrón se le denomina unidad de trabajo que en inglés es conocido como *Unit of Work*.

Para asegurar que la misma instancia del contexto sea compartida dentro de una petición (véase sección de inyección de dependencias) se registrará como *Scoped* en el contenedor a través del método `ConfigureServices`.

```

public IServiceCollection ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    //Registramos el contexto de EF como Scoped
    services.AddEntityFrameworkSqlServer()
        .AddDbContext<ApplicantsContext>(options =>
        {
            //Definimos las opciones que recibirá el contexto
            //leyendo la cadena de conexión de la configuración
            options.UseSqlServer(Configuration["ConnectionString"]);
        }, ServiceLifetime.Scoped);
}

```

### 7.2.5 Eventos de dominio

Un evento es algo que ocurrió en el pasado y en consecuencia, un evento de dominio es algo que ha ocurrido en un dominio particular de la aplicación y que se quiere que otras partes del mismo dominio estén al tanto y, potencialmente, puedan reaccionar ante ello.

En la arquitectura DDD se usarán los eventos de dominio para propagar cambios entre múltiples agregados del mismo dominio. Un beneficio importante de los eventos de dominio es la posibilidad de expresar explícitamente los efectos secundarios en vez de que sea algo implícito. Por ejemplo, en el microservicio de solicitudes, el crear una solicitud no trata únicamente de la solicitud; actualiza o crea el agregado solicitante basándose en el usuario original, debido a que un usuario no es un solicitante hasta que solicita participar en un proyecto. Usando eventos de dominio se puede definir esta regla del dominio de forma explícita.

La comunicación basada en eventos no se implementa directamente en los agregados, deben implementarse manejadores de los eventos en la capa de aplicación. La capa de dominio debe encargarse únicamente de la lógica de dominio pero no, por ejemplo, de persistir un cambio en una base de datos como reacción a un evento.

Es importante recalcar que un evento puede desencadenar múltiples acciones, es posible que estas sean procesadas asíncronamente, no se esperará por el resultado de la ejecución de cada uno y la aplicación debe estar diseñada de tal forma que sea posible añadir nuevos manejadores para un evento sin modificar el código original. Esto es el principio Abierto/Cerrado[43] de SOLID[31].

#### 7.2.5.1 Implementación de los eventos de dominio

Un evento de dominio en C# es un simple objeto de datos con información sobre lo que acaba de ocurrir en el dominio:

```

public class SubmissionStartedDomainEvent : INotification
{
    public Submission Submission { get; }
    public string UserId { get; }
    public string UserName { get; }

    public SubmissionStartedDomainEvent(Submission submission, string userId,
                                         string userName)
    {
        Submission = submission;
        UserId = userId;
        UserName = userName;
    }
}

```

En este ejemplo se puede observar la clase con los datos sobre el evento de solicitud iniciada. Una característica importante es que, debido a que un evento es algo que ya ha ocurrido, es inmutable y las propiedades son de solo lectura por lo que no podrá ser modificado una vez creado.

#### 7.2.5.2 Publicar eventos de dominio

Para publicar los eventos de dominio y que éstos lleguen a los manejadores podemos optar por su publicación **justo en el momento en el que ocurren** o **diferir su publicación**. En este proyecto se ha optado por la segunda.

Para ello se añaden los eventos a una colección o lista de eventos por entidad. Esta lista será parte de la entidad base como se puede ver a continuación (la implementación completa se puede consultar en el anexo):

```

public abstract class Entity
{
    private List<INotification> _domainEvents;
    public IReadOnlyCollection<INotification> DomainEvents => _domainEvents;

    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }

    public void RemoveDomainEvent(INotification eventItem)
    {
        _domainEvents?.Remove(eventItem);
    }
}

```

Basándose en el código anterior, cuando se quiere publicar un evento simplemente hay que añadirlo a la lista desde cualquier método de la raíz del agregado. En el siguiente fragmento se puede ver un ejemplo en el agregado de Submission:

```
var submissionStartedEvent = new SubmissionStartedDomainEvent(this, userId);  
this.AddDomainEvent(submissionStartedEvent);
```

*AddDomainEvent* está simplemente añadiendo el evento a la lista pero no se está publicando o invocando ningún manejador en ese momento. Se procederá a despachar los eventos después, a la hora de confirmar los cambios en la base de datos. En este caso, dado que se está usando *Entity Framework*, se hará justo antes de llamar al método *SaveChanges* del contexto.

```
public class ApplicantsContext : DbContext, IUnitOfWork  
{  
    public async Task<bool> SaveEntitiesAsync()  
    {  
        //Se despachan los eventos las entidades que forman parte del contexto  
        await _mediator.DispatchDomainEventsAsync(this);  
        //Una vez procesados todos los eventos se guardan los cambios en la BBDD  
        var result = await base.SaveChangesAsync();  
        return true;  
    }  
}
```

Al despachar los eventos justo antes de guardar los cambios aseguramos la integridad de la transacción. Esta decisión se toma en la capa de infraestructura pues depende del almacenamiento que se esté usando.

La forma en la que se despachan los eventos en el proyecto hace uso del patrón mediador que se explica más adelante en este mismo capítulo.

#### 7.2.5.3 Manejadores de eventos de dominio

Los manejadores serán definidos en la capa de aplicación, a diferencia de los eventos que son parte de la capa de dominio, dado que interactúan con los repositorios que son parte de la de infraestructura.

Como ejemplo de implementación se puede ver el siguiente manejador, parte del microservicio de solicitudes:

```

public class ValidateOrAddApplicantAggregateWhenSubmissionStartedDomainEventHandler
    : INotificationHandler<SubmissionStartedDomainEvent>
{
    private readonly IApplicantRepository _repository;

    public ValidateOrAddApplicantAggregateWhenSubmissionStartedDomainEventHandler(
        IApplicantRepository repository)
    {
        _repository = repository;
    }

    public async Task Handle(SubmissionStartedDomainEvent notification)
    {
        var applicant = await _repository.FindAsync(notification.UserId);
        var exists = applicant is null ? false : true;

        if (!exists)
        {
            applicant = new Applicant(notification.UserId, notification.UserName);
        }

        applicant.Verify(notification.Submission.Id);

        var updated = exists ? _repository.Update(applicant) :
            _repository.Add(applicant);

        await _repository.UnitOfWork.SaveEntitiesAsync();
    }
}

```

### 7.2.6 Eventos de integración

Al igual que los eventos de dominio, un evento de integración es algo que ocurrió en el pasado. Los de integración sin embargo son usados para mantener la consistencia y sincronizar el estado de las entidades del dominio a lo largo de distintos microservicios o sistemas externos. Esto se consigue publicando los eventos de integración fuera del microservicio. Cuando un evento es publicado a los múltiples microservicios receptores que estén suscritos al evento de integración, el manejador apropiado en cada uno de los microservicios se encargará de procesarlo.

Un evento de integración es básicamente un objeto de datos como el siguiente, que será publicado cuando en el microservicio de gestión de proyectos se cambie el título de uno de ellos.

```
public class ProjectTitleChangedIntegrationEvent : IntegrationEvent
{
    public int ProjectId { get; private set; }
    public string NewTitle { get; private set; }

    public ProjectTitleChangedIntegrationEvent(int projectId, string newTitle)
    {
        ProjectId = projectId;
        NewTitle = newTitle;
    }
}
```

El evento de integración puede ser definido en la capa de aplicación de cada servicio, estando así desacoplado de otros microservicios. No es una buena idea compartir un modelo común a lo largo de múltiples microservicios: los microservicios deben ser completamente autónomos.

#### 7.2.6.1 Bus de eventos

El bus de eventos permite comunicaciones del estilo publicador/suscriptor entre microservicios sin la necesidad de que el que publica el mensaje y los que lo reciben tengan consciencia de la existencia del resto. En el patrón publicador/suscriptor existe un elemento llamado bróker o bus de eventos, un componente externo a los procesos y que es conocido por ambos extremos, lo que permite desacoplar todas las partes.

Un bus de eventos típicamente se compone de dos partes: una abstracción o interfaz y una o varias implementaciones. Se pueden tener distintas implementaciones dependiendo de las necesidades del entorno, desde implementaciones ejecutándose en nuestra infraestructura como puede ser *RabbitMQ*[44] o soluciones en entornos Cloud como *Azure Service Bus*[45].

Una definición sencilla de la interfaz que cumplirán las implementaciones puede ser la siguiente:



```

public interface IEventBus
{
    void Publish(IntegrationEvent @event);

    void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>;

    void Unsubscribe<T, TH>()
        where TH : IIntegrationEventHandler<T>
        where T : IntegrationEvent;
}

```

El método *Publish* permite enviar el evento de integración a cualquiera que esté suscrito a ese evento. En este caso es el método usado por el microservicio que quiere publicar el evento.

El método *Subscribe* es usado por los microservicios que quieren recibir eventos. Tiene dos parámetros, uno para indicar el tipo del evento al que se quiere suscribir y el segundo es el tipo manejador que se ejecutará cuando el evento sea recibido.

#### 7.2.6.2 Suscripción a eventos

La suscripción a eventos debe hacerse cuando el servicio esté iniciándose (en las apps de ASP.NET Core como las que se han implementado aquí se haría en la clase Startup). El microservicio de gestión de solicitudes necesita suscribirse al evento *ProjectTitleChangedIntegrationEvent* para poder estar al tanto de los cambios al título que ocurran en los proyectos a los que se ha aplicado.

```

var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProjectTitleChangedIntegrationEvent,
    ProjectTitleChangedIntegrationEventHandler>();

```

Cuando ese código se ejecuta, cada vez que un evento de este tipo sea publicado se invocará el manejador que se está registrando y se procesará el evento.

#### 7.2.6.3 Publicación de eventos

Para publicar los eventos a través del bus de eventos el microservicio de origen usará un código similar al siguiente, que pertenece al microservicio de gestión de proyectos (de forma simplificada). Lo primero es inyectar por constructor el objeto *IEventBus*:

```

[Route("api/v1/[controller]")]
[ApiController]
public class ProjectsController : ControllerBase
{
    private readonly IEventBus _eventBus;
    private readonly ProjectsContext _context;

    public ProjectsController(IEventBus eventBus, ProjectsContext context)
    {
        _eventBus = eventBus;
        _context = context;
    }
}

```

Luego en el método *UpdateProduct* se procede a actualizar el producto (el código se ha simplificado para el ejemplo de los eventos de integración) y posteriormente se publica el evento usando el bus de eventos en caso de que el título haya sido actualizado:

```

[Route("")]
[HttpPut]
public async Task<IActionResult> UpdateProject(Project project)
{
    var item = await _context
        .Projects
        .SingleOrDefaultAsync(p => p.Id == project.Id);

    //...
    if(item.Title != project.Title)
    {
        var @event = new ProjectTitleChangedIntegrationEvent(item.Id,
                                                                project.Title);

        //Guardamos los cambios en la base de datos
        await _context.SaveChangesAsync();

        //Publicamos el evento en el bus de eventos
        _eventBus.Publish(@event);
        //...
    }
}

```

En este caso, dado que el microservicio es un simple CRUD, el código se desarrolla en el propio controlador del API. En casos más complejos como CQRS, se haría en el manejador del comando.

#### 7.2.6.4 Procesamiento de los mensajes recibidos de las suscripciones

A la hora de procesar los mensajes, el tipo que implementa *IntegrationEventHandler<T>* definido a la hora de suscribirse al evento, será instanciado y el método llamado **Handle** será invocado. Es en este método donde se implementará la lógica.

```
interface IIntegrationEventHandler<in TIntegrationEvent>:IIntegrationEventHandler
    where TIntegrationEvent : IntegrationEvent
{
    Task Handle(TIntegrationEvent @event);
}
```

No se incluye un ejemplo por ser un caso muy similar a los manejadores de los eventos de dominio, solo cambia la interfaz que implementan.

### 7.3 Patrón-Arquitectura CQRS

CQRS es un patrón arquitectónico que separa los modelos que usamos para leer y escribir datos. Proviene del término CQS (*Command Query Separation*), originalmente definido por Bertrand Meyer, que se basa en la idea de que podemos dividir las operaciones de un sistema en dos categorías:

- **Queries:** Se traduce como consulta. Devuelven un resultado sin afectar al estado de nuestro sistema.
- **Commands:** Significa comando. Son operaciones que modifican el estado de nuestro sistema.

CQS se puede considerar un principio sobre el que se construye CQRS, pero no es el único patrón que cumple este principio. Por ejemplo, el patrón Repositorio también cumple con CQS.

CQRS proviene del inglés *Command and Query Responsibility Segregation*. Es más detallado que CQS dado que organizamos las responsabilidades de los comandos y las consultas en capas distintas. Cada capa tiene sus propios modelos de datos (que no base de datos distinta) y pueden estar en el mismo microservicio o en microservicios distintos.

Podemos verlo también como un patrón basado en comandos y eventos del dominio junto con mensajes asíncronos de forma opcional, que serán procesados sin necesidad de esperar por su resultado.

Con CQRS podemos tener dos objetos distintos para operaciones de lectura y escritura, cuando en otros contextos, podríamos tener estas operaciones en la misma clase. En este proyecto se ha usado este patrón, de una forma simplificada en el microservicio de gestión de proyectos, dado que, aunque es un patrón muy potente, puede conllevar a una complejidad importante si no lo manejamos con cuidado.

### 7.3.1 Los patrones Command y Command Handler

El patrón se basa en aceptar o recibir comandos desde la parte del cliente, procesarlos en base a las reglas definidas por el modelo de dominio y, finalmente, almacenar o confirmar los estados a través de transacciones.

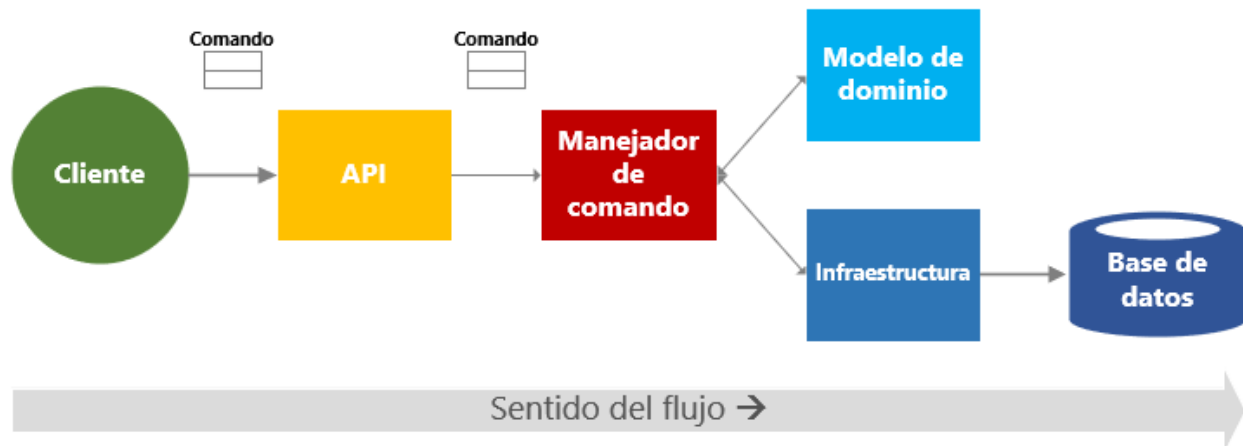


Figura 7-2. Flujo de escritura en CQRS. Fuente: Elaboración propia.

#### 7.3.1.1 Command

Un comando es una petición que ejecutará una acción para cambiar el estado del sistema. Un comando es imperativo y por ello el nombre suele ser un verbo en imperativo (como crear, actualizar) y pueden incluir el agregado sobre el que actuarán: *CreateSubmissionCommand*.

Una característica importante de los comandos es que deben ser procesados una vez por un único manejador. Esto es debido a que un comando es una acción o transacción única que se quiere procesar en la aplicación. Por ejemplo, la acción de crear un proyecto solo debe ser procesada una única vez. En caso de que los comandos sean implementados de una forma idempotente, es decir, tolerantes a reintentos, será el mismo comando el encargado de saber que está siendo ejecutado más de una vez y actuar en consecuencia.

Cuando se habla de la implementación de un comando, se puede ver que es básicamente un objeto que transportará los datos necesarios para su ejecución. Es una clase especial de *DTO* (*Data transfer object* – *objeto de transferencia de datos*), que es específicamente usado para solicitar cambios o transacciones. Como nota importante, el comando incluirá únicamente los datos necesarios para ser procesado.

El siguiente código es un ejemplo de un comando del proyecto de gestión de solicitudes.

```

public class CreateSubmissionCommand : IRequest<bool>
{
    public string UserId { get; private set; }
    public string UserName { get; private set; }
    public int ProjectId { get; private set; }
    public string Title { get; private set; }
    public string Comment { get; private set; }

    public CreateSubmissionCommand(string userId, string userName,
                                   int projectId, string title,
                                   string comment)
    {
        UserId = userId;
        UserName = userName;
        ProjectId = projectId;
        Title = title;
        Comment = comment;
    }
}

```

Los comandos son inmutables dado que su propósito es ser procesados directamente por la capa de dominio, motivo por que la modificación de sus propiedades está limitada al propio comando. El nombre indica el propósito del comando y, aunque en lenguajes como C# se implementan como clases, éstos no son clases completas dado que son simplemente estructuras de datos y no tienen ningún comportamiento adicional.

#### 7.3.1.2 Command handler

Se puede traducir directamente al castellano como manejador del comando. Se debe implementar un manejador por cada uno de los comandos. Ésta es la esencia en la que se basa el patrón y es donde se usará el objeto del comando, los objetos del dominio y los repositorios implementados en la capa de infraestructura. El manejador es el corazón de la capa de aplicación en términos de CQRS y DDD. Sin embargo, toda la lógica de dominio debe ser implementada, como se ha ido comentando, en las entidades de dominio, agregados y demás clases del dominio.

El manejador recibe el comando y obtiene el resultado del agregado que corresponda. El resultado debe ser o la confirmación de que todo ha ido bien o una excepción en caso de que no. En caso de ocurrir una excepción el estado del sistema no debería cambiar.

Los pasos que se suelen seguir a la hora de procesar un comando son los siguientes:

- Recibe el objeto del comando.
- Valida si el comando si procede (en caso de que no haya sido comprobado anteriormente).

- Instancia a la raíz del agregado a la que se aplica el comando.
- Ejecuta el método en el agregado obteniendo los datos necesarios del comando.
- Almacena el nuevo estado del agregado en la base de datos relacionada. La última operación que se realiza es la transacción.

Un manejador debería tratar con un único agregado. En caso de necesitar tratar con varios agregados los estados podrían propagarse usando eventos de dominio.

Como ejemplo, el siguiente código muestra el manejador del comando *CreateSubmission*. Como se puede observar en el constructor, y gracias a la inyección de dependencias, se obtienen los recursos necesarios para procesar el comando. En el método *Handle* es donde se procesa realmente el comando.

```
class CreateSubCommandHandler : IRequestHandler<CreateSubmissionCommand, bool>
{
    private readonly ISubmissionRepository _submissionRepository;

    public CreateSubCommandHandler(ISubmissionRepository repository)
    {
        _submissionRepository = repository;
    }
    public async Task<bool> Handle(CreateSubmissionCommand request)
    {
        var submission = new Submission(request.UserId,
            request.UserName, request.ProjectId,
            request.Title, request.Comment);
        _submissionRepository.Add(submission);

        return await _submissionRepository
            .UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

### 7.3.2 Lanzar y procesar el comando

Una vez definidos los comandos y los manejadores para cada uno de ellos, el siguiente punto es ver cómo se invoca un manejador dado un determinado comando. Una opción es instanciarlo directa y manualmente desde el controlador del API en cuestión. Sin embargo, ese enfoque es demasiado manual y a su vez muy acoplado.

Las opciones recomendadas serían:

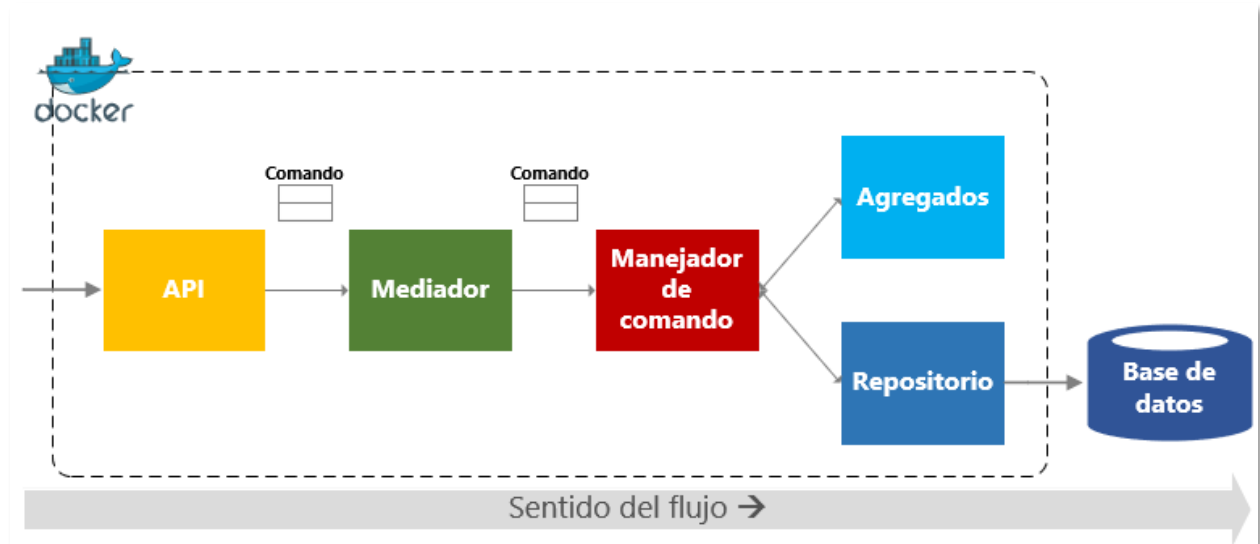
- El uso de un patrón mediador en memoria.

- El uso de mensajes asíncronos entre controladores y manejadores a través de una cola.

En los microservicios desarrollados durante este trabajo se ha optado por el patrón mediador, que se explica a continuación.

### 7.3.3 Patrón mediador

Un mediador es un bus inteligente en memoria capaz de invocar el manejador adecuado basado en el tipo del comando que recibe. Una evolución de la figura 7.2 sería la siguiente en la que ya se incluye el mediador en el flujo (basado en la implementación que se ha seguido en este trabajo).



*Figura 7-3. Implementación CQRS en el microservicio de gestión de solicitudes. Fuente: Elaboración propia.*

Procesar las peticiones puede complicarse en cuanto las aplicaciones empiezan a crecer. Motivos para ello pueden ser los conceptos de validación, telemetría, seguridad, auditoría... El mediador es un objeto que encapsula la forma en la que coordina la ejecución basándose en el estado del elemento a procesar, la manera en la que un manejador es invocado o cómo le llega la información al manejador. Puede proveer de mecanismos para aplicar los citados conceptos de una forma sencilla.

En este proyecto se ha utilizado un mediador de código abierto llamado *MediatR* [46].

### 7.3.4 MediatR como implementación del patrón mediador

*MediatR* es una implementación para .NET Core creada por Jimmy Bogard. Es de código abierto, ligero y simple pero no por ello poco potente. Además de procesar mensajes permite aplicar decoradores[47] y comportamientos [48]

Para usar *MediatR* simplemente hay que instalar el paquete que está disponible en NuGet (véase sección 7.1.3 donde se explica cómo instalar los paquetes).

A la hora de pensar en los controladores de las API hay un factor muy importante para tener en cuenta: estos son simples interfaces de entrada y salida que, en este caso específico, procesan peticiones en un servidor web. Por tanto, **la labor de los controladores debe limitarse a la gestión petición en si misma pero no a la lógica de dominio que desencadenará**. Al igual que en una aplicación de escritorio no se implementaría toda la lógica de dominio en la invocación de un botón, aquí pasaría lo mismo, puesto que es, en esencia, una interfaz sobre la que actúa el usuario.

Si no se hace caso a la premisa anterior y se implementa la lógica de dominio directamente en el controlador, el constructor de éste necesitaría todas las dependencias, convirtiéndose en algo difícil de manejar. Sin embargo, cuando se implementa el patrón mediador, el constructor del controlador podrá ser algo tan sencillo como lo siguiente:

```
public class SubmissionsController : ControllerBase
{
    private readonly IMediator _mediator;
    public SubmissionsController(IMediator mediator)
    {
        _mediator = mediator;
    }
}
```

El controlador recibe como dependencia a un objeto implementando la interfaz *IMediator*. Esta es la que provee MediatR para que sea posible su utilización.

El código de la acción para usar este mediador podría ser prácticamente una línea, pasando directamente el comando recibido en la petición al mediador que se encargará de encontrar al responsable de procesarlo a través del método *Send*. Después de ello podrá actuar en consecuencia basándose en la ejecución del manejador:

```
[HttpPost]
public async Task<IActionResult> CreateSubmission(CreateSubmissionCommand
command)
{
    var result = await _mediator.Send(command);

    if (result)
    {
        return Ok();
    }

    return BadRequest();
}
```



Un comando puede empezar en un controlador así como en otros lugares de la aplicación como, por ejemplo, un evento de integración.

Antes de hablar de la implementación específica de comandos con MediatR, se ha mostrado un ejemplo de comando. A la hora de definir el comando se puede ver que éste implementa la interfaz *IRequest<T>*

```
public class CreateSubmissionCommand : IRequest<bool>
```

*IRequest<T>* es parte de MediatR y es la interfaz que todos los comandos deben implementar para poder ser procesados. El parámetro indica el tipo del objeto que será devuelto una vez procesado por el manejador. En el ejemplo del que se está hablando se puede ver que devolverá un tipo booleano.

Una vez definido el comando se procede a definir su manejador. Cuando se ha mostrado el ejemplo de un manejador se puede ver que éste parte de la clase *IRequestHandler<T, R>* que también forma parte de *MediatR* y que implementa el método *Handle*. Dicho método recibe el comando y se encarga de procesarlo.

```
public class CreateSubCommandHandler : IRequestHandler<CreateSubCommand, bool>
{
    public async Task<bool> Handle(CreateSubCommand request)
    {
    }
}
```

Al igual que los comandos, previamente se ha mostrado un ejemplo de evento de dominio sin hablar de la implementación específica con MediatR. A la hora de definir el evento se puede observar que implementa la interfaz *INotification*, que es también parte de MediatR, y debe ser implementada por todos los eventos de dominio.

```
public class SubmissionStartedDomainEvent : INotification
```

Para definir los manejadores de los eventos se sigue un proceso similar al de los comandos. En este caso estos implementan la interfaz *INotificationHandler<T>* definiendo el método *Handle*. Sin embargo, dada la naturaleza de los eventos de dominio y al contrario que con los comandos, este método no devolverá ningún resultado. Esto es debido a que un evento puede ser procesado por varios manejadores y no se espera específicamente por el resultado de ellos cuando son lanzados.

**Nota:** a pesar de que se ha indicado que el método *Handle* no devuelve ningún resultado, se puede ver que el método realmente devuelve en su firma un objeto de tipo *Task*. Esto es debido a la forma que se gestiona la asincronía en C#.

```

public class ValidateOrAddApplicantWhenSubmissionStartedDomainEventHandler
    : INotificationHandler<SubmissionStartedDomainEvent>
{
    public async Task Handle(SubmissionStartedDomainEvent notification)
    {
    }
}

```

### 7.3.5 Registro de MediatR en el contenedor de dependencias

Para poder usar MediatR en la aplicación y que sea capaz de encontrar los comandos y sus respectivos manejadores, es necesario registrarlo todo en el contenedor de dependencias. Como se comentó a la hora de hablar de inyección de dependencias, en este proyecto se ha usado Autofac en determinados microservicios debido a la necesidad del registro automático de tipos.

```

public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        //Registro para poder referirnos a IMediator desde la aplicación
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        //Registro automático de los manejadores de comandos
        var assembly = typeof(CreateSubmissionCommand).GetTypeInfo().Assembly;
        builder.RegisterAssemblyTypes(assembly)
            .AsClosedTypesOf(typeof(IRequestHandler<, >));

        //Registro automático de los manejadores de eventos
        assembly = typeof(ValidateDomainEventHandler).GetTypeInfo().Assembly;
        builder.RegisterAssemblyTypes(assembly)
            .AsClosedTypesOf(typeof(INotificationHandler<>));

        //Registro necesario para que MediatR pueda resolver dependencias
        //a la hora de instanciar manejadores
        builder.Register<ServiceFactory>(context =>
        {
            var c = context.Resolve<IComponentContext>();
            return t => { return c.TryResolve(t, out object o) ? o : null; };
        });
    }
}

```

Se puede observar en el siguiente módulo de Autofac que se están realizando dos registros automáticos: los comandos y los eventos de dominio. Primero se busca el ensamblado que contiene los comandos y luego se registran automáticamente los tipos que implementan la interfaz `IRequestHandler<T,R>`. Posteriormente se busca el ensamblado que tienen los manejadores de los eventos y luego se procede a registrar automáticamente los tipos que implementan la interfaz `INotificationHandler<T>`.

La forma en la que el método **Send** de `IMediator` encuentra el manejador para un determinado comando A es buscando en la lista de los `IRequestHandler<T,R>` registrados uno cuyo T sea A. En el ejemplo anterior el manejador mostrado es para el comando *CreateSubCommand*.

El método **Publish** funciona con los eventos de dominio de una forma similar al método **Send** cuando se trata de comandos. Éste buscará los manejadores para el evento entre la lista de los `INotificationHandler<T>` registrados cuyo T sea el tipo del evento a procesar.

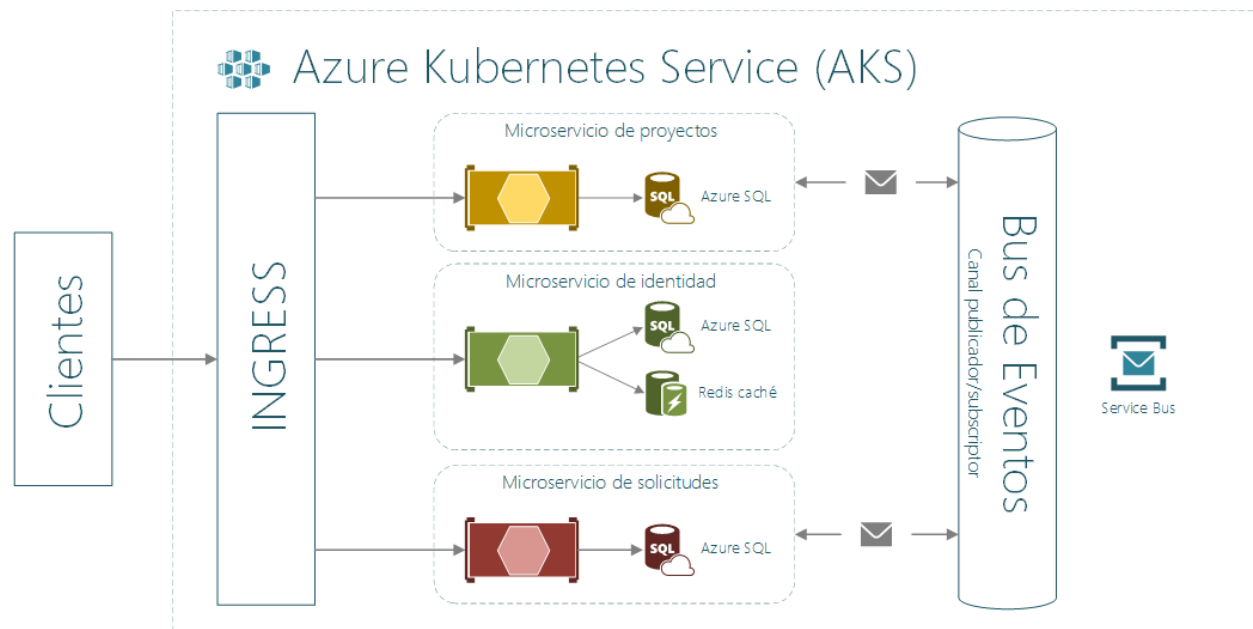
## 8. DESARROLLO DE LA SOLUCIÓN “TITÁN”

Poniendo en práctica todos los conceptos expuestos a lo largo de esta memoria, se ha desarrollado un ejemplo de arquitectura de microservicios desplegada en contenedores, llamada Titán en honor al satélite más grande de Saturno. La aplicación ofrece al usuario la posibilidad de ver un listado de proyectos y solicitar ser el encargo de desarrollarlo.

### 8.1 Visión general de la arquitectura

Es una aplicación multiplataforma desarrollada usando .NET Core. Los servicios son capaces de ejecutarse en Linux y en Windows gracias a los contenedores y dependiendo del sistema operativo en el que se ejecute el host de Docker.

La arquitectura se compone de distintos microservicios autónomos (cada uno es dueño y responsable de sus bases de datos) que implementan distintos enfoques y patrones usando HTTP como protocolo de comunicación. Además, soportan la comunicación asíncrona a través de eventos de integración y buses de eventos que aseguran la integridad de los datos en múltiples microservicios.



*Figura 8-1. Arquitectura de la solución Titán. Fuente: Elaboración propia.*

Como puede observarse en el diagrama, está compuesto por 3 microservicios, uno para la gestión de identidad, otro para la gestión de solicitudes y otro para la gestión de proyectos.

En el diagrama elementos como las bases de datos, el bus de eventos o el orquestador aparecen ya identificados en los servicios usados de la nube de Microsoft Azure en los que la solución ha sido desplegada. Sin embargo durante el desarrollo se han usado equivalentes locales como son SQL Server LocalDB [49].

## 8.2 Microservicio de identidad

Este microservicio se encarga de la gestión de identidad, usuarios y perfiles. Es donde ocurre la autenticación, es decir, es el encargado de identificar quién es el usuario que está iniciando sesión.

Hace las funciones de STS [50], un servicio de generación de tokens de seguridad. Cuando se trata de un API, la forma idónea de autenticar y autorizar al usuario es usando tokens (en lugar de otros mecanismos, como cookies). Los protocolos más usados para este menester son OAuth 2.0 [51] y *OpenId Connect* [52] que funciona sobre OAuth 2.0

El microservicio de identidad está compuesto de un proyecto, llamado Identity.API, que actúa de proveedor OpenId y ofrece vistas para el registro e inicio de sesión. Es un proyecto de ASP.NET Core MVC que además hace uso de Identity Server 4 [53]. Identity Server (como todo lo usado en el proyecto) es de código abierto y multiplataforma.

Cada uno de los proyectos que utilicen el servicio de identidad tienen que ser configurados de la siguiente forma, indicando cuál será el servidor encargado de generar los tokens y la audiencia para la cual son generados:

```
var identityUrl = configuration.GetValue<string>("IdentityUrl");

services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>
{
    options.Authority = identityUrl;
    options.RequireHttpsMetadata = false;
    options.Audience = "applicants";
});
```

En ese caso la audiencia es para el microservicio de gestión de solicitudes (*applicants*).

## 8.3 Microservicio de gestión de solicitudes

Este es el microservicio más complejo y en el que se han aplicado más patrones. Se encarga de la gestión de solicitudes, es decir, es donde los alumnos dirán qué proyecto querrán desarrollar y se les aceptará o denegará.

Se aplica la metodología DDD que organiza los distintos elementos en capas: aplicación, modelo de dominio e infraestructura. En el siguiente diagrama se pueden ver los patrones que se han aplicado en cada proyecto que compone el microservicio.

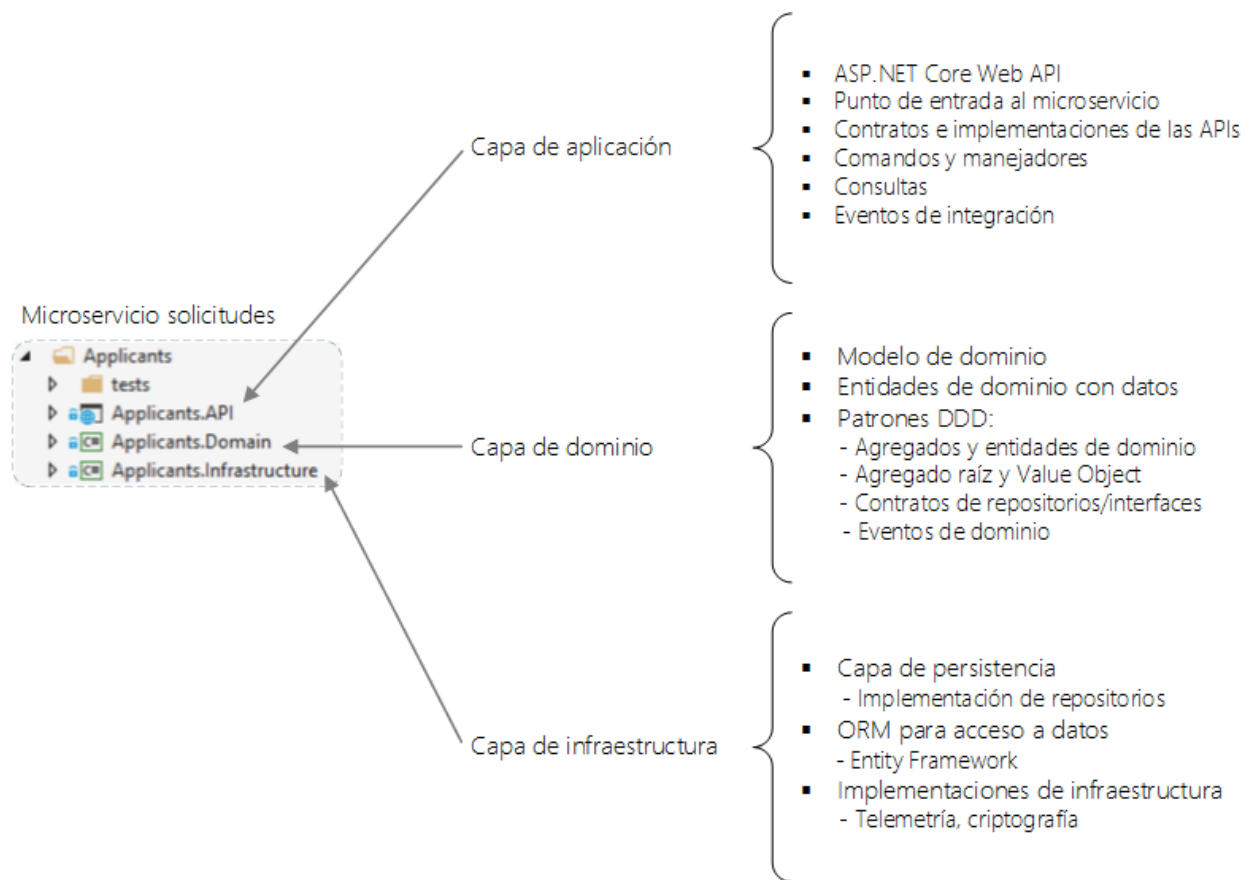


Figura 8-2. Capas DDD en el microservicio de gestión de solicitudes. Fuente: Elaboración propia.

## 8.4 Microservicio de gestión de proyectos

Es el microservicio encargado de la gestión de proyectos. Expone un API CRUD para añadir, leer, actualizar y eliminar proyectos. Todo el código se implementa en un único proyecto ASP.NET Core MVC.

Cada vez que se realiza un cambio a un proyecto se lanza un evento de integración para que el resto de los microservicios, en este caso el de gestión de solicitudes, pueda actuar en consecuencia y mantener la integridad de los datos.

## 9. DESPLIEGUE DE LA SOLUCIÓN

A la hora de hablar de las arquitecturas de microservicios se comentaba que una de sus grandes ventajas es la posibilidad de poder ser escalados independientemente en el momento en el que se requiera y que, uniendo microservicios y contenedores, se podía amplificar y explotar esta cualidad. Sin embargo, con lo aprendido hasta el momento de contenedores, no parece haber una forma clara de definir y trabajar con una serie de contenedores como un grupo. Para ello existen los **orquestadores**.

### 9.1 Orquestadores

El motor de Docker es capaz de manejar instancias únicas de una imagen en un único host o sistema anfitrión pero, a la hora de manejar múltiples contenedores desplegados en múltiples hosts para aplicaciones distribuidas más complejas no es tan eficiente.

En muchos de los casos se necesitará una plataforma que ejecutará automáticamente contenedores, los escalará con múltiples instancias por imagen, se encargará del acceso a recursos como la red y el almacenamiento.

Ésta es la labor que cumplen los orquestadores, permiten escalar aplicaciones a lo largo de múltiples hosts de Docker como si fuera uno único con independencia de la complejidad de la plataforma sobre la que se está ejecutando.

Existen diversos orquestadores en el mercado y, aunque son productos muy complejos de los que se podría hablar individualmente en un proyecto especialmente dedicado al tema, se va a hacer una introducción al elegido en este proyecto: *Kubernetes*.

#### 9.1.1 Kubernetes

*Kubernetes*[54] es una plataforma portable, extensible y de código libre para manejar cargas de trabajo y servicios desplegados en contenedores que facilita establecer su configuración y automatización de una forma declarativa. Es el orquestador más usado actualmente y se está considerando la solución de facto. Su ecosistema y adopción crece a pasos agigantados.

Fue liberado por *Google* en 2014 después de ser usado en la compañía contando con más de una década y media de experiencia ejecutando en producción sistemas de escala mundial. Además de la experiencia de *Google*, también cuenta con las mejores ideas y prácticas de la comunidad, aprovechando el ser de código abierto.

Como curiosidad, su nombre viene del griego, que significa timonel o piloto. Dado que es una palabra bastante larga se suele referir al producto como K8s. Esto es su abreviación siendo K y S la primera y última letra y 8 el número de letras que hay entre ellas “ubernete”.

##### 9.1.1.1 Infraestructura

Desde el punto de vista físico o de infraestructura, *Kubernetes* tiene dos elementos clave: el máster y los nodos.

El máster es el responsable de mantener el estado que se quiere del clúster, siendo el encargado de administrar y gestionarlo todo. Cuando se quiere interactuar con Kubernetes para administrarlo, usando por ejemplo el CLI llamado *kubectl*, se está interactuando con el máster. El máster ocupa normalmente un nodo (una máquina física o virtual dependiendo de la infraestructura) dedicado a sus funciones y puede estar replicado por disponibilidad y redundancia.

Los procesos que ejecuta el nodo máster son: **kube-apiserver**, **kube-controller-manager** y **kube-scheduler**.

Los nodos en un clúster (que anteriormente se denominaban *minions*) son las máquinas (físicas o virtuales) donde se ejecutan las aplicaciones (contenedores). El máster es el encargado de controlar cada nodo, es muy improbable que el administrador del sistema interactúe con ellos directamente.

Los procesos que ejecuta cada nodo son:

- **kubelet**: se comunica con el nodo máster.
- **kube-proxy**: un proxy que se encarga de gestionar la red en cada nodo para reflejar la configuración especificada.

#### 9.1.1.2 Pods

Kubernetes no ejecuta los contenedores directamente, en vez de eso crea un nivel de abstracción superior llamado *Pod* en el que puede ejecutar uno o más contenedores. Los Pods se consideran la unidad de computación más básica en Kubernetes.

Un Pod encapsula un contenedor (o, en algunos casos, múltiples), recursos de almacenamiento, una única dirección IP y opciones que determinan cómo el contenedor debería ejecutarse.

Si se necesitan más instancias de un Pod se puede configurar Kubernetes para desplegar nuevas réplicas. Lo normal es desplegar varias copias de este Pod para asegurar el balanceo y la disponibilidad ante errores.

#### 9.1.1.3 Deployments

Aunque los Pods son la unidad básica, normalmente no son lanzados directamente en un clúster. En vez de eso, normalmente, son gestionados en un nivel de abstracción superior denominado deployment (despliegue en castellano).

El propósito principal de un despliegue es especificar cuántas réplicas de un Pod deberían estar ejecutándose a la vez. Cuando se añade un despliegue al clúster, éste levantará automáticamente el número de Pods especificados y los monitorizará. Si un Pod muere se encargará de volver a levantarlo.

En los despliegues se definen diversos factores como la aplicación y componente al que pertenecen y los contenedores específicos que componen del despliegue. A la hora de especificar cada uno de ellos se define la imagen, las variables de entorno que se inyectarán, el registro en el que se encuentran las imágenes, los puertos, el número de réplicas y otros valores.



En Kubernetes la configuración se define en ficheros con extensión “.yaml”. Un ejemplo del deployment correspondiente al servicio de gestión de solicitudes, de una forma reducida, sería el siguiente:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: applicants
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: titan
        component: applicants
    spec:
      containers:
      - name: applicants
        image: cjaliagav.azurecr.io/applicants.api
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        env:
        - name: PATH_BASE
          value: /applicants-api
```

#### 9.1.1.4 Services

La mejor forma de entender los servicios es verlos del modo siguiente: cada Pod obtiene su propia dirección IP pero esa dirección IP no es una fuente de confianza dado que los Pods pueden ser creados y destruidos. Además, en las arquitecturas de microservicios, normalmente un grupo de Pods puede ofrecer funcionalidad a otro grupo, es decir, un grupo de ellos puede querer contactar al otro grupo. Por tanto surge una necesidad: ¿cómo es capaz de contactar un grupo con el otro sin usar las direcciones IP cambiantes?

Los servicios son la solución a esta necesidad. Son una abstracción que definen a un grupo lógico de Pods y la política para acceder a ellos.

Siguiendo con el caso de gestión de solicitudes, el siguiente ejemplo ilustra la definición del servicio:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: titan
    component: applicants
  name: applicants
```

```
spec:
  ports:
    - port: 80
  selector:
    app: titan
    component: applicants
```

En este caso se creará un servicio cuyo objetivo serán los Pods marcados con las etiquetas **app = titan** y **component = applicants**. Éstos han sido definidos con el despliegue que se ha visto en el apartado anterior.

El servicio expondrá el puerto 80 usando el protocolo TCP (por defecto al no indicar ningún otro) y se podrá acceder a él usando la ruta <http://applicants> (el nombre del servicio).

#### 9.1.1.5 Ingress

Con los conceptos definidos hasta ahora se pueden lanzar despliegues y crear servicios que se podrán comunicar internamente entre ellos. Sin embargo hay un tema pendiente: acceder a los servicios desde el exterior.

Ingress se refiere al acto de abrir un canal de comunicación desde el exterior hacia uno de los servicios que se estén ejecutando en el clúster. Hay diversas formas de hacerlo, como controladores de ingress y balanceadores de carga.

En este proyecto se ha optado por el uso del controlador. Antes de mostrar un ejemplo hay que tener en cuenta que, dado que debe ser accesible desde el exterior, hay factores a configurar que están fuera del ámbito de esta introducción a Kubernetes.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  labels:
    app: titan
    component: frontend
  name: titan-ingress
  annotations:
    ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
    - http:
        paths:
          - path: /applicants
            backend:
              serviceName: applicants
              servicePort: 80
```

### 9.1.2 Azure Kubernetes Service (AKS)

Kubernetes tiene integración con diversos proveedores Cloud como pueden ser Google Cloud [7], Amazon Web Services[6] o Microsoft Azure [5]. En este proyecto se ha optado por el uso de la nube de Microsoft Azure, en la cual Kubernetes está especialmente integrado gracias a AKS (Azure Kubernetes Service).

AKS facilita una forma de simplificar la creación, configuración y administración de un clúster de máquinas virtuales en Azure que están preconfiguradas para ejecutar aplicaciones en contenedores. Ofrece y se integra con las herramientas existentes para gestionar Kubernetes junto a las usadas para gestionar los recursos desplegados en Azure, haciendo de su administración algo muy llevadero.

A la hora de crearse simplemente hay que seleccionar el tamaño, el número de nodos o hosts y alguna configuración adicional más avanzada si se desea y AKS se encarga de todo lo demás.

Al ser un entorno manejado no hay que preocuparse por la actualización de la versión del clúster en general o del máster y de los nodos en particular, al ser capaz de hacerse desde el portal de Azure o desde las herramientas de línea de comandos a golpe de clic. Y lo que es aún más importante, es capaz de hacerlo sin afectar a las aplicaciones que se estén ejecutando al ser capaz de hacerlo de una forma organizada.

Para obtener más información sobre cómo se crea y se trabaja con este recurso lo mejor es referirse a la documentación oficial, la cual estará siempre actualizada y muy detallada, en lugar de repetir aquí el paso a paso: <https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes>.

## 9.2 Del código al orquestador: desplegando la solución

En el capítulo 4 se habló de la gestión del proyecto y se definió la herramienta que se ha usado para ello: Azure DevOps. En ese capítulo, en la sección de gestión del código y de tareas, se introdujo el concepto de pull request, y se comentó que Azure DevOps puede desencadenar acciones cuando se completa un pull request y se introducen sus cambios en la rama principal.

Es ahora el momento de ver cómo utilizar Azure DevOps para compilar la solución, ejecutar los test, publicar las imágenes de los contenedores en el registro y lanzar el despliegue en el orquestador.

### 9.2.1 Definir la compilación de varias imágenes a la vez con Docker-compose

*Docker compose* [55] permite definir y ejecutar aplicaciones multi contenedor. Al igual que se define en el fichero Dockerfile cómo se construye una imagen, el fichero de docker-compose permite definir una serie de imágenes junto a su fichero Dockerfile, y especificar parámetros como el nombre que tendrán, el tag, variables de entorno que se inyectarán en el contenedor (si es docker-compose el que lo ejecuta) o qué imágenes dependen de qué otras para ser construidos en orden.

A la hora de ejecutar comandos con docker-compose se pueden especificar varios ficheros, de tal forma que se defina uno base y otro para cada entorno, añadiendo o modificando características según corresponda.

Un ejemplo del fichero docker-compose.yml es el definido en esta solución. En este caso se usa para construir las imágenes por lo que es un fichero sencillo:

```
version: '3.4'
services:
  applicants.api:
    image: applicants.api
    build:
      context: .
      dockerfile: src/Services/Applicants/Applicants.API/Dockerfile

  projects.api:
    image: projects.api
    build:
      context: .
      dockerfile: src/Services/Projects/Projects.API/Dockerfile
```

### 9.2.2 Azure Pipelines: Definición de la Build

La parte de Azure DevOps que permite hacer compilaciones y despliegues es pipelines. Las definiciones se basan en la ejecución de tareas en un host que también puede especificarse (Linux o Windows).

Las definiciones de Build o de Release (despliegue) pueden ser especificadas usando un diseñador visual o a través de ficheros yaml. Llegados a este punto puede observarse lo populares que son dado que han sido usados con Kubernetes, docker-compose y ahora con las definiciones.

Para definir una compilación lo primero es especificar de dónde saldrá el código fuente que será compilado.

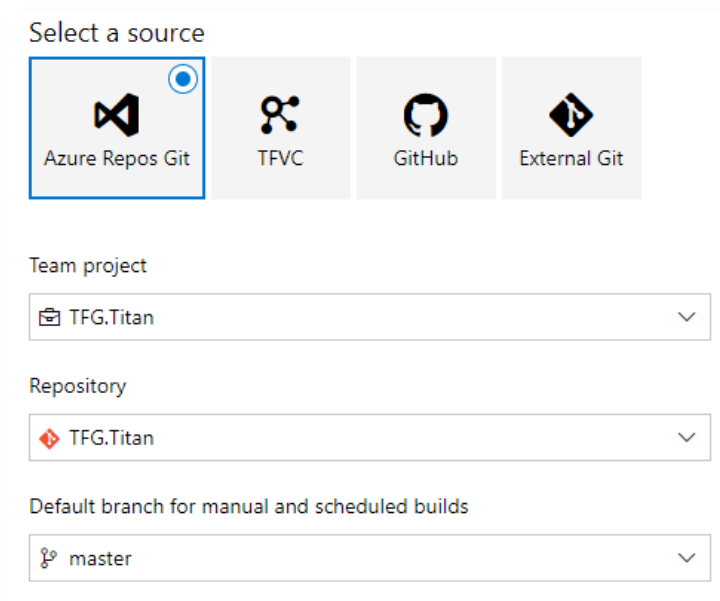
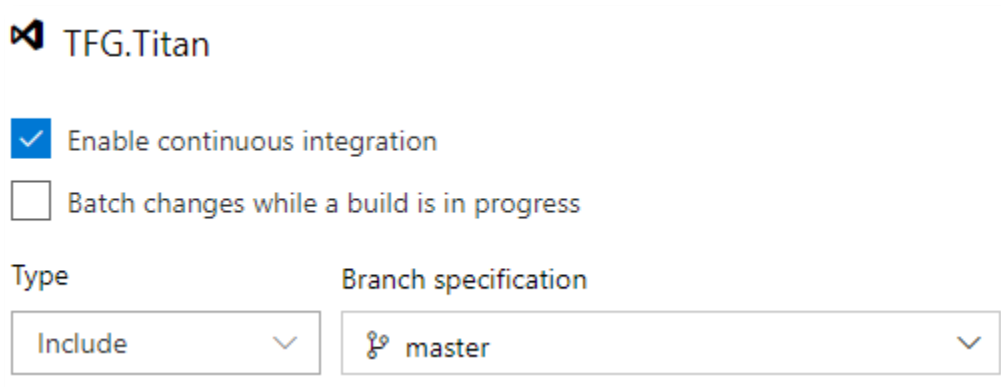


Figura 9-1. Selección de origen en Azure Pipelines

En este caso se ha seleccionado el proyecto realizado y el repositorio asociado. También se elige la rama sobre la que se lanzarán las compilaciones automáticas aunque también es posible programar y lanzar compilaciones sobre otras ramas.

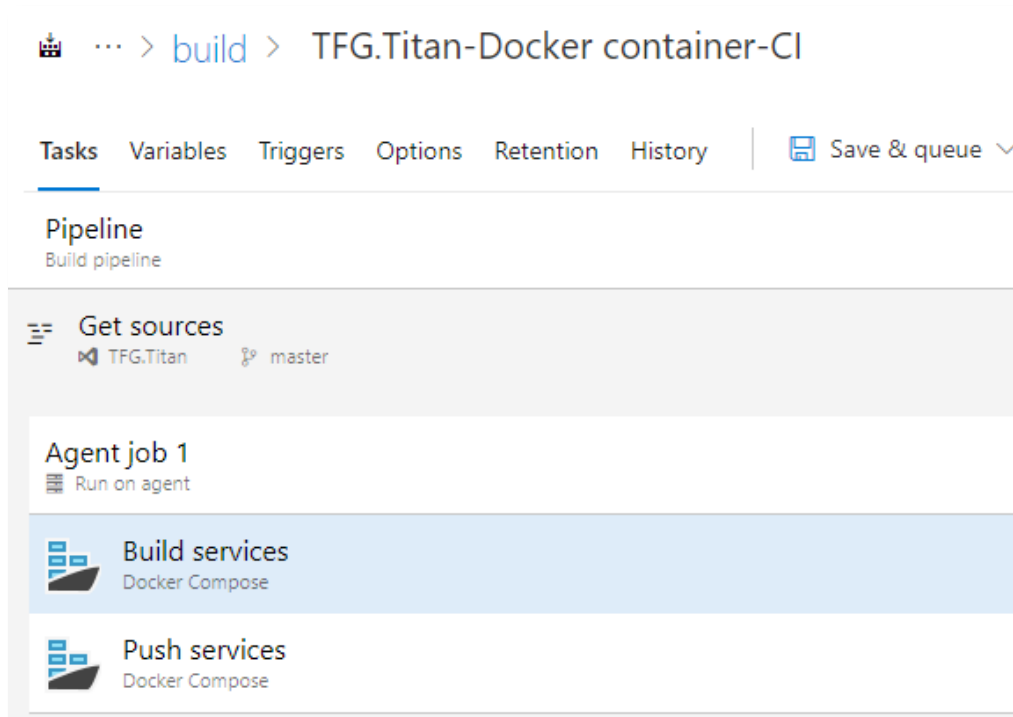


The screenshot shows the configuration for a build named 'TFG.Titan'. It features a checked checkbox for 'Enable continuous integration' and an unchecked checkbox for 'Batch changes while a build is in progress'. Below these, there are two dropdown menus: 'Type' set to 'Include' and 'Branch specification' set to 'master'.

*Figura 9-2. Activación integración continua en build de Azure Pipelines*

En este caso se habilitan cada vez que hay un nuevo commit en la rama máster (que puede venir de un pull request).

Para compilar la solución se necesitan dos tareas que harán uso de docker-compose. La primera será para compilar y generar las imágenes definidas en el fichero. La segunda será para subir estas imágenes, ya compiladas, al registro de imágenes elegido.



The screenshot displays the 'TFG.Titan-Docker container-CI' build definition. It includes tabs for 'Tasks', 'Variables', 'Triggers', 'Options', 'Retention', and 'History', with a 'Save & queue' button. The 'Pipeline' section is titled 'Build pipeline'. The 'Get sources' task is configured with 'TFG.Titan' and 'master'. Below this, 'Agent job 1' is defined with the task 'Run on agent'. The 'Build services' section lists 'Docker Compose', and the 'Push services' section also lists 'Docker Compose'.

*Figura 9-3. Definición tareas en build de Azure Pipelines*

En las tareas de docker-compose se pueden configurar diversos factores. En la siguiente captura se puede ver la definición de la primera de las tareas, aunque la segunda será exactamente igual pero simplemente cambiando la acción.

Dado que la solución será desplegada en AKS, el registro de contenedores privado que se ha elegido está también en la nube de Microsoft Azure: Azure Container Registry(ACR) [13]. Desde la propia tarea de docker-compose se puede seleccionar el tipo de registro, ACR en este caso. Puesto que ACR está en Azure, aparece un desplegable para seleccionar la suscripción en la que está disponible y posteriormente otro para seleccionar el recurso.

Es posible, entre otras cosas, cambiar los tags que se aplicarán a las imágenes y la forma en la que se definirán sus nombres.

Display name *	Build services
Container Registry Type * ⓘ	Azure Container Registry
Azure subscription ⓘ   <a href="#">Manage</a>	Corp
Azure Container Registry ⓘ	cjaliagav
Docker Compose File * ⓘ	**/docker-compose.yml
Project Name ⓘ	\$(Build.Repository.Name)
<input checked="" type="checkbox"/> Qualify Image Names ⓘ	
Action * ⓘ	Build service images

Figura 9-4. Configuración tarea docker-compose en build de Azure Pipelines

Para más información sobre las compilaciones de contenedores con Azure Pipelines el mejor recurso es la documentación [56].

### 9.2.3 Azure Pipelines: Definición de la Release

El formato de la release es muy similar al de la build, también se definen en base a tareas y se pueden lanzar, por ejemplo, como reacción a una compilación usando sus resultados.

Sin embargo ofrecen funcionalidades añadidas respecto a las compilaciones. Se pueden definir entornos en los que se irá desplegando y probando la nueva versión en base a las tareas definidas. Si la nueva versión cumple las expectativas se aprobará el paso al siguiente entorno. Si por el contrario no las cumple se podrá rechazar para evitar ponerla en producción.

En este caso, y por simplicidad, en vez de definir una release se añadirán tareas adicionales a la compilación anterior, que se encargarán de desplegar en AKS las nuevas versiones de las imágenes publicadas en el registro (ACR).

Dichas tareas se conectarán con AKS directamente y ejecutarán a través de la herramienta de línea de comandos de Kubernetes, kubectl, las definiciones de los despliegues, servicios y demás ficheros de configuración de la solución.

Para más información sobre las Releases de Azure DevOps: Pipelines está disponible la documentación en línea [57].

## 10. TESTING

A lo largo de esta memoria se ha ido profundizando en técnicas para integrar patrones, principios y buenas prácticas en el código mientras se desarrollan aplicaciones. Uno de los objetivos de estos patrones es conseguir un código apto para ser probado.

Existen diversas técnicas para ayudar a detectar fallos en el código, aportar información sobre los mismos y utilizar esta información para corregirlos.

- **Pruebas unitarias:** aseguran que los componentes individuales de la aplicación funcionan como se espera sin probar las dependencias de cada uno.
- **Pruebas de integración:** aseguran que los componentes interactúan como se espera contra artefactos externos como bases de datos. Se pueden testear API, interfaz o efectos secundarios como consecuencia de acciones como lectura/escritura en bases de datos, trazas y telemetría, etc.
- **Pruebas funcionales por cada microservicio:** aseguran que la aplicación funciona como se espera desde la perspectiva del usuario.
- **Pruebas de servicio:** aseguran que los casos de uso de principio a fin son comprobados, con pruebas que incluyen varios servicios al mismo tiempo. Para estas pruebas se necesita preparar un entorno completo.

### 10.1 Pruebas unitarias

Las pruebas unitarias implican el comprobar una parte de la aplicación aislándola de su infraestructura y dependencias. Gracias a la inyección de dependencias, inversión de control y especialmente al principio de inversión de dependencias[37], al depender de abstracciones será posible crear implementaciones vacías de las que se hablará más adelante.

Las pruebas unitarias se encargan de probar métodos específicos o fragmentos, comprobando si se comporta como debe en una ejecución real: si los resultados que devuelve son correctos o si controla las situaciones en las que no debería ejecutar y casos extremos respondiendo adecuadamente.

Es importante ir desarrollando las pruebas unitarias al mismo tiempo que se van desarrollando las funcionalidades. Al hacerlo de esta forma es posible anticiparse a situaciones inesperadas e ir programando la funcionalidad en base a las pruebas.

También ayudan a evitar regresiones en el código. Al ir escribiendo las pruebas a la vez que el código, se irán añadiendo más y más pruebas a la vez que estas irán evolucionando. Cuanto más código esté cubierto con pruebas más fácil será detectar situaciones en las que nuevo código afecte a funcionalidades existentes. Para ello simplemente se tendrá que ejecutar la batería de pruebas, que alertará si alguna de las pruebas falla sin llegar a experimentar el fallo en un entorno productivo.

Hay dos términos que serán usados a lo largo de las pruebas: *Mocks* y *Stubs*. En este proyecto en particular se han usado los *Mocks* pero es importante tener una idea general de ambos conceptos.



### 10.1.1 Mocks

Un *Mock* es un objeto que sustituye a la implementación real de otro, normalmente una clase. Imita el comportamiento del objeto original exponiendo los métodos y propiedades de dicho objeto y, permite controlar cómo se comporta definiendo las expectativas sobre su ejecución.

Son importantes para sustituir las dependencias de los componentes que se están probando o código externo que esté fuera del control del proyecto.

Para evitar tener que definir manualmente las implementaciones de los Mocks se ha usado una biblioteca de código abierto llamada **Moq** [58]. La biblioteca está disponible en NuGet.

### 10.1.2 Stubs

Los Stubs son porciones de código que sustituyen a código real. A diferencia de los Mocks, no será posible definir y comprobar expectativas. Normalmente un Stub declara un código que es una dependencia de la porción que se está probando y sustituye al código real, aislando de esa forma, esa parte del código de la prueba.

### 10.1.3 Implementando pruebas unitarias

Visual Studio ofrece una plantilla para proyectos de test para soluciones implementadas usando .NET Core con distintos frameworks de tests. En este proyecto se ha optado por usar *xUnit* [59], un framework de código abierto y multiplataforma.

El siguiente ejemplo muestra un test unitario en el microservicio de gestión de solicitudes, probando específicamente uno de los controladores que forma parte de la capa de aplicación. Lo primero es definir la clase que contendrá los test.

```
public class SubmissionsWebApiTest
{
    private readonly Mock<IMediator> _mediatorMock;

    public SubmissionsWebApiTest()
    {
        //Creando el Mock de IMediator
        _mediatorMock = new Mock<IMediator>();
    }
}
```

Dentro de esta clase se definen los métodos que contienen los test. Estos métodos tienen que estar marcados con el atributo *Fact* si es un hecho o *Theory* si puede recibir parámetros y estos cambian el comportamiento del test.

```

[Fact]

public async Task Create_submission_success()
{
    //Configurando el Mock para el test
    _mediatorMock.Setup(x => x.Send(It.IsAny<CreateSubmissionCommand>()))
        .Returns(Task.FromResult(true));

    //Instanciando y probando el método
    var controller = new SubmissionsController(_mediatorMock.Object);
    var result = await controller.CreateSubmission(GetCommand()) as OkResult;

    //Comprobando la condición esperada
    Assert.Equal(result.StatusCode, (int)System.Net.HttpStatusCode.OK);
}

```

## 10.2 Pruebas funcionales y de integración

Como se ha comentado, las pruebas de integración tienen propósitos y objetivos distintos a los de las pruebas unitarias. Sin embargo la forma en la que implementan ambos cuando se están probando los controladores de ASP.NET Core es muy similar.

A diferencia de los unitarios, los test funcionales y de integración frecuentemente involucran elementos de infraestructura, como bases de datos, sistema de ficheros o la red. Es por ello por lo que en estas pruebas no se crearán Mocks y se usarán las implementaciones originales.

ASP.NET Core incluye un host de test para ejecutar las aplicaciones y manejar peticiones HTTP sin usar la red. Está disponible en NuGet instalando el paquete *Microsoft.AspNetCore.TestHost*.

### 10.2.1 Implementación de pruebas funcionales y de integración

En el siguiente ejemplo se puede observar cómo se prueba el controlador de gestión de solicitudes al igual que en el test unitario. Esta vez sin embargo, se crea un servidor y se envía un objeto serializado como JSON en vez de instanciar directamente al controlador:

```

public class SubmissionScenarios
{
    private readonly TestServer _server;
    private readonly HttpClient _client;

    public SubmissionScenarios()
    {
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());

        _client = _server.CreateClient();
    }

    [Fact]
    public async Task Create_submission_bad_request_response()
    {
        var content = new StringContent(
            BuildSubmission(), Encoding.UTF8, "application/json");
        var response = await _client.PostAsync("api/v1/submissions", content);

        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    }

    string BuildSubmission()
    {
        var submission = new CreateSubmissionCommand("", null, -1, "", "");
        return JsonConvert.SerializeObject(submission);
    }
}

```

## 11. CONCLUSIONES

Llegados a este punto, podemos decir que se ha cumplido con el objetivo de explicar los contenedores, sus beneficios, en que están basados y cómo se trabaja con Docker. Posteriormente se ha visto cómo se gestionan los grupos de contenedores, gracias a los orquestadores, y la forma en la que las arquitecturas de microservicios se integran con todo lo anterior.

Se ha pasado por una explicación teórica de diversos patrones de código, que si bien pueden ser aplicados independientemente, en conjunto funcionan muy bien y consiguen un código que puede probar de una forma sencilla, algo que es muy importante. Como se suele decir, después de la teoría viene la práctica y por ello se ha desarrollado un ejemplo que ilustra de manera práctica todo lo expuesto en un marco teórico y los beneficios de la arquitectura. El proceso de aprendizaje ha ido en las dos direcciones por lo que el resultado, tanto en el código desarrollado como en lo escrito en la memoria, se ha ido retroalimentando.

Personalmente, este proyecto me ha aportado un mayor dominio de una arquitectura software, de la que tenía un ligero conocimiento, en este lenguaje y en otros muy distintos, y el desarrollo práctico me ha ayudado a entender mejor el porqué de su relevancia.

Me gustaría enfatizar el hecho de que prácticamente todas las piezas utilizadas son tecnologías multiplataforma y de código abierto, pues es una ideología en la que creo firmemente. Y es mucho más importante cuando hablamos de tecnologías desarrolladas por grandes compañías como Microsoft, que ha pasado de ser una compañía muy cerrada en la que todos los productos eran privativos, a una de las que están contribuyendo al software libre. Esto se puede ver con productos como .NET Core que desde un principio se desarrolló en GitHub de forma abierta y que es multiplataforma. De hecho, Microsoft es uno de los mayores contribuidores al núcleo de Linux actualmente.

### 11.1 Responsabilidad

Este proyecto se desarrolla en un marco teórico, por lo que no implica una ejecución real de un proyecto sino que proporciona patrones y guías para ayudar al desarrollo de aplicaciones. Las necesidades del proyecto han sido introducidas en los objetivos y la motivación. Partiendo de ellas se llega a las implicaciones. El proyecto permitirá mejorar la calidad de vida de las personas en varios ámbitos:

**De los desarrolladores y encargados de diseñar y programar las aplicaciones.** Al aplicar los patrones, prácticas y hacer uso de las arquitecturas expuestas en este trabajo se consigue código de calidad, reutilizable y mantenible, que hace que todo sea más fácil, productivo y eficiente.

**De las personas que expondrán los sistemas desarrollados haciendo uso de las prácticas tratadas en este trabajo.** El objetivo final es mejorar la calidad del servicio a los usuarios. Será posible acceder a sistemas siempre disponibles, que funcionan de una forma rápida y eficiente y que serán capaces de

evolucionar según las necesidades de la sociedad. Al replicar los datos geográficamente y desplegar las aplicaciones mundialmente se consigue que la latencia de acceso a los servicios sea la menor posible. Esto es especialmente importante para gente con acceso limitado a internet puesto que ayuda a mejorar su acceso a la información.

Uno de los grandes beneficios de la unión entre microservicios, contenedores y orquestadores es una mejor utilización de los recursos (máquinas físicas, electricidad...). Es posible adaptar y aprovechar al máximo los recursos físicos de los que se disponen para hacer un uso justo y eficiente de ellos. Si además las soluciones se despliegan en entornos Cloud la optimización es mejor pues se puede modificar el número de recursos disponibles.

Anteriormente, si se preveía que un servicio iba a tener un pico de carga durante un determinado momento, era necesario adquirir nuevos recursos para afrontar el trabajo. Una vez pasado este momento crítico, esos recursos quedarían infrutilizados ocasionando un dispendio, innecesario la mayor parte del tiempo. La arquitectura propuesta en este marco teórico soluciona este problema.

Además, los proveedores Cloud, como Microsoft Azure, diseñan los centros de datos para ser lo más eficientes posibles (haciendo uso por ejemplo de energías limpias y renovables), intentando reducir la huella ecológica en el planeta e investigando para mejorar aún más el impacto.

## 11.2 Coste y planificación

A la hora de desplegar la aplicación en la nube de Microsoft Azure se han usado los siguientes servicios. Este es el presupuesto base con el mínimo número de instancias necesarias, por mes.

Servicio	Descripción	Coste
Azure Kubernetes Service (AKS)	1 DS2 v2 (2 vCPU; 7 GB de RAM) nodos x 730 Horas; Pago por uso; 1 discos de SO administrados: P4	€85,71
Container Registry	Nivel Básico, 1 unidades x 30 días, 0 GB de ancho de banda	€4,21
Azure SQL Database	Base de datos única, modelo de compra DTU, nivel Estándar Tier, SO: 10 DTU, 250 GB de almacenamiento incluido por base de datos, 3 bases de datos x 730 Horas, 5, retención GB	€37,24
Service Bus	Nivel Estándar: 0, 1.000 conexiones asincrónicas, 0 agentes de escucha de Hybrid Connect + 0 uso por encima del límite por GB, 0 horas de retransmisión, 0 mensajes de retransmisión	€8,27
<b>Total mensual</b>		<b>€135,44</b>
<b>Total anual</b>		<b>€1.625,26</b>

*Figura 11-1. Costes mensuales de los servicios desplegados en la nube.*

El proyecto se ha desarrollado a lo largo de 5 sprint de duración mensual, con una media de 2 horas por día.

		horas	Precio/hora	TOTAL
COSTE DE MANO DE OBRA (coste directo)		300	15 €	4.500 €
COSTE DE SERVICIOS (coste directo)		Precio mes	Meses	TOTAL
Microsoft Azure		135,44 €	2	270,88 €
GASTOS GENERALES (costes indirectos)	15%	sobre CD		675,00 €
BENEFICIO INDUSTRIAL	6%	sobre CD+CI		310,50 €
SUBTOTAL PRESUPUESTO				5.485,50 €
IVA APLICABLE			21%	1.151,96 €
TOTAL PRESUPUESTO				6.637,46 €

*Figura 11-2. Presupuesto ejecución del proyecto*

## 12. LÍNEAS FUTURAS

En este TFG se ha abarcado el proceso de diseño de arquitecturas de microservicios que son desplegadas en contenedores, explicando para ello en qué se basan los contenedores y presentando distintos patrones y buenas prácticas orientadas a la mantenibilidad y crecimiento de la aplicación.

Por tanto, la primera manera de continuar este trabajo sería aprovechar las ventajas del desarrollo modularizado para añadir nuevas funcionalidades a los microservicios existentes e, incluso, añadir nuevos microservicios para exponer una solución más amplia aplicando nuevos patrones. Sugerencias a estos patrones podrían los agregadores para unificar varios microservicios y exponer un API única a través del uso de puertas de enlaces.

Otra manera podría ser el implementar clientes que utilicen los microservicios expuestos. En este trabajo los microservicios exponen API pero no se han desarrollado clientes o interfaces de usuario, a parte del servicio de identidad. Sería interesante indagar en la aplicación de buenas prácticas a la hora de consumir los microservicios.

Se han introducido los orquestadores y se ha hablado específicamente de Kubernetes, Sin embargo son productos muy potentes que ofrecen muchas más funcionalidades de las tratadas en este trabajo. Por ello, se podría continuar el trabajo investigando en detalle los orquestadores y sus posibilidades.

También se ha hablado de las posibilidades de las plataformas Cloud y se ha desplegado la aplicación en Microsoft Azure. Las posibilidades del Cloud son ilimitadas y las soluciones de microservicios pueden beneficiarse mucho de dichos entornos. Expandir lo tratado en este proyecto replicando las aplicaciones a lo largo de distintos centros de datos de todo el mundo, balancear el tráfico gracias a servicios como *Traffic Manager*[60] o *Azure Front Door*[61] o hablar de las diversas formas de sincronizar y distribuir los datos entre regiones es una continuación muy interesante.

## 13. REFERENCIAS

- [1] «Framework», *Wikipedia, la enciclopedia libre*. 30-oct-2018.
- [2] «Entorno de ejecución», *Wikipedia, la enciclopedia libre*. 07-ago-2018.
- [3] «Transferencia de Estado Representacional», *Wikipedia, la enciclopedia libre*. 22-nov-2018.
- [4] «Advanced Message Queuing Protocol», *Wikipedia, la enciclopedia libre*. 22-ago-2018.
- [5] «Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/>.
- [6] «Amazon Web Services», *Amazon Web Services, Inc.* [En línea]. Disponible en: <https://aws.amazon.com/es/>.
- [7] «Google Cloud», *Google Cloud*. [En línea]. Disponible en: <https://cloud.google.com/products/?hl=es>.
- [8] «Docker Community Edition for Windows - Docker Store». [En línea]. Disponible en: <https://store.docker.com/editions/community/docker-ce-desktop-windows>. [Accedido: 14-oct-2018].
- [9] «Get Docker CE for Ubuntu», *Docker Documentation*, 05-oct-2018. [En línea]. Disponible en: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>. [Accedido: 14-oct-2018].
- [10] «Azure DevOps Services | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/devops/>.
- [11] «Azure Boards | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/devops/boards/>.
- [12] «Docker Hub». [En línea]. Disponible en: <https://hub.docker.com/>.
- [13] «Azure Container Registry - Docker Registry | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/container-registry/>.
- [14] «Comando dotnet new: CLI de .NET Core». [En línea]. Disponible en: <https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-new>.
- [15] «Comando dotnet restore: CLI de .NET Core». [En línea]. Disponible en: <https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-restore>.
- [16] «Comando dotnet publish: CLI de .NET Core». [En línea]. Disponible en: <https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-publish>.
- [17] «docker build | Docker Documentation». [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/build/>.
- [18] «docker run», *Docker Documentation*. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/run/>.
- [19] «docker ps», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/ps/>.
- [20] «docker inspect», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/inspect/>.
- [21] «docker images», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/images/>.
- [22] «docker pull», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/pull/>.



- [23] «docker start», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/start/>.
- [24] «docker stop», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/stop/>.
- [25] «docker restart», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/restart/>.
- [26] «docker push», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/push/>.
- [27] «docker login», *Docker Documentation*, 24-ago-2018. [En línea]. Disponible en: <https://docs.docker.com/engine/reference/commandline/login/>.
- [28] «Iniciativa OPENAPI», *OpenAPI Initiative*. [En línea]. Disponible en: <https://www.openapis.org/about>.
- [29] «The Linux Foundation – Supporting Open Source Ecosystems», *The Linux Foundation*. [En línea]. Disponible en: <https://www.linuxfoundation.org/>.
- [30] «OpenAPI Specification | Swagger». [En línea]. Disponible en: <https://swagger.io/specification/>.
- [31] «SOLID», *Wikipedia*, 08-oct-2018. [En línea]. Disponible en: <https://en.wikipedia.org/wiki/SOLID>.
- [32] «API Management: establecimiento de puertas de enlace de API | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/api-management/>.
- [33] «Object-oriented programming», *Wikipedia*, 08-oct-2018. [En línea]. Disponible en: [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming).
- [34] «Propiedades (Guía de programación de C#)». [En línea]. Disponible en: <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties>.
- [35] «Mutator method», *Wikipedia*, 02-oct-2018. [En línea]. Disponible en: [https://en.wikipedia.org/wiki/Mutator\\_method](https://en.wikipedia.org/wiki/Mutator_method).
- [36] «Factory method pattern», *Wikipedia*, 06-oct-2018. [En línea]. Disponible en: [https://en.wikipedia.org/wiki/Factory\\_method\\_pattern](https://en.wikipedia.org/wiki/Factory_method_pattern).
- [37] «Dependency inversion principle», *Wikipedia*, 20-sep-2018. [En línea]. Disponible en: [https://en.wikipedia.org/wiki/Dependency\\_inversion\\_principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle).
- [38] «Autofac». [En línea]. Disponible en: <https://autofac.org/>.
- [39] «Unity». [En línea]. Disponible en: <https://github.com/unitycontainer/unity>.
- [40] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1 edition. Boston: Addison-Wesley Professional, 2003.
- [41] «bliki: AnemicDomainModel», *martinfowler.com*. [En línea]. Disponible en: <https://martinfowler.com/bliki/AnemicDomainModel.html>. [Accedido: 30-oct-2018].
- [42] «P of EAA: Separated Interface». [En línea]. Disponible en: <https://www.martinfowler.com/eaCatalog/separatedInterface.html>. [Accedido: 03-nov-2018].
- [43] «Open–closed principle», *Wikipedia*. 30-oct-2018.
- [44] «Messaging that just works — RabbitMQ». [En línea]. Disponible en: <https://www.rabbitmq.com/>.
- [45] «Azure Service Bus: servicio de mensajería en la nube | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/service-bus/>. [Accedido: 20-nov-2018].

- [46] J. Bogard, *Simple, unambitious mediator implementation in .NET: jbogard/MediatR*. 2018.
- [47] «Decorator (patrón de diseño)», *Wikipedia, la enciclopedia libre*. 03-nov-2018.
- [48] J. Bogard, «Behaviors in MediatR», 13-nov-2018. [En línea]. Disponible en: <https://github.com/jbogard/MediatR/wiki/Behaviors>.
- [49] «SQL Server 2016 Express LocalDB | Microsoft Docs». [En línea]. Disponible en: <https://docs.microsoft.com/es-es/sql/database-engine/configure-windows/sql-server-2016-express-localdb?view=sql-server-2017>. [Accedido: 02-dic-2018].
- [50] «Security token service», *Wikipedia*. 30-mar-2018.
- [51] «OAuth 2.0 — OAuth». [En línea]. Disponible en: <https://oauth.net/2/>. [Accedido: 03-dic-2018].
- [52] «Welcome to OpenID Connect – OpenID». [En línea]. Disponible en: <https://openid.net/connect/>. [Accedido: 03-dic-2018].
- [53] «IdentityServer». [En línea]. Disponible en: <https://identityserver.io/>. [Accedido: 03-dic-2018].
- [54] «Kubernetes Documentation». [En línea]. Disponible en: <https://kubernetes.io/docs/home/>.
- [55] «Docker Compose | Docker Documentation». [En línea]. Disponible en: <https://docs.docker.com/compose/>. [Accedido: 26-nov-2018].
- [56] «Build Docker apps - Azure Pipelines and TFS | Microsoft Docs». [En línea]. Disponible en: <https://docs.microsoft.com/en-us/azure/devops/pipelines/languages/docker?view=vsts&tabs=designer>. [Accedido: 27-nov-2018].
- [57] «Release pipelines in Azure Pipelines and TFS - Azure Pipelines & TFS | Microsoft Docs». [En línea]. Disponible en: <https://docs.microsoft.com/en-us/azure/devops/pipelines/release/?view=vsts>. [Accedido: 27-nov-2018].
- [58] *The most popular and friendly mocking framework for .NET: moq/moq*. Moq, 2018.
- [59] «Home > xUnit.net». [En línea]. Disponible en: <https://xunit.github.io/>.
- [60] «Azure Traffic Manager». [En línea]. Disponible en: <https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-overview>.
- [61] «Azure Front Door Service Documentation (Preview)». [En línea]. Disponible en: <https://docs.microsoft.com/en-us/azure/frontdoor/>.
- [62] «bliki: Seedwork», *martinfowler.com*. [En línea]. Disponible en: <https://martinfowler.com/bliki/Seedwork.html>. [Accedido: 01-nov-2018].
- [63] C. de la Torre, B. Wagner, y M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, 2017.

## 14. ANEXOS

### 14.1 Seedwork

A nivel de solución o a nivel de microservicio se puede encontrar una carpeta llamada *Seedwork*. Dicha carpeta contiene clases base, que podrán ser usadas a lo largo de los proyectos, para no tener código redundante, por ejemplo, en cada entidad del dominio. Aunque *seedwork* es un término popularizado por Martin Fowler [62], también se podrán encontrar carpetas con el nombre *Common* (*común en castellano*) con el mismo cometido.

#### 14.1.1 La case base Entity

La siguiente clase es una entidad base donde se implementa código que podrá ser usado de la misma manera por cualquier entidad del dominio, cómo el identificador de esta, los operadores de igualdad, una lista de eventos de dominio por entidad, etc.

```
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }

    private List<INotification> _domainEvents;
    public IReadOnlyCollection<INotification> DomainEvents => _domainEvents?.AsReadOnly();

    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }

    public void RemoveDomainEvent(INotification eventItem)
    {
        _domainEvents?.Remove(eventItem);
    }
}
```

```

public void ClearDomainEvents()
{
    _domainEvents?.Clear();
}

public bool IsTransient()
{
    return this.Id == default(Int32);
}

public override bool Equals(object obj)
{
    if (obj == null || !(obj is Entity))
        return false;

    if (object.ReferenceEquals(this, obj))
        return true;

    if (this.GetType() != obj.GetType())
        return false;

    Entity item = (Entity)obj;

    if (item.IsTransient() || this.IsTransient())
        return false;
    else
        return item.Id == this.Id;
}

public override int GetHashCode()
{
    if (!IsTransient())
    {
        if (!_requestedHashCode.HasValue)
            _requestedHashCode = this.Id.GetHashCode() ^ 31;

        return _requestedHashCode.Value;
    }
    else
        return base.GetHashCode();
}
}

```

```

public static bool operator ==(Entity left, Entity right)
{
    if (object.Equals(left, null))
        return (object.Equals(right, null)) ? true : false;
    else
        return left.Equals(right);
}

public static bool operator !=(Entity left, Entity right)
{
    return !(left == right);
}
}

```

### 14.1.2 La clase base Enumeration

Esta clase base ofrece elementos comunes para trabajar con los enumerados, como son las comparaciones y la conversión de valores.

```

public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration()
    { }

    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public override string ToString() => Name;

    public static IEnumerable<T> GetAll<T>() where T : Enumeration
    {
        var fields = typeof(T).GetFields(BindingFlags.Public | BindingFlags.Static |
BindingFlags.DeclaredOnly);

        return fields.Select(f => f.GetValue(null)).Cast<T>();
    }
}

```

```

public override bool Equals(object obj)
{
    var otherValue = obj as Enumeration;

    if (otherValue == null)
        return false;

    var typeMatches = GetType().Equals(obj.GetType());
    var valueMatches = Id.Equals(otherValue.Id);

    return typeMatches && valueMatches;
}

public override int GetHashCode() => Id.GetHashCode();

public static int AbsoluteDifference(Enumeration firstValue, Enumeration secondValue)
{
    var absoluteDifference = Math.Abs(firstValue.Id - secondValue.Id);
    return absoluteDifference;
}

public static T FromValue<T>(int value) where T : Enumeration
{
    var matchingItem = Parse<T, int>(value, "value", item => item.Id == value);
    return matchingItem;
}

public static T FromDisplayName<T>(string displayName) where T : Enumeration
{
    var matchingItem = Parse<T, string>(displayName, "display name", item => item.Name ==
displayName);
    return matchingItem;
}

private static T Parse<T, K>(K value, string description, Func<T, bool> predicate) where T
: Enumeration
{
    var matchingItem = GetAll<T>().FirstOrDefault(predicate);

    if (matchingItem == null)
        throw new InvalidOperationException($"'{value}' is not a valid {description} in
{typeof(T)}");
}

```

```
        return matchingItem;
    }

    public int CompareTo(object other) => Id.CompareTo(((Enumeration)other).Id);
}
```