ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

MIGRACIÓN DE UNA APLICACIÓN MONOLÍTICA WEB DE STREAMING MUSICAL HACIA UNA ARQUITECTURA DE MICROSERVICIOS

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERÍA EN SISTEMAS INFORMÁTICOS Y DE COMPUTACIÓN

FRANCISCO JOSUE CEVALLOS CAIZA
francisco.cevallos01@epn.edu.ec

FRANKLIN ANDRÉS RUIZ GÓMEZ
franklin.ruiz@epn.edu.ec

DIRECTOR: Msc. Victor Vicente Velepucha Bonett
victor.velepucha@epn.edu.ec

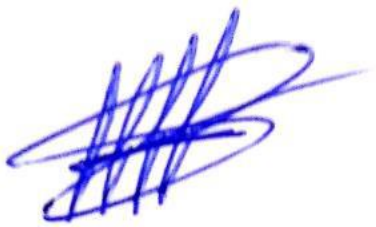
CODIRECTOR: PhD. Pamela Catherine Flores Naranjo
pamela.flores@epn.edu.ec

Quito, 03 de mayo del 2023

DECLARACIÓN

Nosotros, Francisco Josue Cevallos Caiza y Franklin Andrés Ruiz Gómez, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.



Francisco Josue Cevallos Caiza

Franklin Andrés Ruiz Gómez

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por los estudiantes Francisco Josue Cevallos Caiza y Franklin Andrés Ruiz Gómez, bajo nuestra supervisión.

Msc. Victor Velepucha

DIRECTOR DEL PROYECTO

PhD. Pamela Flores

CO-DIRECTOR DEL PROYECTO

AGRADECIMIENTO

A mi familia, que siempre ha sido mi mayor apoyo en todo momento. En especial a mis abuelos Roberto y Nancy; mis padres Francisco y Patricia; mis hermanas Ivy, Arelis, Vianny, Ariel, Jaelle, Franchesca, Fernanda y Santiago; mis tías Verónica y Jimena. Les agradezco por brindarme su amor incondicional y por enseñarme valores tan importantes como la perseverancia y el trabajo duro. Gracias por creer en mí y por darme la fuerza para seguir adelante en los momentos más difíciles.

A mis tutores de tesis Victor y Pamela, quienes me han brindado su orientación y asesoramiento en todo momento. Agradezco su paciencia y comprensión durante los momentos más difíciles del proceso de investigación y desarrollo. Gracias por compartir su experiencia y conocimientos conmigo y por guiarme hacia el éxito en este proyecto académico.

A mi mejor amigo y compañero de tesis Andrés, quien ha sido un apoyo invaluable. Aprecio su dedicación y compromiso para sacar adelante este proyecto; me siento agradecido por su amistad y colaboración.

A Pamela, por tus palabras de aliento, tu constante presencia, tu confianza en mí. Eres una persona excepcional que ha sabido ver más allá de mis limitaciones y siempre me has animado a dar lo mejor de mí mismo. Gracias por ser mi amiga y compañera.

A mis amigos, quienes han estado a mi lado en esta etapa universitaria y me han brindado su compañía y su amistad sincera.

A la Escuela Politécnica Nacional mi universidad, por darme la oportunidad de aprender de grandes profesores y expertos en mi campo de estudio. Gracias por brindarme los recursos y herramientas necesarias para alcanzar mis metas académicas y profesionales.

Francisco Cevallos

DEDICATORIA

A mis madres Patricia y Nancy las mujeres más importantes de mi vida. Ustedes han sido mi mayor inspiración y guía en todo momento. Gracias por darme el amor, la paciencia y la comprensión necesarias para alcanzar mis metas, sobre todo por ser ejemplo de inteligencia y valor.

Este proyecto está dedicado a ustedes, quienes me criaron y otorgaron toda su luz. Espero que este logro sea también un reflejo de su dedicación y amor hacia mí. Pai por siempre creer en mí, por ser las mejores madres mundo.

Con cariño y gratitud infinita.

Francisco Cevallos

AGRADECIMIENTO

En primer lugar, quiero expresar mi más sincera gratitud a mi familia, quienes han sido mi mayor apoyo en todo momento. A mi querida abuelita Alicia y a mi querido abuelito Mario, gracias por sus consejos, amor y apoyo incondicional. A mi mami Diana y a mi papá Fernando, gracias por su paciencia, comprensión y amor infinito. A mi hermano Ricardo, gracias por su motivación constante y por estar siempre en todo momento.

Agradezco enormemente a Johanna Arias, mi enamorada y compañera de carrera, por su incondicional ayuda y su amor. Su apoyo y entusiasmo han sido fundamentales en mi camino académico.

También quiero agradecer a mi mejor amigo Francisco, quien a la vez ha sido mi compañero de tesis. Gracias por su ayuda, apoyo y motivación constante.

Agradezco también a mis tutores, Pamela y Víctor, por su valiosa orientación, consejos y paciencia durante todo el proceso de desarrollo de este proyecto. Su experiencia y conocimientos han sido fundamentales para la consecución de los objetivos planteados y para mi crecimiento como profesional.

No puedo dejar de mencionar a mis amigos y amigas, quienes han sido una fuente constante de alegría y apoyo en mi vida. Gracias por su amistad, compañía y ánimo.

Por último, quiero expresar mi gratitud a mis profesores y a la Escuela Politécnica Nacional por brindarme una educación de calidad y por ayudarme a desarrollar mis habilidades y conocimientos. Sin su ayuda, este logro no hubiera sido posible.

Este trabajo está dedicado a ustedes, quienes han sido parte fundamental de mi vida y mi formación como persona y profesional.

¡Muchas gracias!

Franklin Ruiz

DEDICATORIA

Querida abuelita Alicia y querido papá Fernando,

Esta tesis no habría sido posible sin su amor, apoyo y motivación constante. Ustedes han sido mi roca en los momentos difíciles y mis mayores admiradores en los momentos de triunfo. A través de su ejemplo y enseñanzas, he aprendido la importancia del trabajo duro, la perseverancia y la dedicación.

A mi querido abuelito, gracias por ser un modelo de vida admirable y un pilar de nuestra familia. Su sabiduría y experiencia han sido invaluables para mí en mi formación como persona y profesional.

Les estoy muy agradecido por todo lo que han hecho por mí y espero que esta tesis dedicada a ustedes sea una forma de expresar lo mucho que los quiero y valoro. Espero seguir honrando su legado en mi carrera y en la vida en general, y poder hacerles sentir orgullosos de mí.

Con todo mi cariño y agradecimiento,

Franklin Ruiz

LISTA DE FIGURAS

LISTA DE TABLAS

RESUMEN

El objetivo de este trabajo de investigación es migrar una aplicación monolítica web de streaming musical hacia una arquitectura de microservicios, bajo el modelo MOMMIV (Modelo de Migración a Microservicios Versátil), el cuál guio este proceso mediante sus respectivas fases; y permitirá demostrar la factibilidad de dicho modelo para llevar a cabo una migración.

Para lograr esto, se utilizó el Principio de Ocultación de la Información en la descomposición del aplicativo monolítico en microservicios, lo que permitió separar las diferentes funcionalidades en componentes más pequeños, independientes y especializados. Además, se empleó el marco de trabajo Scrum para el desarrollo de los microservicios, lo que permitió una gestión ágil del proyecto y una mejor comunicación entre el equipo de desarrollo.

El control de versiones y el despliegue se realizaron utilizando la herramienta Azure DevOps, lo que permitió una gestión centralizada y eficiente de todo el proceso de migración. La ejecución de pruebas funcionales y no funcionales fue un aspecto importante del proceso de migración y se llevaron a cabo satisfactoriamente garantizando la calidad y el correcto funcionamiento de los microservicios.

Como resultado, esta tesis describe un proceso de migración exitoso de una aplicación monolítica a una arquitectura de microservicios, utilizando un enfoque sistemático y enfocado en la calidad del sistema resultante. Los resultados obtenidos indican que la migración a microservicios puede ser una solución efectiva para mejorar la escalabilidad, la disponibilidad y la mantenibilidad de sistemas web complejos como el de streaming musical.

Palabras clave: Migración, Aplicación, Web, Monolítica, Microservicios.

ABSTRACT

The objective of this research work is to migrate a web monolithic music streaming application to a microservices architecture, under the MOMMIV (Versatile Microservices Migration Model) model, which guided this process through its respective phases; and will demonstrate the feasibility of this model for carrying out a migration.

To achieve this, the Information Hiding Principle was used in the decomposition of the monolithic application into microservices, allowing the separation of different functionalities into smaller, independent and specialized components. In addition, the Scrum framework was employed for the development of the microservices, allowing for agile project management and better communication among the development team.

Version control and deployment were carried out using the Azure DevOps tool, allowing for centralized and efficient management of the entire migration process. Functional and non-functional testing was an important aspect of the migration process and was successfully carried out, ensuring the quality and proper functioning of the microservices.

As a result, this thesis describes a successful migration process from a monolithic application to a microservices architecture, using a systematic approach focused on the quality of the resulting system. The obtained results indicate that the migration to microservices can be an effective solution to improve the scalability, availability, and maintainability of complex web systems like music streaming."

Keywords: Migration, Application, Web, Monolithic, Microservices.

ÍNDICE DE CONTENIDO

I INTRODUCCIÓN

1.1 Descripción del Problema

Las organizaciones a lo largo del tiempo acumulan aplicaciones, que dada su antigüedad son conocidas como aplicaciones legadas; muchas de estas aplicaciones se construyen internamente por las empresas de desarrollo de software o se compran a proveedores externos y se encuentran desplegadas generalmente en Centros de Datos propios (Datacenters) de la organización conocidos como On-Premise [1]. Actualmente, entre los problemas que se evidencian en las empresas, están que estas aplicaciones legadas fueron construidas bajo una arquitectura monolítica y tienen obsolescencia, lo que aumenta la complejidad para su mantenimiento y evolución de la aplicación en la actualidad. El software monolítico tiene dificultades para ejecutarse sobre nueva infraestructura y sistemas operativos modernos, además existe escasez de profesionales calificados para realizar cambios en un aplicativo legado por lo que la calidad del código fuente nuevo desarrollado se puede ver comprometida [1].

Debido a estas dificultades, las empresas tienen la necesidad de migrar y modernizar sus aplicaciones monolíticas hacia plataformas modernas con el fin de ser competitivas en el mercado, y poder usar las bondades de las nuevas tecnologías, además de incorporar nuevas funcionalidades de negocio de forma rápida y sin errores. Una alternativa, es justamente migrar aplicaciones legadas a arquitecturas que permitan obtener los beneficios mencionados anteriormente, siendo una de las tendencias, la arquitectura de microservicios. Sin embargo, hay desafíos que deben considerarse en el proceso de migración, como son la complejidad de extraer la lógica de negocio embebida en la aplicación monolítica, las dificultades por descomponer la base de datos relacional de la aplicación (ya que cada microservicio puede tener su propia base de datos) y la complejidad en la creación de microservicios que realicen más de una funcionalidad de negocio, entre otras [2].

Este trabajo propone estudiar la migración de aplicaciones monolíticas a aplicaciones con tecnologías web, a través del desarrollo de una aplicación monolítica de streaming de música, construida bajo una arquitectura n-capas y que

usa componentes de software obsoletos, hacia una arquitectura de microservicios. Al no existir alguien que nos provea un sistema monolítico con acceso al código fuente, base de datos, repositorios, configuraciones de servidores y documentación relacionada, hemos escogido como caso de estudio la aplicación construida previamente por el Msc. Victor Velepucha, la cual nos permitirá realizar tal migración, con el propósito de modernizar la aplicación, que sea fácil de dar mantenimiento y que ofrezca facilidad para que la aplicación migrada pueda atender más usuarios bajo demanda [3].

1.2 Objetivos

1.2.1 Objetivo General

Migrar una aplicación web de streaming musical desarrollada bajo una arquitectura monolítica hacia una arquitectura de microservicios bajo el marco de trabajo SCRUM

1.2.2 Objetivos Específicos

- Identificar la lógica de negocio de la aplicación monolítica.
- Generar diagramas de la arquitectura de la aplicación monolítica.
- Descomponer en microservicios la lógica de negocio identificada de la aplicación a monolítica.
- Establecer la arquitectura de microservicios.
- Implementar la arquitectura de microservicios utilizando herramientas de desarrollo modernos.
- Evaluar la aplicación de arquitectura de microservicios mediante pruebas funcionales y no funcionales de software.

II MARCO TEÓRICO

2.1 Aplicación Web

Las aplicaciones Web son una categoría de software centrado en la internet, sin embargo, se define como una aplicación accesible mediante la Web por una red ya sea intranet o la misma internet mencionada anteriormente [4]. En general, se menciona como una aplicación Web a aquellos programas informáticos que son ejecutados sin necesidad de una instalación en el ordenador, tan solo con el uso de un navegador. Brinda contener múltiples ventajas para los usuarios como: acceder a la información de manera ágil y sencilla, recolectar y guardar información, etc[5].

Actualmente una aplicación Web es un conjunto de páginas Web estáticas y dinámicas. A continuación, hablaremos de ellas y de su funcionamiento.

- Estática

  Una página Web estática es aquélla que no cambia cuando un usuario la solicita. Cuando el servidor Web recibe una petición de una página estática, el servidor lee la solicitud, localiza la página y la envía al navegador solicitante [6].

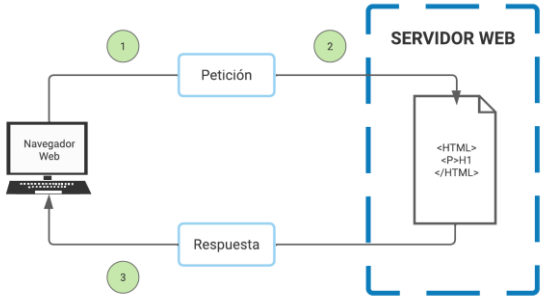


Figura 1 - Procesamiento de páginas web estáticas.

1. El navegador web solicita la página estática. 2. El servidor localiza la página. 3. El servidor Web envía la página al navegador solicitante.

- Dinámica

Las páginas Web dinámicas son aquellas donde el servidor modifica las páginas Web antes de enviarlas al navegador solicitante, he ahí su nombre.

Cuando el servidor Web recibe una petición para mostrar una página dinámica, transfiere la página a un software encargado de finalizar la página. Este software se denomina servidor de aplicaciones.

El servidor de aplicaciones a través de complementos, lee el código de la página, finaliza la página en función de las instrucciones del código y elimina el código de la página. El resultado es una página estática que el servidor de aplicaciones devuelve al servidor Web, que a su vez la envía al navegador solicitante [6].

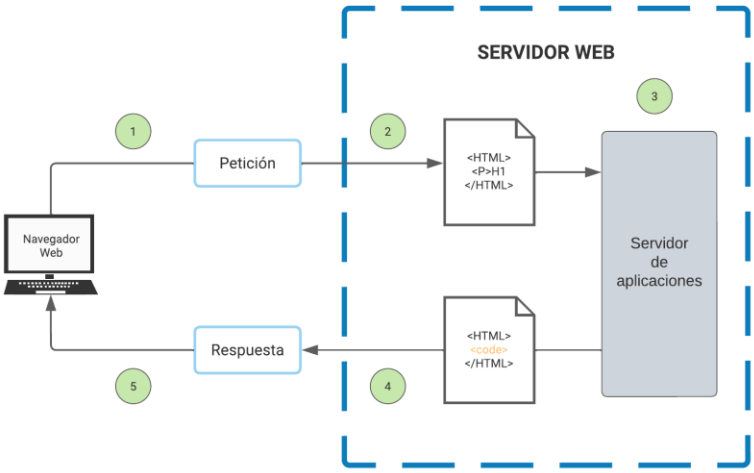


Figura 2 - Procesamiento de páginas web dinámicas.

1. El navegador web solicita la página dinámica. 2. El servidor web localiza la página y la envía al servidor de aplicaciones. 3. El servidor de aplicaciones busca instrucciones en la página y la finaliza. 4. El servidor de aplicaciones pasa la página terminada al servidor web. 5. El servidor web envía la página al navegador solicitante.

- Dinámicas con acceso a base de datos

Las páginas web dinámicas con acceso a base de datos son aquellas las cuales su servidor de aplicaciones le permite trabajar con recursos del lado del servidor, como las bases de datos.

Digamos que una página dinámica puede indicar al servidor de aplicaciones que extraiga datos de una base de datos y los inserte en el código HTML de la página.

El uso de una base de datos para almacenar contenido permite separar el diseño del sitio Web del contenido que se desea mostrar a los usuarios del sitio. En lugar de escribir archivos HTML individuales para cada página, sólo se necesita escribir una página o plantilla para los distintos tipos de información que se desea presentar. Posteriormente, podrá guardar el contenido en una base de datos y, seguidamente, hacer que el sitio Web recupere el contenido en respuesta a una solicitud del usuario. También puede actualizar la información en un único origen y, posteriormente, implantar ese cambio en todo el sitio Web sin necesidad de editar manualmente cada página [6].

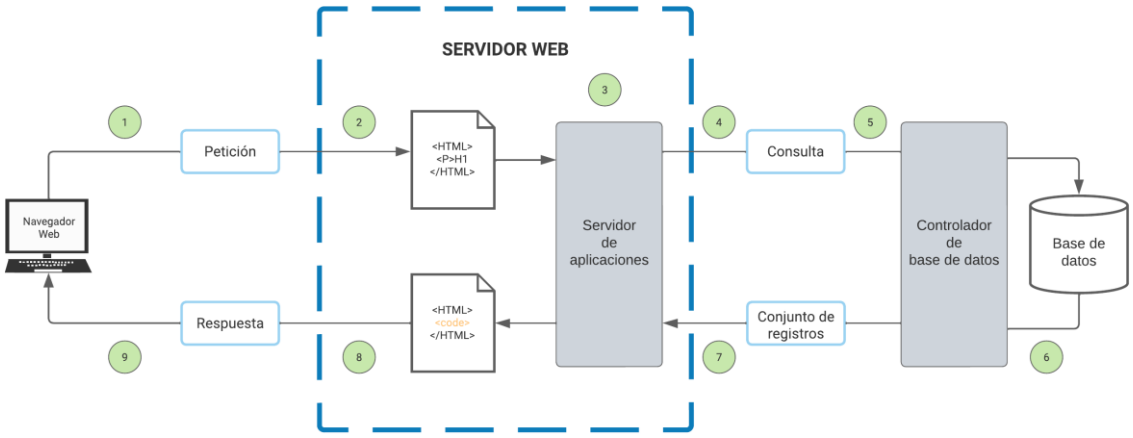


Figura 3 - Procesamiento de una página web dinámica con acceso a una base de datos.

1. El navegador web solicita la página dinámica. 2. El servidor web localiza la página y la envía al servidor de aplicaciones. 3. El servidor de aplicaciones busca instrucciones en la página. 4. El servidor de aplicaciones envía la consulta al controlador de la base de datos. 5. El controlador ejecuta la consulta en la base de datos. 6. El juego de registros se devuelve al controlador. 7. El controlador pasa el

juego de registros al servidor de aplicaciones. 8. El servidor de aplicaciones inserta los datos en una página y luego pasa la página al servidor web. 9. El servidor Web envía la página finalizada al navegador solicitante.

Una gran parte de las aplicaciones web existentes se encuadran dentro de las arquitecturas de software tipo Cliente Servidor, como se muestra en el procesamiento de los diferentes tipos de páginas web. Un cliente es quien realizará invocaciones a otros programas, el servidor es quien brindará la respuesta. En el caso de las páginas web dinámicas, codificadas en lenguaje de programación para el servidor como ASP.NET, PHP o JSP, la codificación HTML que es enviada al cliente se genera de forma dinámica en el aplicativo que está ubicado del lado del servidor en el instante que realiza la solicitud. Las páginas web se generan con la información recibida en la misma invocación o a través de consultas a la base de datos[7].

Aquí suelen distinguirse tres niveles (como en las arquitecturas cliente/servidor de tres niveles): el nivel superior que interacciona con el usuario (el cliente web, normalmente un navegador), el nivel inferior que proporciona los datos (la base de datos) y el nivel intermedio que procesa los datos (el servidor web)[8]. Estos conceptos se los detalla y trata con mayor en el punto 2.3 .

2.2 Arquitectura de software

En 1968 Dijkstra habla de una estructuración correcta de los sistemas de software, debido a las problemáticas que pueden existir en las sentencias de control usadas en lenguajes de programación de alto nivel, sin llevar un control del progreso del proceso del software [9]. Posteriormente Perry y Wolf, en 1992, proponen en su trabajo un modelo para la arquitectura de software, donde indican tres principales componentes: elementos, forma y razón. Los autores mencionan que los elementos pueden ser de procesamiento, datos o conexión, mientras que la forma se define de acuerdo con las propiedades de los elementos y las relaciones entre ellos, y la razón por otro lado, está dada en términos de restricciones del sistema derivados de los requerimientos del mismo [10].

Bass define a una arquitectura de software como "… *la estructura del sistema, la cual comprende elementos de software, las propiedades externamente visibles de esos elementos, y las relaciones entre ellos …"* [11]. Mientras que en el texto de David Fowler dice que la arquitectura de software es una representación de la solución de un programa o sistema computacional [12].

En el transcurso de los años ha sido difícil definir lo que es "Arquitectura de Software", es así que existe cientos de definiciones y ninguna es respaldada o validada por la totalidad de los arquitectos de software [13]. Sin embargo una definición reconocida es la de Clements, la cual menciona: La AS (Arquitectura de Software) es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones [14].

2.3 Arquitectura de software Cliente Servidor

La arquitectura de software Cliente Servidor se basa en una comunicación donde, el Cliente es un ordenador solicitando servicios y el Servidor está a la espera de recibir esas solicitudes y responderlas. Esta arquitectura es un modelo de aplicación distribuida donde las tareas se distribuyen entre los recursos (Servidores), y los demandantes de estas serían los Clientes[15], en donde los clientes necesitan conocer qué servidores están disponibles y cuáles son los servicios que proporcionan, pero normalmente no conocen la existencia de otros clientes y a su vez, los servidores no necesitan conocer la cantidad ni la identidad de los clientes, no obstante, los servidores pueden, a su vez, ser clientes de otros servidores[16].

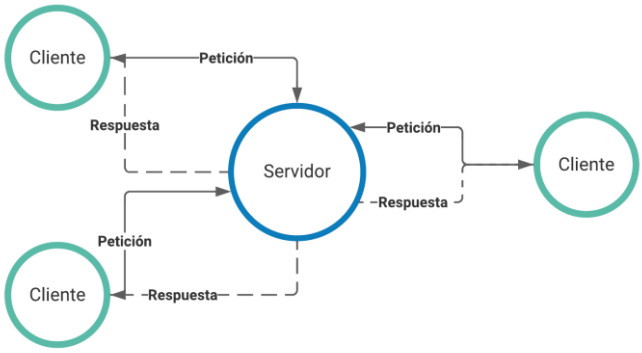


Figura 4 - Arquitectura Cliente Servidor.

2.3.1 Arquitectura Cliente Servidor de 2 Capas

La arquitectura cliente-servidor más simple se denomina arquitectura cliente-servidor de dos capas, en la que una aplicación se organiza como un servidor (o múltiples servidores idénticos) y participan un conjunto de clientes[16].

Las arquitecturas cliente-servidor de dos capas pueden ser de dos tipos:

a) Modelo de cliente ligero (thin-client). En un modelo de cliente ligero, todo el procesamiento de las aplicaciones y la gestión de los datos se lleva a cabo en el servidor. El cliente simplemente es responsable de la capa de presentación del software[8], [17].

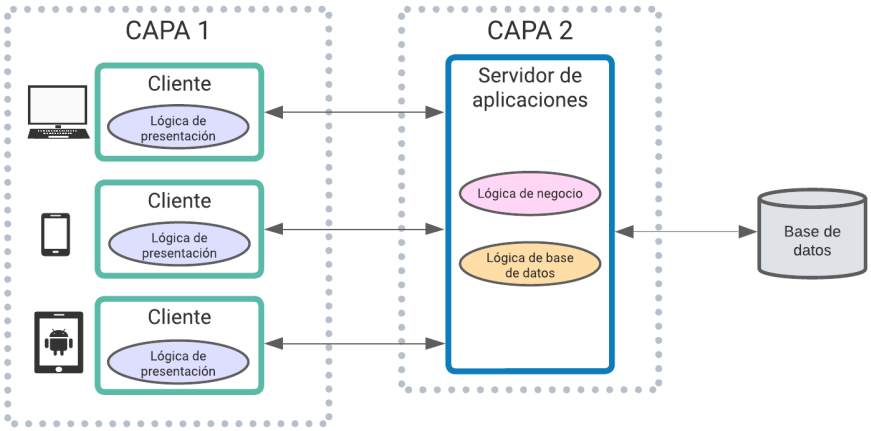


Figura 5 - Arquitectura Cliente Servidor (Modelo Thin-Client)

b) Modelo de cliente pesado (fat-client). En este modelo, el servidor únicamente gestiona los datos, es decir el servidor no requiere de una aplicación extra para proporcionar el servicio. El software del cliente mantiene la lógica de presentación o vista, de negocio y de acceso a datos. Estas aplicaciones son cerradas y supeditan la lógica de procesos cliente al gestor de base de datos que se está usando[8], [16].

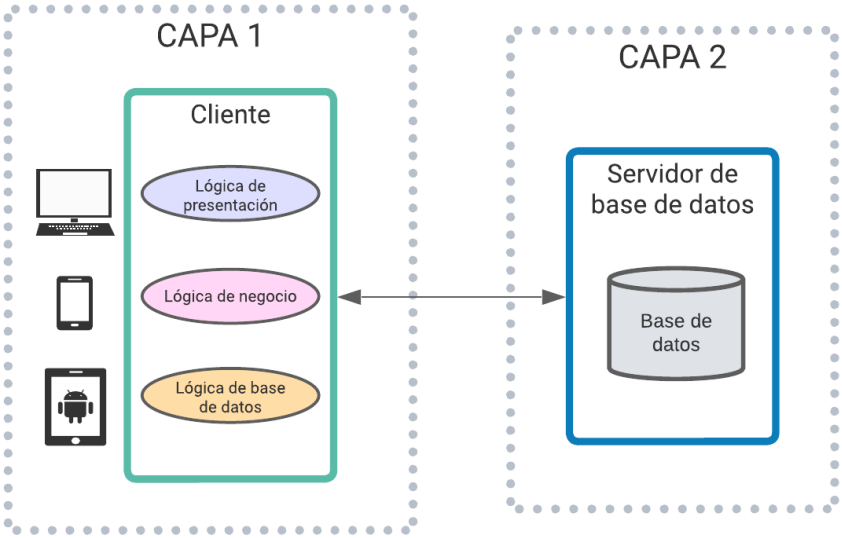


Figura 6 - Arquitectura Cliente Servidor (Modelo Fat-Client)

2.3.2 Arquitectura Cliente Servidor de 3 Capas

En la arquitectura de tres capas existe un nivel intermediario, eso significa que la arquitectura generalmente está compartida por un cliente que como hablamos más arriba es el que solicita los recursos, equipado con una interfaz de usuario (lógica de presentación) o mediante un navegador web. La capa del medio (lógica de negocio), cuya tarea es proporcionar los recursos solicitados pero que requiere de otro servidor para hacerlo. La última capa (lógica de datos), es el servidor de datos que proporciona al servidor de aplicaciones los datos necesarios para poder procesar y generar el servicio que solicito el cliente en un principio[8].

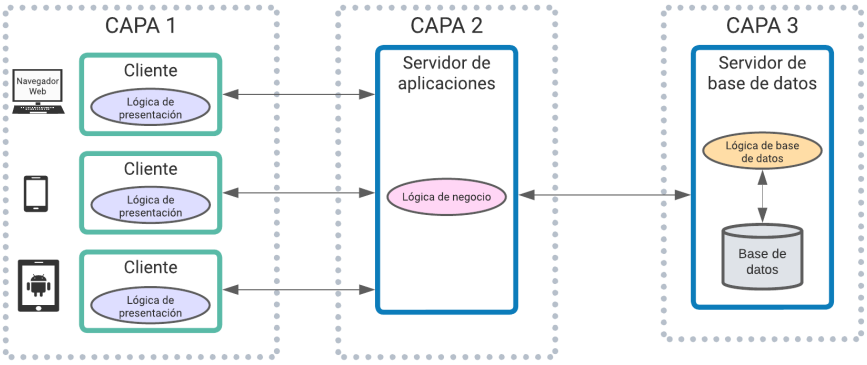


Figura 7 - Arquitectura Cliente Servidor 3 Capas.

2.3.3 Arquitectura Cliente Servidor de N Capas

Una arquitectura de N capas son aquellas que separan las responsabilidades y administrar dependencias específicas. Un primer ejemplo donde se evidencie esta premisa es la arquitectura de 3 capas, en la cual existe una capa de presentación, una capa intermedia, una capa de datos; por lo que esta arquitectura potencialmente de tipo N capas donde cada servidor brinda un servicio específico[16]. El objetivo principal en el empleo de la arquitectura de n-capas es proporcionar un soporte adecuado a los cambiantes requerimientos que se demandan en los procesos de negocios [18].

La separación por capas permite descomponer la aplicación, de manera que cada parte sea gestionada por uno o varios servidores especializados. El cliente hace una solicitud al servidor de la capa inmediata que está preparado para recibir y gestionar las solicitudes entrantes. El servidor de esta capa, a su vez, actúa como cliente y hace una solicitud a otro servidor. Finalmente, el servidor de la última capa devuelve un resultado que se retorna en sentido inverso a la solicitud y se trata en cada capa hasta llegar de vuelta al usuario[16].

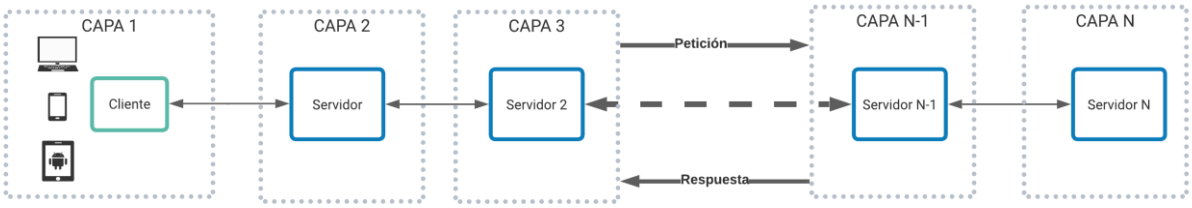


Figura 8 - Arquitectura N Capas

2.4 Arquitectura de software monolítica

La arquitectura monolítica es aquella en la que todos los aspectos funcionales del software quedan acoplados y sujetos en un mismo programa (una sola unidad de despliegue); es decir, la información queda alojada de forma estable en un único servidor, por lo que los módulos no pueden separarse y cada uno depende enteramente del conjunto [19]. En un monolítico, la vista, la lógica de negocio y la capa de acceso a los datos son componentes que conviven en un mismo sistema (ver figura) [20].

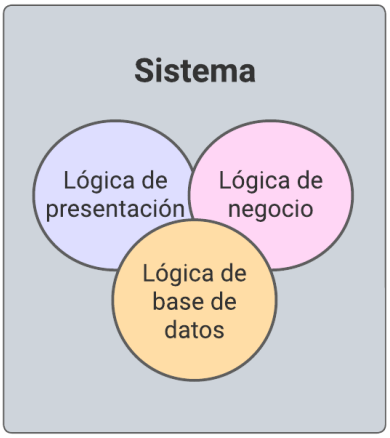


Figura 9 - Sistema monolítico.

Hay al menos tres tipos de sistemas monolíticos de los cuales se debe hablar: el sistema de proceso único, el monolítico distribuido y los monolíticos de sistemas de caja negra de terceros.

2.4.1 Monolítico de proceso único

El monolítico de proceso único es aquel sistema en el que todo el código se despliega como un solo proceso. Usualmente estos sistemas de un solo proceso son usados como simples sistemas distribuidos, ya que siempre terminan leyendo datos de una base de datos o almacenándolos en ella. Además de tener múltiples instancias de dicho monolítico, con el fin de generar robustez en el momento de fallo de uno de ellos[19].

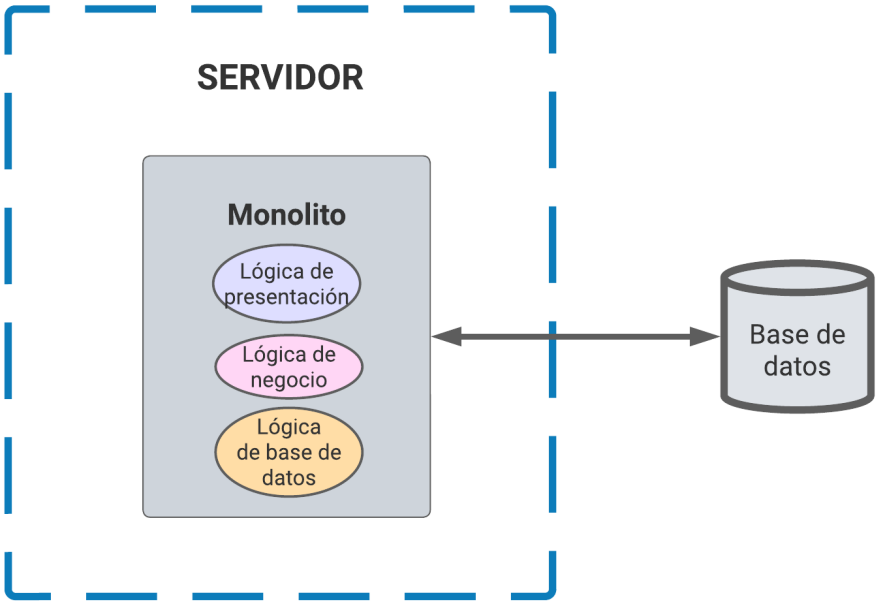


Figura 10 - Monolítico de proceso único

2.4.2 Monolítico modular

Un subconjunto y variación del monolítico de proceso único, es el monolítico modular, donde el proceso único está separado por módulos, cada uno de los cuales puede ser trabajado de forma independiente, pero todavía tienen que ser combinados para el despliegue. En un monolítico modular, la base de datos tiende a carecer de descomposición que encontramos en el nivel de código, lo que lleva a importantes desafíos al querer desechar el monolítico en el futuro[19].

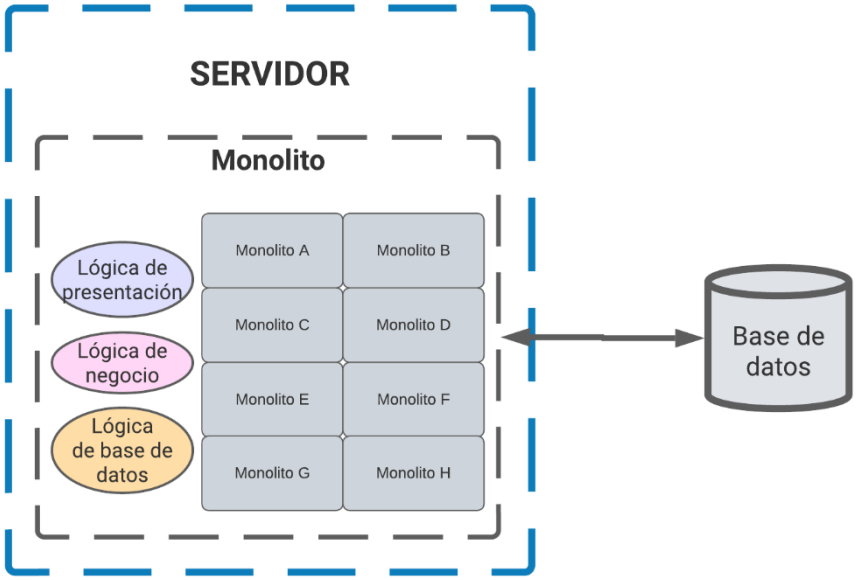


Figura 11 - Monolítico modular.

2.4.3 Monolítico distribuido

Un monolítico distribuido es un sistema que consta de múltiples servicios, pero que, por la razón que sea, todo el sistema debe desplegarse conjuntamente. Un monolítico distribuido puede cumplir la definición de una arquitectura orientada a servicios, pero con demasiada frecuencia no cumple estas mismas promesas. Los monolíticos distribuidos suelen surgir en un entorno en el que no se ha prestado suficiente atención a conceptos como la ocultación de la información y la cohesión de la funcionalidad empresarial, lo que ha dado lugar a arquitecturas muy acopladas en las que los cambios se extienden a través de los límites de los servicios, y cambios aparentemente inocentes que parecen tener un alcance local rompen otras partes del sistema[19].

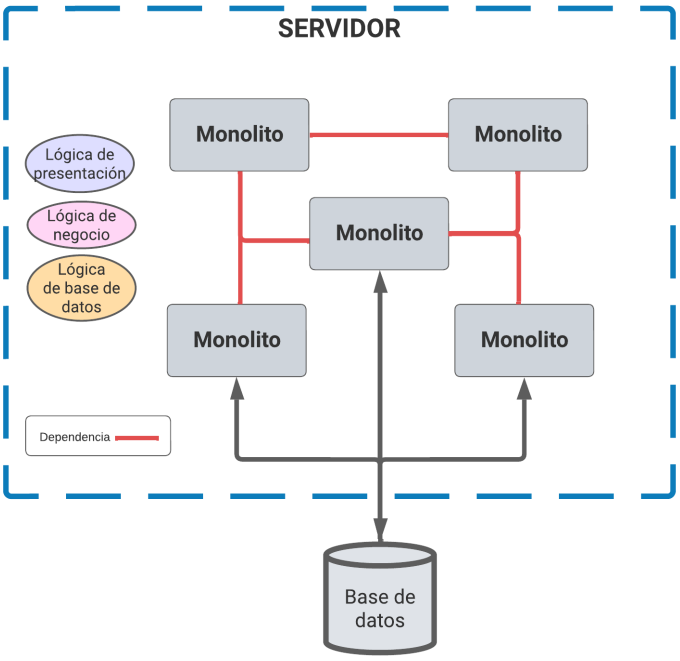


Figura 12 - Monolítico distribuido.

2.4.4 Monolíticos de sistemas de caja negra de terceros

Estos monolíticos son algunos programas de terceros que podemos querer "descomponer" como parte de un esfuerzo de migración. Esto podría incluir cosas como sistemas de nóminas, sistemas CRM y sistemas de RRHH. El factor común en este caso es que se trata de software desarrollado por otras personas, y no se tiene la capacidad de cambiar el código y que el sistema en conjunto depende de este. Puede tratarse de un software estándar que has desplegado en tu propia infraestructura, o puede ser un producto de software como servicio (SaaS) que se está utilizando[19].

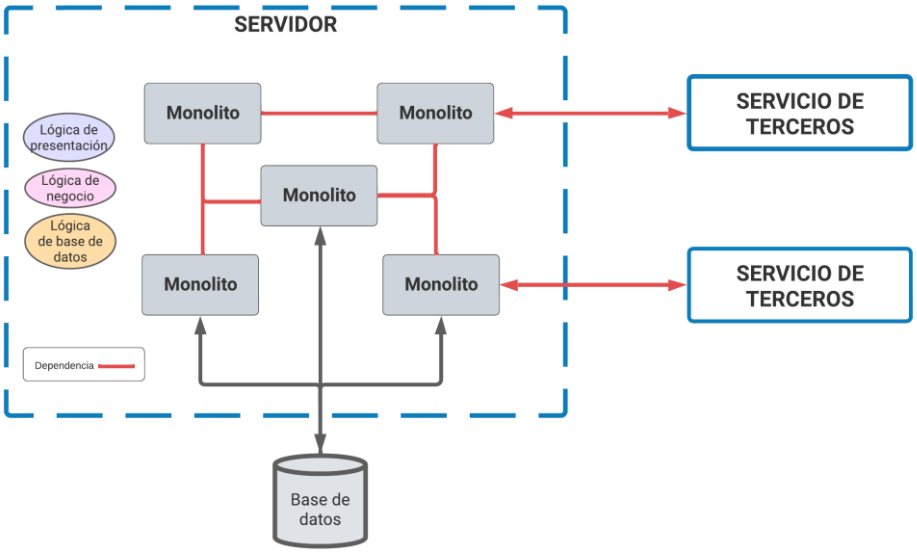


Figura 13 - Monolítico de sistemas de caja negra.

2.5 Arquitectura de software orientada a servicios

Una arquitectura orientada a servicios (SOA) es un patrón arquitectónico en el diseño de software en el que los componentes de una aplicación prestan servicios a otros componentes mediante un protocolo de comunicaciones basado en mensajería, es decir, un Servicio de Bus Empresarial. Los principios de la orientación a servicios son: independientes de cualquier proveedor, producto o tecnología. Los servicios pueden combinarse para proporcionar la funcionalidad de una gran aplicación de software[21].

En SOA hay dos roles principales: el de proveedor de servicios y el de consumidor de servicios. Un componente o agente de software puede desempeñar ambos papeles. La capa de consumidores es el punto donde los usuarios (humanos, otros componentes de la aplicación o terceros) interactúan con la SOA y la capa de proveedores consiste en todos los servicios dentro de la SOA [22].

SOA ha sido introducida para fomentar una interacción dinámica y de bajo acoplamiento entre servicios ofrecidos por diferentes proveedores, permitiendo el desarrollo de sistemas distribuidos altamente escalables, sus principales objetivos

consisten en soportar la interoperabilidad de servicios proveniente de diferentes proveedores y facilitar modificaciones que permitan al sistema evolucionar[23].

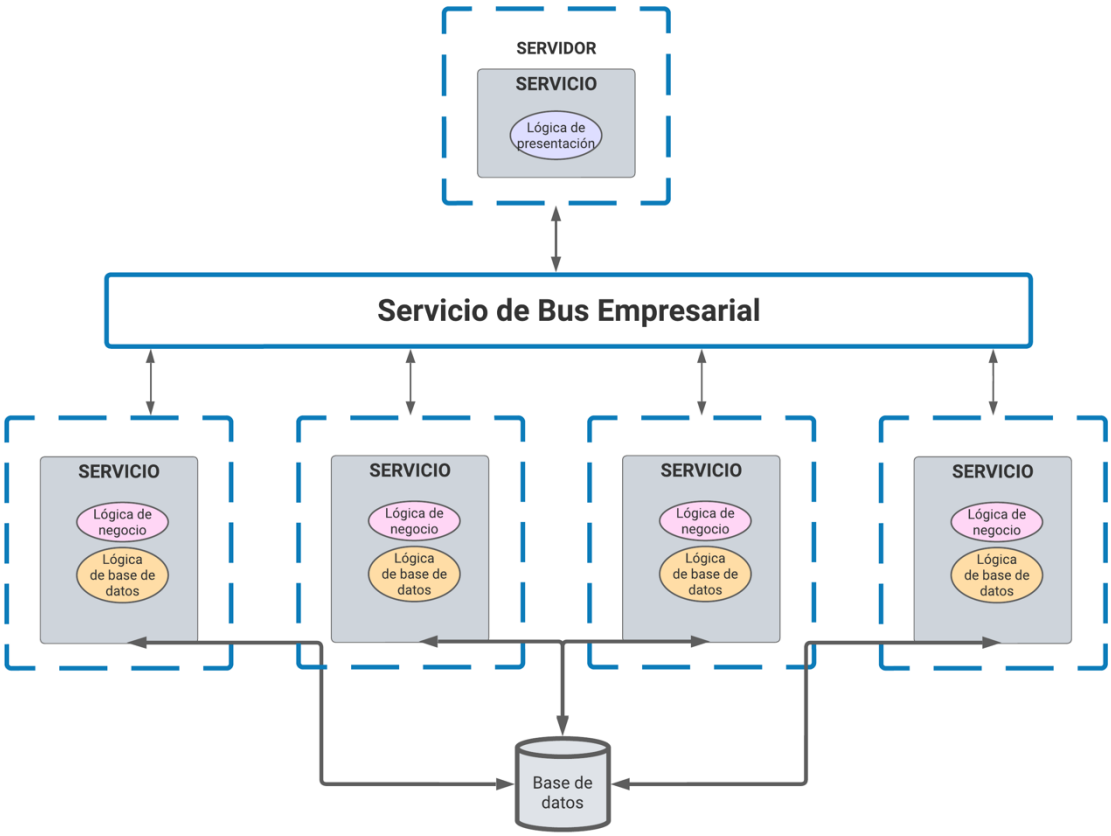


Figura 14 - Arquitectura orientada a servicios.

2.6 Arquitectura de software basada en microservicios

La arquitectura basada en microservicios puede ser considerado un enfoque específico de arquitectura orientada a servicios (SOA), teniendo su diferenciador en cómo se deben trazar los límites del servicio, donde se promueve el desarrollo y despliegue de aplicaciones compuestas por unidades independientes, autónomas, modulares y auto- contenidas, lo cual difiere de la forma tradicional o monolítico[24], [25].

2.6.1 Microservicios

Los microservicios son componentes de software que permiten modularizar un sistema, se podría decir que son pequeños servicios autónomos que trabajan juntos[25], [26]. Cada microservicio con un propósito diferente y específico deben ser lo suficientemente pequeños de tal manera que sean fáciles de mantener, además cada uno ejecutándose en su propio proceso y usando mecanismos ligeros de comunicación[25].

Existe un mínimo de gestión centralizada de estos servicios, los que pueden estar escritos en lenguajes de programación diferentes y utilizar diferentes tecnologías de almacenamiento de datos. También encapsulan el almacenamiento y la recuperación de datos, exponiendo los datos, a través de interfaces bien definidas. Entonces, las bases de datos están ocultas dentro del límite del servicio, ademas de utilizar diferentes tecnologías de almacenamientos de datos[27].

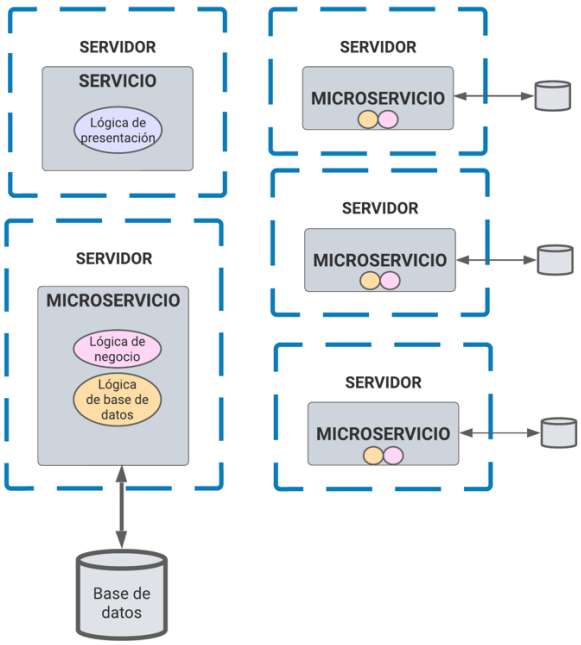


Figura 15 - Arquitectura basada en microservicios.

2.7 Comunicación entre microservicios

En la arquitectura basada en microservicios existen dos modelos de comunicación sincrónica y asincrónica.

2.7.1 Comunicación Sincrónica

La comunicación síncrona es a menudo considerada como el estilo de interacción petición/respuesta. Un microservicio realiza una petición a otro servicio y espera a que éste procese el resultado y envíe una respuesta, por consecuencia se conocerá inmediatamente la disponibilidad de este. En este estilo, es habitual que el solicitante bloquee su operación mientras espera una respuesta del servidor remoto[28].

Cuando se trata de la capa de comunicación entre microservicios, el enfoque síncrono es sin duda el más utilizado. Dentro de este tema, algunos protocolos son bien conocidos y otros no tanto. La gama de protocolos directos es la siguiente[29]:

- HTTP
- TCP
- WebSockets
- Sockets
- RPC
- SOAP

Podría decirse que comúnmente HTTP es implementado por los microservicios para comunicarse entre sí, utilizado normalmente con JSON (JavaScript Object Notation). El problema con este enfoque es que, con HTTP, JSON puede generar un tiempo de procesamiento no deseado para enviar y traducir la información. Algunos equipos que utilizan JSON con HTTP sólo adoptan la estrategia “keep alive” para la comunicación entre aplicaciones y las conexiones convencionales a las API[29]. El término “keep alive” hace referencia a las conexiones de comunicación en una red que no están terminadas, pero se mantienen hasta que el cliente o servidor interrumpen la conexión.

Cuando se trata de HTTP, la API con JSON es prácticamente normativa. Sin embargo, para la comunicación interna entre microservicios, esto es bastante cuestionable. Un buen enfoque, en este caso, teniendo en cuenta los problemas de latencia y traducción de datos, es el uso de tráfico binario para la comunicación entre microservicios. Hay algunas opciones muy interesantes para este enfoque: Avro, Protocol Buffer con CPRM, y Thrift son algunos ejemplos. Otro punto importante es que con binario no estamos atados a ninguna tecnología específica, y cambiar la interfaz de comunicación con esta tecnología es extremadamente sencillo[29].

2.7.2 Comunicación Asincrónica

En algunas comunicaciones directas entre microservicios, la sincronización puede ser importante, pero hay otras ocasiones en las que el proceso puede ser simplemente asíncrono; no hay necesidad de hacer una llamada directa al servidor para obtener una respuesta inmediata o confirmación de éxito, todo lo que se requiere es simplemente ejecutar una tarea[28]. Para este enfoque, un intermediario de mensajes o broker es ideal entre los microservicios coordinando las peticiones y respuestas[30].

Algunas aplicaciones de software parecen una buena opción para los intermediarios de mensajes, como RabbitMQ, ActiveMQ, ZeroMQ, Kafka, y Redis. Cada una de estas opciones tiene sus propias peculiaridades, algunas son más rápidas, otras son más resistentes. Una vez más, el entorno empresarial va a determinar qué tecnología se utiliza[29].

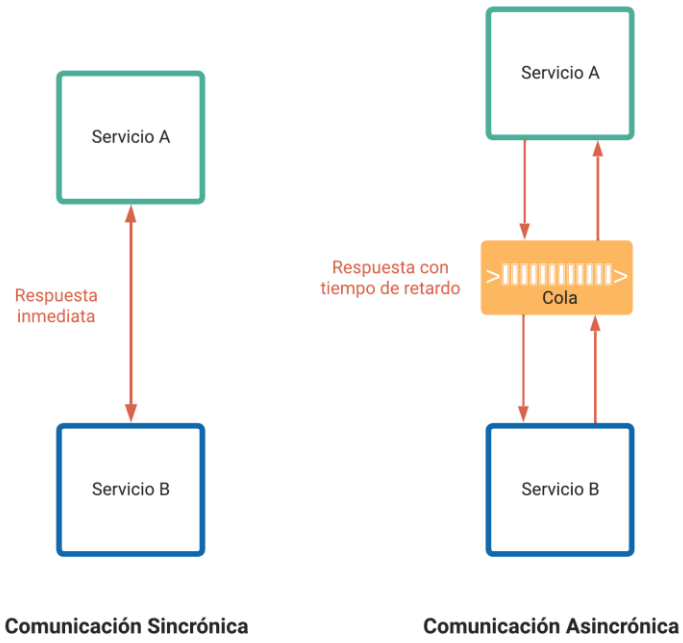


Figura 16 - Comunicación sincrónica vs comunicación asincrónica.

2.8 Patrones de Diseño de Software

En 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides introducen popularmente los patrones de diseño a la industria del software con el libro “Design Patterns: Elements of reusable object oriented programming”, donde hablan de ellos como estructuras de clases e interfaces que solucionan un problema general, pensadas para que las aplicaciones robustas sean escalables y flexibles, además rápidas de desarrollar y diseñar. Ellos mencionan “Cada patrón describe un problema el cual ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución del problema, de una forma que se pueda usar esa solución muchas veces, sin hacerlo de la misma forma dos veces …” [31].

Entonces los patrones de diseño como tal son soluciones típicas a los problemas que se producen habitualmente en el diseño de software. Hay que tener claro que un patrón de diseño no es un trozo de código específico que se pueda copiar, sino un concepto general para resolver un problema concreto, así se podría seguir los

detalles del patrón e implementar una solución que se adapte a las realidades del programa en desarrollo[32].

Los patrones de diseño ayudan a desarrollar con mayor velocidad, debido a que no es necesario pensar detalladamente sobre cómo resolver un asunto específico. Así mismo el uso de estos patrones, nos proporciona la facilidad de prevenir problemas y mejorar la legibilidad y claridad del código debido a que otros ingenieros de software entenderían la lógica utilizada de los patrones de diseño, porque su documentación está en todas partes. Así mismo es importante resaltar que los patrones de diseño nos proporcionan soluciones generales, las cuales no están atadas a un tipo específico de problema[32].

2.8.1 Principios de diseño de Software

2.8.1.1 Principio de Encapsulamiento

El objetivo principal de este principio es minimizar el efecto causado por los cambios, por lo que para aplicar el principio se debe identificar los aspectos variables del aplicativo y separarlos de lo que permanece igual. Con el fin de aislar las partes del programa que varían en módulos independientes, protegiendo el resto del código de efectos adversos. Como beneficio principal se reduce el tiempo invertido en realizar cambios y se puede dedicar dicho tiempo a la implementación de nuevas funciones[32].

2.8.1.2 Principio – Programar una Interfaz no una Implementación

Este principio trata de abstracciones, no de clases concretas, su propósito es tener tal flexibilidad en el código que se pueda extender fácilmente sin romper ningún código existente. La lógica de este principio es generar y hacer uso de interfaces en lugar de clases concretas, específicamente, cuando varios objetos necesiten colaboración de un objeto en común y de sus métodos[32]. Explicándolo mejor, en vez de generar varios objetos tipo clase con los mismos métodos se generará un objeto tipo interfaz que sin problema podrá contener estos métodos y dichas clases únicamente solicitarán uso de la interfaz en cuestión.

2.8.1.3 Principio – Prevalencia de Composición ante la Herencia

La herencia es probablemente la forma más obvia y fácil de reutilizar código entre clases, sin embargo, el mal uso de la herencia se hace evidente cuando la necesidad de una pequeña funcionalidad equivale a un cambio bastante difícil de hacer[32]. A continuación, se menciona algunos los problemas que se pueden presentar:

- Obligatoriamente se debe implementar todos los métodos abstractos de la clase padre, aunque no se los vaya a utilizar[32].
- Cuando se sobrescriben métodos hay que asegurarse de que el nuevo comportamiento sea compatible con el de la base[32].
- La herencia rompe la encapsulación de la superclase porque los detalles internos de la clase padre pasan a estar disponibles para la subclase[32].
- Las subclases están estrechamente vinculadas a las superclases. Cualquier cambio en cualquier cambio en una superclase puede romper la funcionalidad de las subclases[32].
- Intentar reutilizar código a través de la herencia puede llevar a crear jerarquías de herencia paralelas. La herencia suele tiene lugar en una sola dimensión, pero cuando hay dos o más o más dimensiones, tienes que crear muchas combinaciones de clases, inflando la jerarquía de clases[32].

Con el propósito de evitar estos inconvenientes este principio propone la composición. Ejemplificando, mientras que la herencia representa la relación “es un/a” entre clases (una moto es un trasporte), la composición representa la relación “tiene un/a” entre clases (una moto tiene un motor)[32].

2.8.1.4 Principios SOLID

Robert Martin los presentó en el libro “Agile Software Development, Principles, Patterns, and Practices”, en donde menciona los cinco principios en los cuales se basa el mnemotécnico en ingles SOLID destinados a hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener [33]. A continuación, hablaremos de cada uno de ellos.

- Principio de Responsabilidad Única (Single Responsibility Principle)

  Este principio se refiere a que cada clase sea responsable de una sola parte de la funcionalidad proporcionada por el software, y haga que esa responsabilidad esté completamente encapsulada [32], [33].

- Principio Abierto/Cerrado (Open/Closed Principle)

  El principio de Abierto/Cerrado dice que todas las clases debería estar abiertas para extenderse, pero debe estar cerrado para modificarse. Su objetivo es evitar que el código sea modificado sin intención causando errores o funcionalidades no esperadas [32], [33].

- Principio de Sustitución de Liskov (Liskov Substitution Principle)

  El principio de sustitución de Liskov dice que, al extender una clase de otra, se debe poder pasar los objetos de la subclase en lugar de objetos de la clase principal sin romper el código del cliente, esto significa que la subclase debe seguir siendo compatible con el comportamiento de la super clase [32]–[34]. Ejemplificando si la clase A es de un subtipo de la clase B, entonces se puede reemplazar B con A sin afectar el comportamiento del programa.

- Principio de Segregación de Interfaces (Interface Segregation Principle)

  El principio de segregación de interfaz dice que ninguna clase cliente debería estar obligado a depender de los métodos que no utiliza. Según el principio se debe dividir las interfaces “gruesas” en interfaces más granulares y específicas [32], [33]. En la práctica una interfaz existente no se debe agregar nuevos métodos que obliguen a implementar un comportamiento adicional no necesario a todas las clases que hacen uso de la interfaz; en lugar de agregar métodos a una interfaz existente, es mejor crear otra interfaz y que la clase que la necesite la implemente.

- Principio de Inversión de Dependencias (Dependency Inversion Principle)

  El principio de inversión nos dice que no deben existir dependencias entre clases de bajo nivel y de alto nivel. Las clases de alto nivel no deben

depender de las de bajo nivel, este es un error común cuando existe incertidumbre en el funcionamiento del software por lo que primero se diseña las clases de bajo nivel y no se esta seguro de lo que es posible hacer en el nivel superior generando una dependencia erronea, la propuesta de este principio es invertir la dirección de dependencia [32], [33].

Las clases de bajo nivel son aquellas que implementan operaciones básicas como grabar en disco, transferencia de datos a través de una red, conexión a una base de datos, etc. Mientras que las clases de alto nivel contiene la lógica de negocio compleja que dirige a las clases de bajo nivel para hagan algo [32], [33].

2.8.2 Clasificación de los patrones de diseño

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña.

Los patrones más básicos y de más bajo nivel suelen llamarse “idioms” y su traducción al español sería *modismos*, normalmente se aplican a un único lenguaje de programación. Por otro lado, los patrones más universales y de más alto nivel son los *patrones de arquitectura*, estos son capaces de ser implementados prácticamente en cualquier lenguaje, y al contrario de otros patrones pueden utilizarse para diseñar la arquitectura de una aplicación completa[32].

A continuación, vamos a clasificar los patrones de diseño por su propósito:

- Creacional

Los patrones de creación proporcionan varios mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente[32]. Entre los cuales están:

  - Método de fábrica ó Constructor virtual (Factory Method)
  - Fábrica abstracta (Abstract Factory)
  - Constructor (Builder)

- De Prototipo (Prototype)
- Monótono (Singleton)

- Estructural

Los patrones estructurales explican cómo ensamblar objetos y clases en grandes estructuras, manteniendo estas estructuras flexibles y eficientes[32]. Entre los cuales están:

- Adaptador (Adapter)
- Puente (Bridge)
- Compuesto (Composite – Object Tree)
- Decorador (Decorator)
- Fachada (Facade)
- Peso mosca (Flyweight – Cache)
- Delegador (Proxy)

- Comportamiento

Los patrones de comportamiento se refieren a los algoritmos y a la asignación de responsabilidades entre objetos[32]. Entre los cuales están:

- Cadena de responsabilidad (Chain of Responsibility)
- Comando (Command)
- Mediador (Mediator)
- De Recuerdo (Memento)
- Observador (Observer)
- De Estado (State)
- De Estrategía (Strategy)
- Método de Plantilla (Template Method)
- De Visitante (Visitor)

2.8.3 Diseño Guiado por el Dominio

El Diseño Guiado por el Dominio o más conocido por sus siglas en ingles DDD (Domain-Driven Design), inicialmente este patrón de arquitectura fue presentado y popularizado por el programador Eric Evans en 2004 en su libro “Domain-Driven Design: Tackling Complexity in the Heart of Software”, el diseño basado en dominios es la expansión y aplicación del concepto de dominio tal como se aplica al desarrollo de software. DDD tiene como objetivo facilitar la creación de aplicaciones complejas al conectar las piezas de software relacionadas en un modelo que va a estar en constante evolución[35].

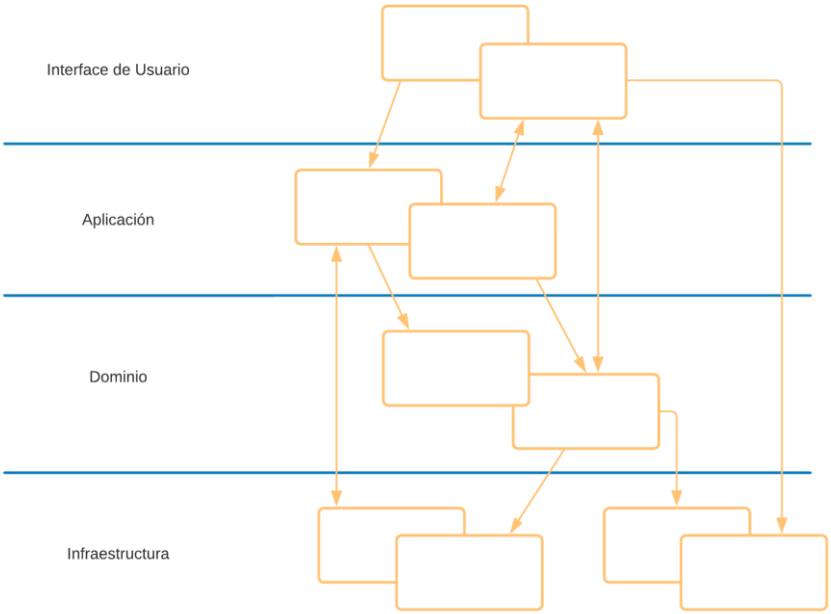


Figura 17 - Modelo de capas del Diseño Guiado por el Dominio.

Cada capa tiene un propósito específico que limita sus funciones, con el fin de cumplir con el objetivo de este patrón arquitectónico.

Capa	Descripción
Interfaz de usuario (o capa de presentación)	Responsable de mostrar información al usuario e interpretar los comandos del usuario. El actor externo podría

	a veces ser otro sistema informático en lugar de un usuario humano.
Capa de aplicación	Define los trabajos que se supone que debe hacer el software y dirige los objetos del dominio expresivo para resolver problemas. Las tareas de las que es responsable esta capa son significativas para el negocio o necesarias para la interacción con las capas de aplicación de otros sistemas. Esta capa se mantiene delgada. No contiene reglas de negocio ni conocimiento, solo coordina tareas y delega trabajar con colaboraciones de objetos de dominio en la siguiente capa hacia abajo. No tiene un estado que refleje la situación comercial, pero puede tener un estado que refleje el progreso de una tarea para el usuario o el programa.
Capa de dominio (o capa de modelo)	Responsable de representar conceptos del negocio, información sobre la situación del negocio y reglas de negocio. Aquí se controla y utiliza el estado que refleja la situación empresarial, aunque los detalles técnicos de almacenamiento se deleguen a la infraestructura. Esta capa es el corazón del software empresarial.
Capa de infraestructura	Proporciona capacidades técnicas genéricas que admiten las capas superiores: envío de mensajes para la aplicación, persistencia para el dominio, widgets de dibujo para la interfaz de usuario, etc. La capa de infraestructura también puede admitir el patrón de interacciones entre las cuatro capas a través de un marco arquitectónico.

Tabla 1 - Detalle de las capas del Diseño Guiado por el Dominio [35]

2.8.4 Repositorio (Repository)

El patrón Repositorio es un parte del patrón DDD, destinado a mantener las preocupaciones de persistencia fuera del modelo de dominio del sistema. Una o más abstracciones de persistencia (interfaces) se definen en el modelo de dominio, y estas abstracciones tienen implementaciones en forma de adaptadores de persistencia definidos en otra parte de la aplicación[36].

Martin Fowler describe a un repositorio como: “Un repositorio realiza las tareas de un intermediario entre las capas del modelo de dominio y el mapeo de datos, actuando de manera similar a un conjunto de objetos de dominio en la memoria. Los objetos del cliente construyen consultas declarativamente y las envían a los repositorios para obtener respuestas. Conceptualmente, un repositorio encapsula un conjunto de objetos almacenados en la base de datos y las operaciones que se pueden realizar sobre ellos, proporcionando una forma más cercana a la capa de persistencia. Los repositorios, además, soportan el propósito de separar, de forma clara y unidireccional, la dependencia entre el dominio de trabajo y la asignación o mapeo de datos”[12].

2.8.5 Arquitectura basada en Eventos (Event-Driven Architecture)

En las aplicaciones monolíticas, la gestión de datos es relativamente sencilla ya que sólo hay una única base de datos y suele ser relacional. Cuando se utilizan bases de datos relacionales, las aplicaciones pueden utilizar el modelo ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) y cambiar múltiples filas y tablas en una transacción[37],[38]. Sin embargo, en la arquitectura de microservicios los datos de cada servicio son privados y sólo se puede acceder a ellos a través de la API que proporciona el servicio [39]. Esto introduce los siguientes retos principales de datos distribuidos[38]:

- Recuperar datos de múltiples servicios.
- Implementar transacciones de negocio que mantengan la consistencia de los datos a través de múltiples servicios.

La arquitectura dirigida por eventos, o EDA por sus siglas en inglés, es un patrón de arquitectura asíncrona distribuida muy popular, que puede utilizarse para superar los retos de los datos distribuidos. Es altamente escalable y flexible[38], [40]. En EDA cada microservicio publica un evento cuando ocurre algo notable, por ejemplo, el Servicio de Pedidos publicaría un nuevo evento cuando un pedido ha sido creado o modificado. Los otros microservicios se suscriben a los eventos que les interesan (por ejemplo, el Servicio de Inventario se suscribiría a los eventos de nuevos pedidos porque necesita reducir el número de productos en la base de datos de inventario). Un evento puede definirse como "un cambio significativo en el estado" [41]. Los eventos pueden utilizarse para implementar transacciones de negocio que abarcan múltiples servicios. Las transacciones pueden ser representadas por una serie de pasos donde cada paso es un microservicio, que actualiza o crea una entidad de negocio y publica un evento que desencadena el siguiente paso. EDA tiene dos topologías principales: Mediador y Broker[38]:

- Topología Mediador: útil para eventos con múltiples pasos. La topología de mediador tiene cuatro componentes principales: colas de eventos, un mediador de eventos, canales de eventos y procesadores de eventos. El flujo de eventos comienza cuando un cliente envía un evento a una cola, que lo transporta al mediador de eventos. A continuación, el mediador envía eventos asíncronos adicionales a los canales de eventos para ejecutar cada paso del proceso. Los procesadores de eventos escuchan en los canales de eventos y ejecutan una lógica de negocio específica para procesar el evento [40].

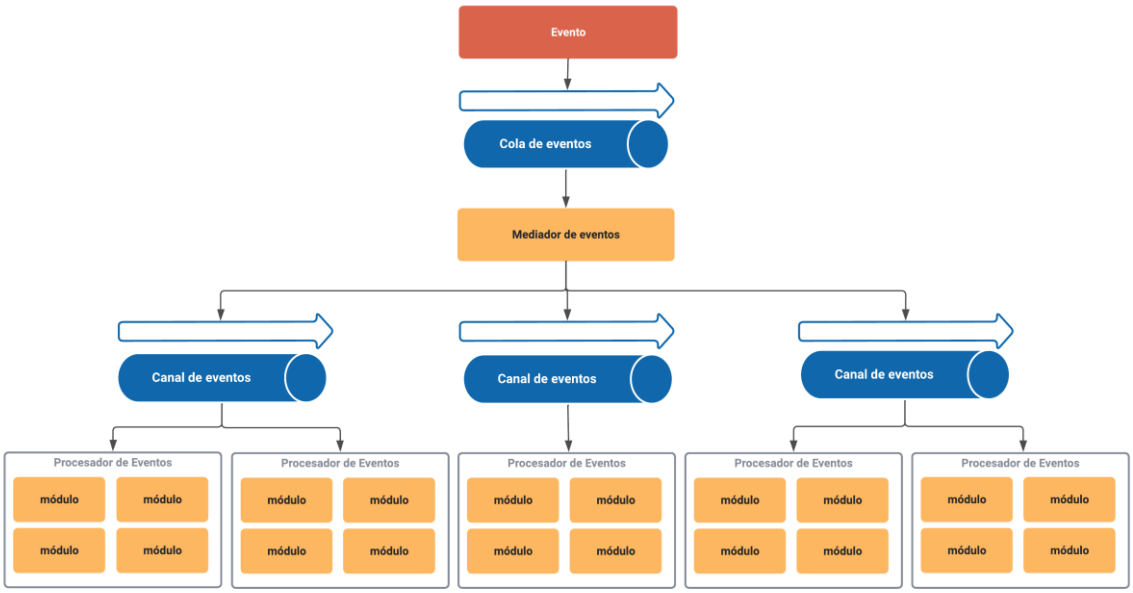


Figura 18 - Arquitectura basada en Eventos, topología mediador.

- Topología Broker: en esta topología no hay un mediador de eventos central que orqueste el evento inicial. Hay dos tipos de componentes: broker y procesadores de eventos. Cada procesador de eventos es responsable de procesar un evento y publicar un nuevo evento para notificar a otros la acción realizada. El broker proporciona los siguientes canales de eventos: colas de mensajes, temas de mensajes o una combinación de ambos. Esta topología resulta útil cuando existe un flujo de procesamiento de eventos relativamente sencillo[40].

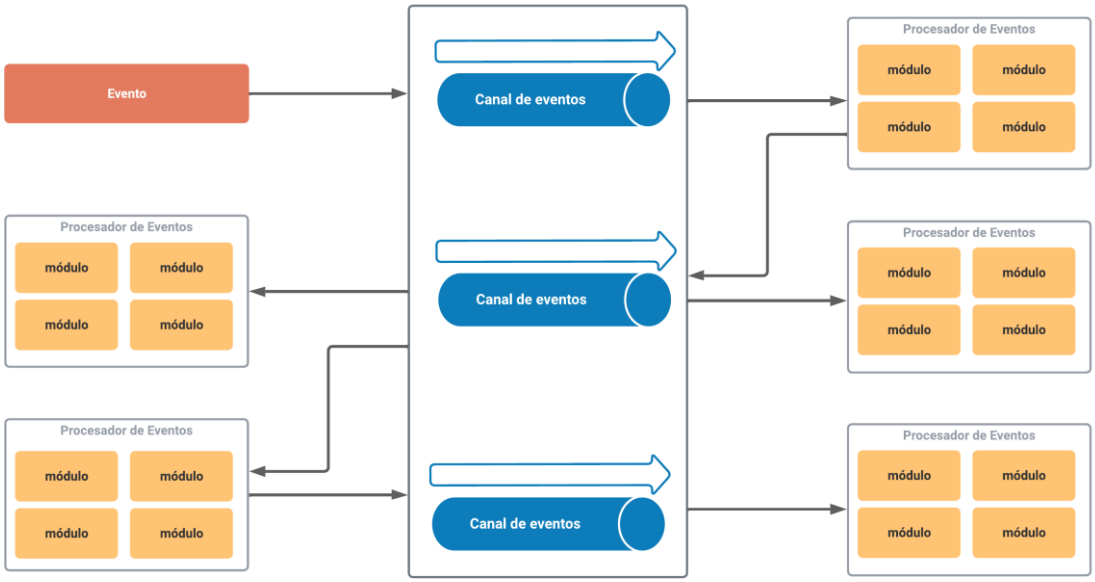


Figura 19 - Arquitectura basada en Eventos, topología broker.

2.8.6 Mediador (Mediator)

Mediador es un patrón de diseño de comportamiento que permite reducir las dependencias caóticas entre objetos. El patrón restringe la comunicación directa entre objetos y los fuerza a colaborar en una solamente a través de un objeto mediador[32].

El patrón Mediador sugiere poner fin a toda comunicación directa entre los componentes que se desea independizar. En su lugar, estos componentes deben colaborar indirectamente, llamando a un objeto mediador especial que redirija las llamadas a los componentes apropiados. Como resultado, los componentes sólo dependerán de una única clase mediadora, en lugar de estar acoplados a docenas de sus colegas[32].

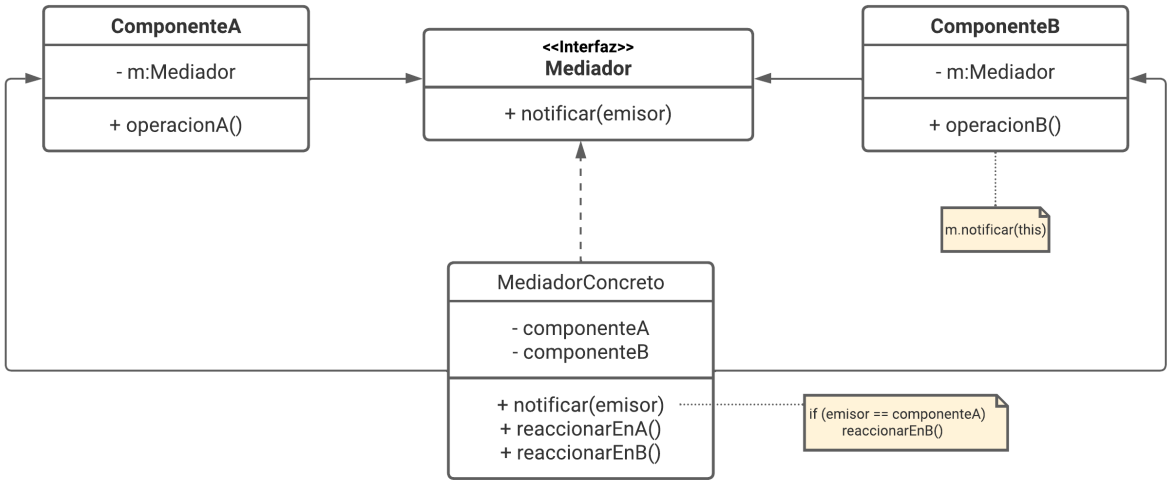


Figura 20 - Estructura del patrón Mediador. [32]

2.9 Scrum

Scrum es un marco de trabajo (framework) liviano que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptativas para problemas complejos[42].

2.9.1 Equipo (Scrum Team)

El núcleo fundamental del Scrum es un equipo de personas conocido como Scrum Team, el cual está conformado por un Scrum Master, un Product Owner y Developers. Algo importante de resaltar es que dentro del Scrum Team no existe subequipos, ni jerarquías; logrando una unidad de profesionales enfocados en un objetivo (Objetivo del Producto) a la vez [42].

El Scrum Team es multifuncional, autogestionable, y está formado generalmente por 10 personas o menos con el sentido de que sea un equipo pequeño permitiendo agilidad, mejor comunicación entre los miembros, por consiguiente, mayor productividad [42].

El Scrum Team es responsable de todas las actividades relacionadas con el producto, desde la colaboración de los interesados, la verificación, el mantenimiento, la operación, la experimentación, la investigación y el desarrollo, y

cualquier otra cosa que pueda ser necesaria. Para así crear un Increment valioso y útil en cada Sprint. Scrum define tres responsabilidades específicas dentro del Scrum Team: los Developers, el Product Owner y el Scrum Master [42].

Desarrolladores (Developers)

Los Developers son las personas las cuales se comprometen a crear los incrementables de cada Sprint. Son responsables de crear un plan para el Sprint, adaptar su plan diariamente hacia el Objetivo del Sprint [42].

Propietario del Producto (Product Owner)

El Product Owner es la persona que tiene a cargo la maximización del valor del producto resultante del trabajo del Scrum Team, por ello representa las necesidades de los diferentes interesados en el Product Backlog. Es responsable de desarrollar y comunicar con claridad el Objetivo del Producto y los elementos del Product Backlog, ordenar los elementos del Product Backlog, y asegurarse de la visibilidad, trasparencia y entendimiento del Product Backlog [42].

Scrum Master

El Scrum Master es el responsable de establecer Scrum, es decir, colabora en la comprensión teórica y práctica de Scrum dentro del Scrum Team y la organización. Dentro de sus compromisos están:

- Con el Scrum Team: guiar a sus miembros en ser autogestionados y multifuncionales, eliminar impedimentos para el progreso, constatar que todos los eventos sean efectuados positivamente;
- Con el Producto Owner: ayudar a encontrar técnicas para estipular los Objetivos del Producto y la gestión del Product Backlog;
- Con la organización: planificar y asesorar implementaciones de Scrum, eliminar barreras entre los interesados y el Scrum Team [42].

2.9.2 Eventos

El principal evento en el marco de trabajo Scrum es El Sprint el cual es un contenedor para los demás eventos (*Sprint Planning*, *Daily Scrum*s, *Sprint Review*

y *Sprint Retrospective*), es decir estos se ejecutarán dentro de El Sprint. A demás cada uno de los eventos son una oportunidad para realizar inspecciones y adaptar los eventos artefactos Scrum [42].

Los eventos se usan con el fin de regularizar y minimizar las reuniones no definidas dentro del marco, es importante que dichos eventos se realicen al mismo tiempo y en el mismo lugar para reducir su complejidad [42].

El Sprint

Los Sprint son eventos que tiene una duración fija de un mes o menos para así crear consistencia, permitiendo cumplir el Objetivo del Producto y garantizar en el transcurso la inspección y adaptación del progreso. Son considerados proyectos cortos y un nuevo sprint comienza apenas concluya el Sprint anterior [42].

Durante el Sprint:

- No se realizan cambios que pongan en riesgo el Objetivo del Sprint;
- La calidad no disminuye;
- El Product Backlog se refina según sea necesario
- El alcance se puede aclarar y renegociar con el Product Owner a medida que se aprende más.

Planificación del Sprint (Sprint Planning)

El *Sprint Planning* comienza el *Sprint* al establecer el trabajo que se realizará para el *Sprint*. El *Scrum Team* crea este plan resultante mediante trabajo colaborativo, es así que el *Product Owner* se asegura de que los asistentes estén preparados para discutir los elementos más importantes del *Product Backlog* y cómo se relacionan con el Objetivo del Producto. El *Scrum Team* también puede invitar a otras personas a asistir a la *Sprint Planning* para brindar asesoramiento [42].

Durante el Sprint Planning se trata de responder las siguientes interrogantes:

1. ¿Por qué es valioso este *Sprint*? Es decir, como el producto puede aumentar su valor y como es importante para los interesados en él [42].

2. ¿Qué se puede hacer en este Sprint? Es decir, cuanto se puede aprender del proyecto y hasta que capacidad se puede llevarlo, para ello, los Developers incluyen elementos del Product Backlog en el Sprint refinándolo y completándolo [42].

3. ¿Cómo se realizará el trabajo elegido? Es decir, la planificación del trabajo para crear un Increment que cumpla con las definiciones de calidad. La forma de hacerlo queda a criterio de los Developers. Nadie más les dice cómo convertir los elementos del Product Backlog en Increments de valor [42].

Scrum Diario (Daily Scrum)

Es un evento que tiene una duración de 15 minutos, en donde se revisa el progreso del Objetivo del Sprint y se ajusta el Sprint Packlog según sea necesario. Los Developers son los encargados en realizarlo para ajustar el plan hacia el Objetivo del Sprint y produzca un plan viable para el siguiente día de trabajo, mejorando la comunicación del proyecto con el fin de evitar errores y tomar decisiones con agilidad [42].

Revisión del Sprint (Sprint Review)

Tiene como finalidad la inspección del resultado de un Sprint y así realizar las adaptaciones futuras al proyecto [42].

Retrospectiva del Sprint (Sprint Retrospective)

Es una retroalimentación del proceso, es decir, identifica y analiza las mejoras y problemas para planificar formas de aumentar la calidad y efectividad al Sprint [42].

2.9.3 Artefactos

Un artefacto de Scrum es una representación de trabajo o valor y son diseñados para maximizar la transparencia de información clave [42].

Product Backlog

Se compone del objetivo del producto y es una lista emergente y ordenada de lo que se necesita para mejorar el producto [42].

Los elementos del Product Backlog se consideran para usarlos en un evento de Sprint Planning donde adquieren refinamiento que es dividir y detallar los elementos del Product Backlog en elementos más pequeños y precisos [42].

Los Developers son los encargados de la dimensión del Product Backlog

Sprint Backlog

Está compuesto por el objetivo del Sprint, los elementos de Product Backlog y el plan de acción del Increment [42].

Es el plan realizado por los Developers durante el Sprint para lograr su objetivo, por ello se actualiza durante toda la iteración del Sprint especialmente a través del Daily Scrum [42].

Incrementable (Increment)

Es el resultado final de un Sprint o la Suma de todos los elementos y desarrollo de los Sprint, estos se verifican cuidadosamente para que funciones juntos y así proporcionen valor [42].

El trabajo no puede considerarse parte de un Increment a menos que cumpla con la Definición de Terminado que es una descripción formal del estado del Increment cuando cumple con las medidas de calidad requeridas para el producto [42].

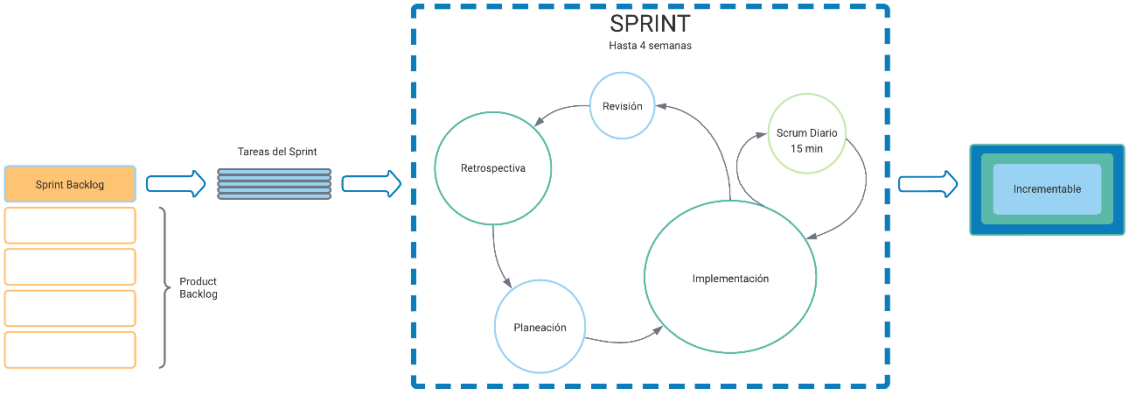


Figura 21 - Marco de trabajo SCRUM.

III ESTADO DEL ARTE

La arquitectura de software en las últimas 4 décadas ha evolucionado significativamente debido al desarrollo de nuevas tecnologías.

Hasta los años 80, los sistemas de software eran en gran medida monolíticos. La tendencia de estos sistemas eran ejecutarse en un único ordenador ya sean mainframes centrales con terminales o computadoras personales. El software se desarrollaba como "programas", y la arquitectura era en gran medida una preocupación del proveedor, heredada de la plataforma que utilizaban los desarrolladores; destacando en esta época las aplicaciones de escritorio que cumplían en su totalidad con dichas características [43].

Ya en los 90’s, los sistemas se orientaron al concepto de distribución de esfuerzos o más conocidos como sistemas distribuidos, en donde el principal cambio radicaba que el procesamiento lineal se convirtió en un procesamiento por lotes y la arquitectura Cliente Servidor de tres niveles era el estándar de los sistemas empresariales [43]. Estos cambios implicaban más decisiones arquitectónicas que antes, sin embargo, el estilo arquitectónico estaba regido en gran parte por los proveedores que suministraban las herramientas de desarrollo y el sistema software, es decir los programas heredaban estructura de aplicaciones monolíticas caracterizándose aún por la rigidez de ser un solo paquete ejecutable [44].

Antes de comenzar los años 2000, el internet ya se había situado como tecnología principal. Esta red mundial exigía a las organizaciones sistemas siempre conectados y encendidos. Sistemas que empezaron siendo sitios web, pero poco a poco fueron evolucionando y cambiando su enfoque para proveer funciones de empresa a consumidor y de empresa a empresa. Así, la arquitectura tuvo grandes desafíos para soportar cualidades no funcionales impredecibles como fueron en su momento el rendimiento, la escalabilidad y la seguridad.

En la década de 2010, Internet ya era un servicio básico con el que se contaba. Esto hizo que los sistemas vuelvan a evolucionar para “ser usados desde cualquier lugar para cualquier propósito” [43]. La arquitectura de software adoptó estilos que

permitían la interconexión entre aplicaciones o incluso componentes de software por medio de APIs. A inicios de esta década se establecieron los protocolos de servicios web SOAP y REST, quienes permiten la comunicación entre aplicaciones independientemente del lenguaje en el que fueron desarrolladas [44]. Una arquitectura flexible en definitiva era una propiedad necesaria para esta época.

Para el 2011, algunas empresas estaban tratando de optimizar sus arquitecturas orientadas a servicios, con el fin de obtener funcionalidades especificas e implementaciones rápidamente [43]. Todo ello bajo la premisa de que el alcance funcional de estos servicios debe ser pequeño, siendo fáciles de entender y por consiguiente de cambiar; y más importante con la perspectiva de que se pudiera reescribir en otras pilas de tecnología siempre y cuando todo necesitara escalar. En este mismo año dicho comportamiento evolutivo, fue detectado por James Lewis trabajador de una consultora llamada ThoughtWorks, nombrando a los servicios con estas características como “microaplicaciones”, término que fue discutido y renombrado más tarde por el mismo Lewis en una cumbre arquitectónica, tomando el nombre de “microservicios”.

La arquitectura de software basada en microservicios asegura la disponibilidad de una aplicación además de su independencia total en la codificación, siendo la arquitectura más moderna y base para los avances futuros [44].

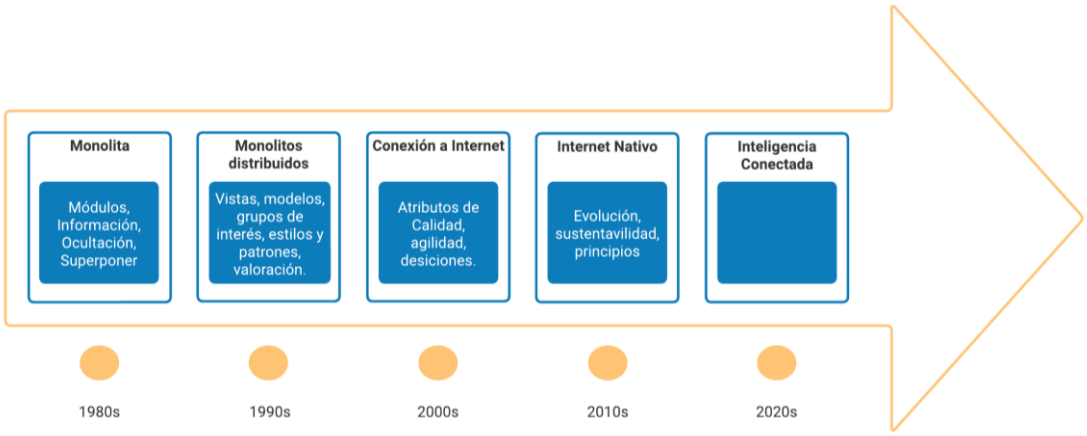


Figura 22 - La evolución de sistemas de software.

Con la aparición de la computación en nube, los proveedores se preocupan por suministrar plataformas de aplicaciones completas basadas en la nube, y muchos

sistemas de Internet se despliegan en entornos de plataformas como servicio Infraestructura como servicio (IaaS) y Plataforma como servicio (PaaS) en lugar de en la infraestructura informática tradicional. Sobre la base de las tendencias actuales, la siguiente fase de evolución parece ser la de los sistemas inteligentes conectados [43]. La IA, en particular el aprendizaje automático se está convirtiendo en la corriente principal, y las redes rápidas y fiables se están volviendo omnipresentes, permitiéndonos conectar "cosas" a nuestros sistemas, así como a los ordenadores tradicionales [45]. Estos sistemas pasarán de proporcionar a los usuarios acceso en cualquier lugar a proporcionar asistencia inteligente.

En esta quinta era, la "inteligencia" será una de las principales preocupaciones de los proveedores, ya que los sistemas convencionales utilizarán plataformas que amplíen las plataformas básicas PaaS y del Internet de las cosas (IoT) con servicios avanzados, como las capacidades de aprendizaje automático empaquetadas[43].

El Internet de las cosas es una de las tendencias de gran avance tecnológico para esta década, basando sus conceptos y estructura en las características primas de los microservicios, evidenciando la importancia de la arquitectura en microservicios en esta época [44]. En definitiva, la evolución de los sistemas de software lleva consigo la necesidad de un cambio en el modelo también evolutivo de despliegue de On-Premises a Cloud Computing.

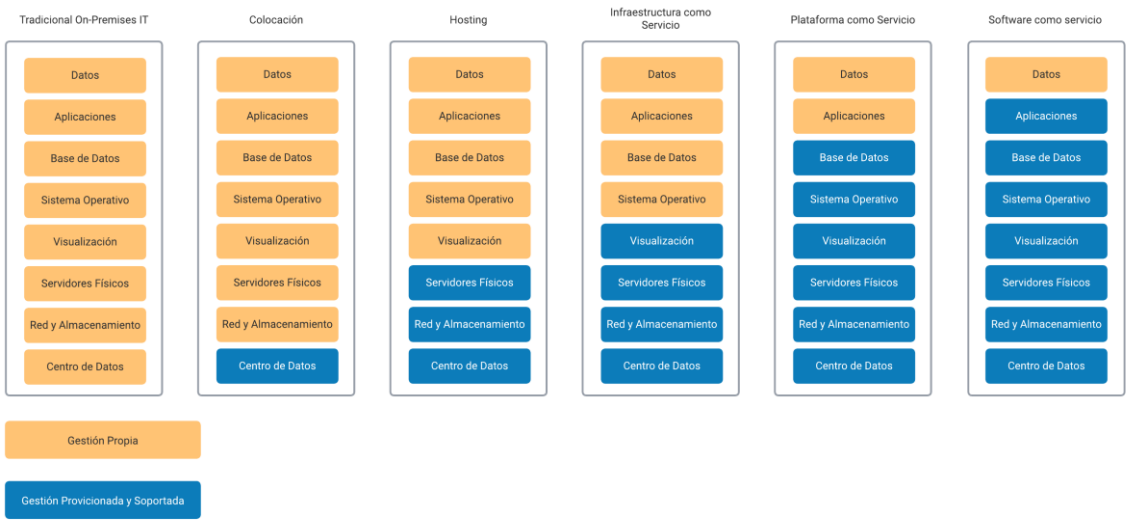


Figura 23 - Modelo de despliegue On-Premises vs Cloud Computing

La evolución de la arquitectura de software ha sido paralela a la de la industria del software, y las técnicas y preocupaciones de los arquitectos han cambiado en respuesta a los retos que ha enfrentado la industria.

Un claro ejemplo de las aplicaciones que han pasado por este proceso evolutivo son las aplicaciones de streaming musical que comenzaron a surgir a principios de la década de 2000, aunque la idea de la transmisión de audio en línea se remonta a mediados de los años 90 con el lanzamiento de RealAudio. Sin embargo, la verdadera popularización de las aplicaciones de streaming musical comenzó a partir del lanzamiento de servicios como Napster, en 1999, que permitía a los usuarios compartir archivos MP3 entre sí a través de una red peer-to-peer[46], [47].

En los años siguientes, surgieron otros servicios de streaming musical como Pandora, Spotify y Apple Music, cada uno con su propia propuesta de valor y modelo de negocio. Es ahí donde las aplicaciones de streaming musical usado la web toman mucha más fuerza, debido en gran parte a la creciente penetración de la banda ancha, lo que permitió a los usuarios acceder a la música en línea de manera más fácil y conveniente[47]–[49].

La evolución de las aplicaciones web de streaming musical ha sido impulsada por la tecnología de la información, los cambios en los hábitos de consumo de música, la competencia en la industria de la música, entre otros. Estos factores han llevado a mejoras en la calidad del servicio, la oferta de características más avanzadas y una experiencia de escucha más personalizada para los usuarios. Todas estas ventajas se asientan sobre el tener aplicaciones más accesibles y eficientes[48], [49].

Es por toda esta colección de tópicos, su envergadura a lo largo del tiempo y trascendencia en el ámbito de la ingeniería de software que se ha asumido el reto de migrar una aplicación monolítica web de streaming musical hacia una arquitectura de microservicios, teniendo en cuenta el salto sustancial que con lleva referente a la evolución de los sistemas de software y del modelo de despliegue que el aplicativo tendrá; por supuesto con el fin de gozar de todas las ventajas que nos brinda esta arquitectura de software.

Pero ¿cómo hacerlo posible? Varios catedráticos e investigadores han planteado varios procesos, estrategias y modelos que intentan guiar la migración de aplicaciones monolíticas a arquitectura de microservicios. Sin embargo, al no existir una guía exacta de migración de aplicaciones, se decidió tomar el modelo MOMMIV, ya que este modelo es el único que presenta una propuesta para la descomposición de arquitecturas monolíticas bajo la visión del Principio de Ocultación de Información. Además, MOMMIV provee lineamientos a través de un número de pasos que se encuentran embebidos en sus tres fases: a) Análisis, b) Diseño y c) Desarrollo y pruebas; orientando de mejor manera el proceso de migración[2].

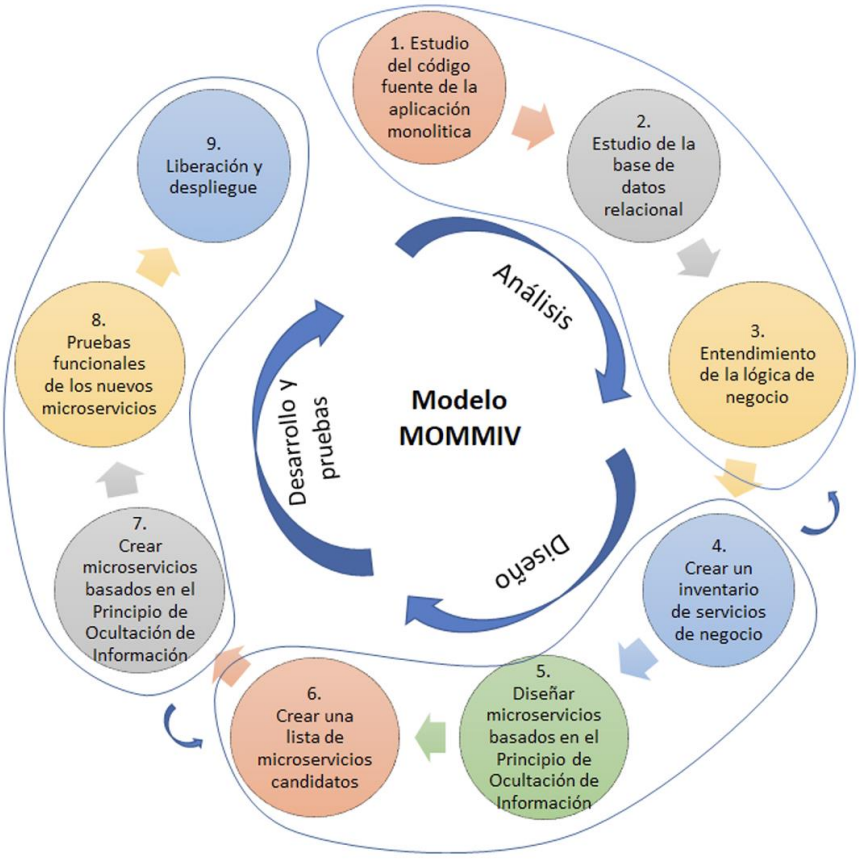


Figura 24 - Modelo MOMMIV para migrar aplicaciones monolíticas hacia microservicios.[2]

Newman, hace hincapié en ciertos desafíos involucrados durante una migración a microservicios y directamente con el diseño de arquitectura, entre ellos menciona: identificación de los límites de negocio, descomposición efectiva del aplicativo monolítico, gestión de la complejidad del sistema[25].

Por ello, una arquitectura adecuada es pieza clave para lograr tanto los requerimientos funcionales como no funcionales de un sistema; por otro lado, una arquitectura no adecuada puede ser catastrófica.

IV DESARROLLO DE MIGRACIÓN

4.1 Fase exploratoria

4.1.1 Alcance

Este proyecto muestra el proceso de migración de una aplicación web de streaming musical en arquitectura monolítica hacia una arquitectura de microservicios bajo el modelo MOMMIV (Modelo de Migración a Microservicios Versátil) donde el aplicativo ya cuenta con características y ventajas competitivas como: fácil escalabilidad, modularidad, tolerancia a fallos; que dan como resultado un producto con alta cohesión y bajo acoplamiento. Todo ello gracias a, la implementación de metodologías ágiles en el desarrollo, codificación en lenguajes y frameworks con soporte extendido, diseño de arquitectura de software mediante patrones de tendencia actual, versionamiento y despliegue haciendo uso de herramientas que permiten aplicar buenas prácticas de DevOps, además de considerar pruebas no funcionales para comprobar la factibilidad y rendimiento de la migración.

Hay que tener claro que el flujo de trabajo realizado “no pretende imponer un modelo de migración”, más bien intenta comprender lo que realmente conlleva el proceso del mismo.

4.1.2 Stakeholders

La siguiente tabla define a las partes interesadas en esta aplicación (también denominados stakeholders):

Partes interesadas (Stakeholders)		
ID	**Stakeholder**	**Descripción**
S1	**Profesor Msc. Victor Velepucha**	Catedrático de la Escuela Politécnica Nacional, que busca evidenciar el proceso de migración de una arquitectura monolítica hacia una arquitectura de microservicios con fines académicos, mediante su propuesta de modelo MOMMIV.

S2	**Profesora Phd. Pamela Flores**	Catedrática de la Escuela Politécnica Nacional, que busca evidenciar el proceso de migración de una arquitectura monolítica hacia una arquitectura de microservicios con fines académicos, mediante la propuesta de modelo MOMMIV.

Tabla 2 - Partes interesadas (Stakeholders)

4.1.3 Planificación del proyecto

En el plan de trabajo de titulación se definió que el proyecto tendría una duración de 800 horas a lo largo de 6 meses de trabajo, que corresponden a 400 horas de cada uno de autores del proyecto de titulación. El detalle de las horas se muestra en la siguiente tabla.

Nombre de tarea	Responsable	Duración
MIGRACIÓN DE UNA APLICACIÓN MONOLÍTICA WEB DE STREAMING MUSICAL HACIA UNA ARQUITECTURA DE MICROSERVICIOS.		**97 días**
Etapa de Desarrollo		**84 días**
Estudio del código fuente de la aplicación monolítica	Francisco Cevallos	32 horas
	Andrés Ruiz	32 horas
Estudio de la base de datos relacional	Francisco Cevallos	20 horas
	Andrés Ruiz	20 horas
Identificación de la lógica de negocio de la aplicación legada	Francisco Cevallos	8 horas
	Andrés Ruiz	8 horas
Creación de inventario de servicios de negocio	Francisco Cevallos	12 horas
	Andrés Ruiz	12 horas

Descomposición en microservicios de la lógica de negocio identificada	Francisco Cevallos	16 horas
	Andrés Ruiz	16 horas
Creación una lista de microservicios candidatos	Francisco Cevallos	12 horas
	Andrés Ruiz	12 horas
Codificación de la aplicación de arquitectura de microservicios bajo el marco de trabajo SCRUM	Francisco Cevallos	248 horas
	Andrés Ruiz	248 horas
Etapa de evaluación y pruebas		**6 días**
Pruebas de funcionalidad de la aplicación de arquitectura de microservicios	Francisco Cevallos	12 horas
	Andrés Ruiz	12 horas
Pruebas no funcionales de la aplicación de microservicios	Francisco Cevallos	12 horas
	Andrés Ruiz	12 horas
Etapa de despliegue		**7 días**
Despliegue de los microservicios en contenedores en la computación en la nube	Francisco Cevallos	28 horas
	Andrés Ruiz	28 horas

Tabla 3 - Planificación del desarrollo del proyecto definido por tareas, responsables y duración.

En el transcurso de desarrollo del software se mantuvo semanalmente reuniones con el director del presente proyecto de titulación, máster Victor Velepucha MSc. Dichas reuniones fueron útiles para el análisis, comprensión y descomposición del aplicativo monolítico. Además de la guía en la definición de la arquitectura de microservicios y el desarrollo de esta, bajo los criterios de aceptación de cada requerimiento funcional.

Para el desarrollo del proyecto que se efectuara con metodología Scrum, es importante, definir los roles de los involucrados, por lo tanto, la siguiente tabla detalla cada uno de ellos.

ROLES DE SCRUM	
Scrum Master	Victor Velepucha
Product Owner	Victor Velepucha
Scrum Team	Francisco Cevallos
	Franklin Ruíz

Tabla 4 - Roles de Scrum

4.2 Fase de inicialización

4.2.1 Herramientas utilizadas en la migración

A continuación, una breve descripción de las herramientas que serán utilizadas a lo largo del desarrollo de este proyecto.

Nombre	Descripción
Asp.Net Core 6.0	Es un marco multiplataforma de código abierto y de alto rendimiento que tiene como finalidad compilar aplicaciones modernas conectadas a Internet y habilitadas para la nube[50]. Usado para el desarrollo de los microservicios.
C#	Es un lenguaje de programación orientado a componentes, moderno, basado en objetos y con seguridad de tipos. La primera versión de este lenguaje salio en el 2002 y la ultima a la fecha de la publicación de esta tesis es la versión 11 publicada en noviembre de 2022. Para el desarrollo de la migración se utilizo la versión 10 publicada en noviembre de 2021, compatible con Asp.Net Core 6.0 [51]–[53].
TypeScript	TypeScript es un lenguaje de programación fuertemente tipado que se basa en JavaScript, es decir, es un superconjunto de JavaScript que se compila para limpiar la salida de JavaScript [54].

	Lenguaje utilizado para el desarrollo del servicio de correo electrónico.
Microsoft Visual Studio 2022	Es un IDE extensible para crear aplicaciones modernas para Windows, Android e iOS, además de aplicaciones web y servicios en la nube[55]. Este IDE hizo posible el desarrollo de los microservicios con Asp.Net Core 6.0.
Visual Studio Code	Visual Studio Code es un editor de código redefinido y optimizado para crear y depurar aplicaciones web y en la nube modernas[56]. Visual Studio Code fue el editor usado para la codificación del servicio de correo electrónico.
Microsoft Visual Studio 2017	Es un IDE con conjunto de herramientas de desarrollo para la generación de aplicaciones web ASP.NET, Servicios Web XML, aplicaciones de escritorio y aplicaciones móviles[57]. Este IDE permitió la visualización y análisis de la aplicación monolítica.
Git	Git es un sistema de control de versiones distribuido gratuito y de código abierto diseñado para manejar todo, desde proyectos pequeños hasta proyectos muy grandes, con rapidez y eficiencia[58]. Git sirvio para el versionamiento local del desarrollo del proyecto.
Azure DevOps	Azure DevOps admite una cultura colaborativa y un conjunto de procesos que reúnen a desarrolladores, administradores de proyectos y colaboradores para desarrollar software. Permite a las organizaciones crear y mejorar productos a un ritmo más rápido de lo que pueden con los enfoques tradicionales de desarrollo de software[59]. Azure DevOps proporcionó servicios de caracteristicas integradas para realizar los flujos de trabajo del proyecto.

Azure DevOps Repos	Es un conjunto de herramientas de control de versiones que se usa para la administración de código [60]. Repos permitío a realizar el seguimiento de los cambios realizados en el código en el transcurso del tiempo de desarrollo.
Azure DevOps Boards	Proporciona a los equipos de desarrollo de software las herramientas interactivas y personalizables que necesitan para administrar sus proyectos de software[61]. Boards permitío llevar gracias a sus herramientas el proceso de desarrollo del proyecto bajo el marco de trabajo Scrum.
Azure DevOps Pipelines	Es un servicio completo de integración continua (CI) y entrega continua (CD). Funciona con el proveedor de Git preferido y puede implementarse en la mayoría de los servicios en la nube principales[62]. Pipelines mediante su servicio de integración continua permitío automatizar la combinación y la prueba de código para detectar con facilidad errores.
Docker	Es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones. Docker permite separar las aplicaciones de la infraestructura para poder entregar software rápidamente[63]. Docker permitío dockenizar(creación de contenedores en base a una imagen generada previamente) los microservicios.
AKS	Simplifica la implementación de un clúster de Kubernetes administrado en Azure, al ser un servicio de Kubernetes hospedado, Azure controla tareas críticas como la supervisión del estado y el mantenimiento[64]. AKS permitío crear un clúster AKS para los microservicios desarrollados, además de cada uno de los nodos generados.
Microsoft SQL Server	Microsoft SQL Server es un sistema de gestión de bases de datos relacionales que admite una amplia variedad de aplicaciones de procesamiento de transacciones, inteligencia

	empresarial y análisis en entornos informáticos corporativos[65]. SQL Server es la tecnología de base de datos usado tanto en la aplicación monolítica, como en la aplicación migrada.
SQL Server Managment Studio	SQL Server Management Studio (SSMS) es un entorno integrado para acceder, configurar, gestionar, administrar cualquier infraestructura de SQL , y desarrollar todos los componentes de SQL Server, Azure SQL Database , Azure SQL Managed Instance , SQL Server en Azure VM y Azure Synapse Analytics[66]. SSMS permitío el análisis y administracion de base de datos SQL Server de la aplicación monolítica y la aplicación migrada.
Azure Service Bus	Es una plataforma en la nube de mensajería asincrónica que permite enviar datos entre sistemas desacoplados[67]. Azure Service Bus permitío la comuninación asincrónica entre alguno de los microservicios desarrollados.

Tabla 5 - Herramientas utilizadas en el proceso de migración

4.2.2 Fase de Análisis

Basado en el modelo MOMMIV, en esta fase se realizará un entendimiento de un aplicativo objetivo, revisando su código fuente y entendiendo su arquitectura, componentes y dependencias y una revisión de su base de datos relacional con el fin de adquirir un entendimiento de la funcionalidad de negocio de la aplicación monolítica[2].

- **Estudio del código fuente de la aplicación monolítica.**

Para realizar un estudio sistemático del código fuente del aplicativo se ha definido un conjunto de ítems que permitirán comprender bajo que perspectivas fue desarrollado el aplicativo monolítico.

  - **Revisión y análisis de componentes de la solución del aplicativo monolítico.**

El aplicativo fue construido bajo una solución llamada “SlnAppBytesMusic”, que contiene los proyectos que se muestran en la figura 25

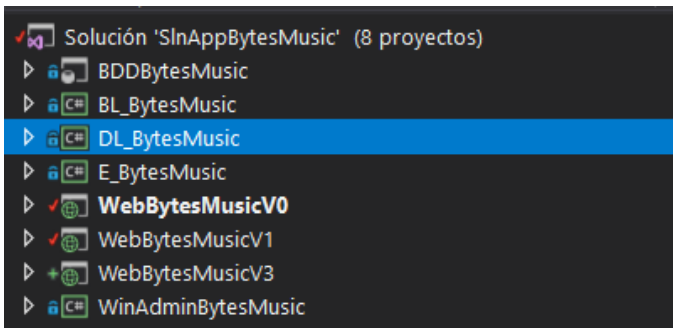


Figura 25 - Proyectos de la solución “SlnAppBytesMusic”

En la solución existen 3 tipos de proyectos según su propósito.

- Los que corresponden al Back-end del aplicativo:
  - “BL_BytesMusic”
  - “DL_BytesMusic”
  - “E_BytesMusic”
- Los que corresponden al Front-end del aplicativo:
  - “WebBytesMusicV0”
  - “WebBytesMusicV1”
- Los que corresponden a almacenar los scripts de base de datos:
  - “BDDBytesMusic”

El conjunto de proyectos de la solución requiere individualmente una inspección de dependencias entre proyectos, asimismo revisión del contenido de los archivos C#, para entender la lógica de desarrollo y funcionamiento del aplicativo.

Por ello, primero se hizo uso del “Administrador de referencias – Proyectos” en los proyectos correspondientes al Back-end y Front-end.

- “BL_BytesMusic”, tiene como dependencias a los proyectos “DL_BytesMusic” y “E_BytesMusic”

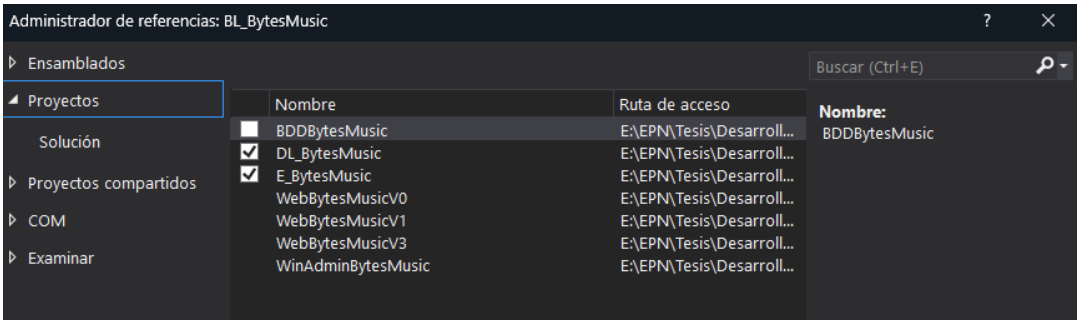


Figura 26 - Administrador de referencias, sección Proyectos: BL_BytesMusic.

- "DL_BytesMusic", tiene como dependencias al proyecto "E_BytesMusic".

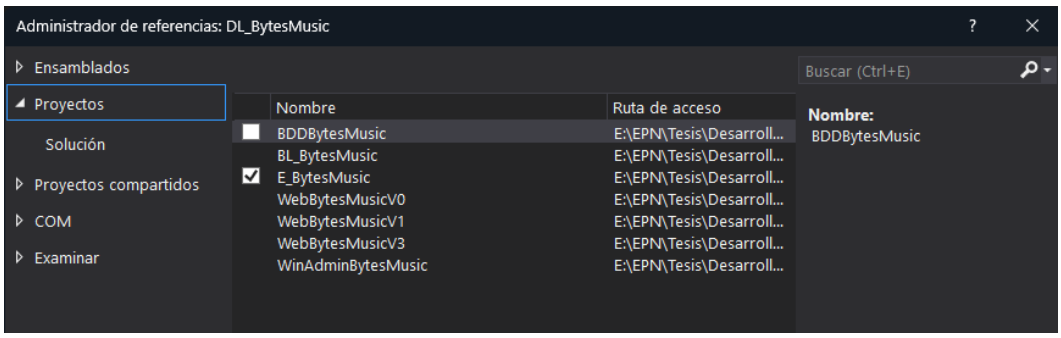


Figura 27 - Administrador de referencias, sección Proyectos: DL_BytesMusic.

- "E_BytesMusic", no tiene dependencias a ningún proyecto.

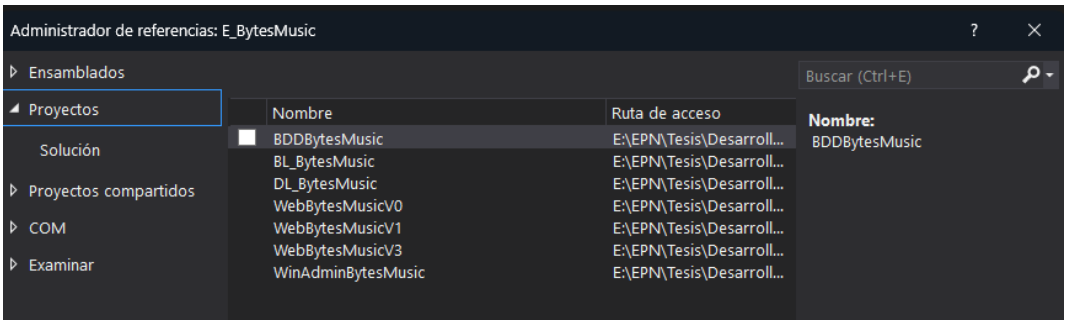


Figura 28 - Administrador de referencias, sección Proyectos: E_BytesMusic.

- "WebBytesMusicV1", tiene dependencias a los proyectos "DL_BytesMusic" y "E_BytesMusic".

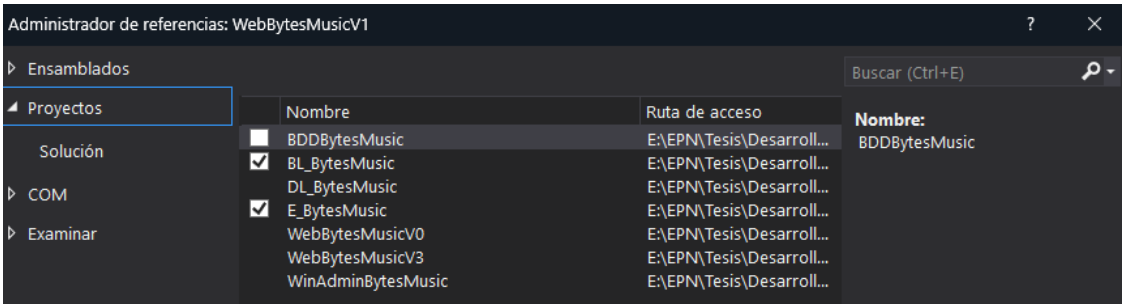


Figura 29 - Administrador de referencias, sección Proyectos: WebBytesMusicV1.

Al analizar las referencias entre los proyectos de la solución, es esencial hacer visual lo encontrado durante la inspección, por tanto, a continuación, la figura 30.

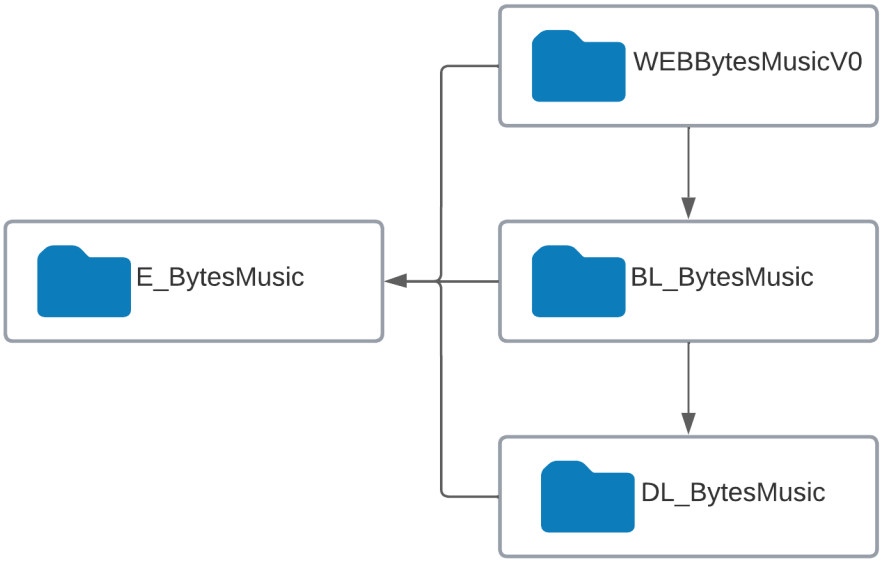


Figura 30 - Referencias entre proyectos de la solución del aplicativo monolítico.

Una vez que se entendió las referencias de proyectos en la solución, se inspeccionará el contenido de los archivos C# abordando progresivamente estos, desde los proyectos no dependientes hasta los proyectos con más dependencias. Consecuente a ello se obtuvo las siguientes deducciones:

- En “E_BytesMusic” se encontró clases C# donde cada una de ellas representaba un modelo de objetos de la realidad.

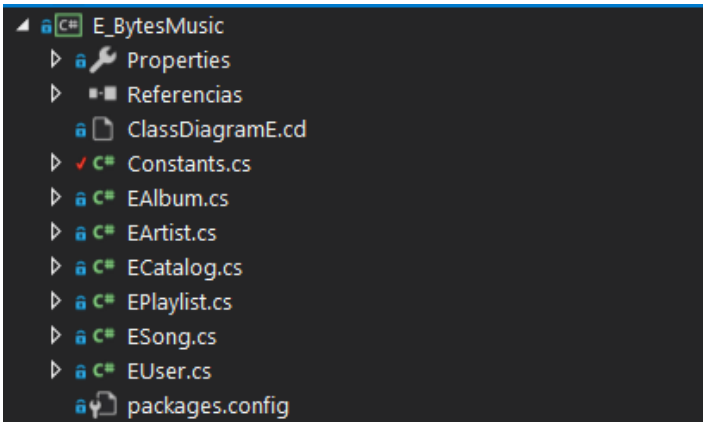


Figura 31 - Scripts del proyecto "E_BytesMusic"

- En "DL_BytesMusic", los scripts están encargados de consultar y regresar datos desde una fuente de información (lógica de base de datos), en este caso, hace uso de la base de datos "BDD_BYTESMUSIC"; es importante mencionar que cada clase hace uso de las entidades antes revisadas según fueron nombradas.

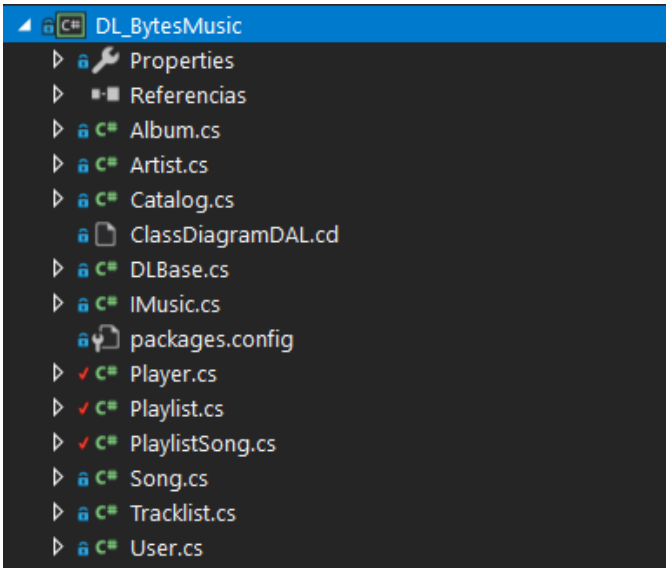


Figura 32 - Scripts del proyecto "DL_BytesMusic"

- En "BL_BytesMusic", los scripts validan y efectúan las reglas del negocio (lógica del negocio) de cada una de las entidades a las que hace alusión. El desglose y detalle del análisis de este proyecto se encuentra en el **Anexo 1.**

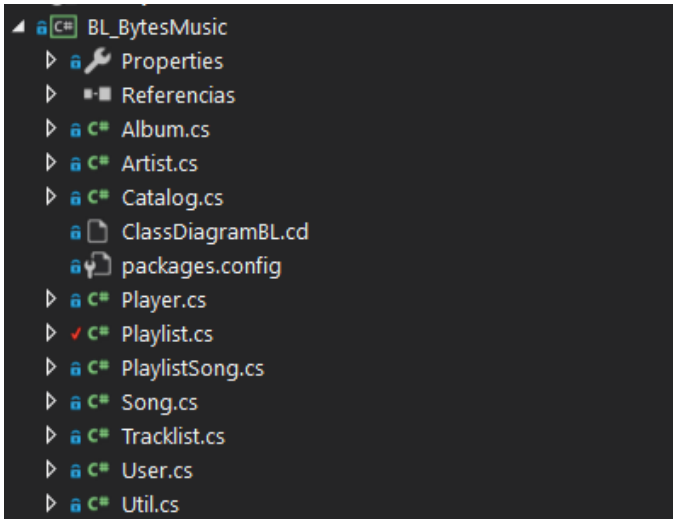


Figura 33 - Scripts del proyecto “BL_BytesMusic”

- En “WebBytesMusic”, posee archivos tipo ASPX que son usadas para el desarrollo de interfaz de usuario explícitamente de páginas web. Estos archivos permitirán mostrar e interactuar directamente con el usuario para la muestra y recolección de datos.

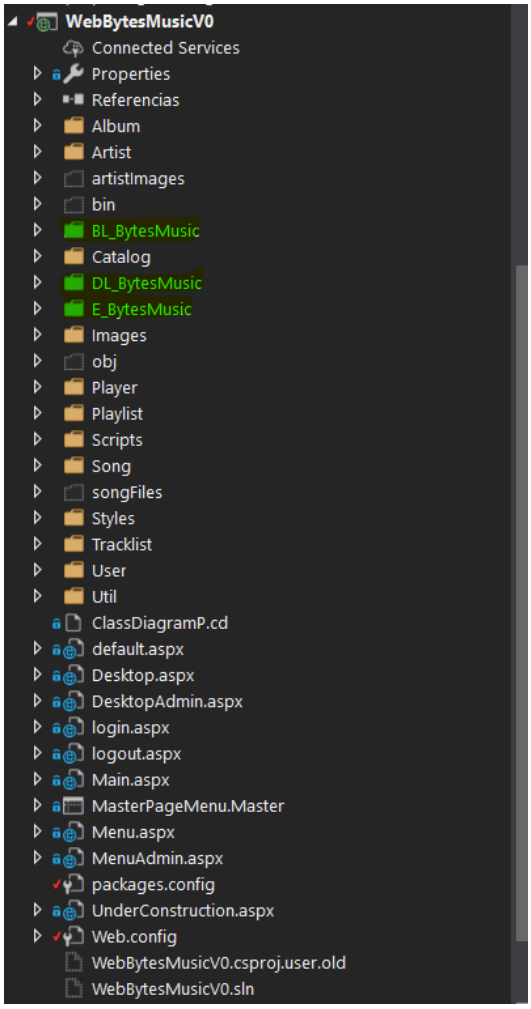


Figura 34 - Proyecto de Front-end "WebBytesMusic"

○ Diagrama de arquitectura de la aplicación monolítica

Posterior a una revisión minuciosa del código fuente y análisis de referencias de la solución entre proyectos. Se entiende claramente que es una aplicación de arquitectura monolítica de proceso único. Además, se constató que la aplicación monolítica asemeja una arquitectura software de n-capas, justificado bajo las funciones y responsabilidades que tenía cada proyecto para el aplicativo, evidenciadas en el punto anterior. Entonces se puede decir que:

- “E_BytesMusic”, hace referencia a la Capa de Entidad.
- “DL_BytesMusic”, hace referencia a la Capa de Datos.
- “BL_BytesMusic”, hace referencia a la Capa de Negocio.
- “WebBytesMusic” hace referencia a la Capa de Interfaz de Usuario.

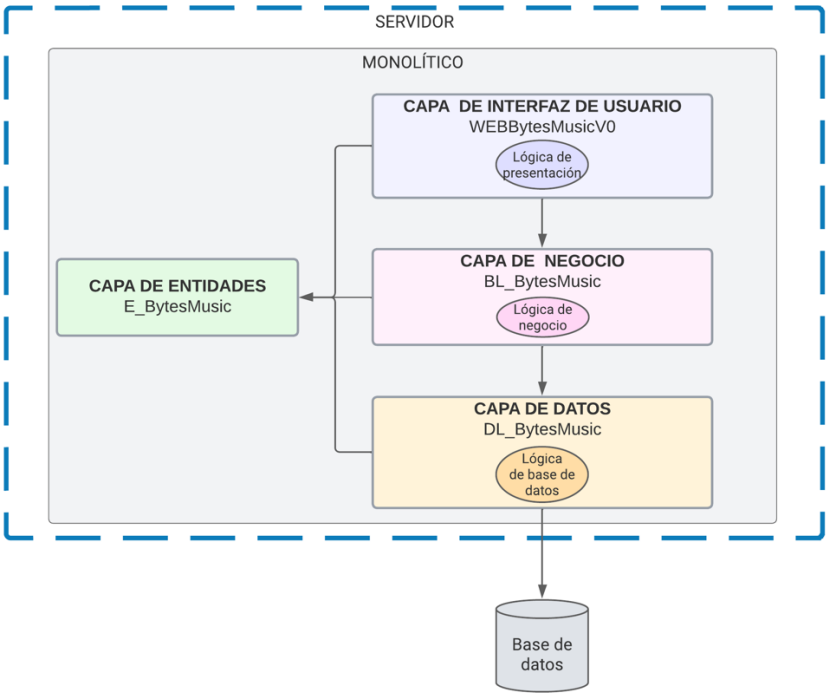


Figura 35 - Arquitectura de la aplicación monolítica.

- **Estudio de la base de datos relacional**

Los scripts de la base de datos otorgada correspondían a un sistema de gestión de base de datos Microsoft SQL Server, donde gracias a la herramienta Microsoft SQL Server Management Studio se pudo obtener el diagrama, que se muestra en la siguiente figura.

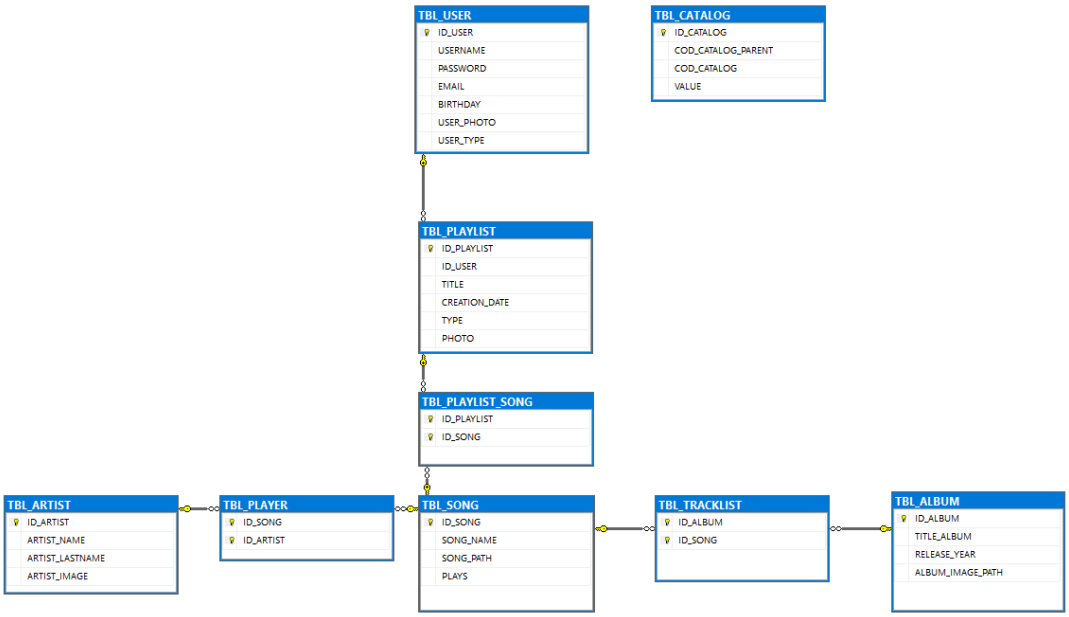


Figura 36 - Modelo de base de datos de la aplicación monolítica.

- **TBL_CATALOG** es una tabla que almacena los registros de catálogos. Sirve para manejar la distinción lógica de tipos de usuario y tipos de listas de reproducción. TBL_CATALOG posee los siguientes campos.
    - ID_CATALOG, es un campo de tipo entero auto incrementable, no nulo y es la clave primaria de la tabla.
    - COD_CATALOG_PARENT, es un campo de tipo varchar con un tamaño de 8 caracteres.
    - COD_CATALOG, es un campo de tipo varchar con un tamaño de 8 caracteres.
    - VALUE, es un campo de tipo varchar con un tamaño de 255 caracteres, no nulo.
- **TBL_SONG** es una tabla que almacena los registros de canciones. TBL_SONG posee los siguientes campos.
    - ID_SONG, es un campo de tipo entero auto incrementable, no nulo y es la clave primaria de la tabla.
    - SONG_NAME, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.

- SONG _PATH, es un campo de tipo varchar con un tamaño de 255 caracteres, no nulo.
- PLAYS, es un campo de tipo entero.

- **TBL_ARTIST** es una tabla que almacena los registros de artistas. TBL_ARTIST posee los siguientes campos.
  - ID_ ARTIST, es un campo de tipo entero auto incrementable, no nulo y es la clave primaria de la tabla.
  - ARTIST_NAME, es un campo de tipo varchar con un tamaño de 100 caracteres, no nulo.
  - ARTIST_LASTNAME, es un campo de tipo varchar con un tamaño de 100 caracteres, no nulo.
  - ARTIST_IMAGE, es un campo de tipo varchar con un tamaño de 255 caracteres.
- **TBL_PLAYER** es una tabla de unión entre las tablas TBL_ ARTIST y TBL_SONG para representar para una relación de muchos a muchos. TBL_PLAYER posee los siguientes campos.
  - ID_ARTIST, es un campo de tipo entero, no nulo y es la clave foránea de la tabla TBL_ARTIST.
  - ID_ SONG, es un campo de tipo entero, no nulo y es la clave foránea de la tabla TBL_SONG.
- **TBL_ALBUM** es una tabla que almacena los registros de álbumes. TBL_ALBUM posee los siguientes campos.
  - ID_ ALBUM, es un campo de tipo entero auto incrementable y es la clave primaria de la tabla.
  - TITLE_ ALBUM, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.
  - RELEASE_YEAR, es un campo de tipo entero.
  - ALBUM_IMAGE_PATH, es un campo de tipo varchar con un tamaño de 255 caracteres.

- **TBL_TRACKLIST** es una tabla de unión entre las tablas TBL_ALBUM y TBL_SONG para representar una relación de muchos a muchos, posee los siguientes campos:
    - ID_ALBUM, es un campo de tipo entero, no nulo y es una clave foránea perteneciente a la tabla TBL_ALBUM.
    - ID_SONG, es un campo de tipo entero, no nulo y es una clave foránea perteneciente a la tabla TBL_SONG.
- **TBL_PLAYLIST** es una tabla que almacena los registros de listas de reproducción. TBL_PLAYLIST posee los siguientes campos.
    - ID_PLAYLIST, es un campo de tipo entero auto incrementable, no nulo y es la clave primaria de la tabla.
    - ID_USER, es un campo de tipo entero, no nulo y es una clave foránea perteneciente a la tabla TBL_ALBUM.
    - TITLE, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.
    - CREATION_DATE, es un campo de tipo datetime, no nulo.
    - TYPE, es un campo de tipo entero, no nulo.
    - PHOTO, es un campo de tipo varchar con un tamaño de 255 caracteres.
- **TBL_PLAYLIST_SONG** es una tabla de unión entre las tablas TBL_ PLAYLIST y TBL_SONG para representar una relación de muchos a muchos, posee los siguientes campos:
    - ID_PLAYLIST, es un campo de tipo entero, no nulo y es una clave foránea perteneciente a la tabla TBL_PLAYLIST.
    - ID_SONG, es un campo de tipo entero, no nulo y es una clave foránea perteneciente a la tabla TBL_SONG.
- **TBL_USER** es una tabla que almacena los registros de usuario, posee los siguientes campos:
    - ID_USER, es un campo de tipo entero, no nulo y es la clave primaria de la tabla.
    - USERNAME, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.

- PASSWORD, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.
- EMAIL, es un campo de tipo varchar con un tamaño de 50 caracteres, no nulo.
- BIRTHDAY, es un campo de tipo datetime.
- USER_PHOTO, es un campo de tipo varchar con un tamaño de 255 caracteres.
- USER_TYPE, es un campo de tipo entero, no nulo.

- **Entendimiento de la lógica de negocio**

El entender la lógica de negocio prima de conocer el principio u objetivo de la aplicación monolítica que es “ofrecer música bajo demanda al cliente, permitir a los diferentes compositores o músicos subir sus canciones”. El entendimiento de lógica de negocio se logró gracias a la revisión de código fuente de la Capa de Negocio en el proyecto “BL_BytesMusic” que se le puede revisar a detalle en el **Anexo 1**. A partir de ello y haciendo uso del aplicativo se describirá las funcionalidades de negocio, obteniendo como resultado el siguiente listado:

1. Registro de usuarios: permite a los usuarios crear una cuenta en la plataforma, mediante una dirección de correo electrónico.
2. Búsqueda de música: permite a los usuarios buscar música por título, artista o álbum.
3. Reproducción de música: permite a los usuarios escuchar música en streaming desde la plataforma, con la opción de pausar, retroceder, adelantar y ajustar el volumen.
4. Listas de reproducción: permite a los usuarios crear y guardar listas de reproducción personalizadas para organizar y acceder fácilmente a sus canciones favoritas.
5. Gestión de contenido: la capacidad de cargar, almacenar y administrar contenido musical, incluyendo la posibilidad de agregar, editar o eliminar canciones, álbumes, artistas y listas de reproducción.

6. Gestión de usuarios: la capacidad de administrar los usuarios de la plataforma, incluyendo la posibilidad de agregar, editar o eliminar cuentas de usuario, así como la capacidad de controlar los permisos y el acceso de los usuarios a diferentes características y contenido.

4.2.3 Fase de Diseño

Basado en el modelo MOMMIV, en esta fase se creará un inventario de la funcionalidad de negocio identificada de la fase de análisis. Con este inventario se procede a diseñar los nuevos microservicios basándonos en los lineamientos del Principio de Ocultación de Información. Con los resultados de los diseños de microservicios se levanta un listado de microservicios candidatos tomando en cuenta que exista trazabilidad con la funcionalidad de negocio[2].

- **Inventario de servicios de negocio**

Dentro de la revisión del código fuente se encontró una distinción en la funcionalidad basado en el tipo de usuario, es así como se procedió a realizar la tabla 6 que detalla el inventario de la funcionalidad de negocio para el tipo de usuario “USER” y “ADMIN”. Se puede decir que “USER” es un usuario cliente, mientras que “ADMIN” es un usuario administrador.

  - **Funcionalidad del negocio**

Inventario de la funcionalidad de negocio	
Código	**Detalle**
FN001	**Validación de datos** Los datos ingresados por el usuario son validados para cumplir con los requisitos establecidos para cada campo. Cualquier dato incorrecto debe es rechazado y se informa al usuario sobre los errores.

FN002	**Privacidad y seguridad** La plataforma asegura la privacidad y seguridad de los datos del usuario, protegiendo la información personal y las contraseñas de acceso.
FN003	**Acceso, autenticación y validación de usuarios** La plataforma garantiza que solo los usuarios autorizados accedan a las funciones y datos de la aplicación.
FN004	**Gestión de usuarios** La plataforma permite la gestión de usuarios, incluyendo la creación, modificación y eliminación de cuentas. Estas operaciones son seguras y confiables.
FN005	**Verificación de duplicados** La plataforma verifica la existencia de usuarios con nombres de usuario o correos electrónicos duplicados antes de permitir la creación de una nueva cuenta. Esto ayuda a evitar la creación de cuentas duplicadas o fraudulentas.
FN006	**Actualización de información** La plataforma permite a los usuarios actualizar su información personal, como el correo electrónico, la dirección. Estas operaciones deben ser seguras y confiables.
FN007	**Gestión de tracklist** La plataforma permite la gestión de tracklists, incluyendo la creación, modificación y eliminación de listas de canciones de un álbum especifico. Estas operaciones son seguras y confiables
FN008	**Verificación de duplicados** La plataforma verifica la existencia de canciones en un álbum específico antes de agregarlas para evitar duplicados.

FN009	**Eliminación de canciones** La plataforma permite a los usuarios eliminar una lista de canciones de un álbum especifico.
FN010	**Integración de música** La plataforma permite la integración de música en los álbumes. La plataforma asegurarse de que las canciones estén disponibles y se puedan reproducir en la plataforma.
FN011	**Lectura de datos** La plataforma debe permitir a los usuarios leer los datos de un álbum específico para que puedan ver las canciones que la componen.
FN012	**Gestión de canciones** La plataforma permite a los usuarios agregar, buscar, leer, actualizar y eliminar canciones de su catálogo musical. Estas operaciones son seguras y confiables.
FN013	**Actualización de información** La plataforma permite a los usuarios actualizar la información de las canciones, incluyendo el título, artista, álbum, entre otros. Los cambios son reflejados de manera precisa y oportuna en el catálogo de canciones.
FN014	**Verificación de duplicados** La plataforma verifica la existencia de canciones duplicadas en su catálogo para evitar repeticiones innecesarias.
FN015	**Gestión de canciones en playlist** La plataforma permite a los usuarios agregar, buscar, leer, actualizar y eliminar canciones de sus listas de reproducción. Estas operaciones deben ser seguras y confiables.

FN016	**Verificación de duplicados** La plataforma verifica la existencia de canciones duplicadas en las listas de reproducción de los usuarios para evitar repeticiones innecesarias
FN017	**Gestión de playlist** La plataforma permite a los usuarios crear, buscar, leer, actualizar y eliminar sus listas de reproducción. Estas operaciones son seguras y confiables.
FN018	**Verificación de duplicados** La plataforma verifica la existencia de listas de reproducción duplicadas en el perfil de los usuarios para evitar repeticiones innecesarias.
FN019	**Personalización de las listas de reproducción** La plataforma permite a los usuarios personalizar sus listas de reproducción con el título, descripción, imagen y otros detalles que deseen.
FN020	**Validación de campos requeridos** La plataforma se asegura de que se ingresen todos los campos requeridos al crear o actualizar una lista de reproducción, como el título de la lista de reproducción. Si faltan campos requeridos, se debe notificar al usuario
FN021	**Gestión de artistas en player** La plataforma permite a los usuarios agregar, eliminar canciones de un artista específico.
FN022	**Validación de canciones duplicadas en player** La plataforma verifica la existencia de canciones duplicadas de un artista para evitar repeticiones innecesarias.
FN023	**Gestión de artistas** La plataforma permite a los usuarios agregar, actualizar y eliminar artistas de la base de datos.

FN024	**Validación de campos requeridos** La plataforma se asegura de que se ingresen todos los campos requeridos al crear o actualizar un artista, como el primer nombre y el apellido. Si faltan campos requeridos, se debe notificar al usuario.
FN025	**Validación de artistas duplicados** La plataforma verifica la existencia de artistas duplicados en la base de datos para evitar repeticiones innecesarias.
FN026	**Gestión de álbumes** La plataforma permite a los usuarios agregar, actualizar, leer y eliminar álbumes de la base de datos.
FN027	**Validación de campos requeridos** La plataforma se asegura de que se ingresen todos los campos requeridos al crear o actualizar un álbum, como el título y el artista. Si faltan campos requeridos, se debe notificar al usuario.
FN028	**Validación de álbumes duplicados** La plataforma verifica la existencia de álbumes duplicados en la base de datos para evitar repeticiones innecesarias.
FN029	**Gestión de Catálogo** La plataforma permite crear, leer, actualizar y eliminar catálogos. También verifica la existencia de duplicados y valida la información ingresada.
FN030	**Validación de catálogos** La plataforma verifica que los campos de código de catálogo, código de catálogo padre y valor no estén vacíos al momento de guardar un nuevo catálogo.

FN031	**Obtención de tipos de usuario** La plataforma obtiene los tipos de usuario registrados en la base de datos para su uso en la interfaz de usuario.
FN032	**Obtención de tipos de playlist** La plataforma obtiene los tipos de playlist registrados en la base de datos para su uso en la interfaz de usuario.

Tabla 6 - Inventario de la funcionalidad de negocio.

- Diseño de microservicios basados en el Principio de Ocultación de Información

El Principio de Ocultación de Información fue una teoría propuesta en 1972 en el artículo “On the Criteria to Be Used in Decomposing Systems into Modules” por David Parnas. Parnas sostiene que el diseño de sistemas modulares debe enfocarse en la identificación y definición de las interfaces entre módulos, con el objetivo de minimizar la cantidad de información que se comparte entre ellos[68]. Esto reduce la complejidad del sistema, aumentar la modularidad y seguridad, y facilita su mantenimiento y evolución otorgando los siguientes beneficios: a) Tiempo de desarrollo cortos, b) Flexibilidad en el desarrollo de un sistema, c) Comprensibilidad de un sistema[2], [68].

Los tiempos de desarrollos cortos, se logra cuando es posible separar una aplicación en módulos que permitan realizar el trabajo por grupos con la menor comunicación posible. La flexibilidad consiste en que se pueden realizar cambios drásticos o de gran impacto en módulos sin que esto afecte a otros módulos. La comprensibilidad se refiere a que se puede entender un módulo a la vez, sin tener que entender todo el sistema. Esto puede beneficiar a que un equipo desarrollo pueda estar especializado para dar mantenimiento y añadir nueva funcionalidad en un módulo específico[2].

También Parnas, propone los siguientes lineamientos para descomponer módulos. a) Cada módulo debe estar caracterizado por su conocimiento de una decisión de diseño, b) Ocultar las decisiones de diseño de otros módulos[68].

Bajo esta perspectiva y con el propósito de cumplir estos lineamientos definiremos una tabla que nos permita separar las funcionalidades de negocio por conjuntos, con las tablas de base de datos a las cuales responden dichas funcionalidades, a fin de descomponer en microservicios de la lógica de negocio identificada.

Tabla de base de datos	Funcionalidades de negocio
TBL_SONG	FN012, FN013, FN014
TBL_ARTIST	FN023, FN024, FN025
TBL_PLAYER	FN021, FN022
TBL_ALBUM	FN026, FN027, FN028
TBL_TRACKLIST	FN007, FN008, FN009, FN010, FN011
TBL_PLAYLIST_SONG	FN015, FN016
TBL_PLAYLIST	FN017, FN018, FN019, FN020
TBL_CATALOG	FN029, FN030, FN031, FN032, FN033.
TBL_USER	FN002, FN003, FN004, FN005, FN006

Tabla 7 - Funcionalidades de negocio clasificadas por su impacto en las tablas de base de datos.

Para identificar las funcionalidades que deben ser ofrecidas por el microservicio, así como los límites de su alcance, es imperativo analizar las dependencias fuertes existentes entre las tablas de base de datos.

Bajo esta premisa es necesario tratar los posibles componentes que se generarían innecesariamente, provocando comunicaciones sobrecargadas entre los microservicios y la complejidad del sistema aumentaría significativamente. En el

caso de las tablas que tienen como objetivo ser tablas de unión para representar relaciones de muchos a muchos en el modelo de base de datos, como son: TBL_PLAYER, TBL_TRACKLIST, TBL_PLAYLIST_SONG; la mejor opción es descartarlos como un componente autónomo por su dependencia de comunicación entre TBL_SONG con las tablas TBL_ARTIST, TBL_ALBUM, TBL_PLAYLIST respectivamente.

Por ello, las tablas que por su naturaleza representan solo una abstracción en una relación de base de datos van a ir junto con las tablas que manejan parte de su funcionalidad principal. TBL_ARTIST y TBL_PLAYER conformarían el desarrollo de un componente autónomo, al igual que TBL_ALBUM y TBL_TRACKLIST, y también TBL_PLAYLIST y TBL_PLAYLIST_SONG.

A pesar de que el Principio de Ocultación (POI) de información nos provee lineamientos base para el diseño de los microservicios, es crucial definir patrones de diseño que se ajusten a dichos lineamientos. Por dicha razón, se realizó una búsqueda de patrones arquitectónicos de vanguardia en la construcción de sistemas basados en microservicios, y que sean compatibles con POI; se obtuvo como resultado los patrones EDA (Event Driven Design), DDD (Domain Driven Design) y Arquitectura Limpia. A continuación, se explicará el por qué estos patrones se ajustan a principio.

En EDA, los eventos son la forma principal en que los servicios interactúan entre sí. Cada servicio publica eventos a un bus de eventos, y otros servicios pueden suscribirse a esos eventos para reaccionar a ellos. Esto permite una separación clara de las responsabilidades y evita la necesidad de que un servicio conozca detalles internos sobre otro servicio.

En DDD, los microservicios se organizan en torno a dominios específicos, y cada servicio es responsable de su propio dominio. Esto significa que los detalles internos del servicio se mantienen ocultos de otros servicios, lo que facilita la evolución independiente de los servicios.

En Arquitectura Limpia, los servicios se dividen en capas, y cada capa se comunica con la capa adyacente a través de una interfaz clara y definida. Esto significa que

los detalles internos de cada capa se mantienen ocultos de las capas adyacentes, lo que facilita el cambio y la evolución independiente de cada capa.

- Listado de microservicios candidatos

Entonces, como resultado de la minuciosidad con la que se ha identificado los límites de servicio y desafíos que conlleva se puede decir que se realizó una descomposición efectiva del aplicativo monolítico. A continuación, la tabla 8 indica el nombre del servicio y las tablas de base de datos de acuerdo con su finalidad.

Tabla de la base de datos	Nombre del servicio
TBL_SONG	Song
TBL_ARTIST TBL_PLAYER	Artist
TBL_ALBUM TBL_TRACKLIST	Album
TBL_PLAYLIST TBL_PLAYLIST_SONG	Playlist
TBL_CATALOG	Catalog
TBL_USER	User

Tabla 8 - Clasificación de las tablas de base de datos según el servicio o función

A continuación, el listado de microservicios candidatos obtenido en base a la información recaba y explicada anteriormente.

MICROSERVICIOS
Song
Artist
Album
Playlist
Catalog
User

Tabla 9 - Microservicios candidatos

4.2.4 Configuración del proyecto

- **Historias Épicas de Usuario**

Para el desarrollo de este proyecto se realizaron reuniones con el Msc. Victor Velepucha (Product Owner), donde basándonos en el "Modelo de Migración a Microservicios Versátil" específicamente en sus primeras dos fases Análisis y Diseño, se obtuvo como resultado una "lista de microservicios candidatos" los cuales sirvieron de guía para definir las historias de usuario. Adicionalmente, para abarcar las ventajas más importantes de la arquitectura basada en microservicios, se pidió la implementación de comunicación asíncrona entre microservicios y la agregación de nuevas funcionalidades al sistema. Como resultado de ello se obtuvo las siguientes historias de usuario.

HISTORIAS DE USUARIO		
Código	**Nombre**	**Descripción**
HU01	Microservicio "Usuario"	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio "User" bajo

		patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio identificadas referente a los usuarios.
HU02	Microservicio “Song”	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio “Song” bajo patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio referente a las canciones.
HU03	Microservicio “Album”	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio “Album” bajo patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio referente a los álbumes.
HU04	Microservicio “Artist”	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio “Artist” bajo patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio referente a los artistas.
HU05	Microservicio “Playlist”	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio “Playlist” bajo patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio referente a las listas de reproducción.

HU06	Microservicio “Catalog”	Como desarrolladores en un proceso de migración hacia una arquitectura de microservicios, necesitamos la creación del microservicio “Catalog” bajo patrones arquitectónicos de diseño que se ajusten al principio POI, el cuál debe contener todas las funcionalidades de negocio referente a los catálogos.
HU07	Comunicación asincrónica en el sistema	Como desarrolladores, necesitaremos diseñar un modelo abstracto para incorporar una arquitectura basada en eventos en nuestro sistema para incorporar la comunicación asincrónica entre ciertos servicios.
HU08	Servicio de “Email”	Como dueño del producto, necesito mostrar la flexibilidad de incorporación de nuevas funcionalidades a una arquitectura de microservicios (escalabilidad) y la independencia de lenguajes. Por ello quiero que el usuario pueda recibir un correo electrónico de bienvenida después de registrarse en la plataforma. Para lograr esto, se necesita desarrollar un servicio de email que pueda enviar correos electrónicos personalizados a cada usuario recién registrado.

Tabla 10 - Historias de usuario

- **Product Backlog**

Tras definir las Historias de usuario se procedió a separarlas en tareas más pequeñas y, a partir de ellas, se obtuvo el Product Backlog (Lista de pendientes del producto). El Product Backlog está conformada por las tareas a realizarse, junto con una estimación de la complejidad para completar cada tarea y su prioridad para cumplir los objetivos del negocio.

Para definir la complejidad de cada historia de usuario se utilizó “Planning Poker”, que es una técnica utilizada para calcular el esfuerzo de las tareas de gestión de proyectos de desarrollo de software. “Planning Poker” manifiesta que para medir la cantidad de puntos de la historia para las tareas relevantes el equipo de desarrollo necesita calcular el esfuerzo que tomaría completar cada tarea en una escala exponencial, conociendo, por ejemplo, que si una tarea es el doble de compleja tomaría más del doble del tiempo en ser completada[69]. Inicialmente, se escogió la tarea que tomaría el menor esfuerzo para ser usada como guía base, en este caso HU07-T41 (ver Tabla 13). La complejidad asignada a las demás tareas es una representación del tamaño de cada historia de usuario en comparación con la guía base. La valoración de la complejidad de las historias de usuario, que fueron definidas por el equipo de desarrollo, está descrita en la tabla a continuación.

VALORACIÓN DE LA COMPLEJIDAD	
Valoración	**Complejidad**
1	Muy Baja
3	Baja
5	Media
8	Alta
10	Muy alta

Tabla 11 - Valoración de la complejidad

Para la estimación por prioridad se usó la técnica “Dot Votting”. Durante la técnica Dot Voting, hay que asegurarse de todos los miembros entiendan claramente cada elemento. Después cada miembro del equipo recibe la misma cantidad de puntos que se utilizarán para priorizar los elementos del Product Backlog, en este caso cada miembro recibió 240 puntos[70], [71].

Al no existir una regla para asignar un número a cada participante se determinó dicho valor suponiendo el caso hipotético que a algún participante le parezca de máxima prioridad cada una de las 48 tareas, es decir, si 5 es el número de puntos que representará la prioridad “muy alta” y hay 48 tareas, # de puntos asignados será igual a 5 *48.

En la votación, a los miembros se les pide que los distribuyan sus puntos en las tareas que consideran más importantes. Los participantes pueden poner todos sus

puntos en un solo elemento o distribuirlos entre varios elementos. Una vez que se han distribuido todos los puntos, se cuentan y se suman los puntos para cada tarea. Las tareas con la mayor cantidad de puntos definen la prioridad según el equipo, es decir obtendrán la valoración más alta usando los valores mostrados en la siguiente tabla [70], [71].

VALORACIÓN DE LA PRIORIDAD	
Valoración	**Prioridad del negocio**
1	Muy Baja
2	Baja
3	Media
4	Alta
5	Muy Alta

Tabla 12 - Valoración de prioridad

Para determinar el orden y conjunto de ejecución de tareas se tomó en cuenta el enfoque de funcionalidad que tenía para el sistema, además se analizó las tareas de mayor complejidad junto con las que tienen mayor prioridad para el negocio, así se fue definiendo el Product Backlog. Dicho “Product Backlog” se puede observar en la siguiente tabla.

PRODUCT BACKLOG				
Historia de usuario	**Código**	**Nombre**	**Complejidad**	**Prioridad**
HU01	T01	Diseñar e implementar la capa de Dominio del microservicio “User” utilizando DDD.	3	3
HU01	T02	Diseñar e implementar la capa de Infraestructura del microservicio “User”	8	3
HU01	T03	Diseñar e implementar la capa de Aplicación del microservicio “User”	5	3
HU01	T04	Implementar la capa de Presentación del microservicio “User”	5	3
HU01	T05	Integrar el microservicio "User" con otros microservicios	8	3

HU01	T06	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "User" la mediante la API por solicitudes HTTP.	5	3
HU02	T07	Diseñar e implementar la capa de Dominio del microservicio “Song” utilizando DDD.	3	4
HU02	T08	Diseñar e implementar la capa de Infraestructura del microservicio “Song”	8	4
HU02	T09	Diseñar e implementar la capa de Aplicación del microservicio “Song”	5	4
HU02	T10	Implementar la capa de Presentación del microservicio “Song”	5	4
HU02	T11	Integrar el microservicio " Song " con otros microservicios	8	4
HU02	T12	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "Song" la mediante la API por solicitudes HTTP.	5	4
HU03	T13	Diseñar e implementar la capa de Dominio del microservicio “Album” utilizando DDD.	3	3
HU03	T14	Diseñar e implementar la capa de Infraestructura del microservicio “Album”	8	3
HU03	T15	Diseñar e implementar la capa de Aplicación del microservicio “Album”	5	3
HU03	T16	Implementar la capa de Presentación del microservicio “Album”	5	3
HU03	T17	Integrar el microservicio " Album " con otros microservicios	8	3
HU03	T18	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "Album" la mediante la API por solicitudes HTTP.	5	3

HU04	T19	Diseñar e implementar la capa de Dominio del microservicio “Artist” utilizando DDD.	3	3
HU04	T20	Diseñar e implementar la capa de Infraestructura del microservicio “Artist”	8	3
HU04	T21	Diseñar e implementar la capa de Aplicación del microservicio “Artist”	5	3
HU04	T22	Implementar la capa de Presentación del microservicio “Artist”	5	3
HU04	T23	Integrar el microservicio "Artist " con otros microservicios	8	3
HU04	T24	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "Artist" la mediante la API por solicitudes HTTP.	5	3
HU05	T25	Diseñar e implementar la capa de Dominio del microservicio “Playlist” utilizando DDD.	3	3
HU05	T26	Diseñar e implementar la capa de Infraestructura del microservicio “Playlist”	8	3
HU05	T27	Diseñar e implementar la capa de Aplicación del microservicio “Playlist”	5	3
HU05	T28	Implementar la capa de Presentación del microservicio “Playlist”	5	3
HU05	T29	Integrar el microservicio "Playlist" con otros microservicios	10	3
HU05	T30	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "Playlist" mediante la API por solicitudes HTTP.	5	3
HU06	T31	Diseñar e implementar la capa de Dominio del microservicio “Catalog” utilizando DDD.	3	2
HU06	T32	Diseñar e implementar la capa de Infraestructura del microservicio “Catalog”	8	2

HU06	T33	Diseñar e implementar la capa de Aplicación del microservicio “Catalog”	5	2
HU06	T34	Implementar la capa de Presentación del microservicio “Catalog”	5	2
HU06	T35	Integrar el microservicio "Catalog" con otros microservicios	8	2
HU06	T36	Realizar pruebas automatizadas con la herramienta SWAGGER , para comprobar la funcionalidad del microservicio "Catalog" la mediante la API por solicitudes HTTP.	5	2
HU07	T37	Diseñar el modelo de eventos utilizando un enfoque de dominio impulsado por eventos (Domain-Driven Design)	5	5
HU07	T38	Definir los flujos de eventos y sus respectivos puntos de integración	5	5
HU07	T39	Diseñar los mecanismos de publicación y suscripción de eventos mediante broker.	8	5
HU07	T40	Diseñar las herramientas de monitoreo y seguimiento de eventos	5	5
HU07	T41	Definir los servicios de que utilizarán comunicación asincrónica	1	5
HU07	T42	Implementar el modelo de eventos en el sistema	10	5
HU08	T43	Definir las tecnologías y lenguaje de desarrollo para la incorporación del servicio de email	1	2
HU08	T44	Diseñar e implementar el servicio de email utilizando una arquitectura de microservicios	5	1
HU08	T45	Implementar la funcionalidad de personalización de correos electrónicos	3	1
HU08	T46	Integrar el servicio de email con el sistema de registro de usuarios	8	1

HU08	T47	Realizar pruebas automatizadas con la herramienta POSTMAN , para comprobar la funcionalidad del servicio de "Email" la mediante la API por solicitudes HTTP.	3	1

Tabla 13 - Product Backlog

- **Planificación de Sprints**

Para el desarrollo del proyecto se planificaron un total de 8 sprints para cumplir con el Product Backlog. Se adicionó un sprint más previo al desarrollo del proyecto para diseñar la arquitectura del aplicativo basada en microservicios, creación de la organización y proyectos en Azure DevOps, creación y configuración de los repositorios con Repos y Git.

Sprint 0	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8
Preparación del entorno de desarrollo	HU07-T37	HU02-T7	HU04-T19	HU03-T13	HU05-T25	HU06-T31	HU08-43	HU01-T01
	HU07-T38	HU02-T8	HU04-T20	HU03-T14	HU05-T26	HU06-T32	HU08-44	HU01-T02
	HU07-T39	HU02-T9	HU04-T21	HU03-T15	HU05-T27	HU06-T33	HU08-45	HU01-T03
	HU07-T40	HU02-T10	HU04-T22	HU03-T16	HU05-T28	HU06-T34	HU08-46	HU01-T04
	HU07-T41	HU02-T11	HU04-T23	HU03-T17	HU05-T29	HU06-T35	HU08-47	HU01-T05
	-	HU02-T12	HU04-T24	HU03-T18	HU05-T30	HU06-T36	HU07-T42	HU01-T06
	-	-	-	-	HU07-T42	-		HU07-T42

Tabla 14 - Planificación de Sprints

4.2.5 Iteración Cero (Sprint 0)

- **Sprint 0 Planning**

En este Sprint se definirá la arquitectura de microservicios para el proyecto, se preparará el entorno, y herramientas de desarrollo y versionamiento que se usará en el trascurso del proyecto.

- **Ejecución del Sprint 0**
    - **Arquitectura del proyecto**

Se definió una arquitectura de microservicios destinada al streaming de música. El Back-end está compuesto por 6 microservicios que se comunican tanto de manera sincrónica por HTTP, como de manera asincrónica por Azure Service Bus lo que permite la gestión de eventos y notificaciones de manera eficiente y escalable.

Cada uno de estos microservicios se encarga de una tarea específica, como la gestión de usuarios, la reproducción de música, la gestión de listas de reproducción, entre otros. Además, se agregó un nuevo servicio de email que se acopla a la arquitectura principal y que tiene como propósito incorporar una nueva funcionalidad.

Para manejar la comunicación entre los microservicios y el Front-end, se utiliza un API Gateway que se encarga de dirigir las solicitudes del usuario al microservicio correspondiente. De esta forma, el API Gateway actúa como un punto de entrada único para la aplicación web del Front-end.

En resumen, el diseño de esta arquitectura de microservicios se basa en la separación de responsabilidades en diferentes microservicios, que se comunican entre sí de manera sincrónica y asincrónica, y son administrados por un API Gateway que funciona como punto de entrada único para el Front-end de la aplicación web (ver figura 37).

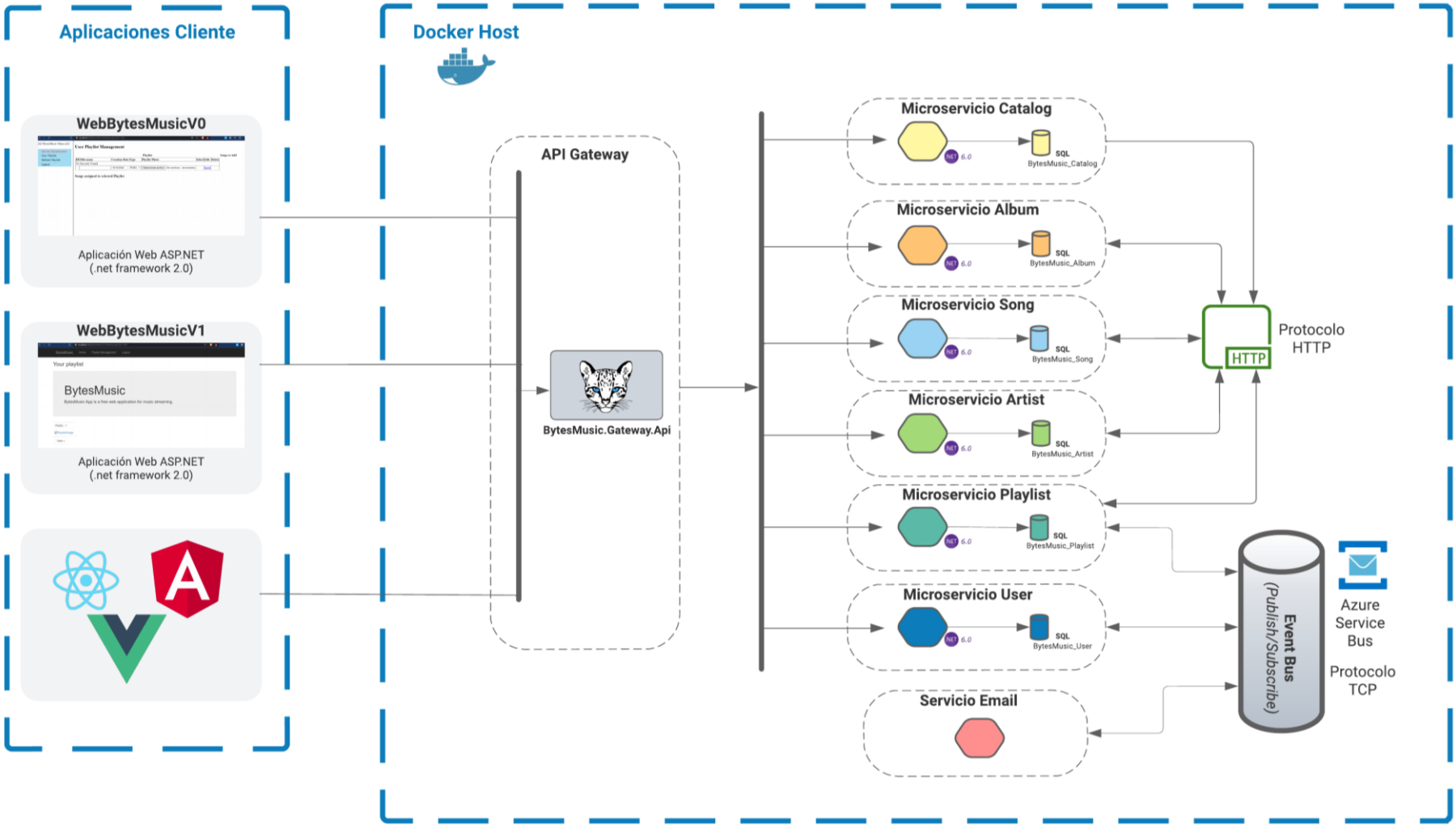


Figura 37 - Arquitectura de la aplicación de microservicios.

- **DevOps y Azure DevOps**

DevOps es una metodología de desarrollo de software que se enfoca en la colaboración y la comunicación entre los equipos de desarrollo y operaciones, para lograr una entrega más rápida y eficiente de software de alta calidad. El objetivo principal de DevOps es eliminar las barreras entre los equipos de desarrollo y operaciones, y fomentar la colaboración, la comunicación y la automatización[59].

Azure DevOps es una plataforma de colaboración y automatización de ciclo de vida de aplicaciones, que ayuda a los equipos de desarrollo y operaciones a trabajar juntos de manera más eficiente y efectiva. Azure DevOps ofrece una amplia gama de herramientas y servicios, como control de versiones, compilación, pruebas, despliegue y monitoreo, que ayudan a los equipos a automatizar los procesos de desarrollo y entrega de software.

Al elegir Azure DevOps, nos beneficiamos de una plataforma de colaboración completa que nos permitirá[59]:

- Automatizar todo el ciclo de vida de desarrollo de software, desde la planificación hasta el despliegue y monitoreo.
- Integrar fácilmente con otras herramientas y servicios, como GitHub, Jenkins y Docker.
- Tener un seguimiento completo del progreso del proyecto y el rendimiento de los equipos mediante paneles y métricas personalizables.
- Gestionar y automatizar los procesos de pruebas y garantizar la calidad del software mediante la integración con pruebas unitarias y funcionales automatizadas.
- Trabajar de forma colaborativa y tener una comunicación más eficiente entre el/los equipos de desarrollo.

Azure DevOps es una herramienta completa para la colaboración y la automatización de procesos en el desarrollo de software, y es una gran opción para

asumir el reto de migración. Por ello se creó dentro de la herramienta la organización “BytesMusicMicroservicesArchitecture” que contiene dos proyectos, el primero “BytesMusicMicroservices” destinado a la construcción de los microservicios y el segundo designado al servicio de email.

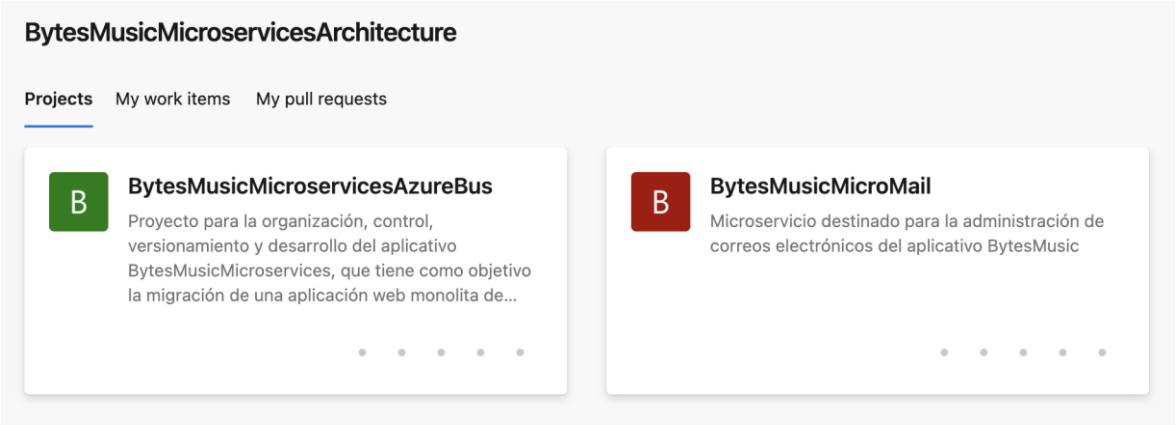


Figura 38 - Proyectos de la organización “BytesMusicMicroservicesArchitecture” en Azure DevOps

- **Azure Repos y control de versiones con Git**

Azure Repos es un servicio de control de versiones en la nube, que se proporciona como parte de Azure DevOps, la plataforma de colaboración y automatización de ciclo de vida de aplicaciones de Microsoft. Azure Repos permite a los equipos de desarrollo colaborar y controlar las versiones del código fuente de sus aplicaciones, así como llevar un registro de los cambios y las actualizaciones realizadas en el código fuente[60].

Azure Repos es compatible con Git y Team Foundation Version Control (TFVC), dos de los sistemas de control de versiones más populares utilizados en el desarrollo de software[60], en nuestro caso usaremos Git. Dentro de cada proyecto antes creado, en Azure Repos se creó su respectivo repositorio.

En el **Anexo 2** se encuentran los enlaces para acceder a los repositorios.

- **Sprint 0 Review**

Tras la finalización de este Sprint se logró obtener un ambiente de trabajo en el cual el equipo está preparado para empezar con el desarrollo del proyecto. Las herramientas de trabajo están listas para ser usadas, la arquitectura del proyecto está definida, y el equipo es consciente del flujo de trabajo que tiene que seguir para cumplir con sus objetivos.

- **Sprint 0 Retrospective**
    - **¿Qué salió bien?**

Se logró definir la arquitectura de manera correcta, crear el repositorio Git para el sistema del control de versiones y las herramientas de trabajo están listas para ser utilizadas.

    - **¿Qué se puede mejorar?**

Es importante que el equipo se familiarice con las nuevas herramientas que se van a utilizar, para lo cual se recomienda leer su documentación.

4.3 Fase de producción y estabilización

Esta fase se basa en la descripción de la fase “Desarrollo y Pruebas” de MOMMIV que consta de tres sub-fases[2]:

- Fase de desarrollo, explica en detalle lo que se realizó en cada Sprint a lo largo del proyecto, para crear los microservicios basándose en el Principio de Ocultación de Información, desarrollar los mecanismos de comunicación como son API’s de mensajería, y posibles llamadas entre microservicios, pruebas unitarias.
- Fase de evaluación y pruebas, realiza pruebas de la funcionalidad de negocio de tal forma que se tengan los mismos resultados que en la aplicación monolítica, además de pruebas no funcionales para comparar el rendimiento resultante de las dos arquitecturas.

- Fase de despliegue, detalla el proceso de liberación de los microservicios en un ambiente productivo con un despliegue controlado con herramientas DevOps.

4.3.1 Fase de desarrollo

4.3.1.1 Sprint 1

- **Sprint 1 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que la implementación de la comunicación asincrónica dentro del sistema.

- **Sprint 1 Backlog**

Sprint 1 Backlog	
Código	**Nombre**
HU07-T37	Diseñar el modelo de eventos utilizando un enfoque de dominio impulsado por eventos (Domain-Driven Design).
HU07-T38	Definir los flujos de eventos y sus respectivos puntos de integración.
HU07-T39	Diseñar los mecanismos de publicación y suscripción de eventos mediante broker.
HU07-T40	Diseñar las herramientas de monitoreo y seguimiento de eventos.
HU07-T41	Definir los servicios de que utilizarán comunicación asincrónica.

Tabla 15 - Sprint 1 Backlog

- **Ejecución del Sprint 1**

Una vez establecida la arquitectura de software a desarrollar es necesario generar una solución en blanco que contendrá una estructura de directorios con sus respectivos proyectos que irán definiendo la arquitectura planificada del sistema.

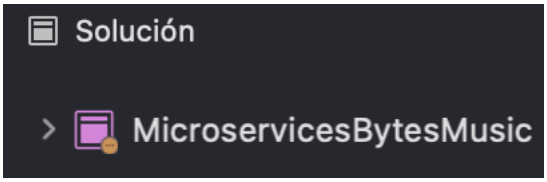


Figura 39 - Solución del proyecto.

En primera instancia se creará una estructura de directorios para cumplir con el objetivo del Sprint, cada directorio responde a un propósito específico:

- “Domain”, aquí estarán los componentes de dominios o artefactos.
- “Infra.Bus”, contendrá las interfaces y contratos que se relacionen al Event Bus o bróker.
- “Infra.IoC”, poseerá las inyecciones de dependencias de los distintos proyectos.

Dentro del directorio **“Domain”**, se creó un proyecto llamado **“MicroBroker.Domain.Core”** de tipo “Class Library”, donde se generará una interfaz genérica para representar el modelo abstracto de cualquier EventBus o bróker. Aquí se realizó una nueva estructura de directorios y sus respectivas clases e interfaces, con los directorios: “Bus”, “Commands” y “Events”, que incorporarán el patrón de diseño Mediador gracias al paquete “MediatR” en Asp.net. Además, representará la capa de Dominio en la Arquitectura basada en Eventos.

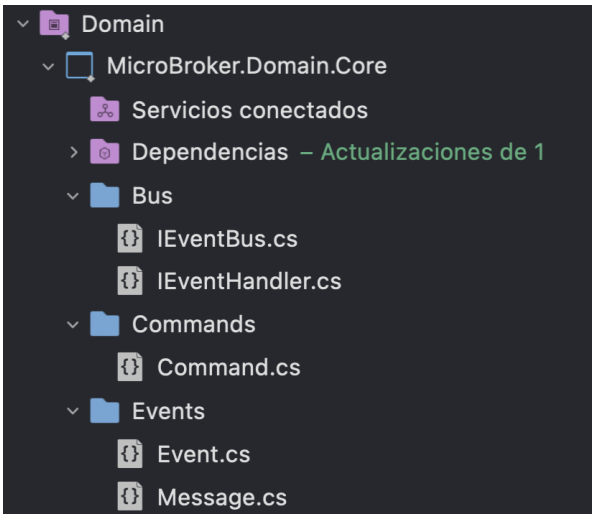


Figura 40 - Proyecto “MicroBroker.Domain.Core”

En el directorio “Bus” se verán representadas las operaciones realizadas con el bróker en la comunicación asincrónica, que podría abarcar cualquier tipo de herramienta EventBus por ejemplo, Azure Event Bus, Kafka, RabbitMQ, etc. En nuestro caso representa la futura implementación de Azure Event Bus.

Se creó la interfaz “IEventBus” donde; se declaró los dos métodos principales “Publish” para poder publicar un mensaje en el Queue y “Subscribe” para consumir ese mensaje del Queue; además para implementar el patrón Mediator en la interfaz se creó la operación “SendCommand” que permite la comunicación; recapitulando dicho patrón busca incorporar la comunicación entre objetos mediante un solo objeto, en nuestro caso el objeto Command que se generará más adelante. También se creó la interfaz “IEventHandler” donde; se generó una operación llamada “Handle” que permite realizar las operaciones sobre los eventos que van al bróker.

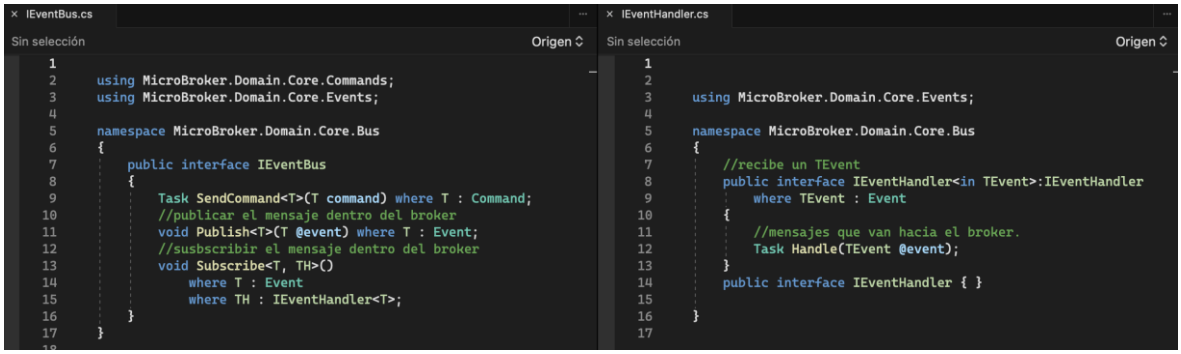


Figura 41 - “IEventBus” y “IEventHandler” scripts

En el directorio “Commands”. Se creó la clase abstracta “Command” esta es la instancia que tiene como objetivo llevar un mensaje de un componente a otro, por ello hereda los atributos y componentes de las clases por crear “Message”; se declaró la variable “Timestamp” para indicar la fecha exacta en la cual se mandó el mensaje y se generó su respectivo constructor donde se asignará el valor de fecha actual a dicha variable.

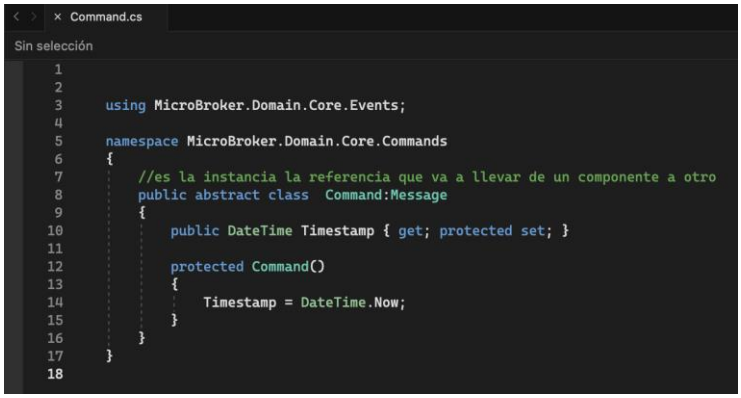


Figura 42 - "Command" script

En el directorio "Events" se creó la clase abstracta "Message" que hereda las propiedades de "IRequest" que es un componente propio de la dependencia "MediatR" que nos permite la implementación del patrón Mediator para generar ese canal de comunicación entre componentes; además se agregó la variable "MessageType" para saber el tipo de mensaje o clase que se está enviando y se generó su constructor donde se extraerá el tipo de clase que se envía como mensaje. Se creó la clase abstracta "Event" que representa un evento dentro de la comunicación asíncrona"; se declaró la variable "Timestamp" para indicar la fecha exacta en la cual se mandó el mensaje y se generó su respectivo constructor donde se asignará el valor de fecha actual a dicha variable.

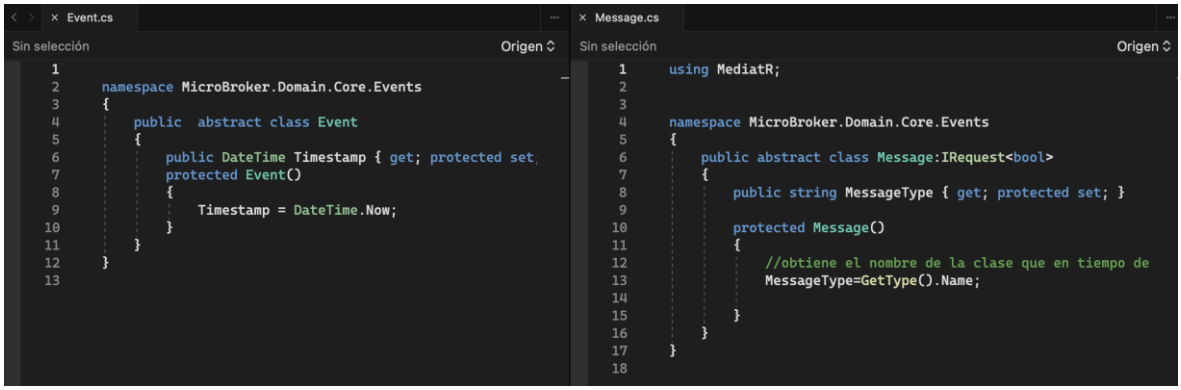


Figura 43 - "Event" y "Message" scripts

En el directorio "**Infra.Bus**", se creó un proyecto llamado "**MicroBroker.Infra.Bus**" de tipo "Class Library", espacio donde se desarrollará la implementación de la lógica de Azure Event Bus haciendo uso de la interfaz genérica y métodos abstractos generados en el proyecto "Microbroker.Domain.Core". Además,

representará la capa de Infraestructura del Bus de Eventos en la Arquitectura basada en Eventos.

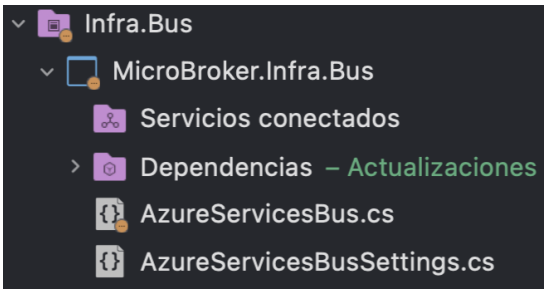


Figura 44 - Proyecto "MicroBroker.Infra.Bus"

Se creó dentro del proyecto la clase "AzureServiceBusSettings" la cual tiene como finalidad asignar valores de credenciales de acceso a nuestro servicio de mensajería asincrónica de forma dinámica.

Figura 45 - "AzureServicesBusSettings" scripts

Se creó una clase llamada "AzureServicesBus" la cual implementará los métodos de la interfaz "IEventBus". Se declaró adicionalmente un grupo de variables como: "_mediator" para la comunicación entre componentes, "_handlers" que es un diccionario que permita controlar los eventos que se produzcan en el bus, "_eventTypes" que permitirá almacenar todos los tipos de eventos que se ejecutarán.

AzureServicesBus.cs

AzureServicesBus(IMediator mediator, IServiceScopeFactory serviceScopeFactory, IOptions<AzureServicesBusSettings> settings)

```
11
12      namespace MicroBroker.Infra.Bus
13      {
14          public sealed class AzureServicesBus : IEventBus
15          {
16              private readonly IMediator _mediator;
17              private readonly Dictionary<string, List<Type>> _handlers;
18              private readonly List<Type> _eventTypes;
19              private readonly IServiceScopeFactory _serviceScopeFactory;
20
21              private readonly AzureServicesBusSettings _settings;
22
23              public AzureServicesBus(IMediator mediator, IServiceScopeFactory serviceScopeFactory,
24                                      IOptions<AzureServicesBusSettings> settings)
25              {
26                  _mediator = mediator;
27                  _handlers = new Dictionary<string, List<Type>>();
28                  _serviceScopeFactory = serviceScopeFactory;
29                  _eventTypes = new List<Type>();
30                  _settings = settings.Value;
31              }
```

Figura 46 - Variables del "AzureServiceBus" script

El método "Publish" contendrá la lógica de pasos para enviar un mensaje, empezando por armar el mensaje a enviar a nivel de bytes, crear un canal para él envió del mensaje basando su nombre el tipo de clase a enviar, y finalmente enviando el mensaje por dicho canal. El método "SendCommand" permitirá la comunicación de dos componentes dentro del bus.

```
33              public async void Publish<T>(T @event) where T : Event
34              {
35                  var connectionString= _settings.ConnectionString;
36
37                  var queueName = @event.GetType().Name;
38
39                  var message = JsonConvert.SerializeObject(@event);
40
41                  var body = Encoding.UTF8.GetBytes(message);
42
43                  ServiceBusMessage messageSend = new ServiceBusMessage(body);
44                  ServiceBusClient clientSender = new ServiceBusClient(connectionString);
45
46                  await clientSender.CreateSender(queueName).SendMessageAsync(messageSend);
47              }
48
49              public Task SendCommand<T>(T command) where T : Command
50              {
51                  return _mediator.Send(command);
52              }
```

Figura 47 - "Publish" y "SendCommand" script

El método "Subscribe" contendrá la lógica para leer los eventos que contiene una cola, por lo que es necesario tener el mensaje que viene representado por el evento y los controladores que corresponden a ese evento; aquí se verifica si no fue tratado un evento del mismo tipo, si no fue tratado un controlador del mismo tipo de evento caso contrario agregarlo a la lista de tipos de eventos y al diccionario respectivamente, y por último llamaremos al método "StartBasicConsumeAsync". El método "StartBasicConsumeAsync" tiene como

finalidad consumir los mensajes, a través de un “Processor” el cual tratará el mensaje lo mandará a procesar en el método “ProcesEvent” y dará como completado para sacar el mensaje de la cola. El método “ProcessEvent” permite procesar los mensajes y a su vez disparar el evento posterior al consumo del mensaje.

```
54      public void Subscribe<T, TH>()
55          where T : Event
56          where TH : IEventHandler<T>
57      {
58
59          var queueName = typeof(T).Name;
60          var handlerType = typeof(TH);
61          if (!_eventTypes.Contains(typeof(T)))
62          {
63              _eventTypes.Add(typeof(T));
64          }
65          if(!_handlers.ContainsKey(queueName)){
66              _handlers.Add(queueName,new List<Type>());
67          }
68
69          if (_handlers[queueName].Any(s => s.GetType() == handlerType))
70          {
71              throw new ArgumentException($"El handler exception {handlerType.Name} ya fue registrado anteriormente " +
72                  $"por '{queueName}'", nameof(handlerType));
73          }
74          _handlers[queueName].Add(handlerType);
75          StartBasicConsumeAsync<T>();
76
77      }
```

Figura 48 - “Suscribe” script.

```
108
109     private async Task ProcessEvent(string queueName, string body)
110     {
111         //consumir los mensajes que se encuentran dentro del queue
112         if (_handlers.ContainsKey(queueName))
113         {
114             using (var scope = _serviceScopeFactory.CreateScope())
115             {
116                 var subscriptions = _handlers[queueName];
117
118                 foreach (var subscription in subscriptions)
119                 {
120                     var handler = scope.ServiceProvider.GetService(subscription);  //Activator.CreateInsta
121                     if (handler == null) continue;
122                     var eventType = _eventTypes.SingleOrDefault(t => t.Name == queueName);
123                     var @event = JsonConvert.DeserializeObject(body, eventType);
124                     var concreteType = typeof(IEventHandler<>).MakeGenericType(eventType);
125
126                     await (Task)concreteType.GetMethod("Handle").Invoke(handler, new object[] { @event });
127
128                 }
129
130             }
131         }
132     }
```

Figura 49 - “ProcessEvent” script.

En el directorio **“Infra.IoC”**, se creó un proyecto se creó un proyecto llamado **“MicroBroker.Infra.IoC”** de tipo “Class Library”, este proyecto tiene como objetivo funcionar como un contenedor de objetos, en este caso específico va centralizar toda la inyección de dependencias de la solución, es decir, de todos los microservicios y del “Infra.Bus”. Además, representará la capa de Infraestructura de Inversión de Control.

Figura 50 - Proyecto “MicroBroker.Infra.IoC”

Se creó una clase llamada “DepencyContainer”, donde se implementó el método “RegisterServices” que retornará todas las configuraciones de los servicios registrados para hacer uso en los diferentes proyectos de la solución.

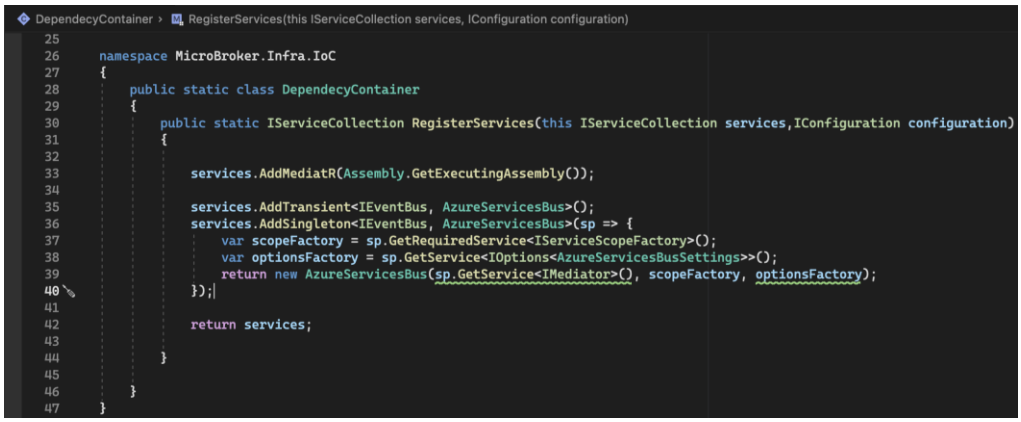


Figura 51 - “DepencyContainer” script

Ahora culminado el desarrollo, es importante definir los microservicios que se comunicarán entre sí para mostrar la comunicación asincrónica. Se escogió la comunicación asincrónica entre el microservicio “User” y “Playlist”, para realizar la creación de una playlist correspondiente a un usuario. Además, la comunicación entre el “User” y “Email”, para el enviar un email de bienvenida a los nuevos usuarios registrados. Como se observa en la figura 52, “User” jugará el papel de publicador, por el contrario “Playlist” y “Email” serán susbcriptores.

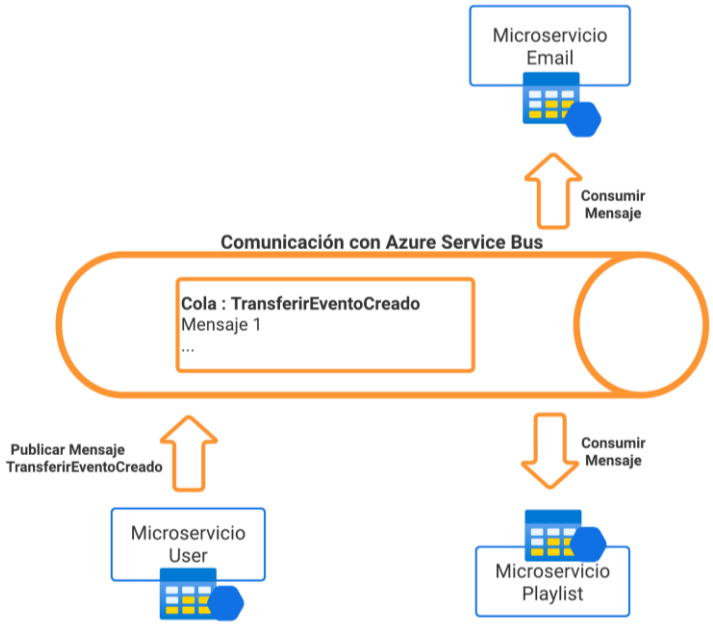


Figura 52 - Comunicación entre microservicios con Azure Service Bus

- **Sprint 1 Review**

Terminada la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU07-T37	Se diseño el modelo de eventos utilizando un enfoque de dominio impulsado por eventos (Domain-Driven Design).	Sí
HU07-T38	Se definió los flujos de eventos y sus respectivos puntos de integración.	Sí
HU07-T39	Se diseño los mecanismos de publicación y suscripción de eventos mediante broker.	Sí
HU07-T40	Se diseño las herramientas de monitoreo y seguimiento de eventos.	Sí
HU07-T41	Se definió los servicios de que utilizarán comunicación asincrónica.	Sí

Tabla 16 - Sprint 1 Review

- **Sprint 1 Retrospective**
    - **¿Qué salió bien?**

Se creó exitosamente un modelo abstracto de un bróker de bus de eventos con su implementación para Azure Service Bus, que nos permitirá su futura ejecución en los microservicios que se comunicarán asincrónicamente.

- **¿Qué se puede mejorar?**

Familiarizarse más con la herramienta Azure Service Bus para que monitorear los eventos entrantes y poder realizar pruebas de funcionamiento de manera más ágil.

4.3.1.2 Sprint 2

- **Sprint 2 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Microservicio “Song” inserte, actualice, elimine y lea registros de canciones del sistema.

- **Sprint 2 Backlog**

Sprint 2 Backlog	
Código	**Nombre**
HU02-T7	Diseñar e implementar la capa de Dominio del microservicio “Song” utilizando DDD.
HU02-T8	Diseñar e implementar la capa de Infraestructura del microservicio “Song”.
HU02-T9	Diseñar e implementar la capa de Aplicación del microservicio “Song”.
HU02-T10	Implementar la capa de Presentación del microservicio “Song”.
HU02-T11	Integrar el microservicio "Song" con otros microservicios.

HU02-T12	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Song" mediante la API por solicitudes HTTP.

Tabla 17 - Sprint 2 Backlog

- **Ejecución del Sprint 2**

Dentro de la solución, para delimitar el espacio de desarrollo de todos los microservicios, se creó el directorio “Microservices” el cuál contendrá un conjunto de directorios donde cada uno de ellos hará referencia al microservicio a desarrollar.

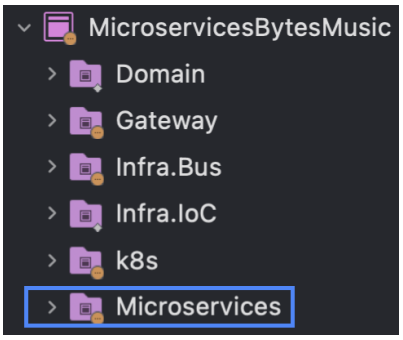


Figura 53 - “Microservices” directorio.

Para la creación del microservicio “Song”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la tabla 18.

Nombre del directorio	Responsabilidad
Domain	Domain hace referencia a la capa de Dominio. Aquí se albergará las entidades las cuales se basan en la/s tabla/s de base de datos y será/n la representación del modelo de

	negocio. El proyecto se construirá así para estar bajo los principios del patrón DDD.
Infrastructure	Infrastructure hace referencia a la capa de Persistencia e Infraestructura. Aquí se albergará las implementaciones de acceso a datos, mediante el uso del patrón de diseño Repositorio. Estas implementaciones incluyen un ORM como motor de persistencia a la base de datos, en este caso se usará Entity Framework Core.
Application	Application hace referencia a la capa de Aplicación. Aquí se declara servicios e interfaces. Estas interfaces incluyen abstracciones para operaciones que se realizarán mediante la infraestructura como acceso a datos, acceso al sistema de archivos, es decir, se definen por su funcionalidad como objetos de transferencia de datos simples (DTO).
Api	Api hace referencia a la capa de Presentación. El propósito de esta capa es retornar respuestas, por solicitudes del cliente. Api hará uso de la capa Aplicación, inyectará cada uno de sus servicios con su respectivo código de implementación y es aquí donde a través de métodos HTTP y sus respectivas rutas (paths) expuestas, el cliente podrá hacer uso del servicio.

Tabla 18 - Responsabilidades de los directorios que conforman un microservicio Asp.net con el patrón de Arquitectura Limpia.

En el directorio **"Domain"**, se creó un nuevo proyecto de tipo "Class Library" llamado **"MicroBroker.Song.Domain"**. El proyecto contendrá una estructura de directorios. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la "Arquitecura basada de Eventos" en la comunicación asincrónica, solo se efectuará si es necesaria una comunicación asincrónica. El segundo conjunto de directorios que se creó tiene dos directorios "Interfaces" y "Models".

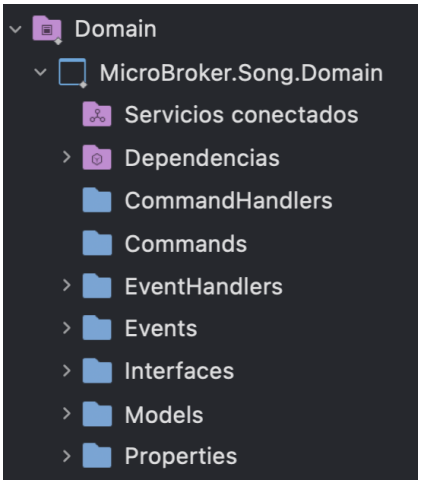


Figura 54 - Directorios principales del proyecto “Microbroker.Song.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará el modelo de la tabla de la base de datos “TBL_SONG”. Para ello se creó la clase “Song”, con sus respectivos atributos, constructores, y métodos getters y setters para acceder a sus atributos (ver figura 55).

Song › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.ComponentModel.DataAnnotations;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace MicroBroker.Song.Domain.Models
9   {
10      public class Song
11      {
12          [Key]
13          public int Id_Song { get; set; }
14          public string Song_Name { get; set; }
15          public string Song_Path { get; set; }
16          public int? Plays { get; set; }
17
18      }
19  }
```

Figura 55 - “Song” script

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó la interfaz

“ISongRepository”, la cual tiene declarado todos los métodos abstractos para las operaciones básicas de CRUD.

ISongRepository › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace MicroBroker.Song.Domain.Interfaces
8   {
9       public interface ISongRepository
10      {
11          IEnumerable<Domain.Models.Song> ReadSong();
12          int SaveSong(Domain.Models.Song song);
13          int UpdateSong(Domain.Models.Song song);
14          int DeleteSong(int id);
15          int  CheckExistSong(string songName);
16          IEnumerable<Domain.Models.Song> ReadSongs(List<int> idSongs);
17
18      }
19  }
```

Figura 56 - “ISongRepository” script

En el directorio **“Infrastructure”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Song.Infrastructure”**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Context”, “Migrations” y “Repository.

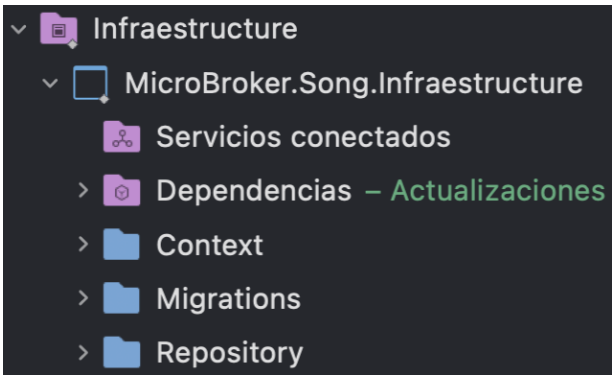


Figura 57 - Directorios principales del proyecto “MicroBroker.Song.Infrastructure”

El directorio “Context” contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext. Por ello se creó la clase “SongDbContext” que heredará las propiedades de DbContext y se la usará para consultar y guardar instancias de

las entidades. Es decir, “SongDbContext” convertirá la clase de C# “Song” a una entidad “TBL_SONG”, para la persistencia de registros en la base de datos “Song”.

SongDbContext › Sin selección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;


namespace MicroBroker.Song.Infraestructure.Context
{
    public class SongDbContext:DbContext
    {
        public SongDbContext(DbContextOptions<SongDbContext> options) : base(options) { }
        public DbSet<Domain.Models.Song> Tbl_Song { get; set; }

    }
}
```

Figura 58 - “SongDbContext” script

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. La clase “SongDbContextModelSnapshot” que está dentro del directorio fue autogenerada gracias a Entity Framework Core.

SongDbContextModelSnapshot › Sin selección

```
// <auto-generated />
using MicroBroker.Song.Infraestructure.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace MicroBroker.Song.Infraestructure.Migrations
{
    [DbContext(typeof(SongDbContext))]
    partial class SongDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.8")
                .HasAnnotation("Relational:MaxIdentifierLength", 128);

            SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder, 1L, 1);

            modelBuilder.Entity("MicroBroker.Song.Domain.Models.Song", b =>
                {
                    b.Property<int>("ID")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("ID"), 1L, 1);

                    b.Property<int>("Plays")
                        .HasColumnType("int");

                    b.Property<string>("SongName")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("SongPath")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.HasKey("ID");

                    b.ToTable("Song");
                });
#pragma warning restore 612, 618
        }
    }
}
```

Figura 59 - “UserDbContextModelSnapshot” script

Es decir, para que “SongDbContextModelSnapshot” se creé, se hizo uso de la “Consola de Gestión de Paquetes” de Visual Studio, donde se ejecutó el comando:

```
add-migration SongDbContextModel
```

El directorio “Repository” contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó la clase “SongRepository”, donde se instanciará la clase “SongDbContext” que permitirá consultar, crear, editar y eliminar registros de canciones en la base de datos “Song”.

Figura 60 - “SongRepository” script, parte 1

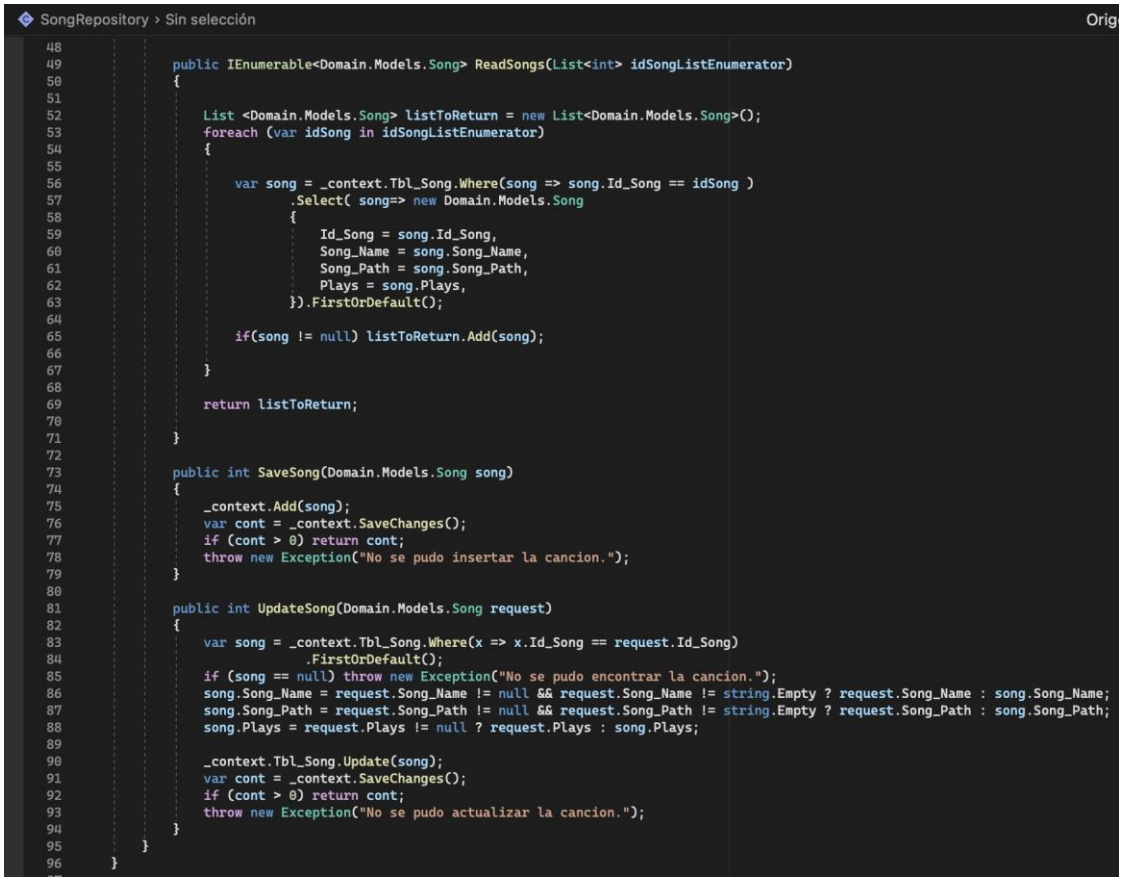


Figura 61 - “SongRepository” script, parte 2

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Song.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

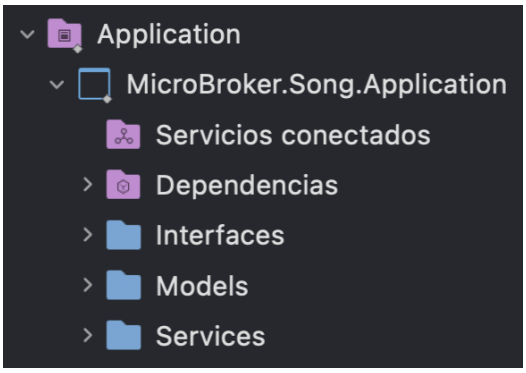


Figura 62 - Directorios principales del proyecto “MicroBroker.Song.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó la interfaz

“ISongService”, donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior.

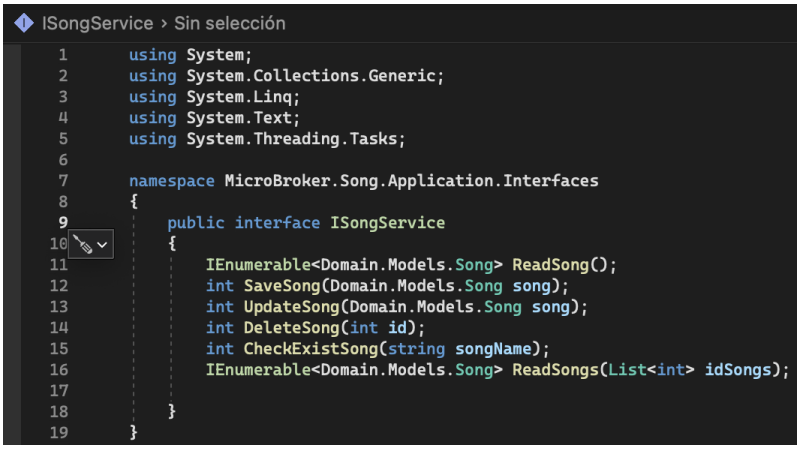


Figura 63 - “ISongService” script

El directorio “Services” contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó la clase “SongService”, que consumirá la interfaz antes creada “ISongService” y además hará uso de la interfaz “ISongRepository” para la implementación de cada método del servicio. “SongService” tiene como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio “Song”.

Figura 64 - “SongService” script

En el directorio “**Api**”, se creó un nuevo proyecto de tipo “Asp.Net Core Web Api” llamado “**MicroBroker.Song.Api**”. El proyecto de tipo “Asp.Net Core Web Api” otorga un conjunto de directorios, cada uno con propósito definido.

Figura 65 - Directorios principales del proyecto “MicroBroker.Song.Api”

El directorio “Controllers” contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó la clase “SongController”, que llamará a los métodos generados en el servicio “SongService”. En la clase se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado.

Figura 66 - “SongController” script

El directorio “Properties” tendrá el archivo autogenerado “launchSettings.json”. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación del microservicio “Song” desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es usado para la publicación del aplicativo en el servidor de producción, si son

necesarias ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo “appsettings.json”.

Figura 67 - Archivo de configuración “launchSettings.json” del microservicio “Song”.

- **Sprint 2 Review**

Terminada la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU02-T7	Se generó las clases en representación de la entidad “Song” y el modelo de negocio en su respectiva interfaz.	Sí
HU02-T8	Se creó las implementaciones de acceso a datos para la tabla de base de datos “TBL_SONG”.	Sí

HU02-T9	Se declaró los servicios e interfaces que funcionen como DTO's.	Sí
HU02-T10	Se creó las APIs REST que ejecutarán los servicios del microservicio "Song"	Sí
HU02-T11	No existe criterio de aceptación, debido a que el microservicio "Song" no depende de comunicación para su funcionamiento.	-
HU02-T12	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Song". La evidencia de las pruebas con el detalle de las mismas se las puede observar en el **Anexo 3 subsección 3.2**.	Sí

Tabla 19 - Sprint 2 Review

- **Sprint 2 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Después de la experticia conseguida del microservicio "Song" se prevé agilizar y mejorar los tiempos de desarrollo para los siguientes microservicios.

4.3.1.3 Sprint 3

- **Sprint 3 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Microservicio "Artist" inserte, actualice, elimine y lea registros de artistas del sistema.

- **Sprint 3 Backlog**

Sprint 3 Backlog	
Código	**Nombre**
HU04-T19	Diseñar e implementar la capa de Dominio del microservicio “Artist” utilizando DDD.
HU04-T20	Diseñar e implementar la capa de Infraestructura del microservicio “Artist”.
HU04-T21	Diseñar e implementar la capa de Aplicación del microservicio “Artist”.
HU04-T22	Implementar la capa de Presentación del microservicio “Artist”.
HU04-T23	Integrar el microservicio "Artist" con otros microservicios.
HU04-T24	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Artist" mediante la API por solicitudes HTTP.

Tabla 20 - Sprint 3 Backlog

- **Ejecución del Sprint 3**

Para la creación del microservicio “Artist”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la **tabla 18**.

En el directorio **“Domain”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Artist.Domain”**. El proyecto contendrá una estructura de directorios. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la “Arquitectura basada de Eventos” en la comunicación asincrónica, si es necesario en un trabajo futuro. El segundo conjunto de directorios que se creó tiene dos directorios “Interfaces” y “Models”.

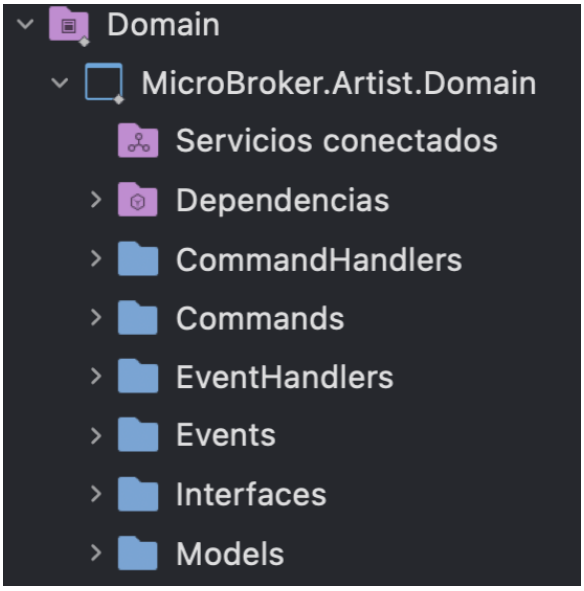


Figura 68 - Directorios principales del proyecto “Microbroker.Artist.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará los modelos de las tablas de la base de datos “TBL_ARTIST”, “TBL_PLAYER” y también a la tabla externa “TBL_SONG”. Para ello se creó las clases “Artist”, “Player” y “SongRemote” respectivamente, con sus respectivos atributos, constructores, y métodos getters y setters para acceder a sus atributos (ver figuras 69, 70, 71).

Artist › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.ComponentModel.DataAnnotations;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace MicroBroker.Artist.Domain.Models
9   {
10      public class Artist
11      {
12          [Key]
13          public int Id_Artist { get; set; }
14          public string Artist_Name { get; set; }
15          public string Artist_LastName { get; set; }
16          public string Artist_Image { get; set ; }
17      }
18  }
```

Figura 69 - “Artist” script

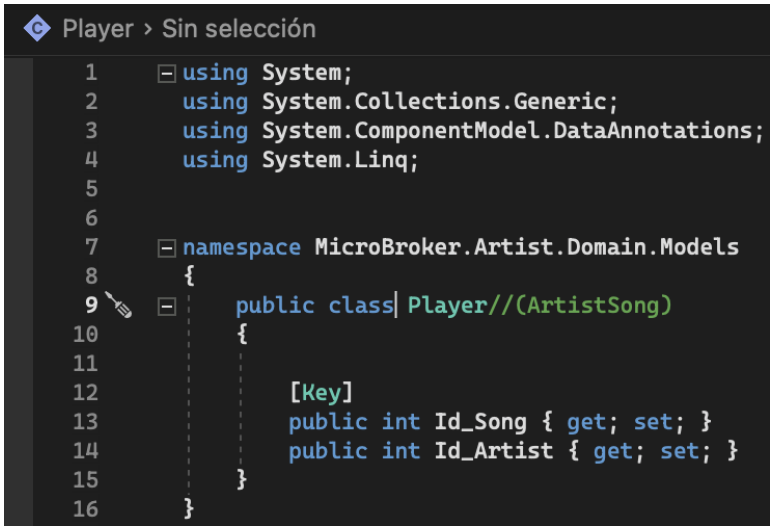


Figura 70 - "Player" script

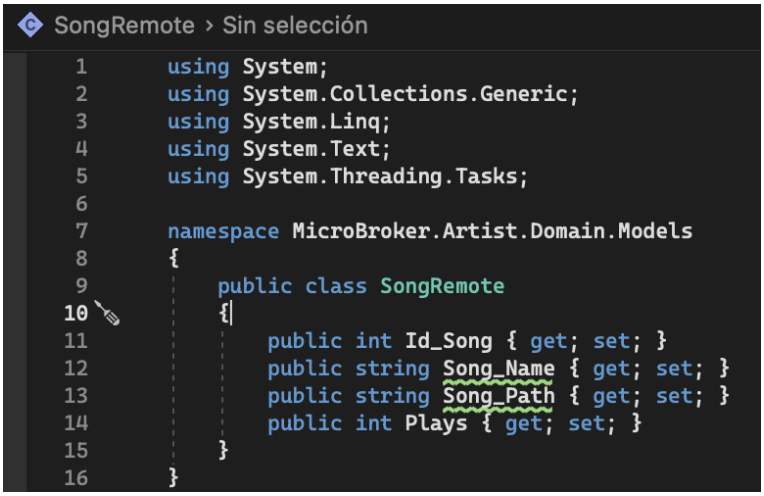


Figura 71 - "SongRemote" script

El directorio "Interfaces" contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó las interfaces "IArtistRepository" y "IPlayerRepository", las cuales tienen declarado todos los métodos abstractos para las operaciones básicas de CRUD.

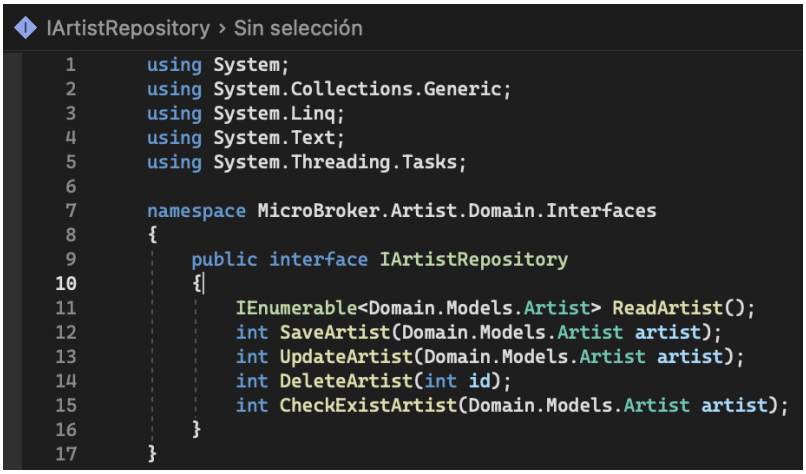


Figura 72 - "IArtistRepository" script

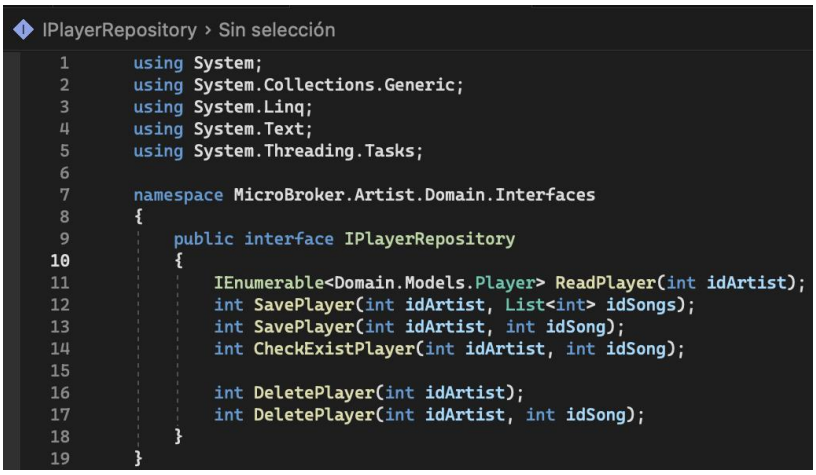


Figura 73 - "IPlayerRepository" script

En el directorio **"Infrastructure"**, se creó un nuevo proyecto de tipo "Class Library" llamado **"MicroBroker.Artist.Infrastructure"**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos "Context", "Migrations" y "Repository.

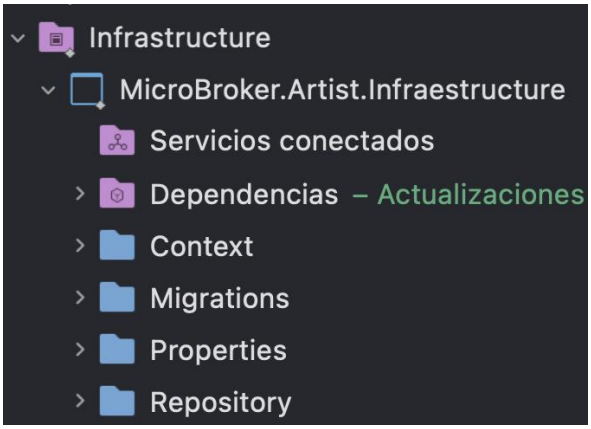


Figura 74 - Directorios principales del proyecto "MicroBroker.Artist.Infrastructure"

El directorio "Context" contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext. Por ello se creó las clases "ArtistDbContext", "PlayerDbContext" que heredarán las propiedades de DbContext y se las usará para consultar y guardar instancias de las entidades. Es decir, "ArtistDbContext". "PlayerDbContext" convertirán las clases de C# "Artist", "Player" a las entidades "TBL_ARTIST", "TBL_PLAYER" respectivamente, para la persistencia de registros en la base de datos "Artist".

ArtistDbContext › Sin selección

```
1   using Microsoft.EntityFrameworkCore;
2
3
4   namespace MicroBroker.Artist.Infraestructure.Context
5   {
6       public class ArtistDbContext:DbContext
7       {
8           public ArtistDbContext(DbContextOptions<ArtistDbContext> options) : base(options) { }
9           public DbSet<Domain.Models.Artist> Tbl_Artist { get; set; }
10      }
11  }
12
```

Figura 75 - "ArtistDbContext" script

PlayerDbContext › Sin selección

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Artist.Infraestructure.Context
{

    public class PlayerDbContext : DbContext
    {
        public PlayerDbContext(DbContextOptions<PlayerDbContext> options) : base(options) { }
        public DbSet<Domain.Models.Player > Tbl_Player { get; set; }
    }
}
```

Figura 76 - “PlayerDbContext” script

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. Las clases “ArtistDbContextModelSnapshot” y “PlayerDbContextModelSnapshot” que están dentro del directorio fue autogeneradas gracias a Entity Framework Core.

ArtistDbContextModelSnapshot › Sin selección

```
// <auto-generated />
using MicroBroker.Artist.Infraestructure.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace MicroBroker.Artist.Data.Migrations
{
    [DbContext(typeof(ArtistDbContext))]
    partial class ArtistDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.8")
                .HasAnnotation("Relational:MaxIdentifierLength", 128);

            SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder, 1L, 1);

            modelBuilder.Entity("MicroBroker.Artist.Domain.Models.Artist", b =>
                {
                    b.Property<int>("ID")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("ID"), 1L, 1);

                    b.Property<string>("FirstName")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("ImagePath")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("LastName")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.HasKey("ID");

                    b.ToTable("Artist");
                });
#pragma warning restore 612, 618
        }
    }
}
```

Figura 77 - “ArtistDbContextModelSnapshot” script

PlayerDbContextModelSnapshot › Sin selección

```
// <auto-generated />
using MicroBroker.Artist.Infraestructure.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace MicroBroker.Artist.Data.Migrations.PlayerDb
{
    [DbContext(typeof(PlayerDbContext))]
    partial class PlayerDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.8")
                .HasAnnotation("Relational:MaxIdentifierLength", 128);

            SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder, 1L, 1);

            modelBuilder.Entity("MicroBroker.Artist.Domain.Models.Player", b =>
                {
                    b.Property<int>("ID")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("ID"), 1L, 1);

                    b.Property<int>("IdArtist")
                        .HasColumnType("int");

                    b.Property<int>("IdSong")
                        .HasColumnType("int");

                    b.HasKey("ID");

                    b.ToTable("Player");
                });
#pragma warning restore 612, 618
        }
    }
}
```

Figura 78 - “PlayerDbContextModelSnapshot” script

Es decir, para que “ArtistDbContextModelSnapshot” y “PlayerDbContextModelSnapshot” se creen, se hizo uso de la “Consola de Gestión de Paquetes” de Visual Studio, donde se ejecutó los siguientes comandos:

```
add-migration ArtistDbContextModel
```

```
add-migration PlayerDbContextModel
```

El directorio “Repository” contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó las clases “ArtistRepository” y “PlayerRepository”, donde se instanciarán las clases “ArtistDbContext” y “PlayerDbContext” respectivamente, que permitirán consultar, crear, editar y

eliminar registros de artistas y también relacionar las canciones con un artista específico, todo ello en la base de datos “Artist”.

ArtistRepository › Sin selección

```
using MicroBroker.Artist.Infraestructure.Context;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Artist.Infraestructure.Repository
{
    public class ArtistRepository : Domain.Interfaces.IArtistRepository
    {
        private ArtistDbContext _context;

        public ArtistRepository(ArtistDbContext context)
        {
            _context = context;
        }

        public int CheckExistArtist(Domain.Models.Artist request)
        {
            var artist = _context.Tbl_Artist.Where(x => x.Artist_Name == request.Artist_Name)
                .Where(x => x.Artist_LastName == request.Artist_LastName)
                            .FirstOrDefault();
             return artist == null ? 0 : artist.Id_Artist;
        }

        public int DeleteArtist(int id)
        {
            var artist = _context.Tbl_Artist.Where(x => x.Id_Artist == id)
                    .FirstOrDefault();
            if (artist == null) throw new Exception("No se pudo encontrar el artista.");
            _context.Tbl_Artist.Remove(artist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar el artista.");
        }

        public IEnumerable<Domain.Models.Artist> ReadArtist()
        {
            return _context.Tbl_Artist;
        }

        public int SaveArtist(Domain.Models.Artist artist)
        {
            _context.Add(artist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el artista.");
        }

        public int UpdateArtist(Domain.Models.Artist request)
        {
            var artist = _context.Tbl_Artist.Where(x => x.Id_Artist == request.Id_Artist)
                    .FirstOrDefault();
            if (artist == null) throw new Exception("No se pudo encontrar el artista.");

            artist.Artist_Name = request.Artist_Name != null && request.Artist_Name != string.Empty ? request.Artist_Name : artist.Artist_Name;
            artist.Artist_LastName = request.Artist_LastName != null && request.Artist_LastName != string.Empty ? request.Artist_LastName : artist.Artist_LastName;
            artist.Artist_Image = request.Artist_Image != null && request.Artist_Image != string.Empty ? request.Artist_Image : artist.Artist_Image;

            _context.Tbl_Artist.Update(artist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo actualizar la playlist.");
        }
    }
}
```

Figura 79 - “ArtistRepository” script

PlayerRepository › Sin selección

```
namespace MicroBroker.Artist.Infraestructure.Repository
{
    public class PlayerRepository:IPlayerRepository
    {
        private  PlayerDbContext _context;

        public PlayerRepository(PlayerDbContext context)
        {
            _context = context;
        }

        public int CheckExistPlayer(int idArtist, int idSong)
        {
            var player = _context.Tbl_Player.Where(x => x.Id_Artist == idArtist).Where(x => x.Id_Song == idSong)
                            .FirstOrDefault();
            return player == null ? 0 : player.Id_Song;
        }

        public int DeletePlayer(int idArtist)
        {
            var player = _context.Tbl_Player.Where(x => x.Id_Artist == idArtist).ToList();
            if (player == null) throw new Exception("No se pudo encontrar el player.");
            _context.Tbl_Player.RemoveRange(player);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar el player.");
        }

        public int DeletePlayer(int idArtist, int idSong)
        {
            var player = _context.Tbl_Player.Where(x => x.Id_Song == idSong).Where(x => x.Id_Artist == idArtist)
                        .FirstOrDefault();
            if (player == null) throw new Exception("No se pudo encontrar el player.");
            _context.Tbl_Player.Remove(player);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar el player.");
        }

        public IEnumerable<Player> ReadPlayer(int idArtist)
        {
            var player = _context.Tbl_Player.Where(x => x.Id_Artist == idArtist).ToList();
            return player;
        }

        public int SavePlayer(int idArtist, List<int> idSongs)
        {
           List<Player> player = new List<Player>();

            foreach (var id in idSongs)
            {
                Player currentPlayer = new Player();
                currentPlayer.Id_Artist = idArtist;
                currentPlayer.Id_Song = id;
                player.Add(currentPlayer);
            }

            _context.Tbl_Player.AddRange(player);

            var cont =_context.SaveChanges();

            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el player.");
        }

        public int SavePlayer(int idArtist, int idSong)
        {
            Player player = new Player();
            player.Id_Artist = idArtist;
            player.Id_Song = idSong;
            _context.Add(player);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el player.");
        }
    }
}
```

Figura 80 - “PlayerRepository” script

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Artist.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

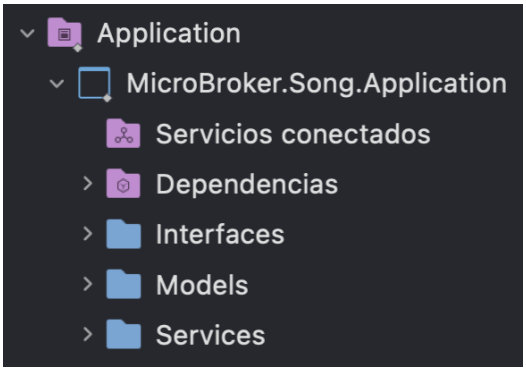


Figura 81 - Directorios principales del proyecto “MicroBroker.Artist.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó las interfaces “IArtistService” y “IPlayerService” donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior. También se creó la interfaz “ISongServiceRemote” donde se declaró los métodos abstractos para la comunicación sincrónica con el microservicio “Song”.

IArtistService › Sin selección

```
1
2
3   namespace MicroBroker.Artist.Application.Interfaces
4   {
5       public interface IArtistService
6       {
7           IEnumerable<Domain.Models.Artist> ReadArtist();
8           int SaveArtist(Domain.Models.Artist artist);
9           int UpdateArtist(Domain.Models.Artist artist);
10          int DeleteArtist(int id);
11          int CheckExistArtist(Domain.Models.Artist artist);
12      }
13  }
14
```

Figura 82 - “IArtistService” script

Figura 83 - "IPlayerService" script

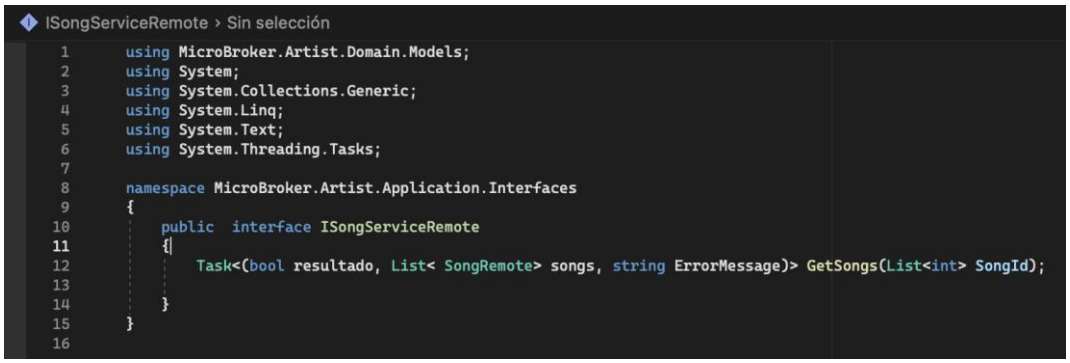


Figura 84 - "ISongServiceRemote" script

El directorio "Services" contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó las clases "ArtistService" y "PlayerService". "ArtistService" consumirá la interfaz "IArtistService" y hará uso de "IArtistRepository" mientras que "PlayerService" consumirá "IPlayerService" y usará las interfaces "IPlayerService" y "ISongServiceRemote", las interfaces usadas son para la implementación de cada método de servicio. "ArtistService" y "PlayerService" tienen como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio "Artist".

ArtistService › Sin selección

```
using MicroBroker.Artist.Application.Interfaces;
using MicroBroker.Artist.Domain.Interfaces;
using MicroBroker.Domain.Core.Bus;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Artist.Application.Services
{
    public class ArtistService : IArtistService
    {

        private readonly IArtistRepository _artistRepository;
        private readonly IEventBus _bus;

        public ArtistService(IArtistRepository artistRepository, IEventBus bus)
        {
            _artistRepository = artistRepository;
            _bus = bus;
        }
        public int CheckExistArtist(Domain.Models.Artist artist)
        {
            return _artistRepository.CheckExistArtist(artist);

        }

        public int DeleteArtist(int id)
        {
            return _artistRepository.DeleteArtist(id);

        }

        public IEnumerable<Domain.Models.Artist> ReadArtist()
        {
            return _artistRepository.ReadArtist();

        }

        public int SaveArtist(Domain.Models.Artist artist)
        {
            return _artistRepository.SaveArtist(artist);

        }

        public int UpdateArtist(Domain.Models.Artist artist)
        {
            return _artistRepository.UpdateArtist(artist);

        }
    }
}
```

Figura 85 - “ArtistService” script

PlayerService › Sin selección

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Artist.Application.Services
{
    public class PlayerService : IPlayerService
    {
        private readonly IPlayerRepository _playerRepository;
        private readonly IEventBus _bus;
        private readonly ISongServiceRemote _songServiceRemote;

        public PlayerService(IPlayerRepository playerRepository, IEventBus bus, ISongServiceRemote songServiceRemote)
        {
            _playerRepository = playerRepository;
            _bus = bus;
            _songServiceRemote = songServiceRemote;
        }

        public int CheckExistPlayer(int idArtist, int idSong)
        {
            return _playerRepository.CheckExistPlayer(idArtist, idSong);
        }

        public int DeletePlayer(int idArtist)
        {
            return _playerRepository.DeletePlayer(idArtist);

        }

        public int DeletePlayer(int idArtist, int idSong)
        {
            return _playerRepository.DeletePlayer(idArtist, idSong);

        }

        public IEnumerable<SongRemote> ReadPlayer(int idArtist)
        {
            var player = _playerRepository.ReadPlayer(idArtist);
            List<int> idSongsList = new List<int>();
            foreach (var song in player)
                idSongsList.Add(song.Id_Song);

            var response = _songServiceRemote.GetSongs(idSongsList);

            return response.Result.songs;
        }

        public int SavePlayer(int idArtist, List<int> idSongs)
        {
            return _playerRepository.SavePlayer(idArtist, idSongs)
        }

        public int SavePlayer(int idArtist, int idSong)
        {
            return _playerRepository.SavePlayer(idArtist, idSong);
        }
    }
}
```

Ln 14, Col 6

Figura 86 - “PlayerService” script

También se creó la clase “SongServiceRemote”, la cual implementará la comunicación sincrónica mediante protocolo HTTP y haciendo uso de las rutas del microservicio “Song” para obtener los registros solicitados de canciones.

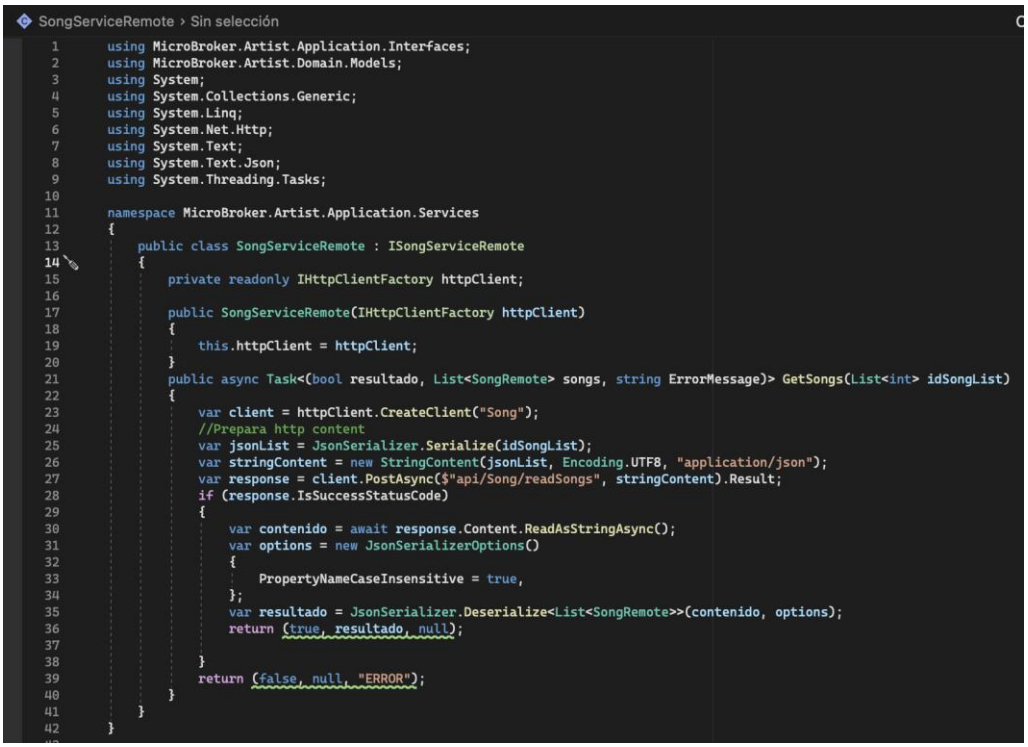


Figura 87 - “SongServiceRemote” script

En el directorio “**Api**”, se creó un nuevo proyecto de tipo “Asp.Net Core Web Api” llamado “**MicroBroker.Artist.Api**”. El proyecto de tipo “Asp.Net Core Web Api” otorga un conjunto de directorios, cada uno con propósito definido.

Figura 88 - Directorios principales del proyecto “MicroBroker.Artist.Api”

El directorio “Controllers” contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó las clases “ArtistController” y “PlayerController”, que llamarán a los métodos generados en los servicios “ArtistService” y “PlayerService” respectivamente, mediante sus interfaces. En las clases se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado.

Figura 89 - “ArtistController” script

Figura 90 - “PlayerController” script

El directorio “Properties” tendrá el archivo autogenerado “launchSettings.json”. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación del microservicio “Artist” desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es usado para la publicación del aplicativo en el servidor de producción, si son necesarias ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo “appsettings.json”.

Figura 91 - Archivo de configuración “launchSettings.json” del microservicio “Artist”.

- **Sprint 3 Review**

Al finalizar con la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU04-T19	Se generó las clases en representación de la entidad “Artist” y “Player”, y el modelo de negocio en su respectiva interfaz.	Sí
HU04-T20	Se creó las implementaciones de acceso a datos para la tabla de base de datos “TBL_ARTIST” y “TBL_PLAYER”.	Sí

HU04-T21	Se declaró los servicios e interfaces que funcionen como DTO's.	Sí
HU04-T22	Se creó las APIs REST que ejecutarán los servicios del microservicio "Artist"	Sí
HU04-T23	Se creó en la capa de Aplicación la llamada al microservicio "Song" mediante HTTP.	Sí
HU04-T24	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Artist". La evidencia de las pruebas con el detalle de estas se las puede observar en el **Anexo 3 subsección 3.5**.	Sí

Tabla 21 - Sprint 3 Review

- **Sprint 3 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Después de la experticia conseguida del microservicio "Artist", donde se incluyó una integración con otro microservicio, se prevé agilizar y mejorar los tiempos de desarrollo para los siguientes microservicios en cuanto a comunicación sincrónica.

4.3.1.4 Sprint 4

- **Sprint 4 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Microservicio "Album" (escrito sin tilde para evitar problemas en la codificación) inserte, actualice, elimine y lea registros de álbumes del sistema.

- **Sprint 4 Backlog**

Sprint 4 Backlog	
Código	**Nombre**
HU03-T13	Diseñar e implementar la capa de Dominio del microservicio “Album” utilizando DDD.
HU03-T14	Diseñar e implementar la capa de Infraestructura del microservicio “Album”.
HU03-T15	Diseñar e implementar la capa de Aplicación del microservicio “Album”.
HU03-T16	Implementar la capa de Presentación del microservicio “Album”.
HU03-T17	Integrar el microservicio "Album" con otros microservicios.
HU03-T18	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Album" mediante la API por solicitudes HTTP.

Tabla 22 - Sprint 4 Backlog

- **Ejecución del Sprint 4**

Para la creación del microservicio “Album”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la Tabla 18.

En el directorio **“Domain”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Album.Domain”**. El proyecto contendrá una estructura de directorios. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la “Arquitectura basada de Eventos” en la

comunicación asincrónica, si es necesario en un trabajo futuro. El segundo conjunto de directorios que se creó tiene dos directorios “Interfaces” y “Models”.

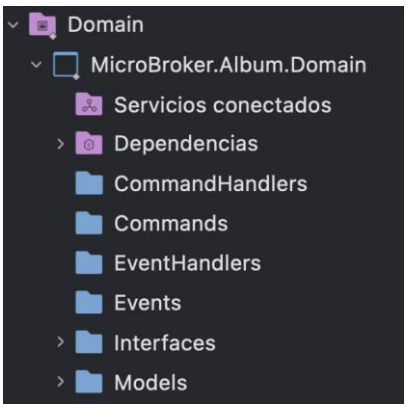


Figura 92 - Directorios principales del proyecto “Microbroker.Album.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará los modelos de las tablas de la base de datos “TBL_ALBUM”, “TBL_TRACKLIST” y también a la tabla externa “TBL_SONG”. Para ello se creó las clases “Album”, “Tracklist” y “SongRemote” respectivamente, con sus respectivos atributos, constructores, y métodos getters y setters para acceder a sus atributos (ver figuras 93,94,95).

Album › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.ComponentModel.DataAnnotations;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace MicroBroker.Album.Domain.Models
9   {
10      public class Album
11      {
12          [Key]
13          public int Id_Album { get; set; }
14          public string Title_Album { get; set; }
15          public int Release_Year { get; set; }
16          public string Album_Image_Path { get; set; }
17      }
18  }
```

Figura 93 - “Album” script

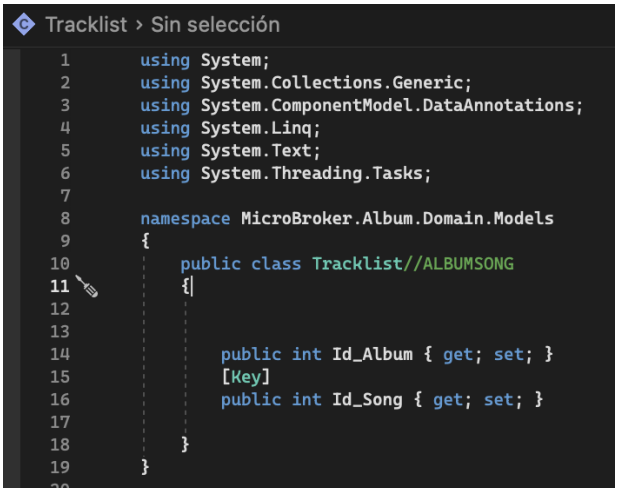


Figura 94 - “Tracklist” script

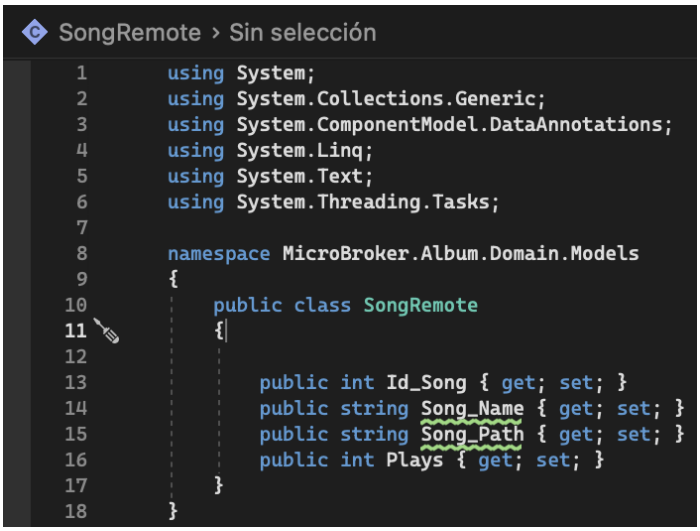


Figura 95 - “SongRemote” script

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó las interfaces “IAlbumRepository” y “ITracklistRepository”, las cuales tienen declarado todos los métodos abstractos para las operaciones básicas de CRUD.

IAlbumRepository › Sin selección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Album.Domain.Interfaces
{
    public interface IAlbumRepository
    {
        IEnumerable<Domain.Models.Album> ReadAlbum();
        int SaveAlbum(Domain.Models.Album album);
        int UpdateAlbum(Domain.Models.Album album);
        int DeleteAlbum(int id);
        int CheckExistAlbum(string albumTitle);

    }
}
```

Figura 96 - "IAlbumRepository" script

ITracklistRepository › Sin selección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Album.Domain.Interfaces
{
    public interface  ITracklistRepository

    {
        IEnumerable<Domain.Models.Tracklist> ReadTracklist(int idAlbum);

        int SaveTracklist(int idAlbum, List<int> idSongs);
        int SaveTracklist(int idAlbum, int idSong);
        int CheckExistTracklist(int idAlbum, int idSong);
        int DeleteTracklist(int idSong);
        int DeleteSongOnTracklist(int idAlbum, int idSong);


    }
}
```

Figura 97 - "ITracklistRepository" script

En el directorio **"Infrastructure"**, se creó un nuevo proyecto de tipo "Class Library" llamado **"MicroBroker.Album.Infrastructure"**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos "Context", "Migrations" y "Repository.

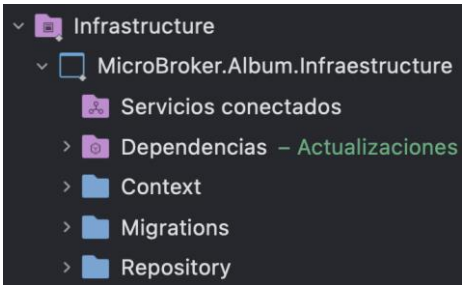


Figura 98 - Directorios principales del proyecto "MicroBroker.Album.Infrastructure"

El directorio "Context" contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext. Por ello se creó las clases "AlbumDbContext", "TracklistDbContext" que heredarán las propiedades de DbContext y se las usará para consultar y guardar instancias de las entidades. Es decir, "AlbumDbContext", "TracklistDbContext" convertirán las clases de C# "Album", "Tracklist" a las entidades "TBL_ALBUM", "TBL_TRACKLIST" respectivamente, para la persistencia de registros en la base de datos "Album".

AlbumDbContext › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using Microsoft.EntityFrameworkCore;
7
8   namespace MicroBroker.Album.Infraestructure.Context
9   {
10      public class AlbumDbContext : DbContext
11      {
12          public AlbumDbContext(DbContextOptions<AlbumDbContext> options) : base(options) { }
13          public DbSet<Domain.Models.Album> Tbl_Album { get; set; }
14      }
15  }
```

Figura 99 - "AlbumDbContext" script

TracklistDbContext › Sin selección

```
1   using Microsoft.EntityFrameworkCore;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace MicroBroker.Album.Infraestructure.Context
9   {
10      public class TracklistDbContext: DbContext
11      {
12          public TracklistDbContext(DbContextOptions<TracklistDbContext> options) : base(options) { }
13          public DbSet<Domain.Models.Tracklist> Tbl_Tracklist { get; set; }
14      }
15  }
```

Figura 100 - "TracklistDbContext" script

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. Las clases “AlbumDbContextModelSnapshot” y “TracklistDbContextModelSnapshot” que están dentro del directorio fue autogeneradas gracias a Entity Framework Core.

AlbumDbContextModelSnapshot › Sin selección

```
// <auto-generated />
using MicroBroker.Album.Infraestructure.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace MicroBroker.Album.Data.Migrations
{
    [DbContext(typeof(AlbumDbContext))]
    partial class AlbumDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.8")
                .HasAnnotation("Relational:MaxIdentifierLength", 128);

            SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder, 1L, 1);

            modelBuilder.Entity("MicroBroker.Album.Domain.Models.Album", b =>
                {
                    b.Property<int>("Id")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("Id"), 1L, 1);

                    b.Property<string>("ImagePath")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("ReleaseYear")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("Title")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.HasKey("Id");

                    b.ToTable("Album");
                });
#pragma warning restore 612, 618
        }
    }
}
```

Figura 101 - “AlbumDbContextModelSnapshot” script

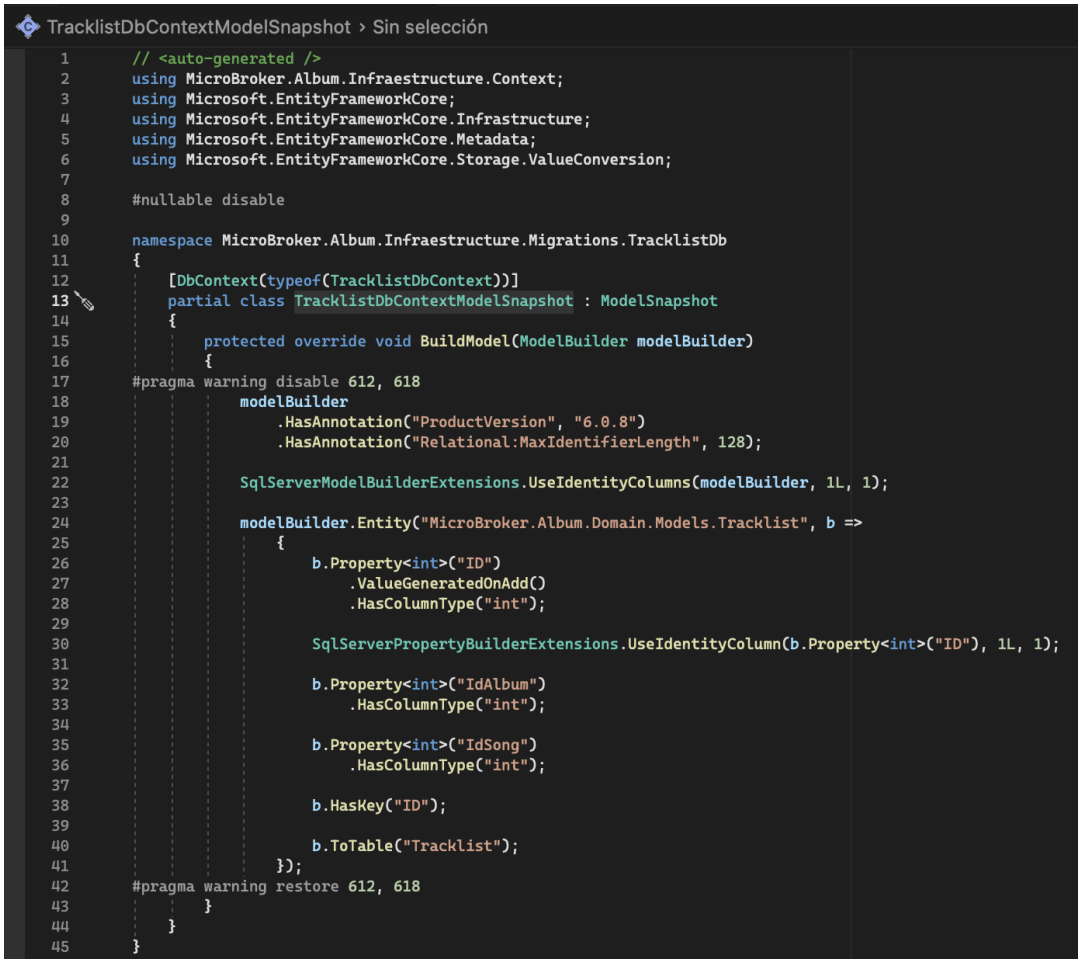


Figura 102 - "TracklistDbContextModelSnapshot" script

Es decir, para que "AlbumDbContextModelSnapshot" y "TracklistDbContextModelSnapshot" se generen, se hizo uso de la "Consola de Gestión de Paquetes" de Visual Studio, donde se ejecutó los siguientes comandos:

```
add-migration AlbumDbContextModel
```

```
add-migration TracklistDbContextModel
```

El directorio "Repository" contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó las clases "AlbumRepository" y "TracklistRepository", donde se instanciarán las clases "AlbumDbContext" y "TracklistDbContext" respectivamente, que permitirán consultar, crear, editar y

eliminar registros de artistas y también relacionar las canciones con un artista específico, todo ello en la base de datos “Album”.

Figura 103 - “AlbumRepository” script

TracklistRepository › Sin selección

```
namespace MicroBroker.Album.Infraestructure.Repository
{
    public class TracklistRepository : Domain.Interfaces.ITracklistRepository
    {
        private TracklistDbContext _context;

        public TracklistRepository(TracklistDbContext context)
        {
            _context = context;
        }

        public int CheckExistTracklist(int idAlbum, int idSong)
        {

            var tracklist = _context.Tbl_Tracklist.Where(x => x.Id_Album == idAlbum)
                                        .Where(x => x.Id_Song == idSong)
                                .FirstOrDefault();
            return tracklist == null ? 0 : tracklist.Id_Song;
        }

        public int DeleteSongOnTracklist(int idAlbum, int idSong)
        {

            using (SqlConnection con = new SqlConnection(_context.Database.GetConnectionString()))
            {
                using (SqlCommand cmd = new SqlCommand())
                {
                    cmd.CommandText = "DELETE [TBL_TRACKLIST] where [ID_SONG] = @ID_SONG AND [ID_ALBUM] = @ID_ALBUM";
                    cmd.Parameters.AddWithValue("@ID_ALBUM", idAlbum);
                    cmd.Parameters.AddWithValue("@ID_SONG", idSong);
                    cmd.Connection = con;
                    con.Open();
                    return cmd.ExecuteNonQuery();
                }
            }
        }

        public int DeleteTracklist(int idAlbum)
        {
            var tracklist = _context.Tbl_Tracklist.Where(x => x.Id_Album == idAlbum).ToList();
            if (tracklist == null) throw new Exception("No se pudo encontrar el tracklist.");
            _context.Tbl_Tracklist.RemoveRange(tracklist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar el tracklist.");
        }

        public IEnumerable<Tracklist> ReadTracklist(int idAlbum)
        {
            var player = _context.Tbl_Tracklist.Where(x => x.Id_Album == idAlbum)
                                .ToList();
            return player;
        }

        public int SaveTracklist(int idAlbum, List<int> idSongs)
        {
            List<Tracklist> tracklist = new List<Tracklist>();

            foreach (var id in idSongs)
            {
                Tracklist currentTracklist = new Tracklist();
                currentTracklist.Id_Album = idAlbum;
                currentTracklist.Id_Song = id;
                tracklist.Add(currentTracklist);

            }

            _context.Tbl_Tracklist.AddRange(tracklist);

            var cont = _context.SaveChanges();

            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el tracklist.");
        }

        public int SaveTracklist(int idAlbum, int idSong)
        {
            Tracklist tracklist = new Tracklist();
            tracklist.Id_Album = idAlbum;
            tracklist.Id_Song = idSong;
            _context.Add(tracklist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el tracklist.");
        }
    }
}
```

Figura 104 - “TracklistRepository” script

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Album.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

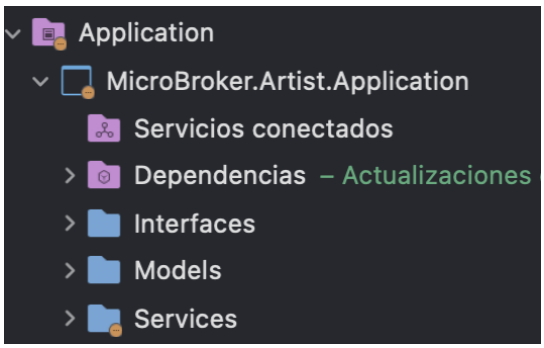


Figura 105 - Directorios principales del proyecto “MicroBroker.Artist.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó las interfaces “IAlbumService” y “ITracklistService” donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior. También se creó la interfaz “ISongServiceRemote” donde se declaró los métodos abstractos para la comunicación sincrónica con el microservicio “Song”.

IAlbumService › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace MicroBroker.Album.Application.Interfaces
8   {
9       public interface IAlbumService
10      {
11          IEnumerable<Domain.Models.Album> ReadAlbum();
12          int SaveAlbum(Domain.Models.Album album);
13          int UpdateAlbum(Domain.Models.Album album);
14          int DeleteAlbum(int id);
15          int CheckExistAlbum(string titleAlbum);
16      }
17  }
```

Figura 106 - “IAlbumService” script

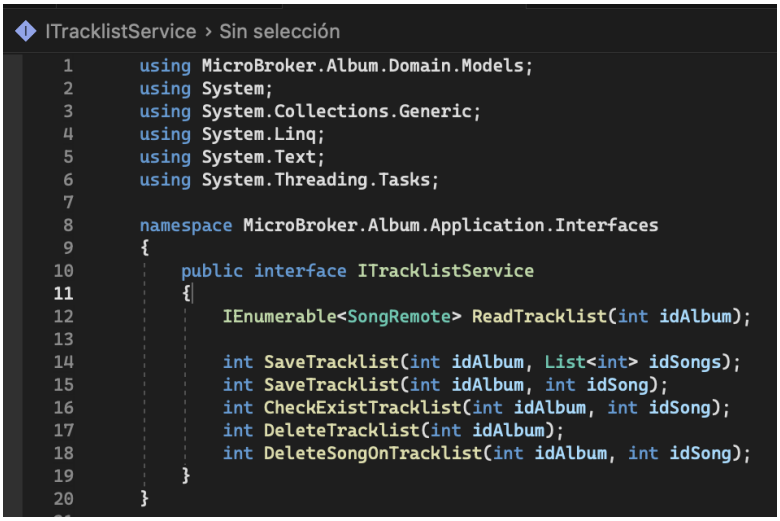


Figura 107 - “ITracklistService” script

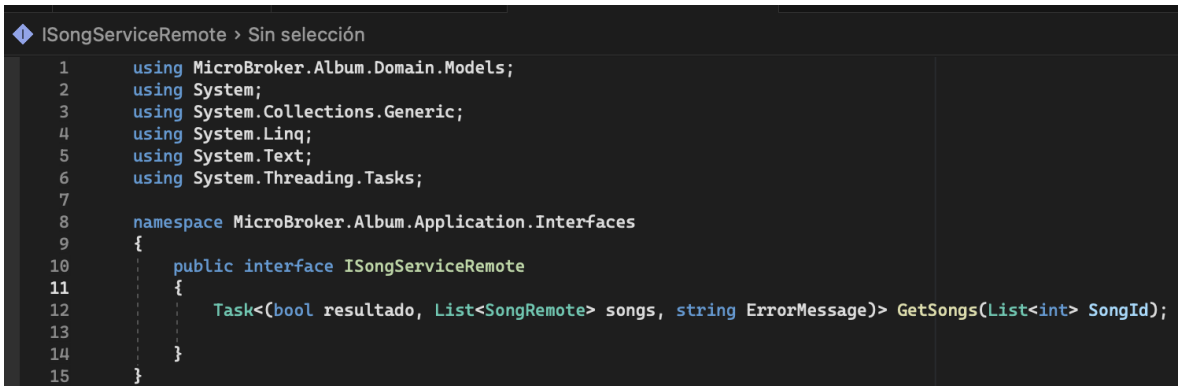


Figura 108 - “ISongServiceRemote” script

El directorio “Services” contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó las clases “AlbumService” y “TracklistService”. “AlbumService” consumirá la interfaz “IAlbumService” y hará uso de “IAlbumRepository” mientras que “TracklistService” consumirá “ITracklistService” y usará las interfaces “ITracklistService” y “ISongServiceRemote”, las interfaces usadas son para la implementación de cada método de servicio. “AlbumService” y “TracklistService” tienen como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio “Album”.

AlbumService › Sin selección

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Album.Application.Services
{
    public class AlbumService:IAlbumService
    {
        private readonly IAlbumRepository _albumRepository;

        //comunicacion microservicios

        private readonly IEventBus _bus;

        public AlbumService(IAlbumRepository albumRepository, IEventBus bus)
        {
            _albumRepository = albumRepository;
            _bus = bus;
        }

        public int CheckExistAlbum(string titleAlbum)
        {
            return _albumRepository.CheckExistAlbum(titleAlbum);

        }

        public int DeleteAlbum(int id)
        {
            return _albumRepository.DeleteAlbum(id);

        }

        public IEnumerable<Domain.Models.Album> ReadAlbum()
        {
            return _albumRepository.ReadAlbum();

        }

        public int SaveAlbum(Domain.Models.Album album)
        {
            return _albumRepository.SaveAlbum(album);

        }

        public int UpdateAlbum(Domain.Models.Album album)
        {
            return _albumRepository.UpdateAlbum(album);

        }
    }
}
```

Figura 109 - “AlbumService” script

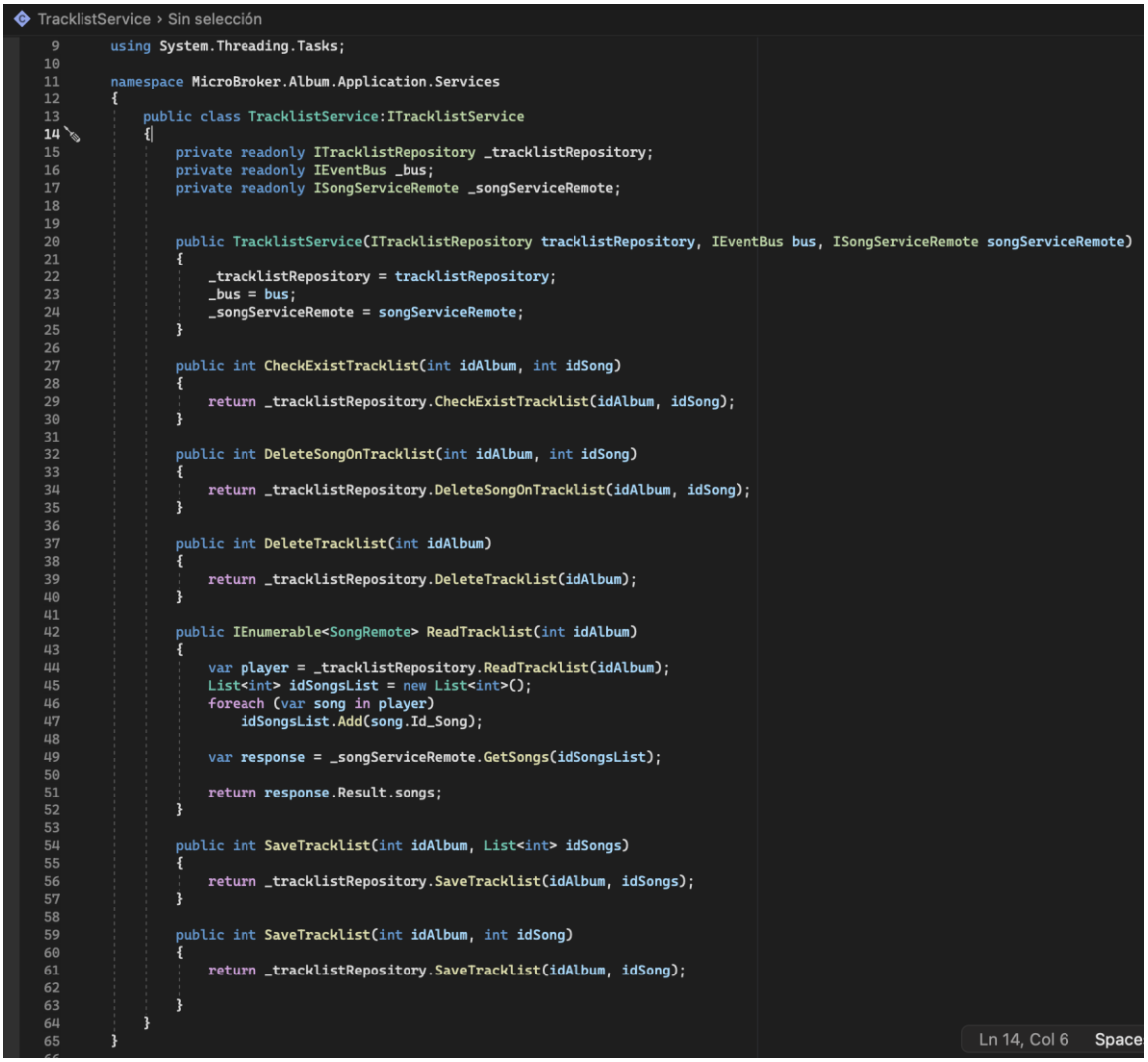


Figura 110 - “TracklistService” script

También se creó la clase “SongServiceRemote”, la cual implementará la comunicación sincrónica mediante protocolo HTTP y haciendo uso de las rutas del microservicio “Song” para obtener los registros solicitados de canciones.

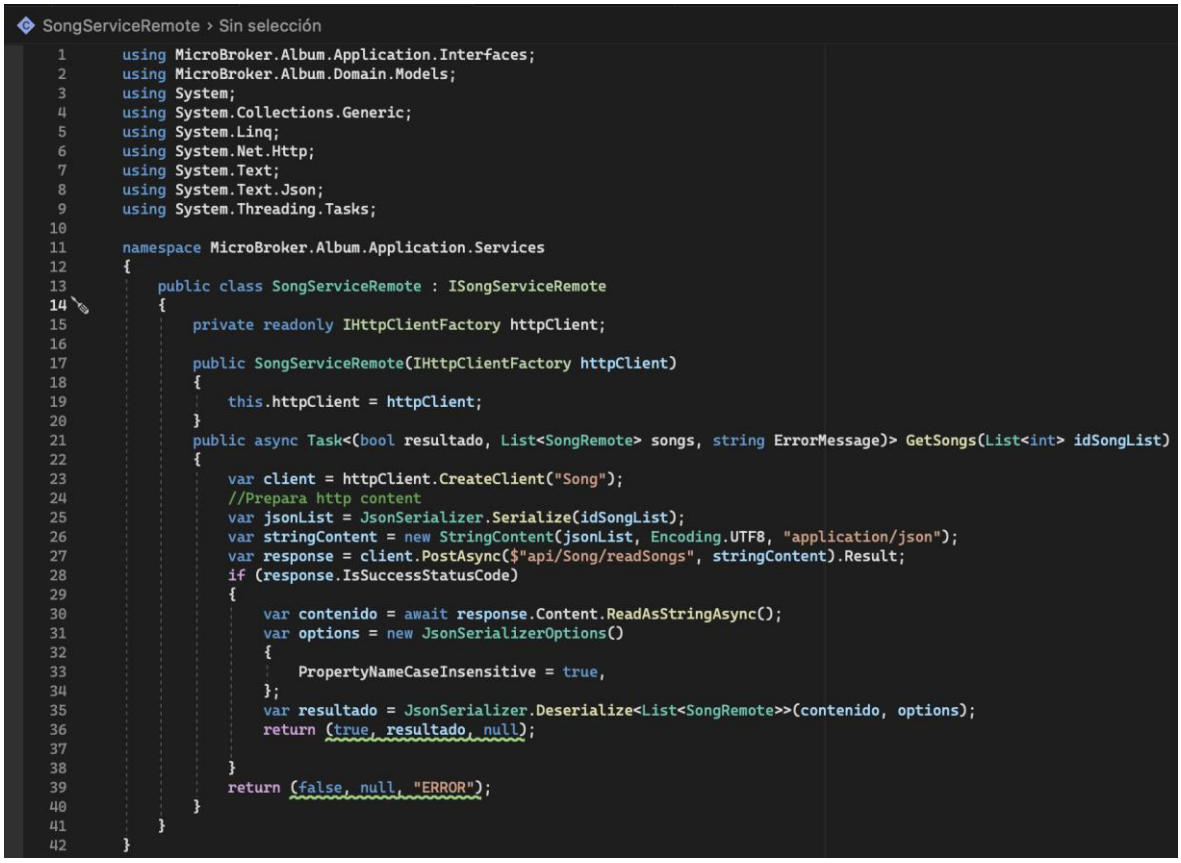


Figura 111 - "SongServiceRemote" script

En el directorio "**Api**", se creó un nuevo proyecto de tipo "Asp.Net Core Web Api" llamado "**MicroBroker.Album.Api**". El proyecto de tipo "Asp.Net Core Web Api" otorga un conjunto de directorios, cada uno con propósito definido.

Figura 112 - Directorios principales del proyecto "MicroBroker.Album.Api"

El directorio "Controllers" contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó las clases "AlbumController" y

“TracklistController”, que llamarán a los métodos generados en los servicios “AlbumService” y “TracklistService” respectivamente, mediante sus interfaces. En las clases se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado.

AlbumController › Sin selección

```
using MicroBroker.Album.Application.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace MicroBroker.Album.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class AlbumController : ControllerBase

    {
        private readonly IAlbumService _albumService;
        public AlbumController(IAlbumService albumService)
        {
            _albumService = albumService;
        }

        [HttpGet]
        public ActionResult<IEnumerable<Domain.Models.Album>> ReadAlbum()
        {
            return Ok(_albumService.ReadAlbum());
        }
        [HttpGet("checkExistAlbum/{albumTitle}")]
        public ActionResult<int> CheckExistAlbum(string albumTitle)
        {
            return Ok(_albumService.CheckExistAlbum(albumTitle));

        }
        [HttpDelete("deleteAlbum/{id}")]
        public IActionResult DeleteAlbum(int id)
        {
            _albumService.DeleteAlbum(id);
            return Ok(id);

        }

        [HttpPost("saveAlbum")]
        public IActionResult SaveAlbum([FromBody] Domain.Models.Album album)

        {
            _albumService.SaveAlbum(album);
            return Ok(album);
        }
        [HttpPut("updateAlbum")]
        public IActionResult UpdateAlbum([FromBody] Domain.Models.Album album)

        {
            _albumService.UpdateAlbum(album);
            return Ok(album);
        }

    }
}
```

Figura 113 - “AlbumController” script

Figura 114 - “TracklistController” script

El directorio “Properties” tendrá el archivo autogenerado “launchSettings.json”. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación del microservicio “Album” desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es usado para la publicación del aplicativo en el servidor de producción, si son necesarias ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo “appsettings.json”.

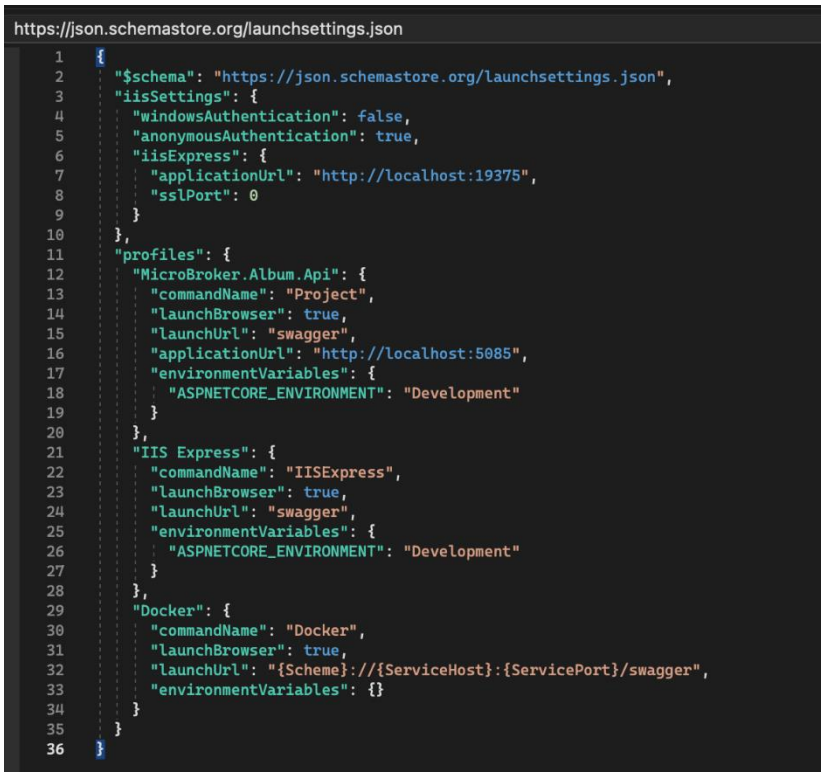


Figura 115 - Archivo de configuración "launchSettings.json" del microservicio "Album".

- **Sprint 4 Review**

Al finalizar con la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU03-T13	Se generó las clases en representación de la entidad "Album" y "Tracklist", y el modelo de negocio en su respectiva interfaz.	Sí
HU03-T14	Se creó las implementaciones de acceso a datos para la tabla de base de datos "TBL_ALBUM" y "TBL_TRACKLIST".	Sí
HU03-T15	Se declaró los servicios e interfaces que funcionen como DTO's.	Sí

HU03-T16	Se creó las APIs REST que ejecutarán los servicios del microservicio “Album”	Sí
HU03-T17	Se creó en la capa de Aplicación la llamada al microservicio “Song” mediante HTTP.	Sí
HU03-T18	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Album". La evidencia de las pruebas con el detalle de estas se las puede observar en el **Anexo 3 subsección 3.6**.	Sí

Tabla 23 - Sprint 4 Review

- **Sprint 4 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Al tener un sprint cumplido con éxito donde hubo mejoras significativas en los tiempos de desarrollo, no existe mejoras evidenciables para el equipo de trabajo.

4.3.1.5 Sprint 5

- **Sprint 5 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Microservicio “Playlist” inserte, actualice, elimine y lea registros de lista de reproducciones del sistema.

- **Sprint 5 Backlog**

Sprint 5 Backlog	
Código	**Nombre**
HU05-T25	Diseñar e implementar la capa de Dominio del microservicio “Playlist” utilizando DDD.
HU05-T26	Diseñar e implementar la capa de Infraestructura del microservicio “Playlist”.
HU05-T27	Diseñar e implementar la capa de Aplicación del microservicio “Playlist”.
HU05-T28	Implementar la capa de Presentación del microservicio “Playlist”.
HU07-T42	Implementar el modelo de eventos en el sistema.
HU05-T29	Integrar el microservicio "Playlist" con otros microservicios.
HU05-T30	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Playlist" mediante la API por solicitudes HTTP.

Tabla 24 - Sprint 5 Backlog

- **Ejecución del Sprint 5**

Para la creación del microservicio “Playlist”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la Tabla 18.

En el directorio **“Domain”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Album.Domain”**. El proyecto contendrá una estructura de directorios. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la “Arquitectura basada de Eventos” en la

comunicación asincrónica, si es necesario en un trabajo futuro. El segundo conjunto de directorios que se creó tiene dos directorios “Interfaces” y “Models”.

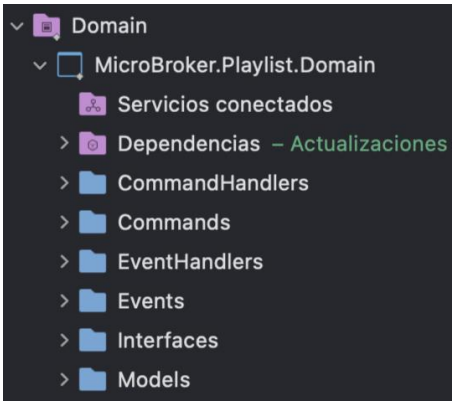


Figura 116 - Directorios principales del proyecto “Microbroker.Album.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará los modelos de las tablas de la base de datos “TBL_PLAYLIST”, “TBL_PLAYLIST_SONG” y también a la tabla externa “TBL_SONG”. Para ello se creó las clases “Playlist”, “PlaylistSong” y “SongRemote” respectivamente, con sus respectivos atributos, constructores, y métodos getters y setters para acceder a sus atributos (ver figuras 17 - 19).

Playlist › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.ComponentModel.DataAnnotations;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace MicroBroker.Playlist.Domain.Models
9   {
10      public class Playlist
11      {
12          [Key]
13          public int Id_Playlist { get; set; }
14          public int Id_User { get; set; }
15          public string Title { get; set; }
16          public DateTime? Creation_Date { get; set; }
17          public int Type { get; set; }
18          public string Photo { get; set; }
19      }
20  }
```

Figura 117 - “Playlist” script

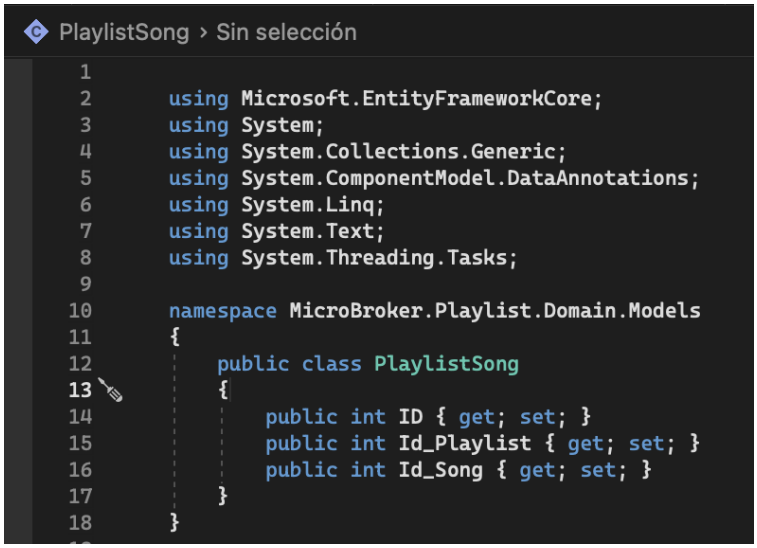


Figura 118 - “PlaylistSong” script

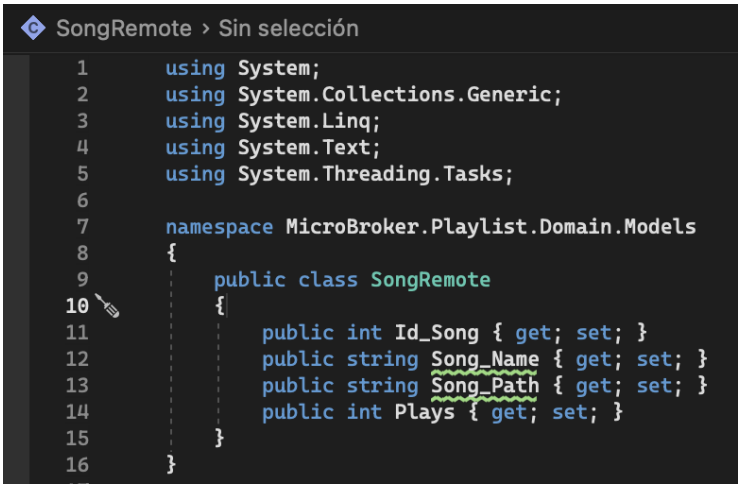


Figura 119 - “SongRemote” script

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó las interfaces “IPlaylistRepository” y “IPlaylistSongRepository”, las cuales tienen declarado todos los métodos abstractos para las operaciones básicas de CRUD.

IPlaylistRepository › Sin selección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Playlist.Domain.Interfaces
{
    public interface IPlaylistRepository
    {
        IEnumerable<Domain.Models.Playlist> ReadPlaylist();
        void AddPlaylist(Domain.Models.Playlist playlist);
        int SavePlaylist(Domain.Models.Playlist playlist);
        int UpdatePlaylist(Domain.Models.Playlist playlist);
        int DeletePlaylist(int id);
        int CheckExistPlaylist(Domain.Models.Playlist playlist);
        IEnumerable<Domain.Models.Playlist> ReadPlaylistsByIdUser(int idUser);
        Domain.Models.Playlist ReadPlaylistByIdUserAndIdPlaylist(int idUser,int idPlaylist);

    }
}
```

Figura 120 - “IPlaylistRepository” script

IPlaylistSongRepository › Sin selección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Playlist.Domain.Interfaces
{
    public interface IPlaylistSongRepository
    {
        IEnumerable<Domain.Models.PlaylistSong> ReadPlaylistSong(int playlistId);

        int SavePlaylistSong(int playlistId, int songId);
        int DeletePlaylistSong(int playlistId);
        int DeletePlaylistSong(int playlistId,int songId);

        int CheckExistPlaylistSong(int playlistId, int songId);

    }
}
```

Figura 121 - “IPlaylistSongRepository” script

En el directorio **“Infrastructure”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Playlist.Infrastructure”**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Context”, “Migrations” y “Repository.

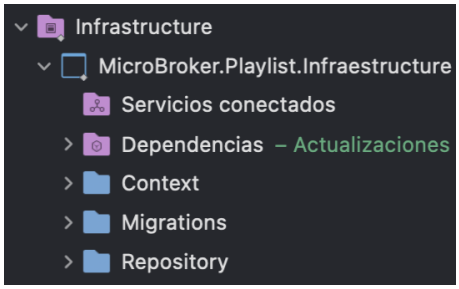


Figura 122 - Directorios principales del proyecto “MicroBroker.Playlist.Infrastructure”

El directorio “Context” contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext. Por ello se creó las clases “PlaylistDbContext”, “PlaylistSongDbContext” que heredarán las propiedades de DbContext y se las usará para consultar y guardar instancias de las entidades. Es decir, “PlaylistDbContext”, “PlaylistSongDbContext” convertirán las clases de C# “Playlist”, “PlaylistSong” a las entidades “TBL_PLAYLIST”, “TBL_PLAYLIST_SONG” respectivamente, para la persistencia de registros en la base de datos “Playlist”.

PlaylistDbContext › Sin selección

```
1   using Microsoft.EntityFrameworkCore;
2
3
4
5   namespace MicroBroker.Playlist.Infraestructure.Context
6   {
7       public class PlaylistDbContext : DbContext
8       {
9
10          public PlaylistDbContext(DbContextOptions<PlaylistDbContext> options) : base(options) { }
11          public DbSet<Domain.Models.Playlist> Tbl_Playlist { get; set; }
12      }
13  }
```

Figura 123 - “PlaylistDbContext” script

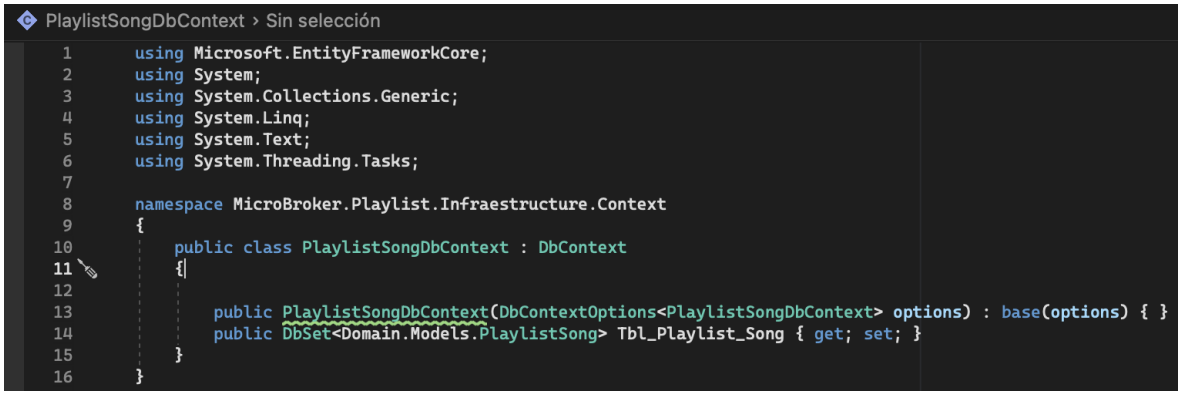


Figura 124 - “PlaylistSongDbContext” script

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. Las clases “PlaylistDbContextModelSnapshot” y “PlaylistSongDbContextModelSnapshot” que están dentro del directorio fue autogeneradas gracias a Entity Framework Core.

PlaylistDbContextModelSnapshot › Sin selección

```
// <auto-generated />
using MicroBroker.Playlist.Infraestructure.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace MicroBroker.Playlist.Data.Migrations
{
    [DbContext(typeof(PlaylistDbContext))]
    partial class PlaylistDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.8")
                .HasAnnotation("Relational:MaxIdentifierLength", 128);

            SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder, 1L, 1);

            modelBuilder.Entity("MicroBroker.Playlist.Domain.Models.Playlist", b =>
                {
                    b.Property<int>("ID")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("ID"), 1L, 1);

                    b.Property<string>("CreationDate")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<int>("IdUser")
                        .HasColumnType("int");

                    b.Property<string>("Photo")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("Title")
                        .IsRequired()
                        .HasColumnType("nvarchar(max)");

                    b.Property<int>("Type")
                        .HasColumnType("int");

                    b.HasKey("ID");

                    b.ToTable("Playlist");
                });
#pragma warning restore 612, 618
        }
    }
}
```

Figura 125 - “PlaylistDbContextModelSnapshot” script

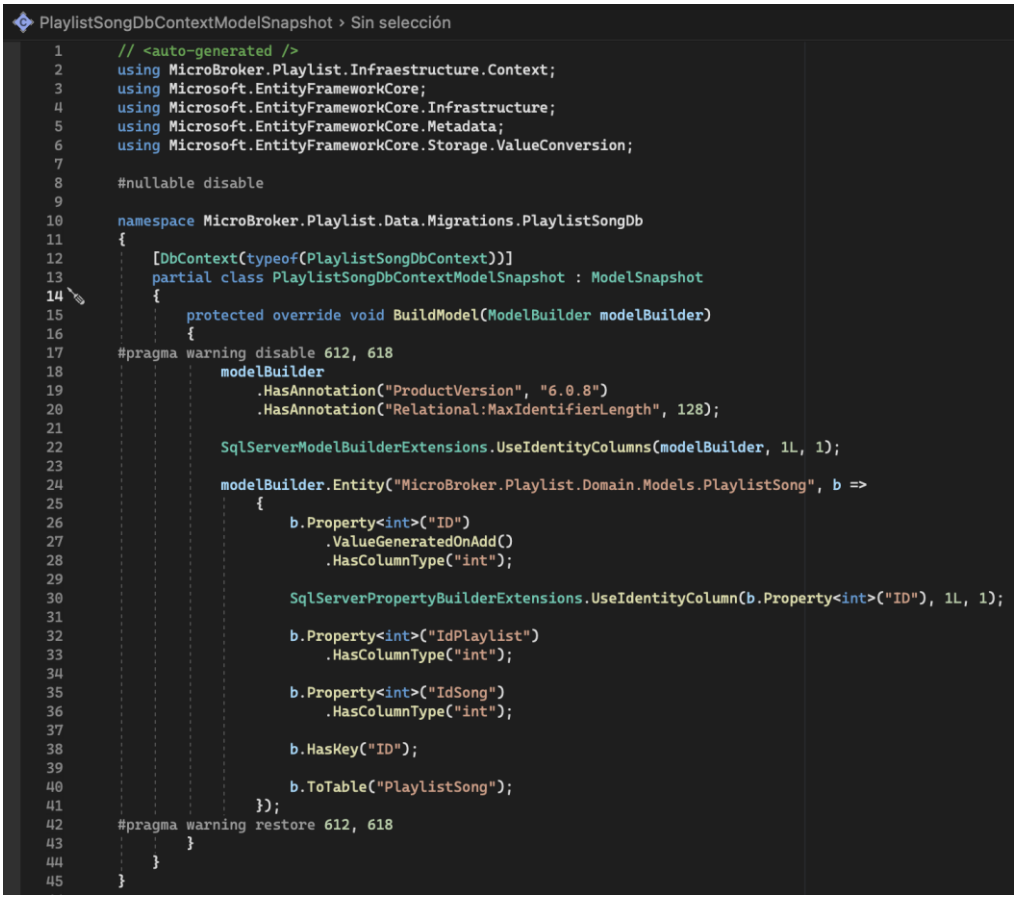


Figura 126 - “PlaylistSongDbContextModelSnapshot” script

Es decir, para que “AlbumDbContextModelSnapshot” y “PlaylistSongDbContextModelSnapshot” se generen, se hizo uso de la “Consola de Gestión de Paquetes” de Visual Studio, donde se ejecutó los siguientes comandos:

```
add-migration PlaylistDbContextModel
```

```
add-migration PlaylistSongDbContextModel
```

El directorio “Repository” contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó las clases “PlaylistRepository” y “PlaylistSongRepository”, donde se instanciarán las clases “PlaylistDbContext” y “PlaylistSongDbContext” respectivamente, que permitirán

consultar, crear, editar y eliminar registros de artistas y también relacionar las canciones con un artista específico, todo ello en la base de datos “Playlist”.

PlaylistRepository › Sin selección

```
namespace MicroBroker.Playlist.Infraestructure.Repository
{
    public class PlaylistRepository :Domain.Interfaces.IPlaylistRepository
    {
        private PlaylistDbContext _context;

        public PlaylistRepository(PlaylistDbContext context)
        {
            _context = context;
        }
        public IEnumerable<Domain.Models.Playlist> ReadPlaylist()
        {
            return _context.Tbl_Playlist;
        }
        public IEnumerable<Domain.Models.Playlist> ReadPlaylistsByIdUser(int idUser)
        {
            var playlists =  _context.Tbl_Playlist.Where(x => x.Id_User == idUser).ToList();

            if (playlists == null)
            {
                throw new Exception("No se encontro ninguna playlist");
            }
            return playlists;
        }
        public Domain.Models.Playlist ReadPlaylistByIdUserAndIdPlaylist(int idUser,int idPlaylist)
        {
            var playlist =  _context.Tbl_Playlist.Where(x => x.Id_User == idUser)
                  .Where(x => x.Id_Playlist == idPlaylist).FirstOrDefault();

            if (playlist == null)
            {
                throw new Exception("No se encontro la playlist");
            }
            return playlist;
        }

        //I-EJEMPLO DE COMUNICACION ENTRE MICROSERVICIOS.....
        public void AddPlaylist(Domain.Models.Playlist playlist)
        {
            _context.Add(playlist);
            _context.SaveChanges();
        }
        //F-EJEMPLO DE COMUNICACION ENTRE MICROSERVICIOS.....
        public int SavePlaylist(Domain.Models.Playlist playlist)
        {
            _context.Add(playlist);
            var cont =_context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar la playlist.");
        }
        public int UpdatePlaylist(Domain.Models.Playlist request)
        {
            var playlist =  _context.Tbl_Playlist.Where(x => x.Id_User == request.Id_User)
                  .Where(x => x.Id_Playlist == request.Id_Playlist).FirstOrDefault();
            if (playlist == null) throw new Exception("No se pudo encontrar la playlist.");


            playlist.Title = request.Title != null && request.Title != string.Empty ? request.Title : playlist.Title;
            playlist.Type = request.Type;
            playlist.Photo = request.Photo != null && request.Photo != string.Empty ? request.Photo : playlist.Photo;
            _context.Tbl_Playlist.Update(playlist);
            var cont =  _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo actualizar la playlist.");
        }
        public int DeletePlaylist(int id)
        {
            var playlist =  _context.Tbl_Playlist.Where(x => x.Id_Playlist == id)
                        .FirstOrDefault();
            if (playlist == null) throw new Exception("No se pudo encontrar la playlist.");
            _context.Tbl_Playlist.Remove(playlist);
            var cont =  _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar la playlist.");
        }
        public int CheckExistPlaylist(Domain.Models.Playlist request)
        {
            var playList = _context.Tbl_Playlist.Where(x => x.Id_User == request.Id_User)
                    .Where(x => x.Title == request.Title)
                            .FirstOrDefault();

            return playList==null?0 : playList.Id_Playlist;
        }
    }
}
```

Figura 127 - “PlaylistRepository” script

Figura 128 - “PlaylistSongRepository” script

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Playlist.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

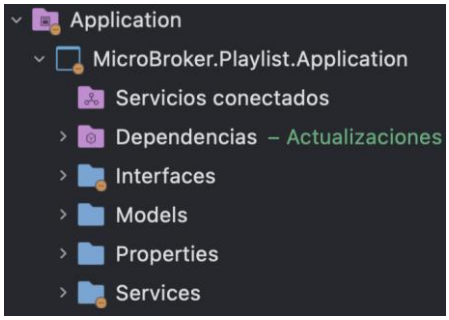


Figura 129 - Directorios principales del proyecto “MicroBroker.Playlist.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó las interfaces “IPlaylistService” y “IPlaylistSongService” donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior. También se creó la interfaz “ISongServiceRemote” donde se declaró los métodos abstractos para la comunicación sincrónica con el microservicio “Song”.

IPlaylistService › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace MicroBroker.Playlist.Application.Interfaces
8   {
9       public interface IPlaylistService
10      {
11          IEnumerable<Domain.Models.Playlist> ReadPlaylist();
12          int SavePlaylist(Domain.Models.Playlist playlist);
13          int UpdatePlaylist(Domain.Models.Playlist playlist);
14          int DeletePlaylist(int id);
15          int CheckExistPlaylist(Domain.Models.Playlist playlist);
16          IEnumerable<Domain.Models.Playlist> ReadPlaylistsByIdUser(int idUser);
17          Domain.Models.Playlist ReadPlaylistByIdUserAndIdPlaylist(int idUser,int idPlaylist);
18
19      }
20  }
```

Figura 130 - “IPlaylistService” script

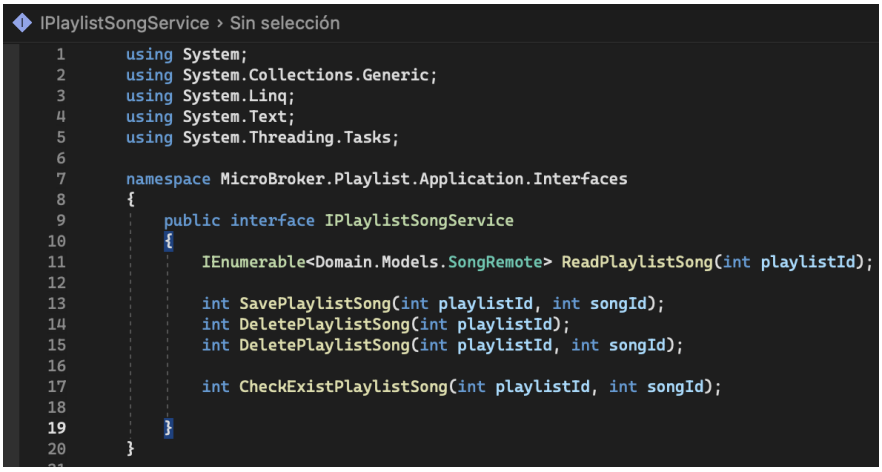


Figura 131 - "IPlaylistSongService" script

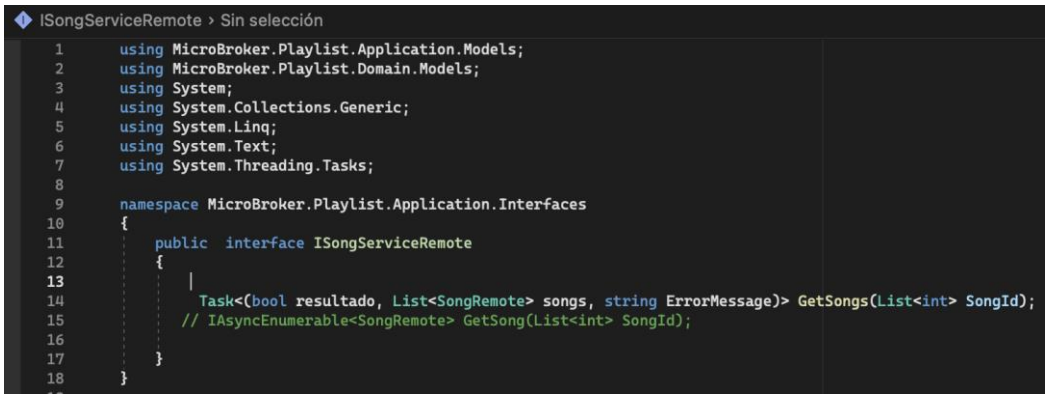


Figura 132 - "ISongServiceRemote" script

El directorio "Services" contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó las clases "PlaylistService" y "PlaylistSongService". "PlaylistService" consumirá la interfaz "IPlaylistService" y hará uso de "IPlaylistRepository" mientras que "PlaylistSongService" consumirá "IPlaylistSongService" y usará las interfaces "IPlaylistSongService" y "ISongServiceRemote", las interfaces usadas son para la implementación de cada método de servicio. "PlaylistService" y "PlaylistSongService" tienen como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio "Playlist".

Figura 133 - “PlaylistService” script

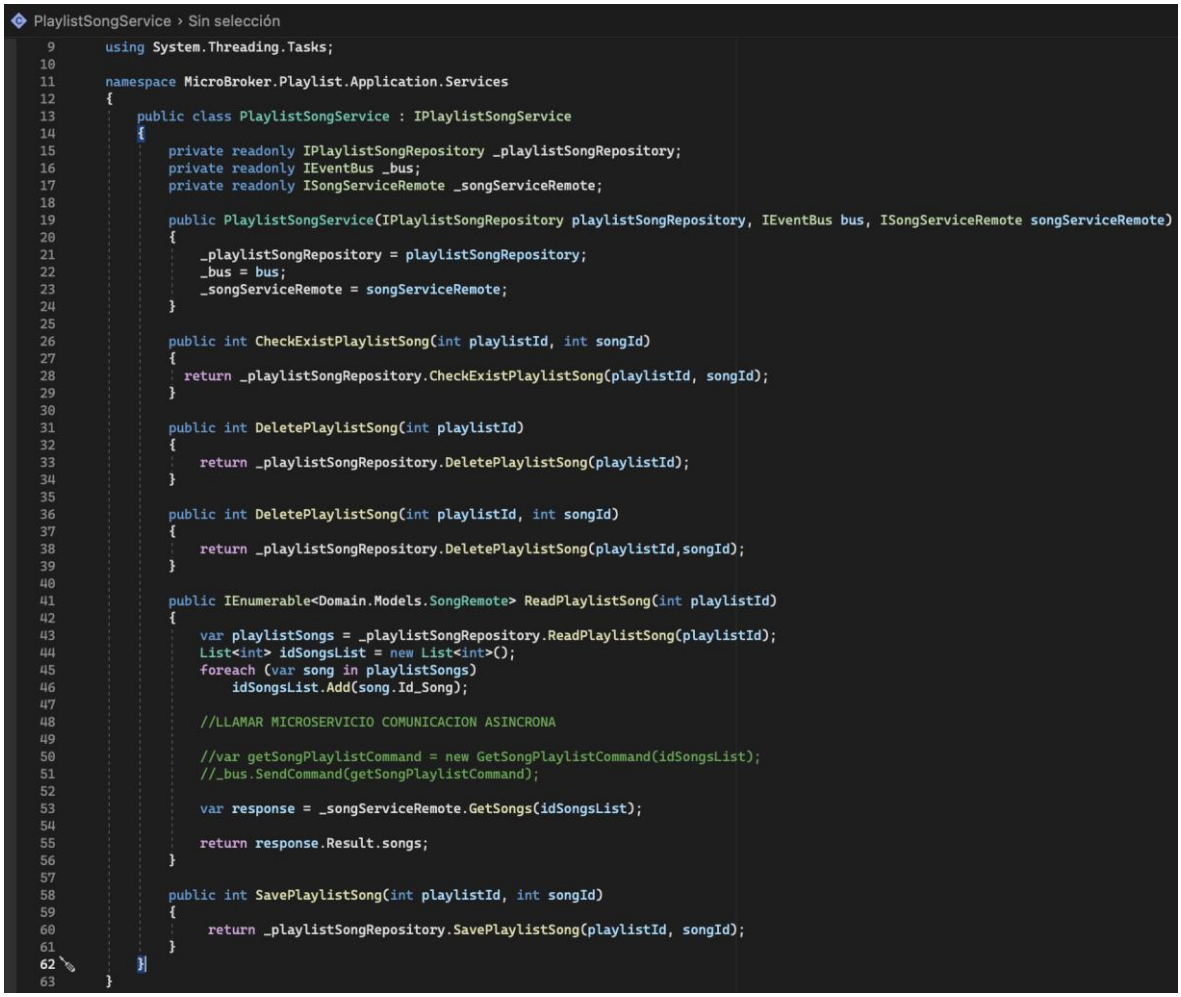


Figura 134 - “PlaylistSongService” script

También se creó la clase “SongServiceRemote”, la cual implementará la comunicación sincrónica mediante protocolo HTTP y haciendo uso de las rutas del microservicio “Song” para obtener los registros solicitados de canciones.

SongServiceRemote › Sin selección

```
using MicroBroker.Playlist.Application.Interfaces;
using MicroBroker.Playlist.Application.Models;
using MicroBroker.Playlist.Domain.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http.Headers;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

namespace MicroBroker.Playlist.Application.Services
{
    public  class SongServiceRemote :ISongServiceRemote
    {
        private readonly IHttpClientFactory httpClient;

        public SongServiceRemote(IHttpClientFactory httpClient)
        {
            this.httpClient = httpClient;
        }

          async Task<(bool resultado, List<SongRemote> songs, string ErrorMessage)> ISongServiceRemote.GetSongs(List<int> idSongList)
        {
            var client = httpClient.CreateClient("Song");
            //Prepara http content
            var jsonList = JsonSerializer.Serialize(idSongList);
            var stringContent = new StringContent(jsonList, Encoding.UTF8, "application/json");
            var response = client.PostAsync($"api/Song/readSongs", stringContent).Result;
            if (response.IsSuccessStatusCode)
            {
                var contenido = await response.Content.ReadAsStringAsync();
                var options = new JsonSerializerOptions()
                {
                    PropertyNameCaseInsensitive = true,
                };
                var resultado = JsonSerializer.Deserialize<List<SongRemote>>(contenido, options);
                return (true,resultado,null);

            }
            return (false, null, "ERROR");

        }
    }
}
```

Figura 135 - “SongServiceRemote” script

En el directorio “**Api**”, se creó un nuevo proyecto de tipo “Asp.Net Core Web Api” llamado “**MicroBroker.Playlist.Api**”. El proyecto de tipo “Asp.Net Core Web Api” otorga un conjunto de directorios, cada uno con propósito definido.

Figura 136 - Directorios principales del proyecto “MicroBroker.Playlist.Api”

El directorio “Controllers” contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó las clases “PlaylistController” y “PlaylistSongController”, que llamarán a los métodos generados en los servicios “PlaylistService” y “PlaylistSongService” respectivamente, mediante sus interfaces.

En las clases se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado.


```
namespace MicroBroker.Playlist.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PlaylistController:ControllerBase
    {
        private readonly IPlaylistService _playlistService;

        public PlaylistController(IPlaylistService playlistService)
        {
            _playlistService = playlistService;
        }
    [HttpGet]
    public ActionResult<IEnumerable<Domain.Models.Playlist>> Get()
        {
            return Ok(_playlistService.ReadPlaylist());
        }

        [HttpPost("savePlaylist")]
        public IActionResult SavePlaylist([FromBody] Domain.Models.Playlist playlist)

        {
            _playlistService.SavePlaylist(playlist);
            return Ok(playlist);
        }

        [HttpPut("updatePlaylist")]
        public IActionResult UpdatePlaylist([FromBody] Domain.Models.Playlist playlist)
        {
            _playlistService.UpdatePlaylist(playlist);
            return Ok(playlist);
        }

        [HttpDelete("deletePlaylist/{id}")]
        public IActionResult DeletePlaylist(int id)
        {
            _playlistService.DeletePlaylist(id);
            return Ok(id);

        }

        [HttpPost("checkExistPlaylist")]
        public IActionResult CheckExistPlaylist([FromBody] Domain.Models.Playlist playlist)
        {
            return Ok(_playlistService.CheckExistPlaylist(playlist));
        }

        [HttpGet("readPlaylist/{idUser}")]
        public ActionResult<IEnumerable<Domain.Models.Playlist>> ReadPlaylistsByIdUser(int idUser)
        {
            return Ok(_playlistService.ReadPlaylistsByIdUser(idUser));
        }

        [HttpGet("readPlaylistsIdUser/{idPlaylist}")]
        public ActionResult<Domain.Models.Playlist> ReadPlaylistByIdUserAndIdPlaylist(int idUser,int idPlaylist)
        {
            return Ok(_playlistService.ReadPlaylistByIdUserAndIdPlaylist(idUser,idPlaylist));
        }
    }
}
```

Figura 137 - “PlaylistController” script

PlaylistSongController › Sin selección

```
using MicroBroker.Playlist.Application.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace MicroBroker.Playlist.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PlaylistSongController : ControllerBase
    {
        private readonly IPlaylistSongService _playlistSongService;

        public PlaylistSongController(IPlaylistSongService playlistSongService)
        {
            _playlistSongService = playlistSongService;
        }

        [HttpPost("readPlaylistSong/{idPlaylist}")]
        public ActionResult<IEnumerable<Domain.Models.SongRemote>> ReadPlaylistSong(int idPlaylist)
        {
            return Ok(_playlistSongService.ReadPlaylistSong(idPlaylist));
        }

        [HttpPost("savePlaylistSong/{idPlaylist}/{idSong}")]
        public IActionResult SavePlaylistSong( int idPlaylist, int idSong)

        {
            return Ok(_playlistSongService.SavePlaylistSong(idPlaylist, idSong));
        }

        [HttpDelete("deletePlaylistSong/{idPlaylist}/{idSong}")]
        public IActionResult DeletePlaylistSong(int idPlaylist,int idSong)
        {
            return Ok(_playlistSongService.DeletePlaylistSong(idPlaylist, idSong));
        }

        [HttpDelete("deletePlaylistSong/{idPlaylist}/")]
        public IActionResult DeletePlaylistSong(int idPlaylist)
        {
            return Ok(_playlistSongService.DeletePlaylistSong(idPlaylist));
        }

        [HttpPost("checkExistPlaylistSong")]
        public IActionResult CheckExistPlaylistSong( int playlistId, int songId)
        {
            return Ok(_playlistSongService.CheckExistPlaylistSong(playlistId, songId));
        }
    }
}
```

Figura 138 - “PlaylistSongController” script

El directorio “Properties” tendrá el archivo autogenerado “launchSettings.json”. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación del microservicio “Playlist” desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es usado para la publicación del aplicativo en el servidor de producción, si son necesarias ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo “appsettings.json”.

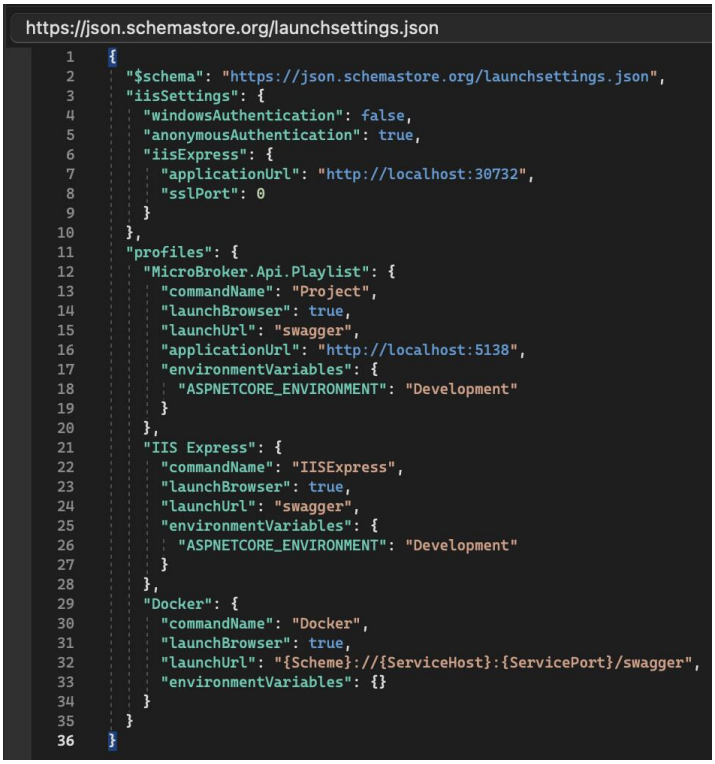


Figura 139 - Archivo de configuración “launchSettings.json” del microservicio “Playlist”.

- **Sprint 5 Review**

Al finalizar con la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU05-T25	Se generó las clases en representación de la entidad “Playlist” y “PlaylistSong”, y el modelo de negocio en su respectiva interfaz.	Sí
HU05-T26	Se creó las implementaciones de acceso a datos para la tabla de base de datos “TBL_PLAYLIST” y “TBL_PLAYLIST_SONG”.	Sí
HU05-T27	Se declaró los servicios e interfaces que funcionen como DTO’s.	Sí

HU05-T28	Se creó las APIs REST que ejecutarán los servicios del microservicio “Playlist”	Sí
HU07-T42	Se implementó los comandos, eventos y controladores de los eventos, en la ejecución de la comunicación asincrónica para la subscripción de eventos proveniente del microservicio “User”.	Sí
HU05-T29	Se creó en la capa de Aplicación la llamada al microservicio “Playlist” mediante HTTP.	Sí
HU05-T30	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Playlist". La evidencia de las pruebas con el detalle se las puede observar en el **Anexo 3 subsección 3.3**.	Sí

Tabla 25 - Sprint 5 Review

- **Sprint 5 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Después de la experticia conseguida del microservicio “Playlist”, donde se incluyó una implementación del modelo de eventos para Azure Service Bus desde la perspectiva de un subscriptor y también una integración con otro microservicio, se prevé agilizar y mejorar los tiempos de desarrollo para los siguientes microservicios en cuanto a comunicación asincrónica.

4.3.1.6 Sprint 6

- **Sprint 6 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Microservicio “Catalog” inserte, actualice, elimine y lea registros de catálogos del sistema.

- **Sprint 6 Backlog**

Sprint 6 Backlog	
Código	**Nombre**
HU06-T31	Diseñar e implementar la capa de Dominio del microservicio “Catalog” utilizando DDD.
HU06-T32	Diseñar e implementar la capa de Infraestructura del microservicio “Catalog”.
HU06-T33	Diseñar e implementar la capa de Aplicación del microservicio “Catalog”.
HU06-T34	Implementar la capa de Presentación del microservicio “Catalog”.
HU06-T35	Integrar el microservicio "Catalog" con otros microservicios.
HU06-T36	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Catalog" mediante la API por solicitudes HTTP.

Tabla 26 - Sprint 6 Backlog

- **Ejecución del Sprint 6**

Para la creación del microservicio “Catalog”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la

funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la Tabla 18.

En el directorio **“Domain”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Catalog.Domain”**. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la “Arquitectura basada de Eventos” en la comunicación asincrónica, si es necesario en un trabajo futuro. El segundo conjunto de directorios que se creó tiene dos directorios “Interfaces” y “Models”.

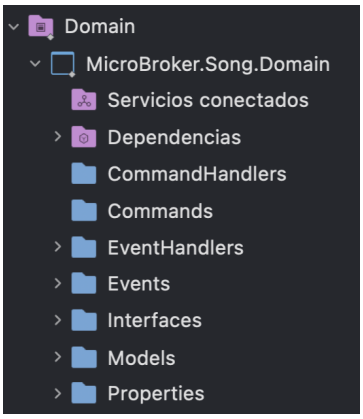


Figura 140 - Directorios principales del proyecto “MicroBroker.Catalog.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará el modelo de la tabla de la base de datos “TBL_CATALOG”. Para ello se creó la clase “Catalog”, con sus respectivos atributos, constructores, y métodos getters y setters para acceder a sus atributos (ver figura 141).

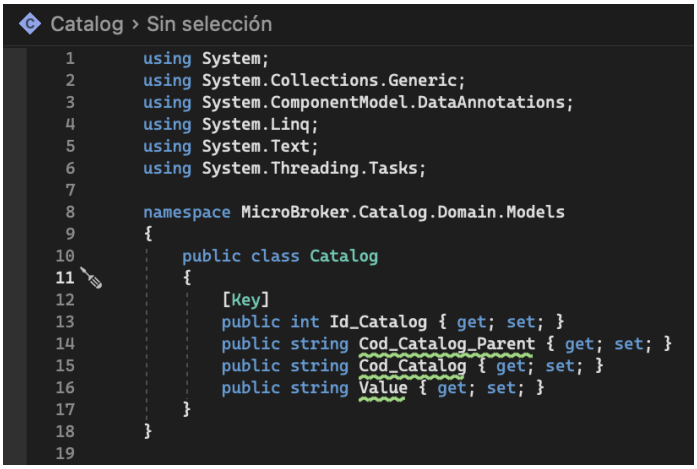


Figura 141 - “Catalog” script

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó la interfaz “ICatalogRepository”, la cual tiene declarado todos los métodos abstractos para las operaciones básicas de CRUD.

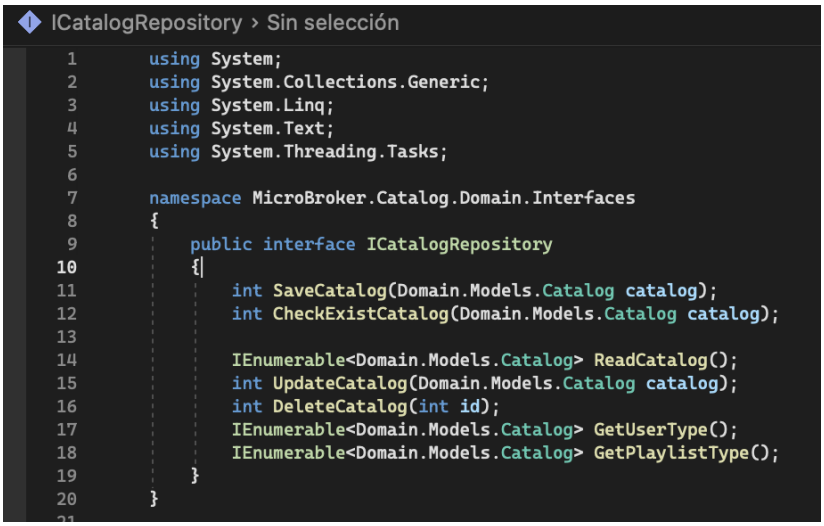


Figura 142 - “ICatalogRepository” script

En el directorio **“Infrastructure”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Catalog.Infrastructure”**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Context”, “Migrations” y “Repository.

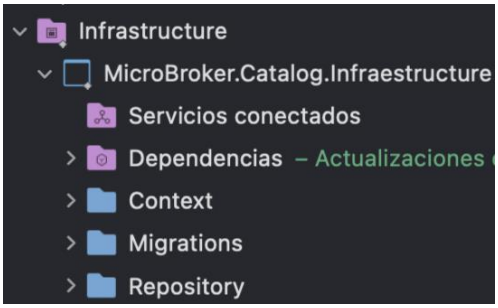


Figura 143 - Directorios principales del proyecto “MicroBroker.Catalog.Infrastructure”

El directorio “Context” contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext.

Por ello se creó la clase “CatalogDbContext” que heredará las propiedades de DbContext y se la usará para consultar y guardar instancias de las entidades. Es decir, “CatalogDbContext” convertirá la clase de C# “Catalog” a una entidad “TBL_SONG”, para la persistencia de registros en la base de datos “Catalog”.

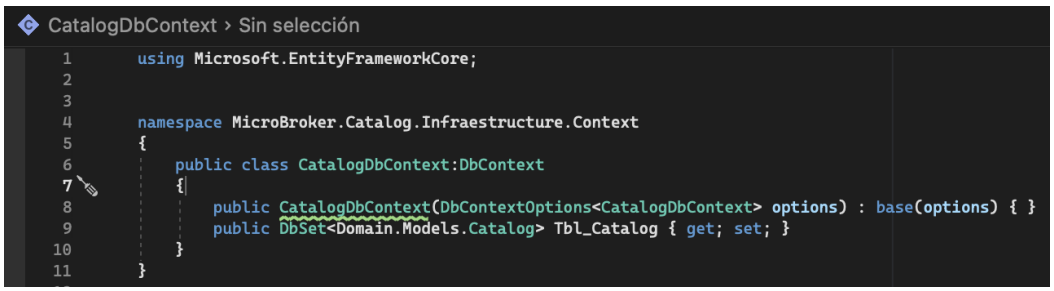


Figura 144 - “CatalogDbContext” script

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. La clase “CatalogDbContextModelSnapshot” que está dentro del directorio fue autogenerada gracias a Entity Framework Core.

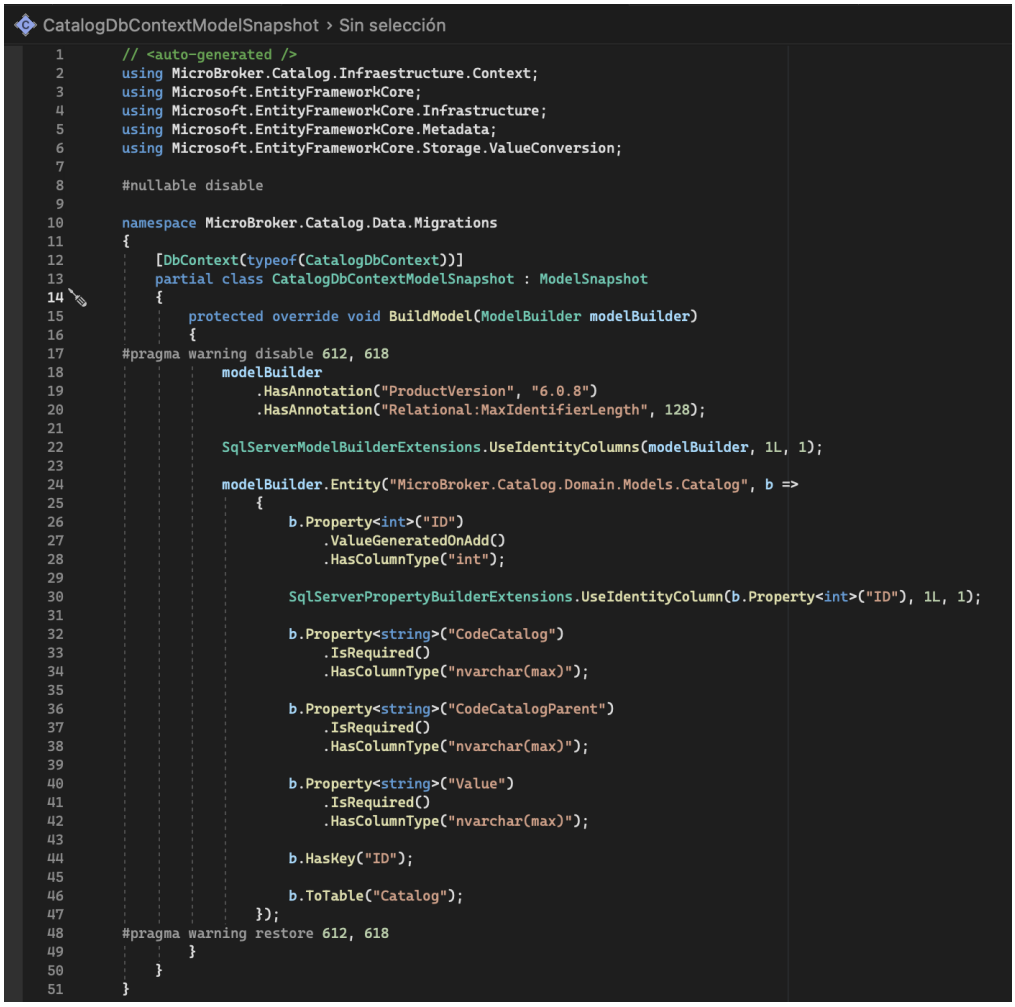


Figura 145 - “CatalogDbContextModelSnapshot” script

Es decir, para que “CatalogDbContextModelSnapshot” se genere, se hizo uso de la “Consola de Gestión de Paquetes” de Visual Studio, donde se ejecutó el comando:

```
add-migration CatalogDbContextModel
```

El directorio “Repository” contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó la clase “CatalogRepository”, donde se instanciará la clase “CatalogDbContext” que permitirá consultar, crear, editar y eliminar registros de canciones en la base de datos “Catalog”.


```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Catalog.Infraestructure.Repository
{
    public class CatalogRepository:Domain.Interfaces.ICatalogRepository
    {
        private CatalogDbContext _context;

        public CatalogRepository(CatalogDbContext context)
        {
            _context = context;
        }

        public int CheckExistCatalog(Domain.Models.Catalog request)
        {
            var catalog = _context.Tbl_Catalog.Where(x => x.Cod_Catalog_Parent == request.Cod_Catalog_Parent)
                    .Where(x => x.Cod_Catalog == request.Cod_Catalog)
                    .Where(x => x.Value == request.Value)
                    .FirstOrDefault();

            return catalog == null?0:catalog.Id_Catalog;
        }

        public int DeleteCatalog(int id)
        {
            var playlist = _context.Tbl_Catalog.Where(x => x.Id_Catalog == id)
                            .FirstOrDefault();
            if (playlist == null) throw new Exception("No se pudo encontrar el catalogo.");
            _context.Tbl_Catalog.Remove(playlist);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo eliminar el catalogo.");
        }

        public IEnumerable<Domain.Models.Catalog> GetPlaylistType()
        {
            var catalogs = _context.Tbl_Catalog.Where(x => x.Cod_Catalog_Parent == Constants.ID_PLAYLIST_TYPE_CATALOG.ToString())
                .ToList();
            if (catalogs == null)
            {
                throw new Exception("No se encontro el catalog");
            }
            return  catalogs;
        }

        public IEnumerable<Domain.Models.Catalog> GetUserType()
        {
            var catalogs = _context.Tbl_Catalog.Where(x => x.Cod_Catalog_Parent == Constants.ID_USER_TYPE_CATALOG.ToString())
                 .ToList();
            if (catalogs == null)
            {
                throw new Exception("No se encontro el catalog");
            }
            return catalogs;
        }

        public IEnumerable<Domain.Models.Catalog> ReadCatalog()
        {
            return _context.Tbl_Catalog;
        }

        public int SaveCatalog(Domain.Models.Catalog catalog)
        {
            _context.Add(catalog);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo insertar el catalogo.");
        }

        public int UpdateCatalog(Domain.Models.Catalog request)
        {
            var catalog = _context.Tbl_Catalog.Where(x => x.Id_Catalog == request.Id_Catalog)
                    .FirstOrDefault();
            if (catalog == null) throw new Exception("No se pudo encontrar el catalogo.");

            catalog.Cod_Catalog_Parent = request.Cod_Catalog_Parent != null &&
                request.Cod_Catalog_Parent != string.Empty ? request.Cod_Catalog_Parent : catalog.Cod_Catalog_Parent;

            catalog.Cod_Catalog = request.Cod_Catalog != null && request.Cod_Catalog != string.Empty ? request.Cod_Catalog : catalog.Cod_Catalog;
            catalog.Value = request.Value != null && request.Value != string.Empty ? request.Value : catalog.Value;
            _context.Tbl_Catalog.Update(catalog);
            var cont = _context.SaveChanges();
            if (cont > 0) return cont;
            throw new Exception("No se pudo actualizar el catalogo.");
        }
    }
}
```

Figura 146 - “CatalogRepository” script

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.Catalog.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

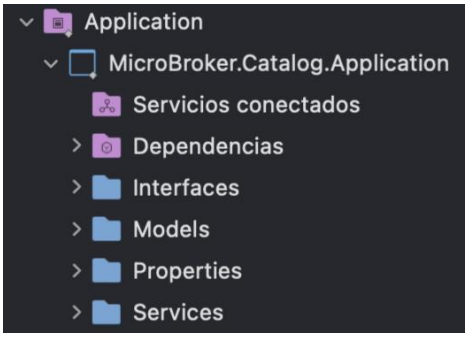


Figura 147 - Directorios principales del proyecto “MicroBroker.Catalog.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó la interfaz “ICatalogService”, donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior.

ICatalogService › Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace MicroBroker.Catalog.Application.Interfaces
8   {
9       public interface ICatalogService
10      {
11          int SaveCatalog(Domain.Models.Catalog catalog);
12          int CheckExistCatalog(Domain.Models.Catalog catalog);
13
14          IEnumerable<Domain.Models.Catalog> ReadCatalog();
15          int UpdateCatalog(Domain.Models.Catalog catalog);
16          int DeleteCatalog(int id);
17          IEnumerable<Domain.Models.Catalog> GetUserType();
18          IEnumerable<Domain.Models.Catalog> GetPlaylistType();
19      }
20  }
21
```

Figura 148 - “ICatalogService” script

El directorio “Services” contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó la clase “CatalogService”, que consumirá la interfaz antes creada “ICatalogService” y además hará uso de la interfaz “ICatalogRepository” para la implementación de cada método del servicio. “CatalogService” tiene como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio “Catalog”.


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MicroBroker.Catalog.Application.Services
{
    public class CatalogService : ICatalogService
    {
        private readonly ICatalogRepository _catalogRepository;
        private readonly IEventBus _bus;

        public CatalogService(ICatalogRepository catalogRepository, IEventBus bus)
        {
            _catalogRepository = catalogRepository;
            _bus = bus;
        }

        public int CheckExistCatalog(Domain.Models.Catalog catalog)
        {
            return _catalogRepository.CheckExistCatalog(catalog);

        }

        public int DeleteCatalog(int id)
        {
            return _catalogRepository.DeleteCatalog(id);

        }

        public IEnumerable<Domain.Models.Catalog> GetPlaylistType()
        {
            return _catalogRepository.GetPlaylistType();

        }

        public IEnumerable<Domain.Models.Catalog> GetUserType()
        {
            return _catalogRepository.GetUserType();

        }

        public IEnumerable<Domain.Models.Catalog> ReadCatalog()
        {
            return _catalogRepository.ReadCatalog();

        }

        public int SaveCatalog(Domain.Models.Catalog catalog)
        {
            return _catalogRepository.SaveCatalog(catalog);

        }

        public int UpdateCatalog(Domain.Models.Catalog catalog)
        {
            return _catalogRepository.UpdateCatalog(catalog);

        }
    }
}
```

Figura 149 - “CatalogService” script

En el directorio “**Api**”, se creó un nuevo proyecto de tipo “Asp.Net Core Web Api” llamado “**MicroBroker.Song.Api**”. El proyecto de tipo “Asp.Net Core Web Api” otorga un conjunto de directorios, cada uno con propósito definido.

Figura 150 - Directorios principales del proyecto “MicroBroker.Song.Api”

El directorio “Controllers” contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó la clase “CatalogController”, que llamará a los métodos generados en el servicio “CatalogService”. En la clase se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado.

Figura 151 - “CatalogController” script

El directorio “Properties” tendrá el archivo autogenerado “launchSettings.json”. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación del microservicio “Catalog” desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es usado para la publicación del aplicativo en el servidor de producción, si son necesarias ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo “appsettings.json”.

Figura 152 - Archivo de configuración “launchSettings.json” del microservicio “Catalog”.

- **Sprint 6 Review**

Terminada la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU06-T31	Se generó las clases en representación de la entidad “Catalog” y el modelo de negocio en su respectiva interfaz.	Sí
HU06-T32	Se creó las implementaciones de acceso a datos para la tabla de base de datos “TBL_SONG”.	Sí

HU06-T33	Se declaró los servicios e interfaces que funcionen como DTO’s.	Sí
HU06-T34	Se creó las APIs REST que ejecutarán los servicios del microservicio “Catalog”	Sí
HU06-T35	No existe criterio de aceptación, debido a que el microservicio “Catalog” no depende de comunicación para su funcionamiento.	-
HU06-T36	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "Catalog". La evidencia de las pruebas se las puede observar en el **Anexo 3 subsección 3.4**.	Sí

Tabla 27 - Sprint 6 Review

- **Sprint 6 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Al tener un sprint cumplido con éxito donde hubo mejoras significativas en los tiempos de desarrollo, no existe mejoras evidenciables para el equipo de trabajo.

4.3.1.7 Sprint 7

- **Sprint 7 Planning**

En este Sprint se planea realizar las historias de usuario que permiten que el Servicio “Email” envíe un correo electrónico personalizado a un usuario que se acaba de registrar.

- **Sprint 7 Backlog**

Sprint 7 Backlog	
Código	**Nombre**
HU08-43	Definir las tecnologías y lenguaje de desarrollo para la incorporación del servicio de email.
HU08-44	Diseñar e implementar el servicio de email utilizando una arquitectura de microservicios.
HU08-45	Implementar la funcionalidad de personalización de correos electrónicos.
HU08-46	Integrar el servicio de email con el sistema de registro de usuarios.
HU07-T42	Implementar el modelo de eventos en el sistema.
HU08-47	Realizar pruebas automatizadas con la herramienta POSTMAN, para comprobar la funcionalidad del servicio de "Email" la mediante la API por solicitudes HTTP.

Tabla 28 - Sprint 7 Backlog

- **Ejecución del Sprint 7**

Para ello, el equipo creará un nuevo proyecto en el directorio "src" y establecerá la estructura de carpetas "api", "config" e "infra.bus". Además, se instalarán las dependencias necesarias para el proyecto.

Dentro de la carpeta "Api", se crearán una nueva carpeta llamada email que contendrá 3 archivos de tipo “ts” que serán nombrados "controller", "router" y "templates".

Dentro de la carpeta "config", se crearán los archivos con la extensión .ts que serán nombrados "Environment", "Logger", "Mailer" y "Swagger".

Por último, dentro de la carpeta "Infra.bus", se creará el archivo para la comunicación por Azure Service Bus.

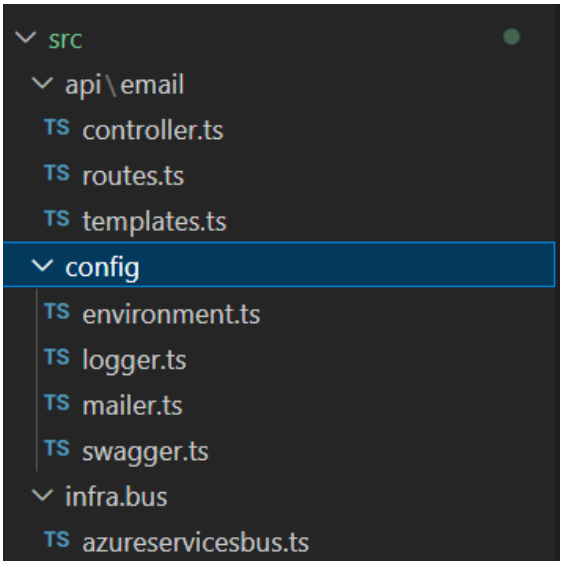


Figura 153 - Directorios principales del proyecto “MicroMailer.bytes_music”

```
"dependencies": {
  "@azure/service-bus": "^7.7.3",
  "@types/winston": "^2.4.4",
  "compression": "^1.7.4",
  "concurrently": "^7.1.0",
  "cors": "^2.8.5",
  "express": "^4.17.1",
  "express-validator": "^6.12.1",
  "helmet": "^4.6.0",
  "morgan": "^1.10.0",
  "nodemailer": "^6.7.0",
  "swagger-jsdoc": "^6.0.1",
  "swagger-ui-express": "^4.1.6",
  "winston": "^3.8.2"
},
```

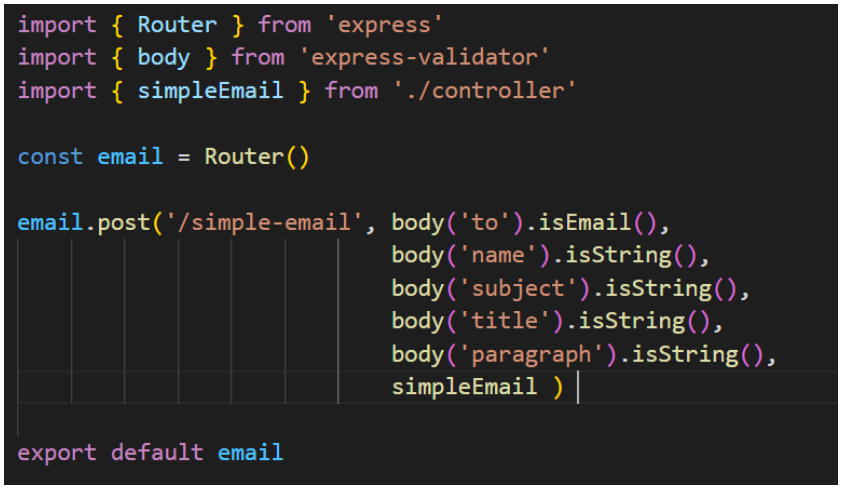
Figura 154 - Dependencias utilizadas en el proyecto “MicroMailer.bytes_music”

Se debe implementar el código necesario para establecer los endpoints básicos del microservicio “Email”. En particular, se creó un archivo llamado "routes.ts" en la carpeta "Api/email" que se encarga de configurar la ruta para el envío de correos electrónicos simples. Este archivo importa el módulo "Router" de Express, así como las funciones "body" y "simpleEmail" del controlador correspondiente.

El controlador "simpleEmail" es el encargado de procesar la solicitud de envío de correo electrónico simple. Primero, se validan los parámetros de entrada utilizando

la función "validationResult" de Express y se retorna un error 400 si alguno de los parámetros no cumple con los requisitos de validación. Luego, se extraen los parámetros necesarios de la solicitud y se utiliza la función "simpleTemplate" para crear el cuerpo del correo electrónico en formato HTML. Finalmente, se llama a la función "_simpleEmail" para enviar el correo electrónico y se devuelve una respuesta indicando que el correo electrónico fue enviado correctamente.

El controlador "_simpleEmail" es el encargado de enviar el correo electrónico utilizando el objeto "transporter" de la configuración de Nodemailer. Este controlador recibe los parámetros necesarios para enviar el correo electrónico (email, subject y html) y los utiliza para construir el objeto de correo electrónico que será enviado. Si ocurre algún error durante el envío del correo electrónico, se registra en el log del sistema mediante el objeto "logger" de la configuración de la aplicación.



```
import { Router } from 'express'
import { body } from 'express-validator'
import { simpleEmail } from './controller'

const email = Router()

email.post('/simple-email', body('to').isEmail(),
                            body('name').isString(),
                            body('subject').isString(),
                            body('title').isString(),
                            body('paragraph').isString(),
                            simpleEmail )

export default email
```

Figura 155 - “Routes.ts” script

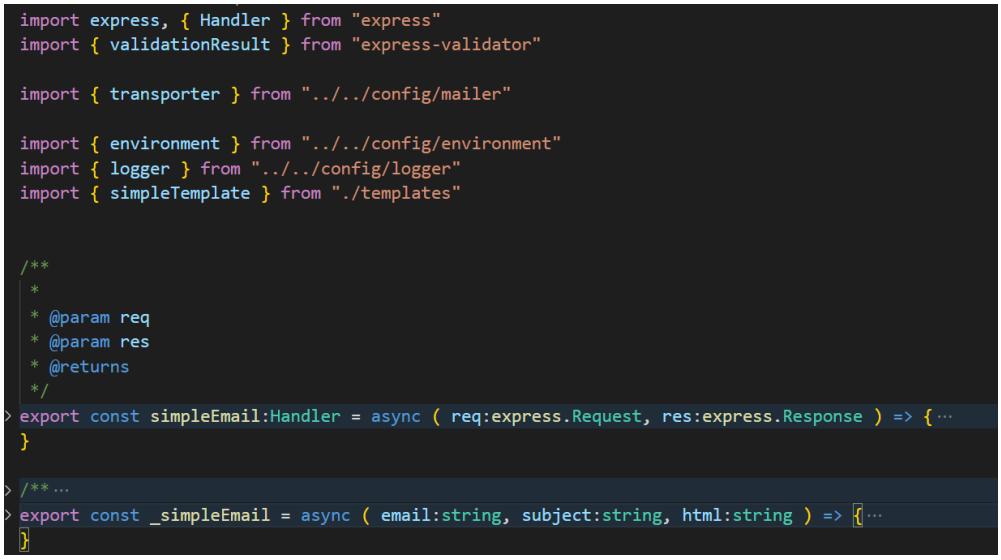


Figura 156 - "Controller.ts" script

Después se implementaron los archivos necesarios para configurar el servicio del "Email". En particular, se creó un archivo llamado "environment.ts" en la carpeta "Config" que contiene las variables de entorno necesarias para el correcto funcionamiento del servicio. En este archivo, se definen variables como el entorno de ejecución (producción o desarrollo), el puerto en el que se ejecuta el servicio, la duración de los tokens de autenticación, las credenciales de correo electrónico, el nombre de la cola de Azure Service Bus, entre otras.

Además, se creó un archivo llamado "mailer.ts" en la misma carpeta que se encarga de configurar el objeto "transporter" de Nodemailer utilizando las credenciales de correo electrónico definidas en el archivo "environment.ts". Este objeto se utiliza para enviar correos electrónicos desde el servicio de "Email".

También se creó el archivo "swagger.ts" para definir las opciones de configuración de Swagger para la documentación del API. En este archivo se especifica la versión de OpenAPI, el título y descripción del servicio, y la URL del servidor donde se encuentra el servicio, entre otros.

Por último, se implementó el archivo "logger.ts" que configura el logger de Winston para registrar los mensajes de información y errores en un archivo "logs.log". Además, se agregó una opción para mostrar los mensajes en la consola durante el desarrollo del servicio. Con la implementación de estos archivos de configuración,

el servicio del “Email” puede ser configurado fácilmente según las necesidades del entorno de ejecución en el que se encuentre. Además, la separación de la configuración en archivos distintos permite una mayor modularidad y facilidad para realizar cambios en la configuración sin afectar otras partes del servicio.

```ts
export const environment = {
    NODE_ENV: 'development',   // production | development
    PORT: 3001,
    TOKEN_EXPIRES_IN:'24h',
    // MAIL_USER:"franklindbruiz@gmail.com", // Correo gmail con verificacion de dos pasos activada
    // MAIL_PASS:"CAMBIAR", // Generar contraseña de aplicaciones de google para enviar correos
    MAIL_USER:"franklindbruiz@gmail.com", // Correo gmail con verificacion de dos pasos activada
    MAIL_PASS:"lqaelxbtqrprykat",
    QUEUE_NAME:"emailuserloginsendedevent",
    QUEUE_CONNECTIONSTRING: "Endpoint=sb://namespacebytesmusic.servicebus.windows.net/;SharedAccessKe
    APP_NAME:'micromailer.bytes_music',
    version:"1.0.0"
  }
```

Figura 157 - “Enviroment.ts” script

```ts
import { createLogger, format, transports } from 'winston'

const { combine } = format

export const logger = createLogger({
    level: 'info',
    format: combine(
      format.json(),
      format.timestamp({
        format: 'YYYY-MM-DD HH:mm:ss'
      }),
      format.printf(info => `${info.timestamp} ${info.level}: ${info.message}`+(info.splat!==undefine
    ),
    defaultMeta: { service: 'micromailer' },
    transports: [
      new transports.File({
        filename: `src/logs.log`,
        level: 'info'
      })
    ],
  })

  if (process.env.NODE_ENV !== 'production') {
    logger.add(new transports.Console({
      format: format.simple(),
    }))
```

Figura 158 - “Logger.ts” script

```
config > mailer.ts > ...
1   import nodemailer from 'nodemailer';
2   import { environment } from "../config/environment"
3
4   export const transporter = nodemailer.createTransport({
5       host: "smtp.gmail.com",
6       port: 465,
7       secure: true, // true for 465, false for other ports.
8       auth: {
9         user: environment.MAIL_USER, // generated ethereal user
10        pass: environment.MAIL_PASS, // generated ethereal password
11      },
12  })
13
14  transporter.verify(() => {
15      console.log('Ready for send emails')
16  })
```

Figura 159 - “Mailer.ts” script

```
config > swagger.ts > options > definition > servers
import { environment } from "./environment";

export const options = {
    definition:{
        openapi:"1.0.0",
        info:{
            title:"Microservicio EMAIL",
            version:"1.0.0",
            decription:"Microservicio encargado de la gestion de correos electronicos",
        },
        servers:[
            {
                url:`http://localhost:${environment.PORT}`
            }
        ]
    },
    apis:[

        "./src/api/email/*.ts",
    ]
}
```

Figura 160 - “Swagger.ts” script

Durante la creación de plantillas de correo electrónico, se implementó el código necesario para crear las plantillas de correo electrónico que se utilizarán en el servicio de “Email”. Para ello, en el archivo templetes creado en la carpeta "api/email”. Contiene la función "simpleTemplate" encargada de generar la plantilla HTML para el envío de correos electrónicos simples.

La plantilla creada consta de una tabla HTML que contiene diferentes secciones para el encabezado, el cuerpo y el pie del correo electrónico. La función "simpleTemplate" recibe tres parámetros: el nombre del destinatario, el título del

correo electrónico y el cuerpo del correo electrónico en formato de párrafo. Estos parámetros se utilizan para rellenar los diferentes campos de la plantilla y generar un correo electrónico personalizado para cada destinatario.

Con esta plantilla implementada, los usuarios podrán enviar correos electrónicos simples con un formato predefinido que asegura que los correos electrónicos enviados desde el servicio de “Email” sigan una estética uniforme y profesional.

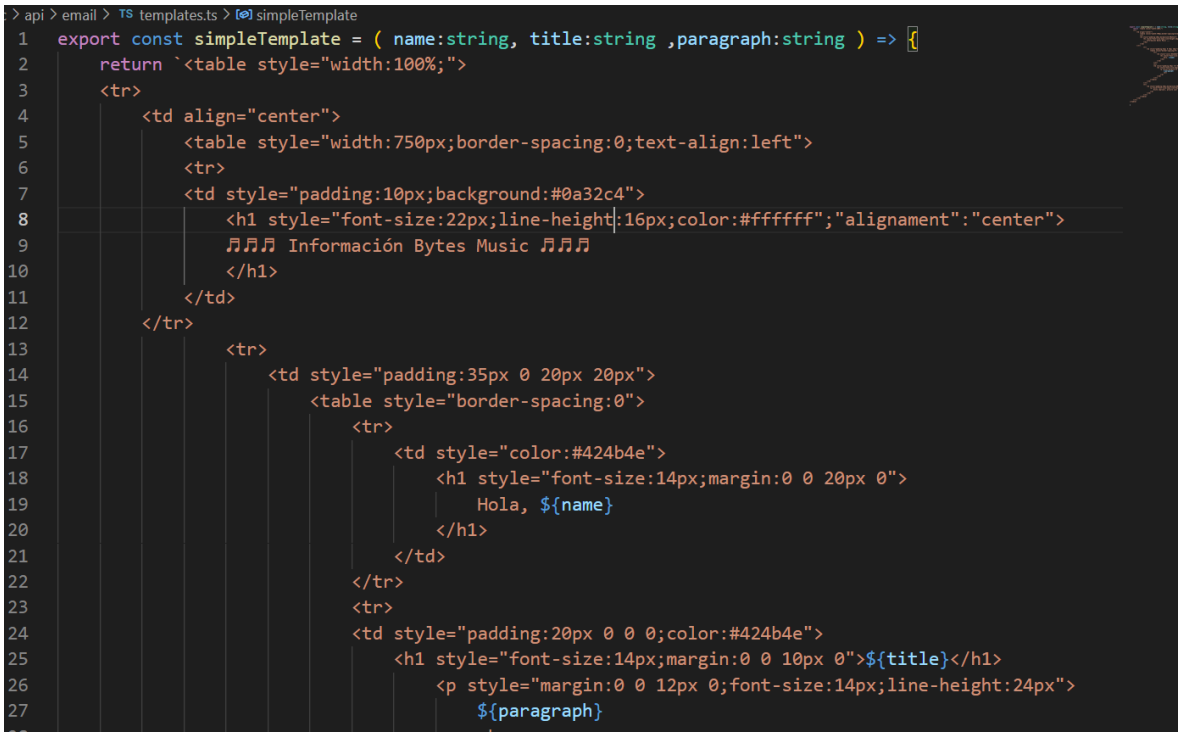


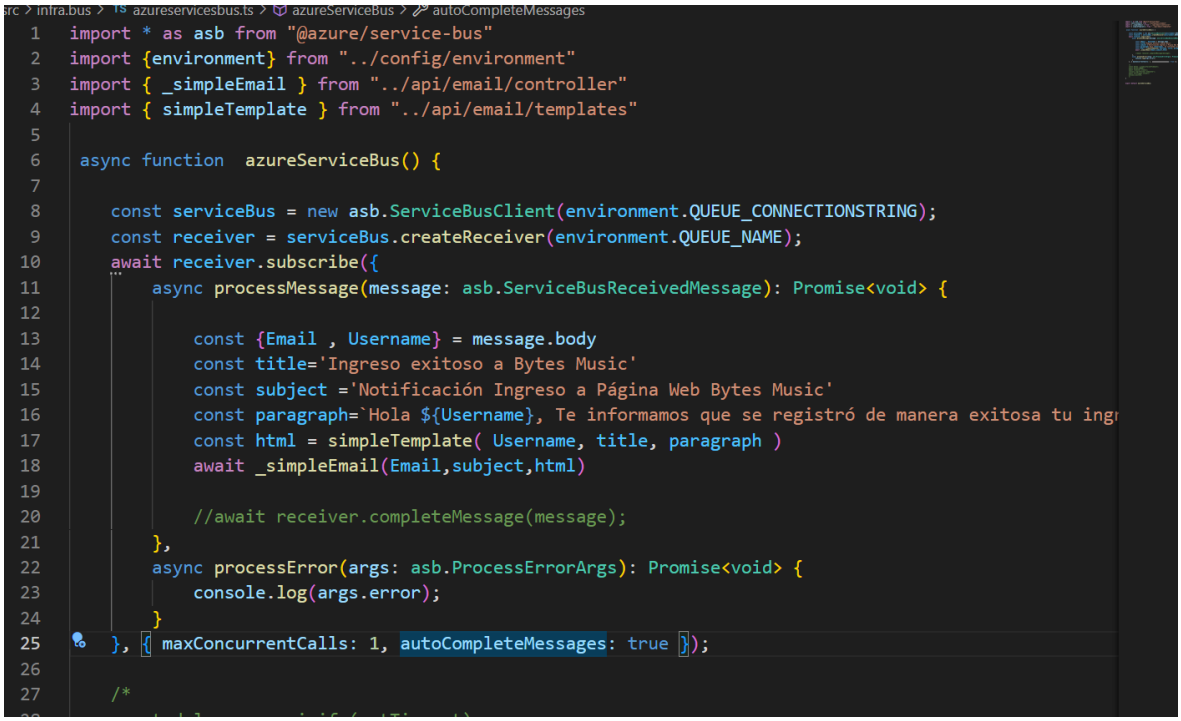
Figura 161 - “templates.ts” script

Durante el Sprint de Comunicación por Azure Service Bus, se implementó el código necesario para establecer la comunicación entre el servicio de “Email” y Azure Service Bus. Para ello, se creó una nueva función llamada "azureServiceBus" en un archivo aparte. Esta función utiliza el módulo "@azure/service-bus" para crear una conexión con el Service Bus de Azure y establecer un receptor para recibir los mensajes enviados a una cola específica.

Cuando un mensaje es recibido, la función "processMessage" se encarga de procesar el mensaje y extraer los datos necesarios. En este caso, se extraen los datos de correo electrónico y nombre de usuario de un objeto JSON incluido en el

cuerpo del mensaje. Luego, se utiliza la función "simpleTemplate" para generar el cuerpo del correo electrónico y se llama a la función "_simpleEmail" para enviar el correo electrónico.

Una vez enviado el correo electrónico, se completa el mensaje para que sea eliminado de la cola de Azure Service Bus y se registra cualquier error en la función "processError".



```
src > infra.bus > azureservicesbus.ts > azureServiceBus > autoCompleteMessages
1  import * as asb from "@azure/service-bus"
2  import {environment} from "../config/environment"
3  import { _simpleEmail } from "../api/email/controller"
4  import { simpleTemplate } from "../api/email/templates"
5
6  async function  azureServiceBus() {
7
8      const serviceBus = new asb.ServiceBusClient(environment.QUEUE_CONNECTIONSTRING);
9      const receiver = serviceBus.createReceiver(environment.QUEUE_NAME);
10     await receiver.subscribe({
11         async processMessage(message: asb.ServiceBusReceivedMessage): Promise<void> {
12
13             const {Email , Username} = message.body
14             const title='Ingreso exitoso a Bytes Music'
15             const subject ='Notificación Ingreso a Página Web Bytes Music'
16             const paragraph=`Hola ${Username}, Te informamos que se registró de manera exitosa tu ingr
17             const html = simpleTemplate( Username, title, paragraph )
18             await _simpleEmail(Email,subject,html)
19
20             //await receiver.completeMessage(message);
21         },
22         async processError(args: asb.ProcessErrorArgs): Promise<void> {
23             console.log(args.error);
24         }
25     }, { maxConcurrentCalls: 1, autoCompleteMessages: true });
26
27     /*
```

Figura 162 - “azureservicebus.ts” script

- **Sprint 7 Review**

Al finalizar con la ejecución del sprint se cumplió con la planificación correctamente, a continuación, se muestra un resumen de los criterios de aceptación en cada historia de usuario realizada:

Código	Criterio de Aceptación	Cumplido
HU08-43	Se seleccionó Express.js como tecnología para la implementación del servicio de email y Typescript como lenguaje de programación para el desarrollo	Sí

HU08-44	Se diseñó una arquitectura de microservicios que utiliza API para la comunicación entre los diferentes componentes y se implementó el servicio de email utilizando dicha arquitectura, siguiendo buenas prácticas y estándares de desarrollo	Sí
HU08-45	Se ha creado un nuevo script que incluye los campos personalizables del correo electrónico, de acuerdo con los requerimientos y limitaciones del proyecto y se ha implementado la funcionalidad de personalización de correos electrónicos utilizando el template "simpleTemplate" y el controlador "_simpleEmail" en dicho script.	Sí
HU08-46	No existe criterio de aceptación, debido a que el microservicio “Email” no depende de comunicación para su funcionamiento.	-
HU07-T42	Se ha definido el modelo de eventos a utilizar en el sistema, se ha implementado la funcionalidad de eventos utilizando Azure Service Bus como el sistema de mensajería, se ha creado una clase llamada "AzureServiceBus" para el manejo de los eventos siguiendo buenas prácticas y estándares de desarrollo, y se han actualizado las configuraciones necesarias para integrar el modelo de eventos en el sistema utilizando la clase "AzureServiceBus".	Sí
HU08-47	Se realizó pruebas automatizadas gracias a la herramienta POSTMAN, para comprobar la funcionalidad del microservicio "Email". La evidencia de las pruebas con el detalle se las puede observar en el **Anexo 3 subsección 3.7**.	Sí

Tabla 29 - Sprint 7 Review

- **Sprint 7 Retrospective**

    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes pruebas comprobando su funcionalidad con la herramienta SWAGGER.

    - **¿Qué se puede mejorar?**

Integrar con el microservicio “User”, para comprobar su funcionamiento haciendo uso de Azure Service Bus. Además, se podría incorporar patrones de diseño correspondiente a la Arquitectura basada en Eventos.

4.3.1.8 Sprint 8

- **Sprint 8 Planning**

En este Sprint planea realizar las historias de usuario que permiten la implementación del microservicio “User” con su funcionalidad de inserción, actualización, eliminación y lectura de registros de usuarios. La creación del microservicio se proyectó realizar bajo los patrones de diseño arquitectónico “Arquitectura Limpia”.

- **Sprint 8 Backlog**

Sprint 8 Backlog	
Código	**Nombre**
HU01-T01	Diseñar e implementar la capa de Dominio del microservicio “User” utilizando DDD.
HU01-T02	Diseñar e implementar la capa de Infraestructura del microservicio “User”.
HU01-T03	Diseñar e implementar la capa de Aplicación del microservicio “User”.

HU01-T04	Implementar la capa de Presentación del microservicio “User”.
HU07-T42	Implementar el modelo de eventos en el sistema.
HU01-T05	Integrar el microservicio "User" con otros microservicios.
HU01-T06	Realizar pruebas automatizadas con la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "User" mediante la API por solicitudes HTTP.

Tabla 30 - Sprint 8 Backlog

- **Ejecución del Sprint 8**

Para la creación del microservicio “User”, se creó un directorio con el mismo nombre dentro del directorio “Microservices”. Además, para el desarrollo del microservicio bajo el patrón de arquitectura de software conocido como “Arquitectura Limpia”, se creó una estructura de directorios correspondientes a la funcionalidad que manejará el proyecto dentro de cada uno de ellos basado en este patrón y que se encuentra detallada en la Tabla 18.

En el directorio **“Domain”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.User.Domain”**. El proyecto contendrá una estructura de directorios. El primer conjunto de directorios que se creó contendrá la lógica de entidades para la implementación de la “Arquitectura basada de Eventos” en la comunicación asincrónica, segundo conjunto de directorios que se creó tiene dos directorios “Interfaces” y “Models”.

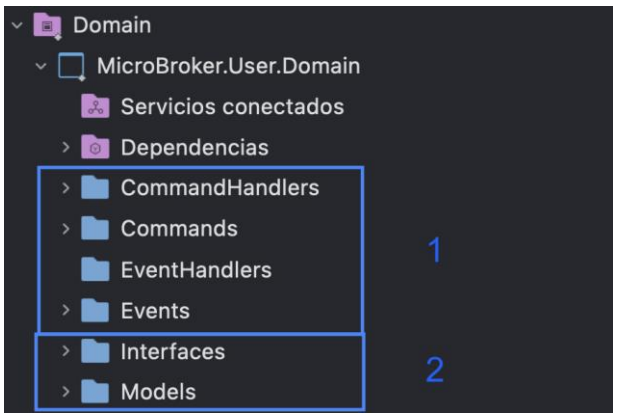


Figura 163 - Directorios principales del proyecto “Microbroker.User.Domain”

El directorio “Models” contendrá los scripts que hacen referencia a las entidades que representará el modelo de la tabla de la base de datos “User”. Para ello se creó la clase “User”, con sus respectivos atributos, constructores, y métodos getters y setteres para acceder a sus atributos (ver figura 164).

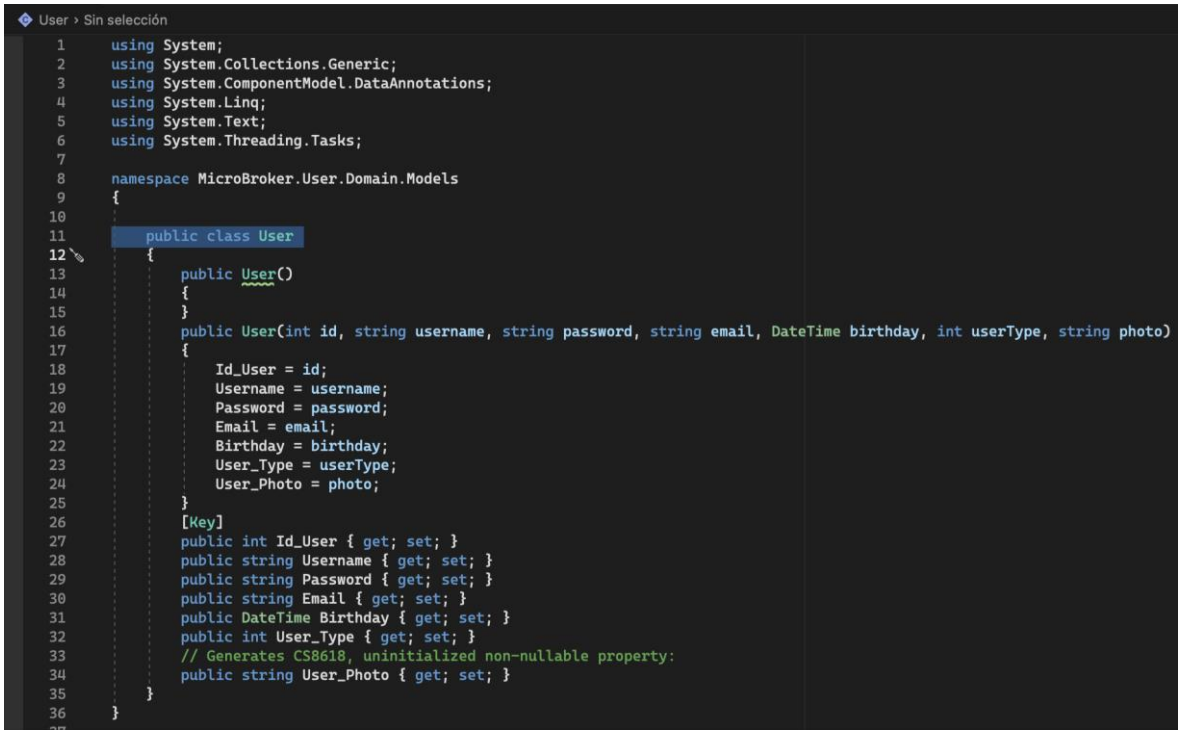


Figura 164 - “User” script.

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso. Por ello se creó la interfaz “IUserRepository”, la cual tiene declarado todos los métodos abstractos para las operaciones básicas de CRUD.

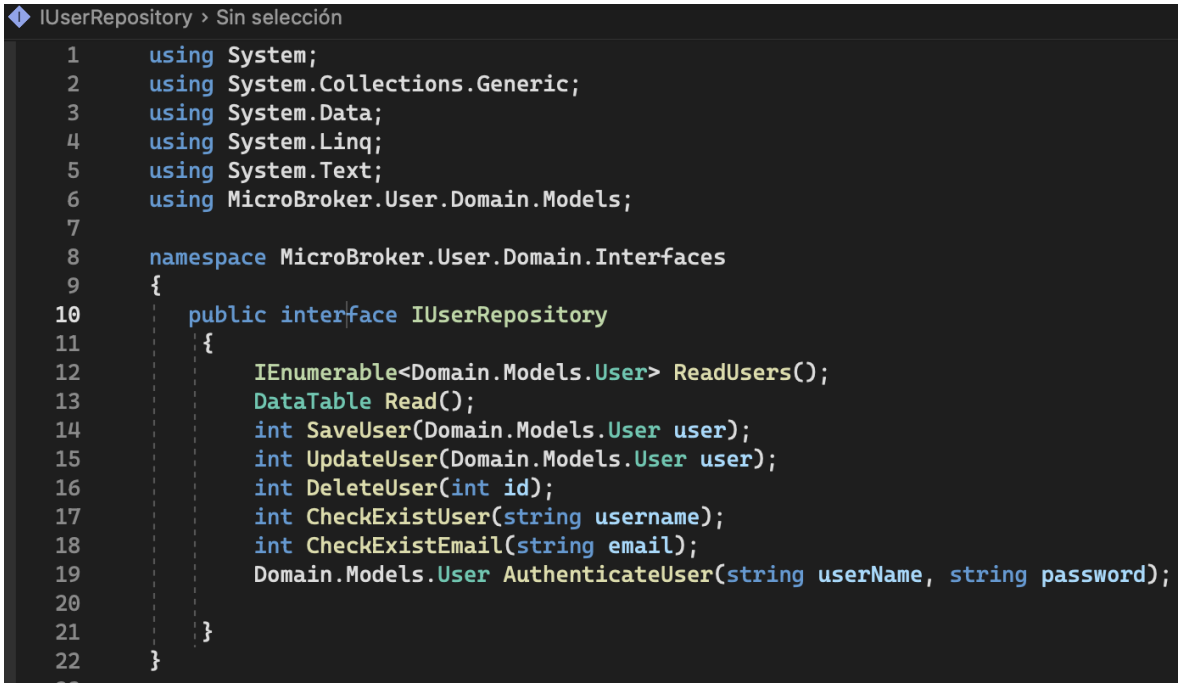


Figura 165 - "IUserRepository" script.

Para cumplir con la comunicación asincrónica entre el microservicio "User" y el microservicio "PlayList" y "Email", donde "User" hará el papel de "Publisher" o publicador dentro de la comunicación como se muestra en la figura 52, mientras que "Playlist" y "Email" harán el papel de subscriptores.

Primero para transmitir el requerimiento desde la Api hacia el AzureServiceBus, en el directorio "Commands" se creó una clase llamada "UserPlayListCommand" que efectuará el uso del objeto "Command" para la comunicación con la arquitectura del EventBus, además para realizar una acción con dicha clase se creó una nueva clase "CreateUserPlayListCommand" que hará uso de la clase abstracta antes desarrollada con el fin de instanciarla. Bajo la misma lógica se creó "EmailUserLoginCommand" y para efectuar la construcción del comando se generó "SendEmailUserLoginCommand".

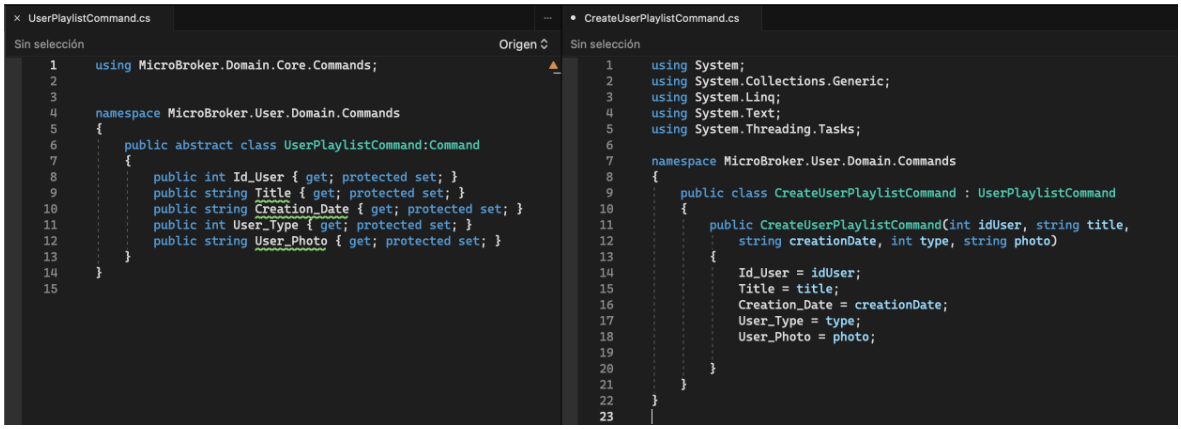


Figura 166 - “UserPlayListCommand” y “CreateUserPlayListCommand” scripts.

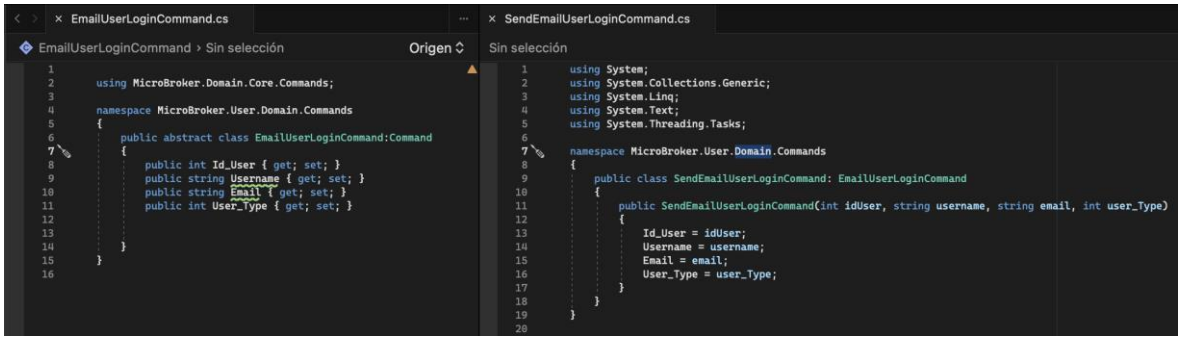


Figura 167 - “EmailUserLoginCommand” y “SendEmailUserLoginCommand” scripts.

Ahora para publicar el mensaje dentro del EventBus. En directorio “Events”, se necesitó la creación de las clases “UserPlaylistCreatedEvent” y “EmailUserLoginSendedEvent” que hacen uso del objeto “Event” y por lo tanto representará el evento a enviar.

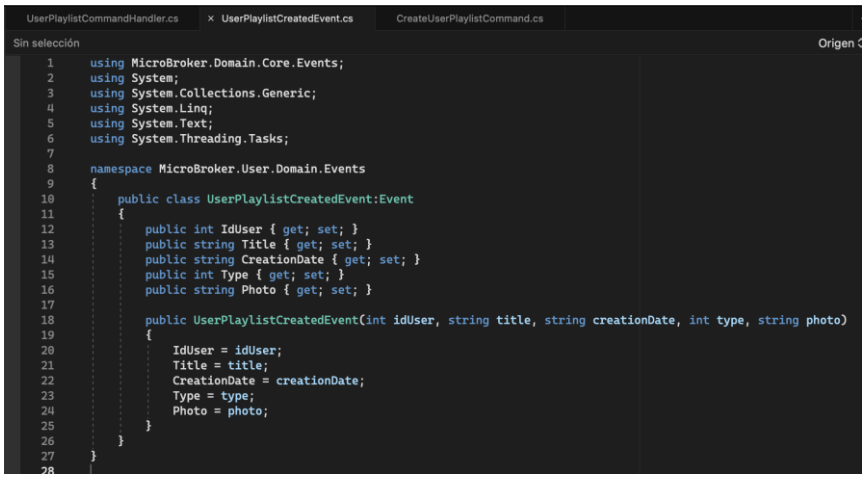


Figura 168 - “UserPlaylistCreatedEvent” script.

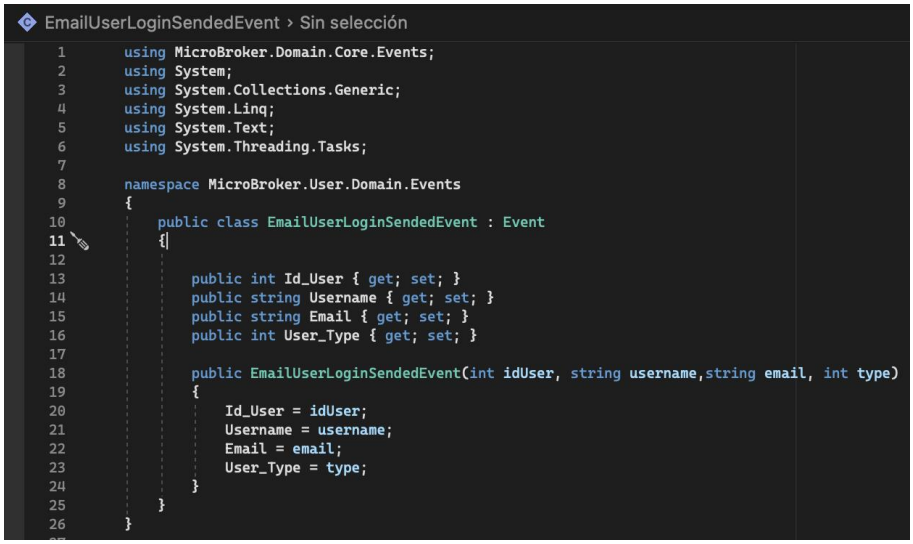


Figura 169 - "EmailUserLoginSendedEvent" script.

Ya al establecer la lógica para publicar el mensaje. En el directorio "CommandHandlers", se creó la clase "UserPlaylistCommandHandler" la cual hará uso del objeto "IRequestHandler" que tendrá como parámetro de entrada el comando "CreateUserPlayListCommand" que por último servirá para crear el evento a enviar de tipo "UserPlaylistCreatedEvent". Bajo la misma lógica se creó la clase "EmailUserLoginCommandHandler" que tendrá como parámetro el comando "SendEmailUserLoginCommand" que se usará para crear el evento a enviar "EmailUserLoginSendedEvent".

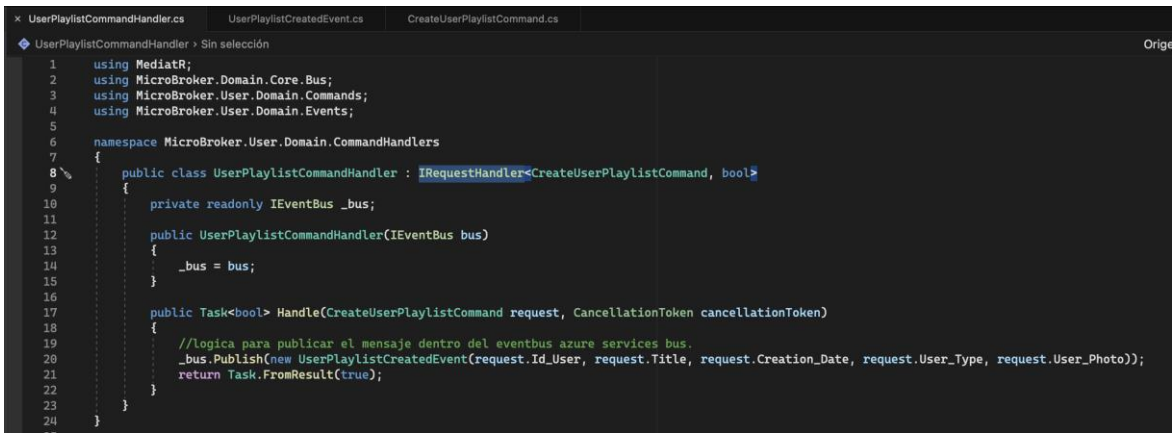


Figura 170 - "UserPlaylistCommandHandler" script.

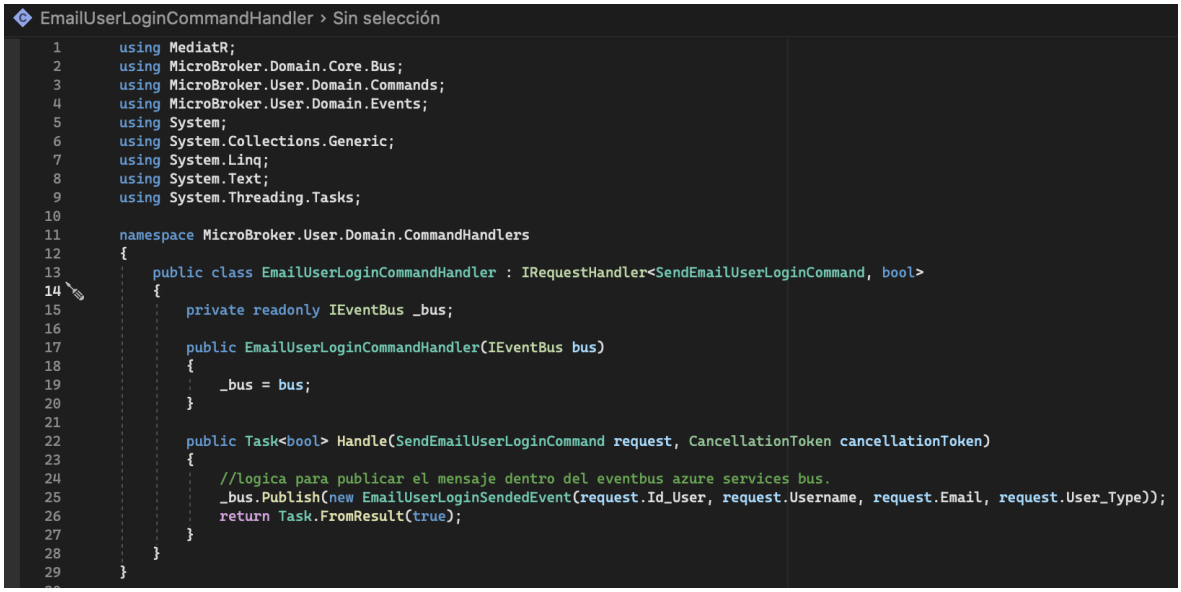


Figura 171 - "EmailUserLoginCommandHandler" script.

En el directorio **"Infrastructure"**, se creó un nuevo proyecto de tipo "Class Library" llamado **"MicroBroker.User.Infrastructure"**. El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos "Context", "Migrations" y "Repository.

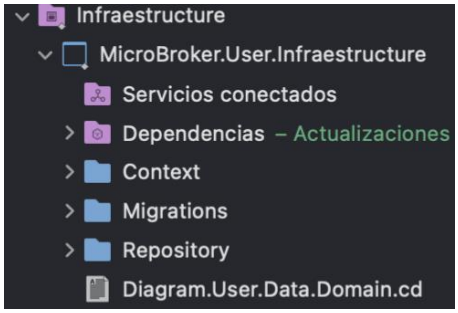


Figura 172 - Directorios principales del proyecto "MicroBroker.User.Infrastructure"

El directorio "Context" contendrá los scripts que instanciados representarán una sesión con la base de datos gracias al Entity Framework Core y su clase DbContext. Por ello se creó la clase "UserDbContext" que heredará las propiedades de DbContext y se la usará para consultar y guardar instancias de las entidades. Es decir, "UserDbContext" convertirá la clase de C# "User" a una entidad "Tbl_User", para la persistencia de registros en la base de datos "User".

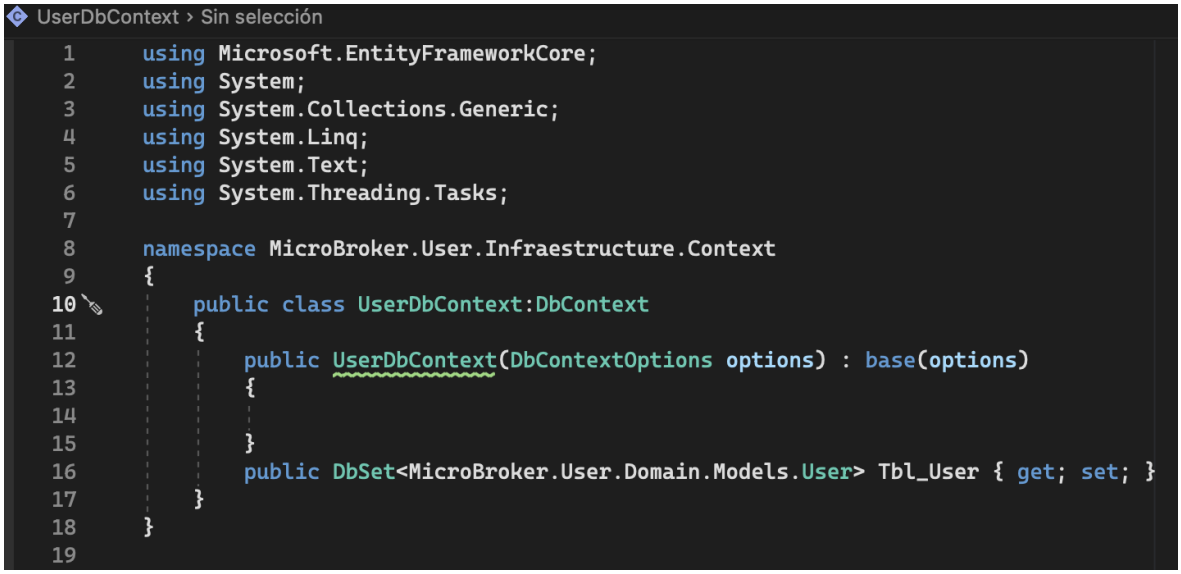


Figura 173 - “UserBbContext” script.

El directorio “Migrations” contendrá la definición en C# de las tablas que se van a crear dentro de la base de datos. La clase “UserDbContextModelSnapshot” que está dentro del directorio fue autogenerada gracias a Entity Framework Core.

Figura 174 - “UserDbContextModelSnapshot” script.

Es decir, para que “UserDbContextModelSnapshot” se creé, se hizo uso de la “Consola de Gestión de Paquetes” de Visual Studio, donde se ejecutó el comando:

```
add-migration UserDbContextModel
```

El directorio “Repository” contendrá los scripts de implementación a acceso a datos para cumplir los casos de uso. Por ello se creó la clase “UserRepository”, donde se instanciará la clase “UserDbContext” que permitirá consultar, crear, editar y eliminar registros “User” en la base de datos “User”.

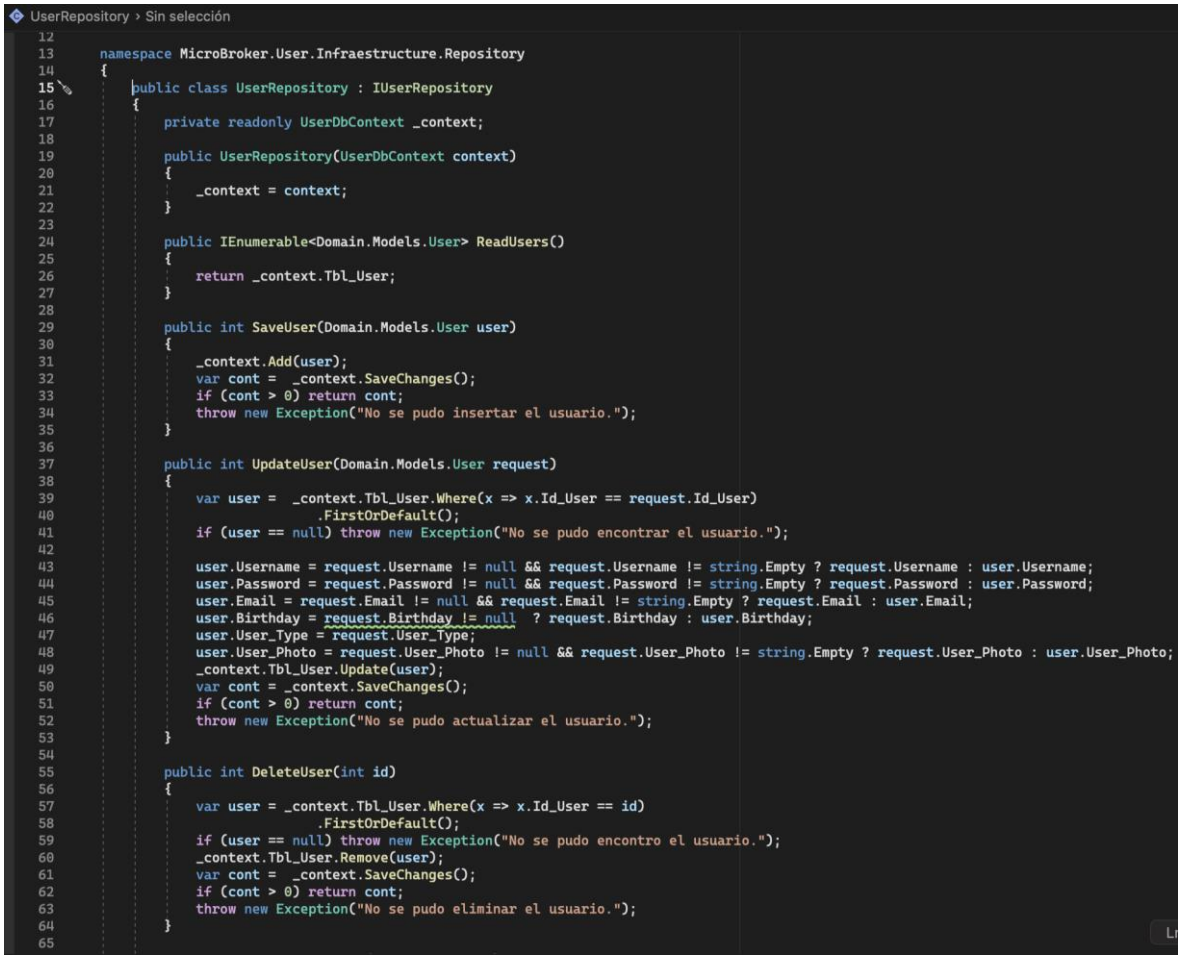


Figura 175 - "UserRepository" script parte 1.

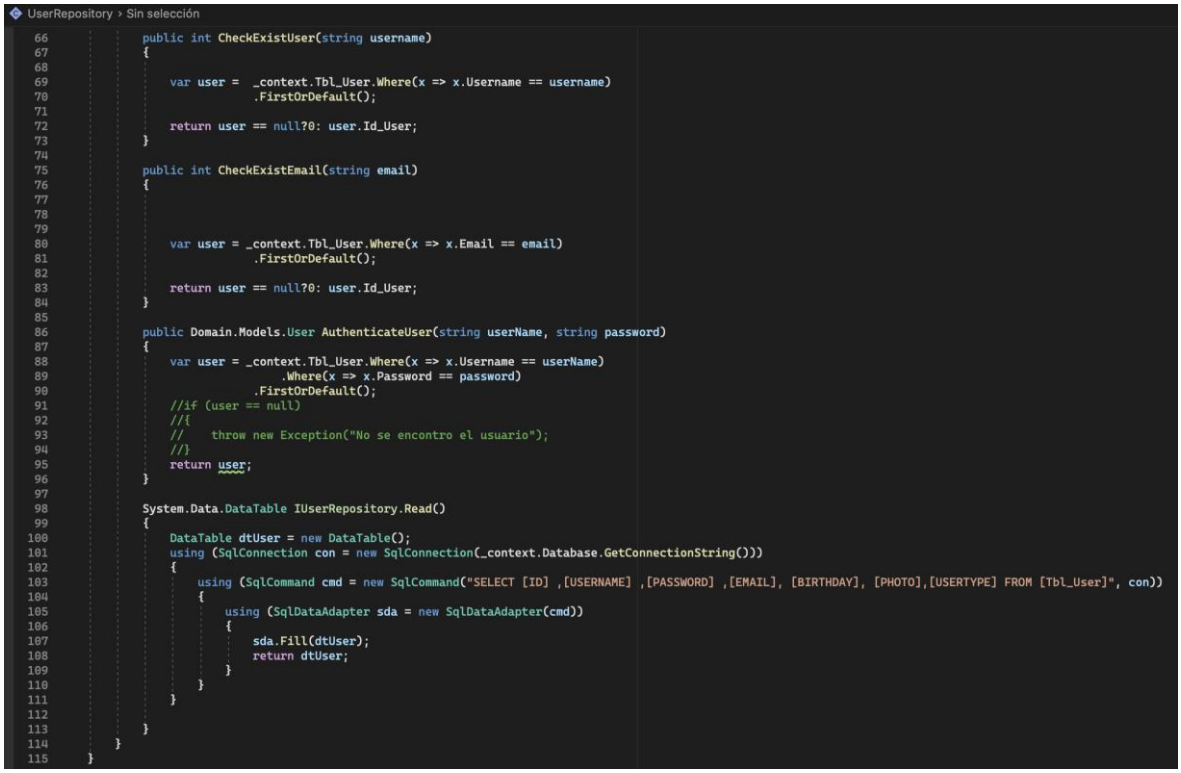


Figura 176 - “UserRepository” script parte 2.

En el directorio **“Application”**, se creó un nuevo proyecto de tipo “Class Library” llamado **“MicroBroker.User.Applicaction”.** El proyecto contendrá un conjunto de directorios que se crearon, entre ellos tenemos “Interfaces”, “Models” y “Services”.

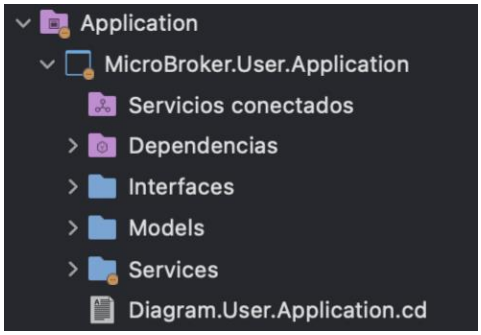


Figura 177 - Directorios principales del proyecto “MicroBroker.User.Application”

El directorio “Interfaces” contendrá los scripts que representen la abstracción de la implementación de los casos de uso del servicio. Por ello se creó la interfaz “IUserService”, donde se declaró los métodos abstractos del servicio que poseerán los mismos nombres usados en su implementación anterior. Además,

para completar la lógica de publicador en la comunicación asincrónica en la interfaz “IUserService” declaramos el método “CreatePlayList”; método que posteriormente es desarrollado en el directorio “Services” en la clase “UserService” con el propósito de enviar el comando para que se cree una nueva Playlist a través del EventBus y la comunicación entre componentes implementada previamente con el patrón Mediator.

```
1
2   using MicroBroker.User.Application.Models;
3   using System.Data;
4
5   namespace MicroBroker.User.Application.Interfaces
6   {
7       public interface IUserService
8       {
9           IEnumerable<Domain.Models.User> ReadUsers();
10          DataTable Read();
11
12
13          int SaveUser(Domain.Models.User user);
14          int UpdateUser(Domain.Models.User user);
15          int DeleteUser(int id);
16          int CheckExistUser(string username);
17          int CheckExistEmail(string email);
18
19          Domain.Models.User AuthenticateUser(string userName, string password);
20
21
22
23
24
25      }
26  }
```

Figura 178 - “IUserService” script.

El directorio “Models” contendrá los scripts que representan el modelo de “Playlist”, por lo que se desarrolló la clase “UserPlayList”. Modelo que va a ser usado en los servicios desarrollados posteriormente para la comunicación asincrónica.

UserPlaylist.cs

Sin selección

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace MicroBroker.User.Application.Models
8   {
9       public class UserPlaylist
10      {
11          public int IdUser { get; set; }
12          public string Title { get; set; }
13          public string CreationDate { get; set; }
14          public int Type { get; set; }
15          public string Photo { get; set; }
16      }
17  }
18
```

Figura 179 - “UserPlayList” script.

El directorio “Services” contendrá los scripts que representarán la implementación de los casos de uso del servicio. Por ello se creó la clase “UserService”, que consumirá la interfaz antes creada “IUserService” y además hará uso de la interfaz “IUserRepository” para la implementación de cada método del servicio. “UserService” tiene como objetivo funcional ser un objeto de transferencia de datos entre las capas de Dominio e Infraestructura del microservicio “User”. También se consumirá la interfaz “IEventBus” la cual incorporará la comunicación a través de Azure Event Bus, su uso de dio en el método “CreatePlaylist” y “AuthenticateUser” que tienen como propósito crear una playlist según un usuario y enviar un correo personalizado al usuario al ingreso al sistema, respectivamente.

Figura 180 - “UserService” script.

En el directorio “**Api**”, se creó un nuevo proyecto de tipo “Asp.Net Core Web Api” llamado “**MicroBroker.User.Api**”. El proyecto de tipo “Asp.Net Core Web Api” otorga un conjunto de directorios, cada uno con propósito definido.

Figura 181 - Directorios principales del proyecto "MicroBroker.User.Api"

El directorio "Controllers" contendrá los scripts que efectuarán el control de todas las APIs REST. Para ello se creó la clase "UserController", que llamará a los métodos generados en el servicio "UserService". En la clase se hizo uso de las peticiones dependiendo de la funcionalidad de los métodos del servicio; GET para solicitudes de lectura, POST para solicitudes de inserción, PUT para solicitudes de actualización, DELETE para solicitudes de borrado. En el API REST encargado de tomar la solicitud de un usuario y generar una playlist mediante comunicación asincrónica, esto se realizó en la clase "UserController" donde se creó un método HTTP tipo POST encargado de recibir la data de "PlayList" y llamar al "CreatePlaylist" de la interfaz de servicio "User" creado anteriormente.

Figura 182 - “UserController” script.

El directorio “Properties” tendrá el archivo autogenerado launchSettings.json. Este archivo tendrá la configuración que se usará cuando se ejecute la aplicación desde Visual Studio o mediante la CLI de .NET Core, es decir, se usará dentro de la máquina de desarrollo local. A pesar que este archivo no es necesaria para la publicación del aplicativo en el servidor de producción, si son oportunas ciertas configuraciones en producción entonces dichas configuraciones deben persistir en el archivo appsettings.json .

https://json.schemastore.org/launchsettings.json

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:9114",
      "sslPort": 0
    }
  },
  "profiles": {
    "MicroBroker.User.Api": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "http://localhost:5037",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Docker": {
      "commandName": "Docker",
      "launchBrowser": true,
      "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}/swagger",
      "environmentVariables": {}
    }
  }
}
```

Figura 183 - Archivo de configuración “launchSettings.json” del microservicio “User”.

- **Sprint 8 Review**

Se cumplió con éxito el objetivo del Sprint de crear el microservicio “User”, así como los criterios de aceptación por cada historia de usuario completada:

Código	Criterio de Aceptación	Cumplido
HU01-T01	Se generó las clases en representación de la entidad “User” y el modelo de negocio en su respectiva interfaz.	Sí
HU01-T02	Se creó las implementaciones de acceso a datos para la tabla de base de datos “TBL_USER”	Sí
HU01-T03	Se declaró los servicios e interfaces que funcionen como DTO’s.	Sí
HU01-T04	Se creó las APIs REST que ejecutarán los servicios del microservicio “User”	Sí
HU07-T42	Se implementó los comandos, eventos y controladores de los eventos, en la ejecución de la comunicación asincrónica para la publicación de eventos destinados a los microservicios “Playlist” y “Email”.	Sí
HU01-T05	Se creó en la capa de Aplicación la llamada al microservicio “User” mediante HTTP.	Sí
HU01-T06	Se realizó pruebas automatizadas gracias a la herramienta SWAGGER, para comprobar la funcionalidad del microservicio "User". La evidencia de las pruebas con el detalle se las puede observar en el **Anexo 3 subsección 3.1**.	Sí

Tabla 31 - Sprint 8 Review

- **Sprint 8 Retrospective**
    - **¿Qué salió bien?**

Se obtuvo las peticiones REST para las diferentes historias de usuario comprobando su funcionalidad a través de la herramienta SWAGGER.

- **¿Qué se puede mejorar?**

Después de la experticia conseguida del microservicio “User” en cuanto a comunicación asincrónica desde la perspectiva de publicador se prevé agilizar y mejorar los tiempos de desarrollo para futuros proyectos.

4.3.2 Fase de evaluación y pruebas

4.3.2.1 Pruebas de funcionalidad de la aplicación de arquitectura de microservicios.

En esta sección se detalla el proceso de pruebas de funcionalidad llevado a cabo en el marco de la implementación de una arquitectura de microservicios para migrar una aplicación monolítica de streaming musical. Las pruebas de funcionalidad son un componente crítico en el ciclo de vida de cualquier aplicación, y en particular en entornos de microservicios, ya que permiten validar el correcto funcionamiento de cada uno de los servicios y la interacción entre ellos[72]. Es así que el detalle de plan de pruebas con las capturas de pantalla que evidencien el resultado desde el Backend-end y Front-end, se las puede encontrar en el **Anexo 3** y **Anexo 4** respectivamente**.**

- Pruebas de funcionalidad en los microservicios

Microservicio	Número de pruebas	Código de prueba	Anexo
User	13	PF001 - PF013	3.1
Song	11	PF001 - PF011	3.2
Playlist	17	PF001 - PF017	3.3
Catalog	19	PF001 - PF019	3.4
Artist	11	PF001 - PF011	3.5
Album	11	PF001 - PF011	3.6

Email	1	PF001	3.7

Tabla 32 - Resumen prueba funcionales en los microservicios

La tabla resume las pruebas funcionales para diferentes microservicios en un sistema. Cada fila representa un microservicio específico y muestra el número total de pruebas funcionales asociadas con el microservicio, así como los códigos de prueba correspondientes. Además, se incluye una columna de anexo para hacer referencia a las pruebas en un documento separado. Esta tabla es útil para tener una visión general rápida del alcance de las pruebas funcionales en el sistema y para ayudar en la gestión y planificación de las pruebas.

- Pruebas de funcionalidad en el front-end

Fuente front-end	Número de pruebas	Código de prueba	Anexo
login.aspx	4	PF001 - PF004	4.1
RegisterUser.aspx	6	PF001 - PF006	4.2
UserManagement.aspx	5	PF001 - PF005	4.3
TracklistManagement.aspx	3	PF001 - PF003	4.4
SongManagement.aspx	7	PF001 - PF007	4.5
PlaylistSongManagement.aspx	8	PF001 - PF008	4.6
UserPlaylistSong.aspx	4	PF001 - PF004	4.7

UserPlaylistSong Management.aspx	6	PF001 - PF006	4.8
PlayerManagement.aspx	4	PF001 - PF004	4.9
CatalogManagement.aspx	4	PF001 - PF004	4.10
ArtistManagement.aspx	3	PF001 - PF003	4.11
AlbumManagement.aspx	3	PF001 - PF003	4.12
Email	1	PF001	4.13

Tabla 33 - Resumen pruebas funcionales en el Front-end

La tabla proporciona una lista detallada de los diferentes microservicios y fuentes Front-end, junto con el número de pruebas y el código de prueba correspondiente para cada uno de ellos. Además, se incluye una columna de Anexo que proporciona un número único para cada fuente. Esta información puede ser útil para cualquier equipo de desarrollo o de prueba que esté trabajando en estos sistemas, ya que les permitirá planificar y ejecutar pruebas de manera más eficiente y efectiva.

4.3.2.2 Pruebas no funcionales de la aplicación de microservicios.

En esta sección, se describe el proceso de pruebas de rendimiento llevado a cabo en el marco de la implementación de una arquitectura de microservicios para migrar una aplicación monolítica de streaming musical. El objetivo de estas pruebas fue evaluar la capacidad de la aplicación para manejar una carga de trabajo alta y constante, así como identificar posibles cuellos de botella y problemas de rendimiento en el entorno de microservicios [73].

Para llevar a cabo las pruebas de rendimiento y carga, se utiliza la herramienta Jmeter para la simulación de carga, monitoreo de rendimiento y el análisis de tráfico[73], [74]. Se llevaron a cabo pruebas bajo diferentes escenarios, incluyendo picos de tráfico, cargas constantes y períodos de carga variable, con el objetivo de evaluar el rendimiento de la aplicación en diferentes condiciones de uso. El detalle de plan de pruebas con las capturas de pantalla que evidencien el resultado se las puede encontrar en el **Anexo 5.**

Microservicio	Número de Pruebas	Código de Prueba	Anexo
User	7	PNF001-PNF007	5.1
Song	6	PNF001-PNF006	5.2
Playlist	11	PNF001-PNF011	5.3
Catalog	7	PNF001-PNF007	5.4
Artist	11	PNF001-PNF011	5.5
Album	11	PNF001-PNF011	5.6

Tabla 34 - Resumen de las pruebas no funcionales en los microservicios

La tabla proporciona un resumen detallado de las diferentes pruebas no funcionales realizadas en cada uno de los microservicios del sistema. Para cada microservicio se incluye el número de pruebas y el código correspondiente. La información es útil para cualquier equipo de desarrollo o de prueba que esté trabajando en estos sistemas, ya que les permitirá planificar y ejecutar pruebas de manera más eficiente y efectiva

4.3.3 Fase de despliegue

En esta sección se describirá el proceso de despliegue de la aplicación de microservicios desarrollada en este proyecto de titulación. Se explicará cómo se

utilizarán los servicios de Azure para alojar los microservicios, la base de datos y el Front-end de la aplicación.

- Despliegue de los microservicios

Para desplegar los microservicios de la aplicación en Azure, se utilizará un cluster de Kubernetes. Se creará un archivo de configuración que especifique los nodos y los servicios que conformarán el cluster. Se configurarán los microservicios para que se desplieguen en el cluster.

- Despliegue de la base de datos

Para alojar las bases de datos de la aplicación, se creará un servidor SQL. Aquí se implantará cada una de las bases de datos correspondiente y se configurarán las opciones de seguridad.

- Despliegue del Front-end

Para alojar el sitio web del Front-end de la aplicación, se utilizará el servicio de Azure App Service. Se creará un sitio web y se desplegará en Azure App Service.

- Configuración de seguridad

Es importante considerar la configuración de seguridad en cada uno de los servicios que se utilicen en Azure. Se utilizarán medidas de seguridad como el cifrado de datos, la autenticación de usuarios y el control de acceso por IPs.

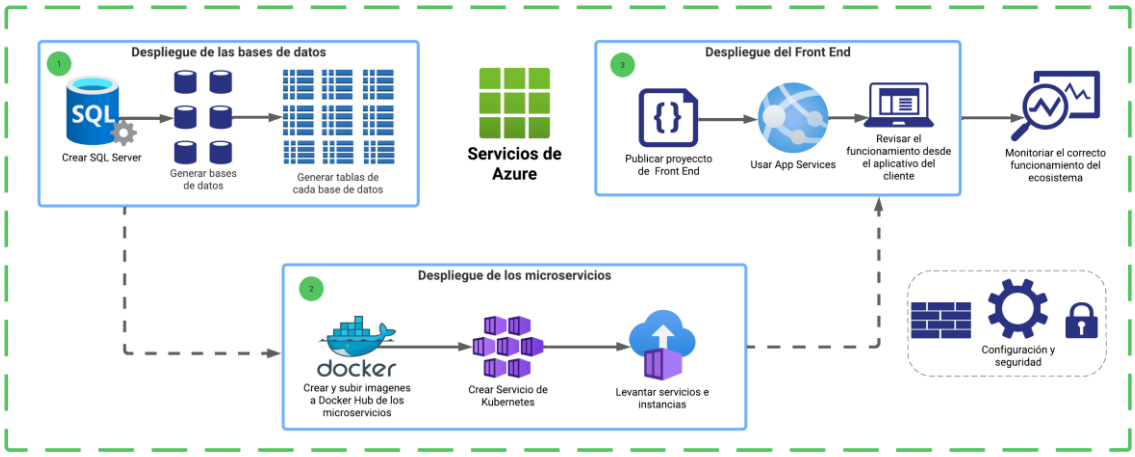


Figura 184 - Proceso de despliegue de la aplicación migrada, con los servicios de Azure.

El despliegue de una aplicación de microservicios en Azure requiere la configuración de varios servicios. Es importante considerar la seguridad y el monitoreo de la aplicación en cada uno de los servicios que se utilicen en Azure. El uso de un clúster de Kubernetes para alojar los microservicios es una solución escalable y flexible que permite la fácil adición o eliminación de microservicios según sea necesario. Para conocer a detalle el proceso de despliegue revisar el **Anexo 6**.

V DISCUSIÓN

La migración de una aplicación monolítica web de streaming musical hacia una arquitectura de microservicios es un tema de actualidad en el mundo del desarrollo de software. En este proyecto de titulación se ha abordado la problemática de cómo realizar esta migración de manera efectiva, asegurando la continuidad del servicio y mejorando su escalabilidad y mantenibilidad.

La arquitectura monolítica, aunque ha sido utilizada durante muchos años en el desarrollo de aplicaciones web, tiene limitaciones importantes en cuanto a escalabilidad y mantenibilidad. Una aplicación monolítica suele ser difícil de escalar, ya que todos los componentes están en una misma unidad, lo que implica que, si uno falla, toda la aplicación se ve afectada. Además, las actualizaciones y mejoras pueden ser muy costosas, ya que una pequeña modificación puede requerir un proceso de despliegue completo. La arquitectura de microservicios, por otro lado, propone una solución para estas limitaciones, ya que divide la aplicación en componentes más pequeños e independientes, lo que permite una mayor flexibilidad en cuanto a actualizaciones y mejoras, así como una mejor escalabilidad y resiliencia.

La migración es una alternativa para esta problemática, pero hay que considerar que puede ser un proceso complejo y lleno de incertidumbres. En este proyecto de titulación, se ha utilizado el marco de trabajo Scrum para abordar la incertidumbre en el proceso de diseño y desarrollo de la nueva arquitectura de software. Scrum al ser un marco de trabajo ágil que se utiliza en el desarrollo de software para gestionar proyectos complejos y cambiantes, es una gran opción. Por ello, Scrum ha sido utilizado para gestionar el proceso de diseño y migración de la aplicación de streaming musical hacia una arquitectura de microservicios. Scrum además se basa en la colaboración y la comunicación constante entre los miembros del equipo, lo que permite abordar la incertidumbre y los cambios en el proceso de diseño de la nueva arquitectura de software.

La incertidumbre en el proceso de diseño de la nueva arquitectura de software surgió debido a diferentes factores, como la falta de experiencia en la implementación de arquitecturas de microservicios y la complejidad de la aplicación

de streaming musical. Scrum ha permitido a los miembros del equipo gestionar la incertidumbre de manera efectiva mediante la creación de un backlog de trabajo y la iteración constante para la mejora del desarrollo de la nueva arquitectura.

El definir un proceso de migración puede resultar complejo sin previa experiencia, por esta razón fue imperativo el uso de un modelo que guie el flujo de trabajo para llevar a cabo con éxito la migración, además que brinda la posibilidad de verificar la factibilidad de la propuesta, como es MOMMIV (Modelo de Migración a Microservicios Versátil). MOMMIV es un modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el Principio de Ocultación de Información. El uso de este modelo trajo grandes ventajas, ya que permitió abstraer la complejidad de la migración en tres fases: Análisis, Diseño, y Desarrollo y pruebas.

En la Fase de Análisis, fue importante establecer revisiones e inspecciones al código fuente de la aplicación monolítica, referencias entre proyectos y su base de datos relacional, que permita entender la complejidad, diseño de la aplicación monolítica y lógica de negocio.

Ya en la Fase de Diseño, gracias al análisis previo se pudo descomponer la aplicación monolítica basado en la funcionalidad de negocio, obteniendo como resultado un listado de microservicios candidatos. Además, se definió un conjunto de patrones que sean compatibles con el principio de ocultación de información, es así que se utilizó los patrones de arquitectura EDA (Arquitectura basada en Eventos), DDD (Diseño basado en el Domino) y Arquitectura Limpia, pues ayudaron a asegurar que cada microservicio tenga una interfaz clara y definida, y mantenga los detalles internos ocultos de otros servicios.

Después en la Fase de Desarrollo y pruebas, se logró con éxito la codificación de cada microservicio y los extraordinarios funcionales solicitados, con la única dificultad a nuestra consideración era la falta de experticia con las herramientas y lenguajes utilizados. En la sub-fase de Pruebas se abordó y evaluó la aplicación migrada funcionalmente desde el Front-end cumpliendo satisfactoriamente en un 100% las pruebas realizadas, también en su evaluación no funcional el aplicativo

respondió óptimamente evidenciando las ventajas de una arquitectura de microservicios.

En este proyecto se ha demostrado que la migración hacia una arquitectura de microservicios es posible, aunque no es una tarea trivial. Se requiere una planificación cuidadosa y un conocimiento profundo de la arquitectura monolítica existente y de las necesidades de la aplicación. También es importante tener en cuenta que la migración puede ser costosa en términos de tiempo y recursos, y que es necesario asegurar la continuidad del servicio durante todo el proceso.

Sin embargo, los beneficios de la migración a una arquitectura de microservicios son significativos. La escalabilidad y resiliencia mejoradas permiten que la aplicación pueda crecer de manera sostenible y enfrentar posibles fallas de manera más eficiente. Además, la independencia de los componentes permite una mayor flexibilidad en cuanto a actualizaciones y mejoras, lo que se traduce en una mayor capacidad de adaptación a las necesidades del mercado.

La migración de una aplicación monolítica web de streaming musical hacia una arquitectura de microservicios es un desafío importante, pero necesario para asegurar la continuidad y el crecimiento de la aplicación. Aunque requiere una planificación cuidadosa y una inversión de tiempo y recursos significativa, los beneficios a largo plazo hacen que valga la pena realizar la migración.

VI CONCLUSIONES Y RECOMENDACIONES

6.1 Conclusiones

- En el proyecto integrador se cumplieron los objetivos propuestos pues logramos efectuar una migración de una aplicación monolítica web de streaming musical hacia una arquitectura de microservicios empleado efectivamente el modelo de migración MOMMIV y el marco de trabajo Scrum.
- La aplicación monolítica desarrollada con tecnologías antiguas como la versión de .NET 2002 y C# 1.2 presenta deficiencias significativas en la actualidad. La aplicación al componerse en un solo bloque de código, lo que significa que todas las funcionalidades se ejecutan en la misma máquina y comparten los mismos recursos, refleja las limitaciones en escalabilidad, flexibilidad y la capacidad de respuesta de la aplicación; limitando valga la redundancia su capacidad para competir en un mercado en constante evolución y mejorar su capacidad para adaptarse a las necesidades cambiantes de los usuarios y del negocio.
- El diseño de la arquitectura de microservicios, en especificó de los microservicios basado en el principio de ocultación de información nos otorgó varios beneficios; permitió al equipo de desarrollo trabajar de manera más independiente, ya que cada servicio pudo ser desarrollado y desplegado de manera independiente; permitió una mayor escalabilidad, ya que los servicios pueden ser escalados individualmente según sea necesario; además la ocultación de información mejora la seguridad, puesto que la información confidencial solo se comparte con los servicios que la necesitan.
- La comunicación entre los diferentes servicios de la aplicación fue un aspecto crítico durante la migración, donde se necesitó herramientas y estrategias de análisis específicos al definir las comunicaciones sincrónicas y asincrónicas para garantizar su correcto funcionamiento.
- La migración a una arquitectura de microservicios es una solución efectiva para mejorar la escalabilidad, flexibilidad y mantenibilidad de las aplicaciones de streaming musical. La implementación de esta arquitectura permitió mejorar la modularidad del sistema que a su vez se evidencia en una mayor

adaptabilidad a las necesidades cambiantes del negocio, una mayor eficiencia en el uso de recursos de hardware, una mayor facilidad en el mantenimiento, un mayor control en la detección y corrección de errores en definitiva la actualización de la aplicación.

- La arquitectura de microservicios permitió adaptarse mejor a las necesidades de los usuarios, ya que se pudo implementar una funcionalidad específica de forma independiente.
- La arquitectura de microservicios demostró ser una alternativa viable para mejorar la capacidad de respuesta de la aplicación en momentos de alta demanda, permitiendo una mayor disponibilidad del servicio y una mejor experiencia de usuario.
- La evaluación de la migración demostró que la arquitectura de microservicios es más compleja que la aplicación monolítica original, pero que sus beneficios en términos de escalabilidad, eficiencia y disponibilidad son significativos.
- La migración hacia una arquitectura de microservicios implicó ciertos desafíos y lecciones aprendidas, como la necesidad de gestionar adecuadamente la comunicación entre microservicios y la importancia de diseñar microservicios independientes y altamente cohesivos. Estas lecciones aprendidas pueden ser valiosas para futuros proyectos de migración hacia arquitecturas de microservicios.
- La migración de la aplicación monolítica web de streaming musical a una arquitectura de microservicios fue una oportunidad para aprender sobre nuevas tecnologías y arquitecturas de software, lo que resultó en una mejora en la capacidad técnica del equipo de desarrollo.

6.2 Recomendaciones

- El poseer un modelo guía para un proceso de migración como MOMMIV es primordial para asumir este tipo de retos. Sin embargo, el modelo no pone en retrospectiva el cómo tratar una migración en la vida real, por lo que se recomienda establecer y agregar a este modelo un plan estratégico de transición entre el aplicativo legado y el de microservicios, para completar un

escenario real donde es necesario cumplir con el 99% de disponibilidad del aplicativo a los usuarios.

- En el diseño de los microservicios y definición de la arquitectura, se recomienda analizar detalladamente las dependencias entre tablas de base de datos y la lógica de negocio, para no construir microservicios demasiado diminutos que en el establecimiento de comunicación entre ellos generen comunicaciones burdas y sobrecargadas.
- Para hacer un contraste de la factibilidad de comunicación sincrónica y asíncrona implementada en la arquitectura de microservicios, se propone el uso de una herramienta de comunicación binaria como Avro, Protocol Buffer con CPRM, y Thrift para volverlo a someter a pruebas y verificar la factibilidad de ello, ya que estas herramientas prevén una mejora en la eficiencia en la comunicación sincrónica.
- Es importante tener en cuenta que la migración a una arquitectura de microservicios requiere un mayor esfuerzo de diseño y planificación, y en un contexto real, la empresa debería evaluar la viabilidad de esta propuesta, ya que puede aumentar los costos del proyecto significativamente generando un impacto en la organización en general.
- En base a la evaluación de las pruebas funcionales de la migración se recomienda la actualización del aplicativo Front-end con el uso de nuevas tecnologías para mejorar la experiencia de usuario.
- En la evaluación de los resultados de las pruebas no funcionales de la migración se recomienda ser más rigurosos y basar las métricas de pruebas de manera más objetiva, es decir tomando en cuenta los recursos y características de hardware donde se va a asentar el aplicativo, para medir con precisión los beneficios obtenidos.

VII REFERENCIAS BIBLIOGRÁFICAS

[1] V. Velepucha, P. Flores, and J. Torres, “Migration of monolithic applications towards microservices under the vision of the information hiding principle: a systematic mapping study,” in *The International Conference on Advances in Emerging Trends and Technologies*, 2019, pp. 90–100.

[2] V. Velepucha, P. Flores, and J. Torres, “MOMMIV: Modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el Principio de Ocultación de Información,” *Revista Ibérica de Sistemas e Tecnologias de Informação*, no. E17, pp. 1000–1009, 2019.

[3] D. A. Ruelas Acero, “Modelo de composición de microservicios para la implementación de una aplicación web de comercio electrónico utilizando kubernetes,” 2017.

[4] R. S. Pressman and others, “A practitioner’s approach,” *Software Engineering*, vol. 2, pp. 41–42, 2010.

[5] M. R. V. Pardo, J. A. H. Tapia, A. S. G. Moreno, and L. F. V. Sánchez, “Comparación de tendencias tecnológicas en aplicaciones web,” *3c Tecnolog\'ia: glosas de innovación aplicadas a la pyme*, vol. 7, no. 3, pp. 28–49, 2018.

[6] Adobe, “Aspectos básicos de las aplicaciones web,” May 03, 2021. https://helpx.adobe.com/la/dreamweaver/using/web-applications.html#how_a_web_application_works (accessed Oct. 15, 2022).

[7] J. Z. Jiménez, *Aplicaciones web*. Macmillan Iberia, SA, 2013.

[8] S. Luján-Mora, *Programación de aplicaciones web: historia, principios básicos y clientes web*. Editorial Club Universitario, 2002.

[9] E. W. Dijkstra, “Go to statement considered harmful,” in *Pioneers and Their Contributions to Software Engineering*, Springer, 2001, pp. 297–300.

[10] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992.

[11] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[12] M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[13] C. B. Reynoso, “Introducción a la Arquitectura de Software,” *Universidad de Buenos Aires*, vol. 33, 2004.

[14] P. C. Clements, “A survey of architecture description languages,” in *Proceedings of the 8th international workshop on software specification and design*, 1996, pp. 16–25.

[15] J. F. Martínez, *Implantación de aplicaciones web (GRADO SUP.)*. Grupo Editorial RA-MA, 2014.

[16] A. Santos Carrazana, “Los sistemas distribuidos. Una aplicación en la enseñanza,” Universidad Central “Marta Abreu” de Las Villas, 2019.

[17] N. Ansari, S. Tiwari, and N. Agrawal, *Practical Handbook Of Thin-Client Implementation*. New Age International, 2005.

[18] F. Xhafa, *Aplicaciones Distribuidas con Java*. Delta Publicaciones, 2007.

[19] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O’Reilly Media, 2019.

[20] A. Guimarey, “Beneficios y riesgos de migrar una arquitectura monolítica a microservicios,” 2020.

[21] A. Dutta, M. S. Devi, and M. Arora, “Census web service architecture for e-governance applications,” in *Proceedings of the 10th International Conference on Theory and Practice of Electronic Governance*, 2017, pp. 1–4.

[22] A. A. R. Gomez and J. C. J. Fernández, “Revisión de la incorporación de la arquitectura orientada a servicios en las organizaciones,” *Revista Colombiana de Tecnolog\'\ias de Avanzada (RCTA)*, vol. 1, no. 31, pp. 77–88, 2018.

[23] M. N. Ibáñez Pozo, “Implementación de un framework para la programación de componentes auto-adaptables,” 2015.

[24] D. López, E. Maya, and others, “Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web,” 2017.

[25] S. Newman, *Building microservices*. “ O’Reilly Media, Inc.,” 2021.

[26] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[27] J. Lewis and M. Fowler, “Microservices: a definition of this new architectural term,” *MartinFowler. com*, vol. 25, no. 14–26, p. 12, 2014.

[28] B. Shafabakhsh, R. Lagerström, and S. Hacks, “Evaluating the Impact of Inter Process Communication in Microservice Architectures.,” in *QuASoQ@ APSEC*, 2020, pp. 55–63.

[29] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. Packt Publishing Ltd, 2018.

[30] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[31] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, and others, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[32] A. Shvets, “Dive Into Design Patterns,” *Refactoring. Guru*, 2019.

[33] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[34] B. Liskov, “Keynote address-data abstraction and hierarchy,” in *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, 1987, pp. 17–34.

[35] E. J. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[36] C. d Torre, B. Wagner, and M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*. 2022.

[37] A. L. Kooijmans *et al.*, *Transaction Processing: Past, Present, and Future*. IBM Redbooks, 2012.

[38] S. Zhelev and A. Rozeva, “Using microservices and event driven architecture for big data stream processing,” in *AIP Conference Proceedings*, 2019, p. 90010.

[39] C. Richardson, “Event-driven data management for microservices.” Nginx [online], 2015.

[40] M. Richards, *Software architecture patterns*, vol. 4. O’Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA~…, 2015.

[41] K. M. Chandy, “Event-driven applications: costs, benefits and design approaches, Gartner Application Integration and Web Services Summit 2006, 2006,” *Avilable from:< http://www. infospheres. caltech. edu/papers/Gartner_20060620. pdf*, 2012.

[42] K. Schwaber and J. Sutherland, “La Guía de Scrum,” *Scrum Alliance*, pp. 1–13, Nov. 2020.

[43] E. Woods, “Software Architecture in a Changing World,” *IEEE Softw*, vol. 33, no. 6, pp. 94–97, 2016, doi: 10.1109/MS.2016.149.

[44] C. B. Pareja Valerio and L. J. Burgos Robles, “La arquitectura de software basada en microservicios: Una revisión sistemática de la literatura,” 2019.

[45] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, “Smart objects as building blocks for the internet of things,” *IEEE Internet Comput*, vol. 14, no. 1, pp. 44–51, 2009.

[46] Sherwin A, “The History of Music Streaming, from Napster to Tidal,” *https://www.independent.co.uk/arts-entertainment/music/features/the-history-of-music-streaming-from-napster-to-tidal-10311069.html*, Jun. 03, 2015.

[47] Spanos B, “A Brief History of Music Streaming: From Napster to Tidal,” *https://www.rollingstone.com/music/music-news/a-brief-history-of-music-streaming-from-napster-to-tidal-102935/*, May 08, 2019.

[48] Opam K, “The Evolution of Music Streaming Services: From Napster to Spotify,” *https://www.theverge.com/2015/6/8/8748749/music-streaming-service-history-napster-spotify-apple*, Jun. 08, 2015.

[49] Dredge S, “The History of Music Streaming: How We Got Here,” *https://musically.com/2019/05/16/the-history-of-music-streaming-how-we-got-here/*, May 16, 2019.

[50] D. Roth, R. Anderson, and S. Luttin, “Información general de ASP.NET Core,” *https://learn.microsoft.com/es-es/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0*, Nov. 28, 2022.

[51] Microsoft, “Paseo por el lenguaje C#,” *https://learn.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/*, Sep. 22, 2022.

[52] Microsoft, “Historia de C#,” *https://learn.microsoft.com/es-es/dotnet/csharp/whats-new/csharp-version-history*, Dec. 09, 2022.

[53] Microsoft, “Control de versiones del lenguaje C#,” *https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/configure-language-version*, Nov. 19, 2022.

[54] TypeScript Lang Organization, “TypeScript for the New Programmer,” *https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html*, Dec. 12, 2022.

[55] Microsoft, “Visual Studio 2022 Preview,” *https://visualstudio.microsoft.com/es/vs/preview/*, 2022.

[56] Microsoft, “Visual Studio Code,” *https://code.visualstudio.com/docs*, 2022.

[57] Microsoft News, “¿Qué es y para qué sirve Visual Studio 2017?,” *https://www.msn.com/es-cl/noticias/microsoftstore/%C2%BFqu%C3%A9-es-y-para-qu%C3%A9-sirve-visual-studio-2017/ar-AAnLZL9?cvid=bd502ee791fe4a5b91e32b928e2d3d4c#image=1*, Dec. 14, 2018.

[58] S. Chacon and B. Straub, *Pro git*. Springer Nature, 2014.

[59] Microsoft, “Azure DevOps - Introducción,” *https://learn.microsoft.com/es-es/azure/devops/user-guide/what-is-azure-devops?view=azure-devops*, Nov. 18, 2022.

[60] Microsoft, “Azure Repos,” *https://learn.microsoft.com/es-es/azure/devops/repos/get-started/what-is-repos?view=azure-devops*, Nov. 16, 2022.

[61] Microsoft, “Azure Boards,” *https://learn.microsoft.com/es-es/azure/devops/boards/get-started/what-is-azure-boards?view=azure-devops*, Nov. 16, 2022.

[62] Microsoft, “Azure Pipelines,” *https://learn.microsoft.com/es-es/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops*, Dec. 08, 2022.

[63] Docker, “Docker overview,” *https://docs.docker.com/get-started/overview/*, 2022.

[64] Microsoft, “Azure Kubernetes Service,” *https://learn.microsoft.com/es-es/azure/aks/intro-kubernetes*, Dec. 21, 2022.

[65] Microsoft, “Documentación técnica de SQL Server,” *https://learn.microsoft.com/es-es/sql/sql-server/?view=sql-server-ver16*, Dec. 20, 2022.

[66] Microsoft, “SQL Server Management Studio,” *https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16*, Dec. 16, 2022.

[67] Microsoft, “Mensajería de Service Bus,” *https://learn.microsoft.com/es-es/azure/service-bus-messaging/service-bus-messaging-overview*, Nov. 03, 2022.

[68] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[69] M. Adams, “A brief overview of planning poker,” *https://www.atlassian.com/blog/platform/a-brief-overview-of-planning-poker*, Sep. 10, 2021.

[70] G. Califano, “Five Ways to Build Consensus,” *https://www.scrum.org/resources/blog/five-ways-build-consensus*, May 22, 2022.

[71] M. Cohn, “Four Quick Ways to Gain or Assess Team Consensus,” *https://www.mountaingoatsoftware.com/blog/four-quick-ways-to-gain-or-assess-team-consensus*, Apr. 24, 2018.

[72] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[73] A. G. Rodrigues, B. Demion, and P. Mouawad, *Master Apache JMeter-From Load Testing to DevOps: Master performance testing with JMeter*. Packt Publishing Ltd, 2019.

[74] Apache Software Foundation, "Apache JMeter™," *https://jmeter.apache.org/index.html*, 2022.

VIII ANEXOS

ANEXOS TRABAJO DE TITULACIÓN.docx