UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de un microservicio para el despliegue de servidores WASM

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Garcia Gonzalez, Izan

Tutor/a: Letelier Torres, Patricio Orlando

CURSO ACADÉMICO: 2024/2025

Dedicatoria

A mis padres, por ser mi ejemplo de esfuerzo, constancia y amor incondicional.
A mi pareja, por su compañía, paciencia y apoyo en cada paso de este camino.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que hicieron posible este trabajo. A cada una que, de una u otra forma, me acompañó en este camino: quienes me guiaron profesionalmente y también quienes me ofrecieron su apoyo personal en los momentos más difíciles. Sin vosotros, nada de esto habría sido posible.

En primer lugar, a mi tutor, Patricio Letelier. Gracias por ser mucho más que un profesor: por tu dedicación, por enseñarnos con honestidad y compromiso, por mostrarnos la realidad del mundo profesional, con sus luces y sus sombras.

A mi familia, que siempre ha estado ahí, apoyándome en cada decisión, dándome libertad para ser yo mismo, para equivocarme, aprender y crecer. Gracias por cuidar de mí sin atarme, por confiar incluso cuando yo dudaba.

A mi novia, Ari, que me ha iluminado hasta los días más oscuros. Gracias por tu risa oportuna, por tu paciencia infinita y por todos esos momentos compartidos que se quedan conmigo, grabados en lo más profundo. Contigo, hasta lo difícil se volvía más llevadero.

A mis amigos del grupo de "Históricos", compañeros de viaje durante estos intensos y bonitos cuatro años. Gracias por la sinceridad, el respeto y el cariño. En tiempos donde todo parece tan efímero, encontrar un grupo así es un regalo.

Y por último, pero no menos importante, a todo el equipo de I+D+i de ADD Informática. Profesionales excepcionales que no solo me ayudaron a aprender, sino que también me hicieron sentir parte de algo. Gracias por vuestra paciencia, generosidad y todos los buenos momentos compartidos.

Resum

El desplegament de microserveis es realitza habitualment mitjançant tecnologies com Docker i Kubernetes, on Docker s'encarrega d'encapsular cada microservei en contenidors, proporcionant entorns aïllats pràcticament independents del sistema operatiu, mentre que Kubernetes actua com el seu orquestrador.

La tecnologia *WebAssembly* (WASM) permet executar codi compilat directament en el navegador del client, millorant així el rendiment i l'eficiència de les aplicacions web. En el marc d'aquest TFG, WebAssembly s'ha emprat per encapsular la interfície d'usuari (UI) dels microserveis, facilitant així el seu posterior desplegament.

L'objectiu d'aquest treball és desplegar els servidors WebAssembly corresponents a la part *frontend* d'un microservei. Aquests es desplegaran mitjançant un microservei prou genèric com per gestionar el desplegament d'altres microserveis, amb la finalitat de reemplaçar Docker i Kubernetes, oferint així una solució que no depenga de tecnologies externes.

Aquest treball es du a terme en el context d'una pràctica en una empresa de desenvolupament de *software* que proporciona solucions tecnològiques per al sector sociosanitari a Espanya. En particular, l'empresa ofereix un sistema de planificació de recursos empresarials (ERP) especialitzada en la gestió de residències de majors. Per a això, s'empra una arquitectura basada en microserveis, la qual cosa ressalta la importància del microservei desenvolupat en aquest projecte, ja que serà el responsable de gestionar el desplegament dels productes per als clients.

Paraules clau: Microserveis, WebAssembly, Desplegament, Desenvolupament de software basat en components, Docker, Kubernetes

Resumen

El despliegue de microservicios se realiza habitualmente mediante tecnologías como Docker y Kubernetes, donde Docker es la encargada de encapsular cada microservicio en contenedores, proporcionando entornos aislados prácticamente independientes del sistema operativo, mientras que Kubernetes actúa como su orquestador.

La tecnología *WebAssembly* (WASM) permite ejecutar código compilado directamente en el navegador del cliente, mejorando así el rendimiento y la eficiencia de las aplicaciones web. En el marco de este TFG, WebAssembly se ha empleado para encapsular la interfaz de usuario (UI) de los microservicios, facilitando así su posterior despliegue.

El objetivo de este trabajo es desplegar los servidores WebAssembly correspondientes a la parte *frontend* de un microservicio. Estos se desplegarán mediante un microservicio lo suficientemente genérico como para gestionar el despliegue de otros microservicios, con el propósito de reemplazar Docker y Kubernetes, ofreciendo así una solución que no dependa de tecnologías externas.

Este trabajo se lleva a cabo en el contexto de una práctica en una empresa de desarrollo de *software* que proporciona soluciones tecnológicas para el sector sociosanitario en España. En particular, la empresa ofrece un sistema de planificación de recursos empresariales (ERP) especializada en la gestión de residencias de mayores. Para ello, se emplea

una arquitectura basada en microservicios, lo que resalta la importancia del microservicio desarrollado en este proyecto, dado que será el responsable de gestionar el despliegue de los productos para los clientes.

Palabras clave: Microservicios, WebAssembly, Despliegue, Desarrollo de software basado en componentes, Docker, Kubernetes

Abstract

The deployment of microservices is commonly carried out using technologies such as Docker and Kubernetes, where Docker is responsible for encapsulating each microservice into containers, providing isolated environments that are practically independent of the operating system, while Kubernetes acts as their orchestrator.

WebAssembly (WASM) technology allows compiled code to run directly in the client's browser, thereby improving the performance and efficiency of web applications. In the context of this Bachelor's Thesis, WebAssembly has been used to encapsulate the user interface (UI) of the microservices, thus facilitating their subsequent deployment.

The aim of this project is to deploy WebAssembly servers corresponding to the frontend part of a microservice. These will be deployed through a custom microservice that is generic enough to manage the deployment of other microservices, with the goal of replacing Docker and Kubernetes, thereby offering a solution that does not rely on external technologies.

This work is carried out within the framework of an internship at a software development company that provides technological solutions for the social and healthcare sector in Spain. In particular, the company offers an enterprise resource planning (ERP) system specialized in the management of nursing homes. To this end, an architecture based on microservices is used, which highlights the importance of the microservice developed in this project, as it will be responsible for managing the deployment of the products for the clients.

Key words: Microservices, WebAssembly, Deployment, Component-based software development, Docker, Kubernetes

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

1.1 Motivación

Un microservicio [4] es un componente software que proporciona una funcionalidad específica como servicio. Entre sus principales ventajas se encuentran la escalabilidad y la modularidad, sin embargo, su ventaja clave radica en la posibilidad de ser desplegado de forma independiente.

El despliegue de microservicios es un concepto que, a priori, puede parecer sencillo, especialmente en el contexto del software actual. Existen numerosas herramientas y soluciones de código abierto que permiten, con un solo click, desplegar aplicaciones completamente contenidas, con gestión automática de balanceo de carga. Si bien esto resulta útil para aplicaciones de pequeña escala o aquellas desarrolladas desde cero, el panorama se vuelve más complejo en el ámbito empresarial.

En este contexto, surgen desafíos como la integración con aplicaciones heredadas, software embebido, tecnologías en desuso, requisitos específicos del cliente y altos costos tecnológicos. A medida que estos factores entran en juego, la complejidad del despliegue aumenta significativamente. En consecuencia, las soluciones que inicialmente parecían adecuadas para la implementación y gestión de microservicios pueden volverse insuficientes debido a sus limitaciones.

El presente trabajo se ha desarrollado dentro del ámbito de una empresa de desarrollo de software para el sector sociosanitario español. Esta compañía desarrolla y comercializa un ERP especializado en la gestión de residencias de mayores. El autor de este proyecto ha contribuido en este contexto desarrollando e implementando soluciones para el despliegue e integración continúa del ciclo de vida del software de la empresa. La motivación de este proyecto ha sido conseguir desplegar a producción todos los microservicios necesarios para ofrecer un producto profesional a los clientes, mediante un microservicio que se encarga, junto con otros componentes, de automatizar y controlar dicho proceso.

1.2 Objetivos

Este TFG tiene como objetivo principal la puesta en marcha del frontend de los microservicios, es decir, de los servidores WebAssembly[1], los cuales encapsulan la interfaz de usuario de cada uno de los microservicios propios de la empresa. Para ello, se ha desarrollado un microservicio encargado de desplegar otros microservicios, proporcionando

[1] Página web oficial de WebAssembly: https://webassembly.org

funcionalidades esenciales como tolerancia a fallos, balanceo de carga y gestión automática de actualizaciones, entre otras.

El principal desafío ha consistido en implementar un mecanismo genérico y reutilizable que pudiera aplicarse de forma uniforme a todos los microservicios, dentro de una solución con más de siete años de desarrollo acumulado y con un volumen superior a tres millones de líneas de código.

De forma más concreta, se plantean los siguientes objetivos específicos:

1. Diseñar una solución para el despliegue de microservicios que no dependa de tecnologías externas como Docker y Kubernetes.
2. Definir los componentes de software involucrados en el proceso de despliegue de microservicios.
3. Proponer una arquitectura de despliegue que permita la publicación y ejecución eficiente de interfaces de usuario (UI) basadas en WebAssembly.
4. Garantizar que la solución propuesta permita el despliegue completo de un conjunto de microservicios en una única máquina cliente, incluyendo características como tolerancia a fallos y actualizaciones en caliente.

1.3 Estructura de la memoria

A continuación se presenta la estructura de la memoria:

- El primer capítulo introduce la memoria a través de la motivación del trabajo, los objetivos a cumplir, y la presentación de la estructura de la memoria.
- El segundo capítulo presenta el estado del arte en el contexto del despliegue de microservicios y expone las tecnologías habitualmente utilizadas, para acabar con una comparativa en la que se incluye la solución propuesta por este trabajo.
- En el tercer capítulo se explican las tecnologías finalmente utilizadas, justificando su elección y mostrando ejemplos básicos de su uso.
- El cuarto capítulo es el más extenso de todos y describe de manera detallada el desarrollo de la solución llevada a cabo, dando importancia a la metodología utilizada y la especificación de requisitos tanto funcionales como no funcionales. Un apartado extenso sobre el diseño, programación, implementación, pruebas y despliegue final.
- El quinto capítulo muestra la cronología durante la escritura y el desarrollo de este TFG, con el fin de justificar el tiempo invertido.
- En el sexto y último capítulo se realiza una reflexión sobre el trabajo realizado y se presentan los próximos pasos a realizar.

Como recurso adicional, se presenta en forma de anexo, los Objetivos de Desarrollo Sostenible y su grado de relación con el presente trabajo, también se encuentra una breve reflexión de los mismos.

CAPÍTULO 2
Estado del arte

En la actualidad, el proceso de despliegue no es responsabilidad de una única aplicación o componente, ni se trata de un procedimiento sencillo. Se compone de múltiples etapas que, en conjunto, ofrecen al cliente la impresión de un único producto desplegado. Sin embargo, detrás de escena se consideran numerosos factores, como la seguridad en la transferencia de datos, el consumo de memoria RAM, el tiempo de respuesta de las peticiones, la tolerancia a fallos, la concurrencia, entre otros.

El despliegue debe entenderse como un proceso integral que no solo busca poner un producto en funcionamiento, sino que también otorga relevancia a los pasos intermedios, asegurando que estos sean seguros, repetibles y medibles[4]. Es posible afirmar con certeza que el despliegue abarca múltiples ámbitos del desarrollo de software, tales como el control de versiones, la publicación de artefactos, las pruebas automatizadas, los entornos de prueba, la gestión de infraestructura como código (IaC[1]), el aislamiento de procesos, la monitorización de recursos, la gestión de fallos y el tratamiento de logs, entre otros.

Dado que se trata de un concepto amplio, las herramientas que lo acompañan también lo son. No obstante, en este proyecto nos centraremos únicamente en aquellas herramientas que se pretende reemplazar. Otras funcionalidades, como la gestión de logs mediante un microservicio propio de telemetría, la medición de recursos a través de un servicio hospedado en la máquina del cliente o la monitorización mediante una interfaz gráfica personalizada, serán mencionadas y descritas brevemente, ya que desarrollar cada una en profundidad excedería el alcance de este trabajo.

2.1 Tecnologías existentes

A continuación se presentan las tecnologías existentes, con especial énfasis en su arquitectura y en la descripción de los componentes que las conforman. Es fundamental comprender la función y las responsabilidades de cada uno de estos elementos, ya que presentan similitudes con las funcionalidades que se integrarán en el microservicio personalizado de despliegue propuesto en este trabajo.

[1] *Infrastructure as Code: Es un enfoque que consiste en gestionar y aprovisionar la infraestructura de los sistemas mediante archivos de código legibles y versionables, en lugar de recurrir a configuraciones manuales.*

2.1.1. WiX Toolset

Windows Installer XML [2] (WiX) es un conjunto de herramientas diseñado para el desarrollo de instaladores para sistemas Windows, específicamente archivos Microsoft Installer (.msi) [2]. Se basa en el uso del lenguaje XML para definir las propiedades y acciones que deben ejecutarse durante el proceso de instalación de una aplicación.

Estas herramientas son empleadas para encapsular los microservicios en un único archivo autocontenido, facilitando así su distribución e instalación en equipos cliente. Proporciona la infraestructura necesaria para instalar un microservicio, incluyendo la modificación de permisos o privilegios dentro del SO, la creación de tareas programadas, la instalación y configuración de bases de datos, entre otras acciones.

Para la encapsulación de microservicios, se utiliza una herramienta incluida en este conjunto denominada Burn, la cual permite la creación de *bundles*. Un *bundle* actúa como una receta de instalación, en la que se definen pasos previos, como la instalación de .NET Framework o Visual C++ Runtime, así como pasos posteriores, como la eliminación de archivos temporales u otras tareas de limpieza.

En la figura 2.1 se muestra la secuencia de pasos que se llevan a cabo durante el proceso de instalación. Entre ellos se incluye la verificación del paquete *NetFx472Redist*, correspondiente al *runtime* de .NET Framework 4.7.2 para aplicaciones Windows, así como la definición del componente principal, que en este caso se identifica como *EnglishSetup*, correspondiente a uno de los microservicios.

```
<!-- Define the list of chained packages. -->
<Chain>
  <PackageGroupRef Id="NetFx472Redist" />
  <RollbackBoundary />
  <!-- The SourceFile of the NetHostingRedist package automatically changes in the pre-build event of the
  bootstrapper project. Should not be changed manually. -->
  <ExePackage Id="NetHostingRedist" SourceFile="\\addhqrdrsfil001\Downloads\NET\net9.0\Latest\dotnet-hosting-9.0.
  0-win.exe" LogPathVariable="NetHostingRedistLog" Permanent="yes" InstallArguments="/q /norestart
  OPT_NO_RUNTIME=1 OPT_NO_SHAREDFX=1 /log &quot;[NetHostingRedistLog].txt&quot;" DetectCondition="">
    <!-- Ignore this error indicating that another version of the product is installed. A higher version of the
    product is valid. -->
    <ExitCode Value="1638" Behavior="success" />
  </ExePackage>
  <RollbackBoundary />
  <MsiPackage Id="EnglishSetup" SourceFile="$(var.EnglishSetupPath)">
    <MsiProperty Name="DATABASELOGIN" Value="[DATABASELOGIN]" />
    <MsiProperty Name="DATABASEPASSWORD" Value="[DATABASEPASSWORD]" />
    <MsiProperty Name="INFRAUSERID" Value="[INFRAUSERID]" />
    <MsiProperty Name="INFRAUSERNAME" Value="[INFRAUSERNAME]" />
    <MsiProperty Name="INFRAUSERPASSWORD" Value="[INFRAUSERPASSWORD]" />
    <MsiProperty Name="INFRAUSERNEWPASSWORD" Value="[INFRAUSERNEWPASSWORD]" />
    <MsiProperty Name="SKIPCONFIGURINGEMAILSERVER" Value="[SKIPCONFIGURINGEMAILSERVER]" />
    <MsiProperty Name="EMAILSERVER" Value="[EMAILSERVER]" />
    <MsiProperty Name="EMAILSERVERPORT" Value="[EMAILSERVERPORT]" />
    <MsiProperty Name="EMAILSERVERUSESSL" Value="[EMAILSERVERUSESSL]" />
    <MsiProperty Name="EMAILSERVERUSESTARTTLS" Value="[EMAILSERVERUSESTARTTLS]" />
    <MsiProperty Name="EMAILSERVERUSERNAME" Value="[EMAILSERVERUSERNAME]" />
    <MsiProperty Name="EMAILSERVERUSERPASSWORD" Value="[EMAILSERVERUSERPASSWORD]" />
    <MsiProperty Name="EMAILADDRESS" Value="[EMAILADDRESS]" />
    <MsiProperty Name="BIOMETRICAUTHENTICATION" Value="[BIOMETRICAUTHENTICATION]" />
    <MsiProperty Name="FORCEDATABASEDELETION" Value="[FORCEDATABASEDELETION]" />
    <MsiProperty Name="INCLUDEDELETIONOFNONPROPRIETARYDATABASES" Value="[INCLUDEDELETIONOFNONPROPRIETARYDATABASES]
    " />
  </MsiPackage>
</Chain>

</Bundle>

</Wix>
```

Figura 2.1: *Bundle* correspondiente a un proyecto de tipo *bootstrap*.

Por último, se utiliza un proyecto del tipo *Custom Actions*, que es requerido para ampliar las capacidades de los instaladores más allá de las opciones estándar. Este se adopta principalmente por dos motivos: en primer lugar, para permitir el desarrollo en C#, lo cual es especialmente relevante en el contexto empresarial; y en segundo lugar, para centralizar aquellas acciones repetitivas y facilitar su reutilización en distintos instaladores sin necesidad de retrabajo.

[2] Página web oficial WiX: https://www.firegiant.com/wixtoolset/

Actualmente, se ha considerado reemplazar esta tecnología debido a los desafíos asociados con su mantenimiento. Aunque parte del desarrollo se realiza en C#, también existe una porción significativa en XML, lo cual exige un conocimiento específico sobre las distintas propiedades y configuraciones que ofrece este paquete. Además, en sus inicios, el uso de estas herramientas era gratuito, pero a comienzos de este año los desarrolladores han decidido introducir una tarifa de mantenimiento[3], lo que refuerza aún más la necesidad de evaluar alternativas tecnológicas.

2.1.2. IIS

Microsoft Internet Information Services [4] (IIS) es un servidor web que proporciona una forma sencilla y segura de desplegar aplicaciones web y servicios. Su arquitectura modular permite instalar únicamente los componentes necesarios, en función del caso de uso que se desee implementar.

La figura 2.2 muestra varios microservicios desplegados como aplicaciones web en una máquina interna, empleando IIS como servidor web. Asimismo, se observa la ejecución de una llamada al endpoint *Health*, cuya respuesta indica un estado *healthy*, confirmando que el servicio se encuentra operativo.

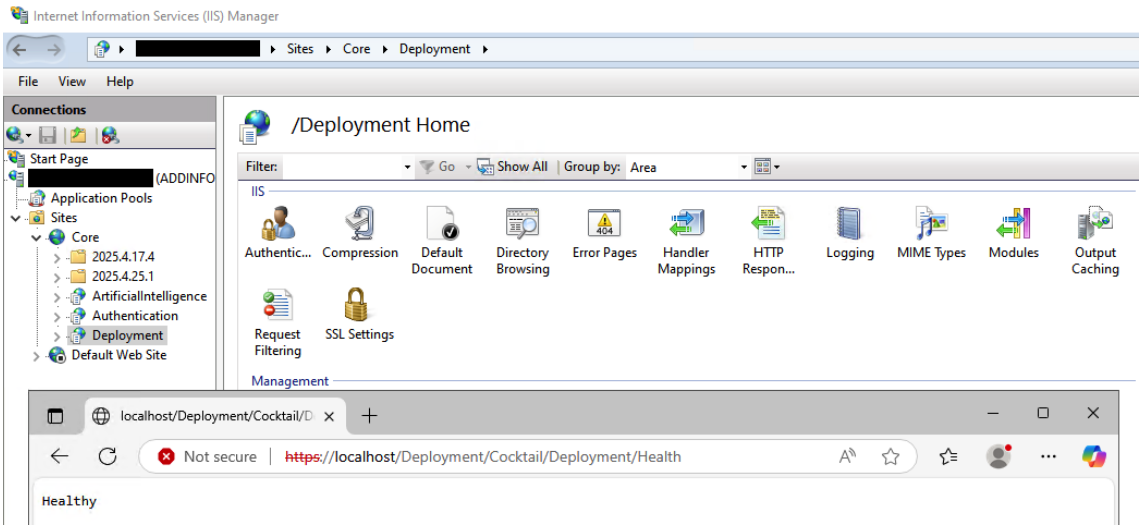


Figura 2.2: Sitios web hospedados en una máquina mediante IIS.

Los componentes principales de IIS [3] incluyen:

- **HTTP.sys**, una pila de protocolos a nivel de kernel capaz de manejar peticiones HTTP y HTTPS.
- **World Wide Web Publishing Service (W3SVC)**, que proporciona la configuración necesaria para que HTTP.sys procese las solicitudes de forma adecuada.
- **Windows Process Activation Service (WAS)**, responsable de la gestión del ciclo de vida de los procesos, incluyendo el inicio, detención y reciclado de los *application pools* (cuyo significado será explicado más adelante). WAS también se encarga de su monitorización durante el tiempo de ejecución.

[3] Anuncio de la tarifa de mantenimiento: https://robmensching.com/blog/posts/2025/02/26/introducing-the-open-source-maintenance-fee/

[4] Página web oficial IIS: https://www.iis.net

- **Configuration Store**, que contiene distintos niveles de configuración dependiendo del componente. Entre ellos destaca el archivo applicationHost.config, situado en el nivel más alto de la jerarquía de configuración, encargado de centralizar y consolidar todas las configuraciones definidas en los niveles inferiores.

- **Worker Process (w3wp.exe)**, es el proceso encargado de recibir las peticiones y generar las respuestas. Pueden existir múltiples instancias de este proceso ejecutándose de manera concurrente, dependiendo de la configuración del servidor y de la carga del sistema.

Por último, en la figura 2.3 se puede observar el flujo de acción de los componentes previamente descritos. Cada *application pool* corresponde a un microservicio o aplicación independiente, lo que permite garantizar la concurrencia entre los distintos *Worker Processes*.

En conjunto, IIS actúa como un proxy inverso, añadiendo una capa de abstracción y ofreciendo un punto de acceso unificado a múltiples microservicios.

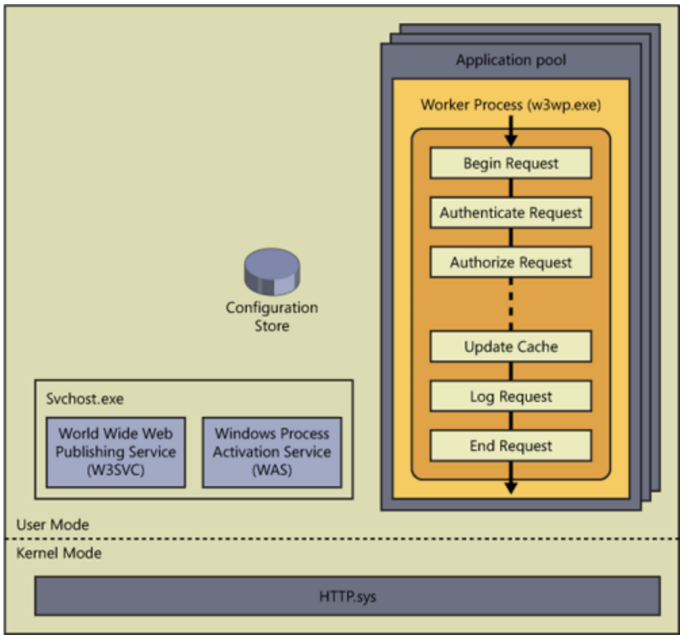


Figura 2.3: Componentes del IIS. Fuente: *Internet Information Services (IIS) 7.0 Resource Kit* [3]

2.1.3. Docker

Docker es una plataforma de gestión de contenedores diseñada para el desarrollo, empaquetado y ejecución de aplicaciones de forma portátil y prácticamente aislada. Los contenedores que ejecuta comparten únicamente el kernel con el sistema operativo anfitrión. El kernel es el componente central del sistema operativo, encargado de gestionar los recursos del sistema y actuar como intermediario entre el hardware y el software.

Docker se utiliza ampliamente debido a sus numerosas ventajas, como la eficiencia en el uso de recursos y la escalabilidad, especialmente cuando se integra con sistemas de orquestación como Kubernetes, que se describirá en el siguiente apartado.

La arquitectura de Docker se compone de dos partes principales: el cliente y el servidor. El cliente está representado por la herramienta de línea de comandos `docker CLI`,

que se comunica mediante llamadas a la API[5] con el *Docker Engine*, el cual actúa como *daemon*[6]. Este daemon está compuesto por varios componentes que, en conjunto, permiten gestionar y ejecutar contenedores. En la figura 2.4 se puede observar la separación de responsabilidades entre el cliente y el servidor.

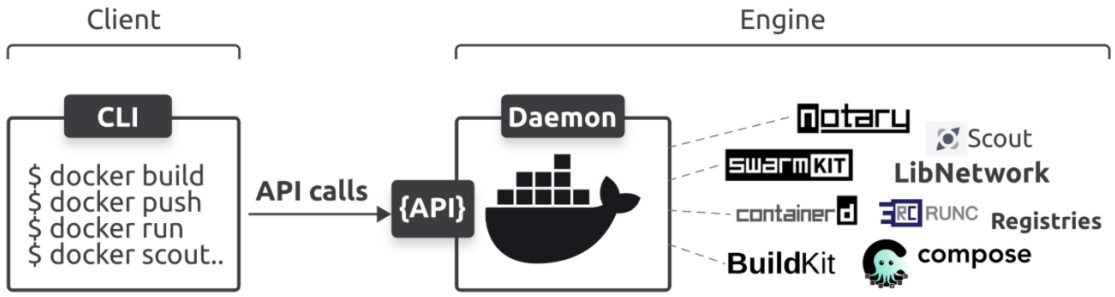


Figura 2.4: Separación de Docker entre *Client* y *Engine*. Fuente: *Docker Deep Dive* [9]

Docker Engine

El Docker Engine es el componente central del ecosistema Docker, y está compuesto por varios elementos encargados de facilitar la creación, gestión y ejecución de contenedores. Entre los principales se encuentran:

- **dockerd**: el *daemon* principal, encargado de exponer la API para que las herramientas cliente, como `docker CLI`, puedan interactuar con el motor.
- **containerd**: responsable de gestionar el ciclo de vida de los contenedores, incluyendo operaciones como el arranque, la detención y la eliminación.
- **runc**: una herramienta de línea de comandos ligera que actúa como interfaz entre Docker y el *kernel* del sistema operativo, encargándose de la ejecución y destrucción de contenedores a bajo nivel.

En la figura 2.5 se muestra un diagrama que representa los distintos componentes que forman el Docker Engine y cómo interactúan entre sí.

Imágenes

Una imagen es un paquete de solo lectura que contiene todo lo necesario para ejecutar una aplicación. Esto incluye el código de la aplicación, sus dependencias, un conjunto mínimo de elementos del sistema operativo y metadatos. A partir de una misma imagen, es posible iniciar múltiples contenedores.

En la figura 2.6 se puede apreciar un archivo de tipo *Dockerfile* cuyo propósito es definir la creación de una imagen para el despliegue de un microservicio.

[5]Una API es un conjunto de puntos de acceso o *endpoints* que facilitan la interacción entre aplicaciones, exponiendo únicamente las funciones necesarias.

[6]Un *daemon* es un programa que se ejecuta en segundo plano en un sistema operativo, generalmente sin interacción directa con el usuario.

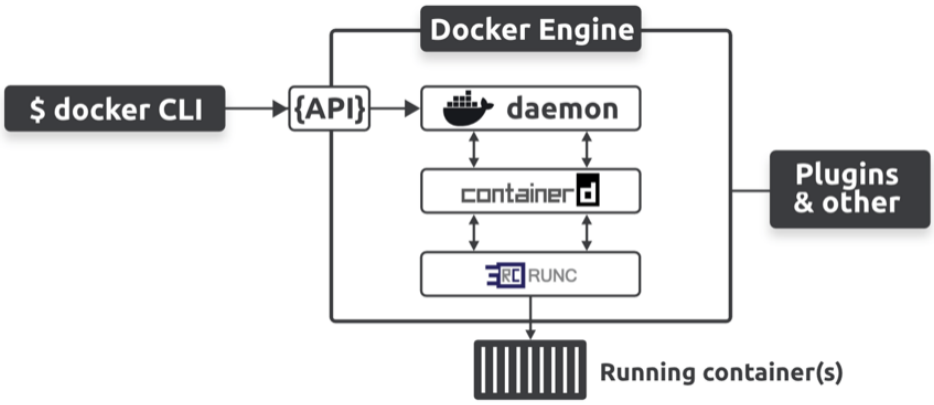


Figura 2.5: Diagrama de componentes que forman el *Docker Engine*. Fuente: *Docker Deep Dive*[9]

```
# This file defines the creation of the image for the Cocktail Deployment microservice.

# Setting the base image.
# This command specifies the base image that will be used.
# This image is produced by Microsoft, and it contains the .NET Core runtime and the ASP.NET Core packages, compiled
# into native code to improve application startup.
FROM mcr.microsoft.com/dotnet/aspnet:9.0-windowsservercore-ltsc2022

# Setting the arguments.
ARG source
ARG environment

# Setting the environment variables.
# ASP.NET Core base images use port 8080 by default but it is explicitly set for clarity.
ENV ASPNETCORE_URLS "http://*:8080"

# Setting the working directory.
# This command creates and sets the path to the /app folder.
WORKDIR /app

# Copying the application files.
# This command copies the files from a folder called ServicesBinaries into a folder called /app in the container.
# The ServicesBinaries folder does not exist at the moment, but it will be created when the Integration pipeline
# publishes the binaries as artifacts.
COPY ${source:-ServicesBinaries} .

# Exposing the HTTP port.
# This command makes the port available for HTTP traffic from outside the container.
EXPOSE 8080

# Running the application.
# This command runs the dotnet command line tool to execute the microservice.
ENTRYPOINT ["dotnet", "ADDInformatica.Cocktail.Deployment.Services.dll"]
```

Figura 2.6: Ejemplo de un *Dockerfile* que define la imagen de un microservicio.

Contenedores

Los contenedores son instancias en tiempo de ejecución de imágenes. Están diseñados para ser inmutables, lo que implica que no deben modificarse una vez desplegados. En caso de fallo, la práctica recomendada no es reparar el contenedor directamente, sino sustituirlo por una nueva instancia a partir de la imagen original.

Para la visualización y gestión de los contenedores se emplea *Portainer* [7], una interfaz gráfica de usuario ligera que permite administrar de manera intuitiva contenedores, imágenes y otros recursos dentro de entornos Docker. En la figura 2.7 se puede observar

[7]Página web oficial Portainer: https://www.portainer.io

la presencia de varios contenedores desplegados, donde cada uno de ellos representa un microservicio. El estado *running* indica que los contenedores se encuentran en ejecución y funcionando correctamente.

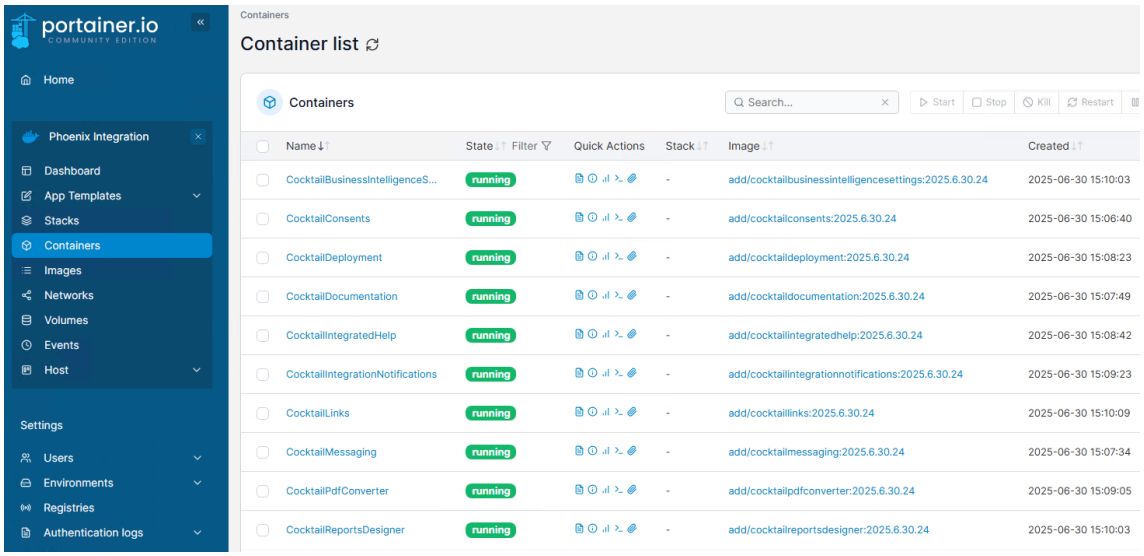


Figura 2.7: Portainer con varios contenedores desplegados y en estado *running*.

Integración con WebAssembly

Cabe destacar que WebAssembly permite construir aplicaciones más ligeras, rápidas, seguras y portables en comparación con los contenedores tradicionales. Para su ejecución, basta con desarrollar la aplicación en el lenguaje de programación deseado, compilarla a un binario en formato WebAssembly y ejecutarlo en cualquier entorno que disponga del runtime correspondiente.

Actualmente, herramientas como Docker también permiten ejecutar aplicaciones WebAssembly de forma nativa, ya que incluyen varios runtimes compatibles desde la instalación por defecto de Docker Desktop. Esto significa que, en muchos casos, el despliegue de servidores WebAssembly puede gestionarse utilizando contenedores convencionales.

Este apartado adquiere una relevancia particular en el contexto de este trabajo, ya que se plantea la posibilidad de realizar dicho despliegue mediante un microservicio personalizado, lo que permitirá explorar una alternativa más ligera y especializada.

2.1.4. Kubernetes

Kubernetes se ha consolidado como el orquestador estándar en la industria para el despliegue de aplicaciones, permitiendo distribuirlas de manera eficiente entre distintos nodos y zonas de disponibilidad. Esta capacidad de distribución no solo optimiza el rendimiento general del sistema, sino que también mejora significativamente su disponibilidad. Además, Kubernetes gestiona de forma automática la recuperación ante fallos, escala los servicios en función de la demanda y administra los procesos de actualización de manera controlada.

Para llevar a cabo estas funciones, Kubernetes se organiza en torno a un *clúster*, entendido como un conjunto de nodos que aportan recursos computacionales como CPU o memoria para el funcionamiento de las aplicaciones. Dentro de un *clúster* se distinguen dos tipos de nodos: los *nodos de control* y los *nodos trabajadores*. Cabe destacar que los

nodos de control deben ejecutarse sobre sistemas Linux, mientras que los nodos trabajadores pueden operar tanto en entornos Linux como Windows.

- **Nodo de control**: agrupa los servicios que conforman el núcleo operativo de Kubernetes, siendo responsable de exponer la API del sistema y de llevar a cabo tareas críticas como la gestión de procesos y el escalado de cargas de trabajo.
- **Nodos trabajadores**: son los encargados de ejecutar la lógica de negocio de las aplicaciones desplegadas, constituyendo así la capa productiva del sistema.

En las figuras 2.8 y 2.9 se ilustran los componentes que integran cada uno de estos tipos de nodo. No se entrará en el detalle de dichos componentes, ya que ello añadiría una complejidad técnica que excede el alcance de este trabajo. No obstante, resulta fundamental comprender la relación jerárquica que se establece entre ellos, siguiendo un patrón maestro-esclavo. En este esquema, los nodos trabajadores dependen funcionalmente de los nodos de control para operar, mientras que estos últimos carecerían de utilidad práctica sin la existencia de nodos trabajadores que ejecuten las cargas de trabajo definidas.

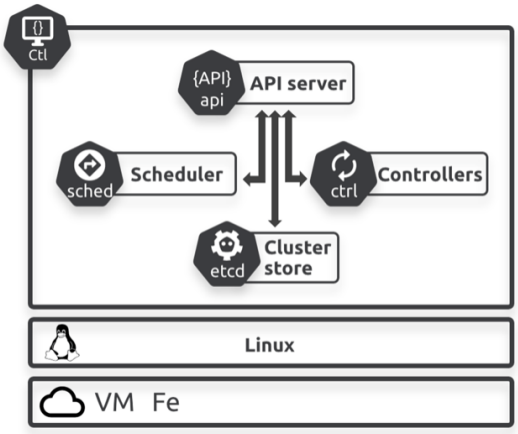


Figura 2.8: Componentes del nodo de control. Fuente: *The Kubernetes Book* [10]

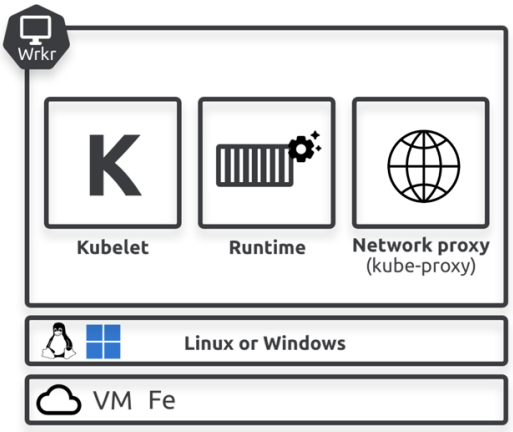


Figura 2.9: Componentes del nodo trabajador. Fuente: *The Kubernetes Book* [10]

Pods

Cada *Pod* representa un entorno de ejecución compartido para uno o más contenedores. Este entorno incluye elementos como la pila de red, volúmenes y memoria compartida. Desde una perspectiva arquitectónica, los Pods pueden entenderse como una envoltura o *wrapper* que abstrae las distintas tareas que Kubernetes puede realizar sobre los contenedores y sus entornos asociados. En el contexto específico de Docker, un Pod actúa como contenedor lógico que encapsula uno o más contenedores de Docker, permitiendo que estos compartan el mismo espacio de red y almacenamiento, como se puede observar en la figura 2.10.

Los Pods son entidades **efímeras**: se crean, viven y eventualmente se eliminan. Cada vez que uno muere, Kubernetes lo reemplaza automáticamente por uno nuevo. Aunque el nuevo Pod pueda parecer idéntico al anterior, en realidad es completamente nuevo, con un ID y una dirección IP distintos. Esto obliga a diseñar las aplicaciones de manera desacoplada, de forma que sean resistentes a fallos individuales de Pods.

Los Pods, al igual que los contenedores, también son **inmutables**, lo que significa que no deben modificarse una vez que están en ejecución. Si se necesita actualizar un Pod,

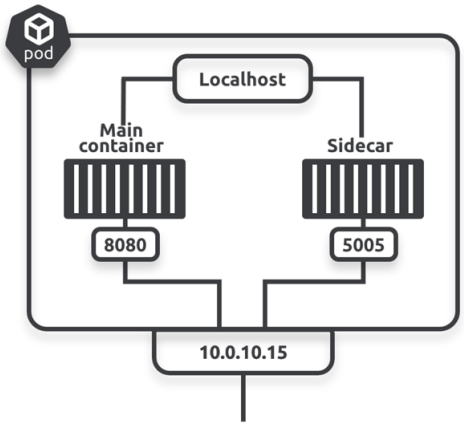


Figura 2.10: Pod formado por dos contenedores de Docker. Fuente: *The Kubernetes Book*[10]

se debe eliminar el existente y crear uno nuevo con los cambios requeridos. Este enfoque puede requerir un cambio de mentalidad, pero se alinea bien con herramientas modernas y flujos de trabajo tipo *GitOps*[8].

2.2 Comparativa

En lugar de realizar una tabla comparativa tradicional entre las tecnologías descritas, se ha optado por un análisis cualitativo. Esta decisión se justifica por la naturaleza heterogénea de las herramientas analizadas: cada una de ellas está orientada a resolver problemáticas distintas dentro del ciclo de vida de una aplicación, y no compiten entre sí en términos funcionales, sino que pueden actuar de forma complementaria en un entorno real.

Por ejemplo, herramientas como WiX están orientadas a la encapsulación y distribución de aplicaciones en entornos Windows, mientras que IIS se encarga de su hospedaje en forma de servicios web. Por otro lado, Docker y Kubernetes representan una filosofía completamente distinta basada en contenedores y orquestación, donde los conceptos de instalación tradicional son reemplazados por despliegues automatizados y entornos efímeros. En este sentido, intentar establecer métricas directas de comparación resultaría poco representativo, ya que cada solución actúa en una capa distinta del sistema o en fases diferentes del ciclo de vida del software.

Además, uno de los principales motivos que impulsan el desarrollo de una solución personalizada es la necesidad de controlar y minimizar las dependencias externas que deben instalarse en las máquinas cliente. En el estado actual, cada una de las soluciones analizadas requiere componentes adicionales que deben ser gestionados manualmente o mediante automatización compleja. A continuación se destacan algunos ejemplos relevantes:

- Para ejecutar contenedores Linux en máquinas Windows, es necesario instalar **Windows Subsystem for Linux (WSL)**, así como habilitar características específicas del sistema operativo.
- El uso de Docker requiere la instalación del **Docker Engine**, así como la configuración de permisos administrativos y el acceso a redes específicas. En entornos corporativos, esto suele implicar restricciones adicionales de seguridad.

[8]Son una serie de prácticas destinadas a gestionar las configuraciones de las aplicaciones y las infraestructuras para mejorar el ciclo de vida de las aplicaciones

- Kubernetes, aunque extremadamente potente, introduce una capa adicional de complejidad al necesitar runtimes como `containerd`, herramientas como `kubectl`, y una distribución local como `Minikube` o `K3s`, lo cual implica no solo un mayor consumo de recursos, sino también un mantenimiento técnico constante.
- IIS, a pesar de ser nativo en entornos Windows, requiere una configuración detallada de *application pools*, bindings, y archivos de configuración como el `web.config`, lo cual puede volverse propenso a errores en escenarios con múltiples microservicios o cuando se requieren despliegues reproducibles.

Considerando que todas las máquinas cliente empleadas actualmente son sistemas Windows, el objetivo del microservicio propuesto es proporcionar una solución de despliegue controlada y contenida, que no dependa de infraestructura externa ni requiera que el cliente instale ni mantenga software adicional. Esta propuesta también contribuye a reducir riesgos de seguridad, facilitar el soporte técnico y estandarizar el proceso de despliegue en toda la base de instalaciones.

En consecuencia, en lugar de determinar qué herramienta es "mejor" de forma absoluta, el enfoque adoptado ha sido extraer y adaptar las responsabilidades clave de cada solución, descartando aquellas cuya complejidad o requisitos exceden las necesidades reales del cliente. Se busca así un equilibrio entre funcionalidad, mantenibilidad y simplicidad operativa.

2.3 Conclusiones

A lo largo de este capítulo se han analizado herramientas ampliamente consolidadas para el despliegue, empaquetado y ejecución de aplicaciones, cada una con enfoques distintos y grados variables de complejidad. Esta exploración ha permitido comprender no solo sus arquitecturas y mecanismos internos, sino también sus implicaciones prácticas en un entorno empresarial.

El estudio detallado de estas soluciones ha revelado una problemática común: la fuerte dependencia de componentes externos, así como la necesidad de configuraciones específicas en la máquina cliente. Este hecho representa una barrera significativa tanto para la escalabilidad del sistema como para su mantenibilidad a largo plazo, especialmente cuando se requiere uniformidad en múltiples entornos controlados.

Por ello, surge la necesidad de desarrollar una alternativa personalizada que se inspire en los principios arquitectónicos extraídos —como la inmutabilidad, el aislamiento de ejecución o la gestión declarativa del ciclo de vida— pero que al mismo tiempo los adapte a las restricciones de un entorno Windows, sin exigir la instalación de software adicional por parte del cliente.

Este enfoque permitirá ofrecer una solución de despliegue ligera, consistente y alineada con las necesidades reales del producto y de los usuarios finales.

CAPÍTULO 3
Tecnologías utilizadas

La tecnología empleada para el desarrollo de los microservicios ha sido el framework ASP.NET Core [18], utilizando C# como lenguaje de programación y Visual Studio 2022 como entorno de desarrollo integrado (IDE). La elección de este IDE se debe a su elevada integración con ASP.NET Core y a las múltiples funcionalidades que proporciona en este contexto.

Para la gestión del código fuente, así como para la integración y entrega continuas (CI/CD), se ha utilizado Azure DevOps [1]. La definición y mantenimiento de las pipelines se ha llevado a cabo mediante el lenguaje YAML (Yet Another Markup Language), y se ha empleado PowerShell [2] como mecanismo interno de Infrastructure as Code (IaC) para la gestión de entornos en hosts privados. La elección de PowerShell sobre otros lenguajes de scripting se fundamenta en la fuerte vinculación tanto de la empresa como de sus clientes con el sistema operativo Windows.

En lo que respecta a la gestión de la infraestructura, se ha optado por Visual Studio Code debido a su sencillez, flexibilidad y capacidad de ampliación mediante extensiones. Adicionalmente, se ha utilizado Postman como entorno para el desarrollo y prueba de APIs, permitiendo realizar peticiones HTTP a los distintos microservicios backend [22]. Para la visualización y administración de los datos, se ha recurrido a SQL Server Management Studio [3] como sistema de gestión de bases de datos (SGBD), complementado con MySQL como motor de base de datos.

En cuanto a la gestión del repositorio de código, se ha hecho uso de una herramienta interna denominada Repository Files Generator (RFG), encargada de la generación automática de ciertos archivos del repositorio, tales como pipelines, soluciones `.sln` o archivos `.targets`, lo cual ha facilitado el desarrollo local al permitir referencias directas a proyectos en lugar de paquetes NuGet.

Por último, para el desarrollo de la interfaz de usuario de los microservicios se ha utilizado el framework Uno Platform, el cual permite generar aplicaciones nativas multiplataforma a partir de una única base de código en C#. En el caso particular de este proyecto, se ha generado una aplicación WebAssembly utilizando el paquete Uno.Wasm.Bootstrap, que permite empaquetar la aplicación en un servidor Kestrel para facilitar su posterior despliegue.

[1] Documentación oficial de Azure DevOps: https://learn.microsoft.com/en-us/azure/devops

[2] Documentación oficial de PowerShell: https://learn.microsoft.com/es-es/powershell/

[3] Documentación oficial de SQL Server: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver17

3.1 Uno Platform

Uno Platform [4] es un marco de desarrollo multiplataforma basado en tecnologías del ecosistema Microsoft [20]. Permite crear aplicaciones utilizando una única base de código y proporciona una experiencia coherente para desarrolladores familiarizados con entornos como *Universal Windows Platform* (UWP) o *Windows UI Library* (WinUI).

WinUI [5], por su parte, es una biblioteca moderna de interfaces de usuario desarrollada por Microsoft. Está diseñada para crear aplicaciones con apariencia nativa sobre Windows, y representa la evolución de los modelos de interfaz anteriores dentro del sistema operativo.

Si bien este trabajo no se centra en describir en profundidad la arquitectura ni el funcionamiento de Uno Platform, ya que su configuración y adopción formaban parte del entorno ya existente al iniciar el desarrollo, se considera importante introducir brevemente la tecnología y mostrar un ejemplo práctico de su uso. Para una descripción más detallada sobre Uno Platform, se recomienda consultar *Getting Started with the Uno Platform and WinUI 3* de Skye Hoefling [11].

En este proyecto, Uno Platform se utilizó como framework para implementar la interfaz de usuario de los distintos microservicios, incluido el microservicio de despliegue. Se definieron varias pantallas comunes a todas las plataformas soportadas. En particular, este trabajo se centra en el despliegue sobre la plataforma *WebAssembly*. En la figura 3.1 se presenta un ejemplo de una de las pantallas desarrolladas.

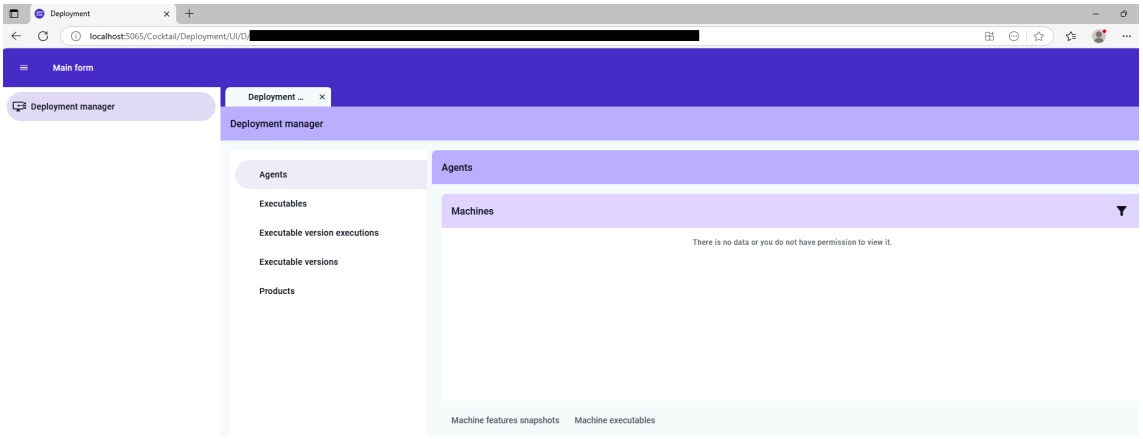


Figura 3.1: Ejemplo de formulario del microservicio de despliegue, desarrollado con Uno Platform y desplegado mediante WebAssembly.

3.2 WebAssembly

WebAssembly es un formato binario de bajo nivel, similar al lenguaje ensamblador, que permite alcanzar un rendimiento significativamente alto, comparable al de las aplicaciones ejecutadas de forma nativa [21]. Esta tecnología ofrece a lenguajes como C/C++, C#, y Rust un nuevo objetivo de compilación (*compilation target*), lo que posibilita generar binarios ejecutables en el entorno WebAssembly a partir de dichos lenguajes.

[4] Página web oficial Uno Platform: https://platform.uno
[5] Documentación oficial de WinUI: https://learn.microsoft.com/en-us/windows/apps/winui/

Cabe señalar que WebAssembly está siendo desarrollado como un estándar web por el *W3C WebAssembly Working Group* [6] y el *Community Group*, con la participación activa de los principales fabricantes de navegadores.

La elección de esta tecnología se fundamenta en su alto rendimiento y en la facilidad de integración con el entorno de trabajo habitual de la empresa, basado en .NET. En particular, Uno Platform ofrece soporte directo para WebAssembly como plataforma de destino dentro de su propio framework, lo que simplifica considerablemente el proceso de desarrollo.

Además, Uno Platform proporciona un paquete NuGet denominado *Uno.Wasm.Bootstrap*, que permite empaquetar y ejecutar aplicaciones en servidores WebAssembly sin requerir conocimientos previos sobre dicha tecnología, lo que facilita significativamente su adopción. En la figura 3.2 se muestra un proyecto que hace referencia al paquete NuGet mencionado.

En la figura 3.3 se presentan los archivos estáticos que sirven la página web, es decir, aquellos contenidos en la carpeta `wwwroot`. Al publicar el servidor WebAssembly como un archivo autocontenido, también se incluye el *runtime* de WebAssembly, lo que permite su ejecución en cualquier máquina.

Por último, en la figura 3.4 se puede observar la salida en consola al ejecutar el archivo `.exe` resultante de la publicación del servidor WebAssembly. En dicha salida se muestra la dirección IP y el puerto en los que se encuentra escuchando peticiones, así como la ruta de los archivos estáticos que está sirviendo, correspondiente a los mostrados en la figura 3.3.

```
<!--------------------------------------------------------------------
  <copyright file="Launcher.D.WasmServer.csproj" company="ADD Informática">
    Copyright (c) Aphelion Soluciones Informáticas S.L.
  </copyright>
--------------------------------------------------------------------->

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

  <Import Project="$(AgilityMSBuildFilesDirectory)Common.UP.Launcher.WasmServer.props" Condition="'$(CommonUnoPlatformLauncherPropsHasBeenImported)' != 'true'" />

  <PropertyGroup>
    <AssemblyName>ADDInformatica.Cocktail.Deployment.UI.Launcher.D.WasmServer</AssemblyName>
    <RootNamespace>ADDInformatica.Cocktail.Deployment.UI.Launcher.D.WasmServer</RootNamespace>
    <Description>ADD Informática Cocktail Deployment UI Launcher Desktop Wasm Server</Description>
    <Product>ADD Informática Cocktail Deployment</Product>
  </PropertyGroup>

  <ItemGroup>
    <!-- The appsettings files are excluded from the publication as this project is only deployed using the Deployment
      microservice system and configuration is obtained from there. -->
    <Content Update="appsettings*.json" CopyToPublishDirectory="Never"/>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Launcher.D.csproj" SetTargetFramework="TargetFramework=$(DotnetVersion)-browserwasm" ReferenceOutputAssembly="false" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="ADDInformatica.Cocktail.Core.Contracts.Services" Version="$(CocktailCoreContractsVersion)" />
    <PackageReference Include="ADDInformatica.Cocktail.Core.Services" Version="$(CocktailCoreVersion)" />
    <PackageReference Include="ADDInformatica.Cocktail.Core.UI.UP.Middlewares" Version="$(CocktailCoreUiUpVersion)" />
    <PackageReference Include="ADDInformatica.Cocktail.ExternalCodeExtensionMethods.Configuration" Version="$(CocktailExternalCodeExtensionMethodsVersion)" />
    <PackageReference Include="McMaster.Extensions.CommandLineUtils" Version="$(McMasterVersion)" />
  </ItemGroup>

  <Import Project="$(AgilityMSBuildFilesDirectory)Common.UP.Launcher.WasmServer.targets" Condition="'$(CommonUnoPlatformLauncherTargetsHasBeenImported)' != 'true'" />

</Project>
```

Figura 3.2: Archivo `.csproj` del servidor WebAssembly.

[6] Página web al estándar W3 de WebAssembly: https://www.w3.org/groups/wg/wasm/

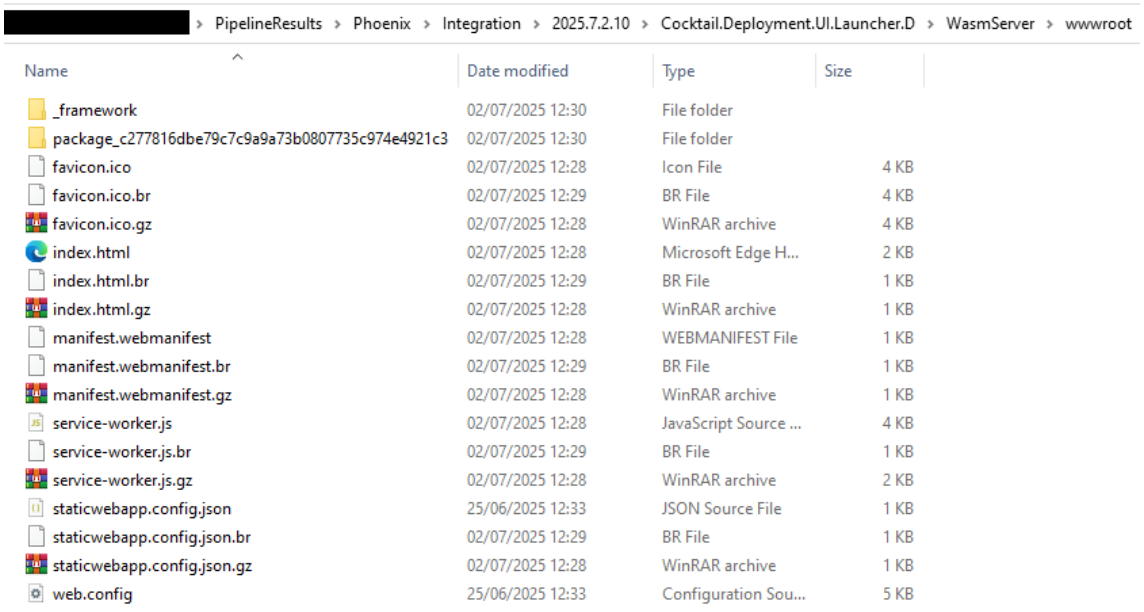


Figura 3.3: Archivos resultantes al realizar la publicación del servidor WebAssembly.

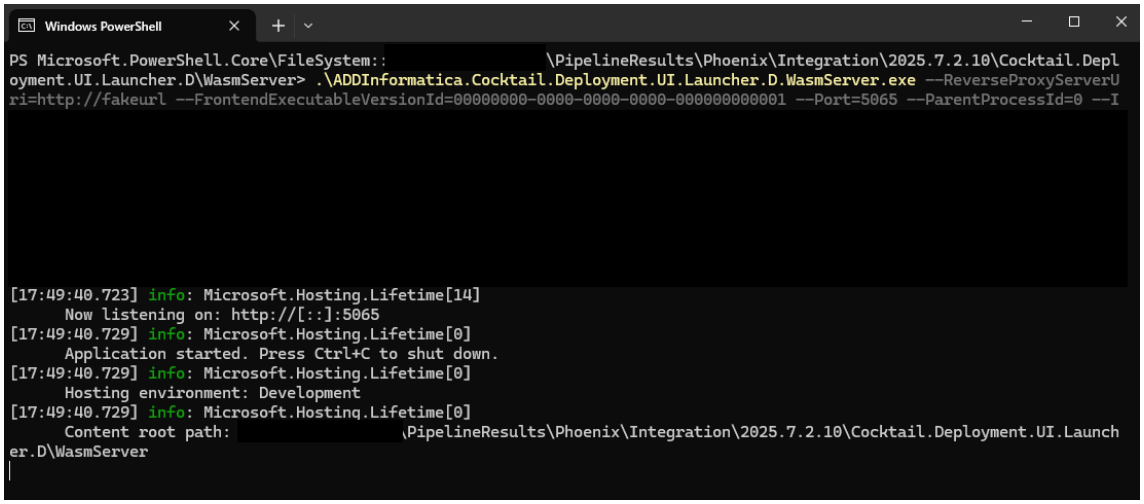


Figura 3.4: Ejecución del servidor WebAssembly mediante línea de comandos.

CAPÍTULO 4
Desarrollo de la solución

En este capítulo se va a explicar el procedimiento que se ha seguido para construir los distintos componentes que participan en el despliegue de la parte frontend de microservicios propios. Primero se explica la metodología utilizada, utilizando para ello un proceso ágil. Después se especifican los requisitos tanto funcionales como no funcionales del producto a desarrollar. A continuación se detalla el diseño tanto previo como final de la solución, junto con la programación y pruebas realizadas sobre el mismo. Por útlimo, se expone el proceso de despliegue de todos los componentes en un escenario real.

4.1 Metodología

En el marco de los métodos ágiles[6], los más populares son Kanban, Scrum o *Extreme Programming* (XP). Para la metodología aplicada en el presente proyecto se han elegido algunas prácticas de dichos métodos que más se adecuaban al proceso de desarrollo llevado a cabo, y que se comentan a continuación.

4.1.1. Backlog y Tablero kanban

Un *backlog* es una representación en forma de lista que contiene tareas ordenadas por prioridad o importancia. Por otra parte, un tablero kanban es una herramienta utilizada para visualizar el flujo de trabajo de un equipo, está compuesta por columnas donde cada una representa el estado de una historia de usuario.

Para el cometido, se emplearán las herramientas que proporciona Azure DevOps Server (ADOS), concretamente las secciones *Backlogs* y *Boards*, tal como se muestra en la figura 4.1 y figura 4.2 respectivamente.

4.1.2. Roles

Los tres roles principales de la metodología Scrum son: el *Product Owner*, encargado de organizar y mantener la lista de tareas del producto o *product backlog*; el *Scrum Master*, responsable de velar por el cumplimiento del proceso ágil; y el equipo de desarrolladores.

Todos estos roles han estado presentes a lo largo del proceso, uno de los socios de la empresa ha actuado como *Product Owner*, y el jefe del departamento de I+D+i ha desempeñado el rol de *Scrum Master*. Esta asignación de funciones ha permitido establecer una estructura de responsabilidades clara y bien definida.

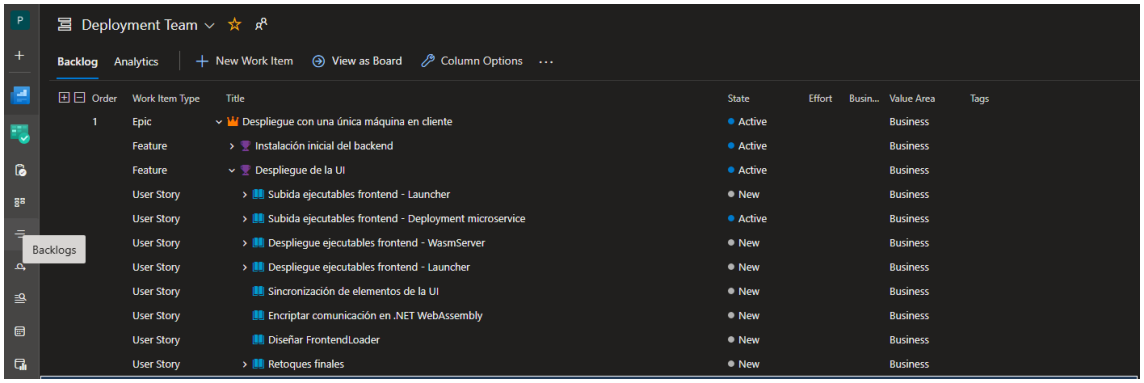


Figura 4.1: Captura del apartado Backlog del ADOS

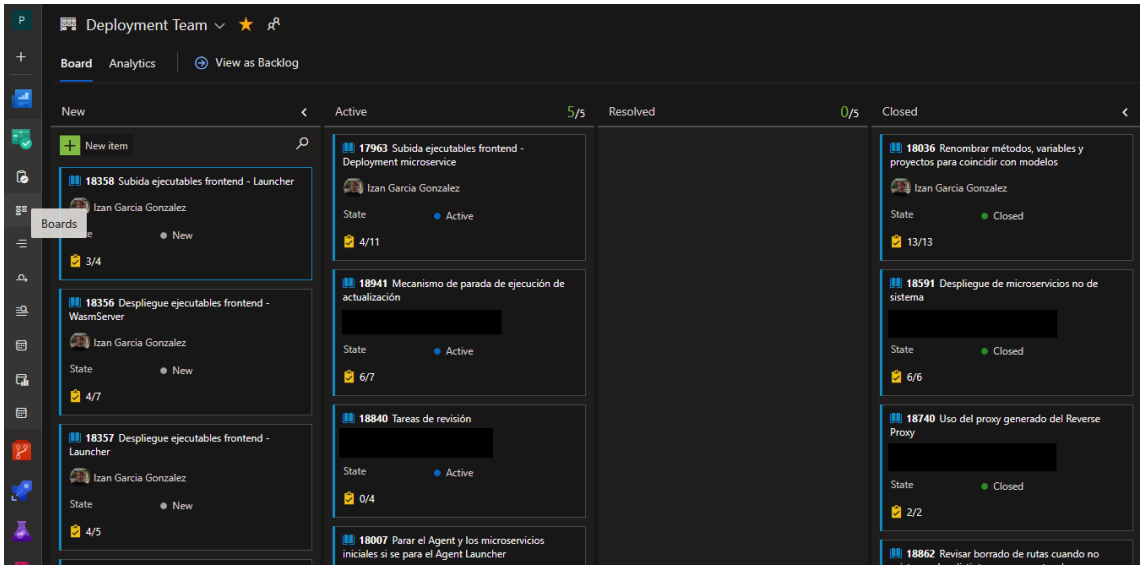


Figura 4.2: Captura del apartado Boards del ADOS

4.1.3. Sprints

Un Sprint es una iteración o ciclo de trabajo con una duración determinada y un objetivo concreto, cuyo propósito es lograr un incremento potencialmente entregable del producto. En este caso, se han definido cuatro Sprints, cada uno con una duración aproximada de 3 semanas.

Como se puede apreciar en la figura 4.3, el Sprint 0, o puesta en marcha, resulta fundamental para llevar a cabo formación necesaria y preparar tanto el *backlog* como empezar a priorizar el trabajo correspondiente a los siguientes Sprints. El último de ellos se denomina *Entrega*, ya que corresponde a la versión que será desplegada en producción y estará disponible para su uso por parte del cliente.

Al finalizar cada iteración, se obtiene una nueva versión del producto, la cual se presenta mediante un *Sprint Review*. Este consiste en una demostración con software operativo ante los usuarios y *stakeholders* [1]. La demostración solo incluye aquellos aspectos que se han conseguido abordar durante el Sprint y que han sido completados satisfactoriamente, incluyendo el diseño y ejecución de las pruebas correspondientes.

Al final de la reunión, los *stakeholders* tienen la oportunidad de compartir su opinión, realizar observaciones y plantear cualquier duda al equipo de desarrollo.

[1] Personas o grupos que tienen un interés legítimo en el producto desarrollado

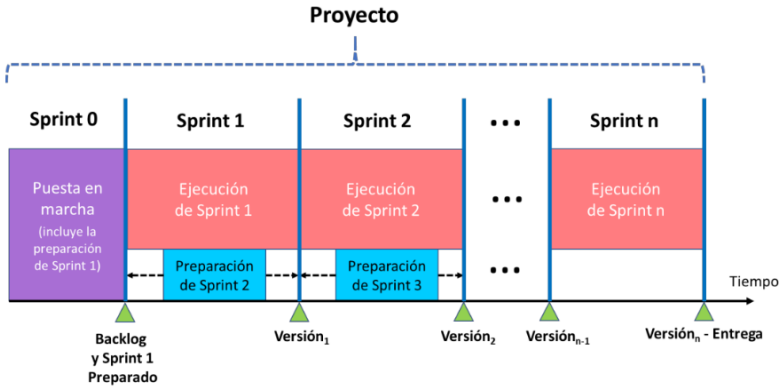


Figura 4.3: Iteraciones en forma de Sprints. Fuente: Metodología ágil utilizada en PIN [1]

4.2 Especificación de Requisitos

Para la especificación de requisitos del microservicio de despliegue se han elaborado diagramas de Casos de Uso que el producto debe cubrir. En el presente apartado, se describen los requisitos centrados en el despliegue de los ejecutables de frontend, así como aquellos requisitos de carácter general. Adicionalmente, se presentan los requisitos no funcionales que deberá satisfacer el producto final.

4.2.1. Casos de Uso

Para describir los Casos de Uso, se utiliza un diagrama que representa visualmente las interacciones, figura 4.4. Asimismo, se incluye una tabla que detalla, para cada Caso de Uso, un identificador, un nombre, el actor correspondiente y una breve descripción de su funcionalidad.

Identificador	CU01
Nombre	Desplegar producto
Actor	Administrador del sistema
Descripción	El administrador del sistema solicita al microservicio de despliegue la instalación de un producto en su máquina. El microservicio se encarga de realizar el despliegue, instalando los componentes necesarios y mostrando en todo momento el estado del proceso. Al finalizar, el producto queda correctamente desplegado en el equipo del administrador.

Identificador	CU02
Nombre	Actualizar producto
Actor	Administrador del sistema
Descripción	El microservicio de despliegue consulta periódicamente si existe alguna actualización disponible. En caso afirmativo, el administrador del sistema puede solicitar la actualización de un producto específico en su equipo. El microservicio se encarga de llevar a cabo el proceso de actualización, sincronizando los componentes necesarios y proporcionando información en tiempo real sobre el estado del proceso. Una vez finalizado, el producto queda correctamente actualizado en la máquina del administrador.

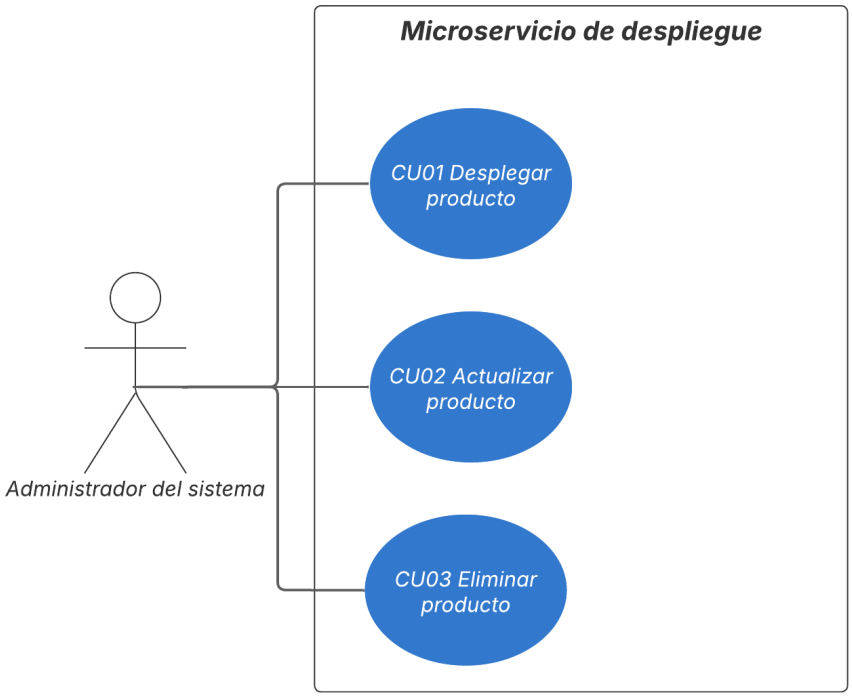


Figura 4.4: Diagrama de casos de uso.

Identificador	CU03
Nombre	Eliminar producto
Actor	Administrador del sistema
Descripción	El administrador del sistema solicita al microservicio de despliegue la eliminación de un producto de su máquina. El microservicio se encarga de llevar a cabo la eliminación, borrando todos los componentes instalados y mostrando en todo momento el estado del proceso. Al finalizar, no queda ningún elemento relacionado con el producto en el equipo del administrador.

4.2.2. Requisitos funcionales detallados

Además de los casos de uso previamente descritos, el microservicio de despliegue debe satisfacer ciertas funcionalidades específicas que, si bien no se expresan formalmente como Casos de Uso, resultan esenciales para comprender su comportamiento general. Estos aspectos funcionales se enfocan en lo que el sistema debe realizar internamente, dejando de lado la interacción directa con el usuario final.

Aunque podrían haberse integrado junto con los casos de uso, se ha optado por presentarlos de forma separada. Esta decisión permite distinguir con mayor claridad, por un lado, las tres operaciones fundamentales que el microservicio debe llevar a cabo, y por otro, una serie de detalles técnicos adicionales. Agrupar todos los requisitos funcionales en un mismo esquema —por ejemplo, mediante relaciones como extend o include— podría dificultar su interpretación.

Por este motivo, se presentan de forma independiente los tres comportamientos principales del microservicio de despliegue. A partir de este punto, dicho microservicio será referido como Deployment, con el fin de facilitar su identificación en el resto del docu-

mento. Cada comportamiento se documenta mediante un identificador, un título descriptivo, su posible vinculación con uno o varios casos de uso, y una breve descripción de su funcionalidad.

Identificador	RF01
Nombre	Versionar entidades
Caso de uso relacionado	CU01, CU02, CU03
Descripción	Todas las entidades del dominio deben contar con un número de versión asociado, correspondiente a la versión del producto que se desea desplegar. Esta versión permite mantener un historial de actualizaciones, y facilita las operaciones de instalación, despliegue, actualización y eliminación de los distintos componentes.

Identificador	RF02
Nombre	Despliegue Deployment Master y Client
Caso de uso relacionado	CU01, CU02, CU03
Descripción	El microservicio Deployment contempla dos posibles comportamientos: el rol *master* y el rol *client*. El Deployment Master actúa como repositorio central de datos y constituye el punto de coordinación para la sincronización y el despliegue de actualizaciones en los nodos Deployment Client. Por tanto, es fundamental garantizar un despliegue correcto tanto del componente *master* en la infraestructura del proveedor, como del componente *client* en la infraestructura del consumidor

Identificador	RF03
Nombre	Sincronización entre Deployment Master y Client
Caso de uso relacionado	CU01, CU02
Descripción	Para garantizar la consistencia de los datos, el Deployment Client debe sincronizar previamente todas las entidades necesarias de la base de datos con el Deployment Master antes de ejecutar cualquier operación. Este mecanismo asegura que, en el entorno del cliente, se disponga de un estado determinista y alineado con la información central, reduciendo significativamente la posibilidad de errores derivados de diferencias entre entornos.

Identificador	RF04
Nombre	Subir ejecutables backend
Caso de uso relacionado	CU01, CU02
Descripción	El microservicio debe ser capaz de almacenar en su base de datos los ejecutables correspondientes al backend. Estos ejecutables representan la lógica de negocio del microservicio, es decir, las acciones que se exponen al exterior a través de una API. Este mecanismo permite garantizar la consistencia del sistema para la sincronización (RF03) y habilita el control de versiones (RF01) para facilitar futuras actualizaciones (RF07).

Identificador	RF05
Nombre	Subir ejecutables frontend
Caso de uso relacionado	CU01, CU02
Descripción	El microservicio debe ser capaz de almacenar en su base de datos los ejecutables correspondientes al frontend. Estos ejecutables constituyen la interfaz de usuario del microservicio, la cual se comunica con su correspondiente backend. Este mecanismo permite garantizar la consistencia del sistema para la sincronización (RF03) y habilita el control de versiones (RF01) para facilitar futuras actualizaciones (RF07).

Identificador	RF06
Nombre	Desplegar ejecutables
Caso de uso relacionado	CU01, CU02
Descripción	Es necesario que el microservicio de Deployment sea capaz de iniciar los procesos de cada uno de los ejecutables dentro de la máquina del cliente. Primero deberán estar subidos a la base de datos y sincronizados.

Identificador	RF07
Nombre	Desplegar actualización
Caso de uso relacionado	CU01, CU02
Descripción	El Deployment Client comprobará periódicamente con el Deployment Master si existe una nueva actualización disponible. En caso afirmativo, el microservicio procederá a realizar la sincronización correspondiente (RF03) y desplegará aquellas entidades que estén asociadas al número de versión especificado (RF01).

Identificador	RF08
Nombre	Deployment como repositorio central de configuraciones
Caso de uso relacionado	CU01, CU02
Descripción	Todas las configuraciones necesarias para el funcionamiento de los microservicios deben estar almacenadas en el microservicio de Deployment. En caso de que un microservicio requiera una configuración para iniciar su ejecución, esta debe solicitarse directamente a Deployment, lo que permite habilitar una configuración dinámica, entendiéndose como dinámica, toda aquella acción que sucede durante el tiempo de ejecución. Esta característica resulta especialmente valiosa en entornos donde no se dispone de acceso directo a los sistemas, ya que facilita la resolución ágil de incidencias y reduce la necesidad de intervenciones manuales.

4.2.3. Requisitos no funcionales

Un requisito no funcional se define como una condición o limitación que afecta al producto software, pero que no está relacionada con sus funcionalidades directas. Este tipo de requisitos está estrechamente vinculado a los atributos de calidad del sistema, y

puede abordar aspectos como la confiabilidad, la facilidad de uso u otras propiedades del comportamiento del software.

La norma ISO/IEC 25010 [7], también conocida como SQuaRE (System and software Quality Requirements and Evaluation), establece un modelo de calidad que identifica ocho características fundamentales, estas se pueden apreciar en la figura 4.5.

CALIDAD DEL PRODUCTO SOFTWARE								
ADECUACIÓN FUNCIONAL	EFICIENCIA DE DESEMPEÑO	COMPATIBILIDAD	CAPACIDAD DE INTERACCIÓN	FIABILIDAD	SEGURIDAD	MANTENIBILIDAD	FLEXIBILIDAD	PROTECCIÓN
COMPLETITUD FUNCIONAL CORRECCIÓN FUNCIONAL PERTINENCIA FUNCIONAL iso25000.com	COMPORTAMIENTO TEMPORAL UTILIZACIÓN DE RECURSOS CAPACIDAD	COEXISTENCIA INTEROPERABILIDAD	RECONOCIBILIDAD DE ADECUACIÓN APRENDIZABILIDAD OPERABILIDAD PROTECCIÓN FRENTE A ERRORES DE USUARIO INVOLUCRACIÓN DEL USUARIO INCLUSIVIDAD ASISTENCIA AL USUARIO AUTO-DESCRIPTIVIDAD	AUSENCIA DE FALLOS DISPONIBILIDAD TOLERANCIA A FALLOS RECUPERABILIDAD	CONFIDENCIALIDAD INTEGRIDAD NO-REPUDIO RESPONSABILIDAD AUTENTICIDAD RESISTENCIA	MODULARIDAD REUSABILIDAD ANALIZABILIDAD CAPACIDAD DE SER MODIFICADO CAPACIDAD DE SER PROBADO	ADAPTABILIDAD ESCALABILIDAD INSTALABILIDAD REEMPLAZABILIDAD	RESTRICCIÓN OPERATIVA IDENTIFICACIÓN DE RIESGOS PROTECCIÓN ANTE FALLOS ADVERTENCIA DE PELIGRO INTEGRACIÓN SEGURA

Figura 4.5: Características y sub-características de calidad de un producto *software* definidas en la ISO/IEC 25010 [17]

.

A continuación, se describen los requisitos no funcionales que se deben satisfacer, indicando para cada uno de ellos, un identificador, un nombre, la característica de la ISO a la que hacen referencia y una breve descripción.

Identificador	RNF01
Nombre	Autenticación entre Deployments
Característica	Seguridad
Descripción	Las peticiones que se realicen desde el Deployment Client al Deployment Master deben estar autenticadas. Sin autenticación, las peticiones se rechazarían, imposibilitando la obtención de información a cualquier persona no autorizada. En el caso del despliegue sobre un cliente, cada cliente contará con un usuario propio que le permitirá ejecutar tareas específicas, como la sincronización (RF03).

Identificador	RNF02
Nombre	Recuperación ante fallos de actualización
Característica	Fiabilidad
Descripción	En caso de producirse un fallo durante la ejecución de una actualización, independientemente de su causa, el sistema debe ser capaz de restaurar el estado previo a dicha actualización. Esto garantiza que la máquina del cliente permanezca en un estado determinista y consistente.

Identificador	RNF03
Nombre	Actualizaciones en caliente
Característica	Fiabilidad
Descripción	Las actualizaciones en caliente hacen referencia a aquellos procesos de actualización que no requieren la detención de los servicios en ejecución. De este modo, se garantiza la disponibilidad continua de la aplicación para los usuarios, evitando interrupciones en el servicio.

Identificador	RNF04
Nombre	Registro de operaciones
Característica	Mantenibilidad
Descripción	Todas las operaciones realizadas, independientemente de su naturaleza, que puedan ser relevantes para la resolución de incidencias, deben ser registradas mediante mecanismos de logs. Estos registros deben almacenarse tanto en el Deployment Master como en el Deployment Client, a fin de facilitar el análisis y la resolución de problemas por parte del proveedor y del administrador del sistema, respectivamente.

Identificador	RNF05
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, con el fin de guardar coherencia con el resto de los microservicios para poder ser mantenido por personas que hayan trabajado con otros de ellos anteriormente.

Identificador	RNF06
Nombre	Estructura del proyecto
Característica	Mantenibilidad y Flexibilidad
Descripción	La organización del proyecto, tanto a nivel de carpetas como de clases, debe seguir un esquema coherente con el utilizado en el resto de los microservicios. Se considera que dicha coherencia se cumple si una persona con conocimiento previo de la estructura de referencia puede entenderla en un plazo no mayor a 10 minutos. Al mantener la misma estructura, también se facilita el desarrollo de un mecanismo genérico para el despliegue de los microservicios, lo que justifica la referencia a la característica de flexibilidad.

4.3 Diseño

En lo que respecta al diseño de la solución, en primer lugar se describirán una serie de conceptos previos. A continuación, se presentarán todos los componentes involucrados en la implementación de los casos de uso y requisitos previamente definidos, así como la interacción entre ellos. Asimismo, se explicarán las distintas capas que conforman un microservicio, el funcionamiento de un *launcher* implementado utilizando Uno Platform

y la estructura del servidor WebAssembly que lo encapsula. Por último, se expondrá en detalle el diagrama de un launcher desplegado sobre el sistema de un cliente.

4.3.1. Conceptos preliminares

Con el fin de facilitar la comprensión de los componentes descritos en secciones posteriores, en este apartado se introducen una serie de conceptos técnicos y aspectos fundamentales del sistema. Estos elementos son necesarios para entender con mayor claridad tanto el funcionamiento como las responsabilidades de los distintos componentes que participan en la arquitectura propuesta.

Secure Sockets Layer

Secure Sockets Layer (SSL) [23] es un protocolo criptográfico que opera al nivel de la capa de transporte del modelo *Open Systems Interconnection* (OSI[2]), cuya función principal es cifrar las comunicaciones extremo a extremo. A su vez, permite verificar la identidad del servidor utilizando certificados digitales, estos también se pueden utilizar a nivel del cliente y han de ser renovados periódicamente.

Proxy Inverso

Un proxy inverso sirve como intermediario entre un cliente y un servidor web, se sitúa en medio de la comunicación y es capaz de recibir las solicitudes del cliente y redireccionarlas a los servidores correspondientes [19]. Todas estas redirecciones se suelen dar utilizando rutas conocidas, entendiendo como conocidas aquellas que el mantenedor o desarrollador del proxy inverso quiere utilizar para redireccionar las peticiones del cliente. Se puede ver en la figura 4.6 el flujo de peticiones que se suele seguir.

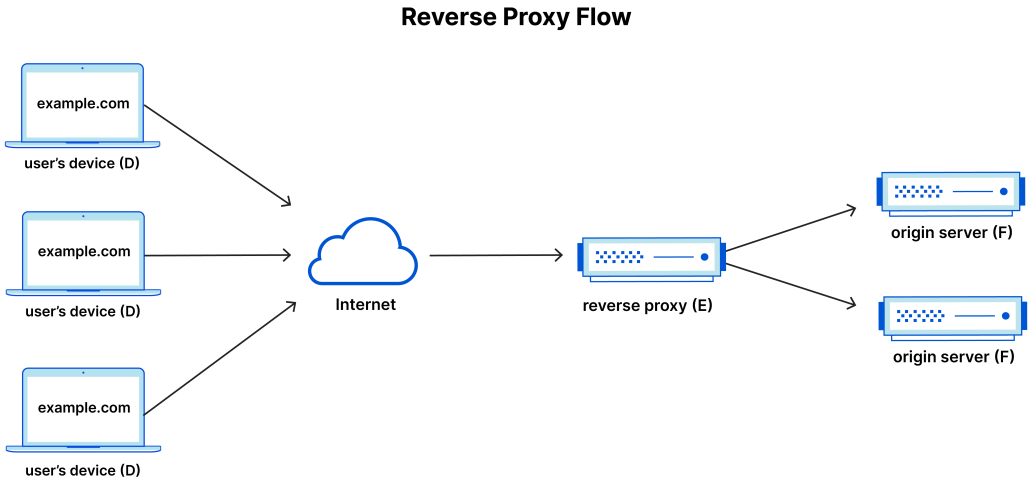


Figura 4.6: Flujo de un proxy inverso. [19]

.

Normalmente este tipo de tecnología suele ser utilizada por cualquier persona que proporciona un servicio en la red, debido a sus numerosas ventajas que ofrece. En primer

[2]Explicación del modelo OSI en detalle: https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/

lugar, reduce la cantidad de accesos o entradas permitidas desde internet al sistema donde se despliega el proxy. Ofrece la posibilidad de implementar un sistema de balanceo de carga que no dependa del servicio dispuesto. Permite a su vez, la separación de máquinas, pudiendo desplegar el proxy por separado de los servicios. También, mejora la protección frente a ataques dirigidos [3] y permite el cifrado y descifrado de las solicitudes mediante SSL.

Ensamblados manejados y no manejados

Un ensamblado no manejado es un archivo de tipo *Dynamic Link Library* (DLL) que contiene código y datos compilados que pueden ser utilizados simultáneamente por múltiples programas. Estas bibliotecas son generadas habitualmente a partir de lenguajes como `C` o `C++`, y contienen código máquina nativo, listo para ser ejecutado directamente por el sistema operativo mediante llamadas a la API de Windows (*Win32*). Este tipo de bibliotecas favorece la modularización del software, la reutilización de componentes y un uso más eficiente de la memoria y del espacio en disco, lo que contribuye a una mayor eficiencia en la ejecución de las aplicaciones.

Por otro lado, un ensamblado manejado también comparte la misma extensión `.dll` pero se genera a partir de lenguajes como `C#` y no contienen código máquina nativo, sino que almacenan código intermedio (*Intermediate Language*, IL[4]), junto con metadatos que describen el ensamblado. Este tipo de archivo, debe ser ejecutado por un entorno de ejecución gestionado, como el *Common Language Runtime* (CLR [5]).

A pesar de compartir la misma extensión, los ensamblados manejados y no manejados presentan diferencias fundamentales en cuanto a su estructura, portabilidad y mecanismos de ejecución. Mientras que los no manejados son específicos de Windows y de la arquitectura para la que fueron compiladas, los ensamblados manejados pueden ejecutarse en diferentes plataformas, siempre que exista un entorno de ejecución compatible.

Assembly Load Context

Todas las aplicaciones en el entorno de .NET Core utilizan un contexto de carga de ensamblados, y se utiliza para localizar, cargar y cachear los ensamblados manejados, así como otras dependencias que puedan tener. Cada vez que una dependencia se carga en el sistema, una instancia del contexto de carga de ensamblados es invocada para localizar dicha dependencia. En el marco de este trabajo, se emplea un *Assembly Load Context* personalizado para la carga de ensamblados manejados y no manejados, descritos en el apartado anterior.

Windows Service

Un *Windows Service* es una aplicación que se ejecuta en segundo plano en sistemas operativos Windows utilizando un *background service instance*. Este no requiere de intervención directa del usuario y no tiene interfaz gráfica, minimiza los errores del administrador del sistema al evitar la interacción. Estos servicios están diseñados para iniciar

[3] Es un tipo de ciberataque silencioso y dirigido a un individuo, organización o sistema informático, que tiene como objetivo comprometer la seguridad del mismo u obtener información de manera fraudulenta.

[4] Lenguaje de bajo nivel generado a partir de código fuente de alto nivel, utilizado como representación intermedia antes de su compilación final a código máquina.

[5] Es utilizado en el entorno de .NET para la transformación del código intermedio a código máquina nativo, y su posterior ejecución proporcionando servicios como recolección de basura o manejo de excepciones.

automáticamente al arrancar el sistema, incluso antes de que un usuario haya iniciado sesión, y pueden ejecutarse bajo distintas cuentas del sistema.

4.3.2. Componentes que participan y su interacción

En el proceso de despliegue intervienen múltiples componentes, cada uno tiene sus propias responsabilidades y es indispensable para el conjunto. Esto quiere decir que si uno de los componentes falla, el despliegue también lo hará. Por tanto, es muy importante dividir las responsabilidades y diseñarlos de una manera no bloqueante en caso de fallos menores para el proceso.

Los componentes que deben definirse se muestran en la figura 4.7. Para cada uno de ellos se incluye una breve descripción, sus responsabilidades principales y, finalmente, la forma en que interactúan con el resto de los componentes del sistema.

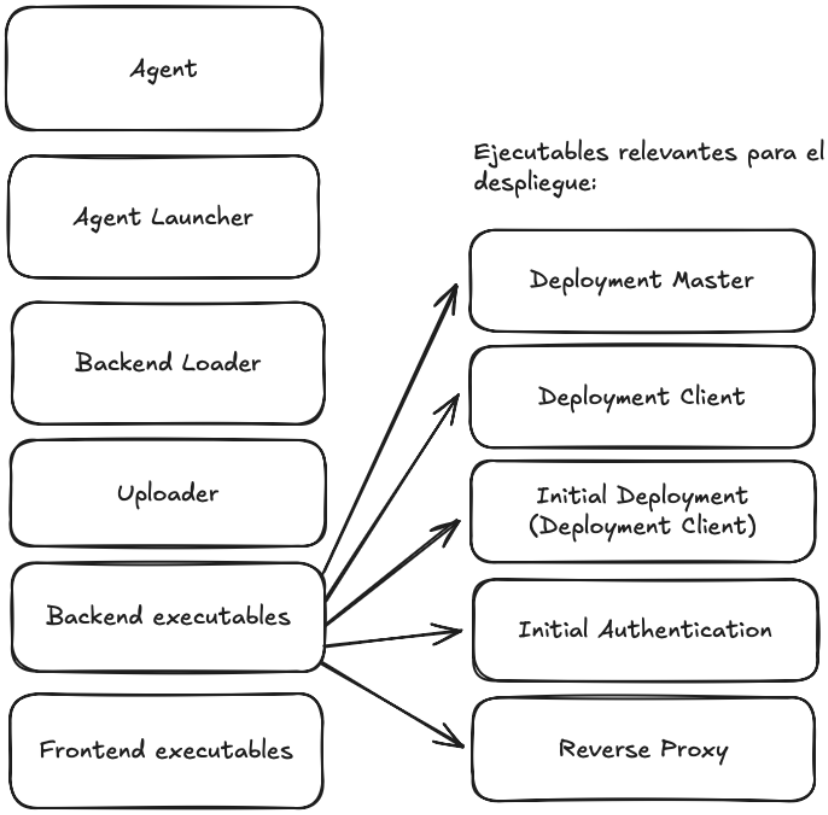


Figura 4.7: Diagrama conciso de los componentes que participan en el despliegue.

.

Agent

Descripción El *Agent* es un microservicio con las mismas capas que todos los microservicios descritos en este proyecto (se detallarán más adelante), y es parecido al Worker Process (w3wp.exe) explicado anteriormente en la sección 2.1.2. Su función principal es la carga de ejecutables de cualquier tipo en la máquina del cliente.

Responsabilidades

- Iniciar y detener los procesos asociados a los ejecutables, tanto de tipo backend como frontend.

- Comprobar el estado de los procesos que ha iniciado previamente.
- Descargar, localizar y extraer archivos de tipo ZIP en el sistema de ficheros.
- Modificar su propia configuración.

Componentes con los que interactúan Funciona como receptor de las solicitudes emitidas por Deployment Client para el inicio y parada de ejecutables. Se encarga de instanciar los *BackendLoader* correspondientes a cada ejecutable de tipo backend, y de poner en marcha los microservicios iniciales, estos son el *Initial Authentication* y *Initial Deployment*.

Agent Launcher

Descripción El *Agent Launcher* se instala como un servicio de Windows en la máquina cliente y actúa como único punto de acceso externo (un solo puerto abierto), funcionando a su vez como proxy inverso. Es necesario para separar las responsabilidades de actualización sobre componentes más importantes como el Agent.

Responsabilidades

- Redireccionar rutas HTTP conocidas, incluyendo la adición y eliminación de estas.
- Realizar operaciones básicas sobre el sistema de archivos, como copiar y pegar archivos de configuración iniciales.
- Iniciar procesos simples en el entorno del cliente.
- Asegurar su correcto funcionamiento tras reinicio de máquina.

Componentes con los que interactúan El Agent Launcher recibe peticiones del Deployment Client para llevar a cabo el proceso de actualización del Agent. Durante este proceso, interactúa con los microservicios iniciales Initial Authentication e Initial Deployment, que le proporcionan la información necesaria y garantizan la autenticación de todas las peticiones. Además, al estar implementado como un servicio de Windows, asegura que el Agent se reinicie automáticamente con la versión actual tras un reinicio del sistema.

Backend Loader

Descripción El *Backend Loader* es una aplicación de consola que se encarga de envolver a los ejecutables de tipo backend en un contexto de carga personalizado, asegurando la carga dinámica de los ensamblados que los componen.

Responsabilidades

- Proporcionar un *Assembly Load Context* personalizado para cargar ensamblados a demanda.
- Cargar en memoria los ensamblados evitando el sistema de ficheros.
- Desencriptar *tokens* de autorización.

Componentes con los que interactúan Son desplegados a través del Agent y encapsulan a los ejecutables backend. Su función principal es desencriptar los argumentos recibidos por el Agent y comunicárselos al ejecutable correspondiente. Para permitir la carga dinámica, deben interceptar las solicitudes de ensamblados requeridos por el ejecutable y redirigirlas al componente de Deployment Client, encargado de proporcionar los ensamblados a cargar.

Uploader

Descripción El *Uploader* es una aplicación de consola de uso exclusivo interno, es decir, no se desplegará en ninguna máquina. Su propósito principal, como indica su nombre, es la carga de artefactos. Esta herramienta es utilizada por las pipelines de CI/CD, las cuales, a partir del código fuente del repositorio, instruyen al Uploader sobre los artefactos que deben ser cargados.

Responsabilidades

- Carga de artefactos en la base de datos a través de endpoints.

Componentes con los que interactúa Carga los ejecutables del frontend, los ejecutables del backend y los Backend Loader en la base de datos del Deployment Master.

Frontend executables

Han sido mencionados anteriormente como ejecutables de tipo frontend, constituyen la interfaz de usuario del microservicio, la cual se comunica con su correspondiente backend. No se van a exponer las responsabilidades y componentes con los que interactúan ya que estas dependen del microservicio que envuelvan.

Backend executables

Han sido mencionados anteriormente como ejecutables de tipo backend, representan la lógica de negocio del microservicio, es decir, las acciones que se exponen al exterior a través de una API. No se van a exponer las responsabilidades y componentes con los que interactúan ya que estas dependen del microservicio que envuelvan.

Deployment Master

Descripción Como se ha mencionado en el RF02, el Deployment Master corresponde al mismo microservicio de Deployment, pero operando con el rol de *master*. Este rol únicamente determina las acciones que el microservicio puede ejecutar. El Deployment Master estará desplegado en nuestra infraestructura.

Responsabilidades

- Almacenar toda la información relacionada con los productos, manteniendo un historial de versiones.
- Registrar todos los entornos de cliente junto con su correspondiente código de entorno.

- Asociar cada versión disponible del producto con el código de entorno de cada cliente.

Componentes con los que interactúa El Deployment Master recibe peticiones del Uploader para almacenar artefactos en su base de datos y se comunica con el Deployment Client para sincronizar las entidades correspondientes a la versión que se desea desplegar en la infraestructura del cliente.

Deployment Client

Descripción El Deployment Client corresponde al mismo microservicio de Deployment, pero operando con el rol de *client*. Este rol únicamente determina las acciones que el microservicio puede ejecutar. El Deployment Client estará desplegado en la infraestructura del cliente.

Responsabilidades

- Desplegar una actualización en la máquina donde se encuentre instalado.
- Sincronizar la última actualización disponible y comprobar el estado de la actual.
- Almacenar el estado de las ejecuciones.
- Proporcionar ensamblados para la carga dinámica.
- Actuar de repositorio central para las configuraciones de los ejecutables.

Componentes con los que interactúa Interactúa con el Deployment Master para obtener la última actualización disponible y sincronizar sus entidades. Asimismo, se comunica con el Agent Launcher para enviarle peticiones de actualización, ya sea del Agent o de los microservicios iniciales. La actualización se entiende como un proceso que no solo actualiza el producto, sino también los componentes que hacen posible su despliegue.

También se encarga de registrar y gestionar el estado de todos los ejecutables desplegados en la máquina, ya sea en ejecución, detenidos o con errores. De esta manera, se mantiene un historial de las ejecuciones.

Reverse Proxy

Descripción Se trata de un microservicio genérico, al igual que los demás, que utiliza la biblioteca *Yet Another Reverse Proxy* [6] (YARP) para la implementación de un proxy inverso. Esta librería, desarrollada específicamente para el entorno `C#`, resulta especialmente adecuada para el requisito RNF05. Además, YARP es ampliamente adoptado y recomendado dentro del entorno `.NET` por su integración nativa y su flexibilidad.

Responsabilidades

- Redirigir todas las peticiones internas entre microservicios y componentes dentro de la máquina del cliente.
- Exigir autenticación a aquellas rutas que lo requieran.

[6] Página web oficial de YARP: https://github.com/dotnet/yarp

Componentes con los que interactúan Interactúa con el otro proxy inverso del sistema, el Agent Launcher, y con los demás ejecutables desplegados, interceptando y redireccionando peticiones.

Initial Authentication e Initial Deployment

Se trata de los microservicios iniciales, los cuales son tan extensos como los microservicios origen, en este caso, Authentication y Deployment Client, pero se publican junto con el Agent para formar un instalador que el cliente pueda descargar y utilizar para iniciar el proceso de instalación por primera vez. Como se ha mencionado anteriormente, Initial Authentication es necesario para verificar que el usuario encargado de la instalación posee los permisos requeridos, y Initial Deployment se utiliza para sincronizar lo necesario con el Deployment Master y obtener una versión mínima de la actualización a desplegar.

Una vez que estos microservicios iniciales han cumplido su cometido, se detienen los procesos correspondientes y se inician los microservicios completos, Authentication y Deployment Client, esta vez incorporando el Backend Loader respectivo para proporcionar la carga dinámica de ensamblados.

Interacción final

En la figura 4.8 se pueden ver todos los componentes que participan en el despliegue y cómo interaccionan entre ellos. Para la exposición se ha utilizado una estrategia de despliegue denominada *Blue-green Deployment* donde existe un servidor azul y un servidor verde, pero solo uno de ellos encargado de recibir peticiones. Esta estrategia resulta especialmente relevante para permitir actualizaciones en caliente RNF03, ya que posibilita la sustitución del agente cuando finaliza el proceso de actualización sin incurrir en interrupciones del servicio que puedan afectar a los usuarios finales.

Se puede observar la sincronización mencionada anteriormente, tanto por parte del Initial Deployment como del Deployment Client ya cargado con el Backend Loader, una vez finaliza el proceso de instalación inicial.

Los dos proxies inversos que intervienen en este flujo son, por un lado, el Agent Launcher, que es el componente que se comunica con el exterior y por ello se denomina *Main Reverse Proxy*, y por otro, el Reverse Proxy que se despliega como un Backend Executable.

También cabe mencionar que existen múltiples ejecutables de frontend y backend que pueden ser desplegados, en función del producto que se desea instalar y de los microservicios que lo conforman.

4.3.3. Diseño y estructura del microservicio

En esta sección se describe la arquitectura general del microservicio, así como las decisiones de diseño adoptadas para garantizar la coherencia con el resto de componentes del sistema [13]. En línea con el requisitos no funcional RNF06 definido previamente, todos los microservicios comparten una estructura común basada en una organización por capas, lo que favorece la mantenibilidad, escalabilidad y reutilización del código.

A continuación, se detalla cada una de las capas que conforman dicha estructura, poniendo especial énfasis en aquellas que adquieren un papel relevante en el microservicio en cuestión. Esta segmentación facilita la comprensión de las responsabilidades distribui-

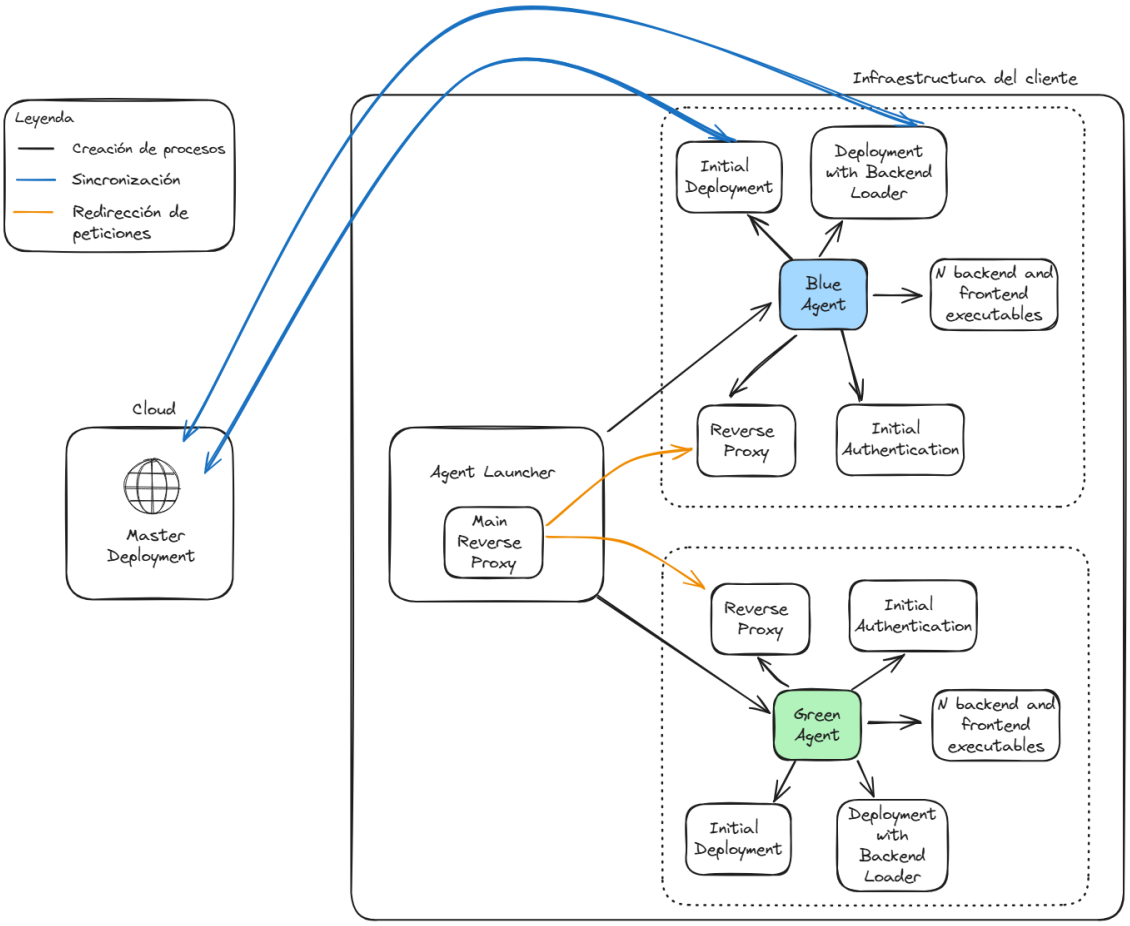


Figura 4.8: Diagrama completo de los componentes que participan en el despliegue y sus interacciones.

.

das entre los distintos niveles del sistema y permite distinguir claramente entre la lógica de negocio, el acceso a datos y los puntos de integración con otros servicios [14].

Estructura por capas

- **Dominio**: Contiene las entidades fundamentales del sistema, modeladas en base a la lógica de negocio. Estas entidades representan los conceptos clave del microservicio y constituyen la base sobre la que se construyen el resto de las capas.
- **Contratos**: Agrupa los objetos de transferencia de datos (DTO) y las interfaces públicas del backend. Su objetivo es desacoplar la lógica interna de la representación expuesta a otros microservicios o a la interfaz de usuario, garantizando así un mayor control sobre los datos que se intercambian.
- **Aplicación**: Esta capa incluye el soporte para operaciones CRUD, así como las comprobaciones de permisos sobre entidades y campos. Parte del contenido de esta capa es generado automáticamente mediante herramientas de modelado, lo que acelera el desarrollo y reduce errores.
- **Lógica**: Aloja tanto la lógica de negocio desarrollada manualmente como la generada a partir de modelos. Es en esta capa donde se implementan las funcionalidades específicas del microservicio, tales como el procesamiento, transformación y envío de datos.

- **Persistencia**: Encargada de gestionar el acceso a la base de datos. Aunque no todos los microservicios requieren persistencia de datos, aquellos que lo hacen utilizan esta capa para abstraer las operaciones relacionadas con almacenamiento, seguimiento de conflictos o registro de eventos.
- **Servicios**: Actúa como punto de entrada al microservicio, exponiendo los controladores y métodos accesibles a través de peticiones HTTP. Es el nexo entre las solicitudes externas y la lógica del sistema.
- **Proxy**: Permite la invocación de acciones del backend a través de llamadas HTTP. Esta capa es utilizada tanto por la interfaz de usuario como por otros microservicios para establecer comunicación con el servicio.
- **Referencias externas**: Gestiona las dependencias hacia otros servicios o microservicios. Su función principal es desacoplar el código de estas dependencias externas, permitiendo una mejor modularidad y evitando problemas derivados de dependencias cíclicas.

Este enfoque estructurado no solo promueve una mayor claridad en el desarrollo, sino que también permite la colaboración eficiente entre distintos perfiles dentro del equipo de trabajo, al delimitar con precisión las responsabilidades técnicas y funcionales de cada capa. En la figura 4.9 se observa como las diferentes capas interactúan entre sí.

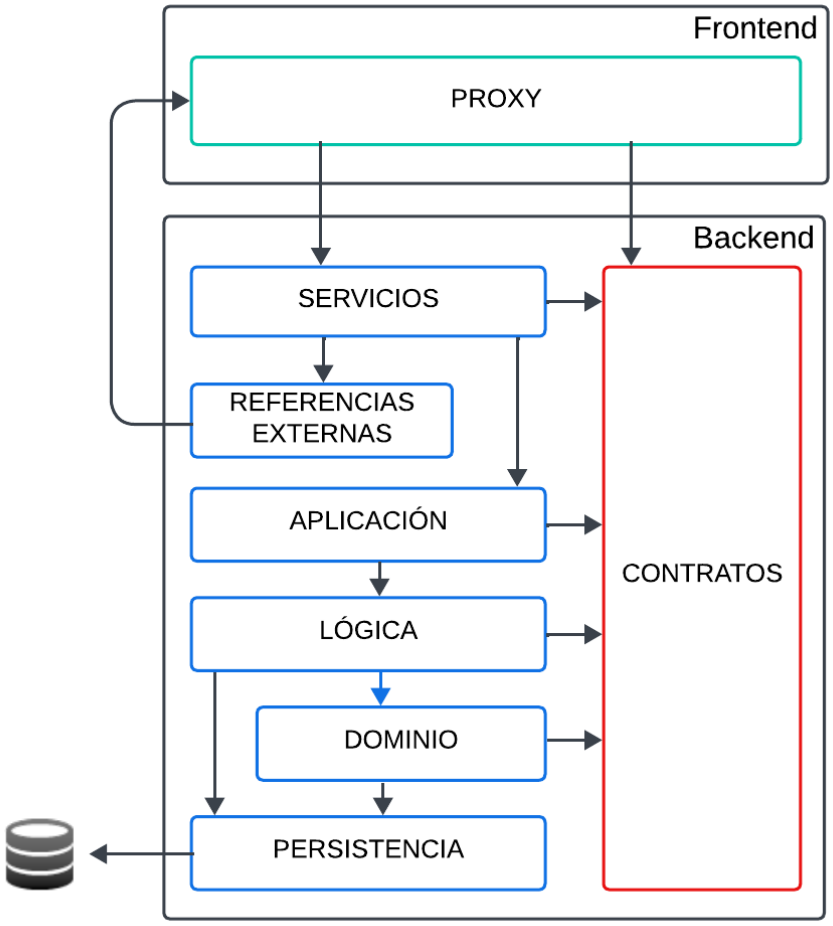


Figura 4.9: Estructura por capas del microservicio de Deployment.

.

Capa de lógica

Durante el desarrollo del microservicio de Deployment, la capa que más se ha modificado y ampliado es la capa de lógica, ya que el resto de capas son prácticamente todo código autogenerado gracias a herramientas de tipo DSL desarrolladas por parte de la empresa.

Esta perspectiva está basada en aplicar un desarrollo dirigido por modelos o *Model Driven Development* (MDD), que permite construir aplicaciones de una manera más visual abstrayendo muchas de las complejidades del código.

La estructura de la capa de lógica se presenta en la figura 4.10, donde se destacan las carpetas más relevantes:

- `Data managers`: clases cuya responsabilidad principal es ejecutar operaciones de tipo *create, remove, update y delete* (CRUD) sobre las entidades del dominio almacenadas en la base de datos. En la figura 4.11 se pueden observar ejemplos de estas clases. Asimismo, existe una subcarpeta denominada `validated`, que contiene versiones de estas operaciones que incorporan validaciones básicas antes de delegar la ejecución a los *data managers*. El objetivo de esta capa es mantener la lógica lo más sencilla y acotada posible.
- `Managers`: representan acciones *ad hoc* públicas (también denominadas AHA, por sus siglas en inglés: Ad Hoc Action). Se entiende por acción ad hoc una operación que no forma parte de las operaciones estándar (como CRUD), sino que responde a una necesidad específica de negocio o lógica de aplicación. Al ser pública, esta acción está expuesta como un endpoint accesible externamente. En estas clases se implementa la lógica necesaria para procesar la información recibida y producir un resultado concreto. La estructura de carpetas correspondiente se muestra en la figura 4.12.
- `Internal managers`: también representan acciones *ad hoc*, pero en este caso son de carácter privado. Es decir, no están expuestas como endpoints externos y solo pueden ser invocadas desde acciones públicas. Estas clases permiten encapsular y reutilizar lógica interna, evitando acoplar directamente los endpoints públicos con el detalle de implementación del microservicio. En la figura 4.13 se presentan ejemplos de este tipo de acciones privadas.
- `Role managers`: clases que emplean mecanismos de aspectos propios de .NET para separar responsabilidades en función de los distintos roles que puede asumir el microservicio en tiempo de ejecución. Tal como se mencionó anteriormente, en la figura 4.14 se ilustran los roles definidos y su implementación correspondiente.

4.3.4. Estructura de un servidor WebAssembly

El servidor WebAssembly presenta una estructura sencilla y minimalista, como se muestra en la figura 4.15. Este servidor se encuentra ubicado en la misma solución (`.sln`) que el componente denominado *launcher*, lo cual resulta coherente dado que su función principal es encapsular y gestionar la ejecución del mismo. Por ello, ambos proyectos se sitúan al mismo nivel jerárquico dentro de la solución. El proyecto consta de un número reducido de archivos, suficientes para implementar los adaptadores necesarios que permiten al sistema de despliegue interactuar con el servidor y habilitar una interfaz web accesible mediante los protocolos HTTP y HTTPS.

Entre los archivos más relevantes se encuentran:

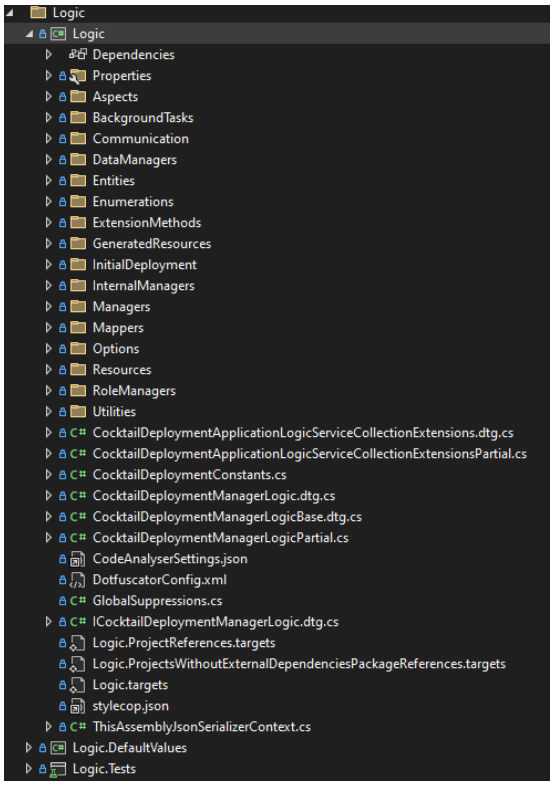


Figura 4.10: Estructura de carpetas de la capa de lógica.

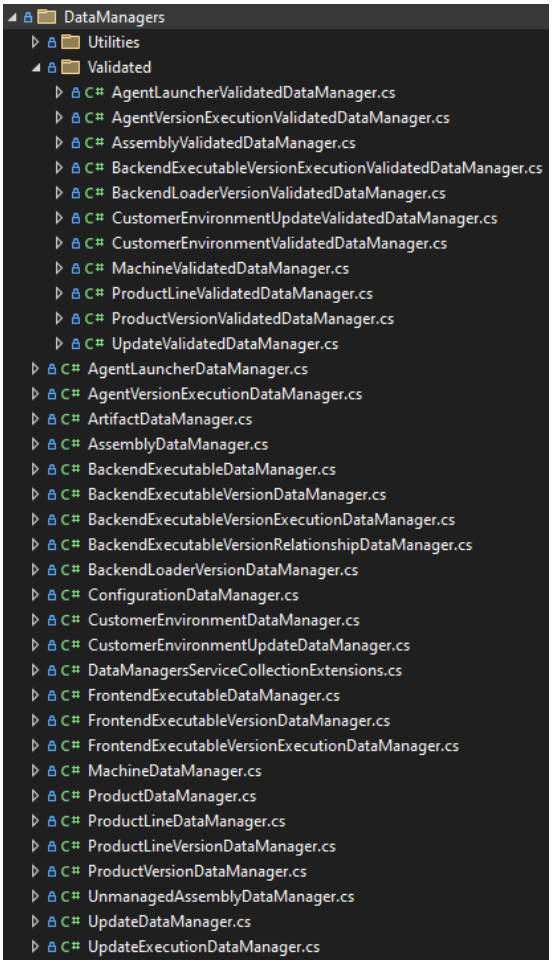


Figura 4.11: Estructura de carpetas de los *data managers* de la capa de lógica.

- `AppSettingsWebAppRunner`: permite ejecutar el servidor de manera local, es decir, directamente desde la solución de Visual Studio 2022. Este archivo facilita la configuración y pruebas en entornos de desarrollo.
- `DeploymentWebAppRunner`: contiene los comandos requeridos para iniciar el servidor en modo de despliegue. Este modo está diseñado para integrarse con el sistema de despliegue personalizado, permitiendo una ejecución controlada y automatizada del servidor WebAssembly en entornos de producción o infraestructura interna.

4.3.5. Despliegue de la UI

En este último apartado se presenta más en detalle solo el despliegue de la interfaz de usuario de los servidores WebAssembly, para ello, partimos de un diagrama de flujo que se expone en la figura 4.16 y que representa la interacción que tendría un usuario potencial de la aplicación con uno de los servidor WebAssembly.

En primer lugar, el usuario, a través de su navegador web, introduce la dirección URL del servicio que desea consumir. Dicha dirección corresponde a un identificador amigable y personalizable por el cliente mediante el uso de un *Servidor de Nombres de Dominio (DNS)*[7]. En caso de que el cliente no disponga de un servidor DNS propio, es posible

[7]Un servidor de nombres de dominio permite traducir nombres de dominio en direcciones IP, facilitando así la interacción de los usuarios con los servicios web.

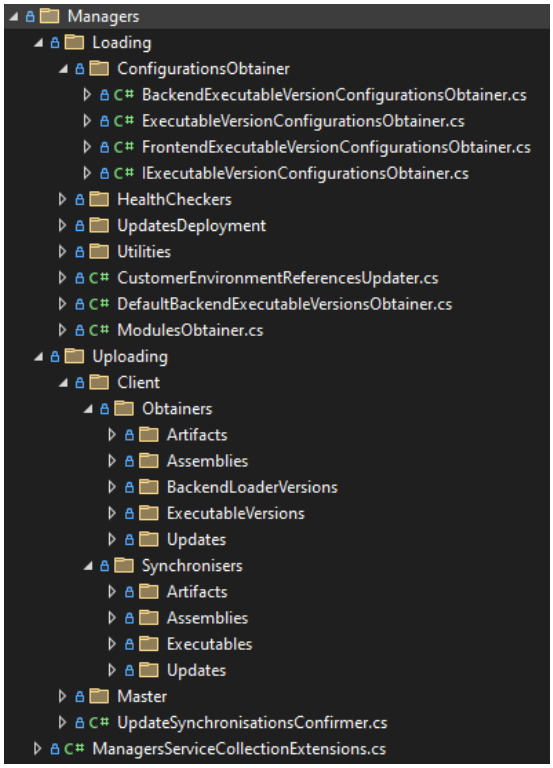


Figura 4.12: Estructura de carpetas de los *managers* de la capa de lógica.

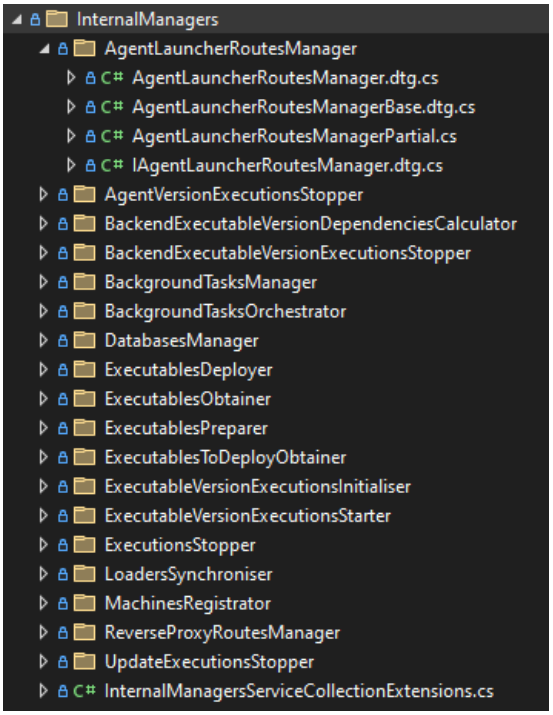


Figura 4.13: Estructura de carpetas de los *internal managers* de la capa de lógica.

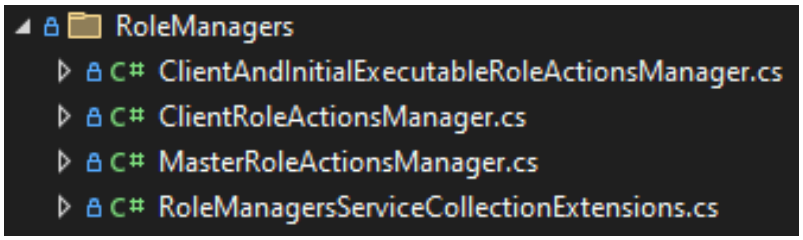


Figura 4.14: Archivos de la carpeta de *role managers* de la capa de lógica.

utilizar directamente el nombre de la máquina acompañado del puerto correspondiente, permitiendo así el acceso de los usuarios al servicio.

La dirección URL en cuestión apunta al Agent Launcher, que constituye el único elemento del sistema que expone su puerto hacia el exterior. El Agent Launcher tiene como función principal redirigir las solicitudes entrantes hacia el Reverse Proxy, el cual a su vez las canaliza al servidor WebAssembly.

En el diagrama correspondiente, también puede observarse cómo los procesos asociados a los microservicios de backend, como el Reverse Proxy, son inicializados por el componente denominado Backend Loader. En contraste, el servidor WebAssembly es iniciado directamente por el Agent, ya que no requiere una carga dinámica de ensamblados.

Durante la inicialización del proceso del servidor WebAssembly, el Agent registra la dirección del Reverse Proxy en la configuración del servidor mediante variables de entorno, las cuales son proporcionadas como argumentos al momento de iniciar dicho proceso.

Dentro del flujo de procesamiento de solicitudes en el servidor WebAssembly, se encuentra registrado un *middleware* encargado de actualizar dinámicamente la dirección del Reverse Proxy previamente configurada.

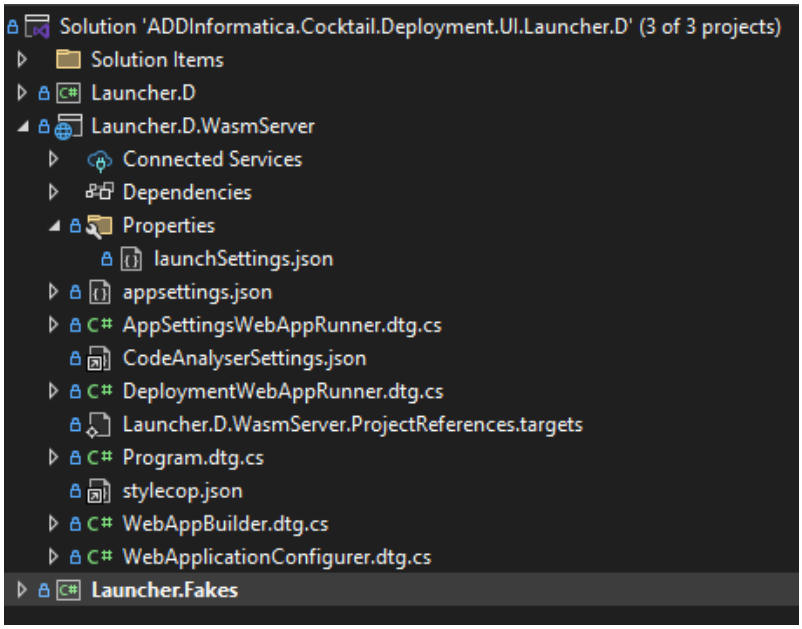


Figura 4.15: Estructura de carpetas del servidor WebAsssembly.

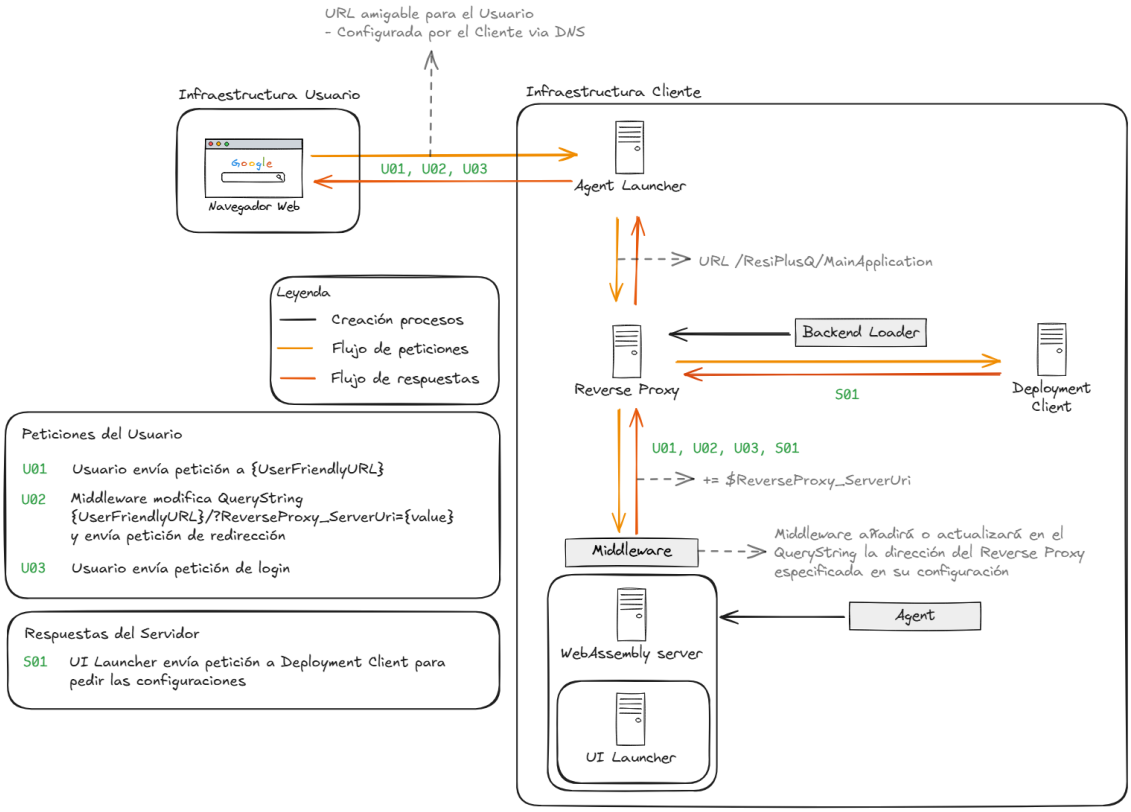


Figura 4.16: Digrama de flujo que representa la interacción entre un servidor WebAssembly y un usuario potencial.

Finalmente, una vez que el usuario ha iniciado sesión correctamente, el sistema genera un token de autenticación que contiene los permisos necesarios. Este token se utiliza para solicitar a Deployment Client la configuración restante requerida para completar de forma adecuada la inicialización del servidor web.

4.4 Implementación

Durante la fase de implementación no solo se desarrolló el microservicio de Deployment, sino que también se ampliaron, adaptaron e integraron todos los componentes definidos previamente en la fase de diseño. Este apartado se centrará exclusivamente en describir los pasos seguidos y los principales problemas encontrados durante el proceso de despliegue de los servidores WebAssembly.

4.4.1. Servidor WebAssembly

La unidad de despliegue en este contexto son los servidores WebAssembly. Tal como se mencionó anteriormente, estos encapsulan componentes launcher desarrollados con Uno Platform y proporcionan la interfaz de usuario de los microservicios a través de un endpoint principal accesible desde el navegador.

El primer paso consistió en comprender el funcionamiento interno de estos servidores y determinar cuál era la configuración mínima necesaria para que pudieran ser gestionados mediante el sistema de despliegue. A diferencia del backend de un microservicio tradicional, estos servidores presentan una estructura considerablemente más simple —como se explicó en su descripción técnica—, lo que los hace más fáciles de tratar desde el sistema de Deployment.

Para desplegar la parte backend de un microservicio convencional, es necesario que otros microservicios estén operativos, lo que introduce dependencias entre servicios. Esta situación plantea varios desafíos, ya que el sistema debe ser capaz de identificar dichas dependencias y orquestar el orden de despliegue adecuado para garantizar que todos los componentes funcionen correctamente.

En cambio, en el caso del frontend basado en WebAssembly, no existe tal dependencia. Estos servidores pueden iniciarse de forma autónoma, sin necesidad de que ninguna API esté activa. Si bien en ese estado las solicitudes enviadas desde el cliente no obtendrán respuesta, este comportamiento no afecta al proceso de despliegue en sí. Desde la perspectiva del sistema de Deployment, esto representa una gran ventaja, ya que simplifica el proceso al eliminar la necesidad de gestionar relaciones entre servicios.

Configuración

La configuración en un microservicio es un aspecto esencial, ya que representa un conjunto de valores dinámicos que pueden variar en función del entorno de ejecución. Estos valores incluyen, por ejemplo, direcciones de servicios externos, configuración de proxies o parámetros relacionados con los recursos de la máquina, como los que afectan al rendimiento del almacenamiento en caché o al tiempo de respuesta del sistema.

En el caso concreto de este sistema, que emplea dos componentes principales —el servidor y el launcher—, la configuración del frontend presentaba particularidades adicionales en comparación con la del backend. Específicamente, la configuración del servidor WebAssembly solo resultaba relevante en escenarios donde este fuera desplegado bajo un servidor web tradicional, como IIS. Dado que este no era el caso en nuestro entorno, dicha configuración resultaba innecesaria y se descartó. Esto se implementó mediante la exclusión explícita del archivo de configuración correspondiente en el archivo del proyecto (`.csproj`), como se muestra en la figura 4.17. De este modo, se garantizaba que el archivo no se generara durante el proceso de compilación o publicación del servidor.

```
<ItemGroup>
  <!-- The appsettings files are excluded from the publication as this project is only deployed using the Deployment
    microservice system and configuration is obtained from there. -->
  <Content Update="appsettings*.json" CopyToPublishDirectory="Never"/>
</ItemGroup>
```

Figura 4.17: Exclusión del `appsettings` durante el proceso de publicación.

La configuración verdaderamente relevante era la del launcher. En ella se especifican, entre otros aspectos, las rutas base de los microservicios con los que necesita interactuar, por ejemplo, para realizar procesos de autenticación del usuario.

Sin embargo, esta configuración presentaba un problema importante. Para entenderlo, conviene recordar cómo funciona el archivo `appsettings.json` en el contexto de .NET [5]. Este archivo contiene los parámetros de configuración de la aplicación y suele ubicarse en la raíz del proyecto. Durante la ejecución, su contenido se carga en un objeto de tipo `IConfiguration`, que es consumido por el entorno de ASP.NET para acceder a los valores desde el código de la aplicación.

Además del archivo principal, es habitual contar con versiones específicas del mismo archivo para cada entorno, denominadas `appsettings.Environment.json`, como `Development`, `Staging` o `Production`. Estos archivos también se cargan en tiempo de ejecución dependiendo del entorno configurado, permitiendo adaptar la aplicación a distintos contextos sin modificar el código fuente.

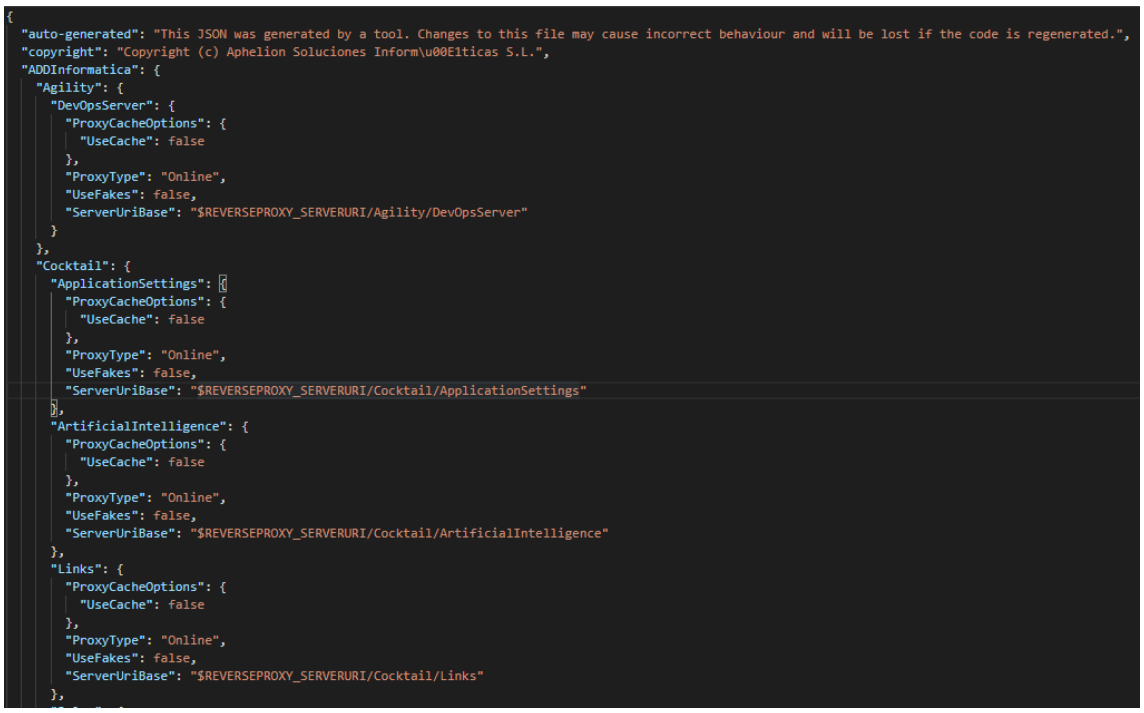
El problema detectado radicaba en que los archivos `appsettings` contenían valores específicos de máquinas internas del equipo de desarrollo, lo cual es una mala práctica, tal como se puede apreciar en la figura 4.18. En sistemas bien diseñados, este tipo de información debe gestionarse mediante variables de entorno. Estas variables, generalmente expresadas con una notación especial (por ejemplo, comenzando por el símbolo $), permiten reemplazar los valores en tiempo de ejecución y facilitan la portabilidad y el despliegue de la aplicación en diferentes entornos.

```
{
  "auto-generated": "This JSON was generated by a tool. Changes to this file may cause incorrect behaviour and will be lost if the code is regenerated.",
  "copyright": "Copyright (c) Aphelion Soluciones Inform\u00E1ticas S.L.",
  "ADDInformatica": {
    "Agility": {
      "DevOpsServer": {
        "ServerUriBase": "http://[redacted]:41100/Agility/DevOpsServer"
      }
    },
    "Cocktail": {
      "ApplicationSettings": {
        "ServerUriBase": "http://[redacted]:41100/Cocktail/ApplicationSettings"
      },
      "ArtificialIntelligence": {
        "ServerUriBase": "http://[redacted]:41100/Cocktail/ArtificialIntelligence"
      },
      "Links": {
        "ServerUriBase": "http://[redacted]:41100/Cocktail/Links"
      },
      "Rules": {
        "Designer": {
          "ServerUriBase": "http://[redacted]:41100/Cocktail/Rules/Designer"
        }
      },
      "Security": {
        "Authentication": {
          "ServerUriBase": "http://[redacted]:41100/Cocktail/Security/Authentication"
        }
      },
      "Telemetry": {
        "ServerUriBase": "http://[redacted]:41100/Cocktail/Telemetry"
      }
    }
  }
}
```

Figura 4.18: Fichero `appsettings.json` que contiene configuración de máquinas internas.

Para resolver este problema, se modificaron las plantillas utilizadas para la generación automática de los archivos de configuración. Estas plantillas, de tipo `.tt` (Text Template

Transformation Toolkit [8]), se adaptaron para incluir variables de entorno en lugar de valores estáticos, lo que permitió una configuración más flexible, segura y adecuada para entornos dinámicos de despliegue. Se puede ver un ejemplo de el fichero correctamente generado y utilizando variables de entorno en la figura 4.19.



```
{
  "auto-generated": "This JSON was generated by a tool. Changes to this file may cause incorrect behaviour and will be lost if the code is regenerated.",
  "copyright": "Copyright (c) Aphelion Soluciones Inform\u00E1ticas S.L.",
  "ADDInformatica": {
    "Agility": {
      "DevOpsServer": {
        "ProxyCacheOptions": {
          "UseCache": false
        },
        "ProxyType": "Online",
        "UseFakes": false,
        "ServerUriBase": "$REVERSEPROXY_SERVERURI/Agility/DevOpsServer"
      }
    },
    "Cocktail": {
      "ApplicationSettings": {
        "ProxyCacheOptions": {
          "UseCache": false
        },
        "ProxyType": "Online",
        "UseFakes": false,
        "ServerUriBase": "$REVERSEPROXY_SERVERURI/Cocktail/ApplicationSettings"
      },
      "ArtificialIntelligence": {
        "ProxyCacheOptions": {
          "UseCache": false
        },
        "ProxyType": "Online",
        "UseFakes": false,
        "ServerUriBase": "$REVERSEPROXY_SERVERURI/Cocktail/ArtificialIntelligence"
      },
      "Links": {
        "ProxyCacheOptions": {
          "UseCache": false
        },
        "ProxyType": "Online",
        "UseFakes": false,
        "ServerUriBase": "$REVERSEPROXY_SERVERURI/Cocktail/Links"
      },
```


Figura 4.19: Fichero `appsettings.json` que contiene variables de entorno.

Middleware de redirección

En escenarios convencionales, las aplicaciones suelen recibir parámetros mediante la línea de comandos. Sin embargo, en el caso del servidor WebAssembly, este mecanismo difiere: no se emplea una interfaz de línea de comandos (CLI) para proporcionar variables de entorno, sino lo que se denomina *query string*.

Un *query string* es la parte final de una URL y se utiliza para transmitir parámetros en forma de pares clave-valor, funcionando de manera análoga a los argumentos en la línea de comandos, pero adaptado al contexto de servidores web. La sintaxis típica de un query string es la siguiente: `?CLAVE1=VALOR1&CLAVE2=VALOR2`.

Durante la inicialización del servidor, es imprescindible pasarle un parámetro concreto: la dirección del proxy inverso. Este valor es fundamental para que el launcher pueda enviar las peticiones de autenticación al servicio correspondiente y obtener así un token con los permisos adecuados. Una vez adquirido el token, el launcher también requiere esta dirección para contactar con el microservicio de Deployment y obtener el resto de la configuración necesaria para su ejecución. En la figura 4.20 se puede ver cómo quedaría en el navegador.

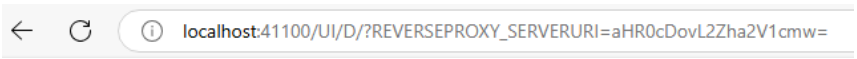


Figura 4.20: Dirección web que contiene el *query string* con la dirección del proxy inverso.

[8] Página web oficial *T4 Text Templates*: https://learn.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2022

Para lograr esto, se optó por implementar un *middleware* personalizado en ASP.NET. El objetivo principal de este componente era interceptar las peticiones entrantes y redirigirlas, añadiendo al destino el query string correspondiente con la dirección del proxy inverso (en este caso, la proporcionada por el Agent Launcher). Para implementar este comportamiento correctamente, fue necesario distinguir entre dos mecanismos relacionados: *URL Rewrite* y *URL Redirect*.

1. **URL Rewrite**: Consiste en la reescritura interna de la URL recibida por el servidor, sin cambiar la URL visible para el usuario. Se utiliza frecuentemente para modificar rutas de manera dinámica dentro del mismo dominio, facilitando estructuras más limpias o adaptativas.

2. **URL Redirect**: Redirige al cliente hacia una nueva URL. El navegador del usuario es notificado del cambio, y se realiza una nueva solicitud hacia la nueva dirección. Este mecanismo es habitual, por ejemplo, en situaciones de mantenimiento o migración de servicios.

En un principio, la estrategia preferida era emplear únicamente *URL Rewrite*, evitando así la necesidad de modificar manualmente el query string proporcionado por el usuario. Esta decisión se basaba en consideraciones de seguridad ya que al utilizar redirección se puede exponer al sistema a ataques, como la suplantación de identidad del servidor. Además, los parámetros añadidos quedarían visibles para el usuario final, lo que puede representar un riesgo si contienen información sensible.

No obstante, durante las pruebas se observó que el launcher no interpretaba correctamente las variables de entorno cuando se utilizaba exclusivamente *URL Rewrite*. Tras un análisis más profundo, se identificó que el problema no radicaba en el *middleware*, sino en el propio framework Uno Platform. Este último extraía los parámetros únicamente de la URL presente en la ventana del navegador, ignorando cualquier modificación interna realizada por el servidor. Debido a esta limitación, se descartó el uso exclusivo de reescritura y se reportó una incidencia en el repositorio oficial de Uno Platform en GitHub.

Como solución final, se optó por una estrategia híbrida, combinando mecanismos de reescritura y redirección. Esta aproximación permitía garantizar que el launcher recibiera correctamente las variables de entorno mediante el query string. No obstante, se extremaron las precauciones para evitar posibles vulnerabilidades: todos los parámetros incluidos en el query string debían ser estrictamente estáticos y definidos de antemano, con el fin de evitar manipulaciones externas.

En la figura 4.21 se muestra la implementación final del *middleware* personalizado encargado de gestionar la inclusión de parámetros mediante *query string*. Esta pieza de software intermedia intercepta las solicitudes entrantes y permite añadir o modificar parámetros en la URL antes de que estas sean procesadas por el resto de la aplicación.

Complementariamente, en la figura 4.22 se presenta la definición del método de extensión correspondiente sobre la interfaz `IApplicationBuilder`. Este tipo de métodos permiten encapsular la lógica de registro del middleware en una única llamada reutilizable y clara.

Gracias a esta definición, se puede integrar el middleware de forma sencilla en cualquier aplicación ASP.NET que implemente la interfaz mencionada, como es el caso concreto de `WebApplication`, que se utiliza para construir el servidor WebAssembly. Esta estructura facilita la extensibilidad y mejora la legibilidad del código al abstraer la lógica de configuración en un punto centralizado.

En la figura 4.23 vemos como se integra la funcionalidad implementada en el sistema de procesamiento de solicitudes de ASP.NET que funciona en base a una tubería, donde la petición va atravesando los distintos middlewares registrados.

```
/// <summary>
///     Query string modifier middleware. This middleware modifies the query string adding the necessary parameters,
///     and then send an HTTP redirect. If the parameters are already in the query string, nothing is done.
/// </summary>
public class QueryStringModifierMiddleware
{
    private readonly RequestDelegate next;
    private readonly IConfiguration configuration;

    /// <summary> Initializes a new instance of the <see cref="QueryStringModifierMiddleware"/> class. </summary>
    /// <param name="next"> The next request. </param>
    /// <param name="configuration"> The configuration. </param>
    public QueryStringModifierMiddleware(RequestDelegate next, IConfiguration configuration)
    {
        this.next = next;
        this.configuration = configuration;
    }

    /// <summary> Invokes the middleware action. </summary>
    /// <param name="context"> The HTTP request context. </param>
    /// <returns> A task with no additional data. </returns>
    public async Task InvokeAsync(HttpContext context)
    {
        // TODO: Add the FrontendExecutableVersionId when launchers register Deployment's ConfigurationProvider.
        string? reverseProxyServerUri =
            this.configuration.GetValue<string>("ADDInformatica:Cocktail:ReverseProxy:ServerUri");

        // The check is done to avoid having to redirect if the received argument is correct.
        if (string.IsNullOrEmpty(reverseProxyServerUri)
            || IsReverseProxyServerUriEqual(context.Request.Query, reverseProxyServerUri))
        {
            await this.next.Invoke(context).ConfigureAwait(false);

            return;
        }

        string redirectURL = $"?ReverseProxy:ServerUri={reverseProxyServerUri.TransformFromStringToBase64()}";

        if (this.configuration.GetValue<string>("UrlPrefix") is string urlPrefix)
        {
            redirectURL = $"{urlPrefix}/{redirectURL}";
        }

        context.Response.Redirect($"/{redirectURL}", permanent: false);

        await this.next.Invoke(context).ConfigureAwait(false);
    }
```

Figura 4.21: Implementación del *middleware* de redirección y reescritura personalizado.

4.4.2. Proxy inverso

La forma en que el *launcher* obtiene los permisos necesarios para realizar peticiones autenticadas consiste en una pantalla inicial de inicio de sesión, donde se realiza una solicitud al microservicio de autenticación para obtener el correspondiente token de acceso. Sin embargo, cuando entre el cliente y el servidor se encuentra un proxy inverso que requiere autenticación previa, surge un problema: si el usuario no puede acceder siquiera a esa pantalla inicial, no podrá obtener ningún token, lo que bloquea completamente el flujo de autenticación.

Ante este escenario, se optó por permitir que todas las solicitudes dirigidas a los servidores WebAssembly desplegados se pudieran realizar sin necesidad de autenticación previa. Esto se justificó por el hecho de que el propio *launcher* ya controla el acceso a

```
//-----------------------------------------------------------------------
// <copyright file="QueryStringModifierMiddlewareExtensions.cs" company="ADD Informática">
//     Copyright (c) Aphelion Soluciones Informáticas S.L.
// </copyright>
//-----------------------------------------------------------------------

namespace ADDInformatica.Cocktail.Core.UI.UP.Middlewares.ExtensionMethods
{
    using Microsoft.AspNetCore.Builder;

    /// <summary> Query string modifier middleware extension method for registering the middleware. </summary>
    public static class QueryStringModifierMiddlewareExtensions
    {
        /// <summary>
        ///     Registers the Query String middleware that modifies the query string when needed.
        /// </summary>
        /// <param name="builder"> The builder to act on. </param>
        /// <returns> The application builder with the middleware added. </returns>
        public static IApplicationBuilder UseQueryStringModifier(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<QueryStringModifierMiddleware>();
        }
    }
```

Figura 4.22: Método extensor de un *IApplicationBuilder* para registrar el *query string modifier middleware*.

```
/// <summary> Configurer of the <see cref="WebApplication"/>. </summary>
[System.CodeDom.Compiler.GeneratedCode("AgilityCodeGenerator", "1.0.0.0")]
public class WebApplicationConfigurer
{
    /// <summary> Initializes a new instance of the <see cref="WebApplicationConfigurer"/> class. </summary>
    /// <param name="configuration"> The configuration. </param>
    public WebApplicationConfigurer(IConfiguration configuration)
    {
        this.Configuration = configuration;
    }

    /// <summary> Gets the configuration. </summary>
    /// <value> The configuration. </value>
    public IConfiguration Configuration { get; }

    /// <summary> Configures the <see cref="WebApplication"/>. </summary>
    /// <param name="application"> The <see cref="WebApplication"/> to configure. </param>
    /// <param name="urlPrefix"> Prefix of the URL of the WebAssembly Server. </param>
    public static void Configure(WebApplication application, string urlPrefix)
    {
        application.UsePathBase($"/{urlPrefix}");

        application.UseUnoFrameworkFiles();

        application.UseStaticFiles();

        application.UseQueryStringModifier();

        application.UseRouting();

        application.MapFallbackToFile("index.html");
    }
```

Figura 4.23: *Pipeline* que procesa las peticiones sobre un servidor WebAssembly con el *middleware* registrado.

las siguientes pantallas del sistema, exigiendo que el usuario haya iniciado sesión para poder continuar.

Para implementar esta lógica, fue necesario modificar el microservicio correspondiente al Reverse Proxy —descrito previamente en la sección de diseño— para admitir rutas "anónimas". Estas se definen como aquellas a las que no se les aplica ninguna política de autorización, permitiendo el acceso directo desde el exterior sin necesidad de un token. En la figura 4.24 se muestra cómo se definen los patrones de este tipo de rutas, así como el método auxiliar que permite identificarlas fácilmente.

```
/// <summary> Gets the anonymous request patterns. </summary>
/// <value> The anonymous request patterns. </value>
private static string[] RequestPatterns =>
[

    // UI desktop launchers do not need authentication, as their first form is the LogIn screen.
    "UI/Launcher/D",
];

/// <summary> Gets the anonymous subroutes from a requests base path. </summary>
/// <param name="requestsBasePath"> The requests base path of the route. </param>
/// <returns> The anonymous subroutes. </returns>
public static IReadOnlyCollection<string> GetAnonymousSubRoutes(string requestsBasePath)
{
    return [.. RequestPaths.Where(s => s.StartsWith(requestsBasePath))];
}

/// <summary> Queries if a route is anonymous. </summary>
/// <param name="requestsBasePath"> The requests base path of the route. </param>
/// <returns> True if it is an anonymous route, false if not. </returns>
public static bool IsAnonymousRoute(string requestsBasePath)
{
    if (string.IsNullOrWhiteSpace(requestsBasePath))
    {
        return false;
    }

    return RequestPatterns
        .Any(s => requestsBasePath
            .Contains(s, System.StringComparison.InvariantCulture));
}
```

Figura 4.24: Patrones de rutas anónimas definidas en el Reverse Proxy.

La implementación del *reverse proxy* se llevó a cabo mediante YARP (Yet Another Reverse Proxy). En este contexto, una *ruta* define las condiciones bajo las cuales una solicitud entrante debe ser procesada (por ejemplo, mediante coincidencia de patrones en la URL), mientras que un *clúster* representa el conjunto de destinos posibles a los que se puede reenviar dicha solicitud. Las rutas actúan como filtros lógicos, y los clústeres como destinos configurables que permiten distribuir la carga. En la figura 4.25, se ilustra cómo se identifican las rutas anónimas y se establece una política de autorización nula tanto en la creación de la ruta como del clúster asociado.

4.4.3. Registro de logs en el Agent Launcher

El *Agent Launcher* es el componente inicial del sistema, es decir, el primero en ser desplegado en la máquina del cliente. Debido a esta característica, presenta un sistema de registro de logs distinto al del resto de componentes. A diferencia de los demás microservicios, que emplean el microservicio de *Telemetry* para registrar sus eventos en una base de datos centralizada (incluyendo tipo de log, mensaje, nivel de severidad —*Information*, *Warning*, *Error*—, *stack trace* e identificador del módulo), el *Agent Launcher* no puede depender de ningún otro microservicio, ya que su ejecución precede a la de todos ellos.

Por este motivo, se optó por un sistema de registro autónomo, basado en el uso del sistema de ficheros local. Sin embargo, tras realizar distintas pruebas de rendimiento en entornos de testing, se identificó un problema relacionado con el acceso concurrente a un mismo archivo de log: múltiples escrituras simultáneas provocaban bloqueos en el sistema de archivos, lo que derivaba en excepciones no controladas.

Antes de abordar la solución implementada, es conveniente explicar el funcionamiento del sistema de logging en `ASP.NET`. Este se basa en dos componentes principales:

```
public void CreateRouteAndCluster(
    RouteSpecificationDTO routeSpecificationDTO,
    string? authorizationPolicy,
    List<RouteConfig> currentRoutes,
    List<ClusterConfig> currentClusters)
{
    string normalizedRequestsBasePath = RouteUtilities
        .GetYarpNormalizedPath(routeSpecificationDTO.RequestsBasePath);

    string routeId = IdentifiersCreator.CreateRouteIdentifier(normalizedRequestsBasePath);
    string clusterId = IdentifiersCreator.CreateClusterIdentifier(normalizedRequestsBasePath);
    string destinationId = IdentifiersCreator.CreateDestinationIdentifier(
        normalizedRequestsBasePath,
        routeSpecificationDTO.Address,
        routeSpecificationDTO.ProductVersions.Select(pvd => pvd.Id));

    RouteConfig? route = currentRoutes.Find(rc => rc.RouteId == routeId);
    ClusterConfig? cluster = currentClusters.Find(cc => cc.ClusterId == clusterId);

    if (DestinationUtilities.DestinationExists(cluster, destinationId))
    {
        this.logger.LogWarning(
            Resources.RouteAlreadyExists,
            normalizedRequestsBasePath,
            routeSpecificationDTO.Address,
            string.Join(", ", routeSpecificationDTO.ProductVersions.Select(pvd => pvd.Id)));

        return;
    }

    if (!string.IsNullOrEmpty(authorizationPolicy)
        && authorizationPolicy != AuthenticationPolicyServicesConstants.IntegratedAuthenticationPolicyName)
    {
        // Anonymous route.
        if (AnonymousRoutes.IsAnonymousRoute(routeSpecificationDTO.RequestsBasePath))
        {
            this.CreateRouteAndCluster(
                routeSpecificationDTO,
                currentRoutes,
                currentClusters,
                specialRoutes: [routeSpecificationDTO.RequestsBasePath],
                authorizationPolicyToApplyToSpecialRoutes: null);

            return;
        }
```

Figura 4.25: Comprobación de rutas anónimas antes de la creación de *clusters* y rutas.

`ILoggerProvider` e `ILogger`, estructurados según el patrón fábrica [9]. El `ILoggerProvider` se encarga de crear y administrar instancias de `ILogger`, mientras que cada `ILogger` es responsable de emitir los registros correspondientes en los destinos configurados. Este enfoque permite generar múltiples loggers de forma aislada y concurrente, lo cual resulta coherente con el modelo asincrónico que caracteriza al entorno `ASP.NET`.

Para solucionar el problema de concurrencia en la escritura, se optó por implementar un sistema basado en colas concurrentes. En concreto, se registró una instancia de `ConcurrentQueue` como *singleton* [10] (figura 4.26) dentro del contenedor de dependencias. Cada `ILogger` generado por el `ILoggerProvider` añadía sus mensajes a esta cola, evitando así el acceso directo al sistema de archivos desde múltiples hilos, como se puede observar en la figura 4.27.

```
services.AddSingleton(new ConcurrentQueue<string>());
```

Figura 4.26: *ConcurrentQueue* registrada como singleton en el contenedor de dependencias.

Posteriormente, se introdujo un componente adicional: un `BackgroundService` personalizado encargado de procesar periódicamente la cola singleton y volcar su contenido al sistema de ficheros. Esta separación de responsabilidades permitió que únicamente una

[9] Patrón de diseño fábrica: https://refactoring.guru/design-patterns/factory-method
[10] Patrón de diseño singleton: https://refactoring.guru/design-patterns/singleton

```
/// <summary> Initializes a new instance of the <see cref="FileSystemLogger"/> class. </summary>
/// <param name="logs"> <see cref="ConcurrentQueue{String}"/>. </param>
/// <param name="timeProvider"> <see cref="TimeProvider"/>. </param>
public FileSystemLogger(ConcurrentQueue<string> logs, TimeProvider timeProvider)
{
    this.logs = logs;
    this.timeProvider = timeProvider;
}

/// <inheritdoc/>
public IDisposable? BeginScope<TState>(TState state)
    where TState : notnull
{
    return default;
}

/// <inheritdoc/>
public bool IsEnabled(LogLevel logLevel)
{
    return logLevel != LogLevel.None;
}

/// <inheritdoc/>
public void Log<TState>(
    LogLevel logLevel,
    EventId eventId,
    TState state,
    Exception? exception,
    Func<TState, Exception?, string> formatter)
{
    if (!this.IsEnabled(logLevel))
    {
        return;
    }

    StringBuilder stringBuilder = new StringBuilder();

    string timeStamp = this.timeProvider.GetUtcNow().ToString("HH:mm:ss.fff", CultureInfo.InvariantCulture);

    stringBuilder.AppendLine(CultureInfo.InvariantCulture, $"[{timeStamp}] [{logLevel}]:");
    stringBuilder.AppendLine(formatter.Invoke(state, exception));
    stringBuilder.AppendLine();

    this.logs.Enqueue(stringBuilder.ToString());
}
```

Figura 4.27: *ILogger* para el registro de mensajes en la cola concurrente.

clase —el servicio en segundo plano— accediera al archivo de log, mientras que los loggers seguían funcionando de forma concurrente sin causar bloqueos, su implementación se puede observar en la figura 4.28.

Gracias a esta arquitectura, se eliminó la causa raíz del problema de concurrencia, mejorando la robustez del sistema de logs del *Agent Launcher* sin sacrificar el rendimiento ni la trazabilidad de los eventos generados durante su ejecución.

4.4.4. Refactorizaciones

Durante el desarrollo e implementación de las funcionalidades relacionadas con el despliegue del frontend, se ha ido identificando funcionalidad común con la parte backend. Esto se debe, en parte, a la metodología de desarrollo basada en modelos adoptada por la empresa, la cual permite la generación automática de objetos de transferencia de datos (DTOs) en la capa de contratos.

```
/// <param name="outputFilesProvider"> <see cref="OutputFilesProvider"/>. </param>
public LoggerHostedService(
    ConcurrentQueue<string> logs,
    IFileSystem fileSystem,
    OutputFilesProvider outputFilesProvider)
{
    this.logs = logs;
    this.fileSystem = fileSystem;
    this.outputFilesProvider = outputFilesProvider;
}

/// <inheritdoc/>
protected override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    string logsFilePath = this.outputFilesProvider.GetOutputFile();

    while (!cancellationToken.IsCancellationRequested)
    {
        if (this.logs.TryDequeue(out string? log))
        {
            await this.LogMessagesAsync(logsFilePath, log, cancellationToken).ConfigureAwait(false);
        }
        else
        {
            await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken).ConfigureAwait(false);
        }
    }
}

private async Task LogMessagesAsync(string logsFilePath, string log, CancellationToken cancellationToken)
{
    try
    {
        await this.fileSystem.File
            .AppendAllTextAsync(logsFilePath, log, cancellationToken)
            .ConfigureAwait(false);
    }
    catch (Exception ex)
    {
        await Console.Error
            .WriteLineAsync(
                string.Format(CultureInfo.InvariantCulture, Resources.UnexpectedLoggingError, ex.Message))
            .ConfigureAwait(false);
    }
}
```

Figura 4.28: *HostedService* para vaciar la cola concurrente y escribir en sistema de ficheros.

El patrón de diseño más empleado ha sido el de método plantilla [11], o *Template Method*. Este patrón consiste en definir el esqueleto de un algoritmo en una superclase, delegando en las subclases la implementación específica de ciertos pasos del mismo.

En el caso de ASP.NET, este patrón se implementa mediante clases abstractas. La clase base, declarada como *abstract*, define una serie de métodos que también son abstractos, obligando así a sus clases derivadas a proporcionar una implementación concreta de los mismos. Posteriormente, las clases específicas se registran en el contenedor de dependencias, lo cual permite su inyección en aquellas clases que las necesiten.

Uno de los *refactors* [15] más interesante se llevó a cabo durante la implementación del proceso de subida de ejecutables del frontend a la base de datos de Deployment. En ese momento ya existía una funcionalidad equivalente para los ejecutables del backend, por lo que se partía de una base reutilizable. Se identificaron y aislaron en una clase base aquellas funcionalidades comunes no dependientes de frontend o backend. Se observó que estas funcionalidades compartidas estaban relacionadas con conceptos como productos y líneas de producto.

[11] Patrón de diseño plantilla: https://refactoring.guru/design-patterns/template-method

Gracias a esta separación, se constató que todos los campos del DTO utilizado para la subida de ejecutables del backend podían ser extraídos en un nuevo DTO independiente, que también podía ser utilizado por la funcionalidad del frontend. Este nuevo DTO fue denominado `ProductsInformationToUploadDTO` y encapsulaba toda la información relevante relativa a productos.

Finalmente, faltaba permitir que las clases derivadas de la clase base pudieran utilizar su propio DTO. Para ello, se hizo uso de los tipos genéricos en ASP.NET, lo cual permite definir una familia de tipos que pueden ser utilizados por dichas clases. Dado que ambos DTOs implementaban la interfaz `IDTO`, fue posible restringir el tipo genérico a esta interfaz, aprovechando así todas las ventajas que ofrece el tipado fuerte en ASP.NET.

En la figura 4.29 se puede observar cómo se emplea la palabra reservada *where* para imponer esta restricción de tipo. Además, se aprecia que las dependencias registradas están relacionadas exclusivamente con la información de productos y líneas de producto, lo que demuestra que la separación de responsabilidades mencionada previamente se ha aplicado satisfactoriamente.

Por su parte, en la figura 4.30 se muestra cómo la clase específica hereda de la clase base *ExecutablesUploader*, sobreescribiendo así el tipo genérico de TDTO para utilizar el tipo concreto *FrontendExecutableToUploadDTO*.

```
/// <summary> <see cref="ExecutableBase"/> uploader. </summary>
/// <typeparam name="TDTO"> The type of DTO. </typeparam>
public abstract class ExecutablesUploader<TDTO>
    where TDTO : IDTO
{
    private readonly ProductLineVersionsCreator productLineVersionsCreator;
    private readonly ProductVersionDataManager productVersionDataManager;
    private readonly ProductVersionsCreator productVersionsCreator;
    private readonly ProductVersionValidatedDataManager productVersionValidatedDataManager;

    /// <summary> Initializes a new instance of the <see cref="ExecutablesUploader{TDTO}"/> class. </summary>
    /// <param name="productLineVersionsCreator"> <see cref="ProductLineVersionsCreator"/>. </param>
    /// <param name="productVersionDataManager"> <see cref="ProductVersionDataManager"/>. </param>
    /// <param name="productVersionsCreator"> <see cref="ProductVersionsCreator"/>. </param>
    /// <param name="productVersionValidatedDataManager">
    ///     <see cref="ProductVersionValidatedDataManager"/>.
    /// </param>
    protected ExecutablesUploader(
```

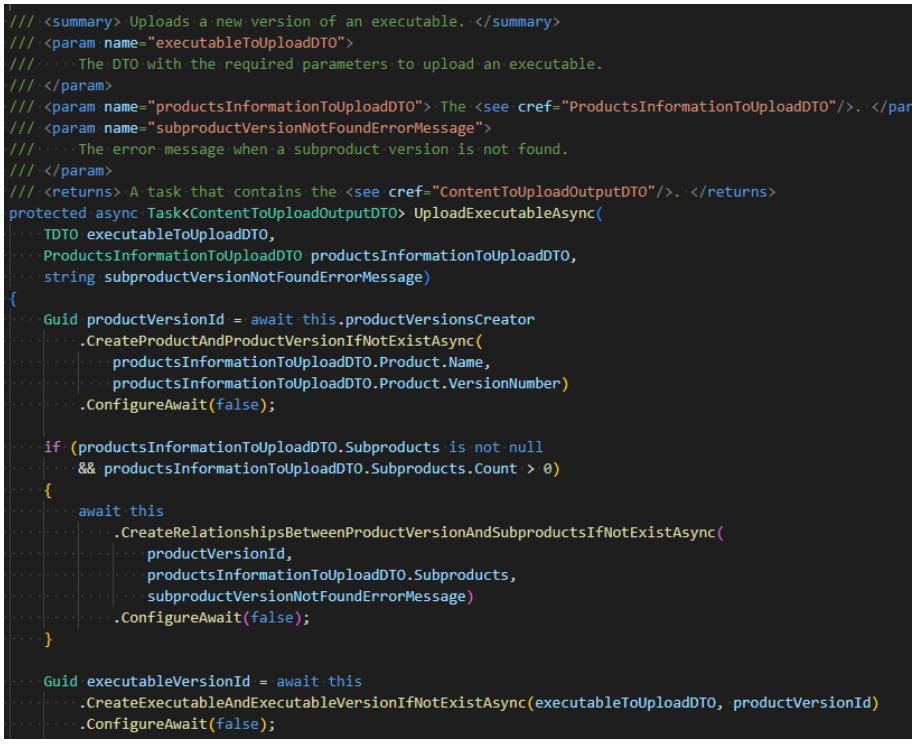
Figura 4.29: Clase base abstracta con el tipo genérico definido.

```
/// <summary> The uploader of <see cref="FrontendExecutable"/>. </summary>
public class FrontendExecutablesUploader : ExecutablesUploader<FrontendExecutableToUploadDTO>
{
    private readonly FrontendExecutableDataManager frontendExecutableDataManager;
    private readonly FrontendExecutableVersionDataManager frontendExecutableVersionDataManager;
    private readonly FrontendExecutableVersionConfigurationsCreator frontendExecutableVersionConfigurationsCreator;
```

Figura 4.30: Clase específica para ejecutables frontend que hereda de la clase base ExecutablesUploader.

Finalmente, en la figura 4.31 se presenta un método general en el que se ejecutan operaciones comunes para ambos tipos de ejecutables, y posteriormente se invoca el método abstracto *CreateExecutableAndExecutableVersionIfNotExistAsync*, cuya implementación específica es responsabilidad de las clases derivadas.

Este caso constituye un ejemplo claro de cómo la aplicación de patrones de diseño facilita la adopción de los principios SOLID [13]. En particular, se evidencia el *Single Responsibility Principle* a través de las dependencias inyectadas en cada clase, el *Open/Closed Principle* al permitir la extensión de la funcionalidad sin modificar la lógica base, y el *Liskov Substitution Principle* al sustituir el tipo genérico `IDTO` por tipos concretos sin afectar el comportamiento de la clase base.



```
/// <summary> Uploads a new version of an executable. </summary>
/// <param name="executableToUploadDTO">
///     The DTO with the required parameters to upload an executable.
/// </param>
/// <param name="productsInformationToUploadDTO"> The <see cref="ProductsInformationToUploadDTO"/>. </par
/// <param name="subproductVersionNotFoundErrorMessage">
///     The error message when a subproduct version is not found.
/// </param>
/// <returns> A task that contains the <see cref="ContentToUploadOutputDTO"/>. </returns>
protected async Task<ContentToUploadOutputDTO> UploadExecutableAsync(
    TDTO executableToUploadDTO,
    ProductsInformationToUploadDTO productsInformationToUploadDTO,
    string subproductVersionNotFoundErrorMessage)
{
    Guid productVersionId = await this.productVersionsCreator
        .CreateProductAndProductVersionIfNotExistAsync(
            productsInformationToUploadDTO.Product.Name,
            productsInformationToUploadDTO.Product.VersionNumber)
        .ConfigureAwait(false);

    if (productsInformationToUploadDTO.Subproducts is not null
        && productsInformationToUploadDTO.Subproducts.Count > 0)
    {
        await this
            .CreateRelationshipsBetweenProductVersionAndSubproductsIfNotExistAsync(
                productVersionId,
                productsInformationToUploadDTO.Subproducts,
                subproductVersionNotFoundErrorMessage)
            .ConfigureAwait(false);
    }

    Guid executableVersionId = await this
        .CreateExecutableAndExecutableVersionIfNotExistAsync(executableToUploadDTO, productVersionId)
        .ConfigureAwait(false);
```

Figura 4.31: Método definido en la clase base para la subida de ejecutables de cualquier tipo.

4.5 Pruebas

Las pruebas son un componente indispensable en el desarrollo de software. La complejidad, abstracción y casuística que puede presentar un producto software resulta, en muchos casos, inconcebible para un solo ser humano. Gracias a los tests, es posible verificar y asegurar que el comportamiento del sistema se mantiene constante a lo largo del tiempo.

El software no es, por lo general, un producto desarrollado de forma individual, por lo que no podemos garantizar que únicamente nuestras manos intervendrán en su evolución. En este contexto, las pruebas ayudan a definir los casos de uso que el sistema debe cumplir, sirviendo además como una forma de documentación que permite a otras personas comprender mejor su funcionamiento interno.

4.5.1. Criterios Generales para el Diseño de Pruebas

Para diseñar e implementar pruebas de manera efectiva, es fundamental seguir ciertos criterios generales [8]. Estos no dependen del tipo específico de prueba que se realice ni de los componentes involucrados, sino que se definen con el objetivo de garantizar consistencia y continuidad durante todo el ciclo de desarrollo del proyecto.

1. Coherencia en la nomenclatura de los métodos. A la hora de nombrar los métodos de prueba, se debe seguir una sintaxis clara y consistente: `metodoAProbar_contexto_resultadoEsperado`, por ejemplo, `AddItem_ItemAvailable_ItemAdded`.

2. Durante la implementación de una prueba, es recomendable estructurar el código en secciones claramente diferenciadas dentro del mismo test. Estas secciones son: *arrange*, donde se inicializa todo lo necesario para la ejecución de la prueba; *act*,

donde se invoca el método o función que se desea probar; y *assert*, donde se verifica que el resultado obtenido coincide con el resultado esperado.

3. Menos es más. La cantidad de pruebas no garantiza un producto libre de errores; lo verdaderamente importante es que los tests cubran de forma suficiente y precisa los casos relevantes. Al igual que las funcionalidades implementadas, las pruebas también deben mantenerse en el tiempo para asegurar su validez y utilidad.

4. Si un componente depende de otros que ya han sido debidamente testeados, estos deben ser sustituidos, en la medida de lo posible, por versiones simuladas (*fakes*), con el objetivo de reducir la complejidad cognitiva y evitar el sobretesting.

4.5.2. Pruebas unitarias

Para el desarrollo de las pruebas se ha empleado el paquete NuGet `Microsoft.VisualStudio.TestTools.UnitTesting`, comúnmente conocido como MSTest. Este proporciona una serie de utilidades que facilitan la creación de pruebas unitarias mediante el uso de atributos personalizados.

Los atributos son una característica propia de ASP.NET que permiten definir el comportamiento de una clase o método. En el caso de MSTest, se disponen de atributos como `[TestClass]`, que indica que una clase contiene métodos de prueba, y `[TestMethod]`, que marca los métodos individuales como pruebas.

En la figura 4.32 se muestra una prueba unitaria que utiliza un atributo específico denominado `[DataRow()]`. Este resulta especialmente útil para proporcionar múltiples conjuntos de datos de entrada a una misma implementación de prueba, lo cual permite reutilizar el código, facilitar su mantenimiento y mejorar su comprensión.

En el ejemplo ilustrado se evalúan tanto casos comunes como posibles situaciones límite (*edge cases*) que pueden surgir al utilizar el *middleware* de redirección, como el comportamiento ante direcciones vacías o nulas. Además, se comprueba si se ejecutan correctamente dos funcionalidades específicas: la reescritura de URL (*URL Rewrite*) y la redirección de URL (*URL Redirect*), a través del uso de dos valores booleanos.

Este tipo de pruebas ha sido el más numeroso, habiéndose implementado más de 100 pruebas unitarias distribuidas entre el middleware de redirección, el microservicio de Deployment, el Agent, el Agent Launcher, el Reverse Proxy, el Uploader y el Backend Loader. Estas pruebas han resultado especialmente útiles para la detección de errores tanto durante la extensión como en la implementación de nuevas funcionalidades, ya que se encontraban enfocadas en la clase o conjunto de clases responsables de proporcionar dicha funcionalidad. Además, gracias a que la mayoría de los componentes externos eran simulados mediante mocks, estas pruebas se ejecutaban con un alto rendimiento, no superando los 5 segundos en los casos más lentos, lo cual contribuyó a agilizar significativamente el desarrollo.

4.5.3. Pruebas de integración

Este tipo de pruebas busca simular un comportamiento real entre varios componentes, con el objetivo de garantizar su correcta interacción y replicar escenarios de uso representativos.

En este contexto, el concepto previamente mencionado de *fakes* resulta especialmente útil. Un *fake* es una versión simulada de un componente real —ya sea una clase, un método o incluso un proyecto completo— que se utiliza para evitar la sobrecarga de pruebas

```
/// <summary> Query string modifier middleware tests. </summary>
[TestClass]
public class QueryStringModifierMiddlewareTests
{
    /// <summary>
    ///     Method: <see cref = "QueryStringModifierMiddleware.InvokeAsync(HttpContext)" />, when invoked with
    ///     different values, then returns the proper response.
    /// </summary>
    /// <param name="originalReverseProxyServerUri"> The original ReverseProxy server URI. </param>
    /// <param name="expectedReverseProxyServerUri"> The expected ReverseProxy server URI. </param>
    /// <param name="urlPrefix"> The URL prefix of the server. </param>
    /// <param name="modifyQueryString"> Indicates whether the query string is to be modified. </param>
    /// <param name="redirection"> Indicates whether a redirection is expected. </param>
    /// <returns>A <see cref="Task"/> representing the result of the asynchronous operation.</returns>
    [TestMethod]
    [DataRow("http://fakeInvalidUrl", "http://fakeUrl", "Microservice/Fake/Prefix", false, true)]
    [DataRow("http://fakeInvalidUrl", "http://fakeUrl", "", false, true)]
    [DataRow("http://fakeInvalidUrl", "http://fakeUrl", "Microservice/Fake/Prefix", true, true)]
    [DataRow("http://fakeInvalidUrl", "http://fakeUrl", "", true, true)]
    [DataRow("http://fakeUrl", "http://fakeUrl", "", true, false)]
    [DataRow("http://fakeUrl", "http://fakeUrl", "Microservice/Fake/Prefix", true, false)]
    [DataRow("http://fakeUrl", "http://fakeUrl", "", false, true)]
    [DataRow("http://fakeUrl", "http://fakeUrl", "Microservice/Fake/Prefix", false, true)]
    [DataRow("", "", "", false, false)]
    [DataRow("", "", "", true, false)]
    [DataRow(null, null, null, false, false)]
    public async Task InvokeAsync_InvokedWithDifferentValues_ReturnsTheProperResponse(
        string originalReverseProxyServerUri,
        string expectedReverseProxyServerUri,
        string urlPrefix,
        bool modifyQueryString,
        bool redirection)
    {
        // Arrange.
        bool isInvoked = false;

        RequestDelegate next = new RequestDelegate(rd =>
        {
            isInvoked = true;

            return Task.CompletedTask;
        });
```

Figura 4.32: Test unitario utilizando atributo `[DataRow()]`.

en componentes que ya han sido verificados previamente. De este modo, se consigue aligerar las pruebas de integración y reducir su complejidad cognitiva. Estos *fakes* simulan comportamientos específicos, ya sean resultados positivos o negativos, sin necesidad de depender de las clases principales.

Un ejemplo de este tipo de simulación se muestra en la figura 4.33, donde se han creado aplicaciones web muy sencillas que replican funcionalidades esenciales, como el paso de parámetros o la definición de *endpoints* de salud, para su uso en pruebas de integración. Estas aplicaciones se compilan automáticamente al ejecutar los tests y, dado que son muy simples, el tiempo de compilación es mínimo. Otro ejemplo puede observarse en la figura 4.34, donde el mismo concepto se aplica a métodos dentro de una clase, reutilizando el caso de prueba que verifica el estado de salud de los ejecutables.

Finalmente, en la figura 4.35 se puede apreciar también el uso del atributo `[DataRow()]` en una prueba de integración. En este caso, se añade una breve descripción en la cabecera del método de prueba, con el propósito de contextualizar situaciones complejas que no pueden explicarse únicamente a través de los parámetros proporcionados al método de test.

Este tipo de pruebas es el que menos se ha desarrollado, debido a que su ejecución requiere un mayor tiempo al intentar utilizar componentes lo más reales posible. En total, se han implementado más de 20 pruebas de integración. Estas pruebas son más complejas que las pruebas unitarias, ya que requieren la replicación de escenarios reales, pero tienen la capacidad de detectar una mayor cantidad y variedad de errores.

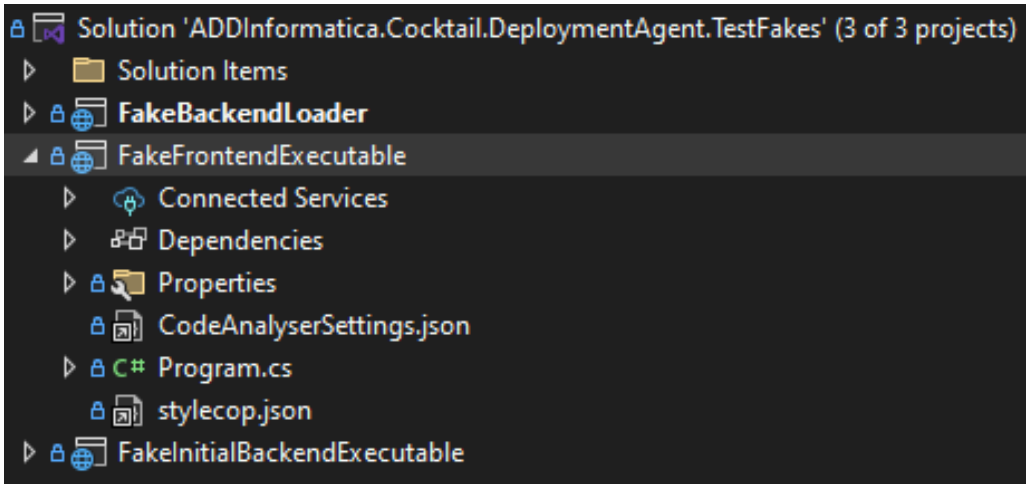


Figura 4.33: Proyectos *fakes* para ser utilizados por los tests de integración.

```
/// <summary> Health checker of fake executables. </summary>
9 references | [redacted] 2 authors, 2 changes
internal static class FakeExecutablesHealthChecker
{
    /// <summary> Checks the health of a fake executable. </summary>
    /// <param name="port"> Port where is listening to requests. </param>
    /// <param name="requestsBasePath"> Requests base path. </param>
    /// <param name="genericProxy"> <see cref="IGenericProxy"/>. </param>
    /// <param name="throwsException"> A flag to indicate if an exception should be thrown. </param>
    /// <returns> An empty <see cref="Task"/>. </returns>
    8 references | [redacted] 1 author, 1 change
    public static Task CheckAsync(
        int port,
        string requestsBasePath,
        IGenericProxy genericProxy,
        bool throwsException = false)
    {
        ActionRequest actionRequest = new ActionRequest(
            basePath: $"http://localhost:{port}",
            relativePath: $"{requestsBasePath}/Health");

        NonGenericAsyncFunctionAssertions nonGenericAsyncFunctionAssertions = genericProxy
            .Awaiting(gp => gp.EnsureResultIsEqualToExpectedResultAsync(
                actionToExecute: rm => rm.GetAsync(actionRequest),
                expectedResult: nameof(HealthStatus.Healthy)))
            .Should();

        if (throwsException)
        {
            return nonGenericAsyncFunctionAssertions.ThrowAsync<InvalidOperationException>();
        }

        return nonGenericAsyncFunctionAssertions.NotThrowAsync();
    }
```

Figura 4.34: Clases *fakes* para replicar comportamiento esperado.

En general, este tipo de pruebas resulta ser el más relevante en el contexto de este trabajo, ya que no solo ha permitido identificar una mayor cantidad de fallos, sino también errores significativamente más complejos que habrían sido muy difíciles de depurar mediante otros enfoques. El principal inconveniente radica en que tanto su diseño como su mantenimiento resultan considerablemente más costosos, y requieren de una documentación exhaustiva, dada la elevada complejidad cognitiva que implican para cualquier persona que no haya participado directamente en su desarrollo.

4.5.4. Pruebas generadas mediante DSL

Durante el desarrollo de este proyecto también se han empleado pruebas generadas automáticamente mediante herramientas basadas en lenguajes específicos del dominio

```
/// <summary>
///     The method <see cref="FrontendExecutablesUploader.UploadAsync(UploaderFrontendExecutable)"/>, with
///     different scenarios:
///     1. Frontend executable version with same version, product version and content hash already uploaded.
///     Nothing happens.
///     2. Frontend executable version with same version and content hash but different product version already
///     uploaded. Only product version relationships are created.
///     3. Frontend executable version with same product version and content hash but different version already
///     uploaded. Unique key failure because of same content hash.
///     4. Frontend executable version with same version and product version but different content hash. Unique
///     set of keys failure because of same version and frontend executable name.
/// </summary>
/// <param name="frontendExecutableVersionNumberToUpload">
///     The <see cref="UploaderFrontendExecutable.Version"/> to upload.
/// </param>
/// <param name="productVersionNumberToUpload">
///     The <see cref="UploaderFrontendExecutable.ProductVersionNumber"/> to upload.
/// </param>
/// <param name="frontendExecutablePathToUpload">
///     (Optional) The <see cref="UploaderFrontendExecutable.Path"/> to upload.
/// </param>
/// <param name="uploadShouldFail">(Optional) A value indicating if the uploading should fail.</param>
/// <returns>An empty <see cref="Task"/>.</returns>
[DataTestMethod]
[RetryTestMethod(
    RetryCount = 3,
    Justification = "Unable to configure resilient Entity Framework Core connections with SQL Server LocalDB.")]
[DataRow("2020.1.15.7", "2020.1.15.7", "WasmServer")]
[DataRow("2020.1.15.7", "2020.2.2.3", "WasmServer")]
[DataRow("2020.1.16.7", "2020.1.15.7", "WasmServer", true)]
[DataRow("2020.1.15.7", "2020.1.15.7", "WasmServerWithDifferentHash", true)]
public async Task UploadAsync_WithDifferentScenarios_ReturnsExpected(
    string frontendExecutableVersionNumberToUpload,
    string productVersionNumberToUpload,
    string frontendExecutablePathToUpload,
    bool uploadShouldFail = false)
{
```

Figura 4.35: Test de integración utilizando atributo `[DataRow()]` documentado.

(*Domain Specific Language*, DSL). Para ello, primero es necesario definir el test que se desea diseñar, modelar las distintas entidades involucradas, así como las entidades resultantes tras la ejecución del método a probar.

En la figura 4.36 se presenta un ejemplo de un test modelado. En este caso, se evalúa la llamada a la acción ad hoc para desplegar ejecutables. Las entidades marcadas con la letra `B`, que indica *before*, representan el estado inicial, es decir, aquellas existentes antes de la ejecución de la prueba. Por otro lado, las entidades con la letra `A`, de *after*, corresponden al estado resultante tras dicha ejecución.

Para definir aspectos específicos del test, como la descripción, el nombre o el resultado esperado, se utiliza la ventana de propiedades de Visual Studio 2022, como se muestra en la figura 4.37.

Además de definir las entidades participantes, es necesario especificar los campos que las componen. Para ello, se emplea un menú desplegable dentro del mismo apartado de propiedades, donde se pueden asignar los valores correspondientes a dichos campos. Esta funcionalidad se ilustra en la figura 4.38.

Este tipo de pruebas únicamente es aplicable a los microservicios desarrollados, por lo que se ha utilizado principalmente en componentes como Deployment, Agent y el resto de microservicios asociados. En la figura 4.39 se muestra un ejemplo del código generado automáticamente para el caso de uso correspondiente al despliegue de ejecutables desde el microservicio *Deployment*.

En contraste, para aplicaciones de consola como el Uploader o el BackendLoader, se ha optado por la creación manual de pruebas unitarias, dado que su simplicidad estructural no justifica el uso de herramientas automatizadas de generación de tests.

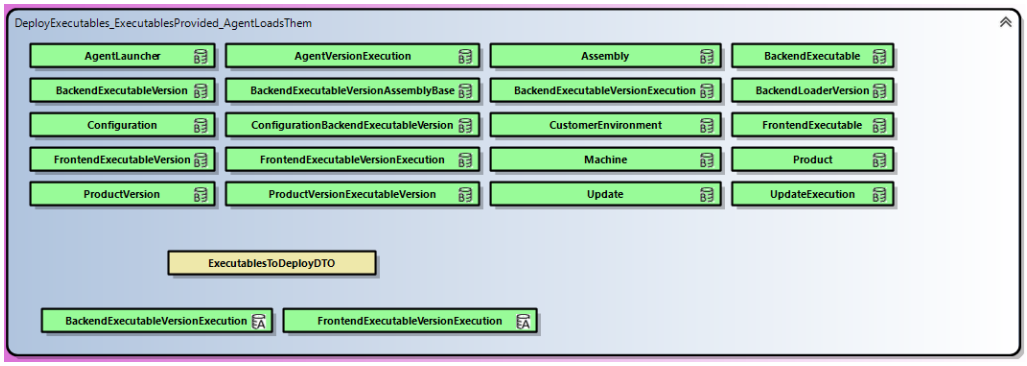


Figura 4.36: Test unitario modelado utilizando lenguaje DSL.

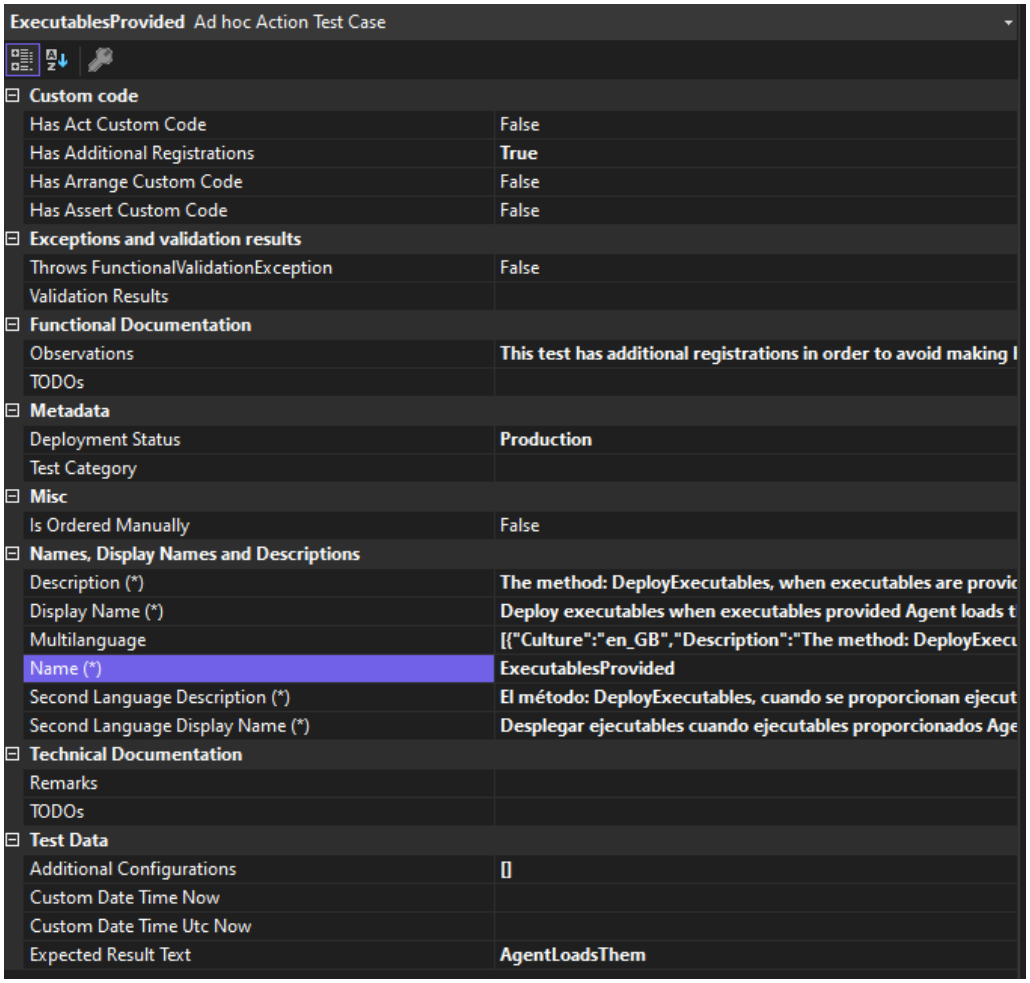


Figura 4.37: Propiedades especificadas para test modelado.

Este tipo de pruebas también se considera de carácter unitario. Se han diseñado más de 50 pruebas autogeneradas, contenidas en su mayoría en el microservicio de Deployment y el Agent. La principal ventaja de este enfoque es su mayor mantenibilidad a lo largo del tiempo, dado que las pruebas son específicas del dominio y abstraen en gran medida los aspectos relacionados con el lenguaje de programación utilizado.

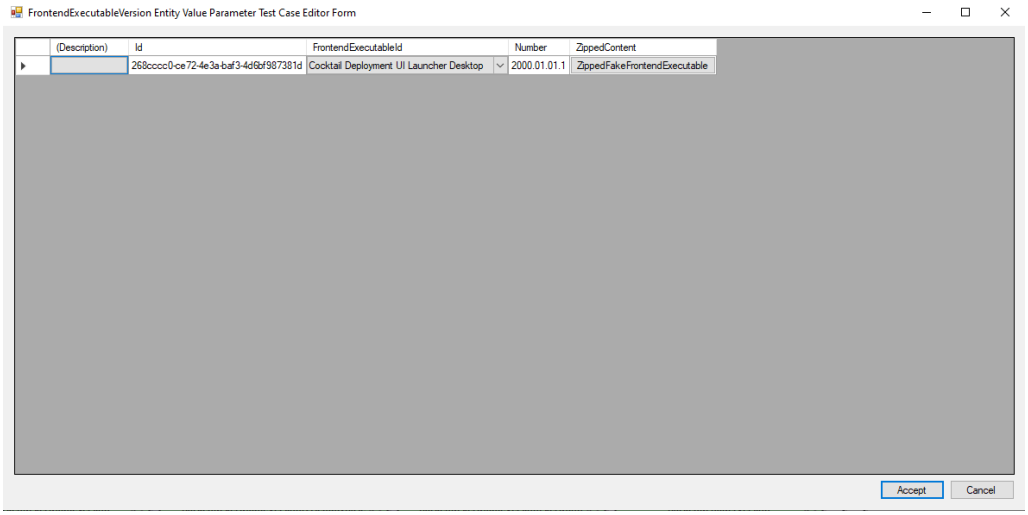


Figura 4.38: Campos de una entidad de base de datos modelada para tests.

```
/// <summary> The method: DeployExecutables, when executables are provided, the Agent loads them. </summary>
/// <returns> An empty <see cref="Task"/>. </returns>
[TestMethod]
[System.Diagnostics.CodeAnalysis.SuppressMessage(
    "Major Code Smell",
    "S125:Sections of code should not be commented out",
    Justification = "Commented code is a template method.")]
| 0 references | TFS2015AGENT, 3 days ago | 1 author, 1 change
public async Task DeployExecutables_ExecutablesProvided_AgentLoadsThem()
{
    // Arrange.
    Guid testId = Guid.NewGuid();

    void AdditionalRegistrations(IServiceCollection services)
    {
        // This code will not compile unless a method with this name is created in a partial class. Here is the
        // template for the method.
        /*
            /// <summary>
            ///     Custom code for registering additional registrations at the test 'Deploy executables when
            ///     executables provided Agent loads them'.
            /// </summary>
            /// <param name="services"> The collection of services to act on. </param>
            private static void ExecutablesProvidedAdditionalRegistrations(
                IServiceCollection services)
            {
            }
        */
        ExecutablesProvidedAdditionalRegistrations(services);
    }

    using ServiceProvider serviceProvider =
        CocktailDeploymentLogicTestsDependencyInjectionProvider
            .ArrangeServiceProvider(
                testId,
                testsPersistenceProviderType: TestsPersistenceProviderType.InMemorySQLite,
                additionalRegistrations: AdditionalRegistrations);

    using IServiceScope arrangeScope = serviceProvider.CreateScope();

    using DeploymentContext context =
        arrangeScope.ServiceProvider.GetRequiredEfCorePersistenceContext<DeploymentContext>();

    await context.Database
        .EnsureCreatedDeploymentDatabaseAsync(TestCaseCollectionId, TablesToCreate)
        .ConfigureAwait(false);

    List<Assembly> arrangeAssemblies =
```

Figura 4.39: Test generado a partir del modelado.

4.5.5. Pruebas de aceptación

Este tipo de pruebas no se ha realizado directamente, ya que como se ha observado en los casos de uso definidos CU01-CU03, no es una aplicación cuyo objetivo sea un usuario final normal y corriente, si no, la mayoría del *heavy lifting* sucede internamente.

Eso quiere decir que el microservicio abstrae en gran medida toda lígica interna dejando al usuario unas funcionalidades muy básicas.

Lo más parecido a las pruebas de aceptación reales, han sido los *sprint reviews* realizados. Donde al final de cada sprint se comprobaba mediante los *stakeholders* relevantes que las funcionalidades eran lo esperado por el usuario final.

4.5.6. Pruebas de regresión

Las pruebas de este tipo se utilizan para garantizar que las funcionalidades ya existentes continúan devolviendo los resultados esperados, incluso después de haber realizado cambios en el código o, por ejemplo, tras aplicar refactorizaciones. Por ello, es fundamental que estas pruebas se ejecuten de forma periódica.

En este caso, se aplicaban de manera automática tras cada cambio, como se explicará en el apartado siguiente. Además, por precaución, se ejecutaban semanalmente mediante *pipelines* especiales programadas para iniciarse cada domingo a las 03:00 a. m. Estas *pipelines* compilaban todo el repositorio y ejecutaban la totalidad de las pruebas disponibles.

4.5.7. Automatización de pruebas

Todas las pruebas mencionadas han sido automatizadas gracias al ciclo de vida DevOps implementado en la empresa. Para ello, se emplea una herramienta de consola personalizada denominada *Repository Files Generator* (RFG), cuyo propósito es generar automáticamente los archivos del repositorio.

En el contexto de las pruebas automatizadas, esta herramienta se encarga de identificar los cambios realizados en el repositorio y calcular las dependencias necesarias entre los distintos proyectos. A partir de esta información, se genera una *pipeline* de integración continua (CI), responsable de compilar todos los proyectos y soluciones, así como de publicarlos como artefactos para su posterior despliegue mediante entrega continua (CD).

Asimismo, la herramienta genera lo que se denomina una *pipeline* de *commit*, la cual también se construye en función de los cambios detectados en el repositorio, aunque en este caso aplicados a los proyectos o soluciones de pruebas [16]. De este modo, antes de cada integración de una *pull request*, es obligatorio que se superen todas las pruebas asociadas.

En la figura 4.40 se muestra un ejemplo de una de estas *pipelines* generadas. Es posible identificar la coletilla de “autogenerado” en la cabecera del fichero, lo que permite distinguirlas de las *pipelines* configuradas manualmente. Por su parte, la figura 4.41 ilustra la generación de una *pipeline* de *commit*.

4.6 Despliegue

En el despliegue se han empleado diversas máquinas con el objetivo de simular un entorno realista entre un cliente y un servidor. Tal como se indicó en el apartado de diseño, el *deployment* contempla dos roles principales; por lo tanto, para mantener dicha estructura, se han creado dos máquinas por cada uno de los entornos (Integración, Desarrollo y Pruebas). Estas máquinas han sido configuradas de forma idéntica y dotadas con los mismos recursos, con el fin de garantizar la validez de las pruebas realizadas e independizar los resultados del hardware subyacente.

```yaml
stages:
- ${{ if eq(parameters.AvoidEnsureIntegrationPipelineYamlGeneration, false) }}:
  - stage: EnsureIntegrationPipelineYamlGeneration
    displayName: Ensure Integration pipeline YAML generation
    jobs:
    - template: \Agility\ADOS\Templates\RepositoryFilesGenerator\EnsurePipelineYamlGenerationTemplate.yaml
      parameters:
        PipelineType: Integration
        PoolName: ${{ parameters.MultiAgentPool }}

- stage: PublishBuildArtifactsStep1
  displayName: Publish build artifacts (step 1)
  ${{ if eq(parameters.AvoidEnsureIntegrationPipelineYamlGeneration, false) }}:
    condition: succeeded('EnsureIntegrationPipelineYamlGeneration')
  jobs:
  - template: \Agility\ADOS\Templates\Integration\IntegrationBuildTemplate.yaml
    parameters:
      BuildAgentPool: ${{ parameters.MultiAgentPool }}
      BuildConfiguration: ${{ parameters.BuildConfiguration }}
      BuildExtraArgs: ${{ parameters.MultiAgentBuildExtraArgs }}
      BuildTool: dotnet
      CheckLastGitPublicationCommit: false
      JobName: AgilityAdosHooksModelsExtractorBackendProjectsWithoutExternalDependenciesIntegration
      ProjectOrSolution: Agility\ADOS\Hooks\ModelsExtractor\ADDInformatica.Agility.ADOS.Hooks.ModelsExtractor.Backend.Pr
      SolutionFolder: Agility\ADOS\Hooks\ModelsExtractor
      VersionNumber: $(VersionNumber)
```

Figura 4.40: Pipeline de integración generada mediante RFG.

```yaml
stages:
- ${{ if eq(parameters.AvoidEnsureCommitPipelineYamlGeneration, false) }}:
  - stage: EnsureCommitPipelineYamlGeneration
    displayName: Ensure Commit pipeline YAML generation
    jobs:
    - template: \Agility\ADOS\Templates\RepositoryFilesGenerator\EnsurePipelineYamlGenerationTemplate.yaml
      parameters:
        PipelineType: Commit
        PoolName: ${{ parameters.MultiAgentPool }}

- stage: ExecuteCommitBuilds
  displayName: Execute commit builds
  ${{ if eq(parameters.AvoidEnsureCommitPipelineYamlGeneration, false) }}:
    condition: succeeded('EnsureCommitPipelineYamlGeneration')
  jobs:
  - template: \Agility\ADOS\Templates\Commit\CommitBuildTemplate.yaml
    parameters:
      BuildAgentPool: ${{ parameters.MultiAgentPool }}
      BuildConfiguration: ${{ parameters.BuildConfiguration }}
      BuildExtraArgs: ${{ parameters.MultiAgentBuildExtraArgs }}
      BuildTool: dotnet
      JobName: AgilityAdosHooksModelsExtractorFullWithoutUiExecutables
      ProjectOrSolution: Agility\ADOS\Hooks\ModelsExtractor\Models\Tests\Models.Tests.csproj
      RunTests: ${{ parameters.RunTests }}

  - template: \Agility\ADOS\Templates\Commit\CommitBuildTemplate.yaml
    parameters:
      BuildAgentPool: ${{ parameters.MultiAgentPool }}
      BuildConfiguration: ${{ parameters.BuildConfiguration }}
      BuildExtraArgs: ${{ parameters.MultiAgentBuildExtraArgs }}
      BuildTool: msbuild
      JobName: AgilityDevOpsServerFullWithoutUiExecutables
      ProjectOrSolution: .repositoryFilesGenerator\Commit\SolutionFilters\Agility\DevOpsServer\ADDInformatica.Agility.De
      RunTests: ${{ parameters.RunTests }}
```

Figura 4.41: Pipeline de *commit* generada mediante RFG.

La relevancia de los distintos entornos mencionados radica en el momento del proceso que representan. Por ejemplo, el entorno de integración es obligatorio antes de incorporar cualquier cambio en la rama `master` que pueda afectar al sistema de despliegue. El entorno de desarrollo se emplea de forma pasiva tras la integración de cada *pull request*, sin generar una carga adicional de integración para los miembros del equipo. Por último, el entorno de pruebas está destinado a la validación manual y es el único al que se permite acceso directo a la máquina correspondiente, constituyendo la última instancia de verificación.

Para la nomenclatura de las máquinas, se ha utilizado el prefijo `DEM` para aquellas que cumplen el rol de Deployment Master, y `DEC` para las correspondientes al rol de Deploy-

ment Client. En la figura 4.42 se pueden ver todas las máquinas que han sido creadas para los distintos entornos y productos.

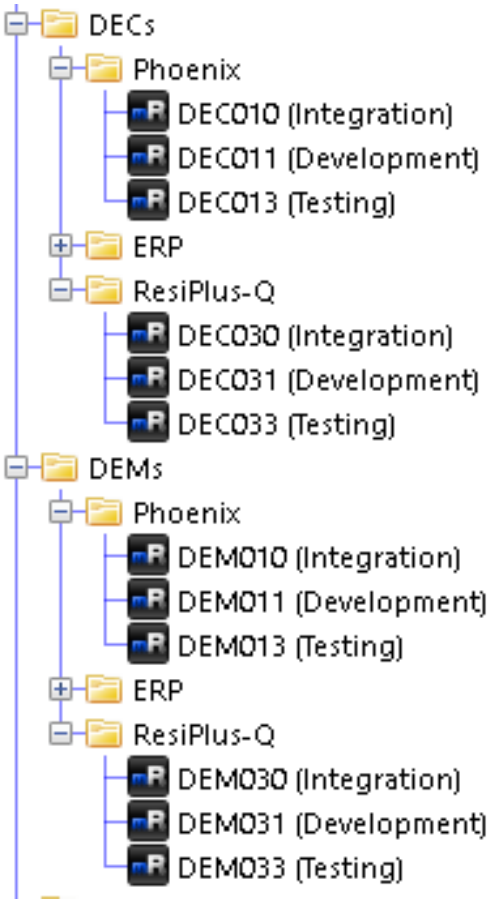


Figura 4.42: Máquinas creadas para el despliegue en los distintos entornos.

Con el objetivo de facilitar el despliegue en los distintos entornos, se han utilizado las *pipelines* de Azure DevOps Services (ADOS), siguiendo la misma estructura definida para el resto del ciclo CI/CD. En la figura 4.43 se muestra una ejecución de una de estas *pipelines*, en la cual se puede observar la instalación de los microservicios iniciales, la subida de los distintos ejecutables al Deployment Master, junto con la subida de la actualización correspondiente, y finalmente, el despliegue en el Deployment Client mediante el instalador del *Agent Launcher* en la máquina cliente.

Una vez finalizado todo el proceso de despliegue, en la figura 4.44 se puede comprobar cómo se han iniciado correctamente todos los procesos asociados a los ejecutables, tanto del frontend como del backend, así como otros componentes relevantes, como el `Agent.Listener`, correspondiente al servicio de Windows del Agent Launcher, o el propio Agent, ejecutado mediante el proceso de `DeploymentAgent.Services.exe`.

Como paso final, es necesario verificar que dichos procesos están operativos y reciben peticiones correctamente. En la figura 4.45 se accede a la dirección configurada del servidor WebAssembly de Deployment, donde se puede observar que la aplicación redirige correctamente al apartado de inicio de sesión, utilizando el *middleware* de redirección. Esto se evidencia en la URL, que incluye el argumento correspondiente al proxy inverso en este caso, la dirección del Agent Launcher.

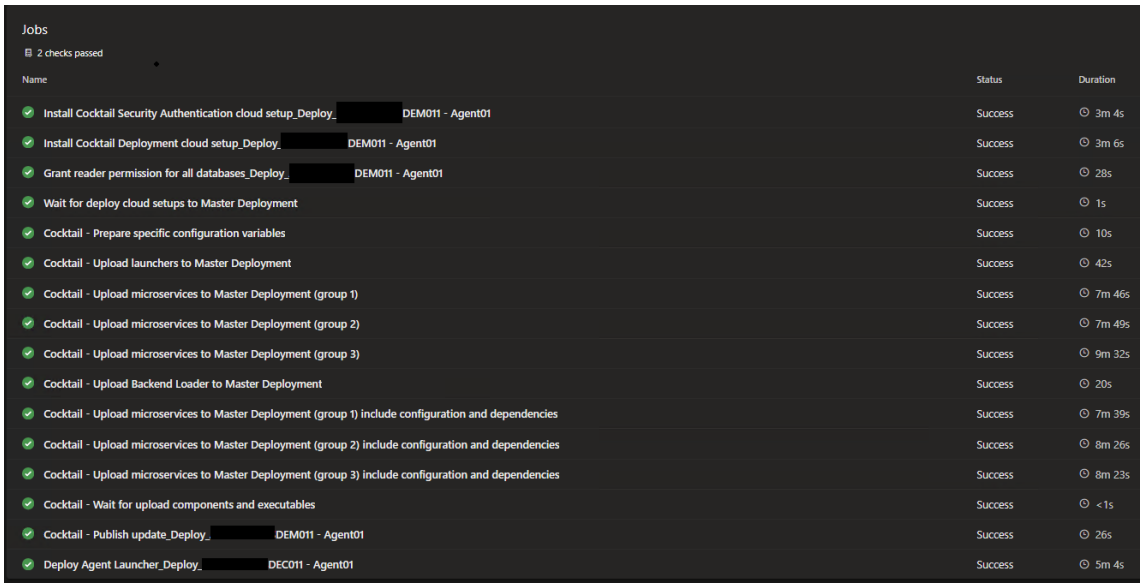


Figura 4.43: Ejecución de una de las pipelines para el despliegue de una actualización.

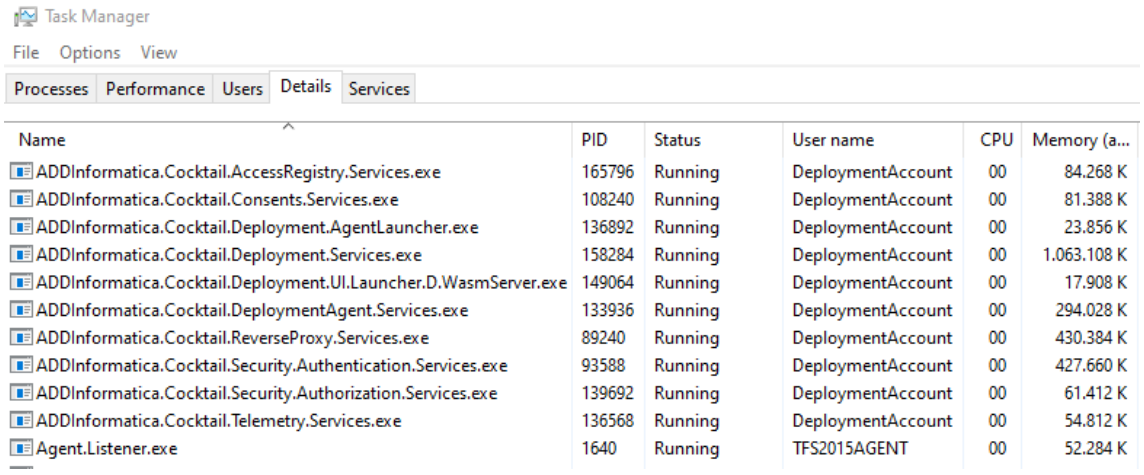


Figura 4.44: Procesos de los ejecutables desplegados.

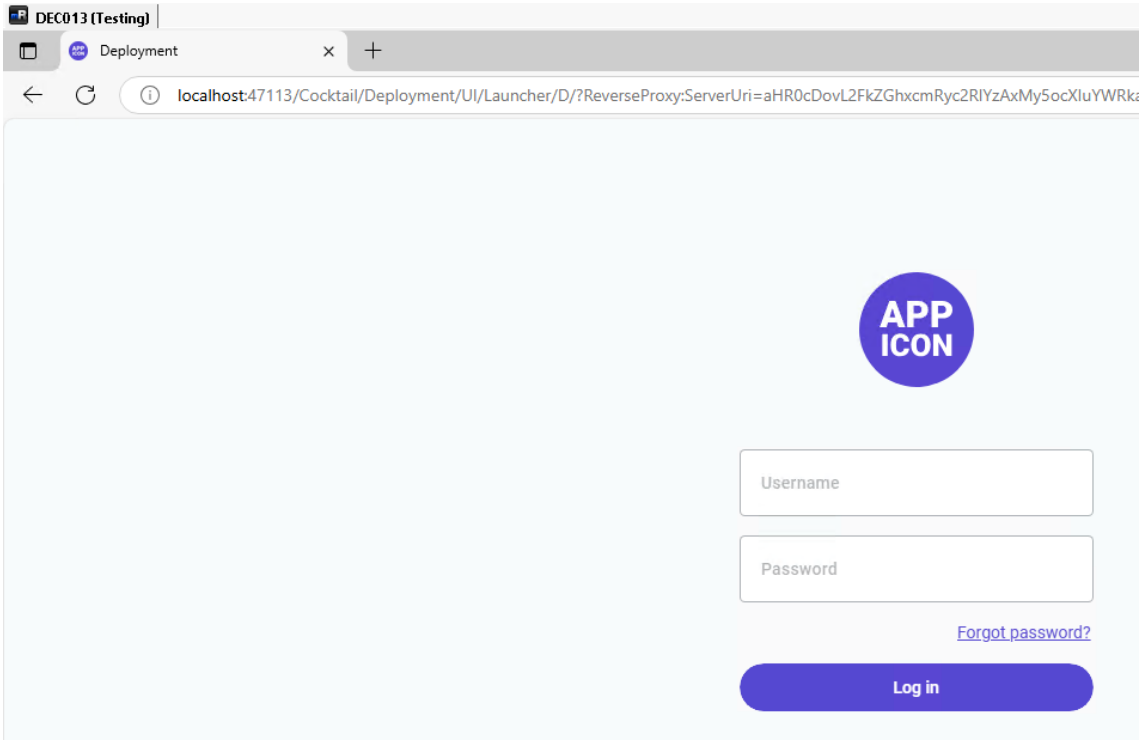


Figura 4.45: Servidor WebAssembly desplegado en la máquina de *testing*.

CAPÍTULO 5
Cronología del TFG

En esta memoria se han presentado las diferentes tareas desarrolladas para la elaboración del Trabajo de Fin de Grado de forma estructurada, haciendo especial énfasis en el apartado 4.4, donde se detallan las distintas fases del desarrollo del sistema. A continuación, se expone un resumen cronológico de las etapas más relevantes del proyecto, teniendo en cuenta que en la sección 4.1 se explica el uso de los sprints durante el proceso.

Estudio de tecnologías utilizadas por el equipo de desarrollo

Durante el primer sprint, se dedicó un esfuerzo inicial a familiarizarse con el conjunto de tecnologías y metodologías adoptadas por el equipo, como `ASP.NET` con C#, así como la arquitectura basada en microservicios. Se profundizó especialmente en el enfoque de desarrollo dirigido por modelos (MDD), ya que representa un pilar fundamental en la forma en que se construyen y despliegan los servicios. Este proceso sentó las bases para comprender tanto la estructura del ecosistema como la lógica subyacente a la solución planteada. Parte de esta fase se corresponde con los contenidos presentados en el apartado 3.

Diseño del sistema y definición de requisitos funcionales

Paralelamente al estudio tecnológico y dentro del mismo primer sprint, se celebraron diversas reuniones con los *stakeholders* implicados con el objetivo de capturar los requisitos del sistema desde el punto de vista del usuario final. En particular, se analizaron las necesidades del administrador del sistema, su flujo de trabajo habitual y los pasos que este debe seguir para instalar el producto en una máquina sin configuración previa. También se evaluaron y seleccionaron las tecnologías necesarias para el despliegue, como el uso de WebAssembly para encapsular y distribuir la interfaz gráfica, así como los distintos componentes funcionales como Agent, Agent Launcher o Uploader. Esta fase se recoge principalmente en los apartados 3.1, 3.2, y 4.3.

Implementación de la solución

Los sprints segundo y tercero se dedicaron de forma casi exclusiva al desarrollo de la solución propuesta. Durante esta etapa se construyeron los distintos servidores WebAssembly que contienen la interfaz de usuario para cada microservicio, y se implementaron los componentes funcionales adaptándolos a los requisitos definidos previamente. Además, se diseñaron e implementaron pruebas unitarias para validar la lógica interna de

cada componente, y pruebas de integración que garantizaran el correcto funcionamiento conjunto del sistema. Esta fase abarca los apartados 4.4 y 4.5.

Refinamiento y pruebas

En el cuarto y último sprint se centraron los esfuerzos en consolidar las funcionalidades ya implementadas, resolver tareas pendientes del *backlog* y realizar pruebas exhaustivas sobre entornos de *testing*. Estas pruebas incluyeron tanto validaciones funcionales como la gestión de casos límite que podrían surgir en escenarios reales. Esta parte se encuentra reflejada en el apartado 4.6.

Documentación

A lo largo de todos los sprints se ha ido elaborando la presente memoria del TFG y la documentación técnica interna asociada a los sistemas desarrollados, facilitando así la mantenibilidad y escalabilidad del sistema a futuro.

En la figura 5.1 se muestran las tareas descritas anteriormente representadas sobre una línea temporal, organizada en los cuatro sprints definidos para el desarrollo del proyecto.

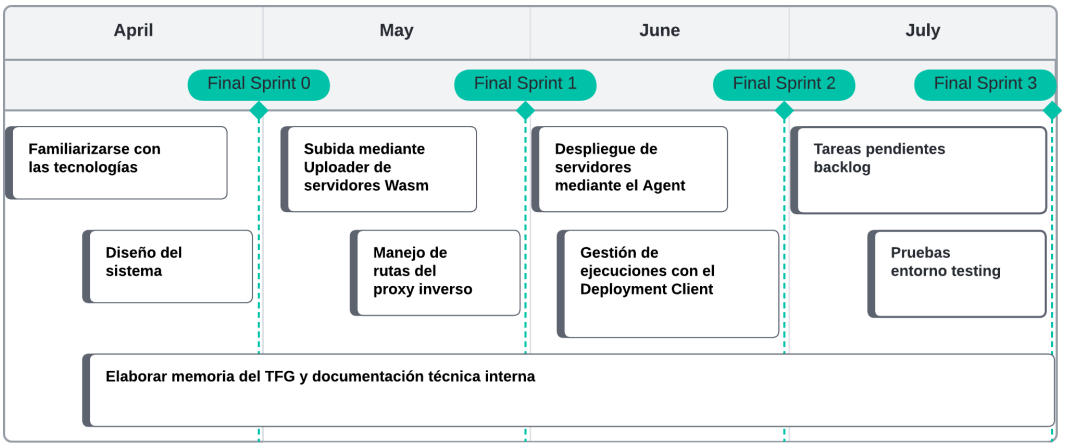


Figura 5.1: Línea de tiempo del desarrollo distribuido en cuatro sprints.

En total, el trabajo ha supuesto una dedicación aproximada de más de 300 horas, distribuidas de forma equilibrada entre el análisis inicial, el diseño del sistema, el desarrollo de la solución, las pruebas y la documentación. Gracias a la reutilización de estructuras comunes y al uso de herramientas automatizadas de despliegue, se ha conseguido optimizar tanto el tiempo de implementación como la validación del sistema.

CAPÍTULO 6
Conclusiones y trabajo futuro

Consideramos que los objetivos del trabajo definidos en el apartado 1.2 han sido cumplidos satisfactoriamente, desarrollando una solución funcional que no depende de ninguna de las tecnologías expuestas, como IIS, Docker o Kubernetes. Se ha conseguido desplegar de manera eficiente, y en un entorno de producción real, un producto compuesto por servidores WebAssembly que proporcionan la interfaz de usuario a los clientes de la aplicación, utilizando para ello una única máquina con un solo agente dentro de uno de los centros de salud a los que la empresa presta servicio. Se trata de un entorno altamente controlado y aún en una fase temprana, pero resulta de gran utilidad para identificar posibles errores que podrían afectar a otros centros en el futuro.

Tanto la aplicación como las pruebas de cada una de sus funcionalidades se ejecutan de forma automática en cada integración sobre la rama *master*, en distintas máquinas y entornos. Este proceso continuo garantiza que los cambios realizados se mantengan y verifiquen, incluso en ausencia de trabajo activo sobre este microservicio.

Varias asignaturas del Grado en Ingeniería del Software impartido por la UPV han contribuido significativamente, no solo a la realización del proyecto, sino también a la mejora de ciertos procesos y a una mejor comprensión de los fundamentos teóricos del trabajo realizado. Por ejemplo, gracias a la asignatura de Desarrollo de software dirigido por modelos (DSM), he podido adaptarme con mayor rapidez a la perspectiva MDD utilizada por la empresa, mediante la elaboración de metamodelos con herramientas de dominio específico (DSL).

Asimismo, la formación recibida en la asignatura de Diseño de Software (DDS) me ha permitido familiarizarme con los patrones de diseño y aplicarlos durante los procesos de refactorización y mantenimiento. En cuanto al desarrollo ágil empleado tanto en la empresa como en este proyecto, la asignatura de Proceso de Software (PSW) ha resultado fundamental. Finalmente, cabe destacar la relevancia de la asignatura de Concurrencia y sistemas distribuidos (CSD), ya que el sistema desplegado es de naturaleza distribuida, lo que requiere tener en cuenta una serie de características propias de este tipo de entornos.

Gracias a la realización de este Trabajo de Fin de Grado he podido aplicar, de forma práctica, los conocimientos adquiridos a lo largo de los cuatro años de formación universitaria. He tenido la oportunidad de trabajar en un entorno de producción real, donde los errores no son una opción, y donde los tests, las metodologías y las tecnologías se convierten en herramientas fundamentales. El software no está exento de errores, pero es posible evitar la mayoría de ellos si se trabaja con conocimiento y profesionalidad.

A nivel personal este proyecto me ha ayudado a ampliar mi perspectiva sobre el mundo profesional y, en definitiva, creo que me permitirá tomar decisiones más acertadas y ganar mayor confianza en mi perfil como profesional del sector.

Como trabajo futuro, aún existen numerosos frentes abiertos, ya que, como se ha mencionado previamente, el despliegue se ha llevado a cabo únicamente sobre una máquina cliente. Aunque se ha diseñado y planteado de forma teórica el despliegue en un entorno con múltiples máquinas y agentes, este escenario no ha sido probado en la práctica. Esto significa que muchos de los requisitos no funcionales relativos a la concurrencia y a los sistemas distribuidos no han sido validados.

Asimismo, con respecto al trabajo realizado, actualmente los usuarios solo podrían acceder a la interfaz de la aplicación a través de un navegador web. Sin embargo, esta solución no se ajusta completamente a las necesidades reales del cliente. Como se ha indicado anteriormente, se está utilizando Uno Platform como framework de desarrollo web, el cual permite generar aplicaciones multiplataforma. Cada una de estas plataformas debería poder integrarse y desplegarse mediante el sistema desarrollado. Por ejemplo, esto incluye aplicaciones móviles para Android e iOS, así como aplicaciones de escritorio destinadas a los equipos personales de los usuarios finales.

Cabe mencionar que WebAssembly plantea, como una promesa a futuro, la posibilidad de ejecutar aplicaciones de manera nativa con un rendimiento comparable en todas las plataformas, lo que potencialmente podría reemplazar las soluciones específicas para cada sistema operativo. Sin embargo, esta capacidad aún no se ha materializado plenamente en la práctica, por lo que su adopción generalizada sigue siendo un objetivo más que una realidad tangible.

Referencias

[1] Patricio Letelier, Proyecto de Ingeniería de Software. Metodología ágil utilizada en PIN. *ETSINF, Universitat Politècnica de València, 2024*.

[2] Nick Ramirez. *WiX 3.6: A Developer's Guide to Windows Installer XML*. Packt Publishing, 2012.

[3] Olga M. Londer Mike Volordarsky Brett Hill, Bernard Cheah, Steve Schofield, Carlos Aguiar Mares, Kurt Meyer, and Microsoft IIS Team. *Internet Information Services (IIS) 7.0 Resource Kit*. Microsoft Press, 2008.

[4] Sam Newman. *Building Microservices: Designing fine-grained systems*. O'Reilly Media, Inc, 2nd edition, 2021.

[5] Andrew Lock. *ASP.NET Core in Action, Third Edition*. Manning Publications, 2023.

[6] Andrew Stellman, Jennifer Greene. *Learning Agile*. O'Reilly Media, Inc, 2014.

[7] International Organization for Standardization. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. ISO/IEC 25010*. International Standard, 2023.

[8] Vladimir Khorikov, Roy Osherove. *The Art of Unit Testing, Third Edition*. Manning Publications, 2024.

[9] Nigel Poulton. *Docker Deep Dive - Fifth Edition*. Packt Publishing, 2025.

[10] Nigel Poulton, Pushkar Joglekar. *The kubernetes book*. Packt Publishing, 2025.

[11] Skye Hoefling. *Getting Started with the Uno Platform and WinUI 3: Hands-On Building of Cross-Platform Desktop, Mobile, and Web Applications That Can Run Anywhere*. Apress, 2022.

[12] Ahmad Mozaffar. *Mastering Blazor WebAssembly*. Packt Publishing, 2023.

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

[14] Chris Richardson. *Microservices Patterns*. Manning, 2018.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[16] Gene Kim, Jez Humble, Patrick Debois, John Willis, Nicole Forsgren. *The DevOps Handbook, 2nd Edition*. IT Revolution Press, 2021.

[17] *ISO/IEC 25010* [consultado en 04/25]: https://www.iso25000.com/index.php/normas-iso-25000/iso-25010.

[18] *.NET Core Guide* [consultado en 05/25]:
https://learn.microsoft.com/en-us/dotnet/core/introduction.

[19] *What is a Reverse Proxy? – Cloudflare* [consultado en 05/25]:
https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

[20] *Uno Platform Documentation* [consultado en 05/25]:
https://platform.uno/docs/articles/overview.html.

[21] *WebAssembly Core Specification* [consultado en 04/25]:
https://www.w3.org/TR/wasm-core-1/.

[22] *Postman – API Platform* [consultado en 06/25]:
https://www.postman.com/.

[23] *What is SSL – Cloudflare* [consultado en 05/25]:
https://www.cloudflare.com/es-es/learning/ssl/what-is-ssl/.

ANEXO A

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**		X		
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

ODS 3: Salud y bienestar

Este objetivo tiene como propósito garantizar una vida sana y promover el bienestar para todas las personas. Aunque desde una perspectiva técnica mi proyecto no está centrado directamente en la atención médica o la salud clínica, su contribución se enmarca en la infraestructura digital de una empresa que desarrolla software especializado en la gestión de residencias de mayores.

El sistema en el que se integra el microservicio desarrollado está destinado a optimizar procesos críticos en la gestión sociosanitaria: planificación de turnos, administración de medicación, control de historiales, entre otros. La fiabilidad y eficiencia del sistema de despliegue —en particular, su capacidad de realizar actualizaciones sin interrumpir el servicio (actualizaciones en caliente), garantizar la tolerancia a fallos y mantener el estado consistente de la infraestructura— tiene un impacto real en la calidad del servicio que reciben los usuarios finales de estas plataformas: profesionales de la salud, cuidadores y, en última instancia, los propios residentes.

De este modo, contribuir a que estas aplicaciones estén siempre disponibles, actualizadas y libres de errores técnicos no es solo una mejora técnica; es una forma de garantizar la continuidad operativa de sistemas que impactan directamente en la salud y el bienestar de personas vulnerables.

ODS 9: Industria, innovación e infraestructuras

Este objetivo promueve el desarrollo de infraestructuras fiables, sostenibles y resilientes, así como la innovación tecnológica. Aquí, la vinculación con mi trabajo es clara. He desarrollado un sistema que reemplaza tecnologías como Docker y Kubernetes, proponiendo una alternativa más ligera, controlada e integrada en entornos Windows, que permite desplegar microservicios sin depender de configuraciones externas complejas.

Esta propuesta no solo resuelve limitaciones técnicas específicas del entorno empresarial, sino que también representa una innovación arquitectónica: permite controlar y adaptar todo el ciclo de vida del despliegue desde un único microservicio, reduciendo la necesidad de herramientas externas.

Además, la solución promueve la modularidad, el aislamiento, la configuración dinámica y el control de versiones, lo cual eleva la madurez tecnológica de la infraestructura empresarial. Esto tiene un impacto directo sobre la capacidad de escalar, mantener y evolucionar productos digitales, objetivos clave del ODS 9.

ODS 12: Producción y consumo responsables

El ODS 12 hace referencia a la necesidad de utilizar los recursos de forma eficiente, reducir el desperdicio y minimizar el impacto ambiental de los sistemas de producción y consumo. Aunque habitualmente asociado a sectores como el industrial o el energético, en el contexto de las tecnologías de la información también tiene plena vigencia.

Uno de los puntos clave del sistema que he desarrollado es su capacidad para decidir de forma dinámica qué microservicios deben ser desplegados y cuáles no, así como para terminar procesos innecesarios o que hayan dejado de estar activos. Este comportamiento reduce significativamente el consumo de CPU, memoria y ancho de banda, especialmente en entornos donde se ejecutan múltiples servicios simultáneamente.

Además, al prescindir de soluciones pesadas como Kubernetes, se evita el uso de múltiples capas de orquestación que suelen mantener contenedores vivos, consumir recursos base y requerir máquinas virtuales adicionales. Esta eficiencia no solo mejora el rendimiento y reduce costos, sino que también implica un menor uso de energía, tanto en entornos locales como en infraestructuras en la nube. En resumen, el sistema fomenta un modelo tecnológico más responsable y sostenible.

ODS 13: Acción por el clima

Directamente relacionado con el anterior, este objetivo busca adoptar medidas urgentes para combatir el cambio climático y sus efectos. La tecnología, aunque intangible, tiene una huella ambiental. El consumo energético de centros de datos, servidores y entornos de ejecución es uno de los grandes desafíos del sector TIC en términos de sostenibilidad.

Al haber diseñado un sistema de despliegue que minimiza el uso de recursos y reduce los procesos innecesarios, el trabajo contribuye, aunque de forma modesta, a reducir el consumo eléctrico de los sistemas en los que se ejecuta. Esto es particularmente relevante en un entorno donde el número de instalaciones es elevado y donde pequeñas optimizaciones, multiplicadas por cientos de máquinas, tienen un impacto real.

Además, al permitir que actualizaciones y despliegues se realicen sin interrupciones, se evita la necesidad de reinicios completos de sistemas o procesos costosos en tiempo y energía. La eficiencia digital es también una forma de acción climática.

Conclusión

A menudo se piensa que el impacto en los Objetivos de Desarrollo Sostenible es exclusivo de proyectos sociales, ambientales o de gran visibilidad pública. Sin embargo, este TFG demuestra que incluso desde un rol técnico y especializado es posible contribuir a un desarrollo más justo, eficiente y sostenible.

Al mejorar la eficiencia de los procesos, al reducir el uso de recursos y al garantizar la estabilidad de aplicaciones críticas para la gestión sociosanitaria, el proyecto se alinea claramente con los ODS 3, 9, 12 y 13.