



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



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

El presente trabajo tiene por finalidad desarrollar un *proxy* inverso para una aplicación con arquitectura de microservicios. Dicha aplicación todavía se encuentra en fase de desarrollo y el *proxy* inverso va a ser un punto clave en su construcción. Más concretamente, será el encargado de ocultar el *backend* formado por los microservicios, siendo el intermediario entre la interfaz de usuario y los mencionados servicios, además de interconectar a estos. De esta forma, se va a convertir en el mediador de todas las comunicaciones de la aplicación y va a desacoplar así la ubicación de los microservicios. Además, también permitirá la coexistencia de múltiples instancias de estos para lograr cierta tolerancia a fallos y mejorar la eficiencia. Por otro lado, el *proxy* inverso adquiere una mayor importancia si se toma en consideración que es requerido por el mecanismo de despliegue automatizado con el que cuenta la aplicación de la que va a formar parte.

Para llevar a cabo este proyecto se ha seguido una metodología ágil adaptada al contexto del desarrollo. Una de las prácticas más destacables ha sido la realización de despliegues del *proxy* inverso de manera incremental. Estos han servido para verificar el correcto funcionamiento del producto *software* en su entorno real de actuación y han permitido descubrir oportunamente errores y mejoras a realizar.

La tecnología utilizada para el desarrollo del producto ha sido ASP.NET Core, con la finalidad de seguir el estándar del resto de microservicios de la aplicación. Para la función de *proxy* inverso se ha elegido una librería de Microsoft llamada YARP, apropiada para C#, el lenguaje de programación empleado.

Palabras clave: *Proxy* inverso, Microservicios, Automatización de despliegues, Metodologías ágiles, .NET, YARP, C#

Resum

El present treball té per finalitat desenvolupar un *proxy* invers per a una aplicació amb arquitectura de microserveis. Aquesta aplicació encara es troba en fase de desenvolupament i el *proxy* invers serà un punt clau en la seua construcció. Més concretament, serà l'encarregat d'ocultar el *backend* format pels microserveis, sent l'intermediari entre la interfície d'usuari i els nomenats serveis, a més d'interconnectar a aquests. D'aquesta manera, es convertirà en el mediador de totes les comunicacions de l'aplicació i desacoblarà així la ubicació dels microserveis. A més, també permetrà la coexistència de múltiples instàncies d'aquests per a aconseguir una certa tolerància a fallades i millorar l'eficiència. D'altra banda, el *proxy* invers adquireix una major importància si es pren en consideració que és requerit pel mecanisme de desplegament automatitzat amb el qual compta l'aplicació de la qual formarà part.

Per a dur a terme aquest projecte s'ha seguit una metodologia àgil adaptada al context del desenvolupament. Una de les pràctiques més destacables ha sigut la realització de desplegaments del *proxy* invers de manera incremental. Aquests han servit per a verificar el correcte funcionament del producte *software* en el seu entorn real d'actuació i han permès descobrir oportunament errors i millores a realitzar.

La tecnologia utilitzada per al desenvolupament del producte ha sigut ASP.NET Core, amb la finalitat de seguir l'estàndard de la resta de microserveis de l'aplicació. Per a la funció de *proxy* invers s'ha triat una llibreria de Microsoft anomenada YARP, apropiada per a C#, el llenguatge de programació emprat.

Paraules clau: *Proxy* invers, Microserveis, Automatització de desplegaments, Metodologies Àgils, .NET, YARP, C#

Abstract

The present work aims to develop a reverse proxy for a microservices architecture application. This application is still in development phase and the reverse proxy is going to be a key point in its construction. More specifically, it will be in charge of hiding the backend formed by the microservices, being the intermediary between the user interface and the aforementioned services, besides interconnecting these. In this way, it will become the mediator of all the communications of the application and will thus decouple the location of the microservices. In addition, it will also allow the coexistence of multiple instances of these to achieve some fault tolerance and improve efficiency. On the other hand, the reverse proxy acquires greater importance if it is taken into consideration that it is required by the mechanism of automated deployment with which the application of which it is going to be part has.

To carry out this project, an agile methodology adapted to the development context has been followed. One of the most notable practices has been to perform reverse proxy deployments incrementally. These have served to verify the correct operation of the software product in its real operating environment and have allowed timely discovery of errors and improvements to be made.

The technology used for the development of the product has been ASP.NET Core, in order to follow the standard of the rest of the application microservices. For function of reverse proxy, a Microsoft library called YARP has been chosen, appropriate for C#, the programming language used.

Key words: Reverse proxy, Microservices, Deployment automation, Agile methodologies, .NET, YARP, C#

Índice general

Índice general	V
Índice de figuras	VII

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	3
1.3	Estructura del documento	3
2	Estado del arte	5
2.1	Patrón API <i>Gateway</i>	5
2.2	Productos	6
2.2.1	NGINX	6
2.2.2	Kemp	7
2.3	Librerías	7
2.3.1	YARP	7
2.3.2	Ocelot	9
2.4	Comparativa	10
2.5	Conclusiones	10
3	Tecnología utilizada	13
3.1	Ejemplo de uso básico de YARP	13
4	Desarrollo de la solución	17
4.1	Especificación de requisitos	17
4.1.1	Casos de uso	17
4.1.2	Otros requisitos funcionales	18
4.1.3	Requisitos no funcionales	20
4.2	Diseño	21
4.2.1	Estructura de la solución	22
4.2.2	<i>Pipeline</i> de YARP	23
4.3	Programación	25
4.3.1	Construcción de prototipos	25
4.3.2	Consolidación del microservicio autogenerado	28
4.3.3	Primeros despliegues	30
4.3.4	Producto final	33
4.4	Pruebas	37
4.4.1	Pruebas de aceptación	38
4.4.2	Pruebas de regresión	39
4.5	Metodología	40
4.5.1	Plan de trabajo	42
4.5.2	Cronología del proyecto	42
5	Conclusiones y trabajo futuro	47
	Referencias	49

Apéndice

A	Proceso de generación automática de código	51
A.1	Modelado	52
A.2	Generación de código	53
A.3	Programación de particularidades	53

Índice de figuras

1.1	Esquema de un <i>proxy</i> frente a un <i>proxy</i> inverso.	2
1.2	Arquitectura de microservicios básica frente a una con <i>proxy</i> inverso. . . .	2
2.1	Productos NGINX.	6
2.2	Ejemplo de configuración de NGINX.	7
2.3	Modelo OSI.	8
2.4	Ejemplo básico de configuración de Ocelot.	9
2.5	Ejemplo de configuración de Ocelot con una agregación de peticiones. . .	11
3.1	Instalación del NuGet de YARP.	14
3.2	Clase <i>Startup</i> después de añadir el código del <i>proxy</i> inverso.	14
3.3	Fichero <i>appsettings.json</i> con la configuración de las rutas para el <i>proxy</i> inverso.	15
3.4	Escenario inicial de prueba de YARP con los servicios preparados.	16
3.5	Escenario de prueba de YARP con los servicios después de realizar algunas consultas.	16
4.1	Diagrama de casos de uso del <i>proxy</i> inverso.	18
4.2	Características y subcaracterísticas de calidad de un producto <i>software</i> definidas en la ISO/IEC 25010.	20
4.3	Esquema de las capas de la arquitectura del <i>proxy</i> inverso.	23
4.4	Proyecto de Visual Studio de la capa de lógica.	24
4.5	Proyecto de Visual Studio de pruebas para la capa de lógica.	25
4.6	<i>Pipeline</i> de YARP configurada para el <i>proxy</i> inverso.	25
4.7	Solución de Visual Studio del prototipo de <i>proxy</i> inverso hecho a mano. . .	27
4.8	Solución de Visual Studio del prototipo de <i>proxy</i> inverso hecho con generación de código.	28
4.9	Código principal de la aplicación de consola para medir los tiempos del enrutamiento de los dos prototipos.	29
4.10	Clase parcial <i>Startup</i> del <i>proxy</i> inverso.	30
4.11	Método <i>GetYarpNormalizedPath</i>	31
4.12	Fichero <i>defaultRoutes.json</i>	32
4.13	Mensajes de <i>log</i> por defecto de YARP y personalizados.	33
4.14	Métodos encargados de crear los identificadores de las entidades de YARP.	34
4.15	Esquema de dos niveles de <i>proxy</i> inverso.	35
4.16	Esquema de dos niveles de <i>proxy</i> inverso.	37
4.17	Esquema de interacción entre el <i>proxy</i> inverso y el resto de componentes de la aplicación al procesar una petición de la interfaz de usuario.	38
4.18	Pruebas de aceptación de la unidad de trabajo "Añadir rutas dinámicamente".	39
4.19	Pruebas de aceptación de la unidad de trabajo "Normalizar rutas base peticiones".	40
4.20	Pruebas de aceptación de la unidad de trabajo "Enrutamiento".	41
4.21	Pruebas de aceptación de la unidad de trabajo "Personalización timeouts".	42

4.22 Pruebas de aceptación de la unidad de trabajo "Multiinstancia de micro-servicios".	43
4.23 Prueba de aceptación automatizada de la unidad de trabajo "Añadir rutas dinámicamente".	43
4.24 Pruebas de aceptación automatizadas de la unidad de trabajo "Normalizar rutas base peticiones".	44
4.25 Pruebas de aceptación automatizadas de la unidad de trabajo "Enrutamiento".	44
4.26 <i>Backlog</i> con las unidades de trabajo iniciales.	45
A.1 Acciones ad hoc del <i>proxy</i> inverso.	52
A.2 Encabezado de código autogenerado.	53
A.3 Ejemplo de clase parcial del <i>proxy</i> inverso.	54

CAPÍTULO 1

Introducción

1.1 Motivación

La automatización de los despliegues de programas es una práctica en auge hoy en día en el mundo del desarrollo de software. Prácticas de DevOps [1] tales como la entrega continua [2] 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 [3] [4] ya está consolidado. Esto 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, incluso, 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, así como la de tener más de una instancia del mismo microservicio, 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, pudiéndose crear equipos especializados.

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. A raíz de otra de sus desventajas surge este trabajo: la exposición de múltiples puntos de entrada al *backend* formado por los microservicios. Para ello, se desarrollará un *proxy* inverso que será su puerta de acceso y actuará de intermediario entre este y la interfaz de usuario, además de interconectar los propios microservicios y desacoplar su ubicación. Por otro lado, también permitirá lanzar a ejecución múltiples instancias de estos con el fin de conseguir cierta tolerancia a fallos.

Para comprender correctamente qué es un *proxy* inverso es conveniente ver su relación con su patrón hermano: el *proxy* de reenvío, o simplemente *proxy*. Un *proxy* es un

componente intermediario que se encarga de proteger una red cliente haciendo que estos consumidores no tengan comunicación directa con los servidores a los que se conectan a través de la red. 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.

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 entre diferentes instancias de un mismo microservicio. La figura 1.2 muestra la comparativa de una arquitectura de microservicios básica y otra que utiliza un *proxy* inverso. En la segunda se observa que, con el uso de un componente de este tipo, los clientes no acceden directamente a los microservicios, ni siquiera los conocen.

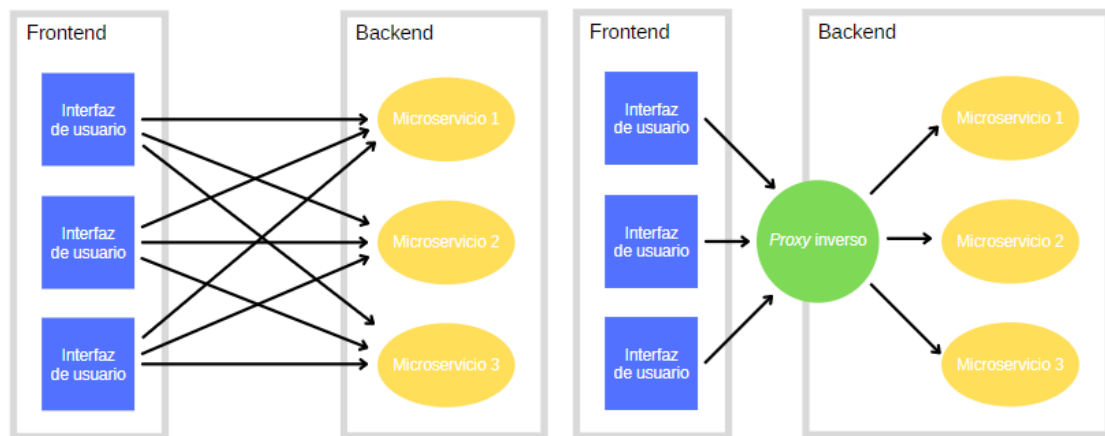


Figura 1.2: Arquitectura de microservicios básica frente a una con *proxy* inverso.

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 software enfocada al sector sociosanitario, durante un periodo de tiempo de más de un año. Esta empresa comercializa un software de gestión de residencias 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 de una forma

automatizada. 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 para una aplicación con arquitectura de microservicios. Su elaboración tiene las siguientes aspiraciones sobre el producto en desarrollo:

- **Ocultar los microservicios** que forman el *backend* de la aplicación para que la interfaz de usuario no acceda directamente a ellos por motivos de seguridad.
- Permitir la ejecución simultánea de **múltiples instancias** de los microservicios, que será fruto de:
 - **Desacoplar los microservicios** entre ellos y evitar que cada uno tenga que conocer donde se encuentra el resto.

y, 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

Este documento se divide en 5 grandes bloques o capítulos:

- El primero de ellos es este y en él se introduce el trabajo realizado a través de la motivación del mismo, los objetivos a cumplir y esta presentación de la estructura de la memoria.
- En el capítulo 2 se expone un breve estado del arte para revisar las soluciones posibles al problema que implica a este trabajo, así como una comparativa de dichas propuestas.
- El capítulo número 3 comenta la tecnología que finalmente ha sido utilizada, justificando su elección y mostrando un ejemplo básico de su uso.
- El cuarto capítulo es el más extenso y comenta el proceso que se ha seguido para desarrollar el *proxy* inverso. Incluye la especificación de requisitos, el diseño de la solución, la metodología seguida, las funcionalidades implementadas en un apartado dedicado a la programación y las pruebas realizadas.
- Por último, el capítulo 5 hace una reflexión sobre el trabajo realizado y concluye la misma valorando el grado de cumplimiento de los objetivos planteados. También presenta el trabajo futuro a realizar.

Como recurso adicional, se presenta un apéndice que explica el proceso de generación de código a partir de modelos que se ha utilizado para construir el *proxy* inverso, en concreto, y los microservicios de la aplicación de la que forma parte, en general.

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 contruidos 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 escogida para desarrollar el *proxy* inverso. Se hablará de la tecnología utilizada en general y de su elección en el siguiente capítulo (3), donde también se mostrará un ejemplo básico de su uso.

Cabe destacar que no hay muchos productos que se dediquen exclusivamente a funcionar como *proxy* inverso. Estos suelen ofrecer otros servicios 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 quiere utilizar. En relación con esto, las librerías elegidas son para ASP.NET Core [5], tecnología empleada en el resto de la aplicación de la que forma parte el *proxy* inverso.

2.1 Patrón 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 concierne a este trabajo: el API Gateway [6], o puerta de enlace de API. 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:

- 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.
- Interpretan los mensajes que reciben y pueden hacer transformaciones sobre ellos, mientras que los *proxy* inversos solo los redirigen donde corresponda.

2.2 Productos

2.2.1. NGINX

Originariamente NGINX ¹ fue construido para ser un servidor web pero más tarde ofreció la posibilidad de actuar como *proxy* inverso, balanceador de carga o incluso *proxy* para protocolos de correo electrónico, entre otros. De este modo, NGINX brinda muchos productos, con un conjunto de servicios cada uno. Se muestran los principales en la figura 2.1. Los dos más utilizados son NGINX Open Source y NGINX Plus. El primero de ellos ofrece un servidor web *open source* mientras que el segundo es de pago y permite utilizar un *API Gateway* o un *proxy* inverso.

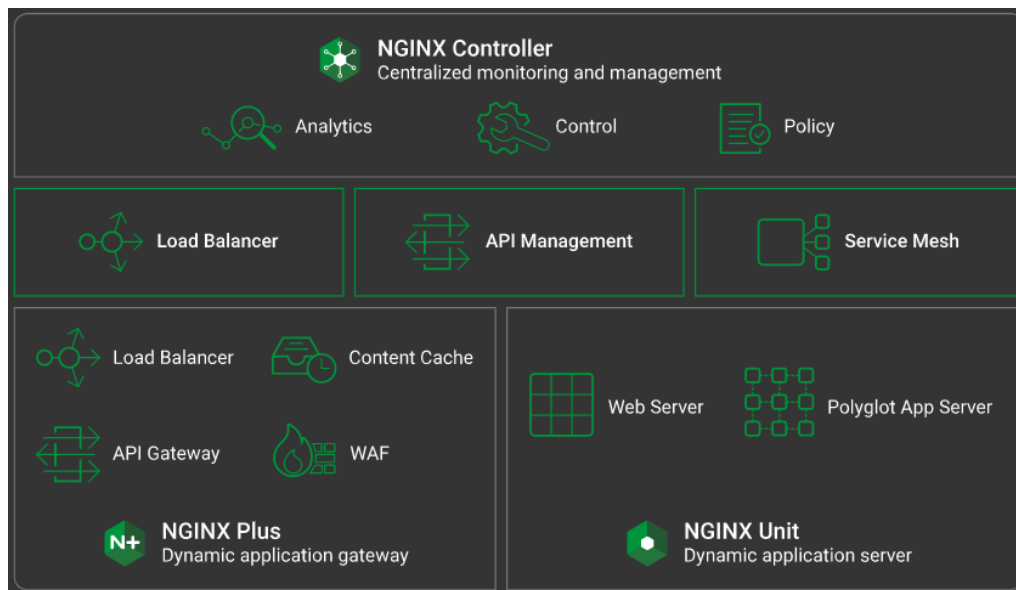


Figura 2.1: Productos NGINX [7].

Desde la vertiente que interesa a este trabajo, se trata de un *proxy* inverso ligero y de alto rendimiento. Sin embargo, no es uno de los servicios más utilizados de NGINX, por lo que no cuenta con muchas funcionalidades ni se puede personalizar en exceso. Las características más importantes que ofrece son transformaciones de *headers*, configuración de *buffers* a modo de memoria caché y creación de reglas de enrutamiento en base a las direcciones origen de las peticiones.

Se configura a través de un fichero el cual puede ser recargado durante su ejecución, es decir, se puede configurar dinámicamente. Un ejemplo de este lo muestra la figura 2.2.

Para finalizar, algunos de los productos y servicios de NGINX son ampliamente conocidos y utilizados por empresas de todo el mundo, entre ellas: Starbucks ², Bank of America ³ o Capital One ⁴. No obstante, su función de *proxy* inverso no es una de ellas, como ya se ha comentado.

¹Web oficial de NGINX: <https://www.nginx.com>.

²Web oficial de Starbucks: <https://www.starbucks.es>.

³Web oficial de Bank of America: <https://www.bankofamerica.com>.

⁴Web oficial de Capital One: <https://www.capitalone.com>.

```
location /app1/ {  
    proxy_bind 127.0.0.1;  
    proxy_pass http://example.com/app1/;  
}  
  
location /app2/ {  
    proxy_bind 127.0.0.2;  
    proxy_pass http://example.com/app2/;  
}
```

Figura 2.2: Ejemplo de configuración de NGINX [8].

2.2.2. Kemp

Kemp⁵ es un balanceador de carga muy utilizado a nivel mundial. El término balanceador de carga se refiere a un *proxy* inverso al que se le quiere dar énfasis a su capacidad de distribuir la carga entre varios nodos. Por esta característica es por la que se suele utilizar un producto de este tipo pero no hay que obviar que no deja de ser un *proxy* inverso, con todo lo que conlleva.

En cuanto a sus características, ofrece ser un balanceador de nivel 4 pero también de nivel 7. Esto hace referencia a la capa del Modelo OSI (figura 2.3) sobre la que actúa. Además, soporta HTTP/2, cuenta con un *firewall* que se puede activar y tiene un mecanismo de *health checks* para comprobar el estado de los *endpoints* con los que trabaja.

Una de los aspectos más especiales de Kemp es el hecho de que brinda múltiples opciones de implementación, y muy flexibles. Destacan la *hardware*, que facilita la instalación de máquinas con el *software* de Kemp, y la *cloud*, la cual pone a disposición los servicios de manera remota y sin necesidad tener servidores físicos. Cada una de ellas tiene diferentes tarifas en función del tráfico a recibir o de la cantidad de puertos disponibles, entre otros.

El precio por utilizar los servicios de Kemp varía en función de las características del proyecto en el que se quiere incluir. Hay que rellenar un pequeño cuestionario para recibir un presupuesto. En él, hay que indicar cosas como los productos que se quiere contratar y la cantidad de los mismos.

Ejemplos de empresas que utilizan Kemp en sus infraestructuras son Audi⁶, Nestle⁷ o Galp⁸.

2.3 Librerías

2.3.1. YARP

YARP⁹ es una librería para ASP.NET Core hecha por el propio Microsoft¹⁰ y su objetivo es facilitar la creación de *proxy* inversos. Surgió dentro de la propia infraestructura de la empresa, en la que diferentes equipos preguntaban si existía algún producto de este tipo disponible para utilizar en sus respectivos proyectos. La respuesta de la com-

⁵Web oficial de Kemp: <https://kemptechnologies.com>.

⁶Web oficial de Audi: <https://www.audi.es>.

⁷Web oficial de Nestle: <https://www.nestle.com>.

⁸Web oficial de Galp: <https://www.galpenenergia.com/>.

⁹Web oficial de YARP: <https://microsoft.github.io/reverse-proxy>.

¹⁰Web oficial de Microsoft: <https://www.microsoft.com>.

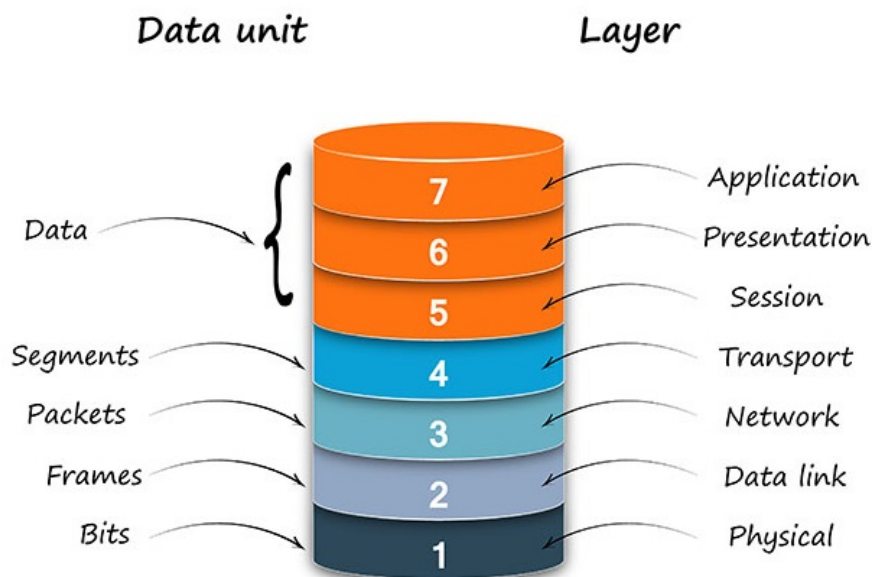


Figura 2.3: Modelo OSI [10].

pañía fue crear un equipo especializado para construir esta librería y estandarizar así su uso. Además, se decidió hacer pública tanto su existencia 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 inmediatamente más moderna.

A raíz de la heterogeneidad de casos de uso que debe cubrir para satisfacer las necesidades de los diferentes equipos de Microsoft, 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 *proxy* inversos de nivel 7, la capa 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 ASP.NET [11], haciendo que su eficiencia sea muy alta.

Por último, la documentación propia no es escueta 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 en portales como Stack Overflow ¹¹.

¹¹Web oficial de Stack Overflow: <https://es.stackoverflow.com>.

2.3.2. Ocelot

Ocelot ¹² es una librería para ASP.NET Core que permite a una aplicación de ese *framework* actuar como *API Gateway*. Como su propia web dice, está pensada para arquitecturas orientadas a servicios o a microservicios que necesitan un punto único de entrada.

Se trata de un proyecto que ya tiene cierto tiempo, pues se inició en 2016. No obstante, su última *release* salió a la luz el 11 de Diciembre de 2020 y es compatible con la última versión de .NET, .NET 5. También lo es con todas las anteriores de .NET Core.

En lo que se refiere a las características propias de *API Gateway*, Ocelot ofrece agregación de peticiones y tareas comunes a todos los *endpoints* como autenticación o trazas distribuidas. También puede hacer transformaciones de *headers* o cambiar el método HTTP utilizado antes de redirigir una petición.

Otra de sus características destacables son el hecho de ser *open source*, su rapidez y su escalabilidad. También es conveniente comentar que puede proporcionar autenticación, descubrimiento de servicios, equilibrio de carga, cacheado y limitación de carga. En sus inicios, uno de sus peores puntos negativos era que no permitía cambiar su configuración de manera dinámica, sin embargo, esta característica fue añadida más tarde y ahora ya está disponible.

Al igual que YARP, Ocelot también está formado por una serie de *middlewares* que se integran en la pipeline de ASP.NET para llevar a cabo sus tareas de *API Gateway*. Su configuración es muy básica, teniendo que especificarla en un fichero Json. La figura 2.4 muestra un ejemplo muy básico de este fichero y la figura 2.5 uno con una agregación de peticiones. De estos ejemplos es interesante comentar la nomenclatura que utiliza esta librería para diferenciar el flujo de entrada de peticiones (*upstream*) del de salida (*downstream*). Por último, en el segundo ejemplo se observa lo sencillo que es configurar agregaciones de peticiones, indicando un identificador, la ruta base de peticiones a capturar y los identificadores de las rutas que agregar.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/todos/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "jsonplaceholder.typicode.com",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/todos/{id}",
      "UpstreamHttpMethod": [ "Get" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://localhost:5000"
  }
}
```

Figura 2.4: Ejemplo básico de configuración de Ocelot [9].

¹²Web oficial de Ocelot: <https://threemammals.com/ocelot>.

2.4 Comparativa

Precio, API Gateway, Reverse proxy, documentación, comunidad, previa configuración, cantidad de posible configuración, extensibilidad, recarga de configuración dinámica, capa modelo OSI.

2.5 Conclusiones

Una vez analizadas las diferentes opciones para conseguir un *proxy* inverso, es conveniente comentar la opción escogida y explicar porqué lo ha sido, a modo de conclusión.

El uso de productos ya hechos proporciona facilidad de configuración y se puede considerar como la opción fácil, ya que se puede tener funcionando un *proxy* inverso a pleno rendimiento en no mucho tiempo. No obstante, el contexto de la aplicación de la que tiene que formar parte hace que sea mejor elaborar un *proxy* inverso propio, utilizando una librería que facilite esta tarea. Las dos razones con más peso son el hecho de que se pretende que el microservicio encargado de los despliegues se comuniquen con dicho producto y la facilidad de despliegue que se puede conseguir al utilizar la arquitectura específica seguida por los microservicios. De esta manera, utilizando un producto ya hecho no se podría conseguir un nivel alto de cohesión y se tendrían que abordar problemas de integración, así como desplegar el mismo de una manera diferente a los microservicios.

Por otro lado, la sencillez de YARP y sus altas prestaciones hacen que destaque frente a Ocelot, la que permite crear *API Gateways*. En concreto, no se quería tener la funcionalidad de ninguno de los tres puntos clave que caracterizan a los *API Gateway* (apartado 2.1), así que, por este motivo, y por la búsqueda de sencillez, ha sido más apropiado decantarse por un *proxy* inverso.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/",
      "UpstreamPathTemplate": "/laura",
      "UpstreamHttpMethod": [
        "Get"
      ],
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 51881
        }
      ],
      "Key": "Laura"
    },
    {
      "DownstreamPathTemplate": "/",
      "UpstreamPathTemplate": "/tom",
      "UpstreamHttpMethod": [
        "Get"
      ],
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 51882
        }
      ],
      "Key": "Tom"
    }
  ],
  "Aggregates": [
    {
      "RouteKeys": [
        "Tom",
        "Laura"
      ],
      "UpstreamPathTemplate": "/",
      "Aggregator": "FakeDefinedAggregator"
    }
  ]
}
```

Figura 2.5: Ejemplo de configuración de Ocelot con una agregación de peticiones [9].

CAPÍTULO 3

Tecnología utilizada

La tecnología utilizada es .NET [12], más concreto, ASP.NET Core [5], con el lenguaje de programación C# [13] y el entorno de desarrollo habitual para él, Visual Studio ¹, en la versión del 2019. Como se comenta en la introducción del capítulo anterior, 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. Esto corresponde al requisito no funcional RNF03, que se verá en el apartado 4.1.3.

En otro orden de ideas, 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.

Por último, se ha utilizado Postman ² 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.

3.1 Ejemplo de uso básico de YARP

Una vez comentada la tecnología elegida, 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 líneas 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 hacia 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 poder visualizar a través de ella la cantidad de peticiones que responden.

Para empezar con el ejemplo, se crea un proyecto web vacío de .NET, en concreto, de ASP.NET Core. 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 [14] y para buscar qué paquetes hay disponibles e instalarlos se puede utilizar un asistente gráfico, figura 3.1.

¹Web oficial de Visual Studio: <https://visualstudio.microsoft.com/es/vs>.

²Web oficial de Postman: <https://www.postman.com>.

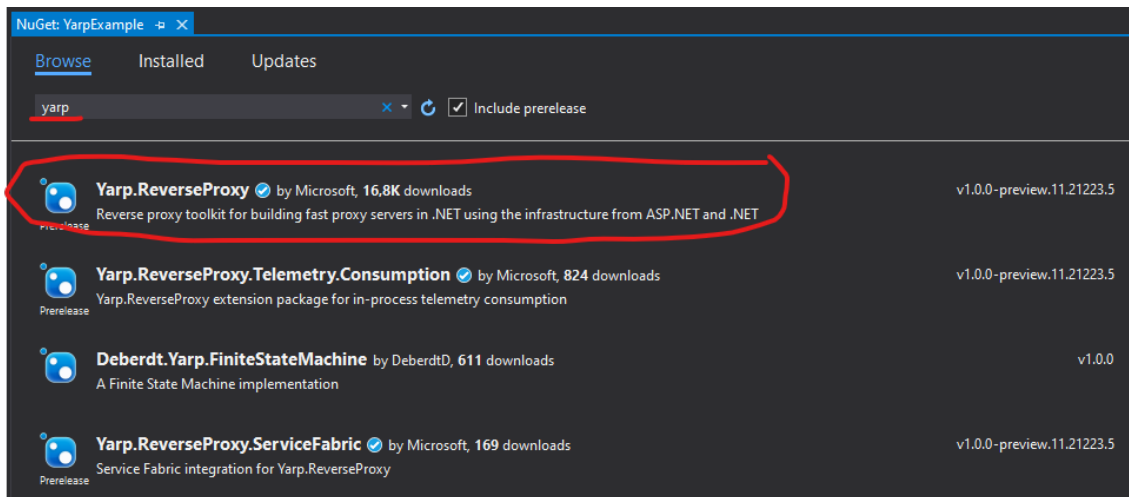


Figura 3.1: Instalación del NuGet de YARP.

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 *MapReverseProxy()* dentro del *UseEndpoint*. Para visualizar mejor estos cambios, figura 3.2.

```
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 3.2: 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 3.3 muestra las rutas creadas para esta prueba. Se ha creado un par *ruta-cluster* con dos destinos. Para no entrar en mucho detalle, lo que es importante es que la ruta tiene un patrón `**catch-all` dentro de la cláusula `Match:Path`, para capturar todas las peticiones, y el *cluster* contiene dos destinos, 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.

```
"ReverseProxy": {
  "Routes": {
    "locationRoute": {
      "ClusterId": "locationCluster",
      "Match": {
        "Path": "**catch-all"
      }
    }
  },
  "Clusters": {
    "locationCluster": {
      "Destinations": {
        "locationCluster_firstDestination": {
          "Address": "http://localhost:5001"
        },
        "locationCluster_secondDestination": {
          "Address": "http://localhost:5002"
        }
      }
    }
  }
}
```

Figura 3.3: Fichero `appsettings.json` con la configuración de las rutas para el *proxy* inverso.

Lo siguiente es preparar los servicios para una prueba rápida. En la figura 3.4 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 los 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 configurado en el `appsettings.json`. En la figura 3.5, se muestra el escenario anterior tras ejecutar la petición Postman repetidas veces. Se observa que 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 3.4: Escenario inicial de prueba de YARP con los servicios preparados.



Figura 3.5: Escenario de prueba de YARP con los servicios después de realizar algunas consultas.

CAPÍTULO 4

Desarrollo de la solución

Este capítulo va a explicar el proceso que se ha seguido para construir el *proxy* inverso. Primero se detallará la funcionalidad que este debe tener mediante la especificación de requisitos previa al desarrollo. En segundo lugar, se verá una sección de diseño para ver cómo se organiza el código de la solución implementada. A continuación, se explicará la metodología *software* seguida. Después, la sección más amplia, correspondiente a la programación del *proxy* inverso, la cual se comentará dividida en fases con las funcionalidades implementadas en cada una de ellas y los problemas encontrados. Por último, se expondrán las pruebas realizadas para verificar el correcto funcionamiento del producto elaborado.

4.1 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 otros requisitos funcionales y no funcionales que el producto debe satisfacer.

4.1.1. Casos de uso

Los casos de uso van a ser expuestos a continuación con un diagrama de casos de uso, figura 4.1, 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:

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 <i>proxy</i> 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.

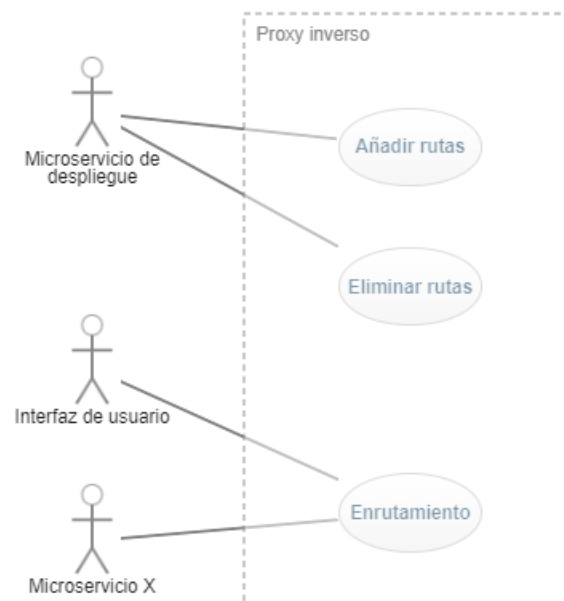


Figura 4.1: Diagrama de casos de uso del *proxy* inverso.

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 <i>proxy</i> 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 <i>proxy</i> inverso y esta es capturada y redirigida hacia un <i>endpoint</i> 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.

4.1.2. Otros requisitos funcionales

Aparte de los casos de uso comentados en el apartado anterior, el *proxy* inverso ha de cumplir otros requisitos funcionales, o características concretas, que no son propiamente casos de uso pero que se considera de interés mencionarlos, pues son relevantes para su funcionamiento. Los requisitos funcionales a mostrar 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, ya que ambos se pueden considerar requisitos funcionales. No obstante, 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 <i>proxy</i> 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 <i>proxy</i> inverso tiene que estar preparado para servir de puerta de entrada al <i>backend</i> desde la interfaz de usuario pero también debe interconectar los microservicios que forman dicho <i>backend</i> . 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 <i>proxy</i> 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 hacia 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 <i>proxy</i> inversos
Caso de uso relacionado	CU03
Descripción	Se debe poder trabajar con más de un <i>proxy</i> inverso a la vez, de manera que uno enrute una petición hacia otro <i>proxy</i> inverso y este segundo la enrute hacia 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 <i>proxy</i> 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.

4.1.3. 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 [15], comúnmente llamada SQuARE (*System and software Quality Requirements and Evaluation*), define ocho características principales y algunas subcaracterísticas específicas para cada una. La figura 4.2 las muestra todas.



Figura 4.2: Características y subcaracterísticas de calidad de un producto *software* definidas en la ISO/IEC 25010 [15].

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 <i>proxy</i> inverso.
Motivación	Evitar que el <i>proxy</i> 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.

4.2 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.

4.2.1. Estructura de la solución

En el apartado 4.1.3, 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 autogenerada de 8 capas:

- **Dominio.** En ella se encuentran las entidades que representan el dominio de la aplicación. Estas son creadas en base a su previo modelado.
- **Contratos.** Principalmente contiene los DTO, u objetos para la transferencia de datos [16]. Son objetos que representan las entidades de dominio que utiliza el sistema internamente con la intención de no darlas a conocer directamente. De esta forma, se ofrece la información que se quiere y se puede ocultar o modificar parte de ella. Además, esta capa también contiene las interfaces de las acciones del *backend* que se invocan a través de la capa de *proxy*.
- **Persistencia.** Es la capa que se encarga de acceder y realizar las operaciones relacionadas con la base de datos. El *proxy* inverso no necesita persistir nada por lo que esta capa no existe en este caso.
- **Lógica.** Esta capa contiene la lógica de negocio. Al igual que la entidades de dominio, las acciones que representan la lógica han de ser modeladas. No solo incluye código autogenerado sino que las acciones son programadas aquí también.
- **Aplicación.** Se encarga de dar soporte a las operaciones CRUD (*create, read, update y delete*). También comprueba los permisos del usuario sobre la acción que se quiere realizar.
- **Servicios.** Representa el punto de entrada al microservicio. Define las acciones HTTP que expone el servicio a través de controladores.
- **Proxy.** Contiene el código necesario para invocar las acciones del backend, expuestas en la capa de servicios. Esta capa es utilizada desde la interfaz de usuario para realizar peticiones al microservicio o cuando un microservicio se comunica con otro.
- **Referencias externas.** Se encarga de registrar los *proxies* que se van a consumir para abstraer estas dependencias y así poder solucionar problemas de dependencias cíclicas.

Tras comentar las capas que forman la arquitectura del *proxy* inverso, se muestra en la figura 4.3 un esquema de cómo interactúan entre ellas. Es interesante resaltar el hecho de que la capa de Contratos actúa como capa base y la mayoría de las demás la referencian. Por otra parte, la distinción *frontend/backend* se aprecia con claridad.

Capa de lógica

La capa que más ha sido necesario modificar es la de lógica, pues las acciones del *proxy* inverso se implementan en ella. Es por esto que se va a entrar en más detalle acerca de esta capa. Las figuras 4.4 y 4.5 muestran la estructura de carpetas y clases que forman los proyectos de la capa de lógica y de sus pruebas, respectivamente. No todas las clases pero sí las más importantes serán comentadas a continuación.

En el proyecto de lógica de la figura 4.4 son reseñables los siguientes aspectos:



Figura 4.3: Esquema de las capas de la arquitectura del *proxy* inverso.

- La carpeta *Configuration* contiene clases que se encargan de procesar y llevar a cabo los cambios de configuración de rutas de manera dinámica. Utilizan y se comunican con las interfaces definidas por YARP para ello.
- La clase *RoutesLoaderHostedService* se encarga de realizar una carga inicial de rutas predefinidas, esto es, cuando se inicia el microservicio.
- El *RoutesLogicManager* implementa las acciones propias del *proxy* inverso, es decir, las acciones de añadir y eliminar rutas.
- La carpeta llamada *Middlewares* contiene los *middlewares* creados que se añaden a la *pipeline* de YARP. Se profundiza en esto en el siguiente apartado, 4.2.2.
- La clase *RoutesUtils* extrae métodos de ayuda que se utilizan de forma repetida tales como la creación de identificadores de manera idempotente.

En el proyecto de pruebas (figura 4.5) simplemente destacar la existencia de tres clases de prueba principales: *AddNewRoutesTests*, *RemoveRoutesTests* y *RoutesUtilsTests*. La primera es para la acción de añadir rutas, la segunda para la de eliminarlas y la tercera para la clase *RoutesUtils*. Por último, la clase *LogicTestUtilities* incluye funciones auxiliares para realizar comprobaciones que se repiten en más de una prueba, como por ejemplo la comprobación de si una determinada ruta existe.

4.2.2. 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. Importante destacar que, debido a esa sucesión, el orden en el que se ejecutan los diferentes procesos sí es relevante. Para gestionar las peticiones, ASP.NET define una de ellas en el



Figura 4.4: Proyecto de Visual Studio de la capa de lógica.

método *Configure* [11]. 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 4.6 muestra como ha quedado configurada esta última en el *proxy* inverso. En ella, se observa que se han utilizado los siguientes:

- **ProductVersionFilter.** Este *middleware* surge de la posibilidad de coexistir más de una versión del mismo *endpoint*. Así pues, es el encargado de averiguar qué versión de API quiere utilizar cada petición que se recibe para eliminar las posibles destinaciones que tengan una diferente. De esta manera, se asegura que las peticiones serán redirigidas a un *endpoint* con la versión correcta. Dicha información se manda en cada petición dentro de un *header* y, en caso de no encontrar ninguna destinación disponible con la versión correcta, se envía hacia el *endpoint* con versión mayor.
- **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 hacia ella. Esto puede suceder cuando el microservicio destino se encuentra desplegado en la

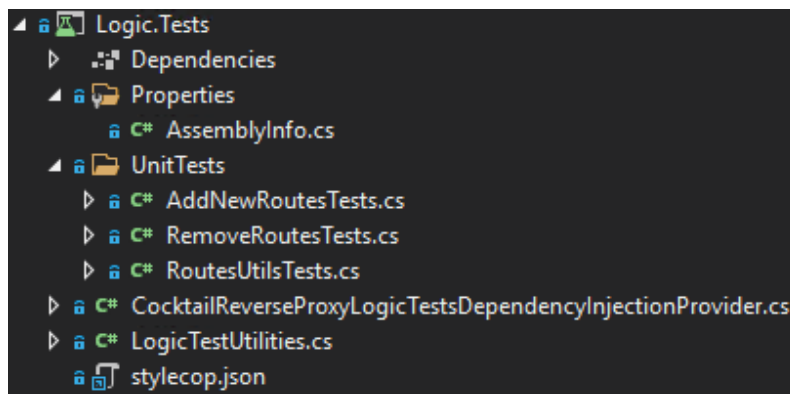


Figura 4.5: Proyecto de Visual Studio de pruebas para la capa de lógica.

```
endpoints.MapReverseProxy(reverseProxyPipeline =>
{
    reverseProxyPipeline
        .UseMiddleware<ProductVersionFilter>()
        .UseMiddleware<SameMachineFilter>()
        .UseMiddleware<CustomProxyLog>()
        .UseProxyLoadBalancing();
});
```

Figura 4.6: Pipeline de YARP configurada para el proxy inverso.

misma máquina. De esta manera se ahorra tiempo evitando hacer una petición a otra dirección.

- **CustomProxyLog.** Se encarga de personalizar los mensajes de *log* que muestra el proxy inverso por consola cuando enruta una petición. YARP muestra por defecto un mensaje pero se considera interesante mostrar más información para facilitar las tareas de depuración, por ejemplo.
- **ProxyLoadBalancing.** No es un *middleware* creado para este proyecto, se trata de uno predefinido por YARP que se encarga de realizar las tareas de balanceo de carga cuando una misma ruta tiene diferentes posibles destinaciones. En concreto, elige una entre las disponibles en base a un algoritmo que se puede configurar. Tiene que ser el último *middleware* añadido porque es él el que se encarga de elegir finalmente dónde se redirigen las peticiones de entre las destinaciones disponibles y no filtradas por los anteriores.

4.3 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.

4.3.1. Construcción de prototipos

El desarrollo ha tenido un inicio particular, 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.

De esta forma, se construyó un microservicio sin utilizar la generación de código y otro utilizándola. Esto culminaría con la medición de las diferencias de rendimiento entre ambos y la consiguiente elección, para continuar con el desarrollo de uno u otro prototipo.

Por otro lado, cabe destacar que a los *proxy* 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 4.1.3. 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 4.1.1) 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 3.1 y un ejemplo de configuración de rutas estáticas se puede ver en la figura 3.3.

Después de realizar las tareas comentadas queda un proyecto muy simple pero totalmente funcional. Se muestra la solución de Visual Studio [17] correspondiente en la figura 4.7.

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 4.5) 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 desde 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.

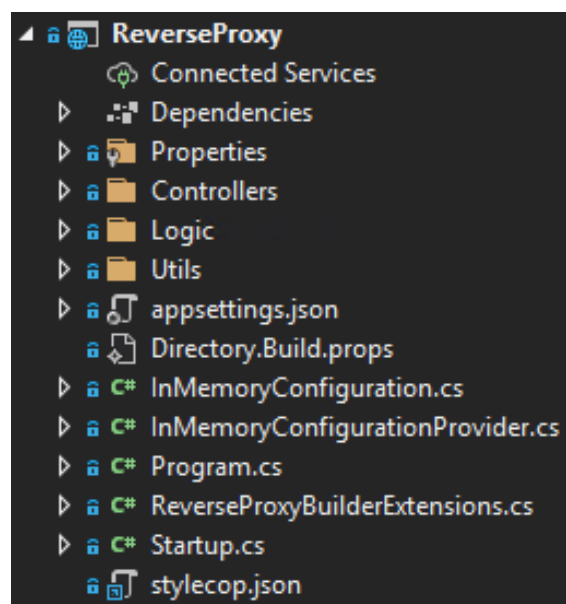


Figura 4.7: Solución de Visual Studio del prototipo de *proxy* inverso hecho a mano.

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 4.8. Es interesante comparar esta solución con la del prototipo hecho a mano para comprobar que la complejidad y cantidad de proyectos de Visual Studio, y por consiguiente también de clases, es muchísimo mayor en este caso. Cabe resaltar que faltarían los proyectos correspondientes a la capa de dominio y contratos, los cuales se encuentran en una solución aparte y no es necesario mostrarlos para realizar esta comparativa puesto que las evidencias comentadas son suficientes.

Comparativa

Una vez contruidos 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 un número determinado de veces la misma petición sobre el *proxy* inverso que se le indicara para que este la redirigiera y calcular así el tiempo medio de respuesta. La figura 4.9 muestra su código principal, el cual era ejecutado una vez por cada prototipo.

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,

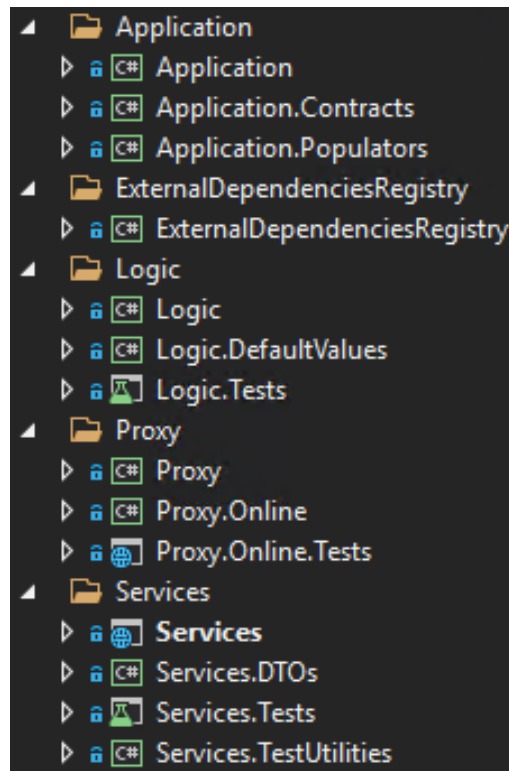


Figura 4.8: Solución de Visual Studio del prototipo de *proxy* inverso hecho con generación de código.

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 4.1.3.

Los resultados de las mediciones fueron claros: ambos prototipos tenían un rendimiento casi idéntico. En concreto, para 10000 iteraciones se obtuvieron tiempos medios de respuesta de 3,6 milisegundos para el prototipo a mano y de 3,7 para el autogenerado. Se realizaron pruebas con diferente número de iteraciones pero los resultados no variaron en demasía. Con esto se concluye que utilizar el segundo prototipo supone una penalización de tiempo de no llega al 3 %. Esto se consideró despreciable e insuficiente para contrarrestar las ventajas comentadas por lo que se decidió eliminar el primer prototipo y continuar así con la versión autogenerada.

4.3.2. 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 ajustados. 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 4.10 muestra cómo queda la clase *Startup*.

```
private static async Task MeasureAverageTime(int listeningPort)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    for (int i = 0; i < iterations; i++)
    {
        await MakeRequest(listeningPort);
    }

    stopwatch.Stop();

    Console.WriteLine((stopwatch.Elapsed.TotalMilliseconds / iterations) + " ms");
}
```

Figura 4.9: Código principal de la aplicación de consola para medir los tiempos del enrutamiento de los dos prototipos.

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 4.1.1. También 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 4.1.3. 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, requiere que se utilicen barras como separadores de ruta en lugar de barras invertidas. Para ello se creó un método *GetYarpNormalizedPath*, en la clase *RoutesUtils*, que normaliza una ruta dada según el esquema que YARP acepta. El método en cuestión se muestra en la figura 4.11 y hace cosas como forzar el uso de las barras como separadores o añadir el patrón "*{*remainder}*" para capturar todas las peticiones con rutas que empiecen con el prefijo indicado independientemente de como sigan.

Además, se consideró oportuno automatizar algunas de las pruebas de aceptación de las acciones de añadir y eliminar rutas y para el método *GetYarpNormalizedPath*, recién comentado en el párrafo anterior. Se entrará más en detalle acerca de estas pruebas en el apartado de pruebas 4.4.

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. Se ahondará en este aspecto de los despliegues en el siguiente apartado. 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.

```

/// <summary> The class for an ASP.NET Core Application where the services are c ...
8 references | TFS2015AGENT, 26 days ago | 4 authors, 6 changes
public partial class Startup
{
    /// <summary> This method gets called by the runtime. Use this method to configu ...
    0 references | Víctor Alberto Iranzo Jiménez, 44 days ago | 2 authors, 2 changes
    public void Configure(IApplicationBuilder application, IWebHostEnvironment environmen
    {
        ConfigurePipeline(application);
    }
}

/// <summary> The class for an ASP.NET Core Application where the services are c ...
11 references | Alejandro Carrión Sanmartín, 18 days ago | 2 authors, 6 changes
public partial class Startup
{
    /// <summary> Configures the pipeline with custom code.
    2 references | Alejandro Carrión Sanmartín, 18 days ago | 2 authors, 5 changes
    public static void ConfigurePipeline(IApplicationBuilder application)
    {
        application.UseEndpoints(endpoints =>
        {
            endpoints.MapReverseProxy(reverseProxyPipeline =>
            {
                reverseProxyPipeline
                    .UseMiddleware<SameMachineFilter>()
                    .UseMiddleware<CustomProxyLog>()
                    .UseProxyLoadBalancing();
            });
        });
    }
}

```

Figura 4.10: Clase parcial *Startup* del *proxy* inverso.

4.3.3. Primeros 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 hacer algunas modificaciones para que la interfaz de usuario y los microservicios lo utilizaran. Derivado de estos despliegues se presentaron algunos problemas que también serán comentados.

Sistema de despliegue

Primero de todo, se va a explicar como se realizan los despliegues de la aplicación. Como se ha comentado de pasada en varios puntos de esta memoria, la aplicación sobre la cual se trabaja tiene un microservicio especial encargado de orquestar los despliegues. La cuestión es que no está del todo operativo todavía, pues la aplicación se encuentra en desarrollo, y por este motivo se realizan dos tipos de despliegue:

- El primero de ellos consiste en desplegar los microservicios mediante contenedores Docker ¹. Se trata de una manera de poder realizar despliegues antes de que el microservicio específico esté terminado. Permite así hacer pruebas con el resto y ver como se comportan.
- El segundo es el que lleva a cabo el microservicio pensado para ello. A grandes rasgos, este ordena y organiza los servidores preparados para correr la aplicación y les dice a cada uno qué microservicio o microservicios tiene que ejecutar. Por el momento, los microservicios que despliega no se utilizan porque se encuentra en fase de pruebas.

¹Web oficial de Docker: <https://www.docker.com>.


```
/// <summary> Gets the path that Yarp requires from the requests base path.
3 references | Alejandro Carrión Sanmartín, 40 days ago | 1 author, 1 change
public static string GetYarpNormalizedPath(string requestsBasePath)
{
    string trimmedRequestsBasePath = requestsBasePath
        .TrimStart('/') .TrimStart('\\') .TrimEnd('/') .TrimEnd('\\');

    DirectoryInfo requestsBasePathDirectoryInfo =
        new DirectoryInfo(Path.AltDirectorySeparatorChar + trimmedRequestsBasePath);

    string normalizedRequestsBasePath = requestsBasePathDirectoryInfo.FullName
        .Replace(requestsBasePathDirectoryInfo.Root.Name, string.Empty, StringComparison.Ordinal)
        .Replace(Path.DirectorySeparatorChar, Path.AltDirectorySeparatorChar);

    return Path.AltDirectorySeparatorChar + normalizedRequestsBasePath
        + Path.AltDirectorySeparatorChar + "{*remainder}";
}
```

Figura 4.11: Método *GetYarpNormalizedPath*.

Una vez expuestos los modos de despliegue y el por qué de su coexistencia, cabe resaltar que el *proxy* inverso se ve obligado a ser capaz de trabajar con ambos. No obstante, en un futuro desaparecerá el primero de ellos y el segundo será el que se emplee de manera definitiva.

Interfaz de usuario

Para empezar con los despliegues, se decidió hacer que solo la interfaz de usuario utilizara el *proxy* inverso para no romper toda la aplicación en caso de fallar. Cabe destacar que, como la aplicación se encontraba todavía en fase de desarrollo, no hubiera sido un gran problema si la interfaz de usuario quedara inutilizable por un breve periodo de tiempo por culpa del nuevo microservicio. Sin embargo, la caída del *backend* podría haber supuesto algún inconveniente para el resto del equipo de la aplicación puesto que no sería posible hacer pruebas sobre los microservicios desplegados. De esta manera, se pretendía minimizar riesgos.

Además, como paso previo a incluir el *proxy* inverso al proceso de despliegue, también se hicieron pruebas de manera local para comprobar que todo funcionaba correctamente y que la interfaz de usuario no se iba a quedar caída. Se desplegó el *proxy* inverso en una máquina a parte y se le cargó de rutas que apuntaban a los microservicios ya desplegados para que la interfaz de usuario lo utilizara. Se hicieron peticiones desde dicha interfaz de usuario y no hubo problemas.

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*. Para ello, fue necesario modificar las direcciones a las que apuntaba, en concreto, substituir las de cada microservicio por la del *proxy* inverso. De esta forma, la interfaz de usuario dejaba de conocer todos los microservicios para solo conocer su dirección.

No obstante, las direcciones de los microservicios ahora debían ser conocidas por el *proxy* inverso, por lo que hubo que cargárselas. Aquí es donde entran en juego los diferentes modos de despliegue. Para los despliegues con el microservicio específico, éste es quien tiene que cargar de rutas el *proxy* inverso cada vez que le ordene a un servidor ejecutar un determinado microservicio. Con respecto al despliegue en Docker, las rutas han de ser cargadas de una manera predefinida. Para especificar dichas rutas, se ha creado un fichero llamado *defaultRoutes.json* (figura 4.12), que es leído al iniciarse el *proxy* inverso y las rutas que describe son añadidas. De esto se encarga una clase llamada *Rou-*

tesLoaderHostedService, que, como su propio nombre indica, es un *hosted service* [18]. Este tipo de clase ejecuta una determinada lógica como tarea en segundo plano.

```
"Routes": {
  "Microservice1": {
    "RequestsBasePath": "Microservice1/api",
    "Address": "http://localhost:4220"
  },
  "Microservice2": {
    "RequestsBasePath": "Microservice2",
    "Address": "http://localhost:4125"
  },
  "Microservice3": {
    "RequestsBasePath": "Microservice3/requests",
    "Address": "http://localhost:4621"
  }
}
```

Figura 4.12: Fichero *defaultRoutes.json*.

Microservicios

Una vez la interfaz de usuario funcionaba correctamente con el *proxy* inverso desplegado, se hizo que los propios microservicios lo utilizaran para comunicarse entre ellos. Igual que en el caso anterior, también fue necesario cambiar las direcciones a las que apuntaban los microservicios, ahora pasarían a conocer solo al *proxy* inverso.

Por otro lado, una vez más hay que diferenciar los cambios para los dos modos de despliegue. Al respecto del de Docker, también se añadieron al *defaultRoutes.json* las rutas que fueron necesarias, correspondientes a algunos microservicios que no son utilizados por la interfaz de usuario. En referencia al segundo, se mejoró la manera en la que el microservicio de despliegue añadía rutas al *proxy* inverso.

Como modificación común para los dos modos, se tuvo que repensar el orden de despliegue de los microservicios. Estos, cuando son levantados, intentan contactar con sus dependencias y fallan si no las encuentran vivas. Por este motivo, el *proxy* inverso debe estar en marcha cuando esta especie de saludos se lleven a cabo. El problema es que él también depende de otros microservicios, creando así un círculo de dependencias. La solución por la que se apostó fue levantar a la vez los microservicios que se entrelazan en este círculo utilizando reintentos en todas sus llamadas.

Problemas

El problema más grande que surgió a raíz de los primeros despliegues fue con unos *timeouts*. Resulta que dos microservicios concretos realizaban determinadas peticiones con un *timeout* muy grande debido al elevado coste de las mismas y por algún motivo dichas peticiones no se completaban satisfactoriamente al pasar por el *proxy* inverso. El código de estado de error que tenían era el 504, correspondiente al mensaje en inglés "*Gateway timeout*". Parecía que el *proxy* inverso cortaba las llamadas antes de que terminaran. Para solucionarlo momentáneamente y no dejar caídos esos microservicios, se optó por hacer que ellos y solo ellos no lo utilizaran hasta que el problema fuera analizado en detalle.

El problema recién expuesto fue estudiado a fondo más tarde y al final resultó ser culpa de YARP, que utiliza un *timeout* por defecto de 100 segundos para redirigir peticiones. La solución fue aumentarlo para las rutas correspondientes a las peticiones que se redirigen hacia los microservicios en cuestión. Para ello, fue necesario modelar y añadir un

parámetro opcional para el *timeout* en la creación de las rutas. De esta forma se permite su personalización en función de la ruta que se utilice.

La solución dada al problema puede parecer que no es la óptima ya que cualquier petición relacionada con una ruta de *timeout* grande podría durar más de lo esperado, es decir, el aumento del *timeout* afecta a todas las peticiones que se emparejan con esa ruta, siendo no del todo correcto. Una posible mejora sería crear rutas específicas con *timeouts* grandes para solo las peticiones que lo necesiten. No obstante, no se cree necesario complicar el funcionamiento de esa manera ya que, en la práctica, el comportamiento del *proxy* inverso no variaría demasiado. Además, si un cliente con un determinado *timeout* realiza una petición a través del *proxy* inverso y la ruta correspondiente tiene uno mayor, el que prevalecerá será el del cliente y el del *proxy* inverso no tendrá efecto. De esta manera, solo sirve de límite en caso de que el del *proxy* inverso sea menor.

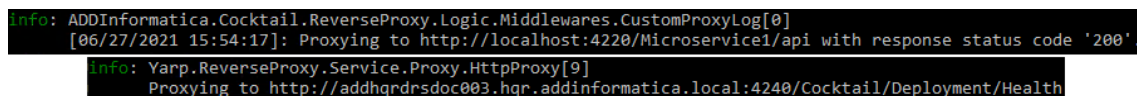
Un problema menor que también surgió fue uno relacionado con los códigos de estado de las peticiones. El *proxy* inverso devolvía uno con número 502. Este a menudo era confundido con el 504 recién explicado y provocaba malentendidos. Su mensaje en inglés era "*Bad Gateway*", lo que parecía ser culpa del nuevo componente, generando confusión en el resto de desarrolladores de la aplicación. Finalmente resultó ser debido a enrutamientos cuyo destino no estaba disponible, es decir, el *proxy* inverso no tenía ninguna culpa, simplemente mostraba un 502 en lugar del habitual 404, "*Not Found*", que se hubiera mostrado en caso de que las peticiones no fueran respuestas por su destinatario.

4.3.4. Producto final

En esta etapa final se hicieron algunos retoques y se implementaron algunas funcionalidades más que se pasan a detallar.

Personalización de los mensajes de log

Una de las mejoras que se llevó a cabo fue personalizar los mensajes de *log* para mostrar más información acerca de las peticiones enrutadas. YARP ya muestra mensajes pero se consideraba interesante añadir la fecha y hora en la que se produjo y el código de estado resultante de la petición. De esta manera, se facilitaría la labor de encontrar posibles errores derivados del enrutamiento de peticiones tales como el de los *timeouts*. Para ello, fue necesario crear y añadir un *middleware* nuevo a la *pipeline* de YARP, explicada en el apartado 4.2.2. El *middleware* creado ha sido llamado *CustomProxyLog* y la figura 4.13 muestra los mensajes por defecto de YARP y los personalizados a través de este *middleware* para ver las diferencias y el resultado final.



```
Info: ADDInformatica.Cocktail.ReverseProxy.Logic.Middlewares.CustomProxyLog[0]
[06/27/2021 15:54:17]: Proxying to http://localhost:4220/Microservice1/api with response status code '200'.
Info: Yarp.ReverseProxy.Service.Proxy.HttpProxy[9]
Proxying to http://addhqrdrsd0c003.hqr.addinformatica.local:4240/Cocktail/Deployment/Health
```

Figura 4.13: Mensajes de *log* por defecto de YARP y personalizados.

Multiinstancia de microservicios

Una de las funcionalidades más importantes añadidas en esta etapa ha sido permitir la multiinstancia de microservicios. Esta consiste en soportar la coexistencia de varias instancias de un mismo microservicio de forma que las mismas peticiones puedan ser enrutadas a diferentes direcciones. De esta forma se permite tener más de un recurso aten-

diendo determinadas peticiones, si se necesita. Relacionado con esto, se permite también realizar balanceo de carga entre las diferentes instancias de un mismo microservicio.

Para llevar a cabo esta modificación fue necesario primero mejorar el sistema de creación de identificadores. YARP utiliza tres entidades para configurar rutas: *Route*, *Cluster* y *Destination*. No es relevante entrar en mucho más detalle del funcionamiento interno pero sí comentar que las entidades de estos tipos necesitan un identificador único. Este se construía cuando se añadían rutas y de manera un poco rudimentaria por lo que hubo que hacer algo al respecto. La solución fue sistematizar su creación utilizando los parámetros de entrada y asegurando que no se pudieran repetir. Además, era importante que la forma de crearlos fuera idempotente, es decir, que siempre se obtuviera el mismo resultado para los mismos parámetros de entrada. Los métodos encargados de esto se encuentran en la clase *RoutesUtils* y se muestran en la figura 4.14.

```

/// <summary> Creates the identifier of a route in an idempotent way.
2 references | Alejandro Carrión Sanmartín, 36 days ago | 1 author, 1 change
internal static string CreateRouteIdentifier(string requestsBasePath)
{
    return CreateIdempotentIdentifier("route", requestsBasePath);
}

/// <summary> Creates the identifier of a cluster in an idempotent way.
2 references | Alejandro Carrión Sanmartín, 36 days ago | 1 author, 1 change
internal static string CreateClusterIdentifier(string requestsBasePath)
{
    return CreateIdempotentIdentifier("cluster", requestsBasePath);
}

/// <summary> Creates the identifier of a destination in an idempotent way.
2 references | Alejandro Carrión Sanmartín, 36 days ago | 1 author, 2 changes
internal static string CreateDestinationIdentifier(string requestsBasePath, string address)
{
    return CreateIdempotentIdentifier("destination", requestsBasePath, address);
}

/// <summary> Creates an idempotent identifier based on the parameters.
3 references | Alejandro Carrión Sanmartín, 36 days ago | 1 author, 1 change
private static string CreateIdempotentIdentifier(string prefix, string requestsBasePath)
{
    return $"{prefix}_{requestsBasePath}";
}

/// <summary> Creates an idempotent identifier based on the parameters.
1 reference | Alejandro Carrión Sanmartín, 36 days ago | 1 author, 2 changes
private static string CreateIdempotentIdentifier(string prefix, string requestsBasePath, string address)
{
    return CreateIdempotentIdentifier(prefix, requestsBasePath) + "${address}";
}

```

Figura 4.14: Métodos encargados de crear los identificadores de las entidades de YARP.

De una manera similar, las acciones de añadir y eliminar rutas fueron modificadas para jugar con los tres tipos de entidades de YARP. Un ejemplo del tratamiento que requieren puede ser el hecho de que un *Cluster* tiene una o más *Destinations*, por lo que no tendrá que ser creado si ya existe cuando se añade una nueva destinación. Todo esto sería trivial si se utilizara el mecanismo de carga estático pero, como se ha comentado en varias ocasiones a lo largo de este memoria, no encaja con el contexto de este *proxy* inverso.

Por otro lado, YARP ofrece diferentes algoritmos de balanceo de carga: *First*, *Random*, *RoundRobin*, *LeastRequests* y *PowerOfTwoChoices*. Se ha utilizado el último de ellos, el cual selecciona dos destinos aleatorios y elige la que menos peticiones esté atendiendo. Esta política intenta evitar la sobrecarga de buscar la dirección que está atendiendo menos peticiones y el caso peor de elegir de manera aleatoria una que esté muy ocupada.

Por último, las peticiones Postman se modificaron para hacer pruebas, así como las pruebas de aceptación automatizadas tuvieron que ser adaptadas e incluso se crearon algunas nuevas.

Segundo nivel de redirección

Otra funcionalidad de peso es el segundo nivel de redirección. Se trata de tener un *proxy* inverso en cada servidor que ejecuta alguna parte de la aplicación con la motivación de no exponer un puerto por cada microservicio. De esta forma, todas las peticiones se mandan al *proxy* inverso del servidor en cuestión y este las redirige a los microservicios que corre. Esto desemboca en un esquema de dos niveles como el que se muestra en la figura 4.15, en el que entran en juego un *proxy* inverso general y los concretos de cada servidor.

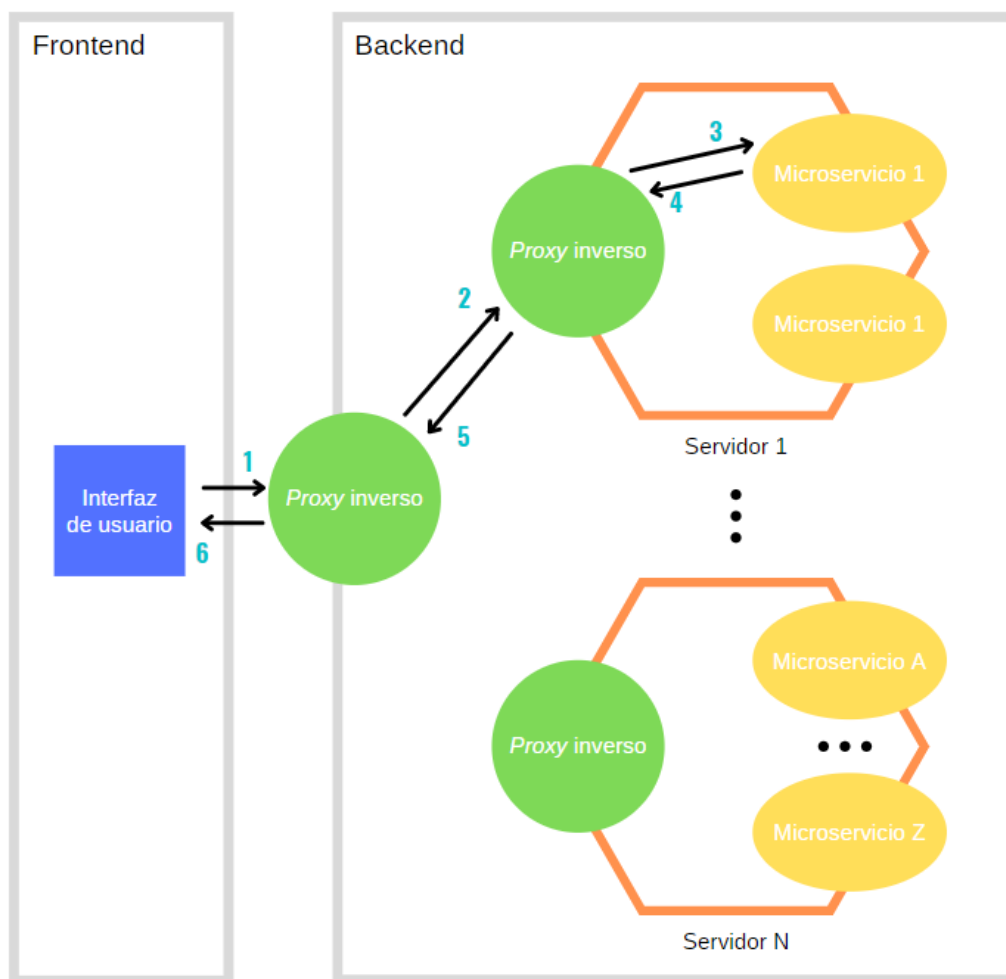


Figura 4.15: Esquema de dos niveles de *proxy* inverso.

Para implementar esta característica se plantearon varias opciones. Primero se decidió utilizar YARP también en el código de los servidores que ejecutan los microservicios pero, al poco tiempo de empezar, se encontraron problemas y aspectos que no terminaban de cuadrar con lo que se buscaba. El más importante de ellos era el hecho de tener que mantener el código que envuelve la infraestructura de YARP en dos lugares diferentes. Para evitarlo, finalmente se optó por hacer que cada servidor ejecutara un *proxy* inverso como si se tratara de un microservicio más. Es momento entonces de darse cuenta de que la elección del prototipo autogenerado fue la correcta, pues ahora permite realizar esto de una manera sencilla. También fue necesario modificar el código que carga el *proxy* inverso de rutas para hacer lo propio con todos los *proxy* inversos.

Filtro misma máquina

Otro *middleware* creado es el llamado *SameMachineFilter*. Surge de las dos nuevas funcionalidades anteriores y trata de mejorar el sistema de enrutamiento disminuyendo el tiempo de algunas peticiones. Se encarga de comprobar si existe una destinación a la que redirigir las peticiones en la misma máquina que se encuentra el *proxy* inverso y, si es el caso, enrutarlas hacia ella en lugar de hacia otra para evitar que la petición salga de la máquina y tarde más tiempo en finalizar.

Filtro versión de API

El último *middleware* desarrollado es el *ProductVersionFilter*. Este nombre lo recibe por aspectos concretos del sistema de despliegue que no vienen al caso. Su función es filtrar las destinaciones posibles a las que se puede redirigir una petición en función de la versión de API que se quiere utilizar. Esta es obtenida de un *header* de las peticiones y, si no existe dicho *header* o no se encuentra un *endpoint* con la versión correcta, la petición es redirigida a uno de los que tienen mayor versión. Este escenario no es del todo correcto y puede producir errores, por lo que se muestra un mensaje de *log* avisando del desfase entre versiones.

Cabe destacar que no fue sencillo implementar el hecho de pasar un *header* en todas y cada una de las peticiones que se lanzan entre los microservicios o desde la interfaz de usuario. Como se ha comentado varias veces, la aplicación ha sido construida con generación automática de código, así que hubo que modificar algunas plantillas a partir de las cuales este es generado para crear y enviar el *header* comentado con todas las peticiones.

Problema peticiones pesadas

De la misma manera que se encontró un fallo del *proxy* inverso correspondiente a las peticiones con *timeouts* grandes gracias a los despliegues progresivos que se han ido realizando, también se descubrió otra deficiencia relacionada con las peticiones que pesaban mucho, es decir, las que tenían un cuerpo muy grande. En este caso la petición problemática devolvía un código de estado 400, error general del lado del cliente.

Es interesante resaltar que la petición que produjo el fallo no podía ser reproducida ya que era fruto de la interacción múltiple entre varios microservicios. Además, el código de estado de error era mucho menos descriptivo que en el otro problema, un 400 genérico que no aportaba casi información. Tras mucho tiempo de investigaciones se llegó a la conclusión de que la única posibilidad era que tratase de una petición demasiado pesada. De esta forma, se intentó reproducir el mensaje de error con una petición Postman con un cuerpo muy grande y, ¡tachán!, problema encontrado. Sin embargo, faltaba averiguar cómo indicar a YARP que dejara de tener ese comportamiento o, al menos, aumentar el peso máximo de las peticiones.

La documentación oficial no decía nada al respecto por lo que se dedujo, después de unas cuantas pruebas, que YARP utilizaba la configuración de Kestrel [19] para establecer el máximo peso que puede tener una petición. Kestrel no es más que el servidor web que utilizan por debajo las aplicaciones de .NET y, por consiguiente, YARP. De esta forma, se eliminó el límite máximo de peso para las peticiones de la manera que muestra la figura 4.16, en el método *CreateHostBuilder* de la clase *Program*. Con esta solución, los microservicios destino de los enrutamientos serían quienes establecerían el peso máximo.

```
return Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder
            .UseStartup<Startup>()
            .UseSetting("UrlPrefix", "ReverseProxy")

            // The maximum size of the body of the received requests is set to null
            // because the Reverse Proxy does not have to stop any request. The target
            // servers will be do it if they are too long.
            .UseKestrel(options => options.Limits.MaxRequestBodySize = null);
    });
```

Figura 4.16: Esquema de dos niveles de *proxy* inverso.

Recorrido peticiones

Una vez comentadas todas la funcionalidades del producto desarrollado se quiere mostrar un esquema (figura 4.17) del recorrido completo de una petición que proviene de la interfaz de usuario y que 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. Entrando en detalle, la interfaz de usuario realiza una petición al microservicio 1 y para ello primero pasa por el *proxy* inverso general y después por el concreto del servidor que contiene el microservicio en cuestión. Para poder responder, el microservicio 1 necesita hacer una petición al A y este, a su vez, al Z. Una vez terminadas estas peticiones anidadas, la respuesta le llega al microservicio 1 y este devuelve a la interfaz la respuesta a la petición primera. Para finalizar, es interesante resaltar el uso de los *proxy* inversos tanto para las peticiones de la interfaz de usuario como para las que se realizan entre microservicios, es decir, nadie se comunica con nadie sin pasar por alguno de ellos.

4.4 Pruebas

Es interesante destacar que los despliegues comentados en el apartado de programación anterior (4.3) se pueden considerar pruebas alfa [20]. En ellos se introduce el *proxy* inverso en el entorno en el que finalmente va a trabajar para ver cómo responde a medida que se le añaden funcionalidades. Se trata, sin duda, de pruebas que pretenden observar el comportamiento del producto *software* y ver si ocurre algún fallo o problema. Como han sido detalladas en dicho apartado no cabe mencionar gran cosa más acerca de ellas, simplemente enfatizar que sirvieron para descubrir pequeños errores tales como el de los *timeouts* grandes o el de las peticiones pesadas. Ambos hacían que las peticiones enrutadas no terminaran correctamente y gracias a estas pruebas se pudieron solucionar antes de dar por finalizado el proyecto.

Estas pruebas realizadas durante el desarrollo tienen un porqué. Este es el hecho de querer seguir la práctica ágil de entregas continuas, comentada en la sección 4.5, con el fin de obtener *feedback* continuo acerca del grado de corrección del *proxy* inverso sobre lo que se espera de él. De esta forma, se ha considerado imprescindible no dejar las pruebas para el final, puesto que se trata de un componente clave en la arquitectura de la aplicación de la que forma parte.

Desde una perspectiva diferente a las pruebas de puesta en funcionamiento real recién descritas, se han llevado a cabo otro tipo de ellas para comprobar el correcto funcionamiento del *proxy* inverso: pruebas de aceptación y pruebas de regresión. De la misma manera, han sido ejecutadas durante el desarrollo en lugar de al final de este y serán comentadas con más detalle a continuación. También es conveniente resaltar se ha utilizado

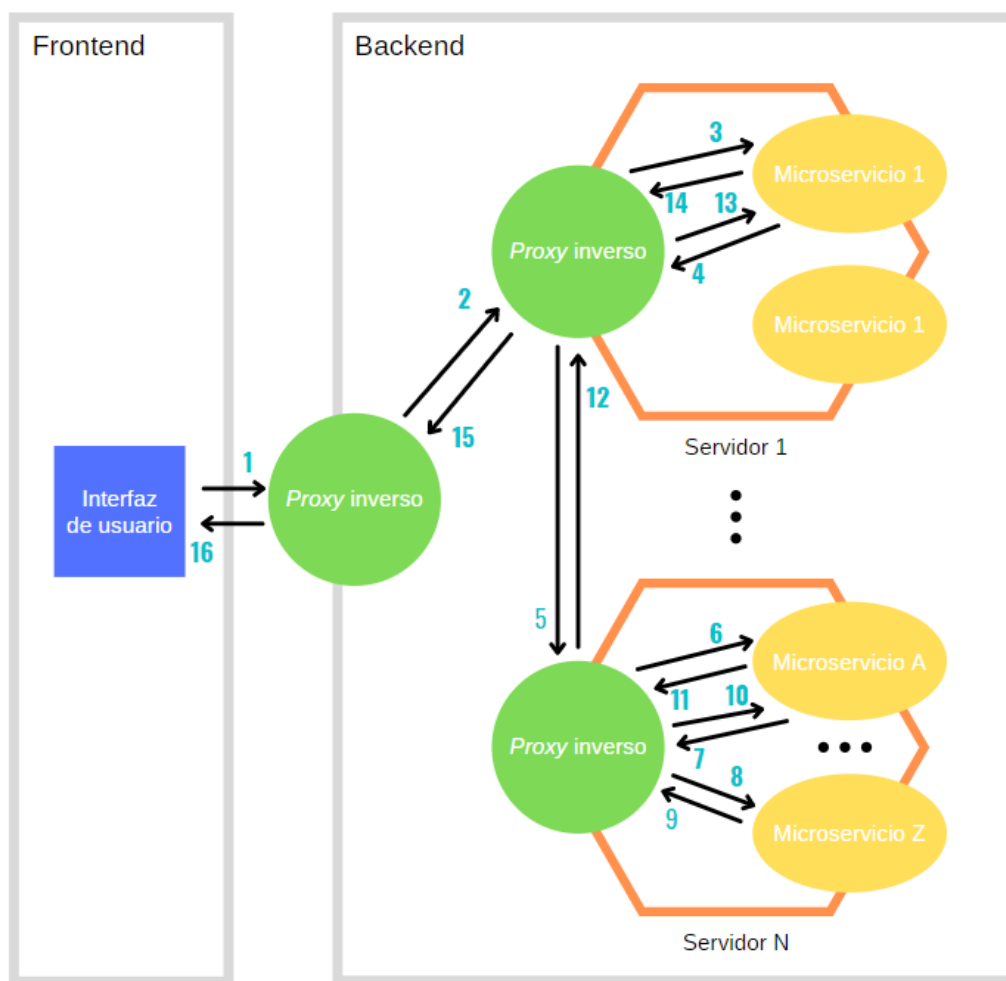


Figura 4.17: Esquema de interacción entre el *proxy* inverso y el resto de componentes de la aplicación al procesar una petición de la interfaz de usuario.

consultas Postman para realizar algunas de las pruebas, por lo que estas han sido actualizadas y se ha ido creando otras a medida que se han añadido nuevas características.

4.4.1. Pruebas de aceptación

Recordando lo comentado en el apartado 4.5, se han definido pruebas de aceptación (PA) para especificar muchas de las unidades de trabajo que representan el trabajo a realizar. Estas determinan el buen o mal funcionamiento de la característica descrita por la unidad de trabajo en cuestión por lo que se utilizan para comprobar si esta se ha implementado correctamente.

Una unidad de trabajo no se considera terminada hasta que no pasa satisfactoriamente las pruebas de aceptación que tiene asignadas. Así pues, las pruebas de aceptación han sido ejecutadas cuando se ha creído que la unidad de trabajo a la cual hacen referencia ha estado correctamente implementada y como paso previo a incluirla en el grupo de las terminadas. En caso de no obtener un resultado positivo, estas se pasaban de nuevo una vez corregido el problema, pudiéndose repetir este proceso más de una vez.

A continuación, se va a mostrar una parte de las pruebas de aceptación elaboradas partiendo del nombre de la unidad de trabajo que prueban. Cabe destacar que, además, se han automatizado algunas de las más importantes para facilitar así su ejecución. Se enseñará también el código de alguna de ellas.

Definición de algunas PAs

Las figuras 4.18, 4.19, 4.20, 4.21 y 4.22 son capturas de pantalla de las pruebas de aceptación de cuatro unidades de trabajo. Para cada PA se indica un nombre que la identifica, un primer punto con la acción o acciones que hay que llevar a cabo para ejecutarla y otro punto con el resultado esperado.

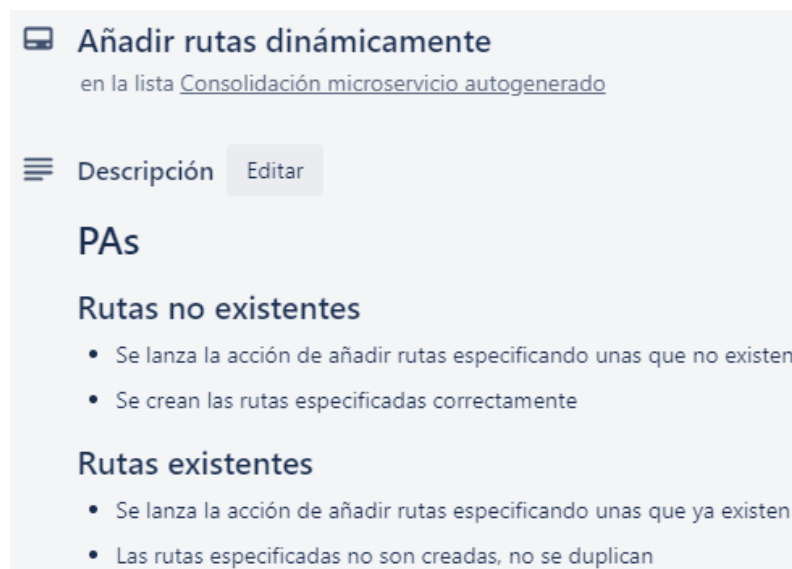


Figura 4.18: Pruebas de aceptación de la unidad de trabajo "Añadir rutas dinámicamente".

Pruebas automatizadas

Las figuras 4.23, 4.24 y 4.25 muestran el código de algunas de las pruebas de aceptación automatizadas. Es interesante mencionar que corresponden a PAs cuyas descripciones se han enseñado en la sección anterior, figuras 4.18, 4.19 y 4.20 en concreto. Así pues, se puede observar cómo se han automatizado y reflejado en código para entenderlas mejor.

4.4.2. Pruebas de regresión

Las pruebas de regresión son pruebas que se realizan sobre un producto *software* anteriormente probado al que se le han hecho modificaciones. Con ellas se pretende asegurarse de que no se han introducido defectos que puedan producir errores sobre las funcionalidades implementadas previamente.

En el caso del *proxy* inverso, han consistido en ejecutar todas las pruebas de aceptación, automatizadas y sin automatizar, de las unidades de trabajo ya terminadas. Se han ejecutado antes de cada despliegue y con ellas se ha verificado el correcto funcionamiento de todas las características del producto de cara a que los despliegues no hicieran fallar el *proxy* inverso ni cualquier otro componente de la aplicación. De esta forma, con estas pruebas se ha pretendido evitar todo tipo de problemas que pudieran entorpecer a los demás miembros del equipo de desarrollo de la aplicación ya que, si el *proxy* inverso quedara caído durante un tiempo, el resto de microservicios y la interfaz de usuario no podrían contactar con nadie y caerían también. Es por esto que se han considerado un seguro de vida para garantizar, en cierta medida, que los despliegues serían un éxito.



Figura 4.19: Pruebas de aceptación de la unidad de trabajo "Normalizar rutas base peticiones".

4.5 Metodología

Para llevar a cabo este desarrollo se ha seguido un enfoque ágil en el sentido de que se han seguido algunas de las prácticas propuestas por varias de las metodologías de este tipo, 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 dividido en tareas relativamente pequeñas denominadas unidades de trabajo. Estas definen funcionalidades más pequeñas y concretas que los casos de uso. También manifiestan de forma clara la posibilidad de tratarse de tareas no solo relacionadas con modificar el producto sino otras como crear una 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*. La única columna reseñable es la segunda, en la que primero se lleva a cabo una pequeña especificación de la unidad de trabajo en cuestión, principalmente definiendo sus pruebas de aceptación, y después, se implementa. 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. La herramienta utilizada a modo de tablero *kanban* ha sido Trello².
- **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

²Web oficial de Trello: <https://trello.com>.

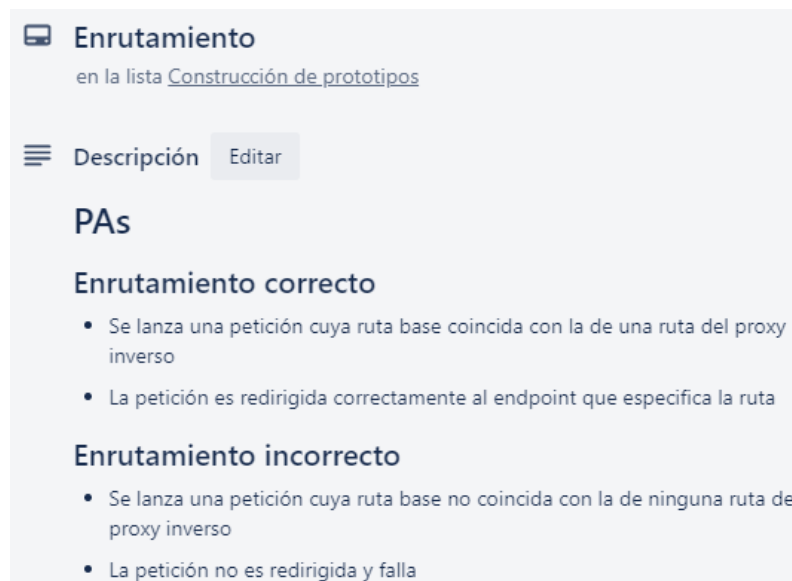


Figura 4.20: Pruebas de aceptación de la unidad de trabajo "Enrutamiento".

importancia 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 las unidades de trabajo correspondientes a la funcionalidad básica del *proxy* inverso. 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 4.4.1.
- **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 finalizadas 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 nueva entrega o despliegue. Las pruebas de regresión llevadas a cabo en este proyecto se comentarán en el apartado 4.4.2.



Figura 4.21: Pruebas de aceptación de la unidad de trabajo "Personalización timeouts".

4.5.1. 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 4.26, las cuales fueron obtenidas a partir de la especificación de requisitos del apartado 4.1. Cabe destacar que estas no han sido las definitivas y posteriormente se han modificado y creado nuevas, como es habitual en las metodologías ágiles.


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.

4.5.2. Cronología del proyecto

Los hitos de los que consta este proyecto han sido expuestos en orden cronológico a lo largo de esta memoria. A continuación serán comentados brevemente los principales para facilitar su visión de forma global y así entender la cronología del proyecto:

- **Estudio de tecnologías.** Uno de los primeros pasos para realizar este proyecto ha sido revisar las tecnologías disponibles y existentes para resolver problemas parecidos al planteado, esto corresponde al apartado 2.
- **Programación.** La fase de programación ha tenido, por su parte, 4 hitos principales, comentados en el punto 4.3: "Construcción de prototipos", "Consolidación del microservicio autogenerado", "Primeros despliegues" y "Producto final".
- **Revisión de la memoria.** Por último, se ha completado y revisado como conjunto la presente memoria, habiendo sido elaborada incrementalmente a la vez que se programaba.


Multiinstancia de microservicios
 en la lista [Producto final](#)


 Descripción

Editar

PA's

Balaneo de carga

- Se lanzan varias peticiones con la misma ruta base para que sean redirigidas por el proxy inverso. Debe existir una ruta con la ruta base de las peticiones y esta tiene que tener varias destinaciones
- Las peticiones son repartidas entre las diferentes destinaciones disponibles

Añadir rutas

- Se lanza la acción de añadir rutas especificando unas con la misma ruta base de peticiones que unas rutas que ya existen
- Las rutas no se duplican, se crean nuevas destinaciones para ellas correspondientes a las direcciones de las rutas especificadas

Eliminar rutas

- Se lanza la acción de eliminar rutas especificando unas con la misma ruta base de peticiones que unas rutas que ya existen, habiendo más de las segundas
- Solo se eliminan las rutas especificadas y no todas con las rutas base de peticiones especificadas

Figura 4.22: Pruebas de aceptación de la unidad de trabajo "Multiinstancia de microservicios".

```

/// <summary> The method: AddNewRoutes, when the routes do not exist, the routes ...
[TestMethod]
0 references | Alejandro Carrión Sanmartín, 32 days ago | 1 author, 3 changes
public void AddNewRoutes_RoutesNotExist_RoutesAreCreated()
{
    // Arrange.
    IServiceProvider serviceProvider = CocktailReverseProxyLogicTestsDependencyInjectionProvider.GetScopedServiceProvider();
    RoutesLogicManager routesLogicManager = serviceProvider.GetRequiredService<RoutesLogicManager>();
    InMemoryConfigurationProvider inMemoryConfigurationProvider = serviceProvider.GetRequiredService<InMemoryConfigurationProvider>();

    string microservice1RequestsBasePath = "Microservice1/api";
    string microservice1RegistryAddress = "http://localhost:4220";
    string microservice2RequestsBasePath = "Microservice2";
    string microservice2Address = "http://localhost:4125";

    // Act.
    routesLogicManager.AddNewRoutes(new List<NewRouteDTO>()
    {
        new NewRouteDTO()
        {
            RequestsBasePath = microservice1RequestsBasePath,
            Address = microservice1RegistryAddress,
        },
        new NewRouteDTO()
        {
            RequestsBasePath = microservice2RequestsBasePath,
            Address = microservice2Address,
        },
    });

    // Assert.
    LogicTestUtilities.CheckExistingRoutesNumber(inMemoryConfigurationProvider, 2);
    LogicTestUtilities.CheckExistingRoute(inMemoryConfigurationProvider, microservice1RequestsBasePath, microservice1RegistryAddress);
    LogicTestUtilities.CheckExistingRoute(inMemoryConfigurationProvider, microservice2RequestsBasePath, microservice2Address);
}

```

Figura 4.23: Prueba de aceptación automatizada de la unidad de trabajo "Añadir rutas dinámicamente".

```

/// <summary> The method: GetYarpNormalizedPath, using different requests base p ...
[TestMethod]
[DataRow("\\Microservice1\\api\\")]
[DataRow("Microservice1/api/")]
[DataRow("/Microservice1/api")]
[DataRow("//Microservice1//api//")]
0 references | Alejandro Carrión Sanmartín, 40 days ago | 1 author, 3 changes
public void GetYarpNormalizedPath_DifferentRequestsBasePaths_NormalizedCorrectly(string requestsBasePathToNormalize)
{
    // Act.
    string normalizedPath = RoutesUtils.GetYarpNormalizedPath(requestsBasePathToNormalize);

    // Assert.
    normalizedPath.Should().Be("/Microservice1/api/{*remainder}");
}

```

Figura 4.24: Pruebas de aceptación automatizadas de la unidad de trabajo "Normalizar rutas base peticiones".

```

/// <summary> The proxying system, when the route that will be tried exists, the ...
[TestMethod]
0 references | Alejandro Carrión Sanmartín, 25 days ago | 1 author, 6 changes
public async Task Proxying_RouteExists_ProxiesCorrectly()
{
    // Arrange.
    (TestServer microserviceWithHealthChecksSystemTestServer, string requestsBasePath) = GetMicroserviceWithHealthChecksSystemTestServer();

    TestServer reverseProxyTestServer = GetReverseProxyWithProxyingTestServer();

    CocktailReverseProxyProxyOnline cocktailReverseProxyProxyOnline = CocktailReverseProxyServicesTestUtilities.GetReverseProxyProxyFromTest

    await cocktailReverseProxyProxyOnline.AddNewRoutesAsync(
        new List<NewRouteDTO>()
        {
            new NewRouteDTO()
            {
                RequestsBasePath = requestsBasePath,
                Address = microserviceWithHealthChecksSystemTestServer.BaseAddress.ToString(),
            },
        },
        false);

    HttpClient reverseProxyHttpClient = reverseProxyTestServer.CreateClient();

    // Act.
    HttpResponseMessage httpResponseMessage = await reverseProxyHttpClient.GetAsync($"{requestsBasePath}/Health").ConfigureAwait(false);
    string responseContent = await httpResponseMessage.Content.ReadAsStringAsync().ConfigureAwait(false);

    // Assert.
    httpResponseMessage.StatusCode.Should().Be(HttpStatusCode.OK);
    responseContent.Should().Be(nameof(HealthCheckResult.Healthy));
}

/// <summary> The proxying system, when the route that will be tried does not ex ...
[TestMethod]
0 references | Alejandro Carrión Sanmartín, 40 days ago | 1 author, 2 changes
public async Task Proxying_RouteNotExist_EndpointNotFound()
{
    // Arrange.
    (TestServer _, string requestsBasePath) = GetMicroserviceWithHealthChecksSystemTestServer();

    TestServer reverseProxyTestServer = GetReverseProxyWithProxyingTestServer();

    HttpClient reverseProxyHttpClient = reverseProxyTestServer.CreateClient();

    // Act.
    HttpResponseMessage httpResponseMessage = await reverseProxyHttpClient.GetAsync($"{requestsBasePath}/Health").ConfigureAwait(false);

    // Assert.
    httpResponseMessage.StatusCode.Should().Be(HttpStatusCode.NotFound);
}

```

Figura 4.25: Pruebas de aceptación automatizadas de la unidad de trabajo "Enrutamiento".

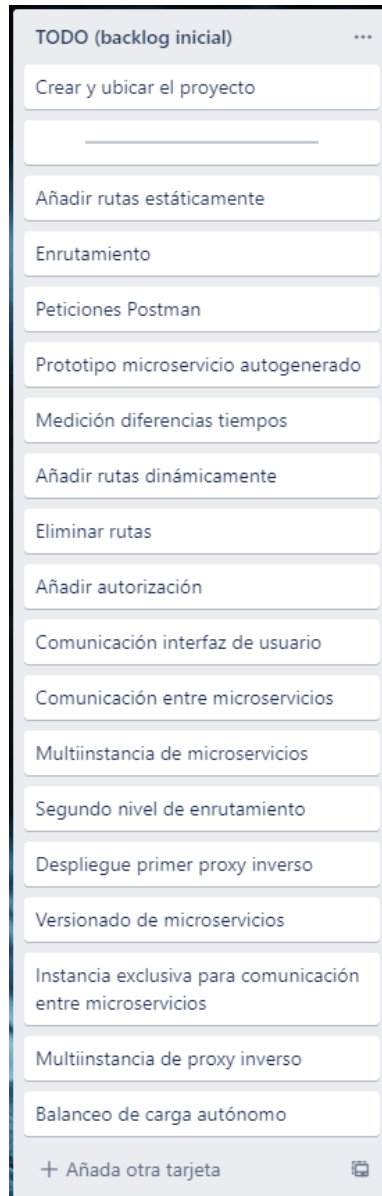


Figura 4.26: *Backlog* con las unidades de trabajo iniciales.

Conclusiones y trabajo futuro

El resultado de este proyecto es un *proxy* inverso totalmente funcional, capaz de cubrir las necesidades planteadas y actuar correctamente en los casos de uso para los que fue concebido. De esta manera, todos los requisitos, tanto funcionales como no funcionales, han sido cubiertos satisfactoriamente. A pesar de ello, es necesario decir que algunas funcionalidades se podrían haber mejorado si no fuera por el hecho de que las partes de la aplicación con las que se integra el *proxy* inverso no están acabadas al tratarse de una aplicación en desarrollo. Por ejemplo, el microservicio encargado de orquestar los despliegues no está cien por cien operativo todavía.

Desde otra perspectiva, el producto resultante también cumple los objetivos inicialmente definidos. El *proxy* inverso logra ocultar los microservicios, haciendo que la interfaz de usuario no conozca donde se encuentran y no acceda directamente a ellos. Permite, además, la ejecución de más de una instancia de un mismo microservicio, otorgando así cierta tolerancia a fallos y aumentando la eficiencia de la aplicación.

A pesar de haber cumplido los objetivos, es interesante hacer una breve reflexión para reconocer los aspectos negativos del producto desarrollado. Así pues, el *proxy* inverso constituye un nuevo punto único de fallo, además de tener la posibilidad de actuar como cuello de botella, ralentizando las peticiones de toda la aplicación. Sin embargo, estos dos aspectos se han combatido permitiendo la existencia de más de una instancia del *proxy* inverso. Por otra parte, un asunto al cual no se le ha dado solución es el hecho de que los *middleware* construidos para la *pipeline* de YARP aumentan el tiempo de respuesta de las peticiones. No obstante, no se supera el límite definido por el requisito no funcional RNF02, en el apartado 4.1.3.

En lo referente a los conocimientos necesarios para la ejecución de este proyecto, es interesante comentar que el lenguaje de programación utilizado, C#, es visto en los estudios en los que se enmarca este trabajo, si bien, no con la profundidad suficiente para poder abordar un problema de estas características. Además, no es lo mismo hacer programas de escritorio con Windows Forms [21] como *framework*, que es lo enseñado en ellos principalmente, que aplicaciones de tipo servicio web. Por otra parte, las asignaturas de metodología *software*, de sistemas en red y de modelado *software* también han resultado una muy buena base. Las primeras han servido para establecer y seguir una disciplina de desarrollo acorde a las características del proyecto y a su contexto. Las segundas, para entender mejor el funcionamiento de las comunicaciones entre varios servicios y las del propio *proxy* inverso. Las terceras, para entender y dominar el proceso de generación automática de código a partir de modelos seguido. Por último, YARP es una librería con un propósito muy específico, por lo que su aprendizaje ha requerido cierto esfuerzo, sobre todo para poder utilizar algunas de sus características más avanzadas.

Por lo que se refiere a experiencia personal, ha sido muy gratificante llevar a cabo este trabajo. Por un lado, ha servido como ensayo para la futura resolución de problemas mayores en el mundo laboral, aunque también se pueda interpretar este como un trabajo profesional al haberse realizado en el contexto de una empresa. Por otro, ha merecido la pena este último hecho, pues ha otorgado cierta confianza dentro de la propia organización y se cree que con un proyecto real el proceso de aprendizaje ha sido más fructífero, teniendo en cuenta las desventajas que conlleva.

En cuanto a líneas de trabajo futuras, el *proxy* inverso quizás requiera alguna nueva característica o modificación que otra a medida que el resto de la aplicación avance y surjan nuevas necesidades. Para finalizar, también se puede querer extender la funcionalidad más adelante y construir encima un *API Gateway* que ofrezca características específicas de ese patrón. Estas pueden ser agregaciones de peticiones o tareas transversales tales como monitorización de tiempos de las peticiones.

Referencias

- [1] ¿Qué es DevOps? (consultado en 08/2021):
<https://azure.microsoft.com/es-es/overview/what-is-devops>.
- [2] *Continuous Delivery*. M. Fowler, 2013 (consultado en 08/2021):
<https://martinfowler.com/bliki/ContinuousDelivery.html>
- [3] *Microservices*. J. Lewis y M. Fowler, 2014 (consultado en 08/2021):
<https://martinfowler.com/articles/microservices.html>
- [4] *Building Microservices*. S. Newman, O'REILLY, 2015. ISBN: 9781491950357
- [5] Documentación oficial de ASP.NET Core (consultado en 08/2021):
<https://docs.microsoft.com/es-es/aspnet/core/?view=aspnetcore-5.0>
- [6] Uso de puertas de enlace de API (API Gateway) en microservicios (consultado en 08/2021):
<https://docs.microsoft.com/es-es/azure/architecture/microservices/design/gateway>
- [7] Web oficial de NGINX (consultado en 08/2021):
<https://www.nginx.com>
- [8] Documentación oficial de NGINX para funcionar como *proxy* inverso (consultado en 08/2021):
<https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy>
- [9] Documentación oficial de Ocelot (consultado en 08/2021):
<https://ocelot.readthedocs.io>
- [10] ¿Qué es el modelo OSI? (consultado en 08/2021):
<https://www.ionos.es/digitalguide/servidores/know-how/el-modelo-osi-un-referente-para-normas-y-protocolos>
- [11] El método *Configure* y la *pipeline* de *middlewares* (consultado en 08/2021):
<https://docs.microsoft.com/es-es/aspnet/core/fundamentals/startup?view=aspnetcore-5.0#the-configure-method>
- [12] Documentación oficial de .NET (consultado en 08/2021):
<https://docs.microsoft.com/es-es/dotnet>
- [13] Documentación oficial de C# (consultado en 08/2021):
<https://docs.microsoft.com/es-es/dotnet/csharp>
- [14] Una introducción a NuGet (consultado en 08/2021):
<https://docs.microsoft.com/es-es/nuget/what-is-nuget>

-
- [15] *System and software Quality Requirements and Evaluation*, ISO/IEC 25010 (consultado en 08/2021):
<https://iso25000.com/index.php/normas-iso-25000/iso-25010>
- [16] *Data Transfer Object*. M. Fowler, 2002 (consultado en 08/2021):
<https://martinfowler.com/eaCatalog/dataTransferObject.html>
- [17] ¿Qué son las soluciones y los proyectos en Visual Studio? (consultado en 08/2021):
<https://docs.microsoft.com/es-es/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2019>
- [18] Tareas en segundo plano con servicios hospedados en ASP.NET Core (consultado en 08/2021):
<https://docs.microsoft.com/es-es/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-5.0&tabs=visual-studio>
- [19] Implementación del servidor web Kestrel en ASP.NET Core (consultado en 08/2021):
<https://docs.microsoft.com/es-es/aspnet/core/fundamentals/servers/kestrel>
- [20] *What is Alpha Testing?* (consultado en 08/21):
<https://www.guru99.com/alpha-testing.html>
- [21] Guía de escritorio, .NET para Windows Forms (consultado en 08/2021):
<https://docs.microsoft.com/es-es/dotnet/desktop/winforms/overview>
- [22] Modelar el SDK de Visual Studio - Lenguajes específicos de dominio (consultado en 08/2021):
<https://docs.microsoft.com/es-es/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2019>
- [23] ¿Qué es PowerShell? (consultado en 08/2021):
<https://docs.microsoft.com/es-es/powershell/scripting/overview?view=powershell-7.1>
- [24] *SOLID: The First 5 Principles of Object Oriented Design* (Consultado en 08/2021):
https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- [25] Clases y métodos parciales, Guía de programación de C# (consultado en 08/2021):
<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods>

APÉNDICE A

Proceso de generación automática de código

A lo largo de toda la memoria se ha mencionado el hecho de que la aplicación de la que forma parte el *proxy* inverso, y finalmente él mismo también, se ha construido utilizando técnicas de generación automática de código a partir de modelos. Sin embargo, no se ha entrado en mucho más detalle ya que no se ha considerado imprescindible y no se ha querido desviar la atención del producto desarrollado. De esta forma, este apéndice está destinado a complementar la información dada hasta ahora, explicando y dando ejemplos para el caso particular, del proceso de generación automática de código a partir de modelos que se ha seguido para elaborar el *proxy* inverso. Cabe destacar que para la realización de este apéndice se ha consultado documentación interna del equipo de desarrollo con la intención de enriquecer el mismo.

Primero de todo, es conveniente explicar qué es generar código de forma automática a partir de modelos. Consiste en autogenerar, a partir de la especificación de ciertos modelos y en base a unas plantillas preconfiguradas, todo el código posible que se repita o que pueda ser escrito de forma automática debido a su poca o escasa variabilidad. En el marco de esta descripción entra el código encargado de comunicarse con la base de datos o el que prepara las acciones HTTP que un servicio expone al exterior, por ejemplo, pues siempre se debe hacer una forma concreta y establecida. Algunas de las ventajas más relevantes que esta aproximación de desarrollo de *software* ofrece son:

- Permite reducir los tiempos de desarrollo debido a la generación automática de gran parte del código.
- Reduce los errores humanos en código repetitivo.
- El desarrollo de acciones CRUD (*create*, *read*, *update* y *delete*), utilizadas en prácticamente todas las aplicaciones informáticas, es prácticamente cero al tratarse de código reiterativo que se puede automatizar.

Como puntos negativos destacar, sobre todo, la dificultad de creación y mantenimiento de las herramientas utilizadas para generar el código. También puede resultar complejo cuadrar el código autogenerado con la programación de algunos flecos o particularidades de una aplicación concreta.

El proceso de generación automática utilizado para este trabajo se divide en 3 pasos que se repiten continuamente en un proceso iterativo e incremental. Estos son 'Modelado', 'Generación de código' y 'Programación de particularidades'. Pasan a ser detallados a continuación.

A.1 Modelado

De forma general, el modelado consiste en construir diagramas que representen una abstracción o simplificación de la realidad. Estos diagramas o modelos permiten especificar aspectos de un sistema *software* tales como requisitos, estructura o comportamiento. En este caso, se utilizan para detallar principalmente las entidades del dominio de la aplicación y las acciones del *backend* denominadas acciones ad hoc. Para realizar estas especificaciones, se hace uso de las llamadas *DSL Tools*. Estas son unas herramientas de modelado para Visual Studio que han sido creadas internamente a partir del soporte que da Microsoft para ello [22] y que están perfectamente adaptadas al proceso seguido de generación automática de código.

Primero de todo, es necesario construir los modelos de dominio. Estos son muy parecidos a los típicos diagramas de entidades UML y en ellos se representan las entidades relevantes del dominio de la aplicación, con las correspondientes propiedades y relaciones entre ellas. En segundo lugar, se elaboran los modelos de aplicación. En ellos se modelan las acciones ad hoc, es decir, las acciones de *backend* que el microservicio va a realizar. Para cada una de ellas es posible especificar un DTO de entrada y uno de salida, ambos opcionales, que encapsulan los datos correspondientes. Adicionalmente, también se puede especificar otros DTO para su uso interno u otras situaciones más específicas.

Centrando la vista en el modelado del *proxy* inverso, se ha definido un único modelo de dominio, el cual se encuentra vacío. Esto está relacionado con el hecho de que no necesita persistir nada. Además, la representación interna que utiliza para guardar las rutas durante la ejecución es una interfaz que expone la propia librería de YARP, ya que ella misma lo gestiona, por lo que no ha sido necesario crear ninguna entidad. Con lo referente al modelo de aplicación, se han creado dos acciones ad hoc, figura A.1. La primera corresponde al caso de uso de añadir rutas (CU01) y la segunda al de eliminarlas (CU02). Resaltar la existencia de los DTO de entrada, con la información necesaria para realizar las acciones, pero no de salida. Otra curiosidad es el asterisco que aparece al lado de las relaciones entre los DTO de entrada y las acciones, que indica que estas reciben múltiples DTO de entrada, es decir, una colección de ellos.

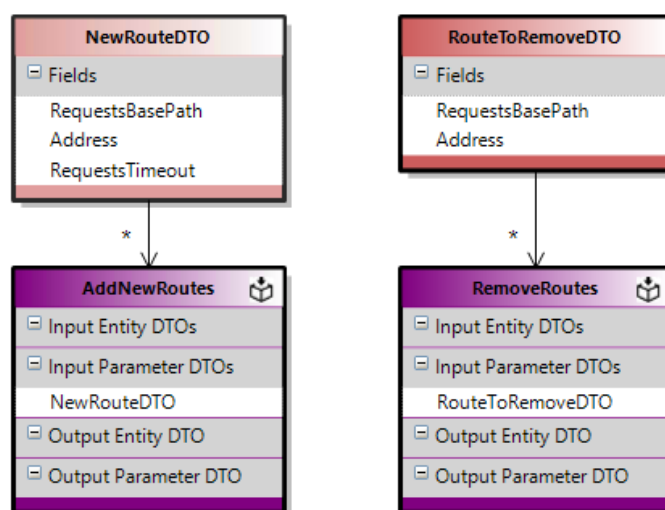


Figura A.1: Acciones ad hoc del *proxy* inverso.

Aparte de los modelos comentados de dominio y de aplicación, existen los de interfaz de usuario. Estos permiten construir formularios utilizando unos patrones preestablecidos y haciendo uso de unos DTO específicos que indican los campos a representar en

ellos. Sin embargo, el *proxy* inverso se trata de una aplicación de tipo servicio web que no requiere interfaz de usuario, por lo que no han sido mencionados antes ni se entrará en más detalle. En general, se puede decir que el modelado para el *proxy* inverso es muy simple, nada comparable con el de microservicios más complejos.

A.2 Generación de código

La generación de código consiste en ejecutar el generador de código. Este utiliza unas plantillas con condiciones y patrones que, a partir de los modelos, genera el código correspondiente. Por otra parte, se ejecuta mediante un *script* de PowerShell [23] que facilita esta tarea.

Con este proceso se genera buena parte de la infraestructura de un microservicio. Esto comprende todas las capas de la arquitectura explicada en el apartado 4.2.1, que trata de seguir siempre buenos principios de programación como los SOLID [24].

Es interesante mencionar que todas las clases autogeneradas se crean con un comentario a modo de encabezado indicando que no se trata de una clase normal (figura A.2). Esto se hace para indicar a los programadores que no debe ser modificada a mano ya que los cambios desaparecerán al generar código de nuevo.

```
//-----  
// <auto-generated>  
//   This code was generated by a tool.  
//   Runtime Version:4.0.30319.17626  
//  
//   Changes to this file may cause incorrect behaviour and will be  
//   lost if the code is regenerated.  
// </auto-generated>  
//-----
```

Figura A.2: Encabezado de código autogenerado.

A.3 Programación de particularidades

Para la elaboración de una aplicación cualquiera no todo el código se puede generar automáticamente. Cada una de ellas tiene cierto comportamiento específico por lo que es necesaria la intervención humana para programar aspectos concretos tales como las acciones ad hoc.

En el caso de este proceso de generación automática de código la mayoría de flecos o particularidades, por no decir todos, son implementados a través de clases parciales de C# [25]. La generación de código crea la clase y el programador tiene que crear otro archivo, con el sufijo "*Partial*", para implementar la clase parcial. Por otro lado, el generador de código se adapta y reconoce si una determinada clase parcial está creada o un determinado método gancho implementado para hacer las llamadas correspondientes o añadir o eliminar un determinado código para que todo encaje y funcione correctamente. Un ejemplo de clase parcial para el caso del *proxy* inverso se muestra en la figura A.3. Corresponde a un código llamado por la clase *Startup* para añadir dependencias y la clase parcial permite añadir las dependencias manuales que se crean para tareas como implementar las acciones ad hoc.

```

/// <summary> Cocktail Reverse Proxy Logic service collection extensions. It is ...
1 reference | Alejandro Carrión Sanmartín, 117 days ago | 1 author, 1 change
public static partial class CocktailReverseProxyApplicationLogicServiceCollectionExtensions
{
    /// <summary> An IServiceCollection extension method that adds the Cocktail Reve ...
    2 references | Alejandro Carrión Sanmartín, 117 days ago | 1 author, 1 change
    public static IServiceCollection AddCocktailReverseProxyApplicationLogic(this IServiceCollection services, IConfiguration configuration)
    {
        services.AddLoggingLogic();

        // This code will not compile unless an extension method with this name is created in a partial class.
        // Here is the template for the method.
        /*
            /// <summary>
            ///     An <see cref="IServiceCollection"/> extension method that adds the Cocktail
            ///     Reverse Proxy Logic dependencies specific for the application.
            /// </summary>
            /// <param name="services"> The services collection to act on. </param>
            /// <param name="configuration">
            ///     The configuration for configuring the services.
            /// </param>
            /// <returns>
            ///     A service collection with the Logic dependencies specific for the application.
            /// </returns>
            private static IServiceCollection AddCocktailReverseProxyApplicationLogicSpecificDependencies(this IServiceCollection services)
            {
                return services;
            }
        */

        services.AddCocktailReverseProxyApplicationLogicSpecificDependencies();
    }
}

```

```

/// <summary> Cocktail Reverse Proxy Logic service collection extensions. It is ...
1 reference | Alejandro Carrión Sanmartín, 71 days ago | 1 author, 5 changes
public static partial class CocktailReverseProxyApplicationLogicServiceCollectionExtensions
{
    /// <summary> An IServiceCollection extension method that adds the Cocktail Reve ...
    1 reference | 0 changes | 0 authors, 0 changes
    private static IServiceCollection AddCocktailReverseProxyApplicationLogicSpecificDependencies(this IServiceCollection services)
    {
        services.AddScoped<RoutesLogicManager>();

        services.AddHostedService<RoutesLoaderHostedService>();

        services.AddSingleton<SameMachineFilter>();
        services.AddSingleton<CustomProxyLog>();

        return services;
    }
}

```

Figura A.3: Ejemplo de clase parcial del *proxy* inverso.