UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Diseño y desarrollo de un microservicio de migración de datos en un contexto de MDD

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Rabal García, Alejandro

Tutor/a: Letelier Torres, Patricio Orlando

CURSO ACADÉMICO: 2021/2022

Agradecimientos

Este trabajo ha requerido de un gran esfuerzo, el cual no podría haber realizado sin la ayuda y soporte de varias personas. Por ello, me gustaría comenzar este trabajo dando las gracias a todos los que me han apoyado durante este proceso y durante todos estos años.

En primer lugar, como no podría ser de otra forma, quisiera agradecer a mi tutor Patricio Letelier por su apoyo y consejos durante todo el proceso de elaboración de este trabajo, por su compromiso por la enseñanza y por mostrarse siempre tan cercano con sus alumnos.

También, me gustaría dedicar este trabajo principalmente a mi madre, a mi hermana y a mi abuela. Quiero daros las gracias por todo el amor y apoyo que me habéis dado y por lo feliz que he sido por vivir todos estos años a vuestro lado. Gracias por ayudarme a superar todas las dificultades y por ayudarme a ser quien soy.

A mis amigos que conocí al inicio de la carrera y con los que he compartido estos 4 años de mi vida. Gracias por todos los buenos ratos que hemos pasado y todas las experiencias vividas con vosotros, y que espero que sean muchas más.

Por último, quiero dar las gracias a mis compañeros de ADD Informática por los buenos momentos que hemos pasado juntos, por todo lo que me habéis enseñado y por vuestra amabilidad.

Resum

Aquest treball és realitzat dins del context d'unes pràctiques d'empresa, la qual ofereix com a producte i servei un sistema de planificació de recursos empresarials (ERP) per a la gestió i administració en l'àmbit sociosanitari. Aquest TFG forma part del treball realitzat en el departament d'R+D+I, en el marc del desenvolupament d'una nova versió del producte que oferirà noves funcionalitats i millors sobre l'actual.

La finalitat d'aquest treball és el desenvolupament d'un microservei de migració de dades integrat dins de l'estructura del nou producte programari compost per diferents microserveis. Aquest microservei s'encarregarà d'efectuar la migració de dades des del producte anterior al nou producte. Això permetrà que els clients puguen utilitzar el nou producte en el menor temps possible des de la seua instal·lació i sense necessitat de crear manualment les dades ja existents de l'anterior producte.

Aquesta migració de dades serà aprofitada no solament per a facilitar l'adaptació dels antics clients al nou producte, sinó que, entre altres coses, es farà una reestructuració de les dades, de manera que es permeten realitzar funcions que abans no es podien desenvolupar, com per exemple, l'anàlisi de dades per al seu ús en tasques d'aprenentatge automàtic o predicció.

El desenvolupament d'aquest projecte s'ha dut a terme en un context de Desenvolupament Dirigit per Models (MDD, per les seues sigles en anglés) la qual cosa ha facilitat el treball realitzat. Així mateix, s'ha aplicat una metodologia àgil i els *tests* s'han elaborat al mateix temps que s'escrivia el codi. També s'han dut a terme diverses proves amb dades reals per a comprovar el correcte funcionament d'aquest microservei.

La tecnologia usada per al desenvolupament del microservei ha sigut ASP.NET Core. Per a la creació dels models s'ha emprat una eina de DSL Tools i el llenguatge utilitzat ha sigut C#.

Paraules clau: Migració de dades, microserveis, Desenvolupament dirigit per models , DSL Tools, .NET, C#

Resumen

Este trabajo es realizado dentro del contexto de unas prácticas en empresa, la cual ofrece como producto y servicio software un sistema de planificación de recursos empresariales (ERP) para la gestión y administración en el ámbito sociosanitario. Este TFG forma parte del trabajo realizado en el departamento de I+D+i, en el marco del desarrollo de una nueva versión del producto que ofrecerá nuevas funcionalidades y mejoras sobre la actual.

La finalidad de este trabajo es el desarrollo de un microservicio de migración de datos integrado dentro de la estructura del nuevo producto software compuesto por diferentes microservicios. Este microservicio se encargará de efectuar la migración de datos desde el producto anterior al nuevo producto. Esto permitirá que los clientes puedan utilizar el nuevo producto en el menor tiempo posible desde su instalación y sin necesidad de crear manualmente los datos ya existentes del anterior producto.

Esta migración de datos será aprovechada no solamente para facilitar la adaptación de los antiguos clientes al nuevo producto, sino que, entre otras cosas, se hará una reestructuración de los datos, de forma que se permitan realizar funciones que antes no se podían desarrollar, como por ejemplo, el análisis de datos para su empleo en tareas de *machine learning* o predicción.

El desarrollo de este proyecto se ha llevado a cabo en un contexto de Desarrollo Dirigido por Modelos (MDD, por sus siglas en inglés) lo cual ha facilitado el trabajo realizado. Asimismo, se ha aplicado una metodología ágil y los *tests* se han elaborado al mismo tiempo que se escribía el código. También se han llevado a cabo diversas pruebas con datos reales para comprobar el correcto funcionamiento de este microservicio.

La tecnología usada para el desarrollo del microservicio ha sido ASP.NET Core. Para la creación de los modelos se ha empleado una herramienta de DSL Tools y el lenguaje utilizado ha sido C#.

Palabras clave: Migración de datos, Microservicios, Desarrollo dirigido por modelos, DSL Tools, .NET, C#

Abstract

This work is done in the context of a company internship, this company offers an enterprise resource planning (ERP) system as a software product for the management and administration in the social and health care field. This thesis is part of the work carried out in the R+D department, in the development framework of a new version of the product that will offer new features and improvements over the current one.

The objective of this work is the development of a data migration microservice integrated in the structure of the new product, consisting of different microservices. This microservice will be in charge of migrating data from the current product to the new one. This will allow the customers to use the new product in the shortest possible time after its installation, without the need to create manually the existing data of the previous product.

This data migration will be used not only to facilitate the adaptation of the current costumers to the new product, but also, among other things, a data restructuring will be carried out, so that new features will be able to be done, for example, the data analysis for its use in machine learning and prediction tasks.

The development of this project has been carried out in a Model Driven Development (MDD) context, which has facilitated the work carried out. Likewise, an agile methodology has been applied, and the tests have been developed at the same time that the code was written. Several tests have also been carried out with real data, in order to verify the correct functioning of this microservice.

The technology used for the development of the microservice is ASP.NET Core. For the creation of the models, some Domain Specific Language tools has been used, and the language used is C#.

Key words: Data migration, Microservices, Model driven development, DSL Tools, .NET, C#

Índice general

Índice general VII
Índice de figuras IX

1 Introducción 1
1.1 Motivación 1
1.2 Objetivos 2
1.3 Estructura de la memoria 3
2 Estado del arte 5
2.1 SQL *scripts* 5
2.2 Librerías 7
2.2.1 SqlClient 7
2.3 Herramientas existentes 9
2.3.1 Xplenty 9
2.3.2 AWS Database Migration Service 9
2.4 Comparativa 10
2.5 Conclusiones 11
3 Tecnologías utilizadas 13
3.1 Moq 13
3.2 DSL 18
3.3 Microservicios 19
3.3.1 Arquitectura monolítica 19
3.3.2 Arquitectura basada en microservicios 20
3.3.3 Tabla comparativa 21
4 Desarrollo de la solución 23
4.1 Planteamiento general 23
4.1.1 Estado del producto actual 23
4.1.2 Planteamiento de la migración 24
4.1.3 División de tareas 24
4.2 Especificación de requisitos 25
4.2.1 Casos de Uso 25
4.2.2 Requisitos no funcionales 29
4.3 Diseño y estructura 31
4.3.1 Estructura de la solución 31
4.3.2 Estructura de modelos DSL 33
4.4 Programación 39
4.4.1 Algoritmo principal 39
4.4.2 Patrones de migración 48
4.4.3 Tipos de entidades 51
4.5 Pruebas 53
4.5.1 Pruebas automatizadas 53
4.5.2 Pruebas sobre datos reales 56
4.5.3 Resultado de la aplicación de las pruebas 57

4.6 Metodología 57
4.6.1 Cronología del proyecto 59
5 Conclusiones y trabajo futuro 61
Referencias 63

Apéndice
A Objetivos de Desarrollo Sostenible 65

Índice de figuras

1.1 esquema de la estructura de la migración 3

2.1 datos utilizados como base para los *scripts* 6
2.2 *script* que combina las columnas Apellido1 y Apellido2 6
2.3 datos combinados tras la ejecución del primer *script* 6
2.4 datos utilizados como base para los *scripts* 7
2.5 datos utilizados como base para los *scripts* 7
2.6 creación de una tabla utilizando el paquete SqlClient 8
2.7 lectura de los datos de una tabla utilizando el paquete SqlClient 8
2.8 tabla comparativa de opciones y herramientas de migración de datos . . . 10

3.1 esquema de la estructura de la solución de ejemplo 14
3.2 constructor UserController 15
3.3 método de obtención de usuario 16
3.4 constructor de clase de pruebas 16
3.5 fragmento *Arrange* de método de prueba 17
3.6 fragmentos *Act* y *Assert* de método de prueba 18
3.7 ejecución de los métodos de prueba creados 18
3.8 tabla comparativa de arquitecturas presentadas 21

4.1 Casos de Uso del nuevo producto 25
4.2 Casos de Uso del proceso de migración 26
4.3 esquema de la arquitectura de las capas del microservicio 32
4.4 modelo de dominio 34
4.5 esquema de base de datos 36
4.6 acción ad hoc pública del modelo de aplicación 36
4.7 ejemplo de petición HTTP en Postman 37
4.8 modelos con las cinco acciones principales de la migración 37
4.9 ejemplo de modelo de métodos de migración 38
4.10 modelos creados para la migración de cada entidad 39
4.11 ejemplo de modelo de métodos de conversión 39
4.12 proceso de migración 41
4.13 DTO modelado que representa la tabla Enseres y sus columnas 42
4.14 esquema de la tabla Enseres 42
4.15 esquema de la estructura de la lógica para los tipos de migración 44
4.16 estructura de del código mediante clases abstractas y sus implementaciones 45
4.17 transformaciones entre estados de migración 46
4.18 entidad maestra con un solo nivel de datos 51
4.19 entidad maestra modificada con dos niveles de datos 51
4.20 modelo de *tests* de conversión de datos 54
4.21 modelo con *tests* de migración 56
4.22 línea de tiempo del proceso de programación 60

CAPÍTULO 1
Introducción

1.1 Motivación

Los datos son una parte casi imprescindible y a la vez única dentro de cada aplicación, pero al mismo tiempo deben ser estructurados, gestionados y mantenidos de una forma correcta, pues si no pueden generan muchos problemas. Estos problemas pueden ir desde comprometer la privacidad de los datos [1] de clientes o usuarios a carencias en cuanto a rendimiento debido a su gestión o estructura [2] en la base de datos o en el propio servidor de esta. Además, también existe la dificultad añadida que puede suponer la gestión de una gran cantidad de datos [3].

Aparte de la necesidad de gestionar y organizar los datos dentro de una misma aplicación, también es necesario tener en cuenta la transferencia de datos entre diferentes sistemas y productos. La transferencia de datos abarca desde el envío de pequeños grupos de datos o documentos, hasta la migración de bases de datos de forma completa. Este último proceso es el que va a desarrollarse en este trabajo, en el que se va a abordar como realizar la migración de los datos [4] existentes desde un producto software utilizado por cientos de clientes a uno nuevo que actualmente se encuentra en desarrollo.

Para concretar a qué se refiere el término migración de datos dentro de este proyecto, puesto que este puede adoptar un significado diferente, este se va a definir como: la transferencia de los datos ya existentes en el producto actual que ofrece la empresa, realizando durante el proceso, la conversión de la estructura de dichos datos para permitir su almacenamiento en el nuevo producto.

Cabe mencionar que el nuevo producto está desarrollándose con una arquitectura basada en microservicios [5], esto significa que el producto está siendo construido siguiendo una arquitectura compuesta de varios servicios independientes débilmente acoplados. Estos servicios funcionan de forma autónoma respecto al resto, teniendo una base de datos propia y realizando su despliegue de forma individual. Por otra parte, la experiencia del usuario es similar o idéntica a la que obtendría si el producto estuviese estructurado de forma monolítica.

Es importante mantener una estructura que se pueda adaptar al producto objetivo, por esta razón se ha decidido crear un microservicio que permita, no solo, migrar los datos existentes a una nueva base de datos, sino modificar y reestructurar los datos existentes durante el traslado de los mismos al nuevo producto. Realizar la transferencia y conversión de los datos permitirá una mejor gestión de los mismos y así poder implementar nuevas funcionalidades en el nuevo producto, como podría ser el análisis de los mismos para posibles predicciones, utilizando técnicas de *machine learning* [6].

El desarrollo de este módulo del producto como microservicio facilitará, entre otras cosas:

- El despliegue tanto del producto como del microservicio. Debido a que al tratarse de un producto compuesto por microservicios se puede decidir en qué máquinas desplegar este microservicio.
- Tolerancia a fallos. Un error durante la migración de los datos, que pueda ocasionar la parada del microservicio de migración, no provocará fallos en el resto de microservicios.
- Eliminar el microservicio una vez los datos hayan sido migrados. Al igual que con el despliegue, al tratarse de un fragmento independiente del producto, este se puede eliminar una vez la migración haya concluido.
- La integración dentro del propio producto. Debido a que el producto estará compuesto por microservicios resultará más sencillo seguir la estructura existente para su desarrollo.

Por contra parte, esta estructura suele generar una complejidad adicional al tener la necesidad de orquestar la comunicación entre los distintos microservicios, además también tiende a requerir una mayor cantidad de recursos al ejecutar cada uno de ellos por separado. Aun así, cuando se trata de un producto de un tamaño considerable, este tipo de estructuras en las que se dividen las diversas tareas en varias partes, provoca, a largo plazo, muy buenos resultados.

El desarrollo de este trabajo está en el contexto de unas prácticas de empresa por parte del autor. La empresa ofrece un software de administración y gestión para el ámbito sociosanitario, más concretamente un sistema de planificación de recursos empresariales (ERP). El autor ha tenido la oportunidad de formar parte del departamento de I+D+i, en el que junto a otros compañeros ha aprendido sobre las estructuras de microservicios y la gestión de datos, a la vez que realizaba diversas tareas siguiendo una metodología ágil y dirigida por modelos (MDD) [7].

Entre las diversas tareas desarrolladas por el autor dentro de la empresa se ha decidido escoger este tema debido a la magnitud y desafío del mismo, siendo, además, la tarea principal que está realizando dentro de ella. En este trabajo se abordará el proceso de diseño y creación del microservicio de migración de datos comentado, además de explicar como realizará las tareas de conversión y transferencia de datos.

1.2 Objetivos

El objetivo principal de este trabajo, es desarrollar una herramienta para la migración automática de datos dentro de del contexto de una aplicación compuesta por microservicios. A este objetivo principal se añaden varios requisitos u objetivos secundarios a cumplir para su completo y correcto funcionamiento:

- Creación y gestión de un histórico, donde poder revisar los datos migrados y las posibles incidencias que se hayan podido dar durante el proceso, para en un futuro permitir a los usuarios de la aplicación resolver dichas incidencias.
- Permitir que la migración de datos se pueda llevar a cabo de forma escalonada en el tiempo mediante la utilización del histórico anteriormente mencionado, pues dicho proceso podría requerir varios días para su finalización.

- Crear la migración de todos los datos necesarios de forma incremental, de manera que en cada incremento o sprint, se pueda probar que la migración funciona correctamente, utilizando para ello datos ofuscados proporcionados por la empresa.

Para entender mejor como se integrará el proceso de migración del objetivo principal, dentro de la estructura del producto se muestra el esquema de la figura 1.1, en el que se puede observar, a grandes rasgos, como el microservicio de migración se encargará de leer los datos desde el producto actual y enviarlos para su creación al microservicio pertinente.

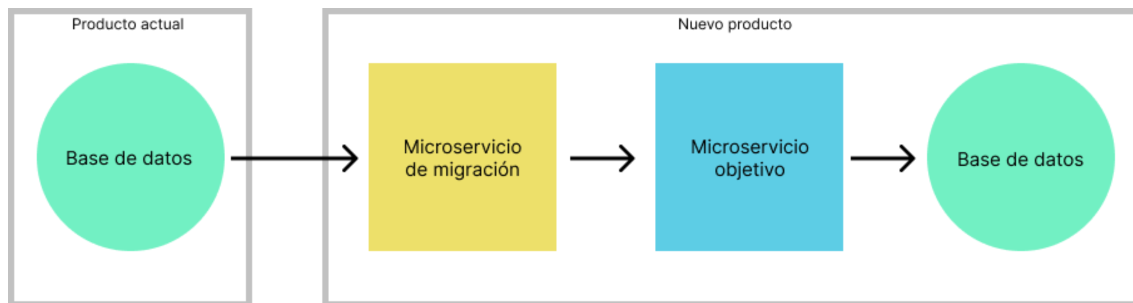

Figura 1.1: esquema de la estructura de la migración

1.3 Estructura de la memoria

Esta memoria se divide en los siguientes capítulos:

- En este primer se introduce el trabajo a realizar explicando la motivación y los objetivos a cumplir del mismo, junto con la estructura que seguirá el documento.
- El segundo capítulo presenta de forma breve el estado del arte, referente a las tecnologías que se utilizan actualmente para la gestión y manipulación de datos, también hablara de otras herramientas que podrían haberse utilizado para llevar a cabo un proyecto similar.
- El tercer capítulo se comenta las tecnologías que se han empleado para desarrollar este proyecto
- El cuarto capítulo explica el procedimiento para el desarrollo de la solución. Este capítulo empieza con un planteamiento generalista del proyecto a realizar, siguiendo por una especificación de requisitos, junto con el diseño y estructura del mismo, también contiene un apartado que habla de la programación, continúa explicando las pruebas realizadas y acaba explicando la metodología seguida durante el proceso.
- El quinto y último capítulo ofrece las conclusiones de este trabaja, así como el trabajo futuro que queda por realizar para la finalización del proyecto hasta su posible despliegue.

CAPÍTULO 2
Estado del arte

En la actualidad, existen diversas formas o herramientas que pueden ser utilizadas para realizar tareas respecto a la conversión, reestructuración o migración de datos. Por lo general, las herramientas existentes que ofrecen una interfaz suelen ser de pago, aunque también existen algunas gratuitas o de código abierto. Estas herramientas permiten realizar las acciones anteriormente nombradas de una forma relativamente sencilla y rápida, pero puede que no se adapten totalmente a las necesidades específicas de cada proyecto. Por otro lado, existen formas de gestionar los datos a través de *scripts* o librerías de código que permiten al desarrollador tener menos limitaciones sobre las operaciones que puede realizar sobre dichos datos. Por contra parte, esta última opción suele requerir de más tiempo y habilidades técnicas para llevar a cabo la creación de las funciones requeridas. Por tanto, cuando se platea realizar la migración o conversión de datos, se suele pensar en dos posibilidades, o bien tratar y gestionar los datos desde código o *scripts* creados por un desarrollador o utilizar herramientas previamente creadas para ello.

Antes de empezar a comentar algunas de las herramientas y librerías que pueden realizar migraciones de datos, es necesario mencionar que normalmente estas no están dedicadas exclusivamente a la ejecución de esta tarea, sino que por lo general poseen un amplio espectro de funcionalidades que pueden ejecutar. Así, para simplificar la relación con este trabajo se tratarán principalmente las funcionalidades que permitan la transferencia y la conversión de datos.

2.1 SQL *scripts*

Un *script* es un fichero que contienen una serie de comandos o código con el objetivo de que al ser ejecutado realice ciertas acciones específicas o lleve a cabo tareas concretas. La mayoría de gestores de base de datos suelen ofrecer la posibilidad de crear y ejecutar *scripts*, ya sea con la finalidad de obtener información a partir de consultas o realizar modificaciones sobre los datos o su estructura, como podría ser la creación, transformación o borrados de datos o tablas. En este apartado se van a presentar exclusivamente los *scripts* sobre bases de datos SQL, pero también existe la posibilidad de crear y ejecutar estos ficheros en base de datos no relacionales.

Para ver en mayor detalle como se relacionan estos tipos de documentos con la conversión o migración de datos, se van a mostrar varios ejemplos de estos ficheros creados y ejecutados utilizando el gestor de base de datos SQL Server Management Studio [1]. En

[1] Documentación oficial de SQL Server Management Studio: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16.

la figura 2.1 se muestran los datos que se utilizarán como base sobre la que aplicar las diferentes transformaciones a realizar.

	Nombre	Apellido1	Apellido2	FechaNacimiento	DNI
1	Ángel	Fernández	Ruiz	1987-07-27	12345678A
2	Beatriz	Jiménez	Serrano	1995-10-12	23456789B
3	Carmen	Martinez	Diaz	1985-02-09	12356789C

Figura 2.1: datos utilizados como base para los *scripts*

La primera modificación que se va a mostrar, es la creación de una nueva tabla en la misma base de datos, donde se va a aplicar una pequeña modificación sobre las columnas referentes a los apellidos de los usuarios. En la nueva tabla a crear se quiere tener una columna con el nombre de los usuarios y otra donde aparezcan los dos apellidos de los usuarios separados por un espacio. En la figura 2.2 se puede ver el *script* creado con sentencias SQL que llevará a cabo esta tarea. En la sentencia *select* se añaden las columnas que debe tener la nueva tabla y las transformaciones que se quieran aplicar sobre estas. La sentencia *into* muestra el nombre de la tabla a crear, o si esta existe, la tabla en la que se intentarán introducir los datos, es importante que el nombre de las columnas sea el esperado y que no se produzca ningún error de validación como que la longitud de uno de los campos sea mayor a la esperada, pues entonces la creación de los datos no podrá realizarse. Por último, la sentencia *from* indica la tabla de la que proceden los datos a copiar o transformar.

```sql
SELECT
    Nombre,
    Apellido1 + ' ' + Apellido2 as Apellidos
INTO NombreClientesCombinados
FROM Clientes
```

Figura 2.2: *script* que combina las columnas Apellido1 y Apellido2

Después de la ejecución de dicho fichero los resultados son los que se enseñan en la figura 2.3, una de las cosas a destacar es que para esta transformación no se quería contar con la columna DNI ni FechaNacimiento, por lo que al no incluirlas en el fichero de transformación no han sido añadidas a la nueva tabla.

	Nombre	Apellidos
1	Ángel	Fernández Ruiz
2	Beatriz	Jiménez Serrano
3	Carmen	Martinez Diaz

Figura 2.3: datos combinados tras la ejecución del primer *script*

Además de estas transformaciones básicas, también se pueden realizar de tal forma que se realicen entre tablas de diferentes bases de datos, o que creen las tablas directamente en la base de datos objetivo. En la figura 2.4 se muestra un ejemplo de esto último, donde partiendo de la base de datos llamada BaseDatosEjemplo y utilizando la tabla Clientes como origen de los datos, se busca crear o añadir los datos de origen a una tabla llamada ClientesNombreCompleto ubicada en la base de datos con nombre BaseDatosEjemplo. Durante este proceso también se va a realizar una conversión donde juntar tanto el nombre como los apellidos en una sola columna, teniendo en cuenta esta vez la columna DNI, pero dejando fuera la que se refiere a la fecha de nacimiento de los usuarios.

```
SELECT
    Nombre + ' ' + Apellido1 + ' ' + Apellido2 as Apellidos,
    DNI
INTO BaseDatosDestino..ClientesNombreCompleto
FROM BaseDatosEjemplo..Clientes
```

Figura 2.4: datos utilizados como base para los *scripts*

Al igual que en el caso anterior, en la figura 2.5 se puede ver el resultado final de la ejecución del fichero, y las transformaciones que este aplica, utilizando los datos de la primera base de datos.

	Apellidos	DNI
1	Ángel Fernández Ruiz	12345678A
2	Beatriz Jiménez Serrano	23456789B
3	Carmen Martinez Diaz	12356789C

Figura 2.5: datos utilizados como base para los *scripts*

2.2 Librerías

Para muchos lenguajes de programación se han creado diversas librerías que ayudan al desarrollador en el proceso de elaboración de proyectos software. Esto también se aplica a la gestión y manipulación de datos, ya sea de forma directa, realizando consultas a la base de datos o a través *frameworks* que simplifiquen este proceso y oculten parte de la lógica intermedia.

2.2.1. SqlClient

Un ejemplo de este tipo de herramienta es la librería o paquete SqlClient [8] del *framework* .NET, que permite interactuar con servidores de bases de datos SQL, de una forma bastante sencilla.

Este paquete permite realizar conexiones con bases de datos, a partir de la introducción de la ubicación de sus servidores y de las credenciales de acceso necesarias. Una vez hecha la conexión se pueden realizar varias tareas, desde la obtención del esquema de la base de datos, pudiendo obtener así todas las tablas que la componen junto con la información referente a cada una de sus tablas, hasta tareas más básicas como la ejecución de consultas para la introducción, modificación o lectura de datos.

El uso de este paquete es bastante sencillo, y permite empezar a trabajar con bases de datos de forma rápida, por contraparte, es necesario tener cuidado al crear las diferentes consultas dentro de la aplicación, pues si estas utilizan variables provenientes de la introducción de datos por parte de los usuarios, podría generarse un riesgo de inyección de SQL [9], esto podría dejar expuesta a la aplicación a posibles ataques o filtrado de datos personales de los usuarios. La parte buena es que este paquete cuenta con herramientas que permiten la construcción de las conexiones y de las consultas de forma segura cuando es necesario utilizar parámetros dentro de estas.

En las figuras 2.6 y 2.7 se muestra un ejemplo de creación y lectura de datos de la tabla similar a la vista en el apartado 2.1, respectivamente. Se puede observar que a partir del nombre de la base de datos y del servidor donde está ubicada se pueden realizar acciones y consultas de forma relativamente simple.

```
static void CreateTable(string databaseName, string databaseServerLocalitation)
{
    SqlConnectionStringBuilder connectionBuilder = new SqlConnectionStringBuilder()
    {
        InitialCatalog = databaseName,
        DataSource = databaseServerLocalitation,
    };

    using (SqlConnection connection = new SqlConnection(connectionBuilder.ConnectionString))
    {
        connection.Open();

        string commandString = "CREATE TABLE Clientes (" +
            "Nombre nvarchar(20), " +
            "Apellido1 nvarchar(20), " +
            "Apellido2 nvarchar(20), " +
            "FechaNacimiento date, " +
            "DNI nchar(9))";

        using (SqlCommand command = new SqlCommand(commandString, connection))
        {
            command.ExecuteNonQuery();
        }
    }
}
```

Figura 2.6: creación de una tabla utilizando el paquete SqlClient

```
static void ShowData(string databaseName, string databaseServerLocalitation)
{
    SqlConnectionStringBuilder connectionBuilder = new SqlConnectionStringBuilder()
    {
        InitialCatalog = databaseName,
        DataSource = databaseServerLocalitation,
    };

    using (SqlConnection connection = new SqlConnection(connectionBuilder.ConnectionString))
    {
        connection.Open();

        using (SqlCommand command = new SqlCommand("SELECT * FROM Clientes", connection))
        {
            SqlDataReader reader = command.ExecuteReader();

            while (reader.Read())
            {
                Console.WriteLine("Nombre: " + reader["Nombre"]);
                Console.WriteLine("Apellido1: " + reader["Apellido1"]);
                Console.WriteLine("Apellido2: " + reader["Apellido2"]);
                Console.WriteLine("FechaNacimiento: " + reader["FechaNacimiento"]);
                Console.WriteLine("DNI: " + reader["DNI"] + "\n");


            }
        }
    }
}
```

Figura 2.7: lectura de los datos de una tabla utilizando el paquete SqlClient

2.3 Herramientas existentes

2.3.1. Xplenty

Xplenty o Integrate.io [2] es una plataforma de integración de datos que permite preparar los datos contenidos en distintos servidores o aplicaciones y gestionarlos utilizando esta única herramienta sin necesidad de grandes conocimientos técnicos. Xplenty permite realizar transformaciones sobre datos con diferente origen para su integración en distintas áreas o su transferencia para ser analizados en herramientas de la nube.

De las características que ofrece este producto, la más destacable es la posibilidad de extraer datos desde diferentes fuentes de datos y su envío a varios destinos, pudiendo gestionar su transformación e integración a través de una interfaz donde se pueden arrastrar y soltar las fuentes de datos de origen y destino, así como las acciones sobre las consultas o transformaciones de datos que se quieran aplicar durante este proceso. Esto se realiza a través de flujos de datos automatizados, los cuales se pueden gestionar y revisar su estado actual.

Otra funcionalidad que Xplenty ofrece, además de gestionar los datos, es el análisis de estos, facilitando las tareas de recolección, limpieza y visualización sobre ellos. Asimismo, también permite obtener información sobre datos de clientes o estadísticas, entre otras cosas.

En cuanto al proceso de migración de datos, sería fácil utilizar esta herramienta para recolectar los datos de las diferentes bases de datos de origen y aplicar las transformaciones necesarias a través de la creación de acciones desde su interfaz gráfica.

El precio de este producto es calculado a partir de los conectores que se utilizan en lugar del uso de los datos. Por lo general, los usuarios pagan una tarifa mensual para poder utilizar el producto. En la página web de Xplenty no se muestran los precios para utilizarlo, sino que se realiza una tarifa dependiendo de las necesidades de cada cliente.

Como se ha comentado, Xplenty permite leer y procesar datos almacenados en la nube, algunas de las fuentes que integra son Amazon S3 [3] e IBM SoftLayer Object Storage [4] y conectar con servicios como Amazon Redshift [5] o Google BigQuery [6]. Por último, algunas de las empresas que utilizan esta herramienta son Feedvisor [7] y CloudFactory [8].

2.3.2. AWS Database Migration Service

Por otro lado, también existen productos cuyo objetivo es algo más específico, en cuanto al proceso de migración de datos. Este tipo de producto se enfocan en transferir los datos existentes en distintas bases de datos o servidores a un único producto o conjunto de ellos, para la utilización de ciertos productos, este es el caso de AWS Database Migration Service [9]. El objetivo principal de esta herramienta, es migrar los datos que tenga un cliente en sus bases de datos a Amazon Web Services (AWS) para permitir que este pueda utilizar sus servicios con los datos que ya tiene.

[2]Web oficial de Integrate.io: https://www.integrate.io/.
[3]Web oficial de Amazon S3: https://aws.amazon.com/es/s3/.
[4]Web oficial de IBM SoftLayer Object Storage: https://www.ibm.com/es-es/cloud/object-storage.
[5]Web oficial de Amazon Redshift: https://aws.amazon.com/es/redshift/.
[6]Web oficial de Google BigQuery: https://cloud.google.com/bigquery?hl=es.
[7]Web oficial de Feedvisor: https://feedvisor.com/.
[8]Web oficial de CloudFactory: https://www.cloudfactory.com/.
[9]Web oficial de AWS Database Migration Service: https://aws.amazon.com/dms/?nc1=h_ls.

AWS Database Migration Service ofrece la posibilidad de realizar tanto migraciones homogéneas como heterogéneas, es decir, transferir los datos desde una única plataforma o desde varias de ellas a la vez, respectivamente. Con esta herramienta también se pueden duplicar los datos ya migrados para su uso en otras herramientas de AWS, permitiendo utilizar las distintas características que cada una de ellas ofrece.

Otra funcionalidad que ofrece este producto que permite migrar los datos a la solución de almacenamiento de datos de AWS que mejor se adapte a su estructura actual. La parte negativa de esta herramienta es que únicamente permite la migración de datos de bases de datos o servidores externos a las herramientas de almacenamientos de datos de AWS y no al revés, por lo que no es una herramienta útil en casos de que se quieran migrar datos entre servidores propios, aunque puede resultar una buena opción para no utilizar hardware propio y en muchos casos abaratar los costes que su renovación y mantenimiento podría ocasionar.

En la página oficial de AWS Database Migration Service se muestran algunos casos de uso de esta herramienta, mostrando además los ejemplos de algunos problemas que tenían varías empresas y como los han podido solucionar a través de la utilización de esta herramienta y de las otras soluciones que ofrece AWS. Las empresas que más suelen utilizar los servicios que ofrece AWS son las de tamaño pequeño y mediano, al no necesitar enfocar sus esfuerzos en gestionar ni la infraestructura ni el hardware de los centros de datos y poder enfocarse en el producto o servicio que ofrecen.

2.4 Comparativa

A modo de resumen, tras haber comentado algunas de las distintas opciones y herramientas que existen para la migración de datos, se muestra en la figura 2.8, un resumen de sus características a grandes rasgos para remarcar sus diferencias. Para cada opción se muestra el tipo al que pertenece, el coste de su uso, los posibles destinos a los que la herramienta puede migrar los datos, la capacidad de personalización sobre la estructura de los datos durante su migración y el tiempo que llevaría realizar la creación del proceso de migración.

	Forma de uso	Coste	Destino	Personalización	Tiempo de desarrollo
Scripts SQL	Consultas SQL	Ninguno	Bases de datos SQL	Media / Alta	Alto
SQL Client	Implementar código	Ninguno	Cualquiera	Alta	Alto
Xplenty	Uso del producto	De pago	Varios disponibles	Media	Medio
AWS Database Migration Service	Uso del producto	De pago	Servicios de almacenamiento de AWS	Baja	Bajo

Figura 2.8: tabla comparativa de opciones y herramientas de migración de datos

2.5 Conclusiones

Antes de finalizar este capítulo se explicará, cuál ha sido la opción elegida para desarrollar el microservicio de migración una vez examinadas varias de las opciones existentes, y el por qué se ha decidido utilizar una de ellas sobre el resto.

Por una parte, se requiere que este proceso se pueda adaptar a las características ya existentes del producto en desarrollo, por lo que es necesario que sea lo más flexible posible y se adapte a las tecnologías que utiliza la propia empresa, por ello, los productos externos como Xplenty y AWS Database Migration Service quedarían descartados desde un principio. Además, otro de los motivos para evitar depender de productos ajenos al control de la empresa, es el carácter sociosanitario y administrativo de los datos, pues estos se tratan de datos personales de los clientes de las empresas que utilizan el producto.

Por otra parte, debido a nuevas validaciones sobre los datos que tendrá el nuevo producto, es necesario que la creación de los datos en la nueva base de datos se haga siguiendo un procedimiento que permita la solución de posibles casos de error durante el proceso de migración. Es por esta razón que es más conveniente desarrollar el procedimiento de migración de datos a través de código utilizando librarías como SQL Client, en lugar de utilizar ficheros con sentencias SQL, pues al migrar los datos de esta forma, no se ejecutarían las validaciones del nuevo producto hasta que los datos estuviesen en la base de datos de destino y en caso de error sería más difícil dar una solución.

Otra de las ventajas de crear este microservicio y su proceso de migración utilizando código propio es que permite integrarlo más fácilmente en el producto junto al resto de microservicios que lo componen, pues existe ya una estructura que facilita su creación y despliegue futuro, pudiendo además realizar la conversión de los datos sin restricciones de ningún tipo facilitando de esta forma la resolución de errores que puedan surgir.

CAPÍTULO 3
Tecnologías utilizadas

La tecnología utilizada para la creación del microservicio es ASP.NET Core [10], empleando C# [11] como lenguaje de programación y Visual Studio como entorno de desarrollo, en la versión 2019, este entorno es comúnmente usado junto con esta tecnología por las facilidades que ofrece al usarse conjuntamente.

Además de las tecnologías ya nombradas, se han utilizado otras herramientas como SQL Server Management Studio [12] un gestor de base de datos, que permite trabajar con infraestructuras SQL y bases de datos relacionales. Este gestor ha sido utilizado principalmente para estudiar los datos ya existentes en el producto actual y el diseño de la estructura de estos. También ha servido para comprobar el correcto funcionamiento del microservicio al permitir revisar la correcta migración de los datos durante las distintas pruebas.

Con el objetivo de iniciar el proceso de la migración de datos se ha utilizado Postman[1] para realizar la petición de ejecución del proceso, y enviar la información que indica donde se encuentra la base de datos sobre la que se quiere realizar el proceso.

Por último, caben destacar las herramientas de Lenguajes de dominio específico (o DSL, por sus siglas en inglés) de Visual Studio ampliadas por la empresa, que permiten la creación de modelos y la generación automáticas de código a través de plantillas y la librería Moq utilizada para la creación de métodos de prueba, estas se explicarán a continuación. A modo de introducción, la librería o framework Moq[2] de .NET facilita la creación de pruebas unitarias de clases aislándolas de sus dependencias, permitiendo asegurar la llamada a los métodos de dichas dependencias. Por otro lado, las herramientas de DSL de Visual Studio que han sido creadas usando el SDK, han posibilitado la creación de modelos que permiten generar de forma parcial el código del microservicio.

3.1 Moq

Esta librería es utilizada en la parte que comprueba el correcto funcionamiento del proceso de migración y que los datos que se enviarán a los otros microservicios para su creación son los esperados y cuya estructura se ha modificado de la forma deseada.

Moq permite aislar las clases, sobre las que se quieren realizar las pruebas, de sus dependencias. Utilizando Moq se puede crear objetos que simulan el comportamiento que implementarían los objetos reales, y utilizarlos en la prueba de la clase a verificar, haciendo que las pruebas creadas comprueben solo la clase en cuestión y no sus dependencias.

[1] Web oficial de Postman: https://www.postman.com.

[2] Repositorio en GitHub de la librería Moq: https://github.com/Moq/.

A continuación, se mostrará un ejemplo en el cual se va a crear una API conectada a una base de datos SQL. La funcionalidad que se desarrolla es una gestión básica de usuarios. Para simplificar esta presentación, la API contendrá únicamente dos métodos, uno de ellos permitirá crear usuarios en la base de datos, mientras que el otro permitirá obtener uno de estos usuarios mediante su identificador asignado. El objetivo de esta aplicación de ejemplo no es tener una utilidad real, sino servir de base para la creación de métodos de prueba y mostrar el funcionamiento de Moq.

El primer paso en la creación de la API, ha sido crear un proyecto web de ASP.NET Core, utilizando para ello el IDE Visual Studio 2019. Este proyecto, al que se ha llamado MoqExample es el que contiene la lógica de la API, es decir, los métodos que se expondrán y que son llamados para la ejecución de las tareas anteriormente descritas. Por otra parte, se ha creado también otro proyecto nombrado MoqExampleDatabase que contendrá la lógica encargada de conectar y enviar los datos recibidos a una base de datos SQL.

La estructura de los proyectos descritos y sus clases es la que se muestra en la siguiente figura 3.1, además también aparece el proyecto de pruebas, nombrado MoqExampleTests, en el cual se utiliza la librería Moq, que se explicará más adelante.

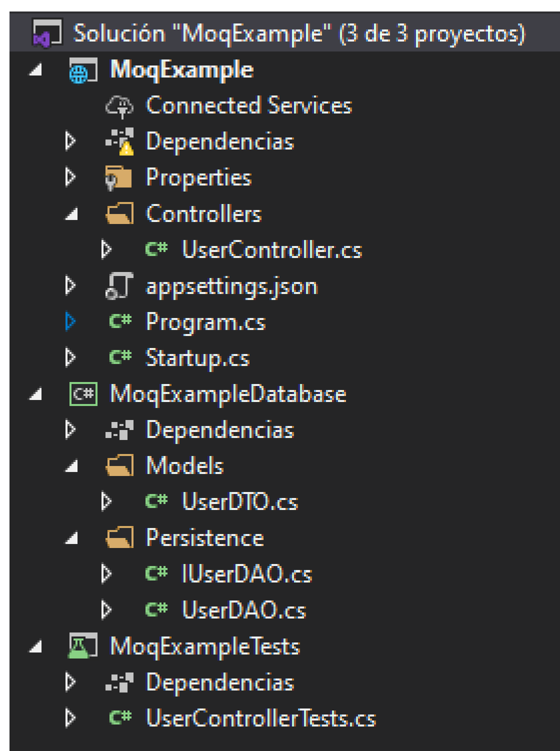

Figura 3.1: esquema de la estructura de la solución de ejemplo

A continuación, se ha creado la clase que contiene los métodos de la API sobre los que se van a realizar pruebas, cómo se puede ver en la figura 3.2, el constructor de dicho método necesita de una clase u objeto que implemente la interfaz *IUserDAO*. Para evitar una dependencia rígida entre una clase que implemente dicha interfaz y la clase UserController, se ha utilizado un patrón de diseño llamado inyección de dependencias [13].

El propósito del patrón de diseño software mencionado posibilita invertir el control para la resolución de dependencias, esto quiere decir, el patrón de inyección de dependencias permite evitar que una clase dependa directamente de otra diferente, haciendo que se utilice una abstracción en vez de dicha clase. En nuestro ejemplo se está utilizando la inyección de dependencias con la finalidad de evitar que la clase UserController

dependa directamente de la clase UserDAO, y en su lugar, lo haga de una abstracción llamada *IUserDAO*, la cual se trata de una interfaz que indica los métodos a definir por parte de la clase que la implemente.

```
[ApiController]
[Route("")]
3 referencias
public class UserController : ControllerBase
{
    private readonly IUserDAO _userDAO;

    2 referencias | 1/1 pasando
    public UserController(IUserDAO userDAO)
    {
        _userDAO = userDAO;
    }
```

Figura 3.2: constructor UserController

En la siguiente figura 3.3 se muestra el método encargado de devolver el usuario cuyo identificador sea el mismo que el que se haya recibido como parámetro en la llamada a la API, y de devolver un mensaje de error en caso de que suceda algún error o no se haya podido encontrar a un usuario con dicho identificador. Para el mensaje de respuesta de la API, se ha decidido seguir la especificación o convenio JSend [14] debido a su simplicidad

Respecto a este método de obtención de usuarios a partir de su identificador, se han implementado tres posibles casos en los que cada uno de ellos devuelve su respectiva respuesta.

- En el primer caso, se devolverá un mensaje de error cuando ocurra un error con el objeto que implementa la interfaz *IUserDAO* o con la ejecución de su método GetUser.
- En el segundo, se retorna un mensaje que indica un error con respecto al identificador del usuario cuando no se ha podido encontrar en la base de datos un registro de usuario que contenga dicho identificador.
- En el tercer y último caso, se devuelve un mensaje de éxito junto con el objeto usuario obtenido de la base de datos.

Una vez presentado el proyecto de ejemplo y el método sobre el que se van a realizar las pruebas, pasamos a comentar la clase de pruebas, creada para comprobar el correcto funcionamiento del método de obtención de usuarios.

Como se puede ver en la figura 3.4, el primer objeto que se define en esta clase de pruebas se trata de un objeto *Mock*, es decir, un objeto simulado que implementa la interfaz *IUserDAO*. Para este objeto se puede definir el comportamiento de sus métodos como se verá a continuación y también se puede obtener una instancia del objeto simulado utilizando su propiedad *Object*. Es esta instancia la que permite crear un nuevo objeto de la clase *UserController* el cual es asignado a la variable homónima de la clase.

```
[HttpGet]
3 referencias | 3/3 pasando
public Dictionary<string, object> GetUser(Guid userId)
{
    UserDTO userDTO;

    try
    {
        userDTO = _userDAO.GetUser(userId);
    }
    catch
    {
        return new Dictionary<string, object>()
        {
            { "status", "error" },
            { "message", "A database error has occurred" },
        };
    }

    if (userDTO == null)
    {
        return new Dictionary<string, object>()
        {

            { "status", "error" },
            { "data", new Dictionary<string, string>()
                {
                    { "userId", "User not found for the provided identifier" },
                }
            },
        };
    }

    return new Dictionary<string, object>()
    {
        { "status", "success" },
        { "data", new Dictionary<string, UserDTO>()
            {
                { "user", userDTO },
            }
        },
    };
}
```

Figura 3.3: método de obtención de usuario

```
[TestClass]
1 referencia
public class UserControllerTests
{
    private readonly UserController userController;
    private readonly Mock<IUserDAO> userDaoMock = new Mock<IUserDAO>();

    0 referencias
    public UserControllerTests()
    {
        userController = new UserController(userDaoMock.Object);
    }
```

Figura 3.4: constructor de clase de pruebas

De los casos anteriormente mencionados sobre la obtención de usuarios, se va a mostrar únicamente los métodos de prueba, referentes al tercero de ellos, debido a su similitud con el resto de métodos de prueba y que este es el más completo de los tres en cuanto a comprobaciones.

Para la creación de los métodos de prueba se ha seguido la estructura *Arrange*, *Act* y *Assert* [15]. Esta estructura permite al programador crear pruebas que sigan un convenio para una mejor organización interna en estas. A modo de resumen, en la primera parte (*Arrange*), se prepara los objetos, dependencias o valores que el método de prueba vaya a necesitar para su ejecución, en la segunda parte (*Act*), se realizan las acciones cuyo resultado o ejecución quieren ser comprobadas, por último, en la tercera parte (*Assert*), se valida el correcto funcionamiento del método probado, por ejemplo, confirmando que el valor devuelto en el método ejecutado en el esperado.

Para la validación del caso se ha seguido la misma estructura descrita, para comprobar que la llamada al método devuelve un mensaje de éxito en la ejecución de la tarea, al obtener correctamente un usuario desde la base de datos. En la parte *Arrange* de la prueba mostrada en la figura 3.5 se crean y se asignan distintos valores a las variables que componen en este caso un objeto usuario, y a continuación se utilizan para crear un objeto de este mismo tipo. Además, utilizando el método *Setup* y luego el método *Returns* del objeto *Mock*, se le indica que cuando se realice la llamada al método *GetUser* (definido por el objeto que implementa la interfaz *IUserDAO* con el mismo identificador definido en la variable *userId*), este devolverá el valor del objeto que representa al usuario definido en la variable *userDTO*.

```
[TestMethod]
| 0 referencias
public void GetUser_UserExists_ReturnUser()
{
    // Arrange

    Guid userId = Guid.NewGuid();
    string userName = "Alberto";
    int userAge = 25;

    UserDTO userDTO = new UserDTO()
    {
        Id = userId,
        Age = userAge,
        Name = userName,
    };

    userDaoMock.Setup(x => x.GetUser(userId))
        .Returns(userDTO);
```

Figura 3.5: fragmento *Arrange* de método de prueba

En la figura 3.6 se puede observar las partes *Act* y *Assert* del método de prueba. La parte *Act* se encarga de ejecutar el método a comprobar en el controlador y guardar el valor devuelto. Por último, en la parte *Assert* se comprueba que ese valor devuelto se corresponda a los valores asignados a las variables del *Arrange*, confirmando que se ha devuelto el usuario esperado. Luego, utilizando una de las propiedades del objeto *Mock* definido, se obtienen las invocaciones que se han realizado para esta clase simulada, es decir, las llamadas que se han realizado a los métodos de dicha clase. Para este caso se

comprueba que se haya realizado una única llamada al método *GetUser*, utilizando un único parámetro que tiene que ser el identificador asignado en el *Arrange* del método de prueba.

```
// Act

Dictionary<string, object> response = userController.GetUser(userId);

// Assert

UserDTO user = ((Dictionary<string, UserDTO>)response["data"])["user"];

Assert.AreEqual("success", response["status"]);
Assert.AreEqual(userId, user.Id);
Assert.AreEqual(userName, user.Name);
Assert.AreEqual(userAge, user.Age);

IEnumerable<IInvocation> invocations = userDaoMock.Invocations;

Assert.AreEqual(1, invocations.Count());

IInvocation userControllerInvocation = invocations.FirstOrDefault();
Assert.AreEqual("GetUser", userControllerInvocation.Method.Name);

Assert.AreEqual(1, userControllerInvocation.Arguments.Count());
Assert.AreEqual(userId, userControllerInvocation.Arguments.FirstOrDefault());
```

Figura 3.6: fragmentos *Act* y *Assert* de método de prueba

Para finalizar, en la figura 3.7 se puede ver el resultado de la ejecución de los métodos de prueba creados. Al ejecutarse, estos *tests* indican que el resultado de la ejecución es el esperado.

Prueba	Duración
MoqExampleTests (3)	471 ms
MoqExampleTests (3)	471 ms
UserControllerTests (3)	471 ms
GetUser_ExceptionIsThrown_ErrorMessageIsReturned	467 ms
GetUser_UserExists_ReturnNullUser	1 ms
GetUser_UserExists_ReturnUser	3 ms

Resumen del grupo
MoqExampleTests
Pruebas en grupo: 3
Duración total: 471 ms
Salidas
3 Correcta

Figura 3.7: ejecución de los métodos de prueba creados

3.2 DSL

Los Lenguajes de Dominio Específico (DSL, por sus siglas en inglés) son notaciones pensadas para usarse con una finalidad única, esto se contrapone a lenguajes similares como UML [16] que se enfocan en un uso más general. Para un DSL se han de definir los elementos que componen los modelos y las posibles relaciones que puede haber entre dichos elementos. Además, por lo general, la notación utilizada en estos lenguajes suele ser gráfica para facilitar su uso por parte de los usuarios y permitir visualizar el dominio del problema de una forma más cercana a la realidad.

Para el desarrollo del microservicio de migración se han utilizado las herramientas creadas por parte de la empresa para su uso interno, elaboradas a partir del SDK de Visual Studio [17]. Este SDK permite crear DSL, llamados metamodelos [18] que más tarde se podrán usar para la creación de modelos, y así facilitar el desarrollo de código.

Los metamodelos son modelos que definen los elementos de un lenguaje, sus relaciones y las reglas o restricciones sobre estos. El SDK proporciona una interfaz con elementos gráficos para la creación de estos metamodelos, que permiten crear Lenguajes de Dominio Específicos.

El uso de estas herramientas permite que para la creación del nuevo producto se siga un Desarrollo Dirigido por Modelos (o MDD, por sus siglas en inglés), donde a partir de plantillas para la generación de código, los programadores puedan ser capaces de generar de forma automática una estructura de código lista para ser implementada y la creación de *tests*. Algunas de las ventajas que proporciona el uso de modelos son [19]:

- Mejora sobre la productividad y la comprensión del *software* a desarrollar. Al utilizar representaciones abstractas, normalmente gráficas del sistema a crear, facilita el entendimiento de los conceptos del problema a resolver, y permite visualizarlo de forma más general. Esto hace que, por lo general, el tiempo necesario para la construcción del *software* disminuya.

- Facilidad para el mantenimiento y evolución del código. Al utilizar modelos para el desarrollo es más fácil crear y mantener las distintas partes que componen el producto *software*, pues los modelos permiten detectar más fácilmente los posibles errores o carencias del diseño.

- Dirigir la implementación del código. Al generar un esqueleto base, los programadores pueden enfocarse en desarrollar la parte del código referente a la lógica del producto, sin necesidad de crear la estructura desde cero.

- Sirven como proceso de documentación y comunicación. En lugar de utilizar otras herramientas para especificar el funcionamiento del sistema, esta información se puede añadir directamente sobre los modelos creados.

3.3 Microservicios

Cuando se habla de microservicios, normalmente se hace referencia a una arquitectura de programación donde cada uno de los componentes o servicios funciona de forma independiente, que pueden, incluso, estar desarrollados en tecnologías diferentes.

Para entender mejor las ventajas que ofrece una arquitectura basada en microservicios, es conveniente compararla con la arquitectura de una aplicación monolítica, debido a que los microservicios se plantearon como una mejora sobre esta última.

3.3.1. Arquitectura monolítica

Una arquitectura monolítica es el diseño tradicional de las aplicaciones *software*, en la cual todo el código o sistema se encuentra dentro del mismo proyecto. Esta arquitectura, aunque normalmente se divide en varias partes internamente, estas son completamente dependientes entre sí. Este diseño puede presentar algunas ventajas como las siguientes [20] [21]:

- Las aplicaciones son más sencillas y rápidas de desarrollar durante las primeras fases. Esto es debido a que toda la lógica de la aplicación se encuentra dentro del mismo proyecto, por lo que no es necesario preocuparse por la comunicación entre distintas partes, más allá de las dependencias entre las posibles capas de la aplicación.
- Simplicidad en el despliegue y la ejecución de la aplicación, pues al contar con un único componente estas tareas resultan más sencillas que si la aplicación estuviese compuesta por varias partes. También el rendimiento de una aplicación con una arquitectura monolítica suele ser superior al de una aplicación con una arquitectura de microservicios.
- Facilidad para la creación de pruebas y depuración. Como todo el código está contenido en un único proyecto, las pruebas requieren de poco tiempo para su desarrollo, al no depender de código externo y es más sencillo localizar posibles errores durante la ejecución.
- El coste de desarrollo suele ser más bajo que con otras arquitecturas al requerir menor tiempo para llevar a cabo la elaboración del producto, por lo menos en las fases iniciales.

Por otra parte, esta arquitectura también plantea diferentes desventajas, estas se acentúan conforme el proyecto aumenta en tamaño, pudiendo provocar que cualquier cambio o aumento en la demanda suponga un gran problema para el funcionamiento del producto. Las principales desventajas son:

- Problemas de escalabilidad. Cuando la demanda es muy grande, por ejemplo en un aumento de peticiones por parte de los usuarios, es muy difícil o imposible ampliar este único componente individual.
- Dificultad de desarrollo. Si el proyecto es muy grande, es complicado añadir nuevas funcionalidades, pues es necesario que se adapten correctamente con el resto de la lógica de la aplicación. También que esté formado por un único componente provoca que este resulte un único punto de fallo, en el que si ocurre un error, este puede afectar a la disponibilidad de toda la aplicación.

Estas desventajas provocan que este tipo de arquitectura no sea adecuada para aplicaciones de gran tamaño, o que necesiten ser escalables para atender las peticiones de miles de usuarios, aunque sigue siendo una opción interesante para proyectos pequeños o productos que no vayan a necesitar atender un gran número de peticiones.

3.3.2. Arquitectura basada en microservicios

En contraposición a la arquitectura monolítica, una aplicación diseñada con una arquitectura basada en microservicios se encuentra dividida en varias partes o servicios que funcionan de forma independiente. Esta arquitectura busca dividir las tareas en distintos componentes, para facilitar el desarrollo de distintas funcionalidades sin que este proceso afecte de forma directa a los otros microservicios pudiendo trabajar simultáneamente en cada uno de ellos. Para funcionar de forma conjunta, las distintas partes se comunican entre sí a través de la API de cada microservicio, permitiendo realizar llamadas y peticiones para ejecutar procesos o transmitir datos. Las ventajas que aporta este tipo de estructura son las siguientes:

- Desarrollo de los microservicios en distintas tecnologías y lenguajes. Al ser completamente independientes entre sí, cada uno de estos servicios puede estar implementado con la tecnología o lenguaje que más se adapte a sus necesidades.
- Las partes que componen la aplicación son escalables, esto permite que cada servicio sea instalado o replicado en varios servidores para adaptarse a un aumento en las peticiones. Además, cada despliegue de un servicio es independiente, por lo que un fallo no ocasionará que la aplicación pare su ejecución.
- Es más fácil de mantener cuando el producto tiene un tamaño considerable, ya que en lugar de buscar y corregir un defecto dentro de un gran proyecto o modificarlo de forma íntegra cada vez que se quiera añadir una nueva funcionalidad, es más sencillo realizar estas tareas cuando este está dividido en partes más pequeñas e independientes.

El principal problema de esta estructura se encuentra en el aumento de la complejidad en el despliegue de los servicios y de las acciones que se realizan, esto se debe principalmente a que las tareas a realizar deben pasar por diferentes módulos, por lo que el desarrollo suele requerir más tiempo y recursos. Para simplificar estas tareas se suelen utilizar herramientas como Docker [3] para el despliegue de software mediante contenedores y Kubernetes [4] para gestionar dichos contenedores.

3.3.3. Tabla comparativa

A continuación, en la figura 3.8 se comparan entre sí las dos arquitecturas explicadas.

	Arquitectura monolítica	Arquitectura basada en microservicios
Complejidad	Baja	Alta
Escalabilidad	Baja	Alta
Fiabilidad	Baja	Alta
Lenguaje	Único	Varios
Velocidad de desarrollo	Más lenta cuanto más grande sea el proyecto	Lenta al principio, rápida una vez creados los servicios
Mejor para	Proyectos pequeños con bajos requerimientos	Productos que requieran ser escalables.

Figura 3.8: tabla comparativa de arquitecturas presentadas

[3]Web oficial de Docker: https://www.docker.com/.
[4]Web oficial de Kubernetes: https://kubernetes.io/.

CAPÍTULO 4
Desarrollo de la solución

En este capítulo se presenta el desarrollo del microservicio de migración. Primero se expone a grandes rasgos el planteamiento del proceso de migración y cómo se quieren procesar las distintas incidencias que puedan surgir al transformar los datos. Después se presenta la especificación de requisitos de dicho microservicio. A continuación, se detalla el diseño y la estructura del proyecto. En las secciones siguientes se comenta el proceso de programación llevado a cabo, junto con las pruebas aplicadas de la solución y los patrones de migración utilizados. Por último, se describe la metodología seguida para desarrollar el microservicio.

4.1 Planteamiento general

Antes de comenzar con una descripción más técnica de las características que tiene el microservicio, se va a presentar de una forma más general el propósito que se quiere cumplir, con el desarrollo de este proyecto. Además de explicar a grandes rasgos algunos detalles de su implementación y como se relaciona dentro del el contexto del producto.

Esta sección no explicará el desarrollo en sí mismo, pero sí que ayudará a entender varias de las decisiones que se han decidido tomar y que han afectado al desarrollo.

4.1.1. Estado del producto actual

Para entender mejor la necesidad de la creación de esta herramienta de migración de datos, conviene explicar primero cuál es el estado del producto actual, comparándolo con el nuevo, y las razones por las que se ha decidido crear una nueva estructura a la que migrar los datos existentes.

El primer motivo es la necesidad de crear una estructura diseñada específicamente para las funcionalidades del nuevo producto. Esto es debido a que, aunque ambas estructuras tendrán bastantes similitudes, la estructura de datos del producto actual cuenta con varias limitaciones, lo que impide poder trabajar con los datos de forma ordenada y eficiente.

La segunda razón es la imposibilidad de realizar tareas o desarrollar funcionalidades de *machine learning* o análisis de datos. Esto es debido a la estructura de guardado de datos actual, en la que se decidió optar por un enfoque que permitiese a la aplicación adaptarse lo más posible a cada cliente, permitiendo modificar la mayor parte de los datos dentro de esta.

Por el contrario, para el nuevo producto se decidió que realizar este tipo de tareas de análisis es esencial, en consecuencia, en la nueva estructura de datos se ha optado por

un enfoque parcialmente más rígido. Esta estructura mantiene la posibilidad de añadir o personalizar ciertos datos, pero obliga a que estos estén enlazados con datos preexistentes. Esto se comentará en mayor profundidad en el apartado 4.4.2 donde se comentará la solución realizada como un patrón de migración de datos.

El tercer y último motivo por el que se ha decidido modificar la estructura de los datos porque se quiere organizar de forma diferente los centros físicos de los clientes dentro de la aplicación. Esto tiene el objetivo de facilitar y hacer más flexible la división lógica de dichos centros dentro de la aplicación.

4.1.2. Planteamiento de la migración

Sobre la migración, existen varias particularidades que cabe destacar para poder entender como se planea realizar el proceso de migración de datos.

La primera es que el proceso debe ser diferido en el tiempo debido a la gran cantidad de datos a migrar y las posibles incidencias a solucionar que aparecerán durante el proceso. Algunos datos tienen dependencias entre sí, esto impedirán que el proceso avance en caso de que existan incidencias que no estén resueltas por los usuarios del producto. Por tanto, los errores que surjan durante la migración de datos pausarán el proceso hasta que estas incidencias sean solucionadas.

La segunda es que para impedir inconsistencias durante el proceso de migración, el nuevo producto estará bloqueado para permitir solo la lectura de datos, pero el producto actual podrá seguir utilizándose mientras se realiza la migración de los datos. Al poder seguirse usando la aplicación, se deberá poder gestionar dentro del producto y del algoritmo de migración la existencia de datos que hayan sido modificados, para que estos vuelvan a ser procesado y actualizados en el nuevo producto.

La tercera trata sobre cómo gestionar las posibles incidencias durante el proceso de migración. Estas incidencias representan posibles errores, inconsistencias o conflictos con los datos, detectados durante la migración, y son guardadas en un registro que indica el estado de migración de cada dato. Las incidencias deberán ser resueltas por los usuarios para permitir la transferencia de todos los datos. Hay varios casos que pueden crear estas incidencias, esto pueden ser datos que no cumplan ciertas validaciones en el nuevo producto o datos sobre los que se detecten ciertos defectos durante su migración. Por ejemplo, un dato que originaría una incidencia sería la detección de un valor anómalo, como podría ser un registro de toma de temperatura de un residente, en el que se escribió que en un momento dado ese residente tenía una temperatura corporal de 370 °C, lo cual es incorrecto y debe gestionarse.

Por último, debido a la naturaleza sociosanitaria del producto, es importante tener todos los datos en cuenta, y que sean los propios usuarios los que, en caso de incidencia, decidan si quieren mantener esos datos, eliminarlos o arreglar la inconsistencia encontrada. Por este motivo y el anterior, se decidió crear un histórico de datos migrados en el que poder tener un registro de registros migrados e incidencias sobre estos, y señalar o distinguir aquellas incidencias que estén solucionadas y las que no, además de almacenar la información necesaria sobre todas las incidencias. Este tema será explicado en mayor profundidad en el apartado 4.3.

4.1.3. División de tareas

Es necesario comentar que para el desarrollo de este proyecto desde el propio departamento se quería dividir las tareas dentro de este. Por una parte, se realiza el estudio más funcional del dominio de los productos y el análisis de su diseño interno, sobre como es-

tán estructurados los datos en ambos proyectos y como se relacionan las estructuras de los datos de entre ambos productos. Por la otra parte, se llevan a cabo las tareas más técnicas del desarrollo del algoritmo de migración, es decir, el desarrollo del propio código en el que gestionar tanto la migración, como el histórico de datos migrados y las posibles incidencias.

Como se verá en el apartado 4.3, para separar las tareas funcionales de las más técnicas se han utilizado las herramientas de modelado aplicando técnicas de MDD. Gracias a estas se ha creado una estructura en los modelos que permite crear los métodos a utilizar por parte de los programadores, en los que después de generar código, se puede programar las partes más específicas dentro de la estructura de clases creada. También los analistas pueden trabajar conjuntamente creando modelos de *tests* en los que probar, sin necesidad de implementar código, que la conversión de los campos programada es la correcta.

4.2 Especificación de requisitos

4.2.1. Casos de Uso

A continuación, se muestran las diferentes funcionalidades, que debe poder desempeñar el microservicio de migración, presentadas como Casos de Uso. Estos Casos de Uso contarán con un identificador, para facilitar su alusión, además de un nombre y una descripción.

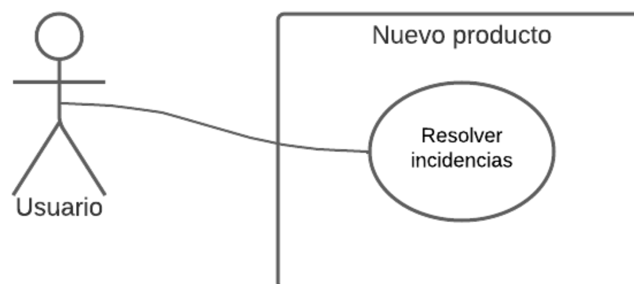

Figura 4.1: Casos de Uso del nuevo producto

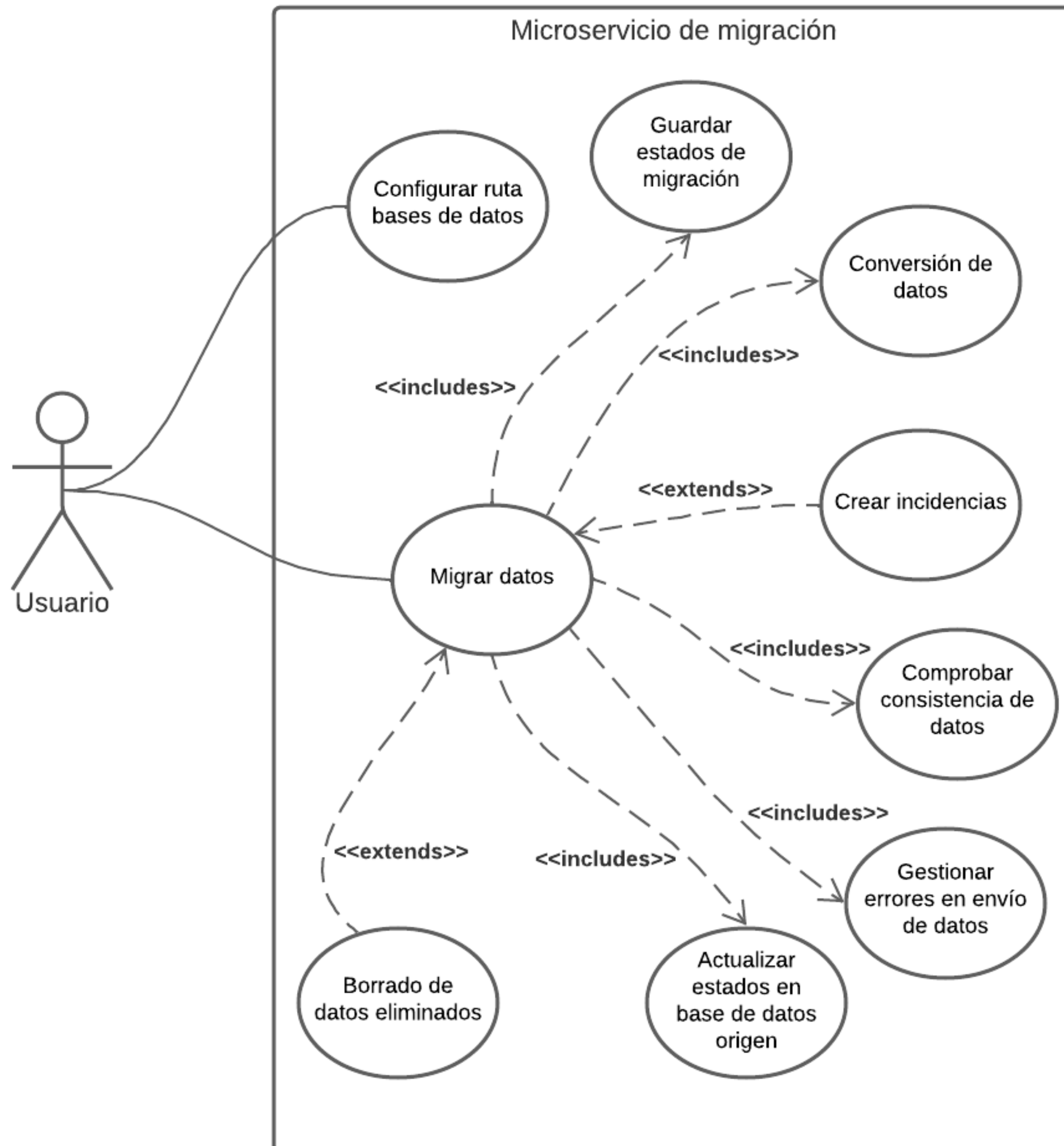

Figura 4.2: Casos de Uso del proceso de migración

Identificador	CU01
Figura	4.2
Nombre	Configurar ruta y obtener registros de distintas bases de datos
Actor	Usuario de la aplicación
Descripción	Es necesario crear una interfaz en la que especificar una configuración inicial para poder realizar la migración. Dentro de esta configuración deberá aparecer como mínimo la opción de especificar la ruta donde se encuentran las bases de datos a migrar y seleccionar en cuáles de estas se encuentran los datos que se quieren migrar, además se ha de permitir obtener estos datos desde distintas bases de datos para el mismo proceso, pues no todos se encuentran en la misma.

Identificador	CU02
Figura	4.2
Nombre	Migrar datos
Actor	Usuario de la aplicación
Descripción	Un usuario del producto actual o el sistema inicia la migración de los datos pasando como parámetros a la llamada al microservicio la ubicación de las bases de datos a migrar y el nombre de esta. A continuación, el proceso de migración obtendrá los datos y procederá a realizar la conversión y envío de estos al microservicio destino.

Identificador	CU03
Figura	4.2
Nombre	Guardar estados de migración
Actor	Usuario de la aplicación
Descripción	Con el objetivo de poder conocer y distinguir los datos migrados de los que no lo han sido, ya sea porque ha ocurrido un error durante su migración o porque esta todavía no se ha ejecutado para dicho dato, debe existir un registro que ayude a evitar migrar por error dos veces el mismo dato o que se queden algunos sin hacerlo. También este histórico o registro deberá servir para almacenar las incidencias generadas.

Identificador	CU04
Figura	4.2
Nombre	Conversión de datos
Actor	Usuario de la aplicación
Descripción	En el nuevo producto existen muchos apartados que han sufrido modificaciones para adaptarlos a las nuevas características. Es por esto conlleva que van a tenerse que crear procesos de migración específicos para tipo de registro con el objetivo de poder manipular los datos que contenga.

Identificador	CU05
Figura	4.2
Nombre	Crear incidencias
Actor	Usuario de la aplicación
Descripción	Si durante la transformación y el envío de los registros leídos, se detectasen inconsistencias o errores en los datos, o validaciones fallidas durante su creación en el microservicio de destino, se procederá a crear incidencias que registren información sobre estas y su estado actual.

Identificador	CU06
Figura	4.2
Nombre	Comprobar consistencia de los datos
Actor	Usuario de la aplicación
Descripción	Para muchos de los registros existen datos que son calculados internamente a partir de otros datos del mismo registro, como podría ser el resultado de una encuesta. Para asegurar la consistencia de los datos durante su migración, habrá que comprobar que el valor de ese dato calculado mantenga el mismo valor en el nuevo producto, ya que la forma de calcular el valor podría ser distinta. En caso de conflicto, deberá crearse una incidencia como se verá en el apartado 4.3.2.

Identificador	CU07
Figura	4.2
Nombre	Gestionar errores en el envío de datos
Actor	Usuario de la aplicación
Descripción	En el nuevo producto existen validaciones que no existen en el producto actual, como pueden ser rangos de valores permitidos para ciertos datos. Al igual que en el CU06, deberá crearse una incidencia en caso de conflicto al intentar migrar dicho dato al nuevo producto, indicando el origen del error. Esta información será explicada en más detalle en el punto 4.3.2.

Identificador	CU08
Figura	4.2
Nombre	Actualizar estado de datos en base de datos origen
Actor	Usuario de la aplicación
Descripción	Durante todo el proceso de migración los usuarios podrán utilizar el producto actual de forma habitual, es por esta razón que para distinguir los datos que hayan sido migrados de los que no lo hayan sido o hayan sido modificados después de una migración se creará para cada tabla de cada tipo de dato una nueva columna que indicará si el registro debe ser migrado, y será el proceso de migración el que se encargará de actualizar dicho valor.

Identificador	CU09
Figura	4.2
Nombre	Borrado de datos eliminados en el producto actual
Actor	Usuario de la aplicación
Descripción	Deberán de gestionarse aquellos registros de datos que hayan sido borrados durante el uso habitual por parte de los usuarios del producto actual, para ello, en caso de que dichos registros hayan sido migrados y creados en el nuevo producto, deberán borrarse durante el nuevo proceso de migración que se ejecute.

Identificador	CU10
Figura	4.1
Nombre	Resolver incidencias
Actor	Usuario del producto
Descripción	Desde la interfaz del nuevo producto los usuarios podrán visualizar las incidencias generadas durante cada proceso de migración, además, en dicha interfaz, se ofrecerá la posibilidad de resolver algunos de los errores detectados, actualizando el estado de las incidencias una vez resueltas. También esta información permitirá que los usuarios puedan resolver el resto de errores modificando los datos en el producto actual.

4.2.2. Requisitos no funcionales

Por último, es necesario mencionar aquellas propiedades o restricciones que este microservicio debe tener para ofrecer a los clientes un sistema de calidad. Para analizar estas condiciones se utilizan los llamados requisitos no funcionales. Estos se refieren a aquellos requisitos que debe cumplir el producto y que no representan el funcionamiento del mismo. Para facilitar su análisis, la ISO/IEC 25010 [22], también llamada SQuaRE (*System and software Quality Requirements and Evaluation*), organiza estos requisitos no funcionales en ocho características principales, e internamente, cada una de ellas define a su vez algunas subcaracterísticas más específicas.

A continuación, se detallan los requisitos no funcionales que se deben satisfacer, indicando para cada uno de ellos un identificador para su posterior mención, un nombre, la característica a la que se hace referencia y una descripción.

Identificador	RNF01
Nombre	Ejecución del proceso en *background* de forma continua
Característica	Compatibilidad
Descripción	Para evitar bloquear la interfaz, el proceso debe ejecutarse en segundo plano y de forma continua, es decir, una vez termine la ejecución en curso debe de comenzar automáticamente de nuevo el proceso de migración. El objetivo de esto es migrar aquellos registros cuyos conflictos hayan sido solucionados y los nuevos datos que hayan podido introducirse.

Identificador	RNF02
Nombre	Proceso realizado por lotes
Característica	Adecuación Funcional
Descripción	La migración debe poder ser capar de ejecutarse por partes o lotes, esto se debe a que la resolución de los conflictos o incidencias requiere la intervención de un usuario de la aplicación, ya que en el momento de la generación del conflicto el usuario podría no estar presente. Por tanto, el proceso debe poderse interrumpir sin generar ningún tipo de fallo, apoyándose para ello en el histórico del CU03.

Identificador	RNF03
Nombre	Permitir migración diferida en el tiempo
Característica	Eficiencia de desempeño
Descripción	La migración de los datos no se realizará de forma única, sino que probablemente se necesitará de varios días para llevarse a cabo debido a la necesidad de solucionar las distintas incidencias que se hayan producido. Por esta razón, el tiempo que tarde la migración en completarse no debe de afectar de ninguna forma al resultado final y aunque se realicen modificaciones en los datos.

Identificador	RNF04
Nombre	Migración eficiente
Característica	Eficiencia de desempeño
Descripción	Aunque no se trata de algo imprescindible, es importante que la migración de los datos pueda llevarse a cabo en el menor tiempo posible. Para poder decir que la migración se está ejecutando de forma eficaz, el proceso no debería abarcar más de una semana para migrar todos los datos de las empresas más grandes, suponiendo que no han surgido conflictos. Esto facilitará resolver las incidencias lo antes posible y permitirá a los usuarios empezar a usar el nuevo producto en un tiempo mínimo.

Identificador	RNF05
Nombre	Utilización mínima de recursos
Característica	Eficiencia de desempeño
Descripción	Debido a la arquitectura de microservicios que compondrá el nuevo producto, es importante que el proceso de migración de datos utilice la mínima cantidad de recursos posible para llevarse a cabo con el objetivo de evitar sobrecargas en el hardware destino y la ralentización del resto de procesos o microservicios. Para verificar que este proceso es eficiente, desde la perspectiva de un usuario que utiliza la aplicación, no debería suponer ningún cambio visible cuando el proceso se esté ejecutando.

Identificador	RNF06
Nombre	Prevenir acceso a datos no autorizados
Característica	Seguridad
Descripción	Teniendo en cuenta la naturaleza sociosanitaria del producto y de los datos, es indispensable que durante la resolución de los conflictos por parte del usuario, solo el que tenga un rol que permita el acceso a dichos datos sea capaz de ver y resolver las incidencias encontradas. Esto ha de ser así por dos motivos, el primero es que estas incidencias pueden contener información de datos confidenciales de los clientes, por tanto, no todos los usuarios de la aplicación deberían poder verlos, y el segundo motivo es que para resolver algunas incidencias se necesita modificar estos datos, por lo que solo un usuario con el rol indicado debería ser capaz de realizar cambios en estos.

Identificador	RNF07
Nombre	Tecnología y estructura definidas
Característica	Mantenibilidad
Descripción	En el desarrollo de este proyecto deben utilizarse las mismas tecnologías y estructurar el proyecto de la misma forma que en el resto de microservicios del producto, con el objetivo de facilitar la comprensión del código por parte del resto de miembros de la empresa. Por tanto, la estructura general del proyecto debería poder ser entendida por una persona del equipo de desarrollo en menos de una hora.

4.3 Diseño y estructura

En esta sección se va a explicar la estructura del microservicio y su diseño. Por un lado, se explicará la estructura por capas que, como el RNF07 del apartado 4.2.2 sugiere, que es la estructura que todos los microservicios del producto deben tener para mantener una organización común. Asimismo, se comentan algunos detalles específicos del microservicio de migración de datos sin entrar en profundidad todavía en su implementación interna. Por otro lado, se explica el rol que desempeñan las herramientas de modelado y generación de código en el proceso de elaboración de este proyecto, y como permiten que varías personas puedan desempeñar roles diferentes en el desarrollo.

4.3.1. Estructura de la solución

Empezando por la estructura interna del proyecto, que como se ha comentado, tiene que ser semejante a la de los otros microservicios, se va a explicar cada una de las 8 capas de las que está compuesto este microservicio de migración.

- **Contratos:** En esta capa se sitúan las interfaces que contienen las acciones del *backend* y que pueden ser invocadas desde el exterior, por ejemplo desde otros microservicios. Aún más importante, es en esta capa donde se ubican los DTO, estos son objetos utilizados para la transferencia de datos. Estos son utilizados para desacoplar los servicios que se ofrecen en la API de su representación interna por parte del sistema de dichas entidades. Es decir, estos DTO permiten utilizar de forma interna y devolver al cliente objetos con un formato diferente al son almacenados

- **Aplicación:** Esta capa contiene el código generado automáticamente de la DSLTools de aplicación (explicada a continuación en el apartado 4.3.2). Este código se encarga de dar soporte a las operaciones CRUD [23] y de comprobar los permisos del usuario sobre las entidades y campos del microservicio.

- **Lógica:** En esta capa se encuentra tanto el código creado por las herramientas de modelado y generación, como el código que implementa el programador y que complementa a este código generado. Para este microservicio, esta es una de las capas con mayor tamaño y que se desarrolla su función principal, esto es, la lectura, conversión y envío de los datos al microservicio de destino.

- **Dominio:** En esta capa se encuentran las entidades de dominio modeladas en la DSL Tool de dominio (explicada a continuación en el apartado 4.3.2). Además, contiene el código relacionado con campos calculados y validaciones, aunque estos no son utilizados en este microservicio.

- **Persistencia:** Es la capa que se encarga del acceso a base de datos para la persistencia de estos. En el caso del microservicio de migración, esta capa será usada exclusivamente para la gestión de conflictos e incidencias y registrar que datos han sido migrados o cuáles han ocasionado errores.

- **Servicios:** Se utiliza como punto de entrada a las acciones o métodos que expone el microservicio a partir de controladores. Cada método se define a través de acciones HTTP.

- **Proxy:** Contiene los métodos necesarios para invocar al *backend*, generando para ello una llamada HTTP. Esta es la capa que es referenciada por el *frontend* u otros microservicios a través de un paquete NuGet [24] con el objetivo de comunicarse con este microservicio.

- **Referencias externas:** Aquí es donde se gestionan los servicios a consumir, utilizando esta capa se pueden abstraer las dependencias y evitar problemas con dependencias cíclicas.

Ahora que se han presentado las distintas capas que componen el microservicio de migración, en la figura 4.3 se muestra un esquema de como se ubican y comunican entre sí.

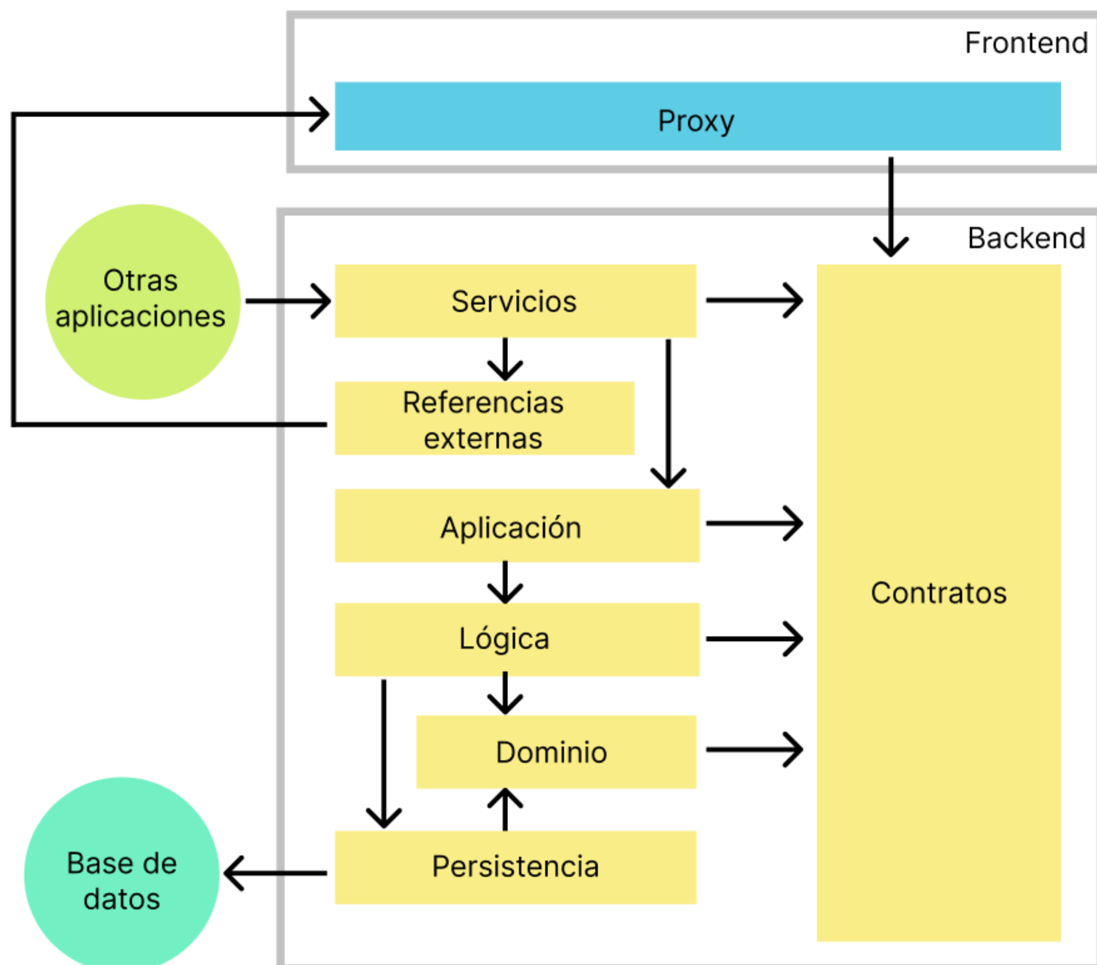

Figura 4.3: esquema de la arquitectura de las capas del microservicio

4.3.2. Estructura de modelos DSL

Algunas de las capas de la solución son automáticamente generadas a partir de los modelos creados en las herramientas DSL desarrolladas por parte de la empresa a partir del SDK que ofrece Visual Studio para la creación de metamodelos. Para generar el código a partir de los objetos modelados en estas herramientas a las que nos referiremos como DSL *Tools*, se necesita hacer uso de plantillas [25], las cuales son ejecutas en la aplicación con el objetivo de generar cadenas de texto, el cual en nuestro caso se trata de código fuente. Entrando más en detalle, todas las capas se generan en mayor o menor medida mediante el uso de estas plantillas, y es mediante el uso de los modelos creados a partir de las DSL *Tools* que el programador puede generar el esqueleto del código específico a desarrollar en el microservicio.

El propósito principal del uso de estas herramientas es aplicar un desarrollo dirigido por modelos (MDD), que permite a los usuarios construir aplicaciones complejas mediante abstracciones para facilitar su elaboración. Utilizando este enfoque junto con la generación automática de código, no solo se permite que estas representaciones sirvan como esquemas para entender mejor el comportamiento del código a desarrollar, sino que a su vez sirve para generar una estructura que facilite al programador la implementación de las distintas funciones del producto, agilizando a su vez este proceso.

Siguiendo este enfoque, junto con las herramientas de la empresa, se siguen los siguientes pasos:

- Primero se crean los modelos añadiendo y modificando los objetos que se necesitarán para generar la estructura del código.
- Segundo se genera el código mediante *scripts* que hacen uso de las plantillas de generación de código, que, como su nombre indica, producen el código que compone las distintas capas.
- Por último, aunque el código generado facilita una estructura y una lógica base, es una persona la que debe programar las partes específicas de la implementación de las funcionalidades de la aplicación.

A continuación, se explica en mayor detalle el primer paso, y cómo se ha planeado el diseño del microservicio a partir de estas DSL *Tools*. Para esta explicación se tratarán únicamente las dos herramientas que se han usado para el microservicio de migración, la DSL *Tool* de dominio y la DSL *Tool* de aplicación, además de estas dos existe la DSL *Tool* de interfaz de usuario, que no será explicada debido a que por el momento este microservicio no cuenta con interfaz, aunque en un futuro podría crearse para ayudar a los usuarios en la resolución de incidencias.

DSL Tool de dominio

Empezando por la DSL *Tool* de dominio, los modelos que esta herramienta puede crear son muy similares a los diagramas UML típicos, en estos modelos se representan las entidades que forman parte del dominio del microservicio. Además, utilizando la generación de código, estas entidades, junto a sus propiedades y relaciones que se definan en el modelo, serán representadas en una base de datos SQL, esto permitirá persistir los datos utilizados en el microservicio siguiendo esta misma estructura modelada por el programador.

En el caso del microservicio de migración, existe un único modelo de dominio, el motivo de esto es que en la base de datos perteneciente al propio microservicio, no se

van a almacenar los datos obtenidos del producto actual ni su conversión a datos del nuevo producto, sino que únicamente se almacenará lo definido en el RF01, es decir, un histórico de datos migrados y no migrados junto con la información de los conflictos o incidencias producidas durante la migración de los datos.

En la figura 4.4, se muestra dicho modelo de dominio, en este se pueden observar las distintas entidades que componen este registro de datos (de color azul), que relaciona aquellos datos del producto actual con los nuevos creados en el nuevo producto a partir de los identificadores de cada registro. También se pueden observar los enumerados (de color amarillo), que ayudan a estructurar la información con respecto al estado de la migración de los datos y de las incidencias creadas.

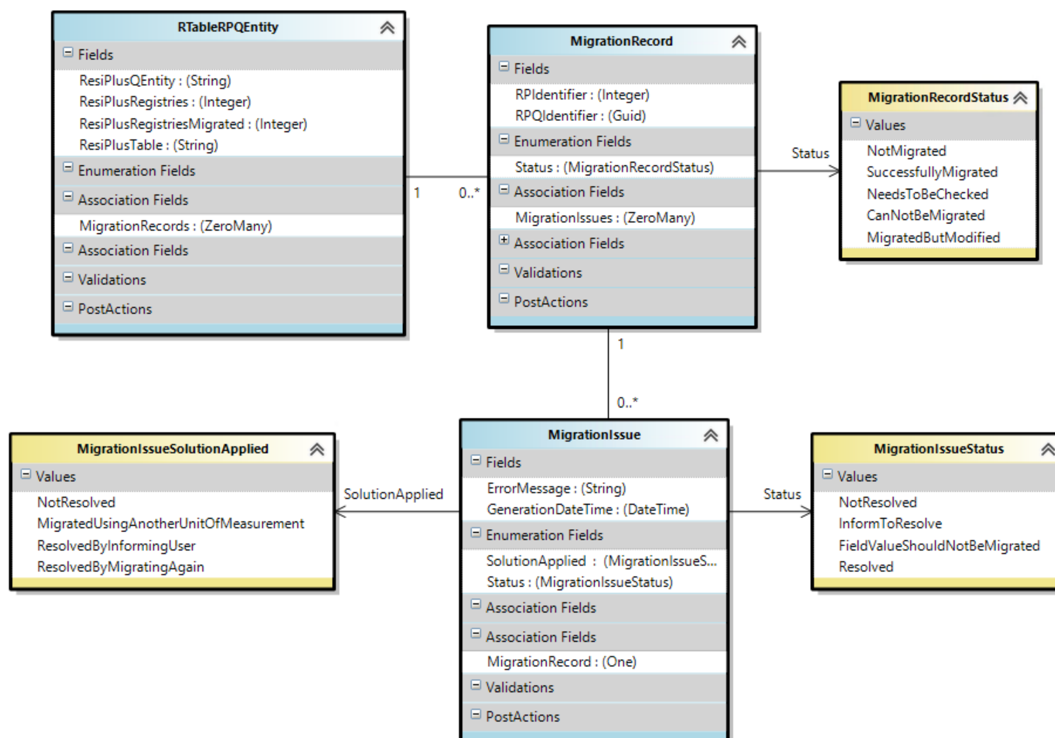

Figura 4.4: modelo de dominio

El diseño del histórico de migración de datos se representa a partir de las tres entidades que se ven en la figura 4.4:

- ***RTableRPQEntity*:** Esta entidad se encarga de relacionar las tablas de la base de datos del producto actual, origen de los datos, con las entidades modeladas en el microservicio de destino. También en esta entidad se guarda el número de registros existentes en la tabla origen y junto con el número de registros correctamente migrados, con el objetivo de poder conocer a grandes rasgos la cantidad de datos migrados y los que quedan por migrar.

- ***MigrationRecord*:** Esta entidad es la encargada de relacionar cada registro o dato de la base de datos origen con la instancia correspondiente de la entidad del microservicio objetivo, indicando para cada dato su estado de migración (migrado correctamente, con errores pendientes, sin migrar, etc.) utilizando para ello el enumerado llamado *MigrationRecordStatus*. Asimismo, cada instancia de *MigrationRecord* estará siempre relacionada con otra instancia de *RTableRPQEntity*, y la entidad *RTableRPQEntity* tiene una relación de cero a muchos con la entidad *MigrationRecord*, es decir,

cada instancia o registro de la entidad *RTableRPQEntity* está relacionada ninguna o varias instancias de la entidad *MigrationRecord*, al igual que una tabla de una base de datos SQL puede contener ninguna o varias filas o registros de datos.

- ***MigrationIssue*:** Esta última entidad se encarga de almacenar la información referente a cada incidencia y su estado actual, para ello almacena un mensaje de error, la fecha en la cual fue creada la incidencia, el estado de resolución y el tipo de solución que ha sido aplicada, utilizando para estas dos últimas propiedades los enumerados *MigrationIssueStatus* y *MigrationIssueSolutionApplied* respectivamente. Al igual que en el caso anterior, una instancia de *MigrationRecord* puede tener una o varias incidencias y una instancia *MigrationIssue* debe estar relacionado con una instancia de *MigrationRecord*.

Por último, en la figura 4.5 se muestra cómo el esquema representado en este modelo se muestra en una base de datos SQL, donde las entidades son las tablas y las propiedades son las columnas.

DSL Tool de aplicación

Continuando por la DSL Tool de aplicación, los elementos de los modelos creados por esta herramienta representan la lógica que será implementada en la aplicación y los tipos de datos u objetos que serán utilizados dentro de esta lógica.

Por lo general, dependiendo de los elementos que se introducen en cada modelo se suele dividir en dos tipos, el primero suele contener las llamadas acciones ad hoc y los DTO que suelen actuar como parámetros de dichas acciones, el segundo tipo se trata de modelos dedicados a contener elementos de *tests* que prueban el funcionamiento de dichas acciones ad hoc de los otros modelos. Además, dentro de las acciones ad hoc existen las acciones públicas que permiten ser llamadas mediante peticiones HTTP o desde otro microservicio, y las acciones privadas que solo pueden utilizarse desde el microservicio en cuestión. Para los DTO también existen varios tipos diferentes, pero en este proyecto solo se han utilizado de dos tipos, los *ParameterDTO* cuyos campos y nombres son creados por el programador y los *DefaultDTO* que representan una entidad de las existentes en los modelos de dominio.

A continuación, se va a mostrar el diseño de la solución mediante algunos de los modelos creados.

Empezando por la única acción ad hoc pública de la aplicación y único punto de entrada de la aplicación, en la figura 4.6 se muestra dicho elemento, el cual recibe como parámetro de entrada un *ParameterDTO*, este contendrá la información relativa a la ubicación del servidor y el nombre de las bases de datos de las que se procederá a extraer los registros para su migración. Un ejemplo de petición HTTP se puede ver en la figura 4.7, donde se utiliza Postman para realizar la llamada que ejecuta esta acción.

Dicha acción procede a ejecutar las siguientes acciones ad hoc privadas que se muestran en la figura 4.8, estas serán ejecutadas de izquierda a derecha dividiendo la migración en varias fases, que se comentarán en el apartado 4.4, y ejecutando las migraciones de cada entidad a migrar. Todas estas acciones reciben como parámetro de entrada una *ParameterDTO* con la información necesaria para conectar a la base de datos requerida.

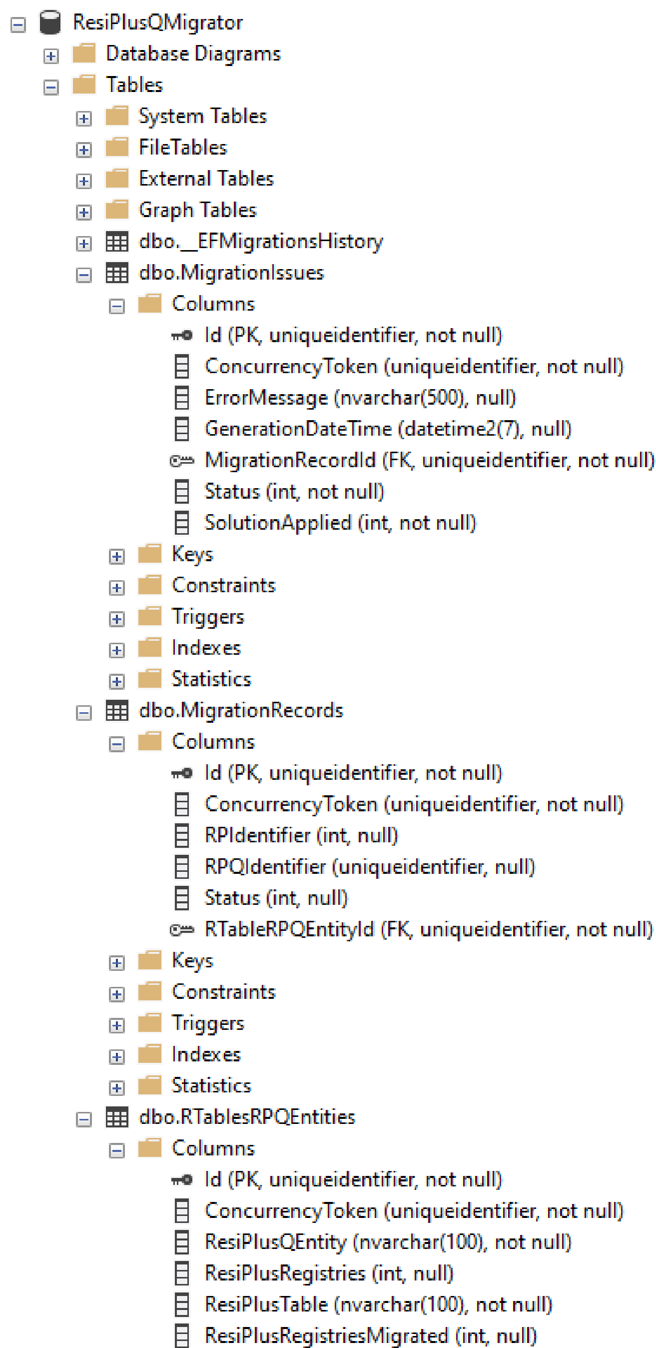

Figura 4.5: esquema de base de datos

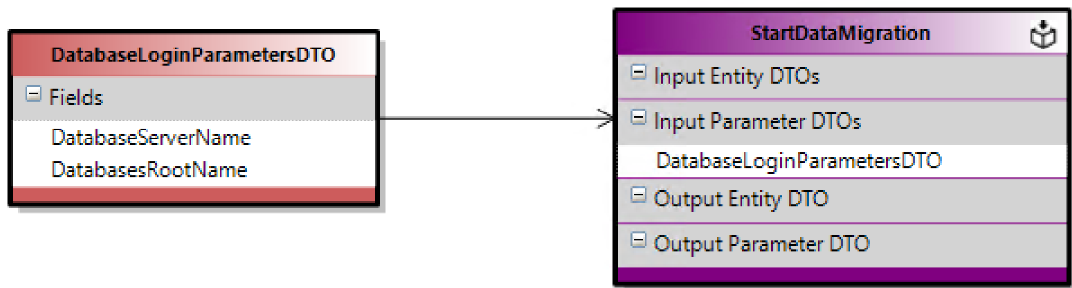

Figura 4.6: acción ad hoc pública del modelo de aplicación

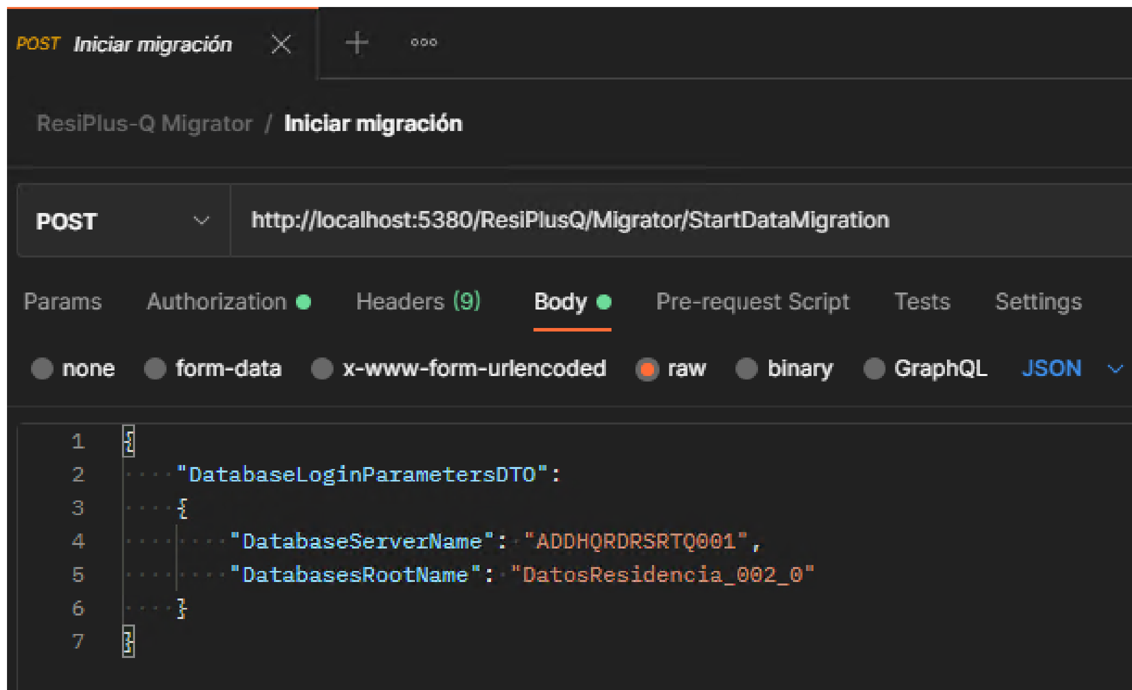

Figura 4.7: ejemplo de petición HTTP en Postman

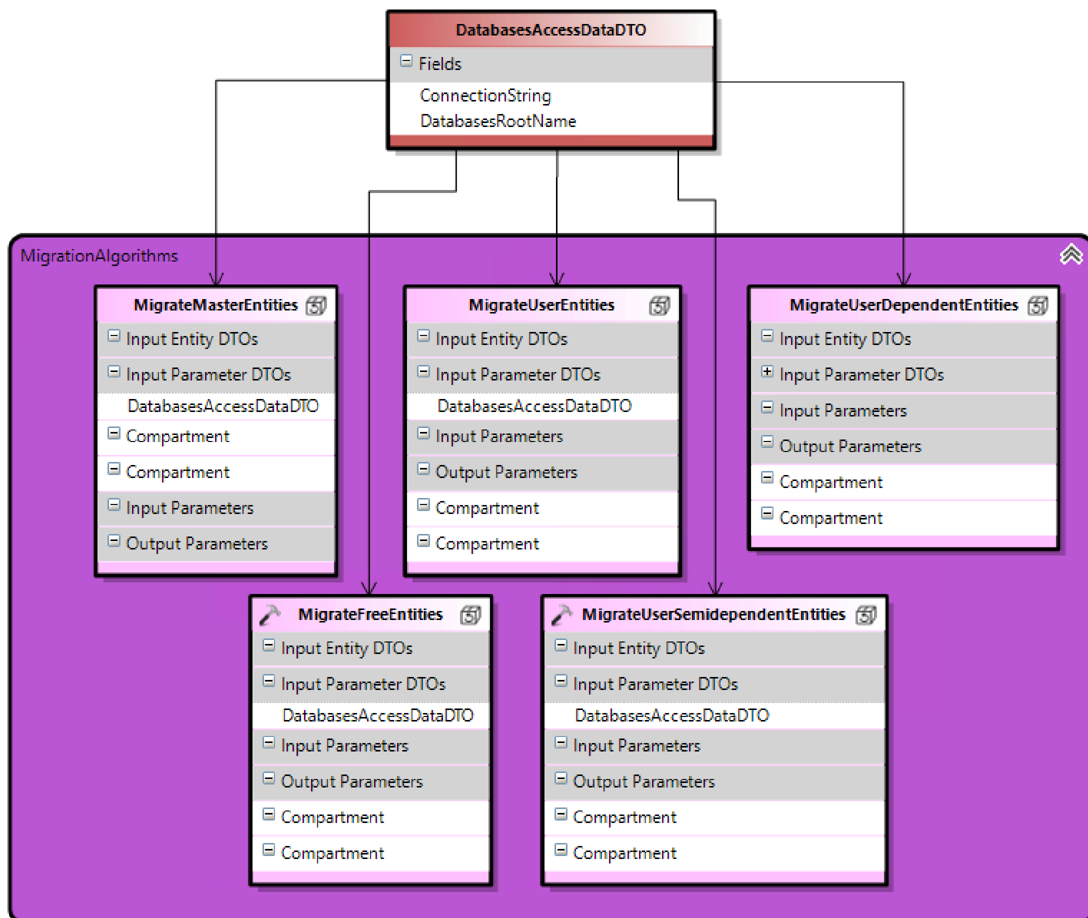

Figura 4.8: modelos con las cinco acciones principales de la migración

Por último, como se ha comentado, las cinco acciones anteriores llamarán a la migración de cada entidad, para diseñar este proceso se ha decidido crear cuatro modelos cuyos ficheros se muestran en la figura 4.10. El primero de ellos (figura 4.9), contiene dos acciones ad hoc por cada entidad a migrar, una de ellas es la encargada de ejecutar la migración y el otro se utiliza para facilitar la creación de *tests*, ya que implementa la misma lógica que el primero, pero pudiendo introducir datos sin necesidad de leerlos de una base de datos, el segundo modelo (figura 4.11) contiene una acción ad hoc que se utiliza para realizar la conversión de los datos de origen al formato requerido y enviarlos para su creación al microservicio de destino. Los dos restantes se tratan de modelos de *tests* que comprueban el correcto funcionamiento de las acciones modeladas en los otros archivos.

Para la estructura de estos ficheros se ha seguido una aproximación del RNF04, aplicando a la estructura de ficheros de los modelos de dominio del microservicio de destino, para facilitar la creación y mantenimiento de los modelos en la medida de lo posible.

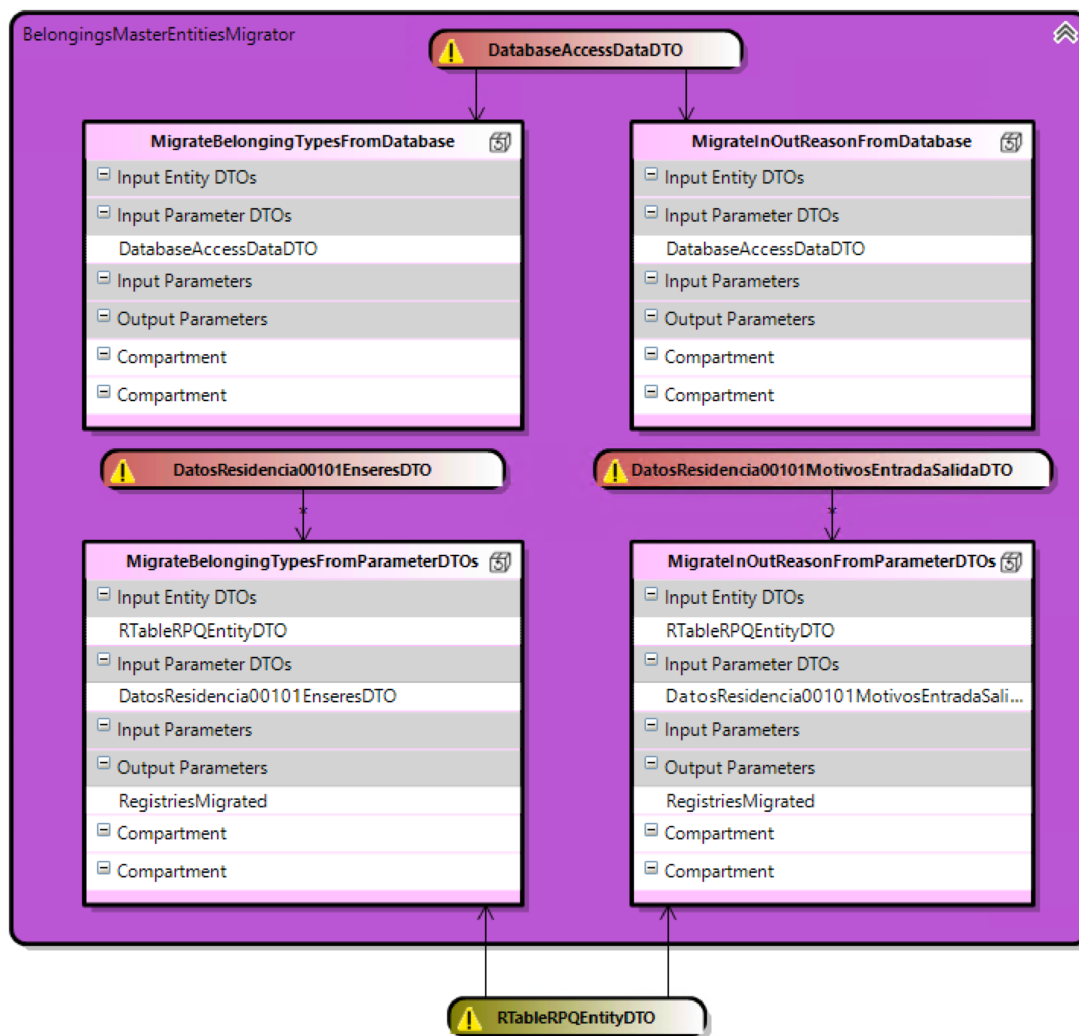

Figura 4.9: ejemplo de modelo de métodos de migración

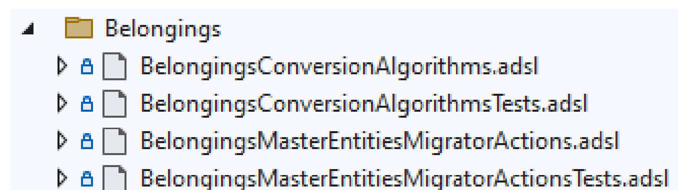

Figura 4.10: modelos creados para la migración de cada entidad

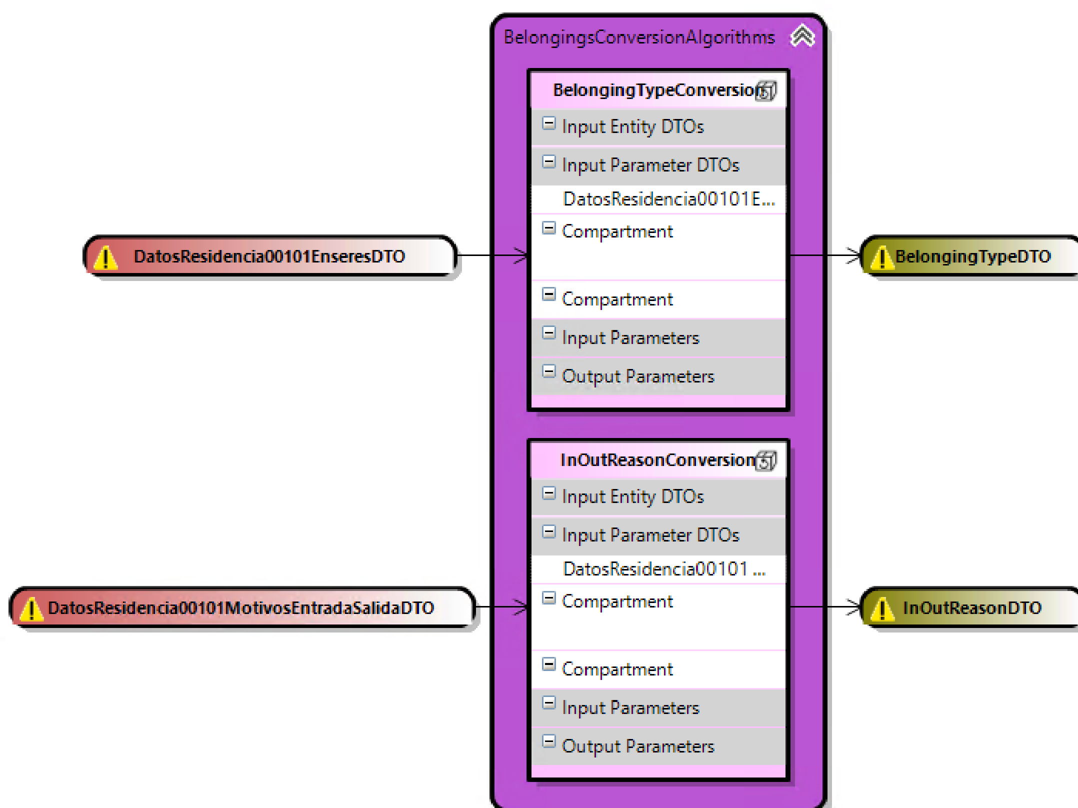

Figura 4.11: ejemplo de modelo de métodos de conversión

4.4 Programación

En este apartado se explican los hitos logrados en el desarrollo del microservicio de migración. Estos van a ayudar a explicar las diferentes partes de las que se compone la solución creada. Por otro lado, también se detallan algunas particularidades que tiene este algoritmo y que ha sido necesario tratar de forma específica.

4.4.1. Algoritmo principal

Para empezar comentaremos el proceso de desarrollo principal y cómo ha ido mejorando hasta el punto actual. Este proceso comenzó con una estructura ya creada, pero vacía de un microservicio, la cual aún no tenía ningún tipo de lógica implementada, a excepción de la común con el resto de microservicios. Esta lógica sería la relacionada con las distintas capas vistas en el apartado 4.3.1 y la que permitiría a este microservicio ser desplegado y comunicarse con otros microservicios.

Después de realizar una revisión de los casos de uso a llevar a cabo y requisitos a cumplir, se decidió enfocar el inicio del desarrollo en los requisitos CU03 y CU05, planteando el objetivo inicial de crear un modo de persistir la información sobre los datos que hubiesen sido, o no, migrados con éxito, y los errores que hubiesen impedido la migración de estos últimos. Además, este proceso debía de proveer una forma de poder ejecutar la migración de los datos a través de peticiones que se realizasen a este microservicio.

Creación del algoritmo principal y primeras entidades

Para llevar a cabo estas tareas, se empezó con la creación de los primeros elementos en los modelos de dominio y aplicación. Por un lado, para permitir esta persistencia de datos, en el modelo de dominio se crearon las tres entidades existentes actualmente, mostradas anteriormente en la figura 4.4. Estas fueron sufriendo ligeras modificaciones durante el proceso de programación, no obstante, el concepto general de la estructura se ha mantenido prácticamente durante todo el desarrollo. Por otro lado, se crearon los primeros elementos en varios modelos de aplicación que se encargarían de iniciar el proceso de migración. Para ello, se creó la única acción pública del microservicio mostrada en la figura 4.6, y una acción privada que se encargaría de realizar la migración de una de las entidades existentes. Esta entidad se eligió debido a su simplicidad en cuanto a la conversión de datos se refiere para evitar complicar estas primeras tareas.

Durante esta primera fase, uno de los objetivos principales que se propuso resolver, para cumplir con los Casos de Uso nombrados, fue realizar la migración de forma completa sobre una entidad elegida, es decir, la finalidad de esta fase sería implementar la lógica que permitiese leer los datos, cambiar su formato y enviarlos al microservicio de destino registrando alguna incidencia básica. Este proceso es el que se muestra en la figura 4.12.

Una de las tareas a realizar para llevar a cabo este objetivo fue implementar la lectura de las filas de una tabla de una de las bases de datos, donde estuviesen ubicados los registros a migrar. Esto no resultó una tarea compleja, puesto que, conociendo el nombre de la base de datos y el nombre o dirección del servidor donde está ubicada, es relativamente sencillo obtener todos los datos de una tabla en concreto, y estos valores son recibidos como parámetros al inicio del proceso de migración. Una dificultad que surgió durante la implementación de esta parte del proceso, fue que era necesario tener en cuenta el usuario que accedería al servidor. La solución que se aplicó fue añadir de forma temporal las credenciales de un usuario por defecto en el propio microservicio, hasta que se realicen las pruebas de despliegue.

Otra tarea consistió en crear la lógica necesaria para convertir los datos leídos de la base de datos origen, estructurarlos para facilitar su manipulación, convertirlos a su formato de destino y por último enviarlos para su creación al microservicio objetivo. Para efectuar esta tarea, se decidió que se crearía un DTO en los modelos de aplicación por cada tabla presente de la base de datos origen, donde los campos de dicho DTO representasen las columnas de la tabla, y las instancias que se creasen durante la ejecución del proceso representarían las filas o registros de la tabla. En otras palabras, por cada fila leída de una tabla se crea una instancia del DTO que la representa, permitiendo manipular los datos leídos como si se tratasen de instancias de un objeto, y donde sus atributos contienen los valores de cada celda. Por el momento, para esta primera instancia se determinó crear este DTO a mano en los modelos de aplicación, pero debido al gran número de tablas que componen la base de datos, se tomó la decisión de que sería necesario poder generar estas representaciones de forma automática.

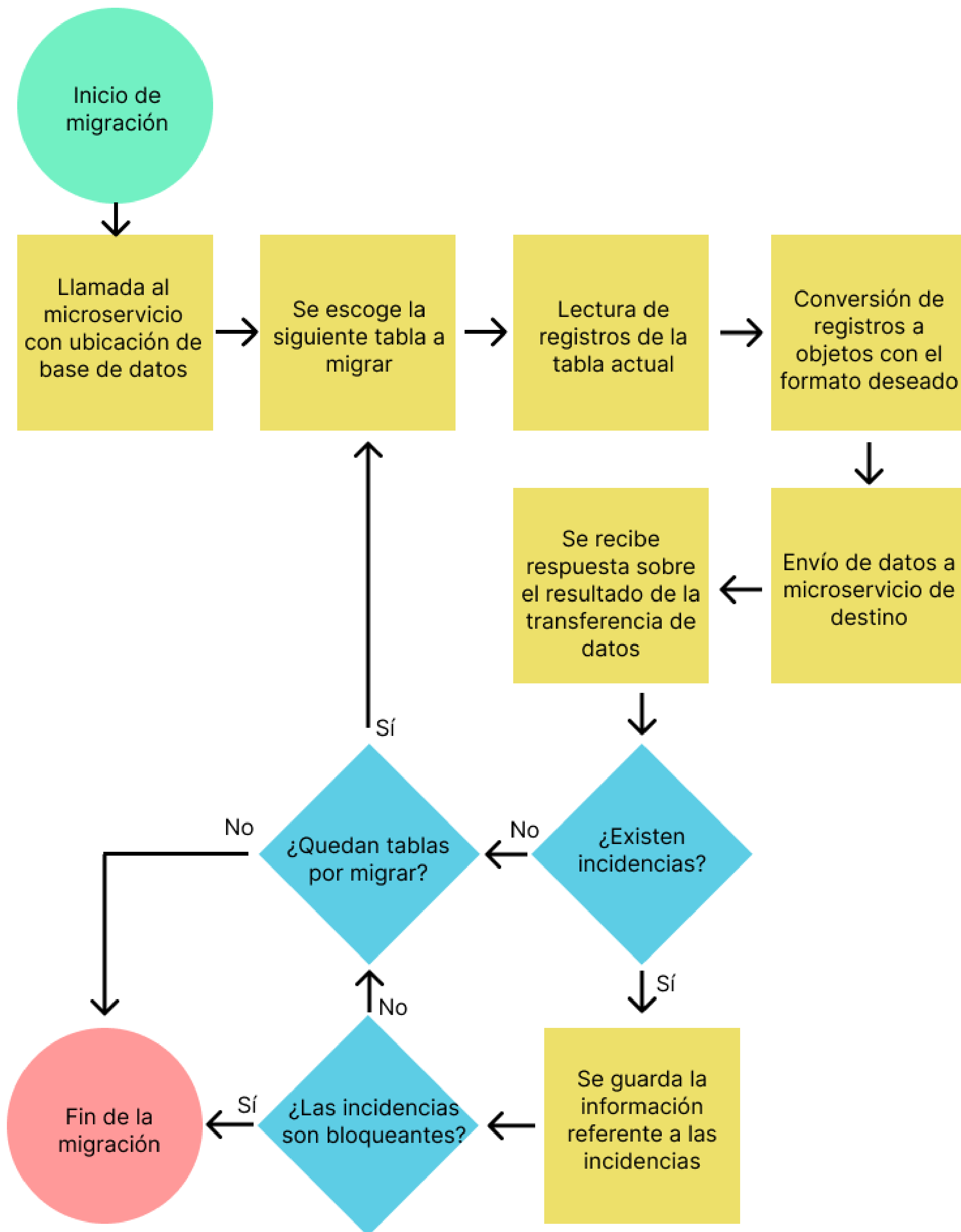

Figura 4.12: proceso de migración

Para ilustrar este ejemplo en la figura 4.13 se muestra un ejemplo de un DTO modelado, que representa la tabla Enseres y sus columnas, que se enseña en la figura 4.14. Además, se puede observar que los campos que componen dicho DTO representan las columnas de la tabla.

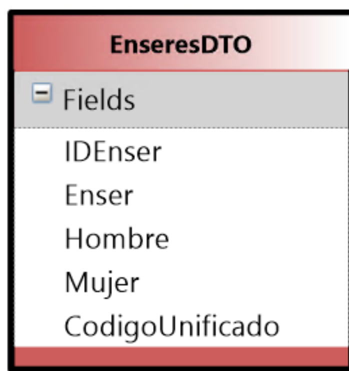

Figura 4.13: DTO modelado que representa la tabla Enseres y sus columnas

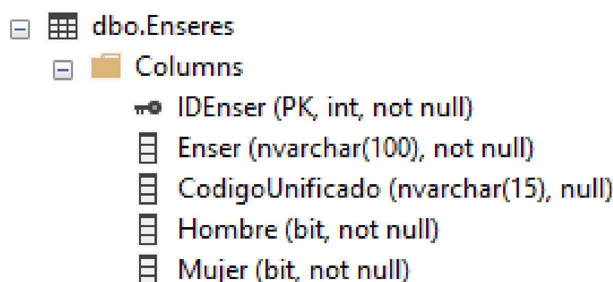

Figura 4.14: esquema de la tabla Enseres

Una vez elaborada la representación de la entidad como un DTO e implementada la lógica que permite leer los registros de la tabla y convertirlos a instancias de este objeto, se creó la parte del código encargada de convertir los datos de las instancias del DTO actual, que representan las filas de las tablas de base de datos, a las instancias de otro DTO que representase la entidad objetivo. Esto es necesario, debido a que para enviar los datos al microservicio de destino es necesario que los datos tengan dicha estructura. Asimismo, se elaboró la parte encargada de enviar dichos datos al microservicio de destino, para que este se encargue de crear los datos en su propia base de datos y devolver las incidencias que hayan surgido durante dicho proceso.

Para dar por terminado el proceso, se programó un procedimiento básico, por el cual registrar las posibles incidencias causadas por errores en el microservicio de destino, durante el transcurso del proceso de verificación y creación las instancias recibidas en este. Para permitir esta función se requirió crear un método de procesamiento de la respuesta que devuelve el microservicio de destino, la cual retorna un objeto del que se pueden obtener los errores de validación encontrados durante la gestión de los datos. Si ningún error es devuelto por este objeto, se procede a guardar la información referida a la correcta migración de los registros en el histórico de migración, en caso de que se devuelva algún error en dicha respuesta se pasa a realizar un tratamiento de dicho objeto. La finalidad de este procedimiento es obtener cuáles de los datos han provocado el error y guardar dicho mensaje de error de forma individual para cada registro, junto con el estado de migración que indica que ha surgido un error.

Durante esta primera fase, se decidió crear adicionalmente una segunda migración de otra de las entidades, elaborando para ello una segunda acción privada de la que también se implementó su lógica de migración y conversión de datos correspondiente. El propósito de crear la migración y conversión de datos para dos entidades diferentes desde el principio, era facilitar la búsqueda de partes comunes entre la migración de las

distintas entidades, y ver que fragmentos de código tendrían que programarse de forma específica para cada entidad a migrar, y cuáles se podrían generalizar. Como indica el CU04, existen muchos cambios con respecto al producto actual y se requiere realizar el mapeo y conversión de los datos de cada tabla de forma particular.

Las conclusiones que se obtuvieron de este proceso fueron las siguientes:

- Existe una gran parte del código que se puede modificar para utilizarse de forma común en la migración de la mayor parte de entidades.
- Debido a la gran cantidad de tipos de datos cuya migración debe programarse de forma concreta, es necesario encontrar alguna forma de que la cantidad de código específico a implementar por cada tipo de datos sea la menor posible.
- Para facilitar la creación de los modelos y evitar posibles errores en su elaboración, es necesario la creación de una herramienta que permita la creación de los DTO que representan las tablas de base de datos del producto actual.
- Tiene que tenerse en cuenta y seguir la estructura generada a partir de los modelos para facilitar la creación de futuras pruebas, por lo que el algoritmo genérico debe poder ser ejecutado desde el código generado a partir de los modelos.

Algoritmo genérico y división en tipos de entidades

Después de implementar el algoritmo necesario para la migración de dos entidades, se procedió a analizar las partes repetidas de ambos procedimiento para obtener el código que pudiese ser utilizado por todas las distintas migraciones de forma genérica, y que el código específico de cada una de estas migraciones fuese el menor posible para agilizar la confección de la herramienta. Asimismo, se buscó la forma de dividir este algoritmo de migración en varias fases, centrando cada una de estas en migrar distintas agrupaciones de tipos de entidades, estos tipos serán explicadas en el apartado 4.4.3. A modo de introducción, estos tipos de migración se muestran en la figura 4.15, donde cada cuadrado representa una clase que añaden o implementa cierta lógica sobre la clase principal con el código general.

Para llevar a cabo esta tarea de crear un único código que sirviese para varios tipos de migración, se utilizó un patrón de diseño de software llamado patrón plantilla [26]. Este patrón permite resolver el problema de la duplicidad de código cuando una o varias clases contiene código repetido o muy similar. La solución que plantea este patrón es dividir el algoritmo o lógica a realizar en varias partes distintas, crear los métodos necesarios para su ejecución, teniendo una implementación por defecto o una abstracta, y delegar a varias subclases la implementación de estos métodos.

Aplicando este patrón al algoritmo de migración, usando como base el código similar entre las dos migraciones ya creadas, se ha elaborado una clase abstracta que contiene la mayor parte de la lógica a utilizar y donde se sitúa el proceso general de migración y de registro de incidencias. También se han creado otras subclases abstractas, que se encargan de implementar o añadir parte de la lógica común de algunas migraciones, pero que no se pueden poner en la clase abstracta principal, debido a que solo es compartida, por una parte, de las entidades. A su vez, la lógica, que, por otro lado, no se podía modificar y hacerla de uso general, se ha desplazado a unas subclases por debajo de estas últimas que implementan una de las subclases abstractas. Cada una de ellas se encarga de sobrescribir aquellos métodos necesarios para la lectura y conversión específica de cada migración de entidad.

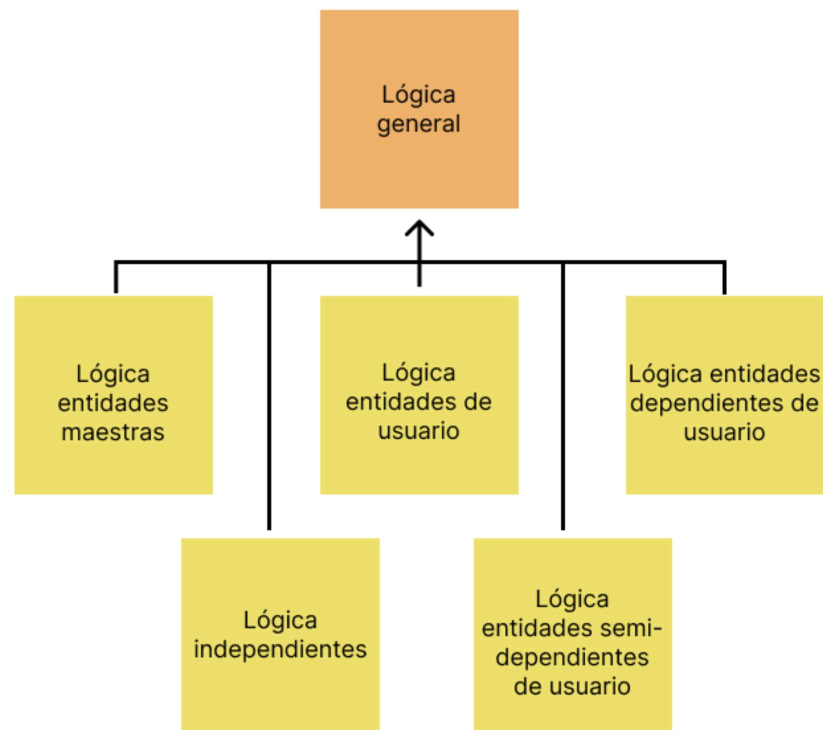

Figura 4.15: esquema de la estructura de la lógica para los tipos de migración

En la figura 4.16 se muestra el esquema que estas clases siguen para implementar el patrón plantilla y utilizar el menor código posible. La clase abstracta principal es la que orquesta los pasos principales del algoritmo, como la lectura de datos y la gestión de incidencias. Las clases secundarias agrupan código similar entre diferentes migraciones con un proceso parecido, como puede ser el guardado del estado de la migración de cada dato. Por último, en las subclases con código específico se implementa la lógica particular de ciertos métodos, por ejemplo, la conversión de los datos y creación de DTO con el formato de destino.

Durante la creación de las clases secundarias, se empezaron también a analizar e implementar los primeros patrones de migración, que serán explicados en el apartado 4.4.2. A modo de introducción, los patrones de migración es la forma en la cual se ha decidido nombrar a aquellos casos, que debido a la gran similitud entre distintos tipos de datos, son gestionados de una forma muy parecida o idéntica en lo referente a la creación y gestión de incidencias. Estos se han decidido tratar de forma especial, con la intención de crear un procedimiento general, y agilizar el desarrollo del proyecto mediante la creación de un código común que gestione dichos casos.

Paralelamente a la mejora del algoritmo, se elaboró la herramienta anteriormente mencionada, que pese a que no se va a entrar en detalles sobre su funcionamiento, es la que ha permitido crear de forma automática los DTO que representan el esquema de una base de datos SQL en los modelos de aplicación. Cada uno de estos DTO representa una tabla de dicha base de datos, cuyas columnas adoptan la forma de campos de dichos DTO como se mostraba en las figuras 4.13 y 4.14.

Finalmente, también se añadieron nuevas migraciones de entidades, aplicando la estructura de código diseñada, y se crearon las primeras pruebas con respecto a la ejecución del microservicio. Estas pruebas se ejecutaron utilizando datos reales de un centro desde una base de datos local, y permitieron probar el correcto funcionamiento del proceso de migración. Estas pruebas serán explicadas en el apartado 4.5.1

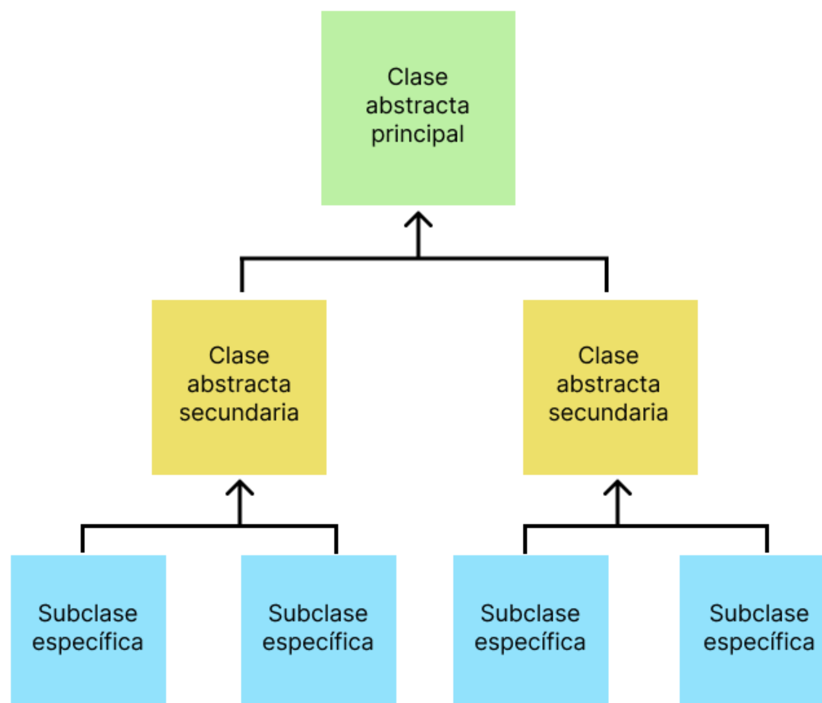

Figura 4.16: estructura de del código mediante clases abstractas y sus implementaciones

Migración por estados y división de tareas

Una vez explicado el proceso de migración y su estructura de clases interna, es necesario concretar cuáles son los distintos estados de migración y como estos afectan a los datos. Según el estado de migración de cada dato, el proceso de migración debe tratarlo de una forma u otra, ya sea para actualizar los datos migrados o gestionar incidencias existentes. En total existen cinco tipos de estados de migración y estos se pueden visualizar en el enumerado 'MigrationRecordStatus' de la figura 4.4.

- **NotMigrated.** Este estado de migración es el que tienen los datos cuya migración aún no se ha ejecutado por primera vez.
- **SuccessfullyMigrated.** Este estado representa los datos que han sido migrados satisfactoriamente durante alguno de los procesos de migración. Si los datos transferidos correctamente fuesen modificados por los usuarios, estos pasarían a ser del tipo MigratedButModified, ya que necesitarían actualizar su valor modificado.
- **NeedsToBeChecked.** Este estado representa a aquellos datos que durante su migración han provocado algún tipo de incidencia que puede ser resuelta por los usuarios de la aplicación.
- **CanNotBeMigrated.** Al igual que el estado anterior, este representa datos que han intentado ser migrados, pero que han generado una incidencia. Estos se diferencian de los anteriores en que su incidencia no puede ser resuelta por los usuarios y debe ser gestionada desde la propia aplicación o empresa.
- **MigratedButModified.** Como se ha comentado en el segundo estado, este representa registros que se han podido migrar correctamente, pero que han sido modificados y se han de migrar de nuevo, con el objetivo de actualizar su valor en el microservicio de destino.

En la siguiente figura 4.17, se indica las transformaciones que se pueden dar en el estado de migración de los datos.

	Migración satisfactoria	Migración con errores
NotMigrated	SuccessfullyMigrated	NeedsToBeChecked o CanNotBeMigrated
SuccessfullyMigrated	MigratedButModified (si el dato es modificado)	
NeedsToBeChecked	SuccessfullyMigrated (si la incidencia es resuelta)	
CanNotBeMigrated	SuccessfullyMigrated	NeedsToBeChecked o CanNotBeMigrated
MigratedButModified	SuccessfullyMigrated	NeedsToBeChecked o CanNotBeMigrated

Figura 4.17: transformaciones entre estados de migración

De todos estos estados de migraciones, el proceso de migración solo debe encargarse de gestionar tres de ellos, además de administrar un proceso de reintento y actualización de aquellos datos que hayan generado una incidencia o hayan sido modificados. Asimismo, este algoritmo es también capaz de detectar aquellos registros que ya hayan sido migrados, de los que no, cuando han generado una incidencia. A continuación, se presentan los tres tipos de procesos de migración de datos que el microservicio es capaz de llevar a cabo.

- **Datos no migrados.** Este proceso busca ejecutar la primera migración de los datos del producto actual para su creación en la base de datos de destino. Para este tipo de datos todavía no existe ninguna referencia ni en el registro de migraciones ni en el microservicio objetivo. Por esta razón, además de llevar a cabo la transferencia del dato, se debe crear una representación de su estado en el histórico de migración.
- **Datos con errores durante la migración.** En su último proceso de migración, estos datos han sufrido errores a lo largo dicho procedimiento que ha impedido su creación o actualización, dependiendo esto último de si ya existían los datos en el microservicio de destino o no. Para estos datos ya existe una referencia en el registro de migraciones, por lo que cuando se reintente su migración se deberá actualizar su estado en el histórico de migración, además de crear una incidencia en caso de nuevo error.
- **Datos migrados pero modificados.** Como se ha comentado anteriormente, durante el proceso de migración de datos, el producto actual se podrá utilizar de forma habitual por parte de los clientes, es por esta razón que algunos de los datos podrían ser modificados antes de acabar la migración de todos ellos y empezar a utilizar el nuevo producto. Por ello, los datos modificados deberán ser migrados otra vez actualizando los valores. Para estos datos existirá una referencia tanto en el microservicio de destino como en el registro de migraciones.

Además de estos tres procesos, hay que tener en cuenta que al igual que los datos pueden ser modificados por los usuarios, provocando que tengan que actualizarse en la

ejecución del siguiente proceso de migración, también se puede dar el caso de que estos datos sean borrados. Por tanto, ha sido necesario implementar un proceso anterior a estos tres, en el cual, si se detecta que un dato ha sido borrado, se procederá a eliminarlo tanto de la base de datos del microservicio de destino, como su referencia en el registro de migraciones.

Una vez ya implementados los diferentes casos que se pueden dar sobre el estado de migración de los datos, está finalmente cubierto todo el ciclo de vida de la migración de los datos, pudiendo ejecutarse en bucle hasta que, tras la resolución de las distintas incidencias, se hayan migrado de forma exitosa todos los datos.

Para poder trabajar de forma conjunta y facilitar la creación de *tests* de los métodos de conversión, los cuales se encargan de asignar y relacionar los campos de un registro de una tabla del producto actual con los campos de una entidad del nuevo producto, se decidió dividir la parte del algoritmo específico encargada de la lectura y gestión de incidencias de la conversión de los datos. Para separar estas partes se decidió que la lógica de conversión sería modelada de forma específica en la DSL *Tool*, como se ha enseñado anteriormente en la figura 4.11. Esto se hizo con el propósito de permitir a las personas encargadas del análisis del mapeo de los datos, pudiesen crear directamente este mapeo en los propios modelos, a la vez que se sigue desarrollando el algoritmo de migración. Además, esto permite la creación de métodos de prueba de la conversión directamente en los modelos, sin necesidad de crear la implementación de dichos métodos de prueba mediante código.

Mejoras realizadas y estado actual

A continuación se van a comentar algunas de las mejoras llevadas a cabo sobre el algoritmo de migración, ya que este se ha ido mejorando durante todo el proceso, y existen algunas partes que merecen la pena destacar.

Una de las mejoras realizadas ha sido implementar un segundo intento de reenvío de datos. Dicho de otra forma, esta modificación del proceso, permite que cuando se intenten crear los datos en el microservicio de destino y por algún problema con una validación, no se puedan crear correctamente en la base de datos objetivo, esta migración se volverá a intentar con los datos que no hayan generado ningún fallo. Esto se hace al filtrar los registros enviados junto a los errores de validaciones recibidos para guardas las incidencias necesarias y volviendo a realizar la transferencia de datos por segunda vez. De esta forma se permite migrar la mayor cantidad de registros por intento de migración.

Otra mejora que se ha llevado a cabo con respecto al algoritmo, es la gestión de inconsistencias entre los datos existentes en el microservicio de destino y sus referencias en el registro de migraciones. Con esto se planea evitar posibles fallos en el microservicio que tengan relación con errores que puedan surgir en la máquina donde se esté ejecutando. Uno de los casos que se quiere evitar sería que se envíen los datos al microservicio destino y se creen correctamente en este, pero que esta acción no sea registrada en el histórico de migración, lo que podría dar lugar a duplicidades de datos.

Una vez creado todo este proceso de lectura, conversión y envío de datos, junto con su propia gestión de incidencias, y diseñada la forma en la que se implementarán el resto de pruebas, únicamente queda pendiente proseguir con el proceso de creación de nuevas migraciones de datos, el análisis e implementación de nuevos patrones y la producción de métodos de pruebas de las migraciones. Este es el punto en el que actualmente se encuentra el proceso de desarrollo del microservicio de migración de datos.

4.4.2. Patrones de migración

Durante el desarrollo del proyecto y la creación de las distintas migraciones específicas para cada entidad, han surgido lo que se ha llamado internamente en la empresa, patrones de migración, estos son, tratamientos similares que deben realizarse durante la conversión de los datos o la gestión de las posibles incidencias que pueden surgir debido a las validaciones aplicadas en el nuevo producto. Estos patrones se han decidido tratar de forma especial con el objetivo de agilizar el proceso de desarrollo, debido al gran número de tablas sobre las que se ha de realizar la migración de sus datos. Por otro lado, no hay que confundir estos patrones con el algoritmo genérico creado utilizando como base el patrón plantilla, pues este algoritmo es utilizado por todos las migraciones, mientras que estos patrones son aplicados solo a aquellas migraciones que lo requieren.

En el transcurso de la elaboración del proyecto han surgido múltiples patrones de migración, desde valores en ciertos campos que deben provocar la creación de una incidencia, incluso antes de ser enviados al microservicio de destino, a la gestión de los datos de archivos temporales del producto actual. A continuación se explicarán algunos de los patrones que más relevancia han tenido, ya sea por un gran número de casos o por la importancia en cuanto a procesamiento de estos.

Campos con valores calculados

Empezando por campos con valores calculados, estos se tratan de campos cuyo valor no puede ser asignado directamente, y es a partir, de la ejecución de una acción tras la asignación de los otros campos que forman parte de la instancia de la entidad, que se calcula dicho valor y posteriormente es asignado en el propio DTO. Por ejemplo, para el campo de un registro que indique la edad del usuario, su valor no puede asignarse directamente, sino que este es calculado de forma automática a partir del campo que indique su fecha de nacimiento. Esto a nivel técnico no supone mayor problema, ya que se puede comprobar el valor calculado, nada más crear el DTO y asignar el campo que actúa como base para el cálculo, pero es necesario avisar de la posible discrepancia al usuario de la aplicación.

El objetivo de este patrón es evitar cualquier posible discrepancia durante la migración de los datos y que no dependa directamente del proceso de mapeo y conversión de estos, sino de cálculos ajenos al microservicio de migración. Para llevarlo a cabo, se decidió que la incidencia creada por este patrón no sería bloqueante, es decir, esta incidencia no se tiene en cuenta como un error y permite que la migración se desarrolle de forma normal.

A fin de gestionar esta incidencia, la idea que plantea el patrón es migrar el registro de la forma habitual, generando una incidencia en el histórico, que posteriormente se mostrará al usuario indicando que ese dato de la entidad en cuestión ha sufrido una modificación y que debe ser revisado en caso de que el valor modificado no sea correcto, y dicho usuario podrá marcar la incidencia como revisada cambiado el estado de migración de dicho registro para indicar que se ha migrado correctamente.

Valores no válidos

Durante el análisis de ciertas entidades se ha encontrado que, ya sea por alguna validación o restricción, existen ciertos valores para algunos campos que generarán un error al intentar crearse en la base de datos de destino, o directamente las validaciones anteriores a dicho proceso impedirán que se intenten crear, pues su valor no sería correcto. En este patrón se dan principalmente dos tipos de casos.

Por un lado, existen las restricciones propias de una base de datos, el tipo de incidencias que pueden surgir a partir de estas restricciones está relacionada, con la estructura o diseño que se le da a los datos, por ejemplo, un error con estas restricciones que provocaría la posterior creación de una incidencia sería la migración y posterior inserción en base de datos de un registro donde uno de sus campos de texto tuviese una longitud mayor a la permitida en la columna de la tabla objetivo.

Por otro lado, se encuentran las incidencias generadas, por errores detectados a partir de las validaciones aplicadas antes y después de la creación, modificación o borrado de datos en la nueva aplicación. Estas validaciones comprueban que los datos creados sean lógicos de acuerdo al producto y a la realidad, un ejemplo sobre este tipo de validación sería el dato relacionado con la toma de temperatura de un residente, donde existe un rango de valores aplicados sobre el campo que indica el valor de dicha temperatura, impidiendo que se le asigne valores irreales como una temperatura inferior a 30 °C, o superior a 50 °C. Cabe mencionar que este tipo de validaciones no existen en el producto actual, donde se deja al usuario asignar este tipo de valores sin apenas restricciones.

En ambos casos, al no poderse migrar el registro, causando, por tanto, que este no se cree en el microservicio de destino, provoca que la migración deba cancelarse antes de continuar con otros datos que puedan depender de este primer registro, en este caso se procederá a aplicar el RF02 y el RF03, donde se señala que la migración debe realizarse por bloques permitiendo al usuario resolver las incidencias actuales, y ejecutar de forma continua el proceso de migración en caso de que la incidencia se solucione permitiendo migrar los registros que anteriormente provocaron la creación de una incidencia.

Entidades maestras con dos niveles de datos

Mediante el uso de este patrón se quiere afrontar la migración de los registros de datos maestros que existen en el producto actual, estos datos son, tanto los que ya existen cargados de forma previa en la aplicación, como los creados o modificados por parte de los clientes.

Sobre los datos maestros, estos son un tipo de datos existentes en el producto que por lo general funcionan como opciones a elegir para los valores de ciertos registros. Un ejemplo de entidad o tabla de datos maestros serían las alergias de los residentes del centro, donde para cada usuario se pueden seleccionar las alergias que este tiene de las distintas opciones disponibles, para el producto actual estas pueden haber sido añadidas durante la instalación del producto, o haber sido creadas o modificadas por parte de los usuarios de la aplicación.

El problema actual que plantea este diseño de datos maestros es que aunque existan varios datos base, estos pueden haber sido modificados o ampliados por los usuarios, impidiendo realizar estadísticas, tareas de análisis o utilizar *Business Intelligence* (BI), por parte de la empresa para plantear nuevas funcionalidades o mejorar al producto existente.

La solución que se propone es la creación de una estructura con dos niveles de datos de las entidades maestras del nuevo producto. Los datos en el primer nivel o de primer nivel, servirán para utilizar BI y sacar estos análisis o estadísticas, mientras que los datos de segundo nivel serán tanto los que se creen en el nuevo producto como los que se migren desde el producto actual.

Algunas de las mejoras que otorgará la estandarización de los datos maestros son:

- Reducir errores en la introducción o creación de estos datos a mano, eliminando posibles duplicidades de estos datos.

- Utilizar un estándar común dentro de los distintos centros, permitiendo que no haya apenas diferencias entre los centros de un mismo cliente.
- Posibilidad de comparar datos entre distintos centros de países o idiomas distintos que utilicen datos maestros de segundo nivel diferentes.
- Mejorar funciones de la aplicación para que reaccione mejor a los datos introducidos, por ejemplo, se podrá programar un procedimiento habitual cuando a un residente se le registre una alergia.

Para poder utilizar los datos creados o migrados de segundo nivel dentro del nuevo producto se obligará a que estos tengan que estar relacionados con un dato de primer nivel, esta asignación se realizará de forma obligatoria para los datos de segundo nivel que se creen dentro de la aplicación y para los datos migrados esta relación estará vacía desde un principio, pero esto tendrá un carácter temporal, pues desde un punto de vista conceptual no tiene sentido que se utilicen datos de segundo nivel que no estén asociados con uno de primer nivel. Para evitar saturar a los usuarios con la resolución de las incidencias de asociar los datos migrados con los datos existentes, antes de poder utilizar el producto, esta asignación se realizará en el momento en el que se vaya a utilizar uno de los datos de segundo nivel, obligando al usuario a que realice la asignación antes de poder utilizar el dato, pero permitiendo que esta tarea se realice de forma más natural y resulte menos tediosa.

Además de los datos de segundo nivel creados por los usuarios o migrados desde el otro producto, existirán datos de segundo nivel cargados de forma previa dentro de la aplicación, pues en ocasiones existirán sinónimos de uso frecuente de datos estandarizados y que así aparezcan de forma común en todos los centros. Estos datos, por supuesto, deberán estar asociados desde un principio con un dato de primer nivel.

Para implementar este patrón se han realizado modificaciones en las herramientas de modelado de la empresa, estas se basan en transformar las entidades maestras existentes de forma automática, para añadir los elementos que les permitan tener los dos niveles de datos. Estas modificaciones han sido las siguientes:

- Añadir una nueva propiedad a las entidades maestras llamada 'Is Standardized Master Entity', esta propiedad será de tipo booleano e indicará si la entidad maestra permite dos niveles de datos o no. Por defecto, esta propiedad tendrá el valor falso, pero al cambiar a verdadero se añaden de forma automática los campos y asociaciones necesarios.
- Creación de un campo llamado 'TypeOfData' que indicara si el dato maestro se trata de un dato de primer nivel o de segundo nivel. A modo de permitir futuras validaciones, se han creado tres tipos de valores para los datos de segundo nivel que indican si se trata de un dato de segundo nivel cargado durante la instalación del producto, un dato creado por un usuario desde la aplicación o de un dato migrado desde el producto actual.
- Creación automática de una relación reflexiva que permita enlazar los datos de segundo nivel con uno de primer nivel, además de la creación de validaciones que impidan que por ejemplo un dato de primer nivel esté asociado con cualquier otro.

En las figuras 4.18 y 4.19 se muestra un ejemplo de una entidad maestra, donde en la primera figura aparece esta entidad sin modificar con un solo nivel de datos, y en la segunda figura es el resultado después de aplicar las modificaciones para permitir el doble nivel de datos, en la que se ha añadido el campo 'TypeOfData' y la asociación reflexiva llamada 'AssociationToFirstLevelData'.

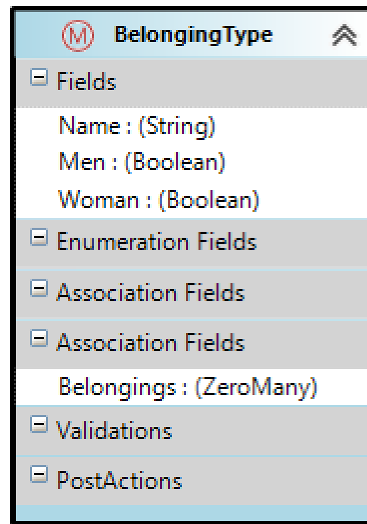

Figura 4.18: entidad maestra con un solo nivel de datos

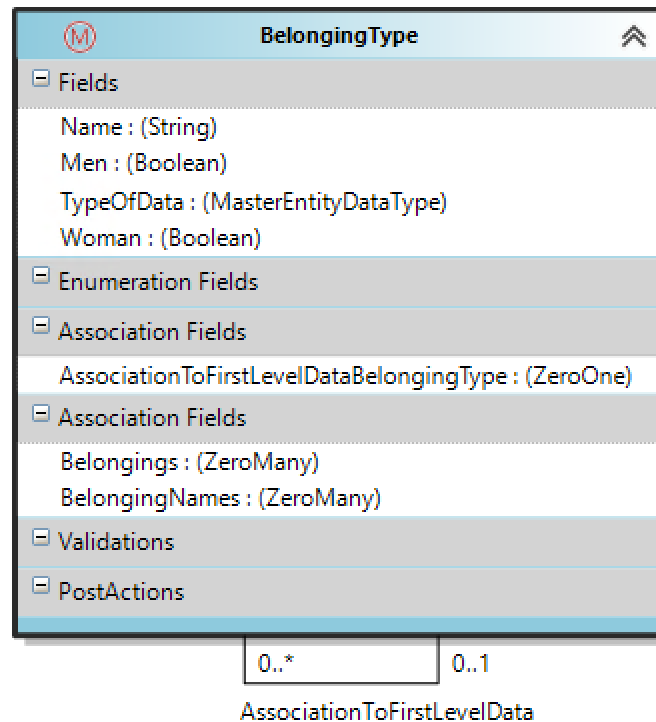

Figura 4.19: entidad maestra modificada con dos niveles de datos

4.4.3. Tipos de entidades

Para realizar el proceso de migración este debe realizarse en un orden lógico, realizando primero la migración de los datos que no tengan ningún tipo de dependencia, es decir, que la relación con otras entidades tenga como valor mínimo cero, para luego continuar con los que dependan de estas primeras entidades, con el objetivo de evitar dejar relaciones vacías entre los datos. Por tanto, se ha decidido dividir la migración en cinco fases en las que en cada una de ellas se migrará un conjunto de entidades divididas en varios grupos.

Entidades maestras

El primer grupo se trata de las entidades maestras comentadas en el apartado 4.4.2, estas cuentan con la peculiaridad de que ninguna de ellas depende de otra entidad, sino que son el resto de entidades las que dependen en su mayoría de este tipo de entidades.

Como se ha comentado, los datos contenidos en estas entidades maestras se tratan de datos estandarizados cargados de forma previa en la aplicación, por ello para su migración se deberá comprobar si estos datos existen previamente en la aplicación antes de proceder a la migración para evitar posibles duplicidades.

Entidades libres

El segundo grupo de entidades a migrar se tratan de las entidades libres, este tipo de entidades dependen únicamente de entidades maestras o de otras entidades libres. Dentro de este grupo no se contemplan las entidades que dependan de la entidad User, ya que esta requerirá un tipo especial de migración.

Las entidades libres se podrán migrar solo una vez se haya terminado de migrar las entidades maestras, pues si la migración de estas últimas no se han terminado, las entidades libres que dependan de ellas no podrán ser migradas por completo al no poder relacionar el registro creado en el microservicio de destino con el dato maestro correspondiente, dejando este dato incompleto o con un error de validación que impedirá su creación.

Entidades de Usuario

Después de la migración de los dos grupos anteriores, se procederá a migrar los datos de los residentes de los centros de los clientes y los usuarios de la aplicación. Dentro de este grupo se encuentra la entidad User y todas aquellas entidades que esta requiera para poder ser migrada y que no se hayan creado todavía tras ejecutar el proceso de los dos grupos anteriores.

La entidad User requiere un tratamiento especial, pues esta utiliza datos de evolución temporal, los cuales forman una parte esencial de las tareas que se podrán realizar en el producto. Asimismo, este se trata de un grupo independiente, ya que existen muchas otras entidades que están relacionadas con los datos de los usuarios y se requiere que todos se hayan migrado correctamente antes de pasar a los dos últimos grupos.

Entidades semi-dependientes de User

Son entidades que dependen de User, pero no de una manera directa, en otras palabras, para la existencia de estos datos no es necesario que estén relacionados con un dato de usuario, ya que están asociados con una multiplicidad de cero a uno o de cero a muchos. Además, estos datos pueden depender de las entidades migradas en cualquiera de los dos primeros grupos o de este mismo.

Por otra parte, para su migración, los datos ya existentes podrían estar relacionados con un dato de usuario, por lo que para evitar posibles errores y que estos datos se creen de forma parcial, solo serán migrados una vez se hayan migrado los datos que compondrán la entidad User.

Entidades dependientes de User

Este será el último grupo a migrar, se tratan de entidades que dependen de estar relacionadas con un usuario para que sus datos puedan existir y tengan un sentido lógico, este sería el caso de por ejemplo, mediciones de temperatura o de peso. Al igual que el grupo anterior, las entidades de este grupo pueden tener dependencias con cualquiera de las anteriormente migradas.

Una vez ejecutado el proceso de migración de todos los grupos, el proceso de migración se mantendrá en ejecución para actualizar los datos que puedan ser modificados durante el uso del producto actual y migrar aquellos datos que se hayan creado nuevos durante o después del proceso de migración.

4.5 Pruebas

Para comprobar que el proceso de migración funcione de forma correcta, se decidió realizar tres tipos de pruebas, dos de ellas se tratan de pruebas automatizadas, que verifican a partir de datos artificiales y creados dentro de los métodos de prueba, que el código implementado realice las migraciones y conversiones individuales de forma precisa, por otro lado, el último tipo de prueba se trata de pruebas manuales realizadas mediante la ejecución del proceso de migración sobre datos reales.

Cabe destacar que la ejecución de dichas pruebas se ha realizado cumpliendo las prácticas de la metodología ágil que se han seguido durante todo el proceso de desarrollo, y que se explicarán en más detalle en el apartado 4.6. Durante todo el desarrollo, las pruebas han sido adaptadas a las necesidades del proyecto, en el que se han diseñado dichas pruebas realizando un compromiso sobre la cantidad y la profundidad de estas, debido a la gran cantidad de distintos tipos de datos sobre los que hay que ejecutar la migración.

4.5.1. Pruebas automatizadas

Las pruebas automatizadas son pruebas creadas para automatizar el proceso manual de verificación y validación del funcionamiento correcto de las partes que componen un producto *software*. Este tipo de pruebas ayudan a agilizar el desarrollo de cualquier proyecto, esto es debido a que permite ejecutar, de forma rápida y sencilla, varios procesos que analizan el buen desempeño del código creado.

En el caso del microservicio de migración, estas pruebas se han dividido en dos grandes subgrupos, por un lado, se han creado las llamadas pruebas de conversión cuyo objetivo es comprobar que la creación de un DTO, utilizado para el envío y creación de los datos en el microservicio de destino, y la asignación de los valores a sus respectivos campos se ha programado de la forma correcta, por otro lado, se han elaborado las pruebas de migración, estás comprueban que, tanto el algoritmo genérico, como el específico para cada una de las migraciones, funcione de la forma esperada.

Pruebas de conversión

Empezando por las pruebas de conversión, estas realizan la función de pruebas unitarias [27], su objetivo es probar que la implementación del código de la conversión y asignación de valores es correcta, para ello comprueba de forma única dicho método evitando que el error pueda provenir del resto del código.

Para la creación de estas pruebas se han utilizado las herramientas de modelado anteriormente mencionadas, debido a que simplifican el proceso de creación de los métodos de prueba y permite la creación de estos sin necesidad de realizar ninguna implementación por código. Para ello, es necesario crear una caso de *test* en los modelos de aplicación por cada método de prueba que se quiera crear. Un ejemplo de modelado de dichos test se muestra en la figura 4.20.

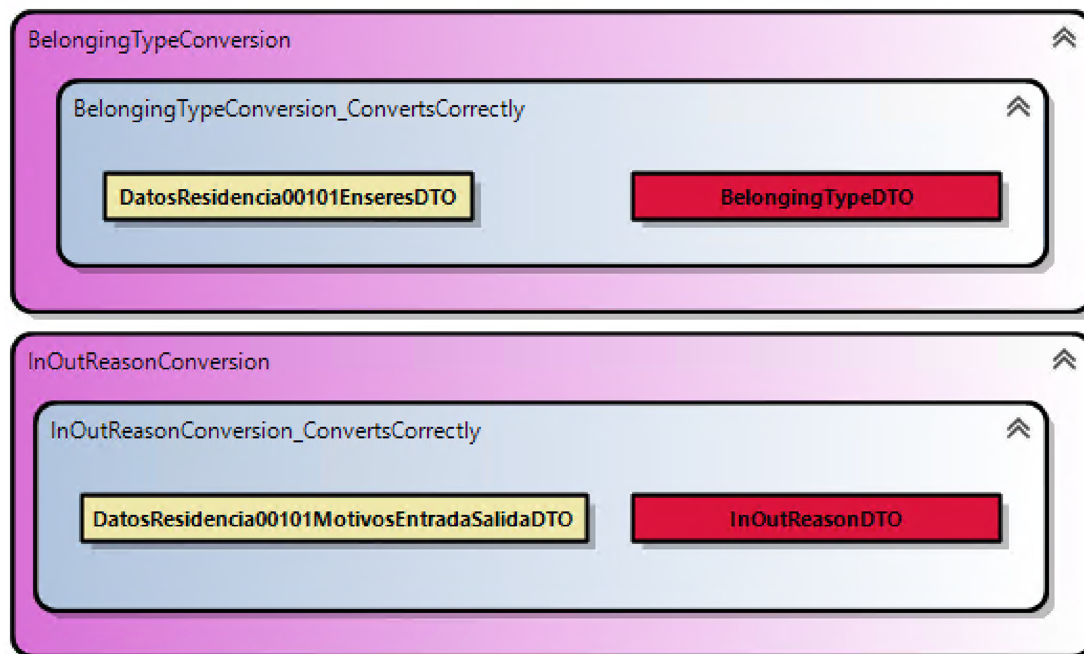

Figura 4.20: modelo de *tests* de conversión de datos

Los dos *tests* que se muestran en la figura 4.20 comprueban que la parte de la lógica encargada de la conversión de los datos de las tablas Enseres y MotivosEntradaSalida se realizan correctamente. Los métodos de conversión son utilizados para llevar a cabo el mapeo de los datos, creando un nuevo objeto que será enviado y procesado en el microservicio de destino, a partir de los datos contenidos en los objetos que representan las filas de las tablas. Al ejecutar los *tests*, estos aseguran que al ejecutar el método de conversión utilizando un objeto con ciertos datos como entrada, el método devuelve el nuevo objeto con los datos esperados.

El hecho de que estos métodos de prueba se puedan crear directamente desde los modelos y utilizando posteriormente la generación de código, ha ofrecido varias ventajas durante el desarrollo. Para empezar, brinda la posibilidad que dichas prueba sean creadas por personas que no desempeñen una labor de programación en la empresa y que en su lugar tengan más conocimientos sobre la estructura de los productos, por ello, después de la creación de las primeras pruebas, se dejó esta tarea a cargo de los analistas que al realizar el estudio sobre el mapeo de la conversión de los datos pueden elaborar estas pruebas en mayor profundidad. Esto lleva a otra ventaja, el código y las pruebas sobre este son creados por distintas personas, lo cual facilita la detección de posibles errores por ambos participantes.

Una cuestión que se tuvo que resolver con respecto a estas pruebas, fue el número que había que crear para cada una de las conversiones. Dependiendo del número de campos que tenga el registro a migrar, y a sus posibles valores, este podría llevar a desarrollar

multitud de casos de pruebas para comprobar todos los posibles caminos independientes. Por poner un ejemplo, existen entidades compuestas por varios enumerados cuya asignación debe realizarse mediante sentencias *switch*, cuyo número de condiciones es el mismo que el número de valores del enumerado, para comprobar todos los caminos de forma totalmente independiente se debería programar una prueba por cada enumerado y valor. Para resolver este problema se decidió que se probarían varios caminos de forma conjunta, en los que al menos cada uno de ello este cubierto por una prueba.

Pruebas de migración

Con respecto a las pruebas de migración se puede decir que cumplen una doble función, por un lado, actúan como pruebas unitarias si se considera toda la migración de una entidad como un solo módulo o como pruebas de integración si se considera que está compuesto por varios, estas pruebas se encargan de probar la ejecución del proceso de migración desde justo después de la lectura de los datos hasta el fin del proceso, utilizando la librería Moq comentada en el apartado 3.1, para gestionar el envío de los datos y realizarlo sobre una clase que suplante al microservicio objetivo en lugar de utilizar uno real. Por otra parte, la lectura de base de datos no es comprobada en estas pruebas para evitar tener esta dependencia sobre una base de datos, debido a que cualquier cambio sobre esta podría dejar las pruebas inservibles.

Al contrario que las pruebas de conversión, estas han de crearse por programadores, debido a que el código comprobado es el algoritmo de migración y, por tanto, el análisis de los datos y la conversión no es una parte tan fundamental en estas pruebas, al ser ya comprobado en las pruebas de conversión. Otra razón sobre esto, es que aunque la generación de código crea gran parte de este, no es suficiente para comprobar todo el funcionamiento en profundidad debido a la dependencia con el microservicio de destino, motivo por el cual es necesario complementarlo añadiendo parte del código a mano.

Estas pruebas al igual que las anteriores pueden usarse como pruebas de regresión, este tipo de pruebas son empleadas para comprobar que un cambio en el código o de funcionalidad en un programa no afecta de forma negativa al resto del funcionamiento del producto o funcionalidades. Estas pruebas ayudan a comprobar que cualquier cambio en el algoritmo genérico de migración, para implementar alguna funcionalidad nueva, cambiar su comportamiento, o modificar el funcionamiento de algún patrón, no ha introducido posibles defectos en el código.

En la figura 4.21 se muestra un ejemplo de un caso de *test* construido en los modelos de aplicación, por un lado, las cajas verdes de la izquierda representan los elementos que están presentes en la base de datos del microservicio, las cajas marrones del medio son los parámetros que recibe el método donde uno de ellos representa la entidad RTableRPQEntity vista en la figura 4.4 y el otro representa los registros que se habrían leído en la base de datos del producto actual, por último las cajas verdes de la derecha representan los datos que se espera que la base de datos del microservicio de migración tenga, después de haber ejecutado el proceso de migración de los datos leídos.

Los *tests* modelados representados en la figura 4.4, comprueban que el algoritmo que se ejecuta para la migración de los registros contenidos en una tabla del producto actual funcionaría correctamente. Estos *tests* aseguran que tras la lectura de los datos, los procesos de conversión, envío y guardado del estado de migración de los registros se realizan correctamente, comprobando además que los datos que serían enviados al microservicio de destino son los esperados. El primero de ellos realiza las pruebas para datos cuya migración se va a ejecutar por primera vez, mientras que en el segundo los datos habrían intentado ser transferidos, pero habrían generado una incidencia, y durante la ejecución

de la prueba se realiza la segunda migración de estos registros, que se intentarán migrar de nuevo, consiguiéndolo satisfactoriamente.

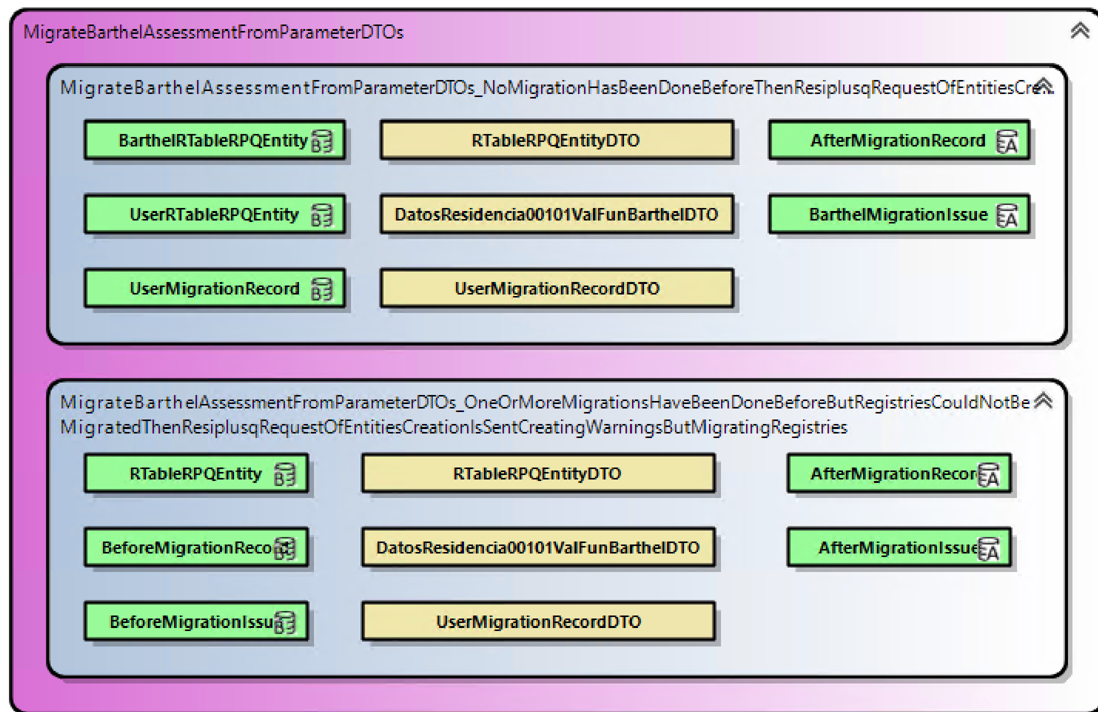

Figura 4.21: modelo con *tests* de migración

4.5.2. Pruebas sobre datos reales

El tercer tipo de pruebas han sido realizadas mediante la puesta en marcha del microservicio y la ejecución de este sobre una base de datos con datos reales. Estos datos forman parte de la base de datos de uno de los clientes, cuyos campos han sido ofuscados, sustituyendo cualquier dato de carácter personal y que pudiese identificar a cualquier persona. El objetivo de utilizar datos reales para realizar las pruebas del microservicio es examinar el comportamiento sobre registros similares a los que se encontrarán en el futuro despliegue del microservicio con el nuevo producto.

La finalidad de estas pruebas es comprobar si existen posibles fallos en el proceso de migración que no hayan sido considerados en los otros tipos de pruebas, por ejemplo, podrían existir casos donde algunos valores de los datos originen errores, ya sea en su lectura, conversión o procesamiento. Otra de las utilidades que proveen estas pruebas es la posibilidad de realizar estudios o análisis sobre los resultados del proceso, es decir, observar que incidencias son las más frecuentes, por si existiese alguna forma de reducir su número, o medir el tiempo que lleva realizar la migración de cierta cantidad de datos.

Al igual que el resto, las pruebas sobre datos reales han sido realizadas de forma iterativa, para encontrar lo antes posible los fallos que hubiese que resolver o las mejoras que hubiera que realizar. Por otra parte, estas pruebas se diferencian del resto en que dependen que exista un proceso del microservicio objetivo en funcionamiento, además de requerir bastante más tiempo para su ejecución.

Para su ejecución, es necesario levantar ambos servicios de forma local en la misma máquina, de forma que se puedan comunicar entre sí, una vez iniciados se realiza la lla-

mada HTTP utilizando la herramienta Postman, como se mostraba en la figura 4.7, tanto si ha aparecido un error como si todo ha ido correctamente Postman mostrará la respuesta de la llamada mostrando esto último. Por otro lado, una vez terminada la llamada o durante su misma ejecución, se puede observar en una base de datos local el estado de la migración, pudiendo revisar el número de entidades migradas por cada tabla y las incidencias creadas.

Además de estas pruebas en local, en un futuro se realizarán pruebas beta con clientes, donde se les instalará una versión de este nuevo producto, ejecutando la migración de los datos para enseñar las nuevas funcionalidades, y donde la migración de los datos formará una parte fundamental de este proceso.

4.5.3. Resultado de la aplicación de las pruebas

Gracias a las pruebas implementadas y ejecutadas se han detectado varios fallos, no solo sobre carencias en el proceso del microservicio, como en la lectura de datos o la gestión de incidencias, sino también errores externos, como por ejemplo, sobre la comunicación entre los microservicios o errores preexistentes en alguno de los productos. A continuación se comentan algunos de estos.

Unos de los primeros fallos que surgieron durante las primeras pruebas sobre datos reales estaban relacionados con unos *timeouts*, uno de ellos era provocado al esperar la respuesta después del envío de datos al microservicio de destino, pues dependiendo del número de datos que hubiesen sido enviados, se necesitaba algo más de tiempo para que el microservicio de destino validase y crease los registros necesarios, además de devolver una respuesta con los posibles errores. Como esto se trataba de algo externo al microservicio de migración, la solución fue sencilla, ya que fue suficiente con incrementar el tiempo de espera. Otro tipo de *timeout* que ha provocado ciertos errores está relacionado con la cantidad de registros a crear en la base de datos del microservicio de destino, pues dentro de este último existe un tiempo máximo para su creación generando un error si este es superado, para evitarlo se decidió que el envío de los registros a crear, se realizase por grupos reduciendo el tiempo necesario para la creación de cada grupo de datos.

Otro fallo que apareció durante las primeras pruebas estaba asociado con la lectura de base de datos, el fallo se daba cuando se realizaba la lectura de algún valor nulo, cuando se esperaba que los valores de esa columna de la tabla no pudiesen ser nulos. Para solucionarlo se modificó la herramienta de generación de los DTO que representan la estructura de base de datos para que incluyese si los campos eran requeridos o no dependiendo si la columna permitía valores nulos. Por otra parte, se crearon varios métodos de ayuda para simplificar la lectura de dichos registros y reducir los posibles fallos que se pudiesen producir por una equivocación durante la creación del código de lectura.

Por último, uno de los fallos externos al desarrollo que las pruebas sobre datos reales ayudaron a encontrar, fue sobre un cálculo incorrecto que se estaba realizando en uno de los campos calculados de las entidades del nuevo producto. Este error se encontró observando las numerosas incidencias creadas utilizando el patrón sobre valores calculados explicado en el apartado 4.4.2.

4.6 Metodología

Durante el desarrollo de este proyecto se han seguido múltiples prácticas de las metodologías ágiles, estas han servido, principalmente, para mejorar la flexibilidad sobre la creación de las distintas funcionalidades, conforme se iban requiriendo y organizar el

proceso para un mejor trabajo en equipo. Algunas de las metodologías ágiles más conocidas son Scrum, programación extrema o *Extreme Programming* (XP) y Kanban entre otras, estas se enfocan por lo general en cuatro valores centrales [28].

- Valorar a los individuos y las interacciones por encima de los procesos y herramientas utilizadas, para dar especial atención a la comunicación y trabajo en equipo. Esto es debido a que desde este punto de vista el resultado final del producto que se desarrolle dependerá más de las personas que lo produzcan que de las herramientas que se acaben utilizando. este valor se ha aplicado de forma activa realizando reuniones de forma frecuente donde se comentaban las tareas a realizar o las funcionalidades a mejorar, entre otras cosas.
- Centrar los esfuerzos en el correcto funcionamiento del software, antes que en crear su documentación, que, por un lado, puede ayudar al resto del equipo a entender mejor el trabajo realizado por parte de otros compañeros, pero, por otro lado, se ha de saber que el objetivo final debe ser la creación de un buen producto y no la documentación sobre este. Esto ha sido especialmente destacable en este proyecto, pues se ha buscado que los propios modelos sirvan como documentación al representar la estructura del código.
- Colaborar de forma estrecha con el cliente durante todo el proceso, comunicándose con él frecuentemente para conocer sus ideas y comprender lo que realmente el cliente quiere sobre el producto *software*. No se ha podido aplicar especialmente en este proyecto, pues como el RNF03 indica, el proceso de migración será realizado en segundo plano, pero se ha querido tener presente esta idea de otra forma, utilizando como datos de prueba los datos reales de un cliente y en el futuro utilizando los de varios clientes y realizando pruebas beta del nuevo producto para conocer su opinión.
- Responder y recibir los cambios abiertamente en lugar de trazar un plan permanente desde el inicio, puesto que en la mayoría de los procesos de desarrollo de *software* surgen cambios que el proyecto debe ser capaz de gestionar. Este caso se ha dado durante la creación de las funcionalidades del microservicio donde por una u otra razón se han ido priorizando unas sobre otras conforme se requerían.

Además de haber realizado el proyecto teniendo presentes estos valores centrales, se han utilizado diversas prácticas de las diferentes metodologías ágiles que han ayudado a organizar el proyecto desde este punto de vista. Algunas de estas prácticas utilizadas son las siguientes [29]:

- **Abordar trabajo de forma incremental.** Como se ha comentado en el apartado 4.4 el proceso de desarrollo se ha ido realizando por fases, en las que en cada una de ellas existía un objetivo principal, pero a su vez se añadían nuevas pruebas y migraciones de nuevas entidades que comprobasen el correcto funcionamiento de lo desarrollado hasta el momento. Todo esto se ha tratado de forma conjunta, centrando los esfuerzos no solo en una tarea, sino en crear cada vez una versión mejorada del microservicio.
- **Documentar, pero solo lo estrictamente necesario.** Al igual que se ha comentado en el segundo valor, la documentación del proyecto se ha intentado que sea lo más práctica posible, documentado solo aquellas parte que ayuden a comprender mejor el proyecto y utilizando los modelos, como un esquema que sirva para comprender el diseño interno del código del proyecto.

- **Automatizar las pruebas para poder garantizar que el producto mantiene el comportamiento deseado cuando se realizan cambios.** Esta práctica ha sido ejecutada a partir de las pruebas creadas a partir de los métodos de prueba que se generan desde los modelos, estos no solo comprueban que cada migración por separado se halla programado de forma correcta, sino que también sirven para probar que el proceso general de migración funciona correctamente cuando se produce algún cambio, pues en caso de introducir algún error habría un gran conjunto de métodos de prueba que ayudarían a detectarlo.

- **Realizar entregas frecuentes de unidades de trabajo terminadas.** Se ha llevado a cabo a partir de una integración continua a partir de la integración del trabajo realizado en las herramientas que utiliza el microservicio y de este mismo con el resto del producto en el que trabajan el resto del equipo.

- **Evitar invertir esfuerzo en adelantar trabajo.** Un punto importante ha sido ir desarrollando aquellas herramientas o funcionalidades que se necesitasen en el proceso de migración conforme se fuesen requiriendo, evitando avanzar con otras menos necesarias en cada punto del proceso. Un ejemplo de esto sería que la primera parte del desarrollo, se priorizó crear la migración de unas pocas entidades sobre el desarrollo de la gestión de incidencias, para permitir la realización de pruebas sobre datos reales y comprobar que el algoritmo funcionase correctamente.

4.6.1. Cronología del proyecto

En esta memoria se han presentado las distintas tareas realizadas en orden cronológico, en especial el apartado 4.4 donde se han mostrado en orden las distintas fases del proceso de programación. A continuación se muestran presentados a grandes rasgos de forma cronológica las partes más importantes de la evolución del trabajo.

- **Estudio de tecnologías y herramientas a utilizar**. El primer paso, para poder comprender en profundidad el proceso de migración, fue revisar las distintas herramientas que existen para el desarrollo de esta tarea, eligiendo finalmente el método que más flexibilidad permitiría a largo plazo y que mejor se adaptaría al resto del producto. Esto sería el apartado 2.

- **Estudio del problema a resolver.** A continuación, antes de poder empezar con cualquier tarea, se analizaron los requerimientos del microservicio y la solución que se quería implementar a partir de este. Esto hace referencia al apartado 4.1 y 4.2.

- **Programación**. Esta parte del desarrollo se ha dividido en cuatro fases incrementales explicadas en el punto 4.4, además de incluir el diseño y estructura utilizados, y las pruebas realizadas o implementadas en los apartados 4.3 y 4.5 respectivamente.

- **Finalización de la memoria.** La última tarea en este trabajo ha sido la revisión y finalización de esta memoria que ha sido elaborada de forma conjunta al desarrollo del proyecto.

En la figura 4.22 se muestra una línea de tiempo de las tareas o hitos realizados en el proceso de programación del microservicio.

Figura 4.22: línea de tiempo del proceso de programación

Por último, cabe mencionar que se han creado más de 2000 líneas de código para el desarrollo del algoritmo de migración, y gracias a la implementación del patrón plantilla, para crear la migración de una nueva tabla es necesario elaborar únicamente alrededor de 200 líneas de código. El proceso para construir la estructura del microservicio y el algoritmo principal junto con la creación de algunas migraciones y pruebas sobre estas ha llevado un total de 250 horas.

CAPÍTULO 5
Conclusiones y trabajo futuro

Consideramos que los objetivos de este trabajo se han cumplido. La realización de este proyecto ha dado como resultado la creación de un microservicio, que permite realizar y gestionar la migración de los datos del producto actual de la empresa a uno de los microservicios en el nuevo producto. Además, se administra todo el ciclo de vida del proceso de migración, realizando no solo una primera iteración sino varias de forma consecutiva, permitiendo así que los usuarios puedan seguir utilizando el producto actual durante el proceso de migración.

En cuanto a los Casos de Uso, se han cubierto y cumplido con todos ellos para los datos contemplados actualmente, desarrollando todas las características necesarias para que el microservicio realice un procedimiento de migración, aunque es posible que funcionalidades como la creación de incidencias y su resolución sean mejoradas en el futuro al contemplar más casos cuando se añadan las migraciones de nuevas tablas, pues por el momento solo está disponible la migración de una parte de estas.

Hablando sobre el estado del microservicio, actualmente este todavía se encuentra en desarrollo, pero cabe mencionar que las pruebas realizadas han sido satisfactorias. Para comprobar que se cumplía con todas las restricciones descritas en los requisitos funcionales, se han llevado a cabo las pruebas sobre datos reales mencionadas en una de las máquinas de desarrollo, dando especial atención a los requisitos RNF04 y RNF05. Se ha analizado, por tanto, que el proceso de migración no consume apenas recursos en la máquina y que durante las pruebas, la migración ha sido realizada en un tiempo menor al esperado para una gran cantidad de datos.

Desde el primer momento se le ha dado prioridad al desarrollo del histórico de datos migrados y de incidencias. Esto ha sido a razón de que este registro de migraciones permite obtener información de las posibles carencias del algoritmo de migración, tanto del proceso general como del código específico que realiza la migración de cada tabla por separado. Asimismo, ha ayudado a encontrar peculiaridades de datos reales que podrían no haberse tenido en cuenta durante su análisis.

Uno de los aspectos negativos de este proyecto es la complejidad del proceso o algoritmo general de migración de datos, pues, al reducir todo lo posible la cantidad de código a crear por cada nueva migración específica a implementar, esto ha hecho que el desarrollo sea bastante difícil. Además, este proceso debe tener en cuenta todo el ciclo de vida de la migración para cada tipo de datos. Pese a esto, se ha logrado cumplir con el objetivo reduciendo sustancialmente el código a implementar, disminuyendo en gran medida el tiempo necesario de desarrollo.

Desde el punto personal, los conocimientos adquiridos en los estudios realizados que han dado lugar a este trabajo, han ayudado al autor a seguir buenas metodologías de desarrollo de *software*, blindándole además la capacidad de trabajar utilizando herramientas

como las DSL *Tools*, que no había utilizado anteriormente, siguiendo un proceso de desarrollo dirigido por modelos. Por otro lado, aunque en las asignaturas cursadas no se han utilizado de forma directa las tecnologías empleadas en este proyecto, como pueden ser el lenguaje C# o el *framework* .NET, los conocimientos generales adquiridos sobre el desarrollo software han permitido que al autor no le supusiese un gran esfuerzo adaptarse a estas tecnologías.

Sobre la experiencia personal ha sido muy gratificante haber podido aplicar los conocimientos tanto teóricos como prácticos adquiridos en la carrera en un proyecto real, creando prácticamente desde cero uno de los componentes que compondrá un producto *software* con valor en el mercado. Asimismo, ha permitido ampliar la experiencia laboral del alumno dentro de una empresa del sector de la informática, aprendiendo de compañeros con más conocimiento.

Para acabar, hablando del futuro del microservicio, aún queda bastante trabajo pendiente, entre este, crear las migraciones específicas del resto de datos, de los que posiblemente surjan nuevos patrones de migración de datos o nuevos tipos de incidencias a gestionar, realizar pruebas con clientes para ver el comportamiento del microservicio sobre distintos datos reales y el futuro despliegue de este junto al nuevo producto en desarrollo, además de comprobar que se cumple con las restricciones de los requisitos no funcionales cuando el proceso se ejecute en diferentes ordenadores.

Referencias

[1] ¿Qué es la privacidad de la base de datos? (Consultado en 04/2022) Consultar a https://spiegato.com/es/que-es-la-privacidad-de-la-base-de-datos.

[2] *A Systematic Approach to Bad Data*, 15/06/2021 (Consultado en 04/2022) Consultar a https://towardsdatascience.com/a-systematic-approach-to-bad-data-6c7e2f86e5ef.

[3] *5 V's of big data*, 03/2021 (Consultado en 04/2022) Consultar a https://www.techtarget.com/searchdatamanagement/definition/5-Vs-of-big-data.

[4] *Data Migration*, 29/10/2019 (Consultado en 04/2022) Consultar a https://www.ibm.com/cloud/learn/data-migration.

[5] *Microservices*, 30/03/2021 (Consultado en 04/2022) Consultar a https://www.ibm.com/cloud/learn/microservices.

[6] *Machine learning*, actualizado en 03/2021 (Consultado en 04/2022) Consultar a https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML.

[7] *Model Driven Software Development* (Consultado en 06/2022) Consultar a https://martinfowler.com/bliki/ModelDrivenSoftwareDevelopment.html.

[8] *Documentación oficial de SqlClient* (Consultado en 05/2022) Consultar a https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient?view=dotnet-plat-ext-6.0.

[9] *SQL Injection* (Consultado en 05/2022) Consultar a https://www.w3schools.com/sql/sql_injection.asp.

[10] Documentación oficial de ASP.NET (Consultado en 04/2022) Consultar a https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0.

[11] Documentación oficial de C# (Consultado en 04/2022) Consultar a https://docs.microsoft.com/en-us/dotnet/csharp/.

[12] Documentación oficial de SQL Server Management Studio (Consultado en 04/2022) Consultar a https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15.

[13] Qué es la Inyección de Dependencias y cómo funciona (Consultado en 05/2022) Consultar a https://www.campusmvp.es/recursos/post/que-es-la-inyeccion-de-dependencias-y-como-funciona.aspx.

[14] JSend (Consultado en 05/2022) Consultar a https://github.com/omniti-labs/jsend.

[15] *Arrang-Act-Assert: A pattern for writting good tests* (Consultado en 05/2022) Consultar a https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/.

[16] *Diseño lógico de bases de datos con Unified Modeling Language (Lenguaje de creación de modelos unificados)*, actualizado en 08/06/2022 (Consultado en 06/2022) Consultar a https://www.ibm.com/docs/es/db2-for-zos/12?topic=relationships-logical-database-design-unified-modeling-language.

[17] *Modelar el SDK de Visual Studio - Lenguajes específicos de dominio*, (Consultado en 06/2022) Consultar a https://docs.microsoft.com/es-es/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2022.

[18] *Introducción al Metamodelado*, escrito en 27/09/2017 (Consultado en 06/2022) Consultar a http://timeofsoftware.com/2017/09/27/introduccion_metamodelado/.

[19] *Arquitectura Dirigida por Modelos (MDA)*, (Consultado en 06/2022) Consultar a https://ingenieriadelsoftwareuah2015.wordpress.com/2015/03/23/arquitectura-dirigida-por-modelos-mda/.

[20] *Arquitecturas monolíticas o arquitectura de microservicios: ventajas e inconvenientes*, (Consultado en 06/2022) Consultar a https://www.ilimit.com/blog/arquitecturas-monoliticas-o-arquitectura-de-microservicios-ventajas-e-inconvenientes/.

[21] *Comparación entre la arquitectura monolítica y la arquitectura de microservicios*, (Consultado en 06/2022) Consultar a https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith#:~:text=Una%20aplicaci%C3%B3n%20monol%%ADtica%20se%20compila,pueden%20implementar%20de%20forma%20independiente..

[22] *ISO/IEC 25010* (Consultado en 05/2022) Consultar a https://iso25000.com/index.php/normas-iso-25000/iso-25010.

[23] *¿QUÉ ES CRUD?* (Consultado en 05/2022) Consultar a https://www.cdainfo.com/es/noticias/182-que-es-crud.

[24] *Una introducción a NuGet* (Consultado en 05/2022) Consultar a https://docs.microsoft.com/es-es/nuget/what-is-nuget.

[25] *Generación de código y plantillas de texto T4* (Consultado en 05/2022) Consultar a https://docs.microsoft.com/es-es/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2022.

[26] *Template Method* (Consultado en 05/2022) Consultar a https://refactoring.guru/design-patterns/template-method.

[27] *Unit Tests, How to Write Testable Code and Why it Matters* (Consultado en 05/2022) Consultar a https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters.

[28] *Manifiesto por el Desarrollo Ágil de Software* (Consultado en 06/2022) Consultar a http://agilemanifesto.org/iso/es/manifesto.html.

[29] *Catálogo de Prácticas Ágiles* (Consultado en 06/2022) Consultar a http://agilev-roadmap.herokuapp.com/InfoPracticas.

APÉNDICE A
Objetivos de Desarrollo Sostenible

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

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

Como se ha comentado, la empresa en la que se desarrolla este proyecto ofrece como producto y servicio software un sistema de planificación de recursos empresariales (ERP) para la gestión y administración del ámbito sociosanitario, enfocado principalmente en los centros geriátricos. Actualmente, la empresa tiene la intención de crear una nueva versión del producto, en la que se añadirán nuevas mejoras y funcionalidades, y que tendrá el mismo público objetivo.

El microservicio de migración de datos que se desarrolla en este trabajo tiene como finalidad la transferencia de datos, desde la versión actual del producto hacia la nueva versión de este. Realizar esta migración de los datos de forma automática entre ambos productos, permitirá que los clientes y usuarios puedan empezar a utilizar el nuevo producto, sin necesidad de invertir una gran cantidad de tiempo en realizar a mano este proceso, el cual estaría sujeto a errores.

Es debido a los propósitos que se pretenden cumplir con la realización de este trabajo y a la naturaleza del software que ofrece la empresa del que forma parte este microservi-

cio, que se puede decir que este proyecto tiene un alto grado de relación con el Objetivo de Desarrollo Sostenible de salud y bienestar. Asimismo, aunque no está relacionado directamente con las metas que se buscan cumplir, sí que comparte los fundamentos del objetivo tres, pues se busca garantizar una mejor vida y promover el bienestar en todas las edades. Esto se lleva a cabo mediante la creación de una nueva versión del producto que permita a las residencias geriátricas funcionar mejor y facilitar el trabajo tanto de los cuidadores como del personal sanitario. Además, recientemente la gestión y administración de las residencias geriátricas ha jugado un papel muy importante en la lucha contra la reciente pandemia del COVID-19, es por ello que mejorar el *software* que se utiliza dentro de estas, es indispensable para proveer de mejores herramientas frente a los nuevos problemas sanitarios que puedan surgir en el futuro.

Sobre las contribuciones específicas que este proyecto de migración de datos realiza sobre el objetivo de salud y bienestar, estas se pueden resumir en dos contribuciones principales. Por un lado, facilita una mejor adopción del nuevo *software* por parte de los trabajadores, debido a que los datos y las configuraciones que son utilizados en el producto actual por parte de los usuarios, se encontrarán en la nueva versión desde el primer momento, pudiendo utilizar el producto de forma eficaz desde el primer día. Por otro lado, los usuarios de la aplicación, es decir, los trabajadores de la residencia, no deberán dedicar una gran cantidad de tiempo a copiar los datos en el nuevo producto, o consultarlos reiteradamente utilizando el producto actual, por lo que podrán dedicar este tiempo a la atención de los residentes, en lugar de gestionar los problemas que la nueva versión de este producto pueda suponer.

Por último, esta nueva versión del producto se está desarrollando dentro del departamento de I+D+i de la empresa, donde se están creando nuevas funcionalidades para esta versión. Algunas de estas nuevas funcionalidades están relacionadas con la inteligencia artificial, llevando a cabo varias innovaciones en el *software* del sector sociosanitario. Por ello, este trabajo también está relacionado, aunque en menor medida, con el objetivo nueve de industria, innovación y bienestar, al tratarse de una parte esencial, para que los usuarios puedan utilizar estas nuevas funcionalidades con los datos contenidos en el producto actual, y que el desarrollo de estas funcionalidades se puedan realizar utilizando datos reales dentro del nuevo producto.