ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELECTRÍCA Y ELECTRÓNICA

DESARROLLO DE UN SISTEMA DISTRIBUIDO DE GESTIÓN DE FOTOGRAFÍAS

SUBSISTEMA DE CLASIFICACIÓN

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN TECNOLOGÍAS DE LA INFORMACIÓN

LUIS EDUARDO OÑA NAVARRETE

luis.ona01@epn.edu.ec

DIRECTOR: RAÚL DAVID MEJÍA NAVARRETE

david.mejia@epn.edu.ec

DMQ, Abril 2023

CERTIFICACIONES

Yo, LUIS EDUARDO OÑA NAVARRETE declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

LUIS EDUARDO OÑA NAVARRETE

Certifico que el presente trabajo de integración curricular fue desarrollado por LUIS EDUARDO OÑA NAVARRETE, bajo mi supervisión.

RAÚL DAVID MEJÍA NAVARRETE

DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el (los) producto(s) resultante(s) del mismo, son públicos y estarán a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

LUIS EDUARDO OÑA NAVARRETE

RAÚL DAVID MEJÍA NAVARRETE

DEDICATORIA

Este Trabajo de Integración Curricular va dedicado a todas las personas que estuvieron apoyándome desde el principio de mi etapa universitaria.

A mi madre, quien es mi pilar fundamental, ya que gracias a su apoyo incondicional logré el objetivo de terminar mi carrera universitaria.

A mi padre, quien siempre me dio ánimos y consejos para continuar con mi carrera universitaria.

Luis Eduardo Oña Navarrete

AGRADECIMIENTO

Agradecer primeramente a Dios, por darme cada día la oportunidad de seguir con vida y de permitirme seguir con mi familia.

Agradecer a mi tutor de este Trabajo de Integración Curricular, al Ing. David Mejía, quien estuvo siempre supervisando y observando que la realización de este trabajo sea llevado a cabo en los tiempos establecidos.

Agradecer a mis compañeros de la universidad, con quienes construí una amistad y permitieron que la vida universitaria fuese una etapa equilibrada entre el estudio y la diversión.

Agradecer a mi novia, quien fue un apoyo incondicional durante el transcurso de la carrera en una etapa donde las cosas parecían no tener solución.

Luis Eduardo Oña Navarrete

ÍNDICE DE CONTENIDO

CERTIFICACIONES I
DECLARACIÓN DE AUTORÍA II
DEDICATORIA III
AGRADECIMIENTO IV
ÍNDICE DE CONTENIDO V
RESUMEN VII
ABSTRACT VIII
1 INTRODUCCIÓN 1
1.1 OBJETIVO GENERAL 2
1.2 OBJETIVOS ESPECÍFICOS 2
1.3 ALCANCE 3
1.4 MARCO TEÓRICO 5
1.4.1 ASP.NET WEB API 5
1.4.1.1 NAMESPACES 8
1.4.1.2 INTERFAZ IFORMFILE 12
1.4.1.3 AUTENTICACIÓN CON JWT 13
1.4.1.4 AUTOMAPPER 15
1.4.1.5 NEWTONSOFTJSON 15
1.4.2 ARQUITECTURA REST 16
1.4.3 MODELO DE CAPAS 16
1.4.4 SWAGGER API 17
1.4.5 METODOLOGÍA KANBAN 18
2 METODOLOGÍA 20
2.1 DISEÑO 20
2.1.1 TABLERO KANBAN 20
2.1.2 REQUERIMIENTOS 21
2.1.3 DISEÑO UML (UNIFIED MODELING LANGUAGE) 24

2.2 IMPLEMENTACIÓN 28
2.2.1 CONFIGURACIÓN DE LA CLASE STARTUP 28
2.2.2 IMPLEMENTACIÓN DE ENTIDADES 29
2.2.3 IMPLEMENTACIÓN DE LA CAPA DE ACCESO A DATOS 30
2.2.4 IMPLEMENTACIÓN DE LA CAPA DE MODELO DE DATOS 32
2.2.5 IMPLEMENTACIÓN DE LA CAPA LÓGICA 34
2.2.6 IMPLEMENTACIÓN DEL CONTROLADOR 40
3 RESULTADOS, CONCLUSIONES Y RECOMENDACIONES 52
3.1 RESULTADOS 52
3.1.1 PRUEBAS DE FUNCIONAMIENTO 52
3.2 CONCLUSIONES Y RECOMENDACIONES 62
3.2.1 CONCLUSIONES 62
3.2.2 RECOMENDACIONES 64
4 REFERENCIAS 66
5 ANEXOS 70
ANEXO I 71
ANEXO II 72
ANEXO III 73

RESUMEN

En el presente Trabajo de Integración Curricular se detalla el proceso de diseño e implementación de un subsistema de clasificación, el cual forma parte de un sistema distribuido de gestión de fotografías.

El subsistema de clasificación se encarga de generar los directorios y rutas de almacenamiento de las diferentes fotografías, así como las relaciones que deben existir entre las fotografías y sus etiquetas. Se obtiene, tanto las fotografías como sus etiquetas, del subsistema de adquisición, y esta información se envía al subsistema de almacenamiento para que el usuario pueda recuperarla mediante el subsistema de consultas. Para este subsistema se implementará una web API (*Application Programming Interfaces*), y en este documento se describe el proceso de planificación, diseño, implementación y ejecución de esta web API. Así también este documento está organizado en tres capítulos.

En el primer capítulo se presentan los conceptos necesarios para la realización de una web API, incluyendo el *framework* de desarrollo ASP.NET web API, las herramientas definidas en *namespaces* de .NET, las librerías para el manejo de datos, la arquitectura REST (*Representational State Transfer*) para el desarrollo de microservicios, el modelo de capas para el desarrollo de software y la herramienta Swagger API. Además, se presenta información relativa a la metodología Kanban, la cual fue aplicada durante el proceso de desarrollo del subsistema.

En el segundo capítulo, se detalla el diseño del subsistema que incluye diagramas de estructura, clases y actividades, obtenidos a partir del análisis de requerimientos, posteriormente se describe el proceso de implementación de los componentes que conforman la web API.

En el tercer capítulo, se evalúan los *endpoints* de la web API haciendo uso de la herramienta Swagger API, posteriormente se presenta el conjunto de conclusiones y recomendaciones a las que se llegó al finalizar el Trabajo de Integración Curricular.

Finalmente, se presentan los anexos que contienen: la visualización del tablero Kanban, el proceso de instalación y configuración del entorno de desarrollo, y el código implementado.

PALABRAS CLAVE: Web API, ASP.NET, REST, Swagger, Kanban, *endpoints*.

ABSTRACT

In the present Curricular Integration Work details the design and implementation process of a classification subsystem, which is part of a distributed photo management system.

The subsystem is responsible for generating the directories and storage paths of the different photographs, as well as the relationships that must exist between photographs and their labels. Both the photographs and their labels are obtained from the acquisition subsystem, and this information is sent to the storage subsystem so that the user can retrieve it through the query subsystem. For this subsystem a web API (*Application Programming Interfaces*) will be implemented, and in this document describes the process of planning, designing, implementing, and executing this web API. So too this document is organized into three chapters.

In the first chapter, presents the concepts necessary for the realization of a web API, including the ASP.NET web API development framework, the tools defined in .NET namespaces, libraries for data handling, the REST (*Representational State Transfer*) architecture for microservices development and the Swagger API tool. In addition, information is provided on the Kanban methodology, which was applied during the subsystem development process.

In the second chapter, we detail the design of the subsystem that includes diagrams of structure, classes, and activities, obtained from the requirements analysis, then describes the implementation process of the components that make up the web API.

In the third chapter, web API endpoints are evaluated using the Swagger API tool, then presents the set of conclusions and recommendations reached at the end of the Curricular Integration Work.

Finally, presented, the annexes containing: the visualization of the Kanban board, the process of installation and configuration of the development environment, and the code implemented.

KEYWORDS: Web API, ASP.NET, REST, Swagger, Kanban, *endpoints*.

1 INTRODUCCIÓN

En este Trabajo de Integración Curricular se desarrolló un subsistema de clasificación de imágenes a partir de etiquetas asociadas a las mismas, esto permitirá a los usuarios, que se conecten al subsistema, mediante una interfaz, consumir las funcionalidades implementadas en una web API[1] (*Application Programming Interface*). Este subsistema será desarrollado usando ASP.NET[2] web API.

La web API tiene como objetivo primordial recibir un conjunto de fotografías que puede estar conformado por una o varias imágenes, las cuales pueden tener uno de los siguientes formatos: .JPG (*Joint Photographic Experts Group*), .PNG (*Portable Network Graphics*) o .GIF (*Graphics Interchange Format*), así como un conjunto de etiquetas que el usuario ha agregado a la o las fotografías. Una vez enviada esa información, la tarea que realizará el subsistema es generar los directorios y rutas en los que se almacenará la imagen, así como la relación de estas con las etiquetas correspondientes.

La información generada por el subsistema de clasificación debe ser remitida a un subsistema de almacenamiento que gestione una base de datos, la cual servirá de soporte para tener la información almacenada y que el usuario pueda recuperar sus imágenes a partir de consultas que acepten como argumentos las etiquetas correspondientes.

Para desarrollar la lógica requerida para el almacenamiento masivo de imágenes con etiquetas, se empleará la interfaz `IFormFile`, la cual ofrece acceso a metadatos propios de los archivos, los cuales pueden ser transmitidos en solicitudes del protocolo HTTP (*Hypertext Transfer Protocol*); el acceso a los metadatos es de gran importancia pues permite establecer restricciones en cuanto al peso y extensiones de los datos que se pueden emplear [1]. Por otra parte, se empleó la tecnología DTO (*Data Transfer Object*), la cual permite disponer de objetos que se utilizan para encapsular datos y poder realizar el intercambio de información entre subsistemas [2]. Para facilitar el intercambio y permitir la transformación de objetos se utilizó `AutoMapper`, una biblioteca que permite realizar asociación de objetos, es decir ofrece la posibilidad de recibir un archivo cuyo contenido se guarda de manera local y únicamente en la base

---

[1] API (*Application Programming Interfaces*): Conjunto de protocolos y herramientas que proporcionan un medio para que los componentes de software puedan comunicarse e intercambiar datos.

[2] ASP.NET: Es un *framework* de código abierto utilizado ampliamente para la creación de aplicaciones web empleando el uso de tecnologías basadas en .NET.

de datos se almacena la URL[3] (*Uniform Resource Locator*) combinada para su almacenamiento; con la URL se puede posteriormente recuperar el recurso [3], [4].

Para probar el subsistema, es necesario que exista un cliente que pueda consumir los *endpoints*[4] desarrollados, para esto se utilizará Swagger API, esta herramienta simula un cliente que consume peticiones HTTP alojadas en una web API [5]. Por otro lado, para gestionar la información que se almacena en la base de datos se utilizará *Entity Framework*[5], la cual crea las tablas en la base de datos, con base en las entidades generadas y mediante una función se puede realizar el almacenamiento de la URL en la cual se encuentra ubicada la imagen [6].

1.1 OBJETIVO GENERAL

Desarrollar un subsistema de clasificación de imágenes mediante el uso de ASP.NET web API, que permita construir un esquema de directorios de manera automática para realizar el almacenamiento de fotografías.

1.2 OBJETIVOS ESPECÍFICOS

1. Analizar la teoría necesaria para implementar el subsistema de clasificación de fotografías.
2. Diseñar la estructura del subsistema de clasificación para establecer la lógica operacional entre los componentes definidos con base en los requerimientos previamente establecidos.
3. Implementar la funcionalidad del subsistema de clasificación con base en el diseño realizado.

---

[3] URL (*Uniform Resource Locator*): Es un tipo de URI (*Uniform Resource Identifier*) que se usa para ubicar y recuperar recursos en Internet.

[4] *endpoint*: Corresponde al extremo donde una API recibe las solicitudes de clientes y provee las respuestas de los datos en un formato de texto como JSON.

[5] *Entity Framework*: Permite a los desarrolladores interactuar con bases de datos relacionales de una manera orientada a objetos y utilizando la plataforma .NET.

1.3 ALCANCE

En el presente Trabajo de Integración Curricular se propone el desarrollo de un subsistema de clasificación de fotografías que permita generar una estructura de directorios de manera automática con base en las etiquetas asignadas a cada imagen.

El subsistema de clasificación se encarga de recibir un conjunto de imágenes y un conjunto de etiquetas que serán entregadas por el subsistema de adquisición. Con los datos proporcionados el subsistema de clasificación tiene por objetivo generar las rutas de almacenamiento de las imágenes proporcionadas por el usuario tomando en cuenta que estas recibirán como parámetro adicional etiquetas que servirán para realizar filtros y recuperación de información a partir de un subsistema de consultas. Una vez que el subsistema de clasificación ha generado las rutas, debe proporcionar esa información al subsistema de almacenamiento.

Para el desarrollo de la web API que contendrá los *endpoints* que podrán ser usados por una aplicación cliente para consumir los servicios que se exponga, se gestionarán los metadatos de la imagen (tipo de archivo, tamaño, dimensiones, entre otros), esto permite la realización de validaciones.

Para la lógica de los *stubs*[6], se utilizarán fragmentos de código que permitan simular las funcionalidades de tal manera que el subsistema de clasificación trabaje de manera independiente y pueda presentar los resultados [7]. Como parte de este trabajo se realizará un estudio de las tecnologías requeridas como el *framework*[7] ASP.NET web API, considerando los componentes que conforman una API (controladores, entidades, funciones, etc.), de la misma manera se realizará un estudio de espacios de nombres[8] como: `Microsoft.EntityFrameworkCore`, `System.IO`, `System.IdentityModel`, `System.ComponentModel`, `System.Security`, `Microsoft.AspNetCore`. Por otra parte, se revisará la librería especializada para el manejo de datos denominada: `AutoMapper`. También se revisará la arquitectura REST (*Representational State Transfer*) para el desarrollo de microservicios; y, por último, se realizará una revisión de la metodología ágil Kanban para el desarrollo de software, esto

[6] *stub*: Pequeña pieza de código que se usa se usa para simular el comportamiento del componente faltante.

[7] *framework*: Conjunto de herramientas, bibliotecas y convenciones para crear y estructurar aplicaciones de software.

[8] espacio de nombres: También denominados *namespaces*, se utilizan para agrupar una colección de clases, interfaces y otros tipos de datos relacionados que comparten una funcionalidad o un propósito en común.

con la finalidad de emplear ciertos componentes de esta metodología para el desarrollo de este Trabajo de Integración Curricular.

Por otro lado, como parte de la implementación del subsistema, se considerarán 3 fases para la misma: Diseño, Implementación y Pruebas.

En la fase de diseño se establecerán los requerimientos funcionales y no funcionales del subsistema de clasificación. Se generarán los diagramas haciendo uso de UML[9] (*Unified Modeling Language*); el diagrama de clases detallado con las propiedades, funciones y dependencias entre clases para el funcionamiento de la web API; el diagrama de actividades para representar el flujo de actividades del proceso de clasificación de las imágenes y del *stub* que realiza el filtrado de las imágenes. Finalmente, se establecerá el listado de actividades haciendo uso de un tablero Kanban con la finalidad de manejar ordenadamente todas las fases del trabajo.

En la fase de implementación se realizará la instalación del ambiente de desarrollo con las dependencias para ASP.NET web API y los paquetes Nuget[10] [8]; de manera consecuente se realizará la codificación de clases, funciones y controladores con base en el diseño establecido, así como la lógica requerida, que permitan la creación del esquema de directorios; y, en particular se implementará el *stub* que permita simular la funcionalidad del subsistema de almacenamiento para de manera consecuente realizar las pruebas a partir de Swagger API para emular el funcionamiento del cliente, con el cual se consumirán los *endpoints* del subsistema

Finalmente, se iniciará la fase de pruebas haciendo uso de Swagger API la cual proporciona una interfaz gráfica con el detalle de los *endpoints* desarrollados para la web API, en cada detalle se puede realizar la ejecución del *endpoint* tomando en cuenta los parámetros que debe recibir. Para la prueba de la carga de imágenes y etiquetas se realizarán pedidos usando el método de HTTP POST para enviar al servidor las imágenes y su conjunto de etiquetas; si la operación fue correcta se recibe un `OK` en el cual consta el detalle de la información cargada, caso contrario se recibe un `BadRequest` con el detalle del error generado. El *stub* de recuperación de imágenes es un *endpoint* que realiza una petición GET al servidor y que toma como parámetro las etiquetas anexadas a la imagen; si en el parámetro no se especifican etiquetas

[9]UML (*Unified Modeling Language*): Es un método estandarizado para diseñar, visualizar y documentar sistemas de software.

[10]Nuget**:** Contiene todas las herramientas necesarias para usar una biblioteca en la aplicación, incluidos los ensamblados, los archivos y las dependencias necesarias.

devolverá como resultado un `OK` con la información del conjunto de imágenes que han sido cargadas por el usuario, si se especifica etiquetas se devuelve un `OK` con la información de las imágenes que están asociadas a las etiquetas, caso contrario se devuelve un `BadRequest` en el que se especifica que no existen registros asociados a los parámetros definidos [5].

1.4 MARCO TEÓRICO

Esta sección contiene la información general acerca de los conceptos teóricos del *framework* utilizado para la creación de servicios HTTP; adicionalmente se presenta información sobre el uso de los *namespaces* y librerías requeridas en el desarrollo de *endpoints*, el uso de la arquitectura REST para la generación de microservicios que utilizan de manera directa los métodos ofrecidos por HTTP, y la herramienta Swagger para consumir *endpoints*. De manera final se analiza la metodología Kanban para el desarrollo ágil de software.

1.4.1 ASP.NET WEB API

ASP.NET Web API es un *framework* que sirve para crear servicios HTTP que pueden ser consumidos por una amplia variedad de clientes, incluidos navegadores web, dispositivos móviles, aplicaciones de escritorio, entre otros. Es muy similar a ASP.NET MVC ya que contiene características de MVC (*Model View Controller*) como enrutamiento, controladores, resultados de acciones, filtros, modelos, contenedores IoC [11](*Inversion of Control Container*) o inyección de dependencias, con la diferencia de que envía datos como respuesta, en lugar de una vista HTML (*HyperText Markup Language*) como lo hace MVC [9], [10].

La web API se puede utilizar como una aplicación de servicios web independiente que puede estar asociada a otros tipos de aplicaciones web como ASP.NET MVC [11].

Las aplicaciones desarrolladas con ASP.NET web API sirven para exponer los datos y servicios a diferentes dispositivos. Además, ASP.NET web API es de código abierto, y

---

[11] IoC *(Inversion of Control Container)*: Es un componente de software que administra la creación de instancias y la vida útil de los objetos en una aplicación de software.

es una plataforma ideal para crear servicios REST sobre .NET debido a que utiliza todas las funciones de HTTP, como URI[12] (*Uniform Resource Identifier*), encabezados de solicitud/respuesta, almacenamiento en caché, control de versiones, varios formatos de contenido, entre otros, y no necesita definir ninguna configuración adicional para que sean consumidos desde diferentes dispositivos [4], [10], [11].

En la Figura 1.1. Descripción simple de una web API se observa: el intercambio de solicitudes de HTTP, las cuales son enviadas por los clientes que solicitan recursos alojados en la web API, dichos clientes pueden ser un dispositivo móvil, una aplicación web, entre otros; y el intercambio de respuestas de HTTP, las cuales son enviadas desde la web API y cuyo formato de datos puede ser JSON[13] (JavaScript *Object Notation*), XML[14] (e*Xtensible Markup Language*), entre otros [10].

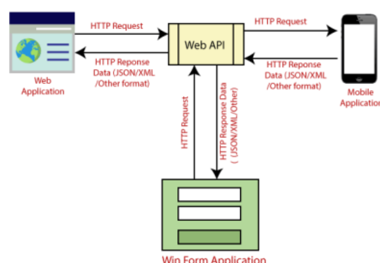

Figura 1.1. Descripción simple de una web API

A diferencia de ASP.NET MVC en ASP.NET Web API se dispone de 2 bloques de desarrollo principales los cuales son entidades y controladores.

Entidades: Son objetos que tienen por objetivo la representación de los datos de la aplicación desarrollada con ASP.NET Web API y a través de los cuales se puede realizar la serialización mediante JSON, XML u otro formato, para consecuentemente escribir las entidades en estos formatos y colocarlas en el cuerpo del mensaje como parte de la respuesta HTTP. De la misma manera, las entidades describen las reglas de manipulación de datos con los cuales se está trabajando, esto quiere decir que se pueden establecer reglas de validación o restricciones a las propiedades de los objetos que se están creando. Esta información que se encuentra en los modelos puede servir de referencia para crear una base de datos usando *Entity Framework,* tecnología que

[12] URI (*Uniform Resource Identifier*): Es una cadena de caracteres que identifican un nombre o un recurso en internet.

[13] JSON (*JavaScript Object Notation*): Es un formato basado en texto que está diseñado para ser fácil de leer, escribir y comprender. Utiliza una sintaxis simple para la representación de datos clave-valor similar a un diccionario.

[14] XML (e*Xtensible Markup Language*): Es un lenguaje de marcado que se utiliza para almacenar y transportar datos.

permite realizar migraciones para la generación de tablas que la aplicación requerirá [6], [9].

Controladores: La idea fundamental de tener una web API, es que se tendrá clientes que harán peticiones HTTP, estas peticiones serán a recursos que se representarán de forma única a través de una URI. La web API tratará estas URI como rutas que se asocian con la ubicación en la que se encuentran los recursos representados a través de las URI. En la Figura 1.2 se presenta un ejemplo de una URI [4], [9].

Figura 1.2. Representación de URI

Al realizar una petición mediante un método HTTP como los descritos en la Tabla 1.1 hacia una ruta gestionada por la web API, se provocará la ejecución de un método específico en un determinado controlador, lo cual se conoce como acción [12].

Tabla 1.1. Métodos del protocolo HTTP

Métodos HTTP	**Funcionalidad**
HTTP HEAD	Tiene funcionalidad similar a HTTP GET, sin embargo, se utiliza solo para recuperar la cabecera del recurso, sin su cuerpo.
HTTP GET	Utilizado para pedir datos al servidor web.
HTTP POST	Sirve para indicar que se requiere enviar información al servidor web, información que viaja en el cuerpo de la petición HTTP.
HTTP PUT	Utilizado para realizar la actualización de recursos.
HTTP PATCH	Sirve para realizar actualizaciones parciales de un recurso.
HTTP DELETE	Utilizado para realizar borrado de recursos.

Teniendo esto en cuenta entonces se puede definir al controlador como una clase que agrupa un conjunto de métodos de acción. En la Figura 1.3 se define la estructura de una clase que representa a un controlador.

```
 1 [ApiController]
 2 [Route("api/imagenes")]
 3 public class ImagenController : ControllerBase
 4 {
 5 public ImagenController()
 6 {
 7     //Constructor del controlador
 8 }
 9
10 [HttpGet("obtenerImgWithDetalle/{id}")]
11 public async Task<ActionResult<ImagenDetalleDTO>> Get(int id)
12 {
13     //Lógica a implementar del endpoint
14 }
15 }
```

Figura 1.3. Estructura de un controlador en ASP.NET Web API

Como se observa en la Figura 1.3 la estructura básica para determinar que una clase es un controlador debe estar definida por algunos parámetros los cuales son: el decorador `[ApiController]` (línea 1) el cual indica que esta clase es un controlador; la ruta del controlador definida por el decorador `[Route]` (línea 2); la definición de la clase que hereda de `ControllerBase`, la cual es una clase que proporciona el soporte para el manejo de solicitudes HTTP (línea 3); el constructor de la clase (de la línea 5 hasta la línea 8); y, los *endpoints* que en este caso se los decora con el método `[HttpGet]` (línea 10) que indica que la función asincrónica `Get` (línea 11) se implementará para obtener un recurso alojado en la web API y se asociará con el método GET de HTTP [13].

Al realizar peticiones HTTP a una aplicación basada en web API, eventualmente se recibe una respuesta HTTP, la cual incluye un código de estado, que corresponde a un número que indica el resultado de la operación, lo cual es de gran ayuda pues indica si las peticiones se respondieron correctamente o si se tiene algún tipo de error. En la Tabla 1.2 se presentan los códigos de estado, en el cual el primer dígito indica la categoría a la que pertenece dicho código [14].

Tabla 1.2. Códigos de estado

Código	Categoría
1XX	Informacional
2XX	Exitoso
3XX	Redirección
4XX	Error del cliente
5XX	Error del servidor

1.4.1.1 NAMESPACES

En el lenguaje de programación C#, para importar un *namespace* o espacio de nombres se coloca la directiva `using` en la parte superior de una clase; con la finalidad de simplificar la codificación y evitar tener que escribir el nombre completo de la clase, el cual incluye el espacio de nombres en el que está definido. En la Figura 1.4 se representa la forma de utilizar la palabra reservada `using` [15].

```
1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace WebApiClasificadorImagenes.DTOS
4 {
5     public class MyPage : Page
6     {
7         //Lógica a desarrollar
8         public override Task ExecuteAsync()
9         {
10            throw new NotImplementedException();
11        }
12    }
13 }
```

Figura 1.4. Uso de la palabra reservada `using`

Como se observa en la Figura 1.4 se define la clase `MyPage` (línea 5) la cual hereda de la clase `Page` que está definida en el espacio de nombres `Microsoft.AspNetCore`, y en el subespacio de nombres `Mvc.RazorPages` [15].

En el espacio de nombres `Microsoft.AspNetCore` están definidos los elementos necesarios para crear aplicaciones web utilizando el *framework* ASP.NET, es decir contiene bibliotecas y herramientas para construir aplicaciones web en la plataforma .NET; así también, contiene subespacios de nombres para funciones que permiten el manejo de solicitudes y respuestas HTTP, el enrutamiento de URI y la gestión del patrón MVC. En la Tabla 1.3 se detallan los subespacios de importancia para el desarrollo de una web API [16].

Tabla 1.3. Subespacios de nombres definidos en `Microsoft.AspNetCore`

Subespacio de nombres	Funcionalidad
`Hosting`	Brinda la capacidad de configurar el entorno y los servicios de almacenamiento, así como la capacidad de iniciar y detener la aplicación.
`Http`	Proporciona herramientas (clases, interfaces, estructuras, etc.) para manejar y manipular solicitudes y respuestas HTTP.
`Mvc`	Proporciona herramientas para la generación de aplicaciones en base de MVC. Incluye clases para manejar solicitudes HTTP y renderizar vistas, así como clases para configurar enrutamiento y validar solicitudes.
`Authorization`	Proporciona herramientas para implementar la autorización en una aplicación y proteger los recursos según el rol o los *claims*[15] del usuario.

En el espacio de nombres `Microsoft.EntityFrameworkCore` existen herramientas para interactuar con bases de datos mediante *Entity Framework Core.* Contiene las clases e interfaces para realizar un CRUD[16] (*Create, Read, Update, Delete*) de información contenida en una base de datos y contiene las clases que representan las

[15] *claims*: Conjunto de datos personales de confianza comúnmente usado en usuarios.

[16] CRUD (*Create, Read, Update, Delete*): Acrónimo para realizar operaciones de generación, lectura, actualización y eliminación de datos almacenados.

entidades de la base de datos. En la Tabla 1.4 se detalla el subespacio de importancia para el desarrollo de una web API [17].

Tabla 1.4. Subespacio de nombre definido en `Microsoft.EntityFrameworkCore`

Subespacio de nombre	Funcionalidad
`Migrations`	Proporciona herramientas para realizar un seguimiento y administrar los cambios en el esquema de la base de datos mediante *EF Core*. Incluye clases para generar, aplicar y revertir migraciones.

Por otro lado, el espacio de nombres `System.IdentityModel` proporciona herramientas para trabajar con *tokens*[17] de seguridad como SAML[18] (*Security Assertion Markup Language*) y JWT[19] (JSON *Web Token*) en ASP.NET.
Incluye clases para crear, leer y validar *tokens* de seguridad y manejar excepciones. Es útil para implementar autenticación y autorización basadas en *tokens* en aplicaciones .NET. En la Tabla 1.5 se detalla el subespacio de importancia para el desarrollo de una web API [18].

Tabla 1.5. Subespacio de nombre definido en `System.IdentityModel`

Subespacio de nombre	Funcionalidad
`Tokens`	Proporciona clases como `JwtSecurityToken` el cual contiene propiedades para acceder al encabezado y la carga útil del *token*, así como funciones para validar el mismo y generar la firma. Se utiliza de manera conjunta junto con la clase `JwtSecurityTokenHandler` para manejar la creación, validación y procesamiento de JWT.

En el espacio de nombres `System.ComponentModel` se tienen clases e interfaces para personalizar el comportamiento de los componentes. Tiene clases para notificar a los clientes de cambios en las propiedades y para proporcionar información sobre errores en las propiedades. También proporciona información sobre los componentes y permite personalizarlos en un entorno de tiempo de diseño. Este espacio de nombres es útil para crear componentes reutilizables, extensibles y configurables. En la Tabla 1.6 se detalla el subespacio de importancia para el desarrollo de una web API [19].

---

[17] *Tokens:* Cadena de caracteres que representan un identificador único que otorga derechos para realizar operaciones dentro de un sistema.

[18] SAML (*Security Assertion Markup Language*): Es un estándar ampliamente utilizado para el intercambio de información de autenticación y autorización entre partes.

[19] JWT (JSON *Web Token*): Es un formato compacto, seguro para URL y firmado digitalmente para transmitir información de forma segura.

Tabla 1.6. Subespacio de nombre definido en System.ComponentModel

Subespacio de nombre	Funcionalidad
DataAnnotations	Incluye atributos como Required, StringLength y Range que se pueden aplicar a clases y propiedades para especificar las reglas de validación y los atributos. Además, proporciona clases base como ValidationAttribute la cual obliga a la clase personalizada a heredar el método IsValid en donde se debe detallar la lógica de validación del objeto.

En el espacio de nombres System.Security se definen clases e interfaces para la protección de aplicaciones en ASP.NET. Incluye clases para realizar cifrado y descifrado, así como para para crear y gestionar firmas digitales al igual que claves criptográficas.

En la Tabla 1.7 se detalla el subespacio de importancia para el desarrollo de una web API [20].

Tabla 1.7. Subespacio definido en System.Security

Subespacio de nombre	Funcionalidad
Claims	Proporciona clases e interfaces para trabajar con identidades basadas en *claims*. Una clase de este subespacio es la de claimTypes la cual contiene un conjunto de tipos de *claims* predefinidos (*UserName*, *Email*, entre otros) que se pueden usar en la identidad basada en *claims*.

En el espacio de nombres System.IO se tienen clases e interfaces para realizar operaciones de entrada y salida, como leer y escribir archivos, trabajar con directorios y sistemas de archivos.

En la Tabla 1.8 se especifican las clases de importancia para el desarrollo de una web API [21].

Tabla 1.8. Clases de importancia definidas en System.IO

Nombre de la clase	Funcionalidad
File	Posee funciones que sirven para realizar un CRUD con archivos los cuales son: Create, Move, Delete, Exists, entre otros. De igual forma contiene funciones para crear archivos a partir de la escritura de una matriz de bytes, por ejemplo, la función WriteAllBytesAsync.
Path	Proporciona funciones para trabajar con rutas de archivos y directorios. Se utilizan para combinar y manipular cadenas de ruta, obtener el nombre y la extensión del archivo, obtener la ubicación de carpetas especiales entre otros. Los principales métodos utilizados son: Combine, GetFileName, Exists, entre otros.
Directory	Proporciona funciones para trabajar con directorios y sistemas de archivos. Incluye métodos como CreateDirectory el cual realiza la generación de directorios y subdirectorios según una ruta especifica. Al igual que File implementa funcionalidades para hacer un CRUD, pero con directorios.

1.4.1.2 INTERFAZ IFORMFILE

Para facilitar el trabajo con archivos, ASP.NET dispone la interfaz denominada `IFormFile`, la cual se encuentra definida en el espacio de nombre `Microsoft.AspNetCore.Http` y en síntesis permite gestionar la representación de un archivo enviado mediante `HttpRequest` [1].

En el desarrollo de una web API se pueden tener casos en los cuales los usuarios trabajen directamente con archivos, estos pueden ser: imágenes, documentos, entre otros. Además, puede ser requerido que esta información sea validada; para realizar validaciones, se puede emplear la interfaz `IFormFile`, sus propiedades se detallan en la Tabla 1.9 [1]. Así también, se puede utilizar el espacio de nombres `System.ComponentModel.DataAnnotations` para realizar comprobaciones entre los metadatos del archivo y el tipo de validación que defina la clase personalizada derivada de la interfaz `IFormFile` [18].

`IFormFile` define propiedades y métodos que permiten el acceso a los metadatos de los archivos que se envían a través de peticiones HTTP, los metadatos son de gran importancia pues permiten el acceso a las características del archivo como: tipo de archivo, tamaño, nombre, entre otros [1].

Tabla 1.9. Propiedades de `IFormFile`

Propiedad	Funcionalidad
`ContentDisposition`	Contiene la información de *disposition* del archivo proporcionado por el cliente. Esta información incluye el nombre del archivo y cualquier otra información proporcionada en el encabezado HTTP de `ContentDisposition` del archivo.
`ContentType`	Adquiere el encabezado del tipo de contenido sin procesar de un archivo cargado.
`FileName`	Adquiere el nombre del archivo del encabezado de `ContentDisposition`.
`Headers`	Adquiere el diccionario de encabezados de un archivo cargado.
`Length`	Adquiere la longitud que tiene el archivo en bytes.
`Name`	Adquiere el nombre del campo de formulario del encabezado de `ContentDisposition`.

Los métodos de `IFormFile` definidos en la Tabla 1.10 trabajan de manera directa en cuanto al procesamiento de los archivos, de tal forma que se extraen los metadatos mediante las propiedades indicadas de la Tabla 1.9 [1]. Adicionalmente gracias a los métodos de esta interfaz se puede extraer secuencias de bytes de tal manera que el servicio de almacenamiento pueda trabajar con el conjunto de clases establecidas en el

espacio de nombres System.IO para generar la lógica de almacenamiento de archivos [21]. Posteriormente con el uso del espacio de nombre Microsoft.AspNetCore.Hosting se puede acceder al directorio raíz wwwroot para almacenar el archivo.

Finalmente, se realiza la generación de rutas con base en el contexto actual de una solicitud HTTP haciendo uso del espacio de nombre Microsoft.AspNetCore.Http y la combinación de parámetros como la carpeta contenedora, el nombre de usuario y el nombre del archivo. Esto con la finalidad de generar una URL con la ubicación del recurso y cuya cadena de texto será almacenada en la base de datos [16].

Tabla 1.10. Métodos de IFormFile

Métodos	Funcionalidad
CopyTo(Stream)	Copia el contenido del archivo cargado, recibe como parámetro un objeto de tipo Stream[20] que contiene la secuencia de bytes de un archivo.
CopyToAsync(Stream,CancellationToken)	Funciona de la misma manera que CopyToStream con la diferencia que este método se utiliza en programación asíncrona[21].
OpenReadStream	Realiza la apertura del flujo de la solicitud para poder examinar el archivo cargado, devuelve un Stream.

1.4.1.3 AUTENTICACIÓN CON JWT

JWT es una forma compacta y autónoma para transmitir información de forma segura entre partes, normalmente se utiliza para autenticación y autorización. JWT está diseñado para ser fácil de usar y para proporcionar una forma segura de transmitir información sin necesidad de almacenar el estado de la sesión en el servidor [22], [23].

El *token* que se genera consta de tres partes las cuales se describen a continuación [22]:

- **Encabezado**: generalmente consta de dos partes: el tipo de token, que es JWT, y el algoritmo de firma que se utiliza, como HMAC (*Hash-Based Message Authentication Code*), SHA256 (*Secure Hash Algorithm 256-bit*) o RSA (*Rivest-Shamir-Adleman*). La definición de seguridad del *token* está dada en el espacio de nombres Microsoft.IdentityModel.Tokens.

[20] *Stream*: Representa la secuencia de bytes de un archivo transmitido a través de una solicitud HTTP.

[21] Programación asíncrona: Paradigma de programación que permite ejecutar múltiples tareas sin necesidad de esperar que una tarea finalice.

- **Carga útil**: contiene los *claims*, que son declaraciones sobre una entidad normalmente la del usuario y metadatos adicionales. Se define en el espacio de nombres `System.Security.Claims`.
- **Firma**: se crea tomando el encabezado codificado, la carga útil codificada, una llave secreta y el algoritmo especificado en el encabezado. La firma y generación del *token* se realiza con base en las 3 partes indicadas. Se define en el espacio de nombre `System.IdentityModel`.

La estructura de un JWT se puede observar en la Figura 1.5.

Figura 1.5. Estructura de un JSON *Web Token*

La autorización mediante JWT es un proceso que permite que una aplicación pueda validar la identidad de un usuario y asegurarse de que tenga los permisos necesarios para acceder a determinados recursos. Esto se hace mediante la inclusión de *claims* en la carga útil de JWT que indiquen las funciones, los permisos y otros atributos del usuario. La aplicación verificará que los *claims* sean apropiados antes de permitir el acceso a un recurso protegido [23].

En la Figura 1.6 se describe el proceso que realiza un aplicativo que utiliza el método de autorización basado en Token.

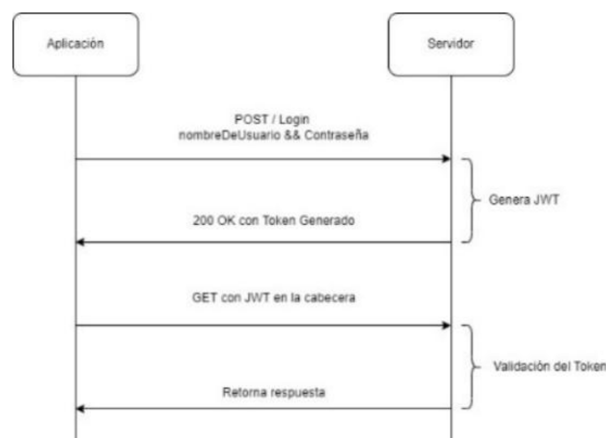

Figura 1.6. Autorización basada en JWT

1.4.1.4 AUTOMAPPER

`AutoMapper` es una librería de C# que se utiliza para realizar una asignación de datos de un objeto en otro. En resumen, se lo define como un elemento que permite asociar un objeto a otro objeto, el cual está basado en convenciones que requieren poca configuración [3].

La funcionalidad de `AutoMapper` se resume en la transformación de un objeto de entrada que genera un nuevo objeto de salida. En la Figura 1.7 se presenta este proceso de asociación entre 2 objetos [3].

Figura 1.7. Proceso de Asociación entre dos objetos

En web API se utilizan los DTO y como su nombre lo indica son objetos que sirven para el transporte de datos, los cuales pueden tener su origen en una o más entidades de información. Es una buena práctica hacer uso de este tipo de objetos, pues las entidades son clases con propiedades que pueden usarse en la generación de la base de datos, pero para los usuarios finales en ciertas ocasiones no se requiere mostrar todos los datos, por lo que el uso de este tipo de objetos solventa este problema. La finalidad de usar estos objetos es entonces, representar los datos que únicamente se quiere que los clientes que consuman la API reciban. Sin embargo, enlazar un objeto DTO con el objeto de la Entidad puede resultar en un uso mayor de número de líneas de código, por lo cual `AutoMapper` ayuda a reducir y simplificar este proceso mediante el uso de una clase que asocia objetos DTO con objetos de Entidades [2], [3].

1.4.1.5 NEWTONSOFTJSON

`NewtonsoftJson` es una biblioteca popular de código abierto para trabajar con datos en formato JSON en aplicaciones ASP.NET. Proporciona un conjunto de

herramientas potentes y fáciles de usar para serializar[22] y deserializar[23] datos en formato JSON, así como para manipular dichos datos [24].

1.4.2 ARQUITECTURA REST

REST (*Representational State Transfer*) es un enfoque arquitectónico para fines de comunicación que se utiliza a menudo en el desarrollo de servicios web. Es un modelo cliente-servidor sin estado[24]. Los servicios web que se definen basados en el enfoque REST son conocidos como servicios web RESTful. Cuando un cliente realiza una solicitud a través de la API RESTful, transfiere la representación del estado de los recursos al servidor. Esta información se puede transferir en varios formatos a través de HTTP como JSON, HTML, texto sin formato, entre otros; sin embargo, JSON es el lenguaje más común debido a su fácil lectura por parte de máquinas y humanos [25], [26].

Con REST, los servicios web transportan datos directamente haciendo uso del protocolo HTTP, empleando de forma directa sus métodos GET, POST, PUT, DELETE, entre otros. En la Figura 1.8 se describe el proceso de envío de información con un servicio basado en la arquitectura REST [25], [26].

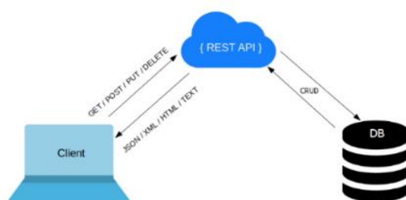

Figura 1.8. Modelo de un servicio con REST

1.4.3 MODELO DE CAPAS

Es una forma de estructurar un sistema de software dividiéndolo en múltiples capas que tienen responsabilidades específicas.

---

[22] Serializar: Es el proceso de convertir datos a un formato que se puede almacenar o transmitir.

[23] Deserializar: Convierte el flujo de bytes nuevamente en un objeto.

[24] Cliente-Servidor sin estado: Permite mantener la eficiencia y escalabilidad de recursos ya que no se requiere almacenar y administrar información del estado de cada cliente.

Con ASP.NET Web API, una arquitectura basada en capas, tiene la siguiente división [27]:

- **Capa de presentación:** es la capa responsable de manejar las interacciones del usuario y mostrar datos al usuario. Por lo general, consta de componentes de la interfaz de usuario como páginas web y formularios.
- **Capa lógica:** es la capa responsable de implementar la lógica de la aplicación, que incluye el procesamiento de solicitudes de la capa de presentación, la realización de cálculos y la coordinación del acceso a los datos.
- **Capa de acceso a datos:** es la capa responsable de interactuar con el sistema de almacenamiento de datos y recuperar o actualizar los datos según sea necesario. Esta capa generalmente se implementa mediante una tecnología de acceso a datos como *Entity Framework*.
- **Capa del modelo de datos:** representa los datos que se intercambian entre el cliente y el servidor. Las clases del modelo de datos definen la estructura de los datos y son utilizadas tanto por la capa de lógica de negocios como por la capa de acceso a datos.

En la Figura 1.9 se observa un diagrama que muestra la interacción entre capas que existen al implementar una web API con ASP.NET.

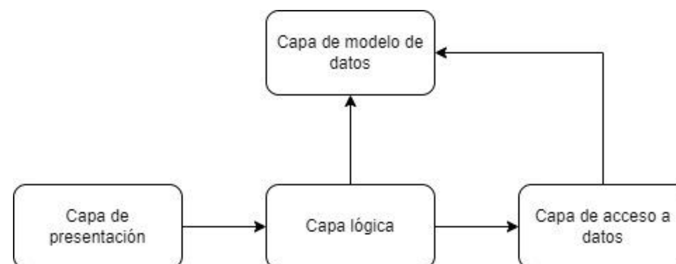

Figura 1.9. Modelo de arquitectura basada en capas

1.4.4 SWAGGER API

Swagger es un *framework* que simplifica el proceso de diseño, construcción, documentación y consumo de servicios web RESTful. Es una herramienta de código abierto que utiliza la especificación OpenAPI[25] que permite a los desarrolladores definir la estructura del API, incluidos los *endpoints*, los parámetros, los tipos de

[25] OpenAPI: Es una especificación que se utiliza para describir la estructura y las operaciones de las API RESTful.

solicitud y respuesta entre otros. Proporciona una interfaz fácil de usar que permite probar los *endpoints* de la API y ver las cargas útiles de solicitudes y respuestas.

Esto lo convierte en una herramienta útil para la realización de pruebas antes de que otras aplicaciones consuman la API.

En la Figura 1.10 se muestra la interfaz gráfica de Swagger [28].

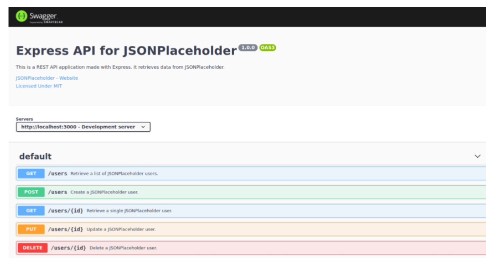

Figura 1.10. Interfaz gráfica de Swagger

1.4.5 METODOLOGÍA KANBAN

Kanban es una metodología que hace uso de técnicas de gestión de proyectos para dirigirlas de manera visual. Esta metodología consiste en trazar un mapa, o una especie de plano que divide y delega fases de desarrollo de manera física en un tablero Kanban como el que se puede visualizarse en la Figura 1.11. Las tareas se anotan en tarjetas y el progreso en columnas, una tras otra. A medida que se completa una tarea, se realiza la transición de la tarjeta respectiva a otra columna, según el estado en el que se encuentren ya sea en inicio, desarrollo, revisión o finalización [29].

Kanban es un proceso de desarrollo continuo que no se detiene después de crear solo una parte, sino el producto completo. Por lo tanto, ayuda a las personas a reducir el tiempo y los costos de desarrollo. Es muy utilizado por equipos de diversas industrias, ya que delega tareas visualmente y facilita la comprensión de lo que sigue en el proceso, lo que reduce el alcance del error [30].

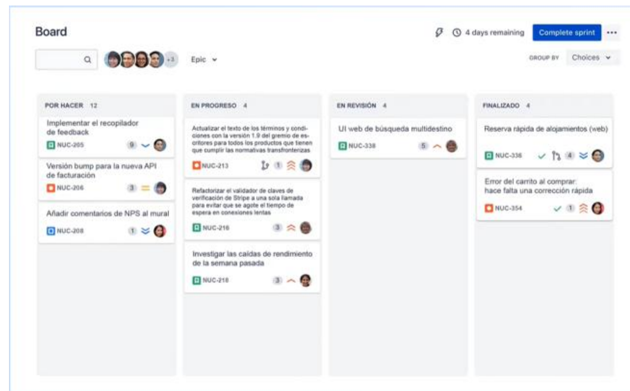

Figura 1.11. Representación de un tablero Kanban

2 METODOLOGÍA

En este capítulo se describe el proceso de desarrollo del subsistema de clasificación usando la metodología ágil Kanban. Con esta metodología se definieron las tareas que se desarrollarán a lo largo del proceso de planificación e implementación de la web API; estas tareas se colocaron en un tablero el cual se lo dividió en fases que definen las etapas para el desarrollo de la web API.

Este capítulo consta de dos secciones: la primera de diseño en donde se plantea la lógica para realizar la implementación de la web API, esta consta de la organización de actividades a partir del tablero Kanban, el análisis de requerimientos que permite establecer la arquitectura del subsistema, permitiendo generar los respectivos diagramas de clases y de actividades; y la segunda de implementación donde se detalla el desarrollo del prototipo haciendo uso del *framework* ASP.NET web API con base a lo establecido en la sección de diseño.

Finalmente, se llevó a cabo pruebas que sirvieron para comprobar el correcto funcionamiento de la integración de componentes, para lo cual se implementaron funcionalidades que permitan realizar la carga y etiquetas de imágenes. Además, de una funcionalidad que permita realizar la búsqueda de la información del contenido cargado a partir del parámetro de etiquetas.

2.1 DISEÑO

En esta sección se presenta un resumen de las especificaciones a tomar en cuenta para el desarrollo del subsistema de clasificación considerando el análisis de requerimientos funcionales y no funcionales, determinación de la arquitectura y su diseño final.

2.1.1 TABLERO KANBAN

Con el uso de la herramienta gratuita Lucidchart, se realiza la organización de actividades en el Tablero Kanban el cual se encuentra dividido en 4 fases: inicio, desarrollo, revisión y completado. El tablero desarrollado se puede visualizar en la Figura 2.1, el cual indica las tareas y su respectiva clasificación en las fases según su estado de cumplimiento.

Figura 2.1. Tablero Kanban

En el ANEXO I se incluye la URL de acceso para la visualización del tablero Kanban completo y su respectivo detalle de tareas.

2.1.2 REQUERIMIENTOS

La identificación de los requerimientos para el desarrollo de software es fundamental ya que permite evaluar bajo qué parámetros se debe realizar las diferentes implementaciones de tal manera que al concluir el proyecto se pueda evaluar el éxito de un sistema. Estos se obtuvieron a partir de la serie de reuniones realizadas con el instructor. Generalmente se dividen en requerimientos funcionales y requerimientos no funcionales.

Requerimientos funcionales: Los requerimientos funcionales son definidos para establecer las características y funcionalidad del subsistema de clasificación; estos se detallan en la Tabla 2.1.

Tabla 2.1. Requerimientos funcionales.

Id	Requerimiento	Características
RF1	Crear Usuarios	• Se recibe como parámetro los datos del usuario (nombre, correo, nombre de usuario y la contraseña). • Se debe considerar que el nombre de usuario y el correo deben ser únicos en cada registro de un nuevo usuario.

RF2	Iniciar sesión	• El *login*[26] de usuario se realizará con los parámetros de nombre de usuario y contraseña. • Si el *login* es correcto devuelve un *token* el cual sirve para la autenticación.
RF3	Acceder a los *endpoints*	• Para acceder a los *endpoints* el usuario debe autenticarse con el token generado al momento de hacer el proceso de *login*.
RF4	Cargar etiquetas con imágenes	• Una o varias imágenes pueden tener una o varias etiquetas. • Las etiquetas son una propiedad de las imágenes que servirán para realizar un filtrado y obtención de información de las imágenes cargadas en la API. • Las imágenes cargadas por el usuario deben tener extensión .JPG, .PNG o .GIF; y tendrán un peso máximo de 4 MB. • El conjunto de las imágenes debe ser enviado al menos con una etiqueta. • Las imágenes cargadas tendrán un GUID[27] (*Globally Unique Identifier*) adicional al nombre, esto con la finalidad de tener archivos únicos en cada registro.
RF5	Generar directorios por usuario	• Una vez realizada la carga de archivos se debe generar un directorio con el nombre de usuario donde se estará guardando el archivo de la imagen. • Se enviará a la base de datos la ruta en donde se ubica la imagen para su posterior recuperación.
RF6	Recuperar información usando filtros de etiquetas	• Se debe recuperar las rutas de las imágenes que coincidan con las etiquetas establecidas. • Se debe mostrar la información con el detalle de las etiquetas asociadas a la o las imágenes que se encuentran en el servidor.

Requerimientos no funcionales: Los requerimientos no funcionales definidos para establecer las características generales del subsistema y el comportamiento se detallan en la Tabla 2.2.

Tabla 2.2. Requerimientos no funcionales

Id	Requerimiento	Características
RNF1	Implementación del subsistema	• Se utiliza Visual Studio[28] • Se utiliza como lenguaje de programación C#. • Se utiliza ASP.NET web API. • Se utiliza la arquitectura cliente-servidor. • Se utilizan los paquetes Nuget adicionales (NewtonSoftJson, AutoMapper, JWTBearer, SwashBucle, Entity Framework SQL Server, Entity Framework Tools).

[26] *login*: Es un mecanismo de seguridad para tener un control de acceso individual en un sistema.

[27] GUID (*Globally Unique Identifier*): Es una cadena de caracteres utilizada para establecer una identificación única de elementos dentro de un sistema.

[28] Visual Studio: Entorno de desarrollo integrado que sirve para realizar aplicaciones compatibles con Windows, MAC entre otros.

RNF2	Usabilidad del subsistema	• El prototipo no cuenta con una interfaz de usuario; para realizar un *frontend*[29] se podría usar tecnologías como *Angular*[30] o *React*[31] [31]. • Se prueba la funcionalidad de los *endpoints* haciendo uso de Swagger API.

Arquitectura del subsistema

Para el subsistema de clasificación se utiliza `AutoMapper` cuya funcionalidad es la asociación de objetos, esto permite separar las preocupaciones de la transferencia de datos y la persistencia, lo que hace que sea factible el uso de una clase que contenga la información de configuración de la base de datos directamente en los controladores.

En la Figura 2.2 se presenta el diagrama de paquetes que indica la arquitectura del subsistema de clasificación, con la cual el código será implementado haciendo uso de Visual Studio*.*

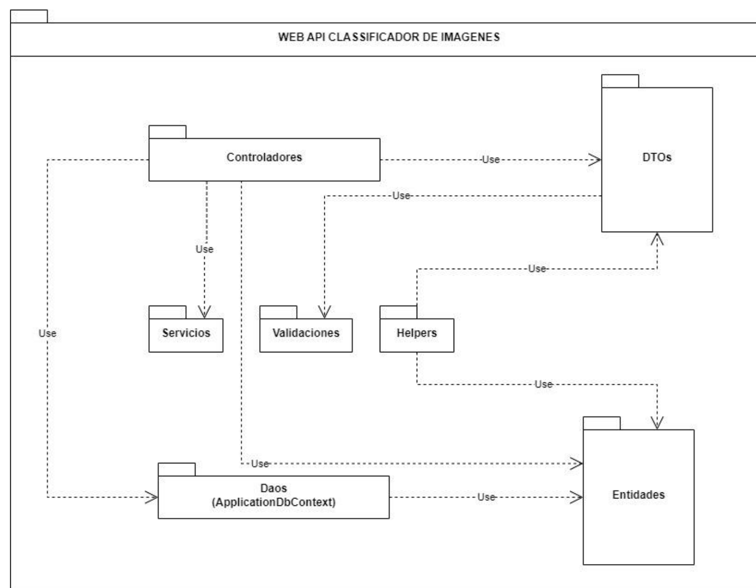

Figura 2.2. Diagrama de paquetes de estructura de la API

En la arquitectura que se muestra en la Figura 2.2 los Controladores son responsables de manejar los *endpoints* de la API y devolver los datos en formato JSON en función de

[29] *frontend:* Es la parte encarga del diseño y la visualización de las páginas web.

[30] Angular: Framework que se basa en *javascript* para el desarrollo web.

[31] React: Librería basada en *javascript* sirve para el desarrollo de componentes e interfaces de usuario.

la información obtenida de los objetos DTO. Estos DTO sirven como capa del modelo de datos y definen las propiedades necesarias para una clase. Los DTO se asocian con Entidades mediante el uso de `AutoMapper`, que maneja el proceso de conversión de objetos. Un caso de uso de esta funcionalidad es el proceso de carga de archivos, donde se recibe y procesa una lista de objetos `IFormFile` para generar una URL para almacenar el archivo. Esta URL luego se almacena en la base de datos. Para realizar la operación de almacenamiento y guardado de la imagen, el controlador hace uso de un servicio de almacenamiento. Esta es la lógica que se sigue para guardar la imagen y generar la URL.

Por otra parte, en los objetos DTO se hace uso de validaciones, para comprobar, por ejemplo, el tamaño de la imagen y el tipo de extensión, de tal manera que se puede establecer restricciones según lo establecido en la Tabla 2.1.

Finalmente, la capa de acceso a datos está formada por una clase `ApplicationDbContext`, la cual contiene la información de la configuración de la base de datos y hace uso de las entidades para la generación de esta. Con el uso de `ApplicationDbContext` el modelo será capaz de realizar las operaciones de búsqueda y guardado de información en la base de datos.

2.1.3 DISEÑO UML (UNIFIED MODELING LANGUAGE)

El diseño basado en diagramas permite tener una representación gráfica del subsistema de clasificación, a partir de los gráficos se establecen las relaciones que tienen las entidades establecidas según los requerimientos y el flujo de actividades que debe tener el subsistema para realizar las peticiones que el usuario requiera haciendo uso de los *endpoints* de la web API.

Diagrama de clases: Inicialmente se trabaja sobre cuatro clases las cuales son las entidades principales (`Usuarios`, `Imágenes`, `Etiquetas` y `EtiquetaImagen`) y se presentan en la Figura 2.3. Estas clases contienen las propiedades principales con las que se generará la base de datos haciendo uso de Entity Framework.

En la Figura 2.3 se observa únicamente el esquema relacional entre las clases que conforman el paquete de entidades. Sin embargo, para el desarrollo completo de la web API hay que considerar que se trabaja con múltiples paquetes cuyo contenido son varias clases que tendrán la lógica para el funcionamiento de los *endpoints* y que realizan las

operaciones necesarias para presentar correctamente los resultados que se requiera en formato JSON.

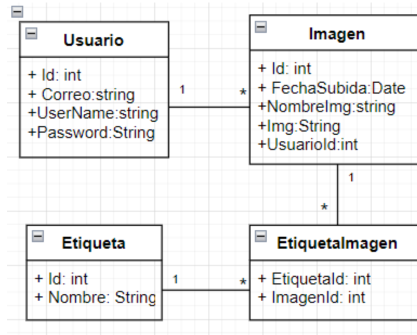

Figura 2.3. Diagrama de clases para gestión de las entidades

En la Tabla 2.3 se detalla un resumen de los paquetes y sus características.

Tabla 2.3. Resumen de elementos del diagrama de clases

Elemento	**Característica**
Clase `AplicationDbContext`	Contiene las propiedades y métodos para la generación de la base de datos haciendo uso de Entity Framework. Permite la interacción con la base a partir de SQL Query y LINQ *(Language Integrated Query)* los cuales sirven para realizar consultas y manipular la información almacenada en la base de datos.
Paquete de Entidades	Contiene el esquema de clases relacionado con las propiedades necesarias para realizar el almacenamiento de información en la base de datos.
Paquete Helpers	A través de la clase `AutoMapper` realiza el proceso de asignación de datos entre objetos DTO y objetos del Paquete de Entidades.
Paquete DTO	Contiene las clases que ayudan a simplificar la estructura de los datos y minimiza la cantidad de datos que deben transferirse entre capas.
Paquete de Validaciones	Contiene la clase con funcionalidades que permiten la verificación de la extensión del archivo y su peso. La emplea el DTO que recibe la imagen.
Paquete de Servicios	Contiene las clases para realizar el proceso de almacenamiento de la imagen y la obtención de los *claims* del usuario que se encuentra autenticado en la web API.
Paquete de Controladores	Contienen los controladores que procesan los diferentes métodos HTTP que son presentados como *endpoints* para el usuario final, usa principalmente el paquete DTO, el paquete de Servicios y la clase `AplicationDbContext`.

En la Figura 2.4 se muestra el diagrama de clases del subsistema.

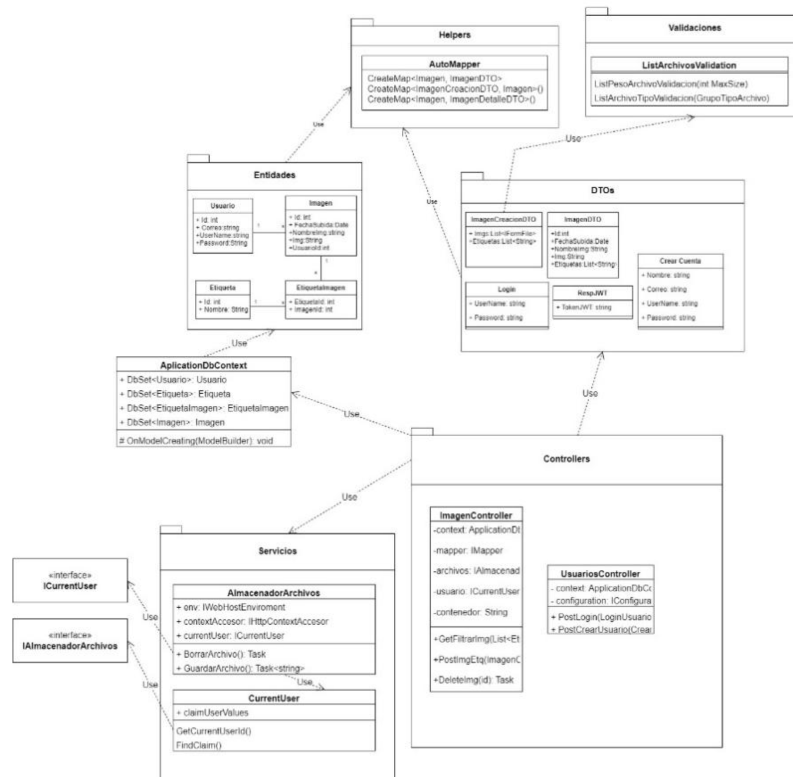

Figura 2.4. Diagrama de clases de la web API

Diagrama de actividades: En la Figura 2.5 se observa el diagrama de actividades que presenta el proceso que la web API realiza al recibir el conjunto de imágenes y etiquetas. Es posible observar que la API debe realizar una validación de información, se valida que el peso de las imágenes enviadas sea máximo de hasta 4 MB, que el tipo de archivo tenga la extensión .JPG, .PNG o .GIF y que se haya definido al menos una etiqueta. De igual forma se indica que el conjunto de imágenes y etiquetas trabajan por separado.

Para las imágenes se utiliza un servicio de almacenamiento de archivos, el cual tendrá los métodos para guardar y eliminar imágenes. La función para guardar las imágenes tendrá como parámetros de entrada los metadatos del archivo y como resultado generará las rutas que serán almacenadas en la base de datos. Por otra parte, para las etiquetas se realiza un proceso de verificación para comprobar que los registros no sean repetidos: si la etiqueta ingresada coincide con algún registro que ya existe en la base se toma el identificador de la etiqueta existente; y, si la etiqueta no coincide con algún registro se procede a generar un nuevo registro con nombre e identificador. Estos identificadores se agregan al listado auxiliar que será utilizado para asociar las imágenes y etiquetas. Finalmente, una vez realizado el guardado y la generación de los registros tanto para la tabla que contiene imágenes como la que contiene etiquetas, se realiza la

asignación de los identificadores respectivos de imágenes y etiquetas en la tabla `EtiquetaImagen`; con el uso de los registros que se almacenen en la web API, se pueden realizar operaciones con filtros, dichas operaciones tendrán como parámetros las etiquetas para la búsqueda y recuperación de información de las imágenes.

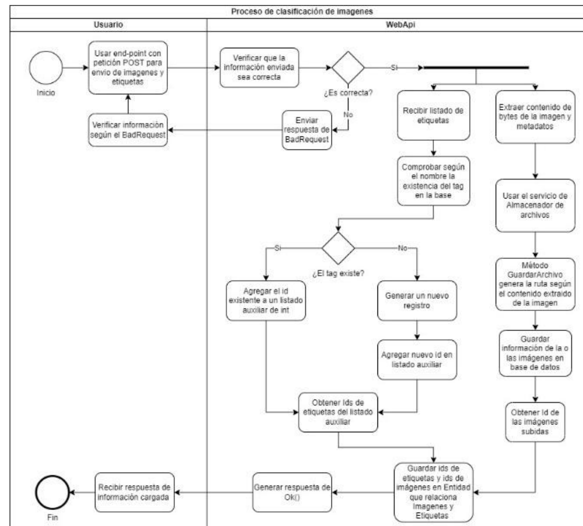

Figura 2.5. Diagrama de actividades para el proceso de clasificación de imágenes

El proceso de filtrado de imágenes se muestra en el diagrama de actividades de la Figura 2.6.

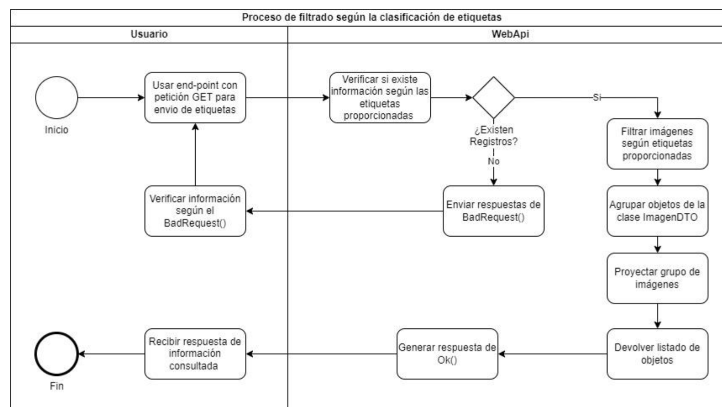

Figura 2.6. Diagrama de actividades para el proceso de filtrado según las etiquetas

En la Figura 2.6 se aprecia que el usuario tiene la libertad de ingresar un listado de etiquetas, a partir de esta información la web API tiene como tarea realizar un proceso de filtrado; si se han encontrado registros de imágenes que correspondan con las etiquetas, se realiza una agrupación de un objeto `ImagenDTO` que tendrá como

parámetros la información de la imagen y un listado de etiquetas. Como resultado se devuelve la información del objeto DTO con base en lo establecido en la búsqueda.

2.2 IMPLEMENTACIÓN

En esta sección se presenta un resumen del desarrollo del subsistema de clasificación.

Para el desarrollo de la web API se utiliza Visual Studio Community 2022, herramienta a través de la cual se realizó la instalación de los componentes necesarios y paquetes Nuget. En particular se utiliza la versión 6 de .NET Framework. Este proceso se lo presenta con mayor detalle en el Anexo II.

2.2.1 CONFIGURACIÓN DE LA CLASE STARTUP

La clase `Startup` se la puede generar para mantener una separación entre los aspectos de ejecución que se darán en la clase `Program` y la configuración de servicios y *middlewares*[32] que estarán en la clase `Startup`.

Como se indica en la Figura 2.7 la clase `Startup` es pública y tiene un constructor que tiene como parámetro un objeto de tipo `IConfiguration` (de la línea 3 hasta la línea 6), que contiene un conjunto de propiedades de configuración; este constructor será utilizado en la clase `Program` para iniciar un objeto de la clase `Startup`. Esta clase cuenta adicionalmente con 2 métodos importantes: el primer método `ConfigureServices` (de la línea 10 hasta la línea 15), sirve para realizar las configuraciones de servicios que requiera la web API; y, el segundo método `Configure` (de la línea 18 hasta la línea 33) se utiliza para configurar la canalización de la solicitud HTTP. Por ejemplo, para Swagger; en el primer método se configura el servicio, es decir se habilita Swagger (línea 14) para que sea utilizado en el proyecto; mientras que en el segundo método se habilita el puente entre Swagger y la web API (de la línea 20 hasta la línea 25) para el paso de solicitudes HTTP. Cabe aclarar que el uso del middleware de Swagger no se limita únicamente al entorno de desarrollo ya que su funcionalidad es principalmente documentar y probar las API, por lo cual se puede utilizar tanto en ambientes de desarrollo como en ambientes de producción.

---

[32] *middleware*: Componente de software que actúa como mediador entre diferentes aplicaciones y sistemas.

```
1 public class Startup
2 {
3     public Startup(IConfiguration configuration)
4     {
5         Configuration = configuration;
6     }
7     public IConfiguration Configuration { get; }
8     //Este método es llamado por el tiempo de ejecución.
9     //Use este método para agregar servicios al contenedor
10    public void ConfigureServices(IServiceCollection services)
11    {
12        services.AddControllers();
13        services.AddEndpointsApiExplorer();
14        services.AddSwaggerGen();
15    }
16    //Este método es llamado por el tiempo de ejecución
17    //Use este método para configurar la canalización de solicitud HTTP
18    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
19    {
20        if (env.IsDevelopment())
21        {
22            app.UseDeveloperExceptionPage();
23            app.UseSwagger();
24            app.UseSwaggerUI();
25        }
26        app.UseHttpsRedirection();
27        app.UseRouting();
28        app.UseAuthorization();
29        app.UseEndpoints(endpoints =>
30        {
31            endpoints.MapControllers();
32        });
33    }
34 }
```

Figura 2.7. Clase `Startup`

En la Figura 2.8 se muestra la clase `Program`, que se encarga de realizar la configuración de servicios y canalizaciones de solicitudes HTTP. Sin embargo, al haber establecido estas actividades como parte de la clase `Startup`, solo se requiere emplear los métodos definidos en esta. Esto provoca que se tenga una mejor estructuración y separación de funcionalidades por lo que únicamente en la clase `Program` se crea un objeto de esta (línea 4) y se inicia la aplicación (línea 6), con base en lo definido en la clase `Startup` (línea 2) mediante sus métodos (líneas 3 y 5).

```
1 var builder = WebApplication.CreateBuilder(args);
2 var startup = new Startup(builder.Configuration);
3 startup.ConfigureServices(builder.Services);
4 var app = builder.Build();
5 startup.Configure(app, app.Environment);
6 app.Run();
```

Figura 2.8. Clase `Program` modificada

2.2.2 IMPLEMENTACIÓN DE ENTIDADES

Las entidades son las clases principales de la web API; en estas se definen las propiedades de los datos cuya información será almacenada en una base de datos. En la Figura 2.9 se muestra la codificación de la entidad `Imagen`.

```
1 public class Imagen
2 {
3     public int Id { get; set; }
4     public DateTime FechaSubida { get; set; }
5     public string NombreImagen { get; set; }
6     public string Img { get; set; }
7     public int? UsuarioId { get; set; }
8     public Usuario Usuario { get; set; }
9     public List<EtiquetaImagen> EtiquetaImagen { get; set;}
10 }
```

Figura 2.9. Codificación de la entidad `Imagen`

En el caso de la Figura 2.9 se definen las propiedades de la imagen (línea 3 hasta la línea 6). Por otra parte, se considera que la relación entre la entidad `Usuario` y la entidad `Imagen` es de uno a muchos por lo que la propiedad `UsuarioId` (línea 7) es la clave foránea de la entidad `Usuario`; también se define una propiedad de navegación (línea 8) esta permite la navegación de extremo a extremo de los procesos de asociación entre entidades. Finalmente, al existir una relación muchos a muchos entre la entidad `Imagen` y la entidad `Etiqueta`, se debe generar la entidad `EtiquetaImagen` la cual relacione las claves primarias de las dos entidades, para esto en las entidades relacionadas se debe establecer una propiedad de tipo `List` conformada por objetos `EtiquetaImagen` (línea 9), lo cual permite que al generarse la base de datos se considere la existencia de dicha relación.

Con la misma lógica y se implementaron el resto de las entidades que conforman la API.

2.2.3 IMPLEMENTACIÓN DE LA CAPA DE ACCESO A DATOS

Para realizar la implementación de la capa de acceso a datos es necesario realizar la instalación de 2 paquetes Nuget:

- `Microsoft.EntityFrameworkCore.Tools`.
- `Microsoft.EntityFrameworkCore.SqlServer`.

La clase `ApplicationDbContext` se encarga de generar la base de datos haciendo uso de las entidades. Su uso, en los controladores, permite la generación de operaciones que involucren a la base de datos, como búsqueda e inserción de datos. En la Figura 2.10 se observa el código implementado.

```
1 public class ApplicationDbContext : DbContext
2 {
3     public ApplicationDbContext(DbContextOptions options) : base(options)
4     {
5     }
6     protected override void OnModelCreating(ModelBuilder modelBuilder)
7     {
8         modelBuilder.Entity<EtiquetaImagen>()
9             .HasKey(x => new { x.EtiquetaId, x.ImagenId });
10        base.OnModelCreating(modelBuilder);
11    }
12    public DbSet<Etiqueta> Etiquetas { get; set; }
13    public DbSet<Imagen> Imagenes { get; set; }
14    public DbSet<EtiquetaImagen> EtiquetaImagen { get; set; }
15    public DbSet<Usuario> Usuarios { get; set; }
16 }
```

Figura 2.10. Clase `ApplicationDbContext`

Como se muestra en la Figura 2.10 la clase publica `ApplicationDbContext` hereda de `DbContext` (línea 1); esto permite generar un constructor que toma un objeto `DbContextOptions` (línea 3 hasta la línea 5) como argumento, a este constructor se le puede pasar múltiples configuraciones de *Entity Framework*, entre estas, la cadena de conexión de la base de datos. Posteriormente se define el método heredado de la clase base `OnModelCreating` (línea 6 hasta la línea 11), el cual permite realizar las configuraciones de relaciones del modelo de base de datos, y se utiliza para establecer la clave principal para que la entidad `EtiquetaImagen` sea una clave compuesta que consta de las propiedades conformadas por los identificadores de etiqueta e imagen (`EtiquetaId` e `ImagenId` respectivamente). Finalmente, se definen las propiedades de las clases que representan la colección de objetos que corresponden a cada entidad respectivamente (línea 12 hasta la línea 15).

Configuración de la cadena de conexión

Para realizar la configuración de la cadena de conexión de la base de datos se utiliza el proveedor de configuraciones `appsettings.Development.json`. En la Figura 2.11 se representa la configuración de la cadena de conexión de la base de datos, en la cual se especifican detalles como: el proveedor de la base de datos, en este caso Microsoft SQL Server; el nombre de la base de datos, en este caso `DbClasificadorImg`; el alojamiento, que en este caso es local, usando la instancia `mssqllocaldb`; la seguridad, que en este caso es la seguridad integrada con la autenticación de *Windows*.

```
1  "ConnectionStrings": {
2    "DefaultConnection": "Data Source=(localdb)\\mssqllocaldb;Initial Catalog=DbClasificadorImg;Integrated Security=True"
3  }
```

Figura 2.11. Cadena de conexión de la base de datos

En la Figura 2.12 se representa la configuración del servicio de base de datos en la clase `Startup` mediante el método `ConfigureServices`, en este se llama al método `AddDbContext` el cual agrega una instancia de la clase `ApplicationDbContext` al contenedor de inyección de dependencia[33] para posteriormente llamar al método `UseSqlServer`, el cual sirve para especificar que se utilizará Microsoft SQL Server como proveedor de base de datos y cuya cadena de conexión se obtiene del archivo de configuración `appsettings.Development.json`.

```
1 services.AddDbContext<ApplicationDbContext>(
2 options =>
3 options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

Figura 2.12. Configuración del servicio de bases de datos

Con estas configuraciones y con la clase `ApplicationDbContext`, se puede utilizar la consola de administración de paquetes Nuget para generar las clases que representan una migración[34] para la base de datos, el comando de Entity Framework utilizado para esta operación es `Add-Migration` y con las clases generadas se utiliza un segundo comando denominado `Update-Database` para generar o actualiza la base de datos tomando en cuenta las modificaciones realizadas en las entidades y cuyos cambios han sido agregados mediante migraciones.

2.2.4 IMPLEMENTACIÓN DE LA CAPA DE MODELO DE DATOS

Para realizar la implementación de la capa de modelo de datos es necesario realizar la instalación del paquete Nuget denominado `AutoMapper`.

[33] contenedor de inyección de dependencia: Patrón de diseño que gestiona las dependencias de objetos en una aplicación de manera centralizada.

[34] migración: Este proceso sirve para crear, agregar o modificar información a la estructura de la base de datos.

En la Figura 2.13 se muestra la clase ImagenCreacionDTO, la cual representa a un DTO que se utiliza para transferir datos entre cliente y servidor, y que consta de dos propiedades: la primera, una lista de objetos de tipo IFormFile (línea 5) que representan archivos que serán enviados a través de una solicitud HTTP y que a través de un atributo personalizado denominado ListArchivoTipoValidacion (línea 3) se puede realizar la validación de que los archivos tangan una extensión definida en GrupoTipoArchivo, y que a través de otro atributo personalizado ListArchivoPesoValidation (línea 4) se puede validar que el peso máximo del archivo sea 4 MB; la segunda propiedad, es una lista de cadenas de texto (línea 6), en la cual cada cadena de la lista representa a una etiqueta.

```
1 public class ImagenCreacionDTO
2 {
3     [ListArchivoTipoValidation(GrupoTipoArchivo.Imagen)]
4     [ListArchivoPesoValidation(4)]
5     public List<IFormFile> Imgs { get; set; }
6     public List<string> Etiquetas { get; set; }
7 };
```

Figura 2.13. DTO generado para la creación de las imágenes

Se debe considerar que en las clases DTO únicamente se especifican propiedades, y que estas clases no deben contener implementaciones adicionales como métodos, ya que únicamente determinan la cantidad necesaria de datos a transferirse por las diversas capas. Con la misma lógica se realiza el desarrollo del resto de DTO que complementan la API.

Proceso de asociación de objetos

Para el funcionamiento de algunos objetos DTO es necesario la existencia de una clase intermedia que establezca las reglas de asociación para realizar la conversión de objetos.

Como se indica en la Figura 2.14 la clase publica AutoMapper hereda de la clase Profile (línea 1), y sirve para configurar las reglas de asociación de diferentes tipos de objetos. En el código se observa que en el constructor (línea 3 hasta la línea 8) se definen las asociaciones entre diferentes tipos de objetos haciendo uso de la propiedad CreateMap. Se dispone de una asociación bidireccional que convierte un objeto

Imagen en un objeto ImagenDTO (línea 5); también se realiza la asociación que convierte un objeto ImagenCreacionDTO en un objeto Imagen (línea 6 y 7) en el que se ignora el campo Img (que corresponde a la ruta de ubicación del archivo) de la entidad Imagen, esto se debe a que la propiedad Imgs de ImagenCreacionDTO se usa para almacenar una lista de archivos de imagen IFormFile, mientras que la propiedad Img de Imagen se usa para almacenar la URL de la imagen, por lo que no es necesario asociar el contenido de Imgs de ImagenCreacionDTO con el contenido de Img de Imagen.

Se aclara que las clases DTO generadas también pueden contener propiedades que únicamente los controladores requieren para realizar operaciones; por tanto, no necesariamente tienen reglas de asociación definidas.

```
1 public class AutoMapper : Profile
2 {
3     public AutoMapper()
4     {
5         CreateMap<Imagen, ImagenDTO>().ReverseMap();
6         CreateMap<ImagenCreacionDTO, Imagen>()
7             .ForMember(x => x.Img, options => options.Ignore());
8     }
9 }
```

Figura 2.14. Asociación de objetos del tipo Entidad y del tipo DTO

2.2.5 IMPLEMENTACIÓN DE LA CAPA LÓGICA

En esta capa se incluyen los servicios que permiten que la web API realice el proceso de almacenamiento de archivos, generación de rutas de almacenamiento y obtención del contexto del usuario actual que está realizando la solicitud HTTP; por otra parte, consta de validaciones personalizadas que permiten verificar que los archivos que se están transmitiendo a través de una solicitud HTTP estén acorde a las restricciones establecidas en los requerimientos funcionales.

Servicio de almacenamiento

En la interfaz IAlmacenadorArchivos se definen las funciones para guardar y eliminar archivos; en la Figura 2.15 se presenta el método GuardarArchivo (línea 3), el cual toma cinco parámetros: una matriz de bytes que representa el contenido del archivo, una cadena que contiene la extensión del archivo, una cadena que contiene la

carpeta en la que se almacenará el archivo, una cadena que indica el tipo de contenido del archivo y una cadena que representa el nombre del archivo que se va a guardar. Este método devuelve una tarea que al completarse proporciona una cadena que indica la ruta en la que se almacenó el archivo; también se incluye la definición del método `BorrarArchivo` (línea 4), el cual tiene dos parámetros: una cadena que indica la ruta del archivo y una cadena que indica el contenedor dónde se almacenó el archivo, y devuelve una tarea que al completarse indica que el archivo ha sido eliminado.

```
1 public interface IAlmacenadorArchivos
2 {
3     Task<string> GuardarArchivo(byte[] contenido, string extension, string contenedor, string contentType, string nombreArchivosub);
4     Task BorrarArchivo(string ruta, string contenedor);
5 }
```

Figura 2.15. Definición de la interfaz `IAlmacenadorArchivos`

En la Figura 2.16 se muestra la clase `AlmacenadorArchivosLocal` que implementa la interfaz `IAlmacenadorArchivos` (línea 1), utiliza una inyección de dependencias para inyectar instancias (de la línea 7 a la línea 12) de `IWebHostEnviroment`, que permite proporcionar la ruta de la carpeta raíz `wwwroot`; de `IHttpContextAccesor`, que permite recuperar información del estado actual de la solicitud HTTP; y de `ICurrentUser`, utilizada para proporcionar datos del usuario en el contexto actual de la solicitud HTTP. Finalmente, se implementan las dos funciones (de la línea 14 a la línea 22).

```
1  public class AlmacenadorArchivosLocal : IAlmacenadorArchivos
2  {
3    private readonly IWebHostEnvironment env;
4    private readonly IHttpContextAccessor httpContextAccessor;
5    private readonly ICurrentUser currentUser;
6
7    public AlmacenadorArchivosLocal(IWebHostEnvironment env, IHttpContextAccessor httpContextAccessor, ICurrentUser currentUser)
8    {
9      this.env = env;
10     this.httpContextAccessor = httpContextAccessor;
11     this.currentUser = currentUser;
12   }
13
14   public Task BorrarArchivo(string ruta, string contenedor)
15   {
16     //Logica a implementar
17   }
18   public async Task<string> GuardarArchivo(byte[] contenido, string extensión, string contenedor, string contentType, string nombreArchivoOriginal)
19   {
20     //Logica a implementar
21   }
22 }
```

Figura 2.16. Clase `AlmacenadorArchivosLocal`

En la Figura 2.17 se presenta el código del método asíncrono `GuardarArchivo` (línea 1), que tiene por finalidad guardar el archivo en el servidor y generar la ruta de acceso al mismo; en esta función se definen variables, de la línea 3 hasta la línea 5, para: almacenar el nombre de archivo que debe ser único, por lo cual se agrega un GUID al nombre original del archivo; para almacenar la ruta de la carpeta a partir de la ruta de la carpeta raíz, el nombre de la carpeta contenedora y el nombre de usuario; y, para almacenar el nombre completo del recurso, es decir su ubicación y nombre único. De manera consecuente se crea el directorio si no existe (línea 6) con base en el nombre único y se guarda el archivo utilizando el método `WriteAllBytesAsync` de la clase `File` del espacio de nombres `System.IO` (línea 7) que recibe como parámetros la ruta generada de la carpeta contendora y el contenido en bytes del archivo.

Finalmente, se genera la URL completa del archivo guardado (línea 9) al combinar el esquema de la solicitud actual, el nombre del equipo (línea 8) y las partes de la ruta del archivo que son visibles y accesibles desde la web, se reemplaza las barras diagonales inversas `\\` por la barra diagonal `/` debido a sintaxis, ya que las barras invertidas se utilizan para la separación de directorios mientras que las barras diagonales son utilizadas como separadores en direcciones URL. El método, al finalizar la tarea, devuelve la URL del archivo guardado (línea 10).

```
1  public async Task<string> GuardarArchivo(byte[] contenido, string extensión, string contenedor, string contentType, string nombreArchivoOriginal)
2  {
3      var nombreArchivoUnico = $"{nombreArchivoOriginal}_{Guid.NewGuid()}{extensión}";
4      var carpeta = Path.Combine(env.WebRootPath, contenedor, currentUser.UserName);
5      var rutaArchivo = Path.Combine(carpeta, nombreArchivoUnico);
6      Directory.CreateDirectory(carpeta);
7      await File.WriteAllBytesAsync(rutaArchivo, contenido);
8      var urlActual = $"{httpContextAccessor.HttpContext.Request.Scheme}://{httpContextAccessor.HttpContext.Request.Host}";
9      var urlBD = Path.Combine(urlActual, contenedor, currentUser.UserName, nombreArchivoUnico).Replace("\\", "/");
10     return urlBD;
11 }
```

Figura 2.17. Función `GuardarArchivo`

Por motivos de espacio no se expone el método para la eliminación del archivo, pero su código se presenta en el Anexo III.

Servicio de CurrentUser

La interfaz `ICurrentUser` define únicamente propiedades, ya que está destinada a representar la información del usuario, como su estado de autenticación, su identificador de usuario, su correo electrónico, su nombre y su nombre de usuario.

Al definir solo propiedades, se establece un contrato para que las clases que implementan la interfaz proporcionen la información necesaria sobre el usuario actual. Esto permite la flexibilidad en la forma en que se almacena y recupera la información. En la Figura 2.18 se presenta la interfaz `ICurrentUser`.

```
1 public interface ICurrentUser
2 {
3     public bool IsAuthenticated { get; }
4     public int? Id { get; }
5     public string Correo { get; }
6     public string Nombre { get; }
7     public string UserName { get; }
8 }
```

Figura 2.18. Interfaz `ICurrentUser`

En la Figura 2.19 se define la clase `CurrentUser`, la cual implementa la interfaz `ICurrentUser` (línea 1) con sus propiedades (de la línea 5 hasta la línea 9). Esta clase proporciona un constructor (de la línea 11 hasta la línea 14) que toma un objeto `IHttpContextAccessor` el cual proporciona acceso al contexto actual de la solicitud HTTP.

Por otra parte, proporciona 3 métodos: `GetCurrentUserId` (de la línea 16 hasta la línea 23), que analiza el valor del *claim* para el identificador del usuario contenido en la propiedad `ClaimTypes.NameIdentifier`; `FindClaim` (de la línea 25 hasta la línea 30), que toma un `claimType` como argumento y devuelve el primer *claim* con el tipo especificado de la colección de *claims* del usuario, en el contexto actual de la solicitud HTTP, para lo cual se usa el método `FirstOrDefault` para encontrar la primera afirmación que coincida con el argumento `claimType`. Si se encuentra un *claim*, se devuelve, por el contrario, si el contexto HTTP o la colección de *claims* del usuario es nula, el método retorna `null`; y, `FindClaimValue` (de la línea 32 hasta la línea 35), que se emplea para analizar el valor del *claim* y para lo cual emplea el método `FindClaim` para encontrar el *claim* que coincida con el argumento `claimType`, si se encuentra el *claim* se devuelve el valor correspondiente.

```
1 public class CurrentUser : ICurrentUser
2 {
3     private readonly IHttpContextAccessor contextAccessor;
4
5     public bool IsAuthenticated => Id.HasValue;
6     public int? Id => GetCurrentUserId();
7     public string Correo => FindClaimValue(ClaimTypes.Email);
8     public string Nombre => FindClaimValue(ClaimTypes.Name);
9     public string UserName => FindClaimValue("User");
10
11    public CurrentUser(IHttpContextAccessor contextAccessor)
12    {
13        this.contextAccessor = contextAccessor;
14    }
15
16    private int? GetCurrentUserId()
17    {
18        if (int.TryParse(FindClaimValue(ClaimTypes.NameIdentifier), out var id))
19        {
20            return id;
21        }
22        return null;
23    }
24
25    Claim FindClaim(string claimType)
26    {
27        return contextAccessor.HttpContext?.User.Claims
28                                           .FirstOrDefault(c =>
29                                                  c.Type == claimType);
30    }
31
32    string FindClaimValue(string claimType)
33    {
34        return FindClaim(claimType)?.Value;
35    }
36 }
```

Figura 2.19. Implementación del servicio de `CurrentUser`

Validaciones

Como parte de las validaciones se tiene la clase `ListArchivosValidations`, la cual emplea dos clases diseñadas para la validación de archivos: `ListArchivoPesoValidation` y `ListArchivoTipoValidation`.

En la Figura 2.20 se observa el código de la clase `ListArchivoPesoValidation`, que se encarga de la validación del tamaño de un archivo, para lo cual realiza la implementación de la clase `ValidationAttribute` (línea 1), estableciendo un constructor que toma el tamaño máximo del archivo en MB y lo convierte en bytes (de la línea 5 hasta la línea 8) e implementando el método `IsValid` (de la línea 10 hasta la línea 34) que permite validar el archivo verificando que el objeto no tenga un valor nulo (de la línea 13 hasta la línea 16), que haya al menos un archivo (de la línea 20 hasta la línea 23) y finalmente que el tamaño de cada archivo no exceda el tamaño máximo (de la línea hasta la línea 32). Si la validación no se cumple se retorna un `ValidationResult` con un mensaje de error; por el contrario, si se cumple se retorna un `ValidationResult.Success` (línea 33) que indica el éxito de la validación.

```
1   public class ListArchivoPesoValidation : ValidationAttribute
2 {
3     private int MaxFileSizeMB { get; set; }
4
5     public ListArchivoPesoValidation(int maxSizeInMB)
6     {
7         MaxFileSizeMB = maxSizeInMB * 1024 * 1024;
8     }
9
10    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
11    {
12
13        if (value == null)
14        {
15            return new ValidationResult($"El objeto que se intenta enviar tiene un valor nulo");
16        }
17
18        var files = value as List<IFormFile>;
19
20        if (files.Count == 0)
21        {
22            return new ValidationResult($"Debe ingresar al menos un archivo");
23        }
24
25        foreach (var formFile in files)
26        {
27            if (formFile.Length > MaxFileSizeMB)
28            {
29                return new ValidationResult($"El peso maximo del archivo no debe ser mayor
30                + "a {MaxFileSizeMB.ToString()[0]} MB", new[] {validationContext.MemberName});
31            }
32        }
33        return ValidationResult.Success;
34    }
35 }
```

Figura 2.20. Clase `ListArchivoPesoValidation`

En la Figura 2.21 se observa el código de la clase `ListArchivoTipoValidation`, la cual se encarga de la verificación de los tipos de extensiones de los archivos y cuya implementación sigue la misma lógica de la clase `ListArchivoPesoValidation`, por lo cual se define el constructor que toma una enumeración de tipo `GrupoTipoArchivo` (de la línea 3 hasta la línea 9) para determinar los tipos de archivos aceptables para este trabajo, los cuales son: `image/jpeg`, `image/png`, e `image/gif` y se definen las especificaciones para el retorno de un indicador de éxito `ValidationResult.Success` (de la línea 11 hasta la línea 19).

```
1  private readonly string[] tiposValidos;
2  //Constructor de la clase con el array de tipos de archivos validos
3  public ListArchivoTipoValidation(GrupoTipoArchivo grupoTipoArchivo)
4  {
5      if (grupoTipoArchivo == GrupoTipoArchivo.Imagen)
6      {
7          tiposValidos = new string[] { "image/jpeg", "image/png", "image/gif" };
8      }
9  }
10 // condicion para el cumplimiento de especificaciones
11 foreach (var formFile in files)
12 {
13     if (!tiposValidos.Contains(formFile.ContentType))
14     {
15         return new ValidationResult(
16             $"El tipo de archivo debe ser uno de los siguientes: JPEG, PNG o GIFT",
17             new[] { validationContext.MemberName });
18     }
19 }
20 //Se retorna un validation success como respuesta
21 return ValidationResult.Success;
```

Figura 2.21. Modificaciones de la clase `ListArchivoTipoValidation`

La clase ImagenCreacionDTO emplea estas validaciones, para determinar si los archivos cargados estén acorde a los requerimientos funcionales.

2.2.6 IMPLEMENTACIÓN DEL CONTROLADOR

Como parte del paquete Controllers, se tienen las clases de los controladores que se implementan en la web API. Los controladores contienen *endpoints*, cada uno con su respectiva lógica para realizar peticiones HTTP y para obtener resultados que se entregan en formato JSON. Para el desarrollo se consideró de manera primordial los controladores principales ImagenController y CuentasController.

ImagenController

En la Figura 2.22 se indica el código de la clase ImagenController, la cual hereda de ControllerBase (línea 3) que es una clase perteneciente al espacio de nombres Microsoft.AspNetCore.Mvc, y que proporciona el soporte básico para el manejo de solicitudes HTTP. La clase ImagenController se decora con [ApiController] (línea 1) que indica que la clase es un controlador y [Route("api/imagenes")] (línea 2) que se utiliza para indicar la ruta para acceder al controlador. El constructor de la clase (de la línea 11 hasta la línea 17) tiene una inyección de las dependencias: ApplicationDbContext, IMapper, IAlmacenadorArchivos y ICurrentUser.

La inyección permite acceder a la base de datos, realizar procesos de asociación de objetos, almacenar archivos y obtener información del usuario en el contexto actual de la solicitud HTTP, según corresponda.

Además, se definen dos métodos de acción: filtrarImg (línea 19), que trabaja con el método HTTP GET y que corresponde al *stub* que simula la funcionalidad de búsqueda de información de imágenes a partir de las etiquetas; y, cargarImgWithTags (línea 25), que trabaja con el método HTTP POST, y que permite recibir las imágenes y etiquetas del usuario que serán cargadas al servidor.

```
1  [ApiController]
2  [Route("api/imagenes")]
3  public class ImagenController : ControllerBase
4  {
5      private readonly ApplicationDbContext context;
6      private readonly IMapper mapper;
7      private readonly IAlmacenadorArchivos archivos;
8      private readonly ICurrentUser usuario;
9      private readonly string contenedor = "Imagenes";
10
11     public ImagenController(ApplicationDbContext context, IMapper mapper, IAlmacenadorArchivos archivos, ICurrentUser usuario)
12     {
13         this.context = context;
14         this.mapper = mapper;
15         this.archivos = archivos;
16         this.usuario = usuario;
17     }
18
19     [HttpGet("filtrarImg")]
20     public async Task<ActionResult<List<ImagenDTO>>> Filtrar([FromQuery] FiltroMultipleDeEtiquetas filtroImagenesEtiqueta)
21     {
22         // Lógica a implementar para la búsqueda de imágenes
23     }
24
25     [HttpPost("cargarImgWithTags")]
26     public async Task<ActionResult<List<ImagenDTO>>> Post([FromForm] ImagenCreacionDTO imagesDto)
27     {
28         // Lógica a implementar para el proceso de carga de imágenes y etiquetas
29     }
30 }
```

Figura 2.22. Estructura del controlador `ImagenController`

En la Figura 2.23 se define la lógica del *endpoint* `filtrarImg`, el cual se decora con `[HttpGet("filtrarImg")]` (línea 1) que indica que la función asíncrona `Filtrar` se asocia a una solicitud HTTP usando el método GET con la URL `imagenes/filtrarImg`, que devuelve una lista de objetos de tipo `ImagenDTO` y que toma como parámetro de consulta un objeto `filtroImagenesEtiqueta` mediante el decorador `[FromQuery]`. Este decorador se usa para especificar que los datos del parámetro se obtienen desde la cadena de consulta de la solicitud HTTP (línea 2). El objeto `filtroImagenesEtiqueta` tiene como propiedad un listado de cadenas de texto que representan las etiquetas con las cuales se realizarán los filtros para la obtención de información de imágenes cargadas en el servidor. Como parte del método se establece una consulta SQL (de la línea 4 hasta la línea 18) para realizar la recuperación de información de imágenes y etiquetas, por lo cual en el objeto `tempQueryable` se almacena la colección de objetos del tipo `Imagen` que contiene los campos de información del archivo que han sido cargados a la API y las etiquetas que representan a la propiedad de la tabla `Etiquetas`. Posteriormente se realiza una validación (de la línea 20 hasta la línea 23) en la cual, si no se agregan etiquetas como parámetro de búsqueda, el *endpoint* devolverá como resultado todas las imágenes que hayan sido cargadas por el usuario. Si con el parámetro definido se encuentra algún resultado (de la línea 24 hasta la línea 38), se procede a realizar una agrupación de objetos `ImagenDTO` con la finalidad de proyectar en un nuevo objeto de tipo `ImagenDTO` la información de las imágenes que se relacionan con las etiquetas del filtro y su

respectivo conjunto de etiquetas. Finalmente, al usuario se le entrega como resultado una lista de imágenes agrupadas en la variable `groupedImagenes` y que corresponden con una lista de objetos `ImagenDTO`. Caso contrario si en el filtrado no encuentra elementos se devuelve un `BadRequest` anunciando que no existen registros (línea 41).

```
1   [HttpGet("filtrarImg")]
2  public async Task<ActionResult<List<ImagenDTO>>> Filtrar([FromQuery] FiltroMultipleDeEtiquetas filtroImagenesEtiqueta)
3  {
4      var tempQueryable = from imagen in context.Imagenes
5                          join etiquetaImagen in context.EtiquetaImagen on imagen.Id equals etiquetaImagen.ImagenId
6                          join etiqueta in context.Etiquetas on etiquetaImagen.EtiquetaId equals etiqueta.Id
7                          where imagen.UsuarioId.Equals(usuario.Id)
8                          select new
9                          {
10                             Imagen = new ImagenDTO()
11                             {
12                                 Id = imagen.Id,
13                                 Img = imagen.Img,
14                                 NombreImagen = imagen.NombreImagen,
15                                 FechaSubida = imagen.FechaSubida,
16                             },
17                             Etiqueta = etiqueta.Nombre
18                         };
19     if (filtroImagenesEtiqueta.Tags != null && filtroImagenesEtiqueta.Tags.Any())
20     {
21         tempQueryable = tempQueryable.Where(i => filtroImagenesEtiqueta.Tags.Contains(i.Etiqueta));
22     }
23     if (tempQueryable.Any())
24     {
25         var groupedImagenes = await tempQueryable.GroupBy(i => i.Imagen.Id)
26                                     .Select(g =>
27                                     new ImagenDTO()
28                                     {
29                                         Id = g.Key,
30                                         Img = g.First().Imagen.Img,
31                                         NombreImagen = g.First().Imagen.NombreImagen,
32                                         FechaSubida = g.First().Imagen.FechaSubida,
33                                         Etiquetas = g.Select(x => x.Etiqueta).ToList()
34                                     }).ToListAsync();
35
36         return mapper.Map<List<ImagenDTO>>(groupedImagenes);
37     }
38     else
39     {
40         return BadRequest(new { message = "No se encontro registros" });
41     }
42 }
```

Figura 2.23. *Endpoint* `filtrarImg` con método HTTP GET

En la Figura 2.24 se define la lógica del *endpoint* `cargarImgWithTags`, el cual se decora con `[HttpPost("cargarImgWithTags")]` (línea 1) que indica la URL y el método POST asociados a este método asincrónico, y que devuelve una lista de objetos `ImagenDTO` que toma como parámetro un objeto `ImagenCreacionDTO` que se decora con el atributo `[FromForm]`, el cual indica que los datos deben recuperarse desde el cuerpo de la solicitud que corresponde a un formulario (línea 2). Además, mediante un bloque `try` (de la línea 4 hasta la línea 50) se pretende atrapar excepciones y en caso de que estas se produzcan se presente un mensaje de error con el bloque `catch` (de la línea 51 hasta la línea 54). Se declaran cuatro objetos (de la línea 6 hasta la línea 9): `image`, para convertir objetos de tipo `ImagenCreacionDTO` a objetos de tipo `Imagen`;

litTags para almacenar una lista de enteros como resultado de llamar al método GuardarNuevosTags; objectsForSave, que es una lista de objetos del tipo EtiquetaImagen, y que se utiliza para generar las relaciones entre etiquetas e imágenes; y el objeto imageDTOs del tipo ImagenDTO, que se utiliza para almacenar el resultado del *endpoint* que será enviado como respuesta a la solicitud del usuario. Posteriormente se realiza una comprobación de las propiedades del objeto imagesDto (de la línea 11 hasta la línea15), es decir se verifica que las propiedades que contienen el listado de archivos y el listado de etiquetas no estén vacías o nulas.

Luego se itera para guardar cada archivo en un flujo de memoria y luego se lo procesa (línea 18); el flujo de memoria se crea en la línea 20, luego este flujo se copia utilizando el método copyToAsync de la interfaz IFormFile (línea 22). De forma continua se extrae la extensión del archivo mediante formFile.FileName y se lo almacena en fileExtension (línea 23), luego de lo cual se procede a guardar el archivo haciendo uso del método GuardarArchivo (línea 24 y 25) que toma los datos del flujo de memoria, la extensión del archivo, el contenedor, el tipo de contenido y el nombre del archivo. Posteriormente se establecen las propiedades de image (de la línea 24 hasta la línea 28): Img, que contiene la URL generada por el método GuardarArchivo; NombreImagen, que contiene el nombre original del archivo; FechaSubida, que contiene la fecha y hora del instante de tiempo en el cual se cargaron las imágenes; y UsuarioId, que contiene la identificación del usuario que realiza la carga de la imagen, la cual se obtiene del contexto actual de la solicitud HTTP. Posteriormente se agrega la información de image en la base de datos utilizando el método await.context.AddAsync (línea 30) y se guardan los cambios utilizando el método await.context.SaveChangesAsync (línea 31).

Una vez que los datos de la imagen han sido procesados y almacenados, iterando (de la línea 33 hasta la línea 40), se procesa litTags, que contiene el listado de los identificadores de las etiquetas, y en cada iteración se asocia cada imagen que ha sido cargada con sus etiquetas generando una instancia de EtiquetaImagen para cada identificador de etiqueta e identificador de imagen. Finalmente, cada registro se agrega en objectsForSave. Una vez terminada las iteraciones, se procede a agregar los datos utilizando el método context.EtiquetaImagen.AddRangeAsync(línea 46), el cual, de forma masiva, almacena la información relacionada entre imágenes y etiquetas, y posteriormente se guardan los cambios en la base de datos (línea 47).

Finalmente, para obtener los resultados de la información cargada se procede a asignar al objeto `image` a un objeto `ImagenDTO` (línea 41) y se establece en su propiedad `Etiquetas` el valor de la propiedad `Etiquetas` del objeto `imagesDto` (línea 42). El objeto `imageDTO` resultante se agrega a una lista de objetos `imageDTOs` (línea 43). De esta forma al finalizar el proceso se retorna como respuesta un `OK` con los objetos `imageDTOs` (línea 48) que representan la información cargada en ese momento por un usuario.

```
1  [HttpPost("cargarImgWithTags")]
2  public async Task<ActionResult<List<ImagenDTO>>> Post([FromForm] ImagenCreacionDTO imagesDto)
3  {
4      try
5      {
6          var image = mapper.Map<Imagen>(imagesDto);
7          var litTags = GuardarNuevosTags(imagesDto);
8          var objetsForSave = new List<EtiquetaImagen>();
9          var imageDTOs = new List<ImagenDTO>();
10
11         if (imagesDto.Imgs == null || !imagesDto.Imgs.Any() || imagesDto.Etiquetas == null
12             || !imagesDto.Etiquetas.Any())
13         {
14             return BadRequest(new { message = "Cargue al menos una imagen con al menos una etiqueta" });
15         }
16         else
17         {
18             foreach (var formFile in imagesDto.Imgs)
19             {
20                 using (var memoryStream = new MemoryStream())
21                 {
22                     await formFile.CopyToAsync(memoryStream);
23                     var fileExtension = Path.GetExtension(formFile.FileName);
24                     image.Img = await archivos.GuardarArchivo(memoryStream.ToArray(),fileExtension, contenedor,
25                                                 formFile.ContentType,formFile.FileName);
26                     image.NombreImagen = Path.GetFileName(formFile.FileName);
27                     image.FechaSubida = DateTime.Now;
28                     image.UsuarioId = usuario.Id;
29
30                     await context.AddAsync(image);
31                     await context.SaveChangesAsync();
32
33                     foreach (var etiquetaID in litTags)
34                     {
35                         objetsForSave.Add(new EtiquetaImagen()
36                         {
37                             ImagenId = image.Id,
38                             EtiquetaId = etiquetaID
39                         });
40                     }
41                     var imageDTO = mapper.Map<ImagenDTO>(image);
42                     imageDTO.Etiquetas = imagesDto.Etiquetas;
43                     imageDTOs.Add(imageDTO);
44                 }
45             }
46             await context.EtiquetaImagen.AddRangeAsync(objetsForSave);
47             await context.SaveChangesAsync();
48             return Ok(imageDTOs);
49         }
50     }
51     catch
52     {
53         return BadRequest(new { message = "Datos invalidos, revise y vuelva a intentarlo" });
54     }
55 }
```

Figura 2.24. *Endpoint* `cargarImgWithTags`

En la Figura 2.25 se define el método privado `GuardarNuevosTags`, el cual toma como parámetro un objeto de tipo `ImagenCreacionDTO` y devuelve como resultado una lista de enteros (línea 1). En el método se genera una lista vacía de números enteros, que se utiliza para almacenar los identificadores asignados a las etiquetas guardadas (línea 2). Si la propiedad `Etiquetas` del objeto `ImagenCreacionDTO` no es nula (línea 4), se itera sobre cada etiqueta de la lista (línea 6). Para cada etiqueta, se verifica si la etiqueta ya existe en la base de datos buscando un registro en la tabla `Etiquetas` y comparando la columna `Nombre` con lo definido en `newTag` (línea 8). Si se encuentra un registro, se agrega el identificador de la etiqueta existente a `result` (de la línea 10 hasta la línea 13). Si no se encuentra ningún registro, se agrega un nuevo registro a la tabla `Etiquetas` con el nombre de la nueva etiqueta (de la línea 14 hasta la línea 19) y se guardan los cambios en la base de datos (línea 20). El identificador de la etiqueta recién agregada se incluye en la lista `result` (línea 21). Finalmente, el método devuelve la lista de identificadores de etiquetas que se han guardado o encontrado en la base de datos (línea 25).

```
1  private List<int> GuardarNuevosTags(ImagenCreacionDTO imagesDto)
2  {
3      var result = new List<int>();
4      if (imagesDto.Etiquetas != null)
5      {
6          foreach (var newTag in imagesDto.Etiquetas)
7          {
8              var existingTag = context.Etiquetas.FirstOrDefault(e => e.Nombre == newTag);
9
10             if (existingTag != null)
11             {
12                 result.Add(existingTag.Id);
13             }
14             else
15             {
16                 var resultEntity = context.Etiquetas.Add(new Etiqueta()
17                 {
18                     Nombre = newTag
19                 });
20                 context.SaveChanges();
21                 result.Add(resultEntity.Entity.Id);
22             }
23         }
24     }
25     return result;
26 }
```

Figura 2.25. Función `GuardarNuevosTags`.

CuentasController

En la Figura 2.26 se indica el código de la clase controlador `CuentasController`, la cual hereda de `ControllerBase` (línea 3), está decorada con `[ApiController]` (línea 1), y con `[Route("api/cuentas")]` (línea 2) para establecer la URL del

controlador. El constructor de la clase (de la línea 8 hasta la línea 13) tiene una inyección de dependencias: `ApplicationDbContext` y `IConfiguration`, que se utilizan para acceder a la base de datos y acceder a la información del archivo de configuración `appsettings.Development.json`, respectivamente.

Además, tiene dos métodos de acción: `login` (línea 15) que corresponde al método HTTP POST y cuya funcionalidad es permitir iniciar sesión con la finalidad de obtener el token para poder realizar el proceso de autenticación; y `crearUsuario` (línea 21) que corresponde al método HTTP POST, y cuya funcionalidad principal es generar el usuario con base en la información proporcionada.

```
1   [ApiController]
2   [Route("api/cuentas")]
3   public class CuentasController : ControllerBase
4   {
5       private readonly ApplicationDbContext dbContext;
6       private readonly IConfiguration configuration;
7
8       public CuentasController(ApplicationDbContext dbContext, IConfiguration configuration)
9       {
10          this.dbContext = dbContext;
11          this.configuration = configuration;
12
13      }
14
15      [HttpPost("login")]
16      public IActionResult Login([FromForm]LoginDeUsuario input)
17      {
18          //Logica a implementarse
19      }
20
21      [HttpPost("crearUsuario")]
22      public IActionResult Create([FromForm] CreacionDeCuenta input)
23      {
24          //Logica a implementarse
25      }
26  }
```

Figura 2.26. Estructura de controlador `CuentasController`

En el controlador, se debe habilitar la opción que permitirá mantenerlos aislados y que solo puedan ser accedidos por usuarios que dispongan de un *token*, para esto se debe configurar Swagger en el método `ConfigureServices` de la clase `Startup`.

En la Figura 2.27 se presenta las dos definiciones necesarias para habilitar la opción de autenticación: la primera especifica la definición de seguridad ,de la línea 8 hasta la línea 16, al hacerlo en la interfaz de Swagger se considera que se debe agregar la función de autorización; la segunda especifica los requerimientos de seguridad, al hacerlo se

agrega un encabezado de autorización en cada extremo cuando se envié la solicitud ,de la línea 19 hasta la línea 33 [32].

```
1  if (HostEnvironment.IsDevelopment())
2 {
3    services.AddSwaggerGen(options =>
4    {
5
6 // Especifica la definicion de seguridad
7 // Al hacerlo Swagger tendrá en cuenta que deberá agregar la función de autorización.
8       options.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme, new OpenApiSecurityScheme
9           {
10              Scheme = JwtBearerDefaults.AuthenticationScheme,
11              BearerFormat = "JWT",
12              In = ParameterLocation.Header,
13              Name = "Authorization",
14              Description = "Autenticación de portador con token JWT",
15              Type = SecuritySchemeType.Http
16          });
17 //Especifica los requerimientos de seguridad
18 //Agregará un encabezado de autorización a cada extremo cuando se envíe la solicitud.
19       options.AddSecurityRequirement(new OpenApiSecurityRequirement
20       {
21         {
22           new OpenApiSecurityScheme
23           {
24              Reference = new OpenApiReference
25              {
26                 Id = JwtBearerDefaults.AuthenticationScheme,
27                 Type = ReferenceType.SecurityScheme
28              }
29           },
30           new List<string>()
31         }
32       });
33    });
34 }
```

Figura 2.27. Modificación de la clase `Startup`

En la Figura 2.28 se define el *endpoint* `crearUsuario`, el cual se decora con `[HttpPost("crearUsuario")]` (línea 1), devuelve un `IActionResult`, que representa el resultado de la ejecución de una acción (línea 2) y toma como parámetro un objeto del tipo `CreacionDeCuenta` el cual está decorado con el atributo `[FromForm]` (línea 2).

Con la finalidad de realizar verificaciones se valida el objeto `CreacionDeCuenta` en tres partes del código: la primera, de la línea 4 hasta la línea 9, realiza la verificación de coincidencia de la contraseña ingresada con la contraseña de confirmación; la segunda, de la línea 13 hasta la línea 16, realiza la búsqueda de un usuario existente en la base de datos, cuyo correo coincida con el dato ingresado; y al tercera, de la línea 17 hasta la línea 20, se valida que el nombre de usuario ingresado coincida con el almacenado en la base de datos.

Si los datos ingresados no superan estas verificaciones se entrega un `BadRequest` especificando el error encontrado, caso contrario con la información ingresada se crea un nuevo objeto `Usuario`, de la línea 23 hasta la línea 29, que se agrega a la base de datos (línea 30) y se guarda (línea 31). Finalmente, se devuelve un mensaje en el que se especifica el usuario creado (línea 32).

```
1  [HttpPost("crearUsuario")]
2  public IActionResult Create([FromForm] CreacionDeCuenta input)
3  {
4      if (input.Password != input.ConfirmPassword)
5      {
6          //Limpia los campos del password para volver a validar
7          input.Password = "";
8          input.ConfirmPassword = "";
9          return BadRequest("La confirmacion de contraseña no coincide");
10     }
11     if (dbContext.Usuarios.Any(usuario => usuario.UserName == input.UserName))
12     {
13         return BadRequest("El usuario ya existe en el sistema");
14     }
15     if (dbContext.Usuarios.Any(usuario => usuario.Correo == input.Correo))
16     {
17         return BadRequest("El correo ya esta registrado en el sistema");
18     }
19     else
20     {
21         var newUser = new Usuario()
22         {
23             Nombre = input.Nombre,
24             Correo = input.Correo,
25             UserName = input.UserName,
26             Password = EncriptadorSHA256.generarClaveSHA1(input.Password),
27         };
28         dbContext.Usuarios.Add(newUser);
29         dbContext.SaveChanges();
30         return Ok("Usuario Creado: " + newUser.UserName);
31     }
32 }
```

Figura 2.28. *endpoint* `crearUsuario` con método HTTP POST

Por otro lado, el método privado `GetUser` (ver Figura 2.29) acepta como parámetro de entrada un objeto del tipo `LoginDeUsuario` y devuelve un objeto del tipo `Usuario` (línea 1). El propósito de este método es consultar la base de datos el usuario que corresponda al nombre de usuario y la contraseña que han sido proporcionados, y devolver el usuario, si se lo encuentra. Para esto se utiliza el método `FirstOrDefault` el cual devuelve el primer elemento de una secuencia que cumple una condición específica (línea 3 y 4).

```
1 private Usuario GetUser(LoginDeUsuario input)
2 {
3     return dbContext.Usuarios.FirstOrDefault(usuario => usuario.UserName == input.UserName
4     && usuario.Password == EncriptadorSHA256.generarClaveSHA1(input.Password));
5 }
```

Figura 2.29. Método `GetUser`

En la Figura 2.30 se define la lógica del *endpoint* `Login` el cual se decora con `[HttpPost("login")]` (línea 1), devuelve un `IActionResult` y toma como parámetro un objeto del tipo `LoginDeUsuario` el cual está decorado con el atributo `[FromForm]` (línea 2). En este método, mediante `GetUser` (línea 4) se recupera el usuario que corresponda a la información provista para posteriormente realizar verificaciones.

La primera verificación (de la línea 7 hasta la línea 10) comprueba si el resultado es nulo, de ser el caso se devuelve un `BadRequest`; la segunda verificación (de la línea 11 hasta la línea 14) comprueba que los datos de nombre de usuario y contraseña proporcionados coincidan con los datos de la base, si no existe la coincidencia se devuelve un `BadRequest`.

Si se supera con éxito las validaciones, se generan cinco objetos (de la línea 24 hasta la línea 30): `claims`, que contiene el listado de datos que sirven para identificar al usuario como su identificador, nombre, correo y nombre de usuario; `secretKey`, que contiene un objeto `SymmetricSecurityKey` que toma como parámetro una matriz de bytes que se obtiene mediante una codificación que utiliza la clave secreta almacenada en el archivo de configuración `appsettings.Development.json` con la clave `jwt:key`; `siginCredentials`, que contiene un objeto `SigningCredentials` que toma como parámetro el objeto contenido en `secretKey` y un algoritmo de firma, que en este caso es HMAC-SHA256[35]; `tokenOptions`, que contiene un objeto `JwtSecurityToken` que toma como parámetros: los *claims*, el tiempo de caducidad y las credenciales de la firma; y `tokenString`, que contiene el *token* JWT.

[35] HMAC-SHA256: Es un método seguro para verificar la autenticidad e integridad de los datos mediante una clave secreta y la función hash SHA-256.

Una vez concluido el proceso de generación del *token* se devuelve como respuesta un objeto del tipo `RespuestaDeJWT` (de la línea 31 hasta la línea 39) en el cual consta el *token* generado y otros datos del usuario.

```
1 [HttpPost("login")]
2 public IActionResult Login([FromForm]LoginDeUsuario input)
3 {
4     var user = GetUser(input);
5     var pass = EncriptadorSHA256.generarClaveSHA1(input.Password);
6
7     if (user == null)
8     {
9         return BadRequest("Usuario desconocido o Password incorrecto");
10    }
11    if (!(input.UserName == user.UserName && pass == user.Password))
12    {
13        return BadRequest("Usuario desconocido o Password incorrecto");
14    }
15
16    var claims = new List<Claim>()
17        {
18            new Claim( ClaimTypes.NameIdentifier, user.Id.ToString()),
19            new Claim( ClaimTypes.Name, user.Nombre),
20            new Claim( ClaimTypes.Email, user.Correo),
21            new Claim( "User", user.UserName),
22        };
23
24    var secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["jwt:key"]));
25    var siginCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);
26    var tokenOptions = new JwtSecurityToken(
27    claims: claims,
28    expires: DateTime.Now.AddMinutes(30),
29    signingCredentials: siginCredentials);
30    var tokenString = new JwtSecurityTokenHandler().WriteToken(tokenOptions);
31    return Ok(new RespuestaDeJWT()
32    {
33        Token = tokenString,
34        Correo = user.Correo,
35        Id = user.Id,
36        Nombre = user.Nombre,
37        UserName = user.UserName,
38        Message = "Login Correcto"
39    });
40 }
```

Figura 2.30. *Endpoint* `login`

Configuraciones Finales

En el archivo de configuración `appsettings.Development.json` se agrega la configuración de la clave `jwt:key` , la cual es una cadena de caracteres que se corresponde a la clave secreta. En la Figura 2.31 se observa esta configuración.

```
1 "jwt": {
2
3     "key": "djsen9w34hrf734f834yeiuwh3u92ewu3e8hf3293i9hf82efsuhuwe9fh83he98h23u29efw94f792h4"
4   }
```

Figura 2.31. Configuración de clave `jwt:key`

Finalmente, una vez configurado el controlador `CuentasController`, se debe verificar que el controlador `ImagenController` se encuentre decorado con el atributo `[Authorize]` (linea1), esto indica que las solicitudes que se deseen realizar a los *endpoints* deben estar autorizadas.

En la Figura 2.32 se observa esta configuración.

```
1 [Authorize]
2 [ApiController]
3 [Route("api/imagenes")]
4 public class ImagenController : ControllerBase
5 {
6     //Logica implementada
7 }
```

Figura 2.32. Modificación al controlador `ImagenController`

3 RESULTADOS, CONCLUSIONES Y RECOMENDACIONES

En este capítulo se detallan los resultados obtenidos a partir de diferentes pruebas realizadas a los *endpoints* pertenecientes a los controladores desarrollados para el subsistema de clasificación. El código del subsistema se encuentra disponible en el Anexo III.

Finalmente, se define un conjunto de conclusiones y recomendaciones, generadas con base en el desarrollo del subsistema.

3.1 RESULTADOS

Como se observó en la Sección 2.2.6, la web API tiene un conjunto de controladores; cada controlador tiene *endpoints* a los cuales se puede acceder a partir de las diferentes peticiones HTTP. Para probar los *endpoints* se utiliza la herramienta Swagger API la cual presenta una interfaz de usuario documentada, y permite la visualización de los campos de información que se envían a la API para posteriormente obtener las respuestas de la solicitud en formato JSON.

3.1.1 PRUEBAS DE FUNCIONAMIENTO

En la Figura 3.1 se indica la interfaz gráfica de Swagger API con los controladores y *endpoints* documentados.

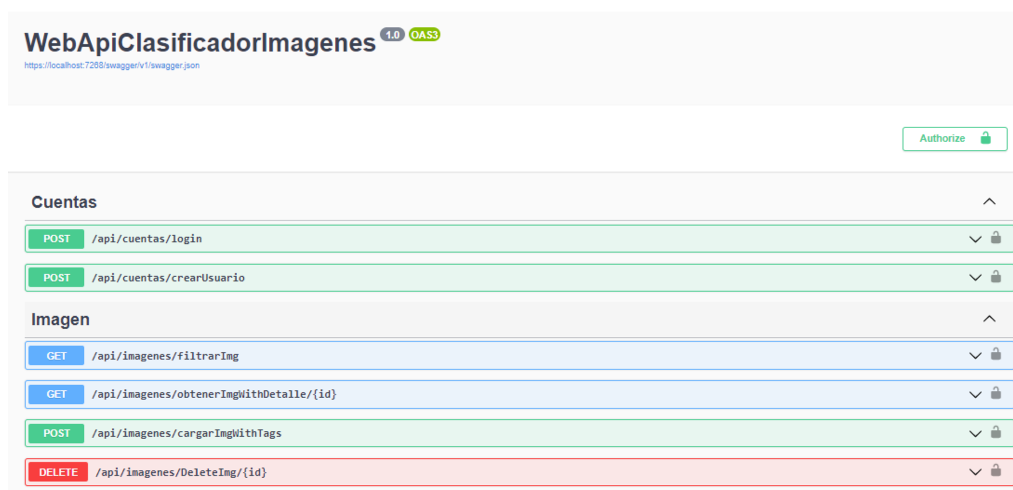

Figura 3.1. Interfaz gráfica de Swagger API

Debido a que el controlador `ImagenController` se encuentra protegido por el decorador `[Authorize]` el acceso a cada *endpoint* se encuentra restringido a menos que el usuario inicie sesión de forma exitosa y obtenga el *token*. En la Figura 3.2 se observa un resultado de error de acceso a un *endpoint* de `ImagenController` sin previa autorización.

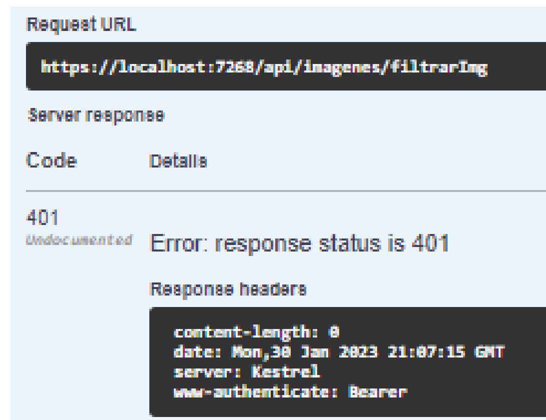

Figura 3.2. Error de acceso al *endpoint* `filtrarImg`

Para las pruebas del controlador `CuentasController` se consideró 5 usuarios que manejarán su información de imágenes y etiquetas de manera separada, para esto se hace uso del *endpoint* `crearUsuario`. En la Figura 3.3 se observa la generación de un usuario.

Nombre * required
string
Fabricio Almeida
Correo * required
string($email)
falmeida411@hotmail.com
UserName * required
string
FAlmeida98
Password * required
string($password)
•••••••••••••
ConfirmPassword * required
string($password)
•••••••••••••

Figura 3.3. Generación de usuario

En la Figura 3.4 se observa la respuesta que se recibe una vez enviada la información a la API.

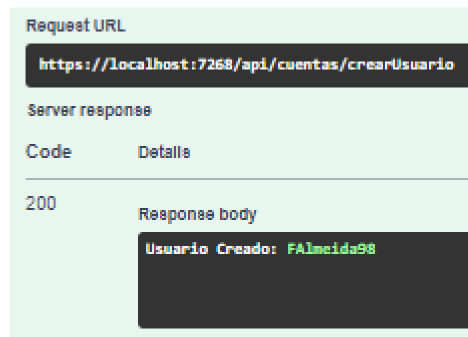

Figura 3.4. Respuesta de la API al crear un usuario

Un requerimiento era la verificación única de correo y nombre de usuario. En la Figura 3.5 se observa los errores que se producen si se envía el mismo correo, pero con diferente nombre de usuario o si se envía un correo diferente, pero con el mismo nombre de usuario.

Figura 3.5. Respuesta de errores al generar nuevos usuarios

Con los usuarios generados se puede hacer uso del *endpoint* `login`. En la Figura 3.6 se observa el ingreso de los parámetros.

UserName * required
string
Gabriela2000
Password * required
string($password)
••••••••••••••

Figura 3.6. Ingreso de parámetros del *endpoint* `login` de usuario

Como se muestra en la Figura 3.7, si el *login* realizado fue correcto se genera la respuesta con el *token* y los datos del usuario que ha realizado la solicitud.

200

Response body


```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8
DA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoiR2FicmllbGEgSGVyZWRpYSIsImh0dHA6Ly9zY2hlbWFzLnhtbHNvY
iOiJHYWJyaWVsYTIwMDAiLCJleHAiOjE2NzUxMTcwNDN9.UhNSgxDQM0hvrLuTqu1jXhmotFzlRTHvZDsF8kUfL9c",
  "id": 2,
  "nombre": "Gabriela Heredia",
  "correo": "herediag911@gmail.com",
  "userName": "Gabriela2000",
  "message": "Login Correcto"
}
```


Figura 3.7. Respuesta al realizar un *login* correcto

Caso contrario se genera una respuesta de error como la visualizada en la Figura 3.8.

Code	Details
400 *Undocumented*	Error: response status is 400 Response body `Usuario desconocido o Password incorrecto`

Figura 3.8. Respuesta de error del *endpoint* `login`

En `ImagenController` se realizaron las siguientes pruebas: al hacer clic sobre el botón `Authorize` de la interfaz de Swagger, se despliega el formulario que se visualiza en la Figura 3.9, con el cual se puede agregar como parámetro el *token* obtenido como respuesta al uso del *endpoint* `login`. Si el *token* es válido se permitirá al usuario realizar las operaciones de la web API, caso contrario los *endpoints* enviarán mensajes de error.

Available authorizations ×

Bearer (http, Bearer)

Autenticación de portador con token JWT

Value:

eyJhbGciOiJIUzI1NiIsInR5cC

Authorize Close

Figura 3.9. Formulario para autorizar el acceso a los *endpoints* a partir del *token*

En el *endpoint* `cargarImgWithTags` se realiza la carga de un conjunto de archivos de imagen y un conjunto de etiquetas, este proceso se visualiza en la Figura 3.10.

Figura 3.10. Proceso de carga de imágenes y etiquetas haciendo uso del *endpoint* `cargarImgWithTags`

Si el proceso de carga se realiza correctamente el usuario recibe como respuesta la información de los objetos que han sido cargados, como se observa en la Figura 3.11.

Figura 3.11. Respuesta a la petición de carga de imágenes y etiquetas

Si alguno de los archivos no tiene extensión .JPG, .PNG o .GIF o si el peso del archivo excede los 4 MB se devuelve como respuesta los mensajes de error que se visualizan en la Figura 3.12.

Para esta prueba dos de los archivos anteriores se modificaron y se pretendió subir a la API un archivo con extensión .ZIP (*Compressed File Archive)* con un peso de más de 4 MB y un archivo PDF (*Portable Document Format*).

Figura 3.12. Respuesta de errores en los archivos

Por otra parte, si el usuario intenta realizar la carga de imágenes, pero sin etiquetas o viceversa, se tiene como respuesta el error que se puede visualizar en la Figura 3.13.

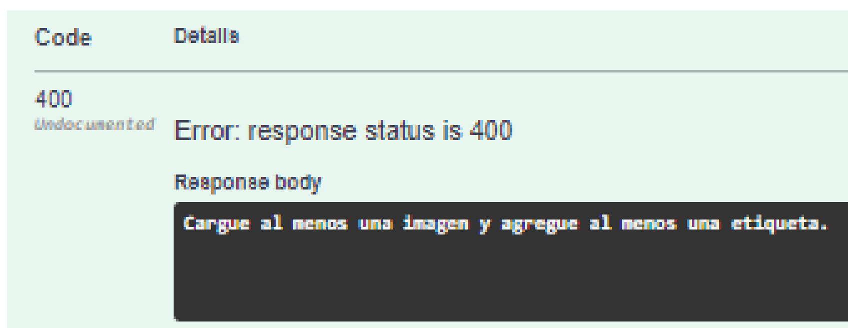

Figura 3.13. Respuesta de error al subir incorrectamente la información

Finalmente, si la información que se va a cargar a la API cumple con las verificaciones, se genera un directorio por cada usuario con su respectiva información de archivos como se indica en la Figura 3.14.

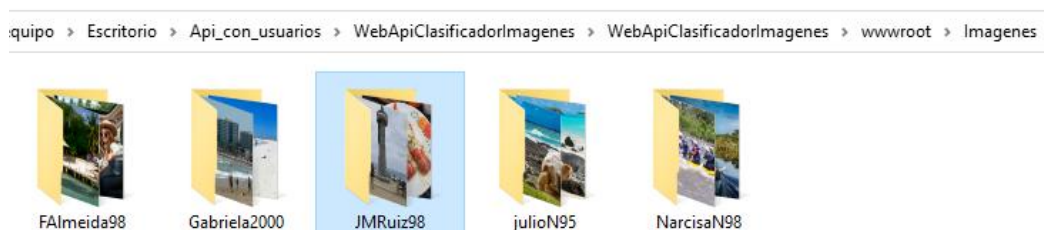

Figura 3.14. Directorios de usuarios

Una vez cargada la información se debe recuperar los datos de la API, para esto se utiliza el *endpoint* `filtrarImg` que utiliza como parámetro de búsqueda un conjunto de etiquetas.

En la Figura 3.15 se presenta la prueba del *endpoint* sin agregar parámetros de búsqueda, al hacerlo la API devuelve como resultado toda la información de imágenes y etiquetas que han sido cargadas por el usuario, en este caso los datos pertenecen al usuario `julioN95`.

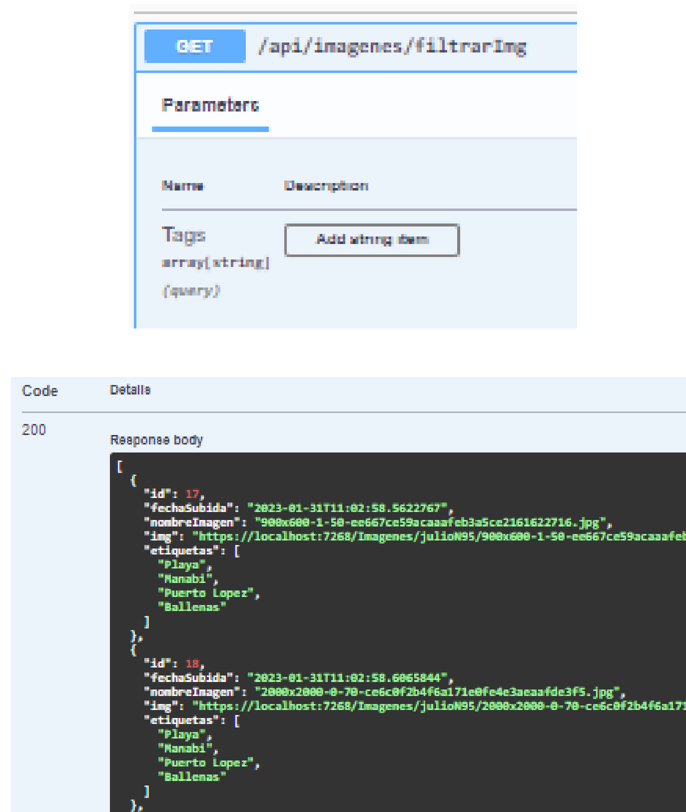

Figura 3.15. Respuesta obtenida sin agregar parámetros de búsqueda

Para el caso de la Figura 3.16, se pasan al *endpoint* parámetros de búsqueda, al hacerlo la API devuelve como resultado la información de imágenes y etiquetas relacionadas a la información proporcionada. En este caso la información recuperada le pertenece al usuario `julioN95` y se recupera los datos en base a las etiquetas `Ecuador` y `Ballenas`.

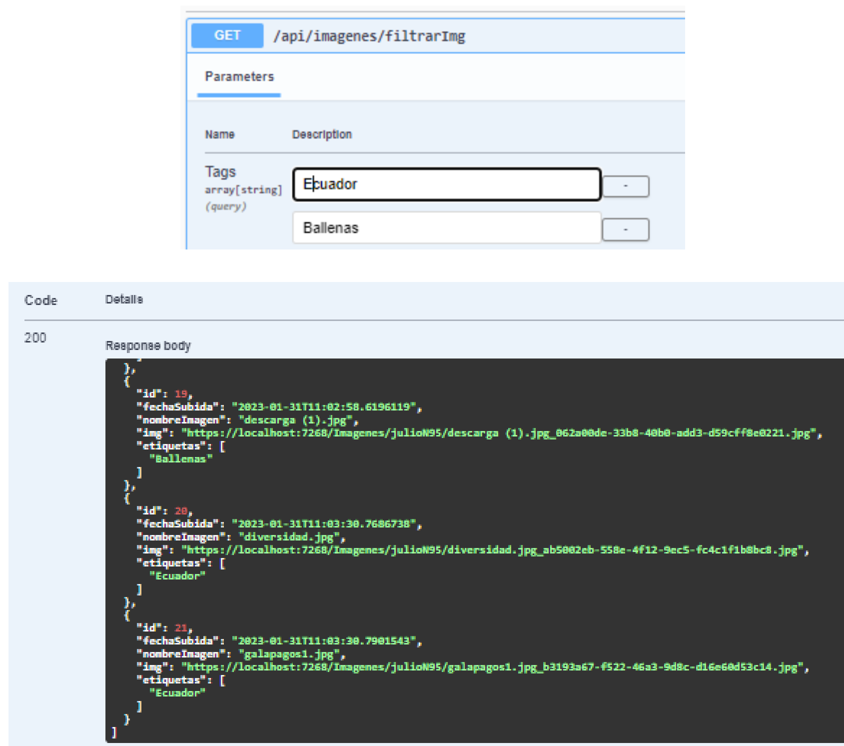

Figura 3.16. Recuperación de información relacionada a los parámetros de búsqueda

Para realizar las pruebas que demuestren que en este caso el usuario `julioN95` no pueda observar la información de otros usuarios, es necesario visualizar la información general de datos que han sido cargados en la base.

En la Figura 3.17 se realiza una recuperación de información de las tablas: `Etiqueta`, `Imagen` y `Usuario`.

La información obtenida está dividida en cuatro columnas que corresponden al identificador de la etiqueta, al nombre de la etiqueta, al identificador de la imagen, al nombre de la imagen, al identificador del usuario, y al nombre del usuario.

De esta forma es posible observar con mayor detalle la información que ha sido cargada en la base de datos haciendo uso del *endpoint* para cargar imágenes con etiquetas de la web API.

Gracias a esta información es posible obtener con relevancia los datos de las etiquetas que se asocian con las imágenes y las imágenes que están asociadas con los diferentes usuarios. Para las pruebas se toma como referencia la información que ha sido proporcionada por el usuario `julioN95` y el usuario `JMRuiz98`.

EtiquetaId	Nombre	ImagenId	NombreImagen	UsuarioId	UserName
14	Oriente	13	cuyabeno-3.jpg	3	NarcisaN98
12	Ecuadpr	14	slider-home-6.jpg	3	NarcisaN98
13	Cuyabeno	14	slider-home-6.jpg	3	NarcisaN98
14	Oriente	14	slider-home-6.jpg	3	NarcisaN98
15	Mundial Q...	15	_128046598_gettyimages-1245712344.jpg	3	NarcisaN98
16	Estadio	15	_128046598_gettyimages-1245712344.jpg	3	NarcisaN98
15	Mundial Q...	16	descarga.jpg	3	NarcisaN98
16	Estadio	16	descarga.jpg	3	NarcisaN98
3	Playa	17	salinas1.jpg	4	JMRuiz98
5	Ecuador	17	salinas1.jpg	4	JMRuiz98
17	Salinas	17	salinas1.jpg	4	JMRuiz98
18	Comida Tr...	17	salinas1.jpg	4	JMRuiz98
3	Playa	18	langostas.jpg	4	JMRuiz98
5	Ecuador	18	langostas.jpg	4	JMRuiz98
17	Salinas	18	langostas.jpg	4	JMRuiz98
18	Comida Tr...	18	langostas.jpg	4	JMRuiz98
3	Playa	19	salinas2.jpg	4	JMRuiz98
5	Ecuador	19	salinas2.jpg	4	JMRuiz98
17	Salinas	19	salinas2.jpg	4	JMRuiz98
18	Comida Tr...	19	salinas2.jpg	4	JMRuiz98
3	Playa	20	900x600-1-50-ee667ce59acaaafeb3a5ce...	5	julioN95
19	Manabi	20	900x600-1-50-ee667ce59acaaafeb3a5ce...	5	julioN95
20	Puerto Lo...	20	900x600-1-50-ee667ce59acaaafeb3a5ce...	5	julioN95
21	Ballenas	20	900x600-1-50-ee667ce59acaaafeb3a5ce...	5	julioN95
3	Playa	21	descarga (1).jpg	5	julioN95
19	Manabi	21	descarga (1).jpg	5	julioN95
20	Puerto Lo...	21	descarga (1).jpg	5	julioN95
21	Ballenas	21	descarga (1).jpg	5	julioN95
3	Playa	22	descarga.jpg	5	julioN95
19	Manabi	22	descarga.jpg	5	julioN95
20	Puerto Lo...	22	descarga.jpg	5	julioN95
21	Ballenas	22	descarga.jpg	5	julioN95
5	Ecuador	23	galapagos1.jpg	5	julioN95
11	galapagos	23	galapagos1.jpg	5	julioN95
5	Ecuador	24	diversidad.jpg	5	julioN95
11	galapagos	24	diversidad.jpg	5	julioN95

Figura 3.17. Información cargada en la base de datos

En la Figura 3.17 se observa el caso del usuario `julioN95` el cual subió imágenes con etiquetas relacionadas a `Playa`, `Manabí`, `Puerto López`, `Ballenas`, `Ecuador` y `Galápagos`; mientras que el usuario `JMRuiz98` realizó la carga de imágenes con etiquetas relacionadas a `Playa`, `Ecuador`, `Salinas` y `Comida Tradicional`. En este caso el usuario `julioN95` no debe recuperar imágenes que se relacionen con `Salinas` y `Comida Tradicional`, como se indica en la Figura 3.18.

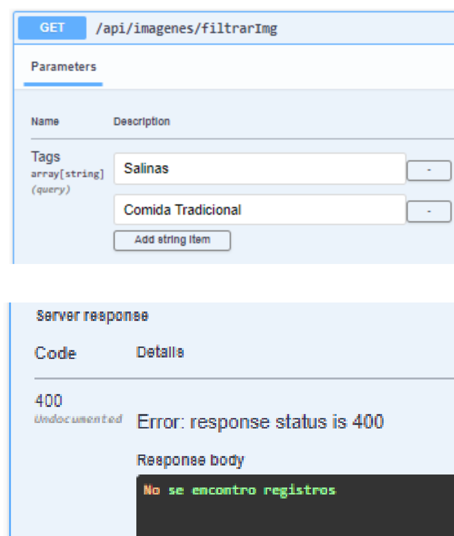

Figura 3.18. Respuesta de error con los parámetros de búsqueda asignados

Por otra parte, si este usuario ingresa `Playa` y `Ecuador` debe únicamente recuperar las imágenes con identificadores `20`, `21`, `22`, `23` y `24`, como se indica en la Figura 3.19.

Name	Description
Tags array[string] *(query)*	Playa [-] Ecuador [-] Add string item


```
[
  {
    "id": 20,
    "fechaSubida": "2023-02-09T18:08:29.0980065",
    "nombreImagen": "900x600-1-50-ee667ce59acaaafeb3a5ce2161622716.jpg",
    "img": "https://localhost:7268/Imagenes/julioN95/900x600-1-50-ee667ce59acaaafeb3a5ce2161622716.jpg_58835dcc-5
    "etiquetas": [
      "Playa"
    ]
  },
  {
    "id": 21,
    "fechaSubida": "2023-02-09T18:08:29.1150191",
    "nombreImagen": "descarga (1).jpg",
    "img": "https://localhost:7268/Imagenes/julioN95/descarga (1).jpg_dbe75c4c-d30f-4ce0-8f43-3c8b1c6cc9a4.jpg",
    "etiquetas": [
      "Playa"
    ]
  },
  {
    "id": 22,
    "fechaSubida": "2023-02-09T18:08:29.1290074",
    "nombreImagen": "descarga.jpg",
    "img": "https://localhost:7268/Imagenes/julioN95/descarga.jpg_b881cd66-bd86-43ff-b730-a4888fd47387.jpg",
    "etiquetas": [
      "Playa"
    ]
  },
  {
    "id": 23,
    "fechaSubida": "2023-02-09T18:09:03.7911466",
    "nombreImagen": "galapagos1.jpg",
    "img": "https://localhost:7268/Imagenes/julioN95/galapagos1.jpg_88e7b152-8e16-4e17-873c-823582d42f9b.jpg",
    "etiquetas": [
      "Ecuador"
    ]
  },
  {
    "id": 24,
    "fechaSubida": "2023-02-09T18:09:03.7995019",
    "nombreImagen": "diversidad.jpg",
    "img": "https://localhost:7268/Imagenes/julioN95/diversidad.jpg_18096b12-c1eb-4510-a43e-6c36ae0d582a.jpg",
    "etiquetas": [
      "Ecuador"
    ]
  }
]
```


Figura 3.19. Respuesta de datos en formato JSON del usuario julioN95

Para el caso del usuario `JMRuiz98`, si los parámetros de búsqueda son `Playa` y Ecuador únicamente se debe recuperar las imágenes con identificadores `17`, `18` y `19`, como se observa en la Figura 3.20.

Name	Description
Tags array[string] (query)	Playa Ecuador Add string item


```
[
  {
    "id": 17,
    "fechaSubida": "2023-02-09T18:04:39.7745967",
    "nombreImagen": "salinas1.jpg",
    "img": "https://localhost:7268/Imagenes/JMRuiz98/salinas1.jpg_e9f33aaa-4f6b-460c-9d90-996d144ac70e.jpg",
    "etiquetas": [
      "Playa",
      "Ecuador"
    ]
  },
  {
    "id": 18,
    "fechaSubida": "2023-02-09T18:04:39.7859347",
    "nombreImagen": "langostas.jpg",
    "img": "https://localhost:7268/Imagenes/JMRuiz98/langostas.jpg_76902239-7882-463f-8521-358ac82b90e2.jpg",
    "etiquetas": [
      "Playa",
      "Ecuador"
    ]
  },
  {
    "id": 19,
    "fechaSubida": "2023-02-09T18:04:39.8005971",
    "nombreImagen": "salinas2.jpg",
    "img": "https://localhost:7268/Imagenes/JMRuiz98/salinas2.jpg_bc7e531b-7772-4272-a6c3-925ddd6ff62c.jpg",
    "etiquetas": [
      "Playa",
      "Ecuador"
    ]
  }
]
```


Figura 3.20. Respuesta de datos en formato JSON

Finalmente, de la misma forma se realiza la comprobación de independencia de cuentas entre usuarios permitiendo así que cada usuario maneje su respectiva información aun cuando la probabilidad de coincidencias en las etiquetas sea alta como en el caso del usuario `JMRuiz98` y `julioN95` los cuales coincidieron en las etiquetas `Playa` y `Ecuador`; donde a pesar de esta coincidencia ninguno de los usuarios fue capaz de observar la información del otro usuario.

3.2 CONCLUSIONES Y RECOMENDACIONES

3.2.1 CONCLUSIONES

- La implementación del subsistema de clasificación de fotografías se la llevó a cabo mediante un proceso ordenado el cual tuvo cuatro fases de trabajo que permitieron la planificación, el diseño, la implementación y la realización de pruebas del proyecto. Al finalizar con este Trabajo de Integración Curricular se

dispone de un subsistema de clasificación de imágenes cuya funcionalidad está avalada de acuerdo con los requerimientos establecidos.

- El análisis mediante un modelado de capas permitió establecer la estructura del proyecto mediante un diseño de diagrama de paquetes el cual se tomó de referencia para el modelado de clases y las relaciones entre componentes del subsistema de clasificación.
- El uso de la arquitectura REST para el desarrollo de microservicios permite definir controladores con *endpoints* que utilizan de manera directa los métodos y los códigos de respuesta HTTP, debido a esto la comunicación entre cliente y servidor se realiza por un lenguaje de intercambio como JSON lo cual permite establecer la independencia entre el lenguaje en el que este programado el cliente y el lenguaje de programación del servidor dando así una mayor fiabilidad, escalabilidad y flexibilidad en el servicio.
- Para el manejo de archivos que fluyen a través de las peticiones HTTP se utilizó la interfaz `IFormFile` la cual ofrece métodos y propiedades que facilitan el acceso a los metadatos de los archivos, con estos metadatos mediante el servicio implementado para realizar el almacenamiento de las imágenes se generan las diferentes rutas que serán almacenadas como un campo en una tabla de la base de datos.
- El manejo de objetos DTO permitió establecer las propiedades necesarias que serán transportadas y transferidas en diferentes procesos realizados por los *endpoints* en el caso de envió de imágenes en la entidad `Imagen`, el campo para almacenar la URL de ubicación del archivo es de tipo `string;` mientras que en el DTO se maneja un `List<IFormFile>` esto permite que el DTO pueda acceder a los metadatos de la imagen y se generen las rutas de almacenamiento para posteriormente almacenar la información de la imagen en un formato adecuado con los campos especificados en la entidad `Imagen`. Para realizar esta operación se utilizó `AutoMapper` el cual realiza el proceso de asociación entre los objetos DTO y los objetos Entidad con la finalidad de garantizar la transferencia de información entre diferentes componentes del subsistema.
- En el proceso de validación de archivos fue necesario el uso de `ValidationAttribute` la cual es una clase que proporciona el método heredado `IsValid` en el cual se establece la lógica de validación para los archivos que se reciben a través de las peticiones HTTP. Esto permite establecer

las restricciones en cuanto al peso y extensión de los archivos que se va a recibir en la API.

- Para realizar el proceso de clasificación de imágenes fue necesario el uso de una entidad intermedia la cual almacene los identificadores de las imágenes y los identificadores de las etiquetas de esta manera al momento de realizar él envió de la información a la base de datos los objetos que se almacenaran de forma masiva serán del tipo `EtiquetaImagen` generando así una estructura de datos organizada entre la entidad `Imagen` y la entidad `Etiqueta`.
- Para el desarrollo del *stub* del subsistema de almacenamiento se utilizó Entity Framework con la finalidad de utilizar migraciones que faciliten la generación de la base de datos en SQL a partir de las entidades establecidas en la web API.
- Para impedir el acceso no autorizado de usuarios se utilizó la autenticación basada en JWT, esto permitió a los usuarios que estén registrados realizar un *login* para que puedan obtener un *token* de acceso, este *token* permitirá que la web API realice el proceso de autorización basado en la estructura de este, permitiendo el acceso a los *endpoints* del controlador de imágenes.
- Para realizar las pruebas del *endpoint* que realiza el proceso de carga de imágenes y etiquetas, se utilizó un *endpoint* que simule la funcionalidad de búsqueda de información de las imágenes utilizando como parámetro las etiquetas que han sido anexadas a las imágenes.
- El consumo de los *endpoints* de la web API se simuló a partir de la interfaz gráfica ofrecida por Swagger API, la cual documenta la web API implementada permitiendo ejecutar las funcionalidades de los *endpoints* de tal manera que se realiza el envío de información al servidor en base a las propiedades que se requiera, y el servidor responderá con cuerpos de mensajes en formato JSON según sea el caso.

3.2.2 RECOMENDACIONES

Durante el desarrollo del subsistema de clasificación se presentó eventualidades que fueron estudiadas y analizadas para dar posibles soluciones. En base a estos problemas y el planteamiento de soluciones se consideran las siguientes recomendaciones que tienen como finalidad ayudar a la mejora del proceso e integración de subsistemas adicionales.

- Se recomienda analizar los componentes y la funcionalidad del subsistema de clasificación para el desarrollo del subsistema de adquisición y el subsistema de almacenamiento, ya que este componente actúa como puente en la comunicación de estos subsistemas.
- Para el desarrollo del subsistema se recomienda el estudio de los diferentes *namespaces* ofrecidos por ASP.NET en las páginas oficiales de Microsoft, con la finalidad de saber los métodos y propiedades con los cuales se puede trabajar de tal forma que se facilite la implementación
- El almacenamiento en un arreglo de bytes de una imagen en la base de datos puede resultar que la misma se vuelva pesada con cada archivo que ingrese al servidor, por este motivo, se recomienda utilizar un manejo de rutas de tal forma que se almacene la imagen en el servidor pero que en la base únicamente se guarde una URL de acceso al recurso.
- Se aconseja el uso de asociación de objetos con la finalidad de separar las preocupaciones que pueden existir en diferentes partes del subsistema en este caso entre los objetos DTO y los objetos Entidad.
- Se recomienda el uso de *claims* en un servicio de usuarios ya que estos proporcionan mediante un contexto de la solicitud actual HTTP la información perteneciente al usuario que se encuentra autenticado en la web API.
- Para mantener ocultos los *endpoints* es recomendable utilizar un esquema de autorización el cual permita que los usuarios que han sido autenticados puedan realizar las diferentes operaciones que ofrece la web API.
- Para probar la web API se recomienda utilizar aplicativos de testeo de *endpoints* como Swagger, ya que los mismos proporcionan una interfaz gráfica de usuario con la cual se puede realizar las peticiones y obtener las respuestas desde el servidor.

4 REFERENCIAS

[1] M. Learn, «learn.microsoft,» Interfaz IFormFile, [En línea]: https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.iformfile?view=aspnetcore-7.0. [Último acceso: 17 Octubre 2022].

[2] B. Hallet, «StackOverflow,» ¿Que es un DTO?, 08 Abril 2021. [En línea]: https://stackoverflow.com/questions/1051182/what-is-a-data-transfer-object-dto. [Último acceso: 17 Octubre 2022].

[3] Pragimtech, «pragimtech,» Usando AutoMapper en Asp.Net, 2020. [En línea]: https://www.pragimtech.com/blog/blazor/using-automapper-in-asp.net-core/. [Último acceso: 17 Octubre 2022].

[4] P. Joshi, «prateekvjoshi,» URL,URI y URN, 22 Febrero 2014. [En línea]: https://prateekvjoshi.com/2014/02/22/url-vs-uri-vs-urn/. [Último acceso: 19 Enero 2023].

[5] D. Garrido, «Idento,» ¿Como testear una API WEB?, [En línea]: https://www.idento.es/blog/desarrollo-web/como-testear-api-rest/?_adin=02021864894. [Último acceso: 17 Octubre 2022].

[6] Admin, «entityframeworktutorial,» ¿Que es entity Framework?, [En línea]: https://www.entityframeworktutorial.net/what-is-entityframework.aspx. [Último acceso: 17 Octubre 2022].

[7] M. Learn, «learn.microsoft,» Uso de stubs para aislar partes de su aplicación, [En línea]: https://learn.microsoft.com/en-us/visualstudio/test/using-stubs-to-isolate-parts-of-your-application-from-each-other-for-unit-testing?view=vs-2022&tabs=csharp. [Último acceso: 29 Octubre 2022].

[8] M. Learn, «learn.microsoft,» ¿Que es un paquete nuget?, [En línea]: https://learn.microsoft.com/es-es/nuget/what-is-nuget. [Último acceso: 29 Octubre 2022].

[9] S. Chauhan, «dotnettricks,» Conceptos y diferencias entre ASP.NET Web Api y ASP.NET MVC, 04 Septiembre 2022. [En línea]: https://www.dotnettricks.com/learn/webapi/difference-between-aspnet-mvc-and-

aspnet-web-api#:~:text=Asp.Net%20Web%20API%20VS,returns%20only%20data%2C%20not%20view.. [Último acceso: 29 Octubre 2022].

[10] T. Teacher, «tutorialsteacher,» ¿Ques es un web Api?, [En línea]: https://www.tutorialsteacher.com/webapi/what-is-web-api. [Último acceso: 29 Octubre 2022].

[11] M. Learn, «learn.microsoft,» wcf-and-aspnet-web-api, [En línea]: https://learn.microsoft.com/es-es/dotnet/framework/wcf/wcf-and-aspnet-web-api. [Último acceso: 29 Octubre 2022].

[12] Tecsify, «tecsify,» Métodos HTTP, 2022. [En línea]: https://tecsify.com/blog/infografia/metodos-http/. [Último acceso: 29 Octubre 2022].

[13] T. Teacher, «tutorialsteacher,» Controladores, [En línea]: https://www.tutorialsteacher.com/mvc/mvc-controller. [Último acceso: 29 Octubre 2022].

[14] M. Ofiwe, «es.semrush,» Códigos de estados, 01 Noviembre 2021. [En línea]: https://es.semrush.com/blog/codigos-de-estado-http/?kw=&cmp=LM_SRCH_DSA_Blog_ES&label=dsa_pagefeed&Network=g&Device=c&utm_content=641222099980&kwid=dsa-1929298975603&cmpid=19249322807&agpid=145221528700&BU=Core&extid=64565383243&adpos=&gclid=Cj0KCQiAt66eB. [Último acceso: 29 Octubre 2022].

[15] Admin, «csharp.net-tutorials,» Namespaces, [En línea]: https://csharp.net-tutorials.com/es/400/clases/namespaces-espacios-de-nombres/. [Último acceso: 11 Noviembre 2022].

[16] M. Learn, «learn.microsoft,» Namespace microsoft.aspnetcore, [En línea]: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore?view=aspnetcore-6.0#classes. [Último acceso: 19 Enero 2023].

[17] M. Learn, «learn.microsoft,» Namespace Microsoft.EntityFramework, [En línea]: https://learn.microsoft.com/en-

us/dotnet/api/microsoft.entityframeworkcore?view=efcore-7.0. [Último acceso: 19 Enero 2023].

[18] M. Learn, «learn.microsoft,» [En línea]: https://learn.microsoft.com/en-us/dotnet/api/system.identitymodel?view=netframework-4.8. [Último acceso: 19 Enero 2023].

[19] M. Learn, «learn.microsoft,» Namespace System.ComponentModel, [En línea]: https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel?view=net-7.0. [Último acceso: 19 Enero 2023].

[20] M. Learn, «learn.microsoft,» Namespace System.Security, [En línea]: https://learn.microsoft.com/en-us/dotnet/api/system.security?view=net-7.0. [Último acceso: 19 Enero 2023].

[21] M. Learn, «learn.microsoft,» Namespace System.IO, [En línea]: https://learn.microsoft.com/en-us/dotnet/api/system.io?view=net-7.0. [Último acceso: 19 Enero 2023].

[22] Rahul, «myview.rahulnivi,» Json Web Token y Payload, 25 Septiembre 2020. [En línea]: https://myview.rahulnivi.net/asp-net-core-authentication-jwt-aka-json-web-token/. [Último acceso: 19 Diciembre 2022].

[23] K. Shah, «c-sharpcorner,» Autenticación basada en JWT, 15 Septiembre 2021. [En línea]: https://www.c-sharpcorner.com/article/asp-net-core-web-api-5-0-authentication-using-jwtjson-base-token/. [Último acceso: 19 Diciembre 2022].

[24] Admin, «thecodebuzz,» Newtonsoftjson Support in .net, 2020. [En línea]: https://www.thecodebuzz.com/add-newtonsoft-json-support-net-core/. [Último acceso: 19 Diciembre 2022].

[25] T. Naem, «astera,» Conceptos básicos de las API REST, 28 Enero 2020. [En línea]: https://www.astera.com/es/tipo/blog/definici%C3%B3n-de-la-API-de-descanso/. [Último acceso: 19 Diciembre 2022].

[26] M. E. Yıldırım, «medium,» Conceptos basicos de restful-apis, 13 Febrero 2020. [En línea]: https://medium.com/@metyildirim/learn-the-basics-of-restful-apis-b53e5d157c76. [Último acceso: 19 Diciembre 2022].

[27] V. Thakur, «ASP.NET 3.5 Application Architecture and Design,» Octubre 2008. [En línea]: https://www.packtpub.com/product/aspnet-35-application-architecture-and-design/9781847195500. [Último acceso: 21 Diciembre 2022].

[28] D. Ellis, «blog.hubspot,» ¿Que es Swagger?, 26 Julio 2022. [En línea]: https://blog.hubspot.com/website/what-is-swagger. [Último acceso: 19 Enero 2023].

[29] C. S. Boto, «profile,» Que es Kanban y como aplicarlo al desarrollo de software, 14 Septiembre 2020. [En línea]: https://profile.es/blog/que-es-kanban-y-como-aplicarlo-al-desarrollo-de-software/. [Último acceso: 19 Diciembre 2022].

[30] J. Martins, «asana,» ¿Que es kanban?, 10 Octubre 2022. [En línea]: https://asana.com/es/resources/what-is-kanban. [Último acceso: 19 Diciembre 2022].

[31] C. Rodriguez, «apliint,» Principales tecnologias frontend, 15 Marzo 2022. [En línea]: https://apliint.com/2022/03/15/10-principales-tecnologias-frontend-para-usar-en-2022/. [Último acceso: 19 Enero 2023].

[32] D. Beti, «code-maze,» dotnet-multiple-authentication-schemes, 14 Junio 2022. [En línea]: https://code-maze.com/dotnet-multiple-authentication-schemes/. [Último acceso: 21 Diciembre 2022].

5 ANEXOS

ANEXO I. Tablero Kanban

ANEXO II. Procedimiento de instalación y configuración del entorno de desarrollo

ANEXO III. Código del subsistema de clasificación.

ANEXO I

El tablero Kanban completo, así como las actividades establecidas se encuentra disponibles en el siguiente enlace:

Tablero_Kanban_en_Lucidchart

Importante: Para poder visualizar el tablero se requiere de una cuenta en Lucidchart.

ANEXO II

En el CD adjunto se encuentra el documento que sirve de guía para realizar el proceso completo de instalación y configuración del entorno de desarrollo.

ANEXO III

En el CD adjunto se encuentra el código implementado.