UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un proxy inverso para una arquitectura de microservicios

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Alejandro Carrión Sanmartín

Tutor: Patricio Letelier Torres

Curso 2020-2021

Resumen

????

Palabras clave: ?????, ???

Resum

????

Paraules clau: ????, ?????????

Abstract

????

Key words: ?????, ?????

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

La automatización de los despliegues de programas, combinada o no con el uso de arquitecturas de microservicios, es una práctica en auge hoy en día en el mundo del desarrollo de software. Prácticas de DevOps tales como la entrega continua son cada vez más utilizadas con el fin de acortar tiempos en el ciclo de vida del desarrollo de sistemas y facilitar así su construcción, además de sistematizar los despliegues para poder llevarlos a cabo de manera sencilla.

Por otro lado, el uso de arquitecturas de microservicios ya está consolidado. Esta consiste en la construcción de servicios independientes, ejecutados en procesos diferentes, que se encargan de realizar funciones concretas y que trabajan de forma conjunta para lograr el objetivo u objetivos globales de la aplicación que constituyen. Los beneficios que otorga este enfoque frente a la aproximación tradicional monolítica son muchos y muy variados. Algunos de ellos son:

- **Uso de diferentes tecnologías**. Cada microservicio puede estar construido con una tecnología diferente y puede utilizar distintos mecanismos de persistencia.
- **Maniobrabilidad en los despliegues**. Ante cualquier cambio no es necesario desplegar la aplicación entera, solamente los microservicios implicados.
- **Tolerancia a fallos**. La posibilidad de desplegar la aplicación de forma que quede repartida en diferentes máquinas, incluso duplicando microservicios, otorga cierta capacidad para tolerar fallos.
- **Escalabilidad y mantenibilidad**. Los microservicios, y la separación funcional que otorgan, facilitan el escalado de las diferentes partes de la aplicación de manera independiente. Lo mismo sucede con el mantenimiento, pudiendo crear equipos especializados.

Se puede obtener más información acerca de los microservicios en el artículo de James Lewis y Martin Fowler titulado "*Microservices*" [1]. Para una lectura con más profundidad, el libro "*Building Microservices*" [2] de Sam Newman.

Por contra, este tipo de arquitecturas aumentan la complejidad del desarrollo en algunos aspectos concretos como pueden ser el versionado de los microservicios o la coordinación de las comunicaciones entre ellos. Es aquí donde entra este trabajo, pues está enfocado a paliar otra de sus desventajas: la exposición de múltiples puntos de entrada al *backend* formado por los microservicios. Para ello, se pretende desarrollar un *proxy* inverso que sea su puerta de acceso.

Para comprender correctamente que es un *proxy* inverso es conveniente ver su relación con su patrón hermano: el *proxy* de reenvío, o *proxy* a secas. Un *proxy* es un com-

ponente intermediario que se encarga de proteger una red cliente haciendo que estos clientes no tengan comunicación directa con los servidores a los que se conectan a través de Internet. Por otro lado, un *proxy* inverso hace la misma función pero protegiendo a un grupo de servidores, ocultándolos así de sus clientes. Ambos pueden coexistir, de hecho suelen hacerlo. Para ilustrar mejor esta diferencia, la figura 1.1 presenta un esquema conceptual de un *proxy* y otro de un *proxy* inverso.

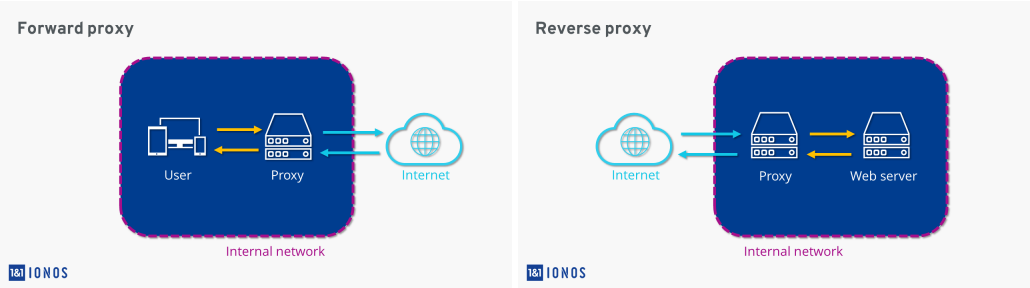

Figura 1.1: Esquema de un *proxy* frente a un *proxy* inverso. Imágenes de https://www.ionos.es/digitalguide/servidores/know-how/que-es-un-servidor-proxy-inverso.

Como se puede inferir de las definiciones anteriores, el uso de un *proxy* inverso no queda restringido a ciertas arquitecturas, pudiéndose utilizar para ocultar el servicio o servicios que consume cualquier aplicación. Sin embargo, este componente adquiere una gran importancia en el enfoque de microservicios, pues es importante no exponer estos al exterior. Además, se suele utilizar también para realizar tareas de balanceo de carga. La figura 1.2 muestra la comparativa de una arquitectura de microservicios básica y otra que utiliza el componente comentado. En la segunda se observa que con el uso de un *proxy* inverso los clientes no acceden directamente a los microservicios, ni siquiera los conocen.

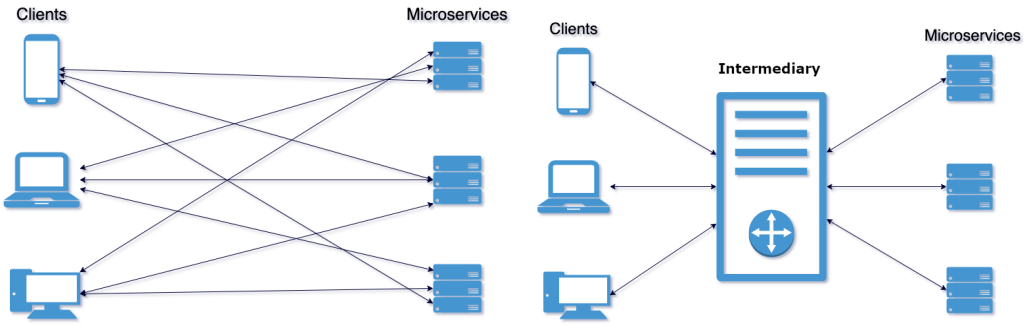

Figura 1.2: Arquitectura de microservicios básica frente a una con *proxy* inverso. Imágenes de https://www.adictosaltrabajo.com/2020/05/27/introduccion-al-api-gateway-pattern.

1.1 Motivación

La temática del trabajo es el desarrollo y despliegue de un *proxy* inverso que actúe de intermediario entre la interfaz de usuario de la nueva aplicación en desarrollo y sus microservicios. Se pretende construir este nuevo componente porque se cree necesario proteger el *backend* de la aplicación y ocultar los microservicios que lo forman. Por otro lado, también se quiere tener la posibilidad de lanzar a ejecución múltiples instancias de los microservicios con el fin de conseguir cierta tolerancia a fallos y poder también realizar balanceo de carga.

Este proyecto surge en el contexto de una práctica en empresa. El autor ha tenido la oportunidad de formar parte del equipo de I+D+i de una empresa de desarrollo de soft-

ware enfocada al sector sociosanitario, durante un periodo de tiempo de más de un año. Esta empresa comercializa un software de gestión geriátrica y actualmente está desarrollando la nueva versión de su producto, utilizando enfoques y tecnologías de vanguardia, entre ellos: arquitecturas de microservicios, desarrollo de software dirigido por modelos y generación automática de código.

La temática comentada fue elegida debido a la estrecha relación que guarda con el desempeño del autor en las prácticas mencionadas, esto es, contribuciones a un microservicio específico destinado a orquestar el despliegue de la aplicación. Por otro lado, el desarrollo a llevar a cabo le va a otorgar una visión más global de la aplicación sobre la que se trabaja, así como aumentar el nivel de conocimiento acerca de la misma, con la motivación de seguir contribuyendo al proyecto por mucho tiempo más. Por último, la tecnología a utilizar, .NET, es de su interés y aspira así a crecer como desarrollador de ese *framework*.

1.2 Objetivos

El objetivo principal de este trabajo es construir un *proxy* inverso. La construcción de este tiene las siguientes aspiraciones sobre la aplicación en construcción:

- **Ocultar los microservicios** que forman la aplicación para que la interfaz de usuario no acceda directamente a ellos por motivos de seguridad.
- Permitir **múltiples instancias** en ejecución de los microservicios que, a su vez, tiene por finalidad:
  - Conseguir que la aplicación sea **tolerante a fallos**, gracias a la posibilidad de tener un mismo microservicio desplegado en máquinas diferentes.
  - **Aumentar la eficiencia**, al poder crear o parar instancias dinámicamente según el tráfico que reciba la aplicación.

1.3 Estructura del documento

?????

CAPÍTULO 2
Estado del arte

En la actualidad existen en el mercado muchas aplicaciones y servicios que se pueden utilizar como *proxy* inverso. Algunas de estas soluciones son de pago, otras gratuitas e incluso algunas de código abierto. Se pueden dividir en dos tipos: productos *software* ya construidos y librerías. Los primeros suelen ser fáciles de configurar y se pueden poner en marcha de una manera muy rápida. Las segundas requieren una parte de programación pero se adaptan mejor a las necesidades particulares, pues permiten tener más control al usuario. A continuación se van a comentar dos herramientas de cada tipo, una de las cuales será la tecnología utilizada. También se mostrará un ejemplo básico de uso de esta.

Cabe destacar que no hay muchos servicios que se dediquen exclusivamente a ofrecer un *proxy* inverso. Estos suelen ofrecer otros productos como servidores web o balanceadores de carga. Por otro lado, lo habitual es hacer uso de un producto ya hecho, por lo que tampoco es fácil encontrar librerías que permitan personalizar un *proxy* inverso, menos aún si hay que tener en cuenta la tecnología que se utiliza. En relación con esto, las librerías elegidas son para el lenguaje de programación C#, utilizado en el resto de la aplicación de la que forma parte el *proxy* inverso.

2.1 API *Gateway*

A modo de aparte, es interesante mencionar un patrón parecido al *proxy* inverso y que también podría haber sido utilizado para resolver el problema que atañe a este trabajo: el API *Gateway*. Ambos comparten algunos casos de uso, por lo que sus diferencias causan confusión y no suelen quedar claras. Generalmente, se entiende que un API *Gateway* es una especialización de un *proxy* inverso, proporcionando así funcionalidades extra. Las más aceptadas e importantes son:

- Interpretan los mensajes que reciben y pueden hacer transformaciones sobre ellos; los *proxy* inversos solo los redirigen donde corresponda.
- Suelen ofrecer agregaciones de peticiones, esto es, aunar dos o más llamadas al *backend* y exponer esta composición a través de un único *endpoint*.
- Realizan tareas transversales a todos los *endpoints* como autenticación, autorización o monitorización.

El patrón recién comentado fue descartado desde el primer momento porque no se quería tener la funcionalidad de ninguno de los tres puntos clave que caracterizan esta aproximación. Por este motivo, y por la búsqueda de sencillez, era más coherente decantarse por un *proxy* inverso.

2.2 Productos hechos

2.2.1. NGINX

Originariamente NGINX [3] fue construido para ser un servidor web pero más tarde ofreció la posibilidad de actuar como *proxy* inverso, balanceador de carga o *proxy* para protocolos de correo electrónico. Desde la vertiente que interesa a este trabajo:

- Se trata de un *proxy* inverso ligero y de alto rendimiento.
- Ofrece una versión gratuita y otra de pago, NGINX Plus, la cual ofrece funcionalidades extra.
- Se configura a través de un fichero el cual puede ser recargado durante su ejecución, es decir, se puede configurar dinámicamente.

2.2.2. Apache HTTPD

Apache HTTPD [4]. Además de ser un servidor web "básico", y proveer contenido estático y dinámico a los usuarios finales, Apache HTTPD puede también actuar como proxy inverso

2.3 Librerías

2.3.1. YARP

YARP [5] es una librería hecha por el propio Microsoft y su objetivo es facilitar la creación de *proxies* inversos. Surgió dentro de la propia infraestructura de la empresa, en la que diferentes equipos preguntaban si existía algún *proxy* inverso disponible para utilizar en sus respectivos proyectos. La respuesta de la compañía fue crear un equipo para construir esta librería y así estandarizar su uso. Además, se decidió hacer público tanto su uso como su código, siendo así una opción *open source* a tener en cuenta.

Se encuentra todavía en desarrollo, habiendo sido lanzada su primera *release* el 25 de Junio de 2020, y, a fecha de este trabajo, todavía solo se puede utilizar una *preview*. Se prevé que vayan saliendo a la luz más versiones con más funcionalidades basadas en la experiencia de la propia empresa pero también en las opiniones de los usuarios externos. YARP es compatible con .NET Core 3.1 y .NET 5 pero algunas funcionalidades solo están disponibles para el segundo *framework*, ya que se trata de la generación siguiente al primero.

A raíz de la heterogeneidad de casos de uso que debe cubrir para satisfacer las necesidades de los diferentes equipos de la compañía, está diseñada para ser muy personalizable y flexible. Otro aspecto importante es que permite cambiar la configuración del *proxy* inverso de forma dinámica, lo que no obliga a tener que volver a lanzarlo a ejecución cuando se añada una nueva ruta, por ejemplo.

Permite construir *proxies* inversos de nivel 7. Esto hace referencia a la capa del Modelo OSI (figura 2.1) sobre la que actúan, la de aplicación en este caso. Gracias a esto, es posible modificar una petición HTTP antes de redirigirla, como por ejemplo sus *headers* o ruta de destino. Sin embargo, no se puede hacer lo mismo con su contenido, ni siquiera es interpretado para tardar el menor tiempo posible en redirigir las peticiones. Además, no

es un componente aparte sino que se integra con la *pipeline* de *middlewares* de .NET [6], haciendo que su eficiencia sea muy alta.

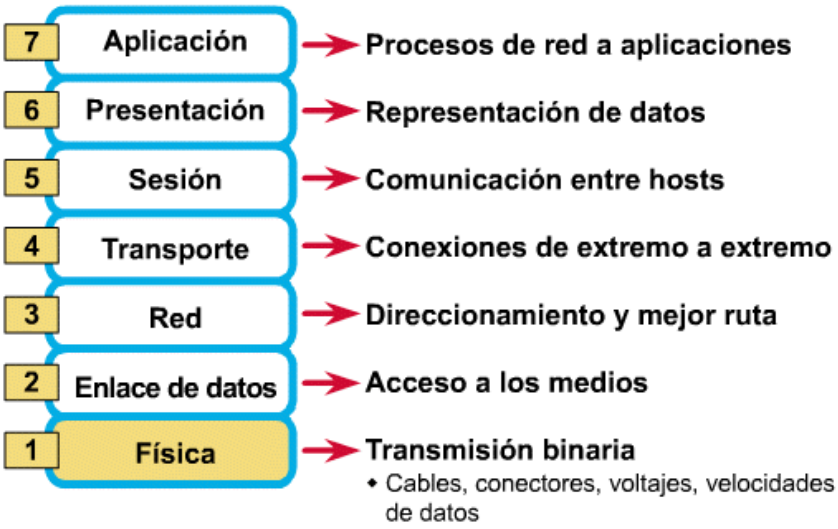

Figura 2.1: Modelo OSI.

Por último, la documentación propia no es corta pero tampoco excesiva. Asimismo, su corta edad hace que no se encuentren referencias o ejemplos de código de la comunidad fácilmente. Tampoco problemas planteados con sus posibles soluciones.

2.3.2. Ocelot

Ocelot [7] es una librería para .NET Core que permite a una aplicación de ese *framework* actuar como API *Gateway*. Posee las características siguientes:

- Está pensada para arquitecturas orientadas a servicios o a microservicios.
- Al tratarse de una librería, se utiliza de forma sencilla añadiéndola como un paquete NuGet más.
- Su configuración es muy básica, teniendo que especificarla en un fichero Json.
- No permite cambiar su configuración de manera dinámica.

2.4 Comparativa

Precio, API Gateway, Reverse proxy, documentación, previa configuración, cantidad de configuración, extensibilidad, recarga de configuración dinámica.

2.5 Tecnología utilizada

Primero de todo, la tecnología utilizada es .NET, más concreto, el lenguaje de programación C# y el entorno de desarrollo habitual para él, Visual Studio, en la versión del 2019. Como se comenta en la introducción de este capítulo, la aplicación en la cual se incluye el *proxy* inverso está construida con ese *framework*, por lo que se quiso mantener

ese aspecto también en el nuevo componente. Este aspecto corresponde al requisito no funcional RNF03, que se verá en el apartado 3.1.3.

Por otro lado, se ha decidido utilizar YARP para construir el *proxy* inverso. La aplicación sobre la que se trabaja posee un mecanismo de despliegue automático muy particular, por lo que la flexibilidad y capacidad de adaptación que ofrece esta librería son muy adecuadas para integrar el *proxy* inverso con dicho proceso. Si se hubiera utilizado un producto ya hecho, no se hubiera podido conseguir un nivel alto de cohesión y se hubieran tenido que abordar problemas de integración. Por otro lado, su sencillez y altas prestaciones hacen que destaque frente a Ocelot, la que, como ya se ha comentado, permite crear API *Gateways*.

Por último, se ha utilizado Postman [8] para realizar consultas de prueba al *proxy* inverso y comprobar su funcionamiento. Postman es una herramienta gratuita que permite hacer justo lo que se ha descrito. Se trata de un cliente para peticiones HTTP REST que se utiliza para probar de manera sencilla servicios web y así agilizar su desarrollo.

2.5.1. Ejemplo de uso básico

Una vez comentado la elección que finalmente se hizo y el porqué, se va a introducir un ejemplo de uso básico de YARP para dar una visión general de lo que es capaz de hacer esta librería. De este modo, también será más fácil comprender el funcionamiento del *proxy* inverso desarrollado.

En lineas generales, empezar a utilizar Yarp no es nada difícil, en un par de horas se puede llegar a tener un *proxy* inverso básico funcionando, si bien es más complejo utilizar características avanzadas o personalizarlo en función de las necesidades particulares.

La demostración que se va a realizar consiste en crear un *proxy* inverso que redirija peticiones hacía dos servicios web de prueba, creados anteriormente para el ejemplo. Estos simulan ser un servicio de localización y devuelven siempre "*Valencia, Spain*". También muestran este mismo mensaje por consola cuando son consultados.

Para empezar con el ejemplo, se crea un proyecto web vacío de .NET. Una vez el proyecto se ha creado, hay que añadir la referencia a la librería de YARP. En .NET las librerías se añaden como paquete NuGet [9] y para buscar qué paquetes hay disponibles e instalarlos se puede utilizar un asistente gráfico, figura 2.2.

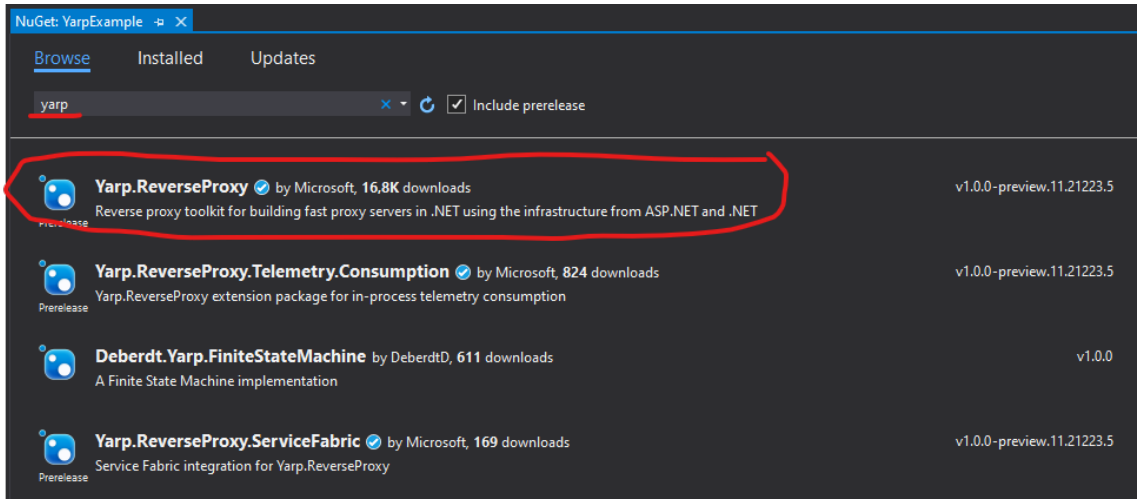

Figura 2.2: Instalación del NuGet de YARP.

Acto seguido, es necesario añadir un poco de código para poner en marcha un *proxy* inverso sencillo. En la clase *Startup* hay que modificar los métodos por defecto *ConfigureServices* y *Configure*. En el primero de ellos es necesario registrar el código del *proxy* inverso, haciendo *services.AddReverseProxy()*, y cargar la configuración de las rutas, *proxyBuilder.LoadFromConfig()*. En el segundo basta con asegurarse de que se haga la llamada *app.UseRouting()* y añadir *endpoints.MapReverseProxy()* dentro del *UseEndpoint*. Para visualizar mejor estos cambios, figura 2.3.

```
public class Startup
{
    2 references
    public IConfiguration Configuration { get; }

    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        IReverseProxyBuilder proxyBuilder = services.AddReverseProxy();

        proxyBuilder.LoadFromConfig(Configuration.GetSection("ReverseProxy"));
    }

    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapReverseProxy();
        });
    }
}
```

Figura 2.3: Clase *Startup* después de añadir el código del *proxy* inverso.

Por otro lado, hay que configurar las rutas que va a tener el *proxy* inverso, es decir, los enrutamientos que debe hacer en tiempo de ejecución. Para ello se necesita especificar dichas rutas en el archivo *appsettings.json*. Este archivo se utiliza para especificar cualquier tipo de configuración y se podría haber utilizado otro diferente. También se puede configurar a través de código pero requiere algo más de trabajo. Por este motivo, en esta demostración se va a utilizar el primer método, ya que se pretende mostrar un ejemplo lo más simple posible.

La figura 2.4 muestra las rutas creadas para esta demostración. Se ha creado un par ruta-*cluster* con dos destinaciones. Para no entrar en mucho detalle, lo que es importante es que la ruta tiene un patrón *{**catch-all}*, para capturar todas las peticiones, y el *cluster* contiene dos destinaciones, una por cada servicio de localización, comentados al principio de este apartado. Lo que se pretende conseguir con esta configuración es que el *proxy* inverso capture todas las peticiones y las envíe de manera aleatoria a cualquiera de los dos servicios de localización.

Lo siguiente es preparar los servicios para una prueba rápida. En la figura 2.5 se observa el escenario inicial. En la parte superior se encuentra el *proxy* inverso creado y, en la inferior, los dos servicios de localización . Importante destacar que los puertos de estos últimos coinciden con lo que se han indicado en el *appsettings.json*.

Para simular una petición de un cliente se va a realizar una consulta con Postman. Esta estará dirigida al *proxy* inverso y deberá ser redirigida a alguno de los dos servicios de localización, como se ha indicado en el *appsettings.json*. En la figura 2.6, se muestra el escenario anterior tras ejecutar la petición Postman repetidas veces. Se observa que

Figura 2.4: Fichero *appsettings.json* con la configuración de las rutas para el *proxy* inverso.

el *proxy* inverso la ha redirigido de manera aleatoria entre los dos otros servicios. En la consola del *proxy* inverso es interesante ver el mensaje por defecto que muestra YARP indicando hacia qué dirección redirige cada petición.

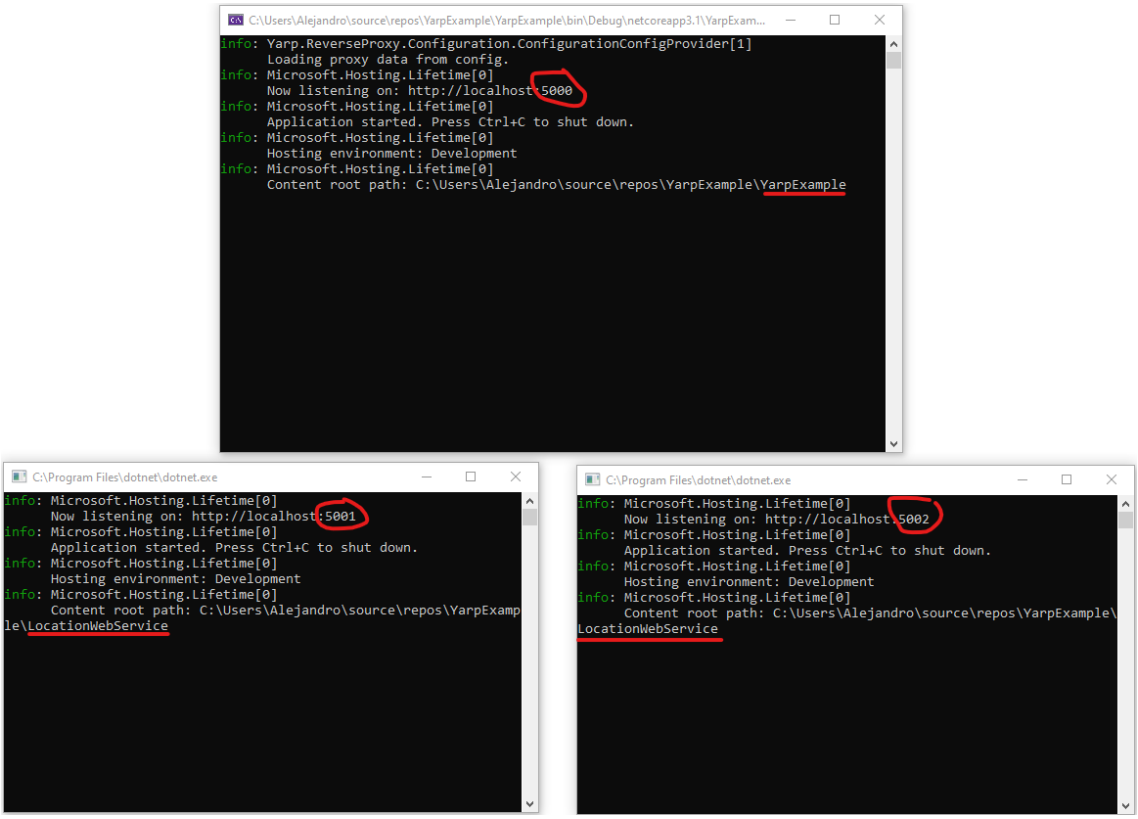

Figura 2.5: Escenario inicial de prueba de YARP con los servicios preparados.

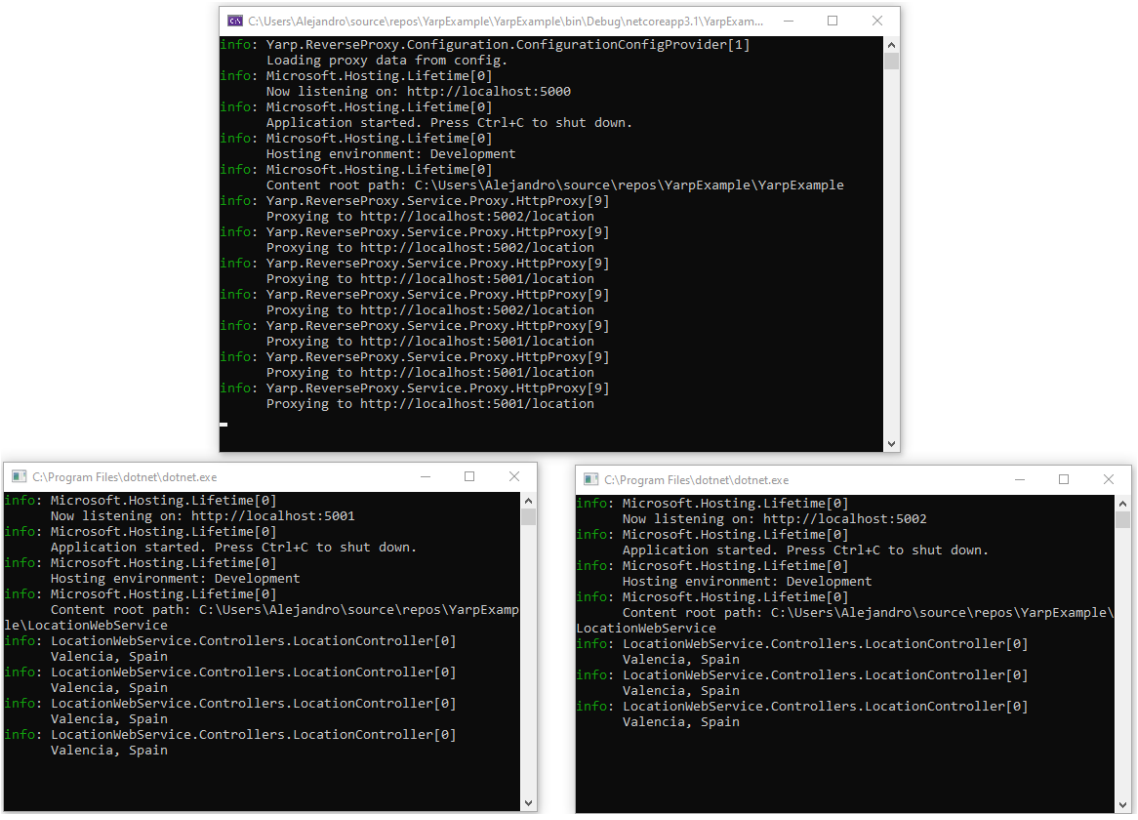

Figura 2.6: Escenario de prueba de YARP con los servicios después de realizar algunas consultas.

CAPÍTULO 3
Desarrollo de la solución

????? Este capítulo alberga el grueso del trabajo. En él..

3.1 Especificación de requisitos

La especificación de requisitos del *proxy* inverso se ha llevado a cabo elaborando los casos de uso que debe cubrir. Adicionalmente a estos, también se expondrán una serie de requisitos funcionales y no funcionales que el producto debe satisfacer.

3.1.1. Casos de uso

Los casos de uso van a ser expuestos a continuación con un diagrama de casos de uso, figura 3.1, y en formato de tabla, indicando para cada uno de ellos un identificador, un nombre, el actor que lo lleva a cabo y una breve descripción:

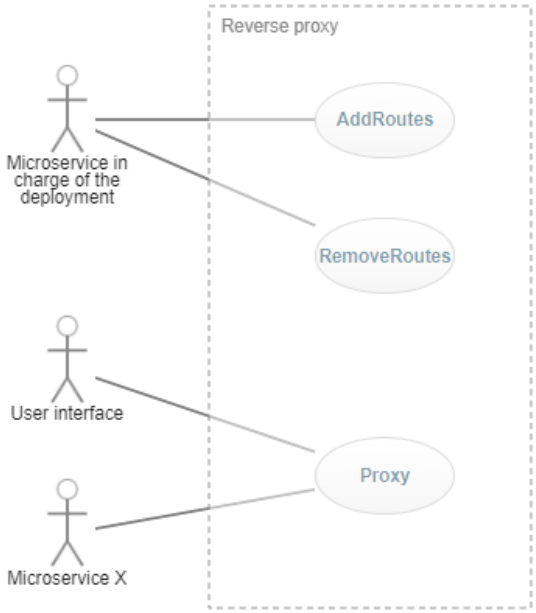

Figura 3.1: Diagrama de casos de uso del *proxy* inverso.

Identificador	CU01
Nombre	Añadir rutas
Actor	Microservicio encargado de orquestar el despliegue
Descripción	El microservicio que organiza los despliegues de la aplicación le indica al *proxy* inverso unas rutas para que las tenga en cuenta a la hora de redirigir peticiones, CU03. Las rutas son descritas mediante la ruta base de las peticiones que tiene que capturar y la dirección a la que redirigirlas.

Identificador	CU02
Nombre	Eliminar rutas
Actor	Microservicio encargado de orquestar el despliegue
Descripción	El microservicio que organiza los despliegues de la aplicación le indica al *proxy* inverso unas rutas para que las deje de tener en cuenta a la hora de redirigir peticiones, CU03. Las rutas son descritas mediante la ruta base de las peticiones que tiene que dejar de capturar y la dirección a la que no redirigir más peticiones.

Identificador	CU03
Nombre	Enrutamiento
Actor	Interfaz de usuario / Microservicio X
Descripción	La interfaz de usuario o cualquier microservicio de la aplicación envía una petición a la dirección en la que escucha el *proxy* inverso y esta es capturada y redirigida hacia un *endpoint* especificado por una determinada ruta. Las rutas determinan qué peticiones tienen que ser encaminadas a qué direcciones. Es necesario cargar previamente la ruta correspondiente mediante el caso de uso CU01.

3.1.2. Requisitos funcionales

El *proxy* inverso ha de cumplir una serie de requisitos funcionales, o características concretas, que no son propiamente casos de uso pero que se considera de interés mencionarlos, pues son relevantes para su funcionalidad. Los requisitos funcionales se centran en lo que debe hacer el sistema y dejan de lado la interacción con el usuario. Complementan así los casos de uso y ayudan también a comprender qué debe hacer exactamente el producto. Cabe destacar que estos aspectos se podrían haber presentado de forma conjunta a los casos de uso pero se cree que de esta manera quedan más claras, por un lado, las tres funcionalidades básicas y, por otro lado, algunos detalles más concretos.

Estos requisitos van a ser representados mediante un identificador, un nombre, el caso de uso con el que guardan relación, y una breve descripción:

Identificador	RF01
Nombre	Carga de rutas dinámica
Caso de uso relacionado	CU01
Descripción	La acción de añadir rutas a las que redirigir peticiones se debe poder hacer de manera dinámica, es decir, durante la propia ejecución. Esto permite no tener que parar y volver a lanzar el *proxy* inverso cada vez que se añaden o eliminan rutas. Esta es una característica bastante importante ya que el tiempo en el que se reinicia no estaría atendiendo peticiones y estas fallarían, quedando el sistema inaccesible durante ese tiempo.

Identificador	RF02
Nombre	Doble comunicación
Caso de uso relacionado	CU03
Descripción	El *proxy* inverso tiene que estar preparado para servir de puerta de entrada al *backend* desde la interfaz de usuario pero también debe interconectar los microservicios que forman dicho *backend*. De esta forma, todas las peticiones que se lleven a cabo desde fuera o dentro del propio sistema deben pasar por él.

Identificador	RF03
Nombre	Multiinstancia de microservicios
Caso de uso relacionado	CU01, CU02 y CU03
Descripción	Se debe soportar la ejecución simultánea de más de una instancia de un mismo microservicio, es decir, el *proxy* inverso debe ser capaz de permitir la existencia de más de una dirección que atienda un mismo grupo de peticiones.

Identificador	RF04
Nombre	Balanceo de carga
Caso de uso relacionado	CU03
Descripción	En relación con el RF03, las peticiones deben ser redirigidas de manera inteligente hacía las distintas direcciones posibles, si las hay para una determinada ruta. En concreto, se busca no inundar de peticiones unas y dejar en el olvido otras.

Identificador	RF05
Nombre	Versionado de microservicios
Caso de uso relacionado	CU01, CU02 y CU03
Descripción	Relacionado con el RF03, las diferentes instancias de un mismo microservicio pueden corresponder a versiones diferentes del mismo. De esta forma, las peticiones deberán ser redirigidas a una dirección u otra en función de la versión del microservicio que se quiera utilizar.

Identificador	RF06
Nombre	Multiinstancia de *proxies* inversos
Caso de uso relacionado	CU03
Descripción	Se debe poder trabajar con más de un *proxy* inverso a la vez de manera que uno enrute una petición hacía otro *proxy* inverso y este segundo la enrute hacía el microservicio final. Este paso por más de un enrutador debe ser transparente para ellos, siendo su única función la de redirigir las peticiones.

Identificador	RF07
Nombre	Instancias exclusivas
Caso de uso relacionado	CU03
Descripción	Derivado del RF06, debe ser posible desplegar un *proxy* inverso que solo se encargue de las peticiones que vienen de la interfaz de usuario y otro para las que van de un microservicio a otro, pudiéndose aumentar el número de estos.

3.1.3. Requisitos no funcionales

Un requisito no funcional se entiende como una restricción impuesta sobre un producto *software* que no corresponde a una funcionalidad del mismo. Están directamente relacionados con la calidad que tendrá el producto en cuestión y pueden referirse a características de diferentes tipos tales como fiabilidad o usabilidad. En concreto, la ISO/IEC 25010 [10], comúnmente llamada SQuaRE (*System and software Quality Requirements and Evaluation*), define ocho características principales y algunas subcaracterísticas específicas para cada una. La figura 3.2 las muestra todas.

Figura 3.2: Características y subcaracterísticas de calidad de un producto *software* definidas en la ISO/IEC 25010. Imagen de https://iso25000.com/index.php/normas-iso-25000/iso-25010.

Sobre el *proxy* inverso que atañe a este trabajo se imponen los siguientes requisitos no funcionales. Se detallan mediante un identificador, un nombre, la característica de la ISO a la que hacen referencia, una breve descripción y el motivo por el cual se considera necesario satisfacerlos:

Identificador	RNF01
Nombre	Autenticación de peticiones
Característica	Seguridad
Descripción	Se debe tener un sistema de autenticación que no permita escuchar ni redirigir peticiones sin autenticar. No se puede llevar a cabo ninguno de los tres casos de uso, (CU01, CU02 y CU03) sin una previa autenticación.
Motivación	Impedir que peticiones ajenas puedan ser atendidas y/o redirigidas para evitar posibles ataques.

Identificador	RNF02
Nombre	Enrutamiento eficaz
Característica	Eficiencia de desempeño
Descripción	El enrutamiento, CU03, no debe ralentizar las peticiones en exceso de forma que las peticiones enrutadas no tomen un tiempo superior al 115 % del tiempo que tardaría la petición si no pasara por el *proxy* inverso.
Motivación	Evitar que el *proxy* inverso suponga un retardo elevado en el tiempo de respuesta de las peticiones.

Identificador	RNF03
Nombre	Tecnología impuesta
Característica	Mantenibilidad
Descripción	La tecnología a utilizar para llevar a cabo el proyecto tiene que ser .NET, con C# como lenguaje de programación.
Motivación	Guardar coherencia con el resto de los microservicios para poder ser mantenido por personas que hayan trabajado con otros de ellos anteriormente.

Identificador	RNF04
Nombre	Estructura del proyecto
Característica	Mantenibilidad
Descripción	La estructura de carpetas y clases del proyecto debe ser similar a la de los demás microservicios, entendiéndose similar como aquella que pueda ser comprendida por una persona familiarizada con la estructura de referencia en un periodo de tiempo de 10 minutos como máximo.
Motivación	Ídem RNF03: guardar coherencia con el resto de los microservicios para poder ser mantenido por personas que hayan trabajado con otros de ellos anteriormente.

3.2 Diseño

En lo que se refiere al diseño de la solución, se va presentar su estructura final. Por otro lado, también se cree conveniente comentar los *middlewares* que forman la *pipeline* de YARP para ver qué componentes personalizados han sido desarrollados y añadidos.

3.2.1. Estructura de la solución

En el apartado 3.1.3, el requisito no funcional RNF04 impone que la estructura del *proxy* inverso debe ser similar a la del resto de microservicios. Por este motivo, la estructura del producto final sigue esa referencia, que pasa a detallarse a continuación.

Un microservicio cualquiera de esta aplicación, y el *proxy* inverso en particular, tiene una arquitectura de 7 capas:

- **Dominio**.
- **Contratos**.
- **Persistencia**.
- **Lógica**.
- **Aplicación**.
- **Servicios**.
- ***Proxy***.

3.2.2. *Pipeline* de YARP

Una *pipeline* se entiende como una sucesión de procesos que se ejecutan en cadena de manera que la salida de cada uno de ellos se conecta con la entrada del siguiente. Para gestionar las peticiones, .NET define una de ellas en el método *Configure* [6]. Está formada por componentes llamados *middlewares*, con un determinado propósito cada uno de ellos. Los *middlewares* disponibles cubren aspectos tales como páginas de excepciones especiales para el desarrollador, control de excepciones o seguridad de transporte.

Por su parte, YARP tiene una propia, integrada en la de .NET. Por defecto, sus *middlewares* se encargan de realizar sesión de afinidad para las peticiones, balanceo de carga, *health checks* pasivos y el enrutamiento final. Sin embargo, se puede personalizar añadiendo nuevos *middlewares* hechos desde cero según la necesidad del desarrollador. La figura (TODO:Añadir figura) muestra como ha quedado configurada esta última en el *proxy* inverso. En ella, se observa que se han utilizado los siguientes *middlewares*:

- ***CustomProxyLog***. Se encarga de personalizar los mensajes de *log* que muestra el *proxy* inverso por consola cuando enruta una petición.
- ***SameMachineFilter***. Su finalidad es filtrar las posibles destinaciones del enrutamiento de una petición de forma que si una de las direcciones corresponde a la misma máquina en la que se encuentra el *proxy* inverso, sea redirigida hacía ella. Esto puede suceder cuando el microservicio destino se encuentra desplegado en la misma máquina. De esta manera se ahorra tiempo evitando hacer una petición a otra dirección.

Por último, la figura (TODO:Añadir figura) muestra un esquema de los *middlewares* utilizados para ilustrar mejor el flujo de una petición al pasar por ellos.

3.3 Metodología

Para llevar a cabo este desarrollo no se ha seguido ninguna metodología concreta, ni tradicional ni ágil. No obstante, se ha optado por la vía ágil en el sentido de que se han seguido algunas de las prácticas propuestas por varias de estas metodologías, tales como Scrum o Kanban, con el fin de seguir un enfoque metodológico personalizado y adaptado a las características del proyecto. Las prácticas utilizadas se detallan a continuación:

- **Organización de las tareas en unidades de trabajo**. El trabajo a realizar para construir el *proxy* inverso se ha divido en tareas más pequeñas denominadas unidades de trabajo. Estas definen funcionalidades más pequeñas y concretas que los casos de uso y son más abstractas que las historias de usuario, ya que manifiestan de forma más clara la posibilidad de tratarse de tareas no solo relacionadas con cambiar el producto sino otras como crear la base de datos o renovar una suscripción de un servicio web.

- **Uso de un tablero *Kanban***. En relación con la anterior práctica ágil, las unidades de trabajo han sido gestionadas a través de un tablero *Kanban* con tres columnas: *TODO*, *DOING*, *DONE*. Cabe destacar que el tablero utilizado es muy simple, sin embargo, podía haberse tratado de uno más complejo si el desarrollo se hubiera llevado a cabo en equipo con más personas, por ejemplo. La herramienta utilizada a modo de tablero *Kanban* ha sido Trello [11].

- **Priorización de las unidades de trabajo**. Dentro de la propia columna *TODO* del tablero *Kanban*, las unidades de trabajo han estado ordenadas de mayor a menor prioridad en todo momento. Este almacén de trabajo priorizado a realizar se conoce como *backlog* y ha permitido conocer con facilidad las unidades de trabajo que se debían abordar al terminar otras.

- **Definición de pruebas de aceptación**. Se ha utilizado este tipo de pruebas para especificar cada una de las unidades de trabajo. Con ellas se establecen los límites de lo que se ha de considerar funcionamiento correcto o incorrecto. Además, algunas de las pruebas de aceptación han sido automatizadas, es decir, programadas dentro del propio proyecto, sistematizando y haciendo más fácil su ejecución. Se entrará más en detalle sobre las pruebas de aceptación definidas en el apartado 3.5.1.

- **Búsqueda de la sencillez y el minimalismo**. Se trata de no realizar esfuerzos innecesarios a la hora de implementar una determinada funcionalidad, esto es, buscar la solución más simple que cumpla los requisitos establecidos. De esta forma, se evita invertir tiempo en aspectos no definitivos o que se modificarán con posterioridad y que no afectan al funcionamiento inmediato del producto.

- **Afrontamiento y entrega de trabajo terminado de forma incremental**. Con el uso de unidades de trabajo se ha pretendido dividir y afrontar el trabajo a realizar mediante incrementos de funcionalidad, definidos precisamente por un conjunto de unidades de trabajo. Además, a medida que se ha terminado cada uno de estos incrementos, se ha entregado una nueva versión del producto. En este caso, las entregas del producto han correspondido con el despliegue de la determinada versión, quedando disponible para ser utilizada por el resto de la aplicación.

- **Ejecución de pruebas de regresión**. Han consistido en ejecutar las pruebas de aceptación de todas las unidades de trabajo terminadas hasta la fecha cada vez que un incremento considerable de funcionalidad era terminado. Con estas se consigue asegurar el correcto funcionamiento del producto antes de cada nuevo despliegue.

3.3.1. Plan de trabajo

Tras la exposición de la metodología seguida para este proyecto, se va a detallar el plan de trabajo trazado. La elaboración del tablero *Kanban* se ha tomado como punto de partida y se muestran sus unidades de trabajo iniciales en la figura 3.3, las cuales fueron obtenidas a partir de la especificación de requisitos del apartado 3.1. Cabe destacar que estas no han sido las definitivas y posteriormente se han creado nuevas, como es habitual en las metodologías ágiles.

Figura 3.3: *Backlog* con las unidades de trabajo iniciales.

Una vez elaborado y priorizado el *backlog*, el plan de trabajo ha sido hacer uso del tablero *Kanban* e ir desarrollando incrementos de funcionalidad poniendo en práctica las características ágiles comentadas en el punto anterior.

Por último, es interesante comentar el inicio particular de este desarrollo, pues se planificó la elaboración de dos prototipos de microservicio. Estos tendrían la misión de demostrar si el uso de código autogenerado a partir de modelos, como en el resto de microservicios, penaliza en exceso el rendimiento del *proxy* inverso. Esto podía deberse al tratarse de un código menos específico y más genérico, así como al poseer un exceso de características innecesarias para el caso en cuestión pero que sí son adecuadas para el resto de microservicios.

Así pues, la construcción de los prototipos culminaría en una medición de tiempos para evaluar la eficiencia de las dos soluciones y la consiguiente elección para continuar

con el desarrollo de uno u otro prototipo. Se comentarán más detalles de todo este proceso en el siguiente apartado.

3.4 Programación

La parte de programación del *proxy* inverso que protagoniza este trabajo ha pasado por distintas etapas o fases que se van a comentar a continuación. Estas etapas han sido creadas para facilitar la comprensión del desarrollo y las funcionalidades añadidas al producto. De esta forma, las agrupan para comentarlas poco a poco junto con los problemas o dificultades encontradas a la hora de implementarlas.

3.4.1. Construcción de prototipos

Como el plan de trabajo del punto 3.3.1 establece, el desarrollo comienza con la elaboración de dos prototipos de microservicio para evaluar la viabilidad de utilizar la estructura autogenerada de los demás microservicios sobre el *proxy* inverso. De esta forma, se construyó un microservicio sin utilizar la generación de código y otro utilizándola para medir posteriormente las diferencias de rendimiento entre uno y otro. Por otro lado, cabe destacar que a los *proxies* inversos que forman parte de una arquitectura de microservicios no se les suele dar tal categoría. Sin embargo, en este trabajo sí se le va a incluir dentro de los denominados microservicios debido a que se quiere que tenga la misma estructura, RNF04 del apartado 3.1.3. Al fin y al cabo es un tema simplemente de nomenclatura.

Microservicio a mano

El primero de los prototipos se hizo lo más simple posible, es decir, con la lógica justa y necesaria para desempeñar su trabajo y sin seguir ninguna arquitectura concreta. Esto último choca con el requisito no funcional comentado en el párrafo anterior pero se hizo para que las diferencias de rendimiento se evidenciaran más todavía, pues así se compararía un prototipo únicamente con el funcionamiento de *proxy* inverso y otro con también una estructura con multitud de clases que pudiera penalizar el rendimiento.

Para la creación de este prototipo, primero fue necesario crear un proyecto desde cero y ubicarlo en el lugar correspondiente del directorio de carpetas de la aplicación, junto a los demás microservicios. Acto seguido, se añadió el funcionamiento de *proxy* inverso dado por YARP (CU03 del apartado 3.1.1) y se le crearon rutas de manera predefinida para hacer las primeras pruebas con peticiones Postman. Las rutas predefinidas se configuran en el archivo *appsettings.json*. Estos dos últimos aspectos se comentan con más detalle en el apartado 2.5.1 y un ejemplo de configuración de rutas estáticas se puede ver en la figura 2.4.

Después de realizar las tareas comentadas queda un proyecto muy simple pero totalmente funcional. Se muestra la solución de Visual Studio correspondiente en la figura (TODO:Añadir figura).

Microservicio autogenerado

Respecto al segundo prototipo, fue construido utilizando las mismas herramientas de generación automática de código con las que fueron construidos el resto de microservicios de la aplicación. Cabe destacar que, para no realizar esfuerzos en vano (práctica ágil comentada en el apartado 3.3) sin saber qué prototipo iba a ser la opción elegida,

se dejaron algunos detalles correspondientes a la generación de código pendientes. Esto se debe a que el proceso de generación de código no se adaptaba al cien por cien a las necesidades del nuevo microservicio, lo que suele ser habitual al crear microservicios con características diferentes.

Así pues, se creó de cero otro proyecto, ahora utilizando la generación automática de código a partir de modelos. Para ello fue necesario modelar una entidad *Route* con los campos necesarios para identificar las rutas, esto es, la ruta base de las peticiones que se tiene que capturar y la dirección a la que redirigirlas. También se escribió el código correspondiente a YARP para crear el *proxy* inverso y se configuraron algunas rutas por defecto. Por último, se crearon algunas consultas Postman para probar este prototipo. Estas tres últimas acciones son las mismas que para el caso del microservicio manual, con algunas diferencias para adaptarse a la solución concreta.

El microservicio autogenerado resultante se muestra en la figura (TODO:Añadir figura). Es interesante comparar esta solución con la del prototipo hecho a mano para comprobar que la cantidad de clases y proyectos de Visual Studio es muchísimo mayor en este caso.

Comparativa

Una vez construidos los prototipos, se procedió a medir el impacto de seguir la norma para decidir qué prototipo desechar y con cuál seguir adelante. Para ello, se creó una aplicación de consola muy simple que se encargara de probar los dos microservicios de *proxy* inverso creados. Esta lanzaba peticiones sobre el *proxy* inverso que se le indicara para que este las redirigiera y calculaba el tiempo medio de estas.

Es importante destacar que las acciones de añadir y eliminar rutas dinámicamente, todavía por implementar, no son relevantes en esta comparativa ya que se llevan a cabo muy pocas veces en relación con el enrutamiento, es decir, no son las más comunes. Por este motivo, no se ha creído conveniente medirlas y han sido implementadas con posterioridad.

Por otro lado, es interesante comentar por qué desde un primer momento se enfocaba la comparativa como una prueba para ver si el microservicio autogenerado era viable y, por tanto, preferible sobre el otro. Uno de los motivos era que, si se utilizaba la generación de código, el *proxy* inverso se podría desplegar con el mismo procedimiento que estaba pensado para el resto de microservicios, lo que evitaría tener que crear un mecanismo especial para él y permitiría conservar esa homogeneidad para simplificar los despliegues. Otro motivo se trataba de la trivialidad para hacer cualquier cambio de estructura, pues la generación de código lo haría de forma automática. Por el contrario, habría que hacerlo manualmente en el microservicio a mano para continuar con el cumplimiento del requisito no funcional RNF04, del apartado 3.1.3.

Los resultados de las mediciones fueron claros: ambos prototipos tenían un rendimiento casi idéntico. El tiempo medio de respuesta de una petición en el caso del prototipo autogenerado solo era un 3 % más lenta que en el microservicio hecho a mano, lo que se consideró despreciable e insuficiente para contrarrestar las ventajas comentadas. De esta forma, se decidió eliminar el primer prototipo y continuar con la versión autogenerada.

3.4.2. Consolidación del microservicio autogenerado

Una vez elegida la opción de seguir el estándar de microservicio de la aplicación, hubo que resolver los aspectos de la generación de código que no quedaron resueltos.

Así pues, se tuvieron que modificar algunas plantillas de código a partir de las cuales se generan las clases y proyectos que forman los microservicios.

Un ejemplo de problemas con la generación de código fue el método *Configure* de la clase *Startup*. Este tenía que ser modificado para añadir la llamada *endpoint.MapReverseProxy()* pero la generación de código no lo permitía, no pudiéndose personalizar este método en función del microservicio en cuestión. De esta forma, la modificación de las plantillas consistió en generar una llamada a un método estático dentro de ese *Configure*, que se implementaría en una clase parcial. La figura (TODO: Añadir figura) muestra cómo queda la clase *Startup*.

Después de terminar los flecos de generación de código, era momento de seguir añadiendo funcionalidad. De este modo, se modelaron e implementaron las acciones de añadir y eliminar rutas dinámicamente, correspondientes a los casos de uso CU01 y CU02 del apartado 3.1.1. Fueron probadas con las correspondientes peticiones Postman.

Por otro lado, se añadió un mecanismo de autenticación de peticiones, tal y como demanda el requisito no funcional RNF01 de la sección 3.1.3. Este consistió simplemente en utilizar el mismo mecanismo de seguridad que se utiliza en el resto de microservicios, el cual utiliza el sistema de autenticación que ofrece .NET y se genera de manera automática para las acciones modeladas, como es el caso de las dos acciones comentadas en el párrafo anterior. Solo sería necesario indicar a YARP que utilizase dicho mecanismo también a la hora de enrutar peticiones.

También se creyó conveniente estandarizar el tratamiento de las rutas base de las peticiones que representan una ruta. Esto se hizo para hacer el *proxy* inverso más robusto y evitar posibles malentendidos con YARP, pues espera recibir las rutas de una manera concreta y no otra, por ejemplo, utilizando barras como separadores de ruta en lugar de barras invertidas. Para ello se creó un método *GetNormalizedPath* que normaliza una ruta dada según el esquema que YARP acepta. El método en cuestión se muestra en la figura (TODO: Añadir figura).

Además, se consideró oportuno crear una serie de pruebas unitarias para las acciones de añadir y eliminar rutas y para el método *GetNormalizedPath*, recién comentado en el párrafo anterior. Se entrará más en detalle acerca de estas pruebas en el apartado de pruebas 3.5.

Por último, se incluyó el *proxy* inverso en el núcleo de microservicios principales llamados Core. Estos son los que se despliegan en primer lugar ya que son utilizados por todos los demás, o al menos por la mayoría. Microservicios de este tipo son los que se encargan de aspectos clave tales como la seguridad, el despliegue de la aplicación o, como es el caso de este, la comunicación entre los microservicios. Se ahondará en este aspecto de los despliegues en el siguiente apartado.

3.4.3. Primeros despliegues

Hacer que la U apunte al Reverse proxy Crear rutas por defecto, a parte de añadirlas dinámicamente. Transición entre un sistema de despliegue y otro. Cuadrar despliegue en doctor: orden de despliegue Cuadrar nuestro despliegue: cargar el Reverse Proxy de rutas desde Deployment

Hacer que los microservicios apunten al Reverse proxy crear rutas por defecto: orden y rellenado de rutas desde deployment cuadrar ambos despliegues

A estas alturas del desarrollo ya se tenía una primera versión del *proxy* inverso perfectamente funcional pero antes de llevar a cabo los primeros despliegues había que adaptarlo al proceso de despliegue que se seguía hasta el momento. La interfaz de usuario de

la aplicación se conectaba directamente a los microservicios por lo que hizo falta interponer el nuevo componente entre ella y el *backend*.

Más tarde, se hizo que los propios microservicios utilizaran el *proxy* inverso para comunicarse entre ellos.

El único microservicio que no se pudo reemplazar fue el de autorización porque tenía una dirección específica, configurada a través del DNS.

Como paso previo a incluir el *proxy* inverso al proceso de despliegue primero se hicieron pruebas de manera local. Se desplegó el *proxy* inverso en una máquina a parte y se le cargó con rutas que apuntaban a los microservicios ya desplegados para que la interfaz de usuario lo utilizara. Se comprobó que todo funcionaba correctamente.

Un problema que surgió fue el de los timeouts. Dos microservicios tenían un timeout muy grande debido al elevado coste de sus peticiones. El *proxy* inverso cortaba las llamadas antes de que terminaran. Al final esto resultó culpa de *YARP*, que tiene un timeout por defecto de 100 segundos. La solución fue aumentar el timeout para las peticiones que se dirigen a los microservicios en cuestión. Para ello, fue necesario añadir un parámetro opcional para el timeout en la creación de rutas. De esta forma se permite la personalización del timeout en función de la ruta que se utilice.

Creación de identificadores inequívocos

3.4.4. Producto final

Multiinstancia de microservicios probar ejemplo estático adaptar carga dinámica crear tests crear peticiones Postman CustomProxyLog

Timeouts configurables problema con algunos microservicios. nombrar el apartado anterior, decir k se hizo provisionalmente k no se utilizara el reveerse proxy prueba a mano para todas las rutas hacerlos configurables, modificar acción de añadir rutas. modelar campo timeout problema, se aplica para todas las llamadas. se podría mejorar creando rutas exclusivas pero no vale la pena xk YARP corta...

segundo nivel de redirección varias opciones k se comentaron en la reunión cargar todos los Reverse Proxies de rutas SameMachineFilter

Para finalizar, se quiere mostrar un esquema (figura (TODO:Añadir figura)) del recorrido completo de una petición que viene de la interfaz de usuario y pasa por varios microservicios. De esta forma, se pretende ilustrar la función del *proxy* inverso y cómo el resto de componentes de la aplicación interactúa con él.

3.5 Pruebas

Para comprobar el correcto funcionamiento del *proxy* inverso se han realizado dos tipos de pruebas: unas más sencillas, hechas cada vez que se producía un incremento considerable de funcionalidad, y otras más formales, al final del proyecto.

Peticiones Postman a saco.

(Pruebas unitarias) tests proxy y añadir/eliminar/getNormalizedPath

3.5.1. Pruebas de aceptación

Algunas de las pruebas de aceptación más importantes se comentan a continuación.

3.5.2. Pruebas de regresión

Se trata de una serie de comprobaciones definidas formalmente que han servido para verificar que las funcionalidades anteriormente implementadas no dejaban de funcionar al añadir nuevas.

3.5.3. Pruebas finales

Con el propósito de garantizar la corrección del comportamiento del producto *software* desarrollado se llevaron a cabo pruebas finales. A las pruebas de regresión anteriormente comentadas se les añade... Pruebas de regresión más exhaustivas, examinando todos los detalles.

CAPÍTULO 4
Conclusiones

Decir si los objetivos se han cumplido. Valorar el cumplimiento de los casos de uso y de los requisitos.

En un futuro se puede querer extender la funcionalidad y construir encima un API *Gateway* que ofrezca agregaciones de peticiones o tareas transversales tales como monitorización de tiempos de las peticiones.

Bibliografía

[1] Artículo de J. Lewis y M. Fowler acerca de los microservicios. Disponible en:
https://martinfowler.com/articles/microservices.html

[2] Libro de referencia sobre microservicios:
Sam Newman, *Building Microservices.* ISBN: 9781491950357

[3] ¿Cómo utilizar NGINX como proxy inverso?:
https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy

[4] ¿Cómo utilizar Apache HTTPD como proxy inverso?:
https://httpd.apache.org/docs/trunk/es/howto/reverse_proxy.html

[5] Sitio oficial de YARP:
https://microsoft.github.io/reverse-proxy

[6] Documentación de Microsoft acerca del método *Configure* y de la *pipeline* de *middlewares*:
https://docs.microsoft.com/es-es/aspnet/core/fundamentals/startup?view=aspnetcore-5.0#the-configure-method

[7] Sitio oficial de Ocelot:
https://threemammals.com/ocelot

[8] Sitio oficial de Postman:
https://www.postman.com

[9] Documentación de Microsoft acerca de los paquetes NuGet:
https://docs.microsoft.com/es-es/nuget/quickstart/install-and-use-a-package-in-visual-studio

[10] ISO/IEC 25010 (SQuaRE):
https://iso25000.com/index.php/normas-iso-25000/iso-25010

[11] Sitio oficial de Trello:
https://trello.com