Facultad
de
Ciencias

DataShotExtractor: Aplicación web con IA para la extracción y exportación de datos

DataShotExtractor: Web application with AI for data extraction and export

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Alejandro Sanchís Sarabia

Director: Diego García Saiz

Co-Director: Javier Carracedo Esteban

Julio - 2025

Agradecimiento

Quisiera agradecer a la universidad y profesores por estos años de estudio y todo lo aprendido. También, me gustaría agradecer a Diego García Saiz por aceptar ser director de este TFG y ayudarme a la realización de la documentación y defensa.

Además, me gustaría agradecer al Observatorio Tecnológico de HP SCDS por la oferta de este TFG y aceptarme para realizarlo, especialmente a Javier Carracedo Esteban por ayudarme y guiarme durante el desarrollo.

Resumen

En los últimos años, la inteligencia artificial se ha convertido en una herramienta clave e importante dentro del entorno laboral con el propósito de automatizar y facilitar las tareas.

En este proyecto se tiene como objetivo el desarrollo de una aplicación web que, mediante el uso de un modelo de inteligencia artificial, permita la extracción precisa de datos tabulares desde una imagen. Además, el sistema será capaz de encolar los trabajos.

El propósito de este proyecto, sobre el uso de cualquier otra IA, es el de poder ser desplegado de forma local, evitando la salida al exterior de posible información sensible, además de estar preparada para una posible tarea común dentro de la empresa, haciendo que sea más fácil de usar y obtener los resultados deseados sin tener conocimiento sobre el uso de IAs.

Palabras clave

Inteligencia Artificial, Prompt, Open Neural Network Exchange (ONNX), Hugging Face.

Abstract

In recent years, artificial intelligence has become important tools within the workplace, aiming to automate and simplify tasks.

This project aims to develop a web application that, by using an artificial intelligence model, enables the accurate extraction of tabular data from an image. Additionally, the system will be capable of queuing all the tasks.

The purpose of this work, compared to using any other AI, is to allow local deployment, thus avoiding the exposure of potentially sensitive information. It is also designed to be prepared for common tasks within the company, making it easier to use and obtain the desired results without requiring knowledge of how to use AIs.

Keywords

Artificial Intelligence, Prompt, Open Neural Network Exchange (ONNX), Hugging Face.

Índice

1. Introducción y Objetivo

1.1. Introducción

En los últimos años, el desarrollo de la inteligencia artificial ha avanzado lo suficiente para ser incluida dentro del entorno laboral. Esta evolución ha permitido que la IA no solo se utilice para el análisis o predicción, sino también como herramienta que optimice y facilite las cargas de trabajo, mejorando la productividad de la empresa. La repentina aparición de la IA ha hecho que el objetivo número uno de muchas empresas sea la implementación de esta nueva tecnología.

Como no es para menos, HP también presenta el interés de añadir esta nueva tecnología en su entorno laboral con el propósito de seguir manteniéndose competitivos en un mercado que cada vez está invirtiendo más en la IA.

Esta rápida implementación de la IA en la empresa trae consigo múltiples problemas, como la falta de accesibilidad, ya que algunos miembros de la empresa pueden que tengan dificultades al utilizar nuevas tecnologías llevando a una peor productividad. Por otra parte, las IAs más potentes del mercado son externas a la empresa, los cuales implican costos elevados por su uso, además de ser una falta de seguridad derivar el trabajo a terceros. Esta preocupación aumenta cuando los datos que manejos, como imágenes o documentos, son sensibles para la empresa.

1.2. Objetivos

Los objetivos que se persiguen en el desarrollo de la aplicación son los siguientes:

- **Fácil uso de la IA.**

Mediante la implementación de la interfaz de usuario, se permitiría la fácil utilización del modelo por parte del usuario para la tarea que se ha deseado que realice. El uso de la interfaz pretende que el usuario no tenga nunca que interactuar directamente con el modelo de IA y simplemente tenga que seguir los pasos descritos

- **Aislamiento.**

Se busca que el servicio web se pueda desplegar de forma local, sin necesidad de conectarse a internet o derivar el trabajo a un servicio fuera de la empresa. En vez de utilizar algunos de los modelos más populares o potentes, como OpenAI[1] o DeepSeek[2], los cuales suponen un costo de dinero, además de una falta de seguridad para las imágenes con las que se quieren trabajar, hacemos uso de un modelo más pequeño que pueda realizar la tarea que deseamos, o que se puede entrenar para que aprenda dicha tarea, la cual sí que se podría desplegar con facilidad y rapidez de forma local.

[1] OpenAI [14] es una organización de investigación en inteligencia artificial que desarrolla y promueve tecnologías de IA avanzadas.
[2] DeepSeek [13] es una compañía especializada en el desarrollo de modelos de IA a gran escala y de código abierto.

- **Sistema de colas.**

Se pretende que la aplicación web se pueda desplegar en un equipo servidor de la empresa y que cada empleado acceda al servicio de la IA desde su propio equipo. Para manejar múltiples peticiones de los empleados, debido a que el proceso de la IA es lento, dependiendo del equipo, es necesario implementar un sistema de colas.

2. Metodología y Planificación

Durante le desarrollo de proyecto se ha utilizado la metodología de Scrum. La metodología Scrum consiste en dividir la duración del proyecto en diferentes reuniones periódicas en las que se describe las tareas realizadas y las tareas a realizar para la siguiente reunión.

Al comienzo del proyecto, se tomó la decisión de realizar una reunión telemática cada dos semanas, sin contar semanas festivas, en las que se realizaba lo comentado anteriormente. Además, Scrum también cuenta con más reuniones como Daily Scrum, en las que se revisa el progreso cada día, o el Sprint Retrospective, para saber que el proyecto avanzaba de forma adecuada, aunque estos dos tipos de reuniones las realizaba de forma individual.

Las reuniones que dividen cada sprint estaban pensadas para durar entre diez minutos, en caso de que todo haya ido fluido, o 30 minutos, en caso de haber ocurrido algún problema o tener alguna duda.

Lo primero que se había realizado es la obtención de los requisitos funcionales del proyecto con los que sacar las primeras tareas a realizar. Las taras se apuntaban en un bloc de notas haciendo uso de Notepad++, ya que resulta más cómodo que utilizar el Issue Board de la plataforma de Gitlab, además de que el proyecto es individual. En caso de ser más personas, el uso del backlog es más útil.

Por otra parte, se ha hecho uso de los Merge Request y Milestones en Git con el propósito de tener ordenado el desarrollo del proyecto en los diferentes Sprints y facilitar el proceso de documentación.

En la tabla 1 se describe el proceso del proyecto dividido en Sprints.

ID	Fecha	Resultados
S00	6/11–20/11	• Estudio del uso de modelos IA y ONNX. • Creación de programas en Python usado transformadores. • Seguimiento del tutorial de ONNX para implementar phy3 en Python.
S01	21/11–4/12	• Estudio de como implementar modelos de IA en .NET • Implementación de chatbot en .NET
S02	22/01–5/02	• Encontrar modelo de IA (Phi3-vision-128k). • Implementación de aplicación de consola en .NET haciendo uso del modelo.
S03	6/02–26/02	• Añadido README y .csproj del prototipo. • Añadido README y .csproj a los tutoriales. • Añadido manejo de errores en el prototipo.

S04	27/02–12/03	• Implementación de aplicación web en .NET haciendo uso el modelo.
S05	13/03–27/03	• Añadido el método HTTP de POST para subir imágenes al prototipo web.
S06	28/03–9/04	• Añadido método asíncrono que realiza la lógica del modelo y evita la espera de la respuesta del método POST.
S07	10/04–22/04	• Eliminación de la rama develop. • Añadido la configuración eliminando la ruta del modelo del código de la aplicación.
S08	23/04–7/05	• El prototipo web se empieza a mejorar de cara al resultado final dejando de ser un prototipo. • Añadido el sistema de colas de tareas. • Añadida la interfaz del servicio de *ModelService*. • Limpieza del código y añadió comentarios explicativos. • Arreglada la estructura de las carpetas de cara a empezar el front-end.
S09	8/05–21/05	• Creación del front-end. • Añadida la comunicación con el back-end. • Cambiado método GET para devolver un texto plano (string) en vez de un fichero. • Añadida tarea en segundo plano, en el back-end, que se encarga de borrar las repuestas del modelo de IA cada cierto tiempo.
S10	22/05–4/06	• Primeras implementaciones de las pruebas del controlador y de los servicios del back-end. • Añadido lanzamiento de errores de los servicios y *BadRequests* y errores 500 del controlador. • Añadido Toasts. • Añadido Docker. • Añadido algunos cambios de diseño en el front-end. • Cambio de prompt.
S11	5/06–9/06	• Añadido *sesison* de .NET y cambios en el front-end eliminado el manejo de IDs por parte del usuario. • Eliminación de memory leak. • Limpieza del controlador creando más servicios.
S12	10/06–3/07	• finalización de las pruebas y arreglos en el back-end. • Refactorización del front-end añadiendo un servicio y eliminando lógica de la capa de presentación. • Pruebas manuales del front-end. • Pruebas de usabilidad. • Realización de la documentación

Tabla 1

3. Herramientas

En el siguiente apartado, se tratan las diferentes tecnologías utilizadas para el desarrollo de la aplicación además del motivo de porque han sido elegidas.

3.1. Python y PyTorch

Python [1] es un lenguaje de programación interpretado, orientado a objetos, de alto nivel y con semántica dinámica, el cual es fácil de aprender y utilizar, además de tener una gran comunidad que permite aumentar la cantidad de librerías aparte de las oficiales que presenta Python. Es debido a estos motivos que Python es el lenguaje principal para el desarrollo y uso de modelos de Inteligencia Artificial.

PyTorch [2] es una biblioteca optimizada de tensores para aprendizaje profundo que utiliza GPUs y CPUs, basada en Torch, necesaria para el desarrollo y entrenamiento de modelos de inteligencia artificial. Pytorch proporciona:

- Computación de tensores con una aceleración fuerte a través de unidades de procesamiento graficas.
- Redes neuronales profundas construidas en un sistema de diferenciación automática de bases de datos.

El único uso que se le ha dado a Python durante este proyecto ha sido el de encontrar un modelo que hiciese o que se acercase a la tarea que se quería resolver. Con unas pequeñas líneas de código y el modelo deseado, se podía testear diferentes prompts sobre el modelo.

Actualmente ya no es necesario hacer un modelo de IA desde cero, ya que se pueden encontrar muchos de código abierto con libertad para ser usados, y como, durante el testeo de diferentes modelos, se encontró un modelo que presentaba las características necesarias, Pytorch no fue necesario para entrenar ni crear ningún modelo.

3.2. Open Neural Network Exchange (ONNX)

Open Neural Network Exchange [3] es un formato abierto de diseño para representar modelos de aprendizaje automático e inteligencia artificial, el cual permite la ejecución de las IAs puedan ser ejecutadas en otros frameworks. Por ejemplo, el desarrollo de un modelo de Pytorch del cual se obtiene su formato ONNX y se utiliza dentro de .Net o en una aplicación de móvil sin hacer uso de Pytorch.

ONNX es un proyecto de comunidad que tiene como objetivo que se puedan usar los diferentes modelos de inteligencia artificial en diferentes entornos de desarrollo, con el propósito de que cada desarrollador haga uso de las herramientas deseadas.

Debido a que la idea del proyecto es la de realizarlo en ASP.NET, era necesario tener el modelo en formato ONNX.

3.3. Hugging Face

Hugging Face es una empresa tecnológica enfocada en la inteligencia artificial de código abierto. Hugging Face ofrece:

- Modelos de inteligencia artificial preentrenados. Hugging fase ofrece un gran repositorio de modelos de IA de diferentes empresas, como Microsoft, que pueden ser descargados.
- Datasets. Se pueden encontrar conjuntos de datos púbicos con los que poder entrenar cualquier modelo.
- Spaces. Los usuarios o empresas pueden desplegar sus aplicaciones web para poder ser utilizadas libremente.
- Transformadores. Se trata de una librería para Python, la cual permite añadir fácilmente un modelo del repositorio de Hugging Face a la aplicación que se está desarrollando sin tener que descargar el modelo manualmente y con unas pocas líneas de código.

3.4. ASP.NET Core

ASP.NET Core [4] es un framework multiplataforma y de alto rendimiento, desarrollado por Microsoft, para construir aplicaciones web modernas. Este framework de código abierto permite a los desarrolladores crear aplicaciones web, servicios y APIs que pueden ejecutarse en Windows, macOS y Linux. En este proyecto, ASP.NET ha sido utilizado como para la realización del back-end, Web API, así como para la realización del front-end, Blazor.

Se ha elegido este framework en vez de otro por el motivo de aprender un framework nuevo durante el desarrollo de este proyecto, además de aprender a integrar modelos de inteligencia artificial fuera del comúnmente usado Python usando ONNX.

3.4.1. ASP.NET Core Web API (RESTful)

ASP.NET Core Web API es un framework para el desarrollo de servicios web RESTful. Este framework permite crear llamada HTTP API que el usuario puede llegar a realizar. Las peticiones se realizan enviando una solicitud al servidor con uno de los diferentes métodos (GET, POST, DELETE, PUT…) a una URI para poder diferenciarlos.

Se ha elegido Web API para el desarrollo del back-end debido a la experiencia previa obtenida durante el transcurso de la carrera.

3.4.2. Blazor

Blazor [5] es un framework para el desarrollo de front-end basado en HTML, CSS y C# desarrollado por Microsoft. La idea de Blazor es la creación de componentes reusables que se pueden ser ejecutados por el cliente o por el servidor. Además, Blazor no requiere el uso de JavaScript.

Blazor ofrece dos modelos principales:

- Blazor Server. Toda la lógica de la aplicación se ejecuta en el servidor y la comunicación con el cliente se realiza mediante SignalR. Permite ejecutar directamente el código que se encuentra en el back-end sin llamadas API.

- Blazor WebAssembly. Permite la ejecución de tanto en el servidor como en el cliente. Se acerca más a los servicios web tradicionales en los que es necesario realizar las llamadas API al servidor.

Se ha elegido Blazor en vez de otro frameworks más populares, por el motivo de seguir dentro del entorno de Microsoft ASP.NET. Además, debido a que la decisión de realizar el front-end en Blazor fue tomada posteriormente, se ha elegido Blazor WebAssembly.

3.5. Swagger UI

Swagger es una herramienta que nos permite probar todas las llamadas HTTP de la interfaz desarrollada.

Para utilizar Swagger, no es necesario el uso de ninguna aplicación, directamente desde el navegador se accede a la interfaz de Swagger a través de la URL en http://localhost:{port}/swagger/index.html. Swagger nos muestra directamente todos los endpoints, así como los elementos o datos que recibe.

Se ha elegido esta herramienta por encima de otras, como Postman, debido a lo fácil que es de utilizar y lo muy como que es de usar. Simplemente añadiendo una línea de código al *Program.cs* de nuestra API podemos hacer uso de Swagger y sus múltiples ventajas.

3.6. Git

Git se ha convertido en el estándar mundial para el control de versiones. Git [6] es un sistema de control de versiones distribuido, lo que significa que un clon local del proyecto es un repositorio de control de versiones completo. Estos repositorios locales plenamente funcionales permiten trabajar sin conexión o de forma remota. Los desarrolladores confirman su trabajo localmente y sincronizan la copia del repositorio con la del servidor.

Pese a que este trabajo se realiza en solitario, el uso de Git trae múltiples ventajas:

- Copia de seguridad. Mantener el proyecto bajo la seguridad de los servidores de Git en el caso de perder todo el proyecto local.
- Versiones. Poder realizar cambios en nuevas ramas, las cuales se podrían eliminar y volver al estado original si los cambios no resultan satisfactorios.
- Seguir el proceso/Documentación. Mediante el uso de los Pull Request, se puede mantener un seguimiento del proyecto que ayude, posteriormente, a la documentación de este.

El repositorio se puede encontrar aquí:

https://github.com/Zafiro26/UNICAN-DataShotExtractor

3.7. Visual Studio Code

Visual Studio Code es un editor de código gratuito de código abierto desarrollado por Microsoft. No confundir con Visual Studio, otro editor de código, también desarrollado por Microsoft, el cual es totalmente independiente de VSCode.

La mayor ventaja de Visual Studio Code es su sencillez, su capacidad de personalización y su gran cantidad de extensiones. Visual Studio Code permite cambiar muchos aspectos del editor para adaptarse a los gustos del desarrollador. También, se incluye una gran cantidad de extensiones que, además, debido a su gran comunidad, no deja de crecer y que permite adaptar el editor para múltiples funciones diferentes. Es por estos motivos que se ha elegido VSCode como el editor de código a utilizar.

3.8. Docker

Docker [7] es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones. Docker ofrece la capacidad de empaquetar y ejecutar una aplicación en un entorno ligeramente aislado llamado contenedor. Estos contenedores incluyen todo lo que la aplicación necesita para funcionar.

En este proyecto se ha utilizado Docker para el fácil y rápido despliegue de la aplicación. El Docker se encarga de descargar el modelo de IA, la creación de carpetas necesarias y el despliegue de la Web API y el Blazor front-end, mediante una única línea de comandos y los ficheros Docker disponibles en el repositorio.

3.9. LibreWolf

LibreWolf es un navegador web. Es necesario cualquier navegador para probar tanto el front-end de la aplicación, como para probar el back-end mediante el uso de Swagger.

3.10. Miniconda

Conda [8] es un sistema de gestión de paquetes y entornos, desarrollado por Anaconda, que en este proyecto sería usado para el testeo de aplicaciones y modelos dentro de Python.

Exactamente, se ha utilizado Miniconda [9] el cual es una versión gratuita y más pequeña de Anaconda, el cual solo incluye Conda, Python, los paquetes necesarios de los que dependen y otros pocos paquetes útiles.

Para utilizar Conda, primero se crea un entorno de Conda donde se instalan los paquetes necesarios desde el repositorio de Conda, aunque se puede hacer uso de pip.

Haciendo uso de múltiples paquetes por primera vez de Python, además de Pytorch, Conda permitía evitar problema de dependencias, además de poder borrar el entorno y crear uno nuevo desde cero por si surgía cualquier tipo de problema.

3.11. NUnit

NUnit [10] es un framework de .NET para realizar pruebas unitarias. Inicialmente, NUnit nació como un port de Junit de Java, aunque con el tiempo se ha ido reescribiendo con nuevas funcionalidades para más plataformas de .NET.

Existen tres frameworks de pruebas que son los más usados dentro del entorno de .NET:

- MSTest
- NUnit
- xUnit.net

De estos tres frameworks, se ha elegido NUnit debido a su gran parecido con JUnit, framework con el que ya presentaba experiencia de antemano.

4. Descripción del sistema

La aplicación dispondrá de una página principal dividida en dos apartados donde poder realizar las dos actividades necesarias, la subida de la imagen al servidor, la cual inicia el proceso de extracción de datos por parte de la IA, y la devolución del output del modelo de IA.

- Upload. El usuario selecciona el formato al que quiere transformar la table y la imagen, en uno de los formatos aceptados. Una vez seleccionada la imagen, esta se sube inmediatamente al servidor e iniciar el proceso del modelo.
- Output. El usuario selecciona el fichero, de los que ha subido en esa sesión, del cual quiere obtener el output del modelo.

Debido a que el servicio puede ser usado por múltiples usuarios o que simplemente un único usuario suba múltiples imágenes, debido a que el proceso de obtención de los datos del modelo puede llevar varios segundos, es necesario un sistema de colas. Cuando el usuario sube la imagen al servidor, la tarea se añade a la cola. Mientras espera su turno, la imagen se guarda en una carpeta del servidor. Si el usuario intenta obtener el output de su imagen, recibirá un error avisando de que no se encuentra disponible.

Una vez llega el turno de la tarea del usuario, la IA procede a obtener y tabular los datos del usuario en el formato seleccionado. Este output se guarda en un nuevo fichero, dentro de otra carpeta del servidor, desde el que el usuario puede acceder a su contenido. Una vez el proceso termina, la imagen es borrada del servidor.

El sistema no hace uso ni de base de datos ni de inicios de sesión, solo hace uso de sesiones. Se ha pretendido que el servicio sea fácil y rápido de usar. Una vez cerrado el servidor o pasado un tiempo, las imágenes y los outputs del modelo de IA se borran.

5. Requisitos del sistema

5.1. Requisitos funcionales

Los requisitos funcionales, necesarios para que el sistema cumpla el comportamiento deseado, se muestra en la tabla 2.

Id	Nombre	Descripción
RF0	Subida de imagen	El usuario debe poder subir la imagen con la tabla a procesar.
RF1	Selección de formato	El usuario debe de poder seleccionar el formato al que quiere transformar la imagen.
RF2	Extracción y exportación de datos	Mediante el uso de un modelo de inteligencia artificial, se procesará, segmentará y extraerá la información de la imagen de forma automática al formato elegido por el usuario.
RF3	Cola de tareas	El sistema debe contar con la implementación de colas de tipo FIFO, con el propósito de manejar múltiples solicitudes.
RF4	Guardado de imagen	El sistema debe de contar con la capacidad de guardar las imágenes, mientras la tarea esta encola o procesándose.
RF5	Guardado del output del modelo IA	El sistema debe de contar con la capacidad de guardar el output de la IA, de forma temporal, para que el usuario pueda acceder a la información.
RF6	Obtener output del modelo IA	El usuario debe de poder obtener el output del modelo de cualquiera de las imágenes que ha subido.

Tabla 2

5.2. Requisitos no funcionales

Los requisitos no funcionales, los cuales son aquellos que describen el sistema, pero no su funcionamiento, se describen en la tabla 3.

ID	Nombre	Descripción
RnF0	.NET Session	El sistema hace uso de las sesiones de .NET para identificar a los usuarios.
RnF1	Interfaz sencilla	Interfaz fácil de usar e intuitiva.
RnF2	Cola de tareas	La cola de tareas debe de soportar un máximo de 100 tareas en espera.
RnF3	Compatibilidad con navegadores	La aplicación debe de funcionar en los navegadores más populares (Google Chrome, Microsoft Edge, Mozilla Firefox).

Tabla 3

6. Casos de Uso

En el siguiente apartado se tratan los casos de uso del sistema, primero se muestra el diagrama y después las tablas. Los casos de uso [11] definen lo que los usuarios o roles están haciendo en la solución.

6.1. Diagrama de Casos de Uso

En la figura 1, se muestra los casos de uso del usuario.

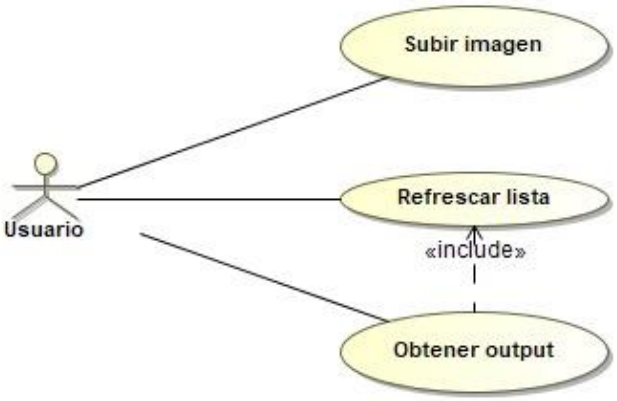


Figura 1

6.2. Tablas de Casos de Uso

En este apartado se tratan los casos de uso de subir imagen, tabla 4, refrescar lista, tabla 5, y obtener output, tabla 6.

Id + Nombre	**US000 – Subir Imagen**
Actor Principal	Usuario
Actor Secundario	-
Descripción	El usuario sube la imagen con el formato deseado para que sea procesada.
Objetivo	Subir la imagen al servidor.
Evento de activación	Seleccionar el botón de “Browse…” y seleccionar la imagen.
Precondición	El usuario selecciona un formato.
Garantías Si Éxito	La imagen se guarda en el servidor, devolviendo una notificación para informar al usuario.
Garantía Mínima	-
Escenario Principal	1. El usuario accede a la página principal de la web. 2. El usuario selecciona el formato, de una lista, al que quiere transformar la imagen. 3. El usuario selecciona el botón de “Browse…”. 4. El usuario selecciona la imagen deseada.

	5. El sistema envía la imagen al servidor. 6. El servidor procesa que todos los parámetros sean correctos. 7. El sistema muestra una notificación con la respuesta del servidor.
Extensiones	6.a. Imagen no valida. a.1. El usuario selecciona imagen con extensión no valida. a.2. El sistema notifica del error al usuario. a.3. La ejecución del caso de uso vuelve al punto 3. 6.b. Servidor API está caído. a.1. El sistema envía la imagen al servidor. a.2. No hay respuesta por parte del servidor. a.3. El sistema notifica del error al usuario. a.4. La ejecución del caso de uso vuelve al punto 3. 6.c. Error por parte del servidor. a.1. El sistema envía la imagen al servidor. a.2. El servidor devuelve error. a.3. El sistema notifica del error al usuario. a.4. La ejecución del caso de uso vuelve al punto 3.

Tabla 4

Id + Nombre	**US001 – Refrescar lista**
Actor Principal	Usuario
Actor Secundario	-
Descripción	El usuario refresca la lista de imágenes que ha subido.
Objetivo	Obtener la lista de imágenes subidas al servidor en esa sesión.
Evento de activación	Seleccionar el botón de “Refresh List”.
Precondición	-
Garantías Si Éxito	La lista se actualiza con las imágenes subidas por el usuario en esa sesión.
Garantía Mínima	
Escenario Principal	1. El usuario accede a la página de “Output”. 2. El usuario pulsa el botón de “Refresh List”. 3. El servidor devuelve la lista de imágenes subidas en esa sesión”. 4. El sistema actualiza la lista.
Extensiones	3.a. Servidor API está caído. a.1. No hay respuesta por parte del servidor. a.2. El sistema notifica del error al usuario. a.3. La ejecución del caso de uso vuelve al punto 1. 3.a. Error por parte del servidor. a.1. El servidor devuelve error. a.2. El sistema notifica del error al usuario. a.3. La ejecución del caso de uso vuelve al punto 1.

Tabla 5

Id + Nombre	US002 – Obtener Output
Actor Principal	Usuario
Actor Secundario	-
Descripción	El usuario obtiene la respuesta del modelo IA.
Objetivo	Obtener la respuesta del modelo IA de la imagen subida al servidor previamente.
Evento de activación	Seleccionar el botón de “Get Output”.
Precondición	El usuario seleccionar refresca la lista y selecciona una de las imágenes subidas.
Garantías Si Éxito	Se muestra al usuario la respuesta del modelo de IA.
Garantía Mínima	-
Escenario Principal	1. El usuario accede a la página de “Output”. 2. <<include>> US001 – Refrescar lista. 3. El usuario selecciona la imagen deseada de la lista. 4. El usuario pulsa el botón de “Get Output”. 5. El servidor devuelve el output del modelo. 6. El sistema muestra el texto. 7. Se notifica al usuario.
Extensiones	5.a. Servidor API está caído. a.1. No hay respuesta por parte del servidor. a.2. El sistema notifica del error al usuario. a.3. La ejecución del caso de uso vuelve al punto 4. 5.b. Error por parte del servidor. b.1. El servidor devuelve error. b.2. El sistema notifica del error al usuario. b.3. La ejecución del caso de uso vuelve al punto 4. 5.c. Imagen no está terminada de procesar (Output no existe). c.1. El servidor devuelve error. c.2. El sistema notifica del error al usuario. c.3. La ejecución del caso de uso vuelve al punto 4.

Tabla 6

7. Diseño Software

En el siguiente apartado se va a tratar el diseño y estructura de la aplicación.

7.1. Arquitectura de la aplicación

En este apartado se va a tratar la estructura de la aplicación. Se optado por una estructura de tres capas, cada uno con un diferente propósito. Las capas son tres:

- Capa de presentación. Esta capa se encarga de las interacciones con el usuario. Esta capa incluye la interfaz de la página web con la que el usuario realiza todas

las acciones, además de incluir los controladores del servidor API que procesa las peticiones HTTP del usuario.

- Capa de negocio. Esta capa incluye toda la lógica de la aplicación. Esta es la capa intermedia que se encarga de realizar las tareas dependiendo de las peticiones del usuario e interactúa con los datos guardados si fuese necesario.
- Capa de persistencia. Esta capa se encarga de almacenar los datos. Esta aplicación no hace uso de una base de datos, si no que se almacenan de forma temporal en unas carpetas del servidor.

En la figura 2 se puede ver una descripción grafica.

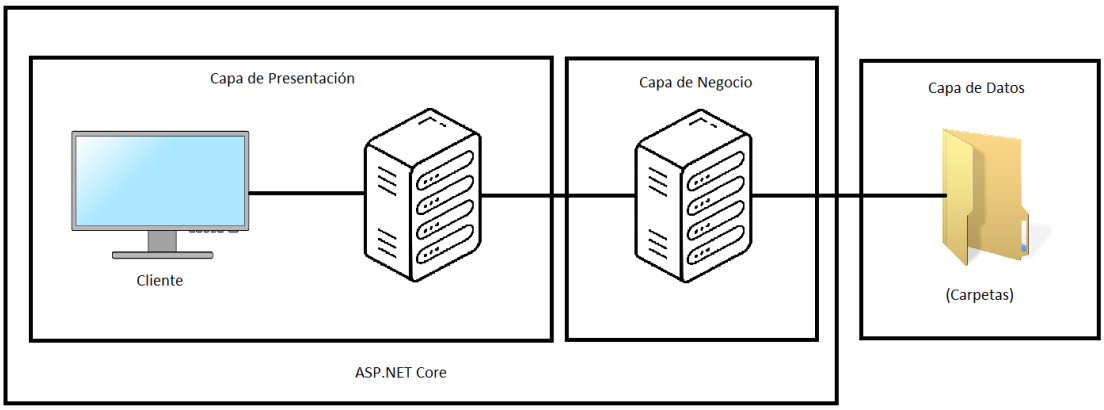


Figura 2

En el siguiente apartado se trata el front-end, el back-end y la persistencia temporal.

7.2. Front-end (Blazor)

El front-end de la aplicación se ha realizado en Blazor para mantener la aplicación completamente en el entorno de Microsoft .NET.

El front-end de Blazor esta divido en los servicios y las páginas:

- Paginas. Las páginas incluyen las interfaces que verá el usuario cuando haga uso de la web. Las páginas se realizan en ficheros *.razor*, estos ficheros utilizan código en HTML para el diseño. Además, en los ficheros *razor* se puede escribir código en C# y es en este apartado donde se llama a los métodos de los servicios.
- Servicios. Los servicios incluyen la lógica del front-end. En los servicios se incluye las comunicaciones con el servidor API además de procesar las respuestas.

7.3. Back-end (ASP.NET Core Web API)

ASP.NET Core Web API, el framework usado para el back-end, está dividido en dos bloques:

- Controlador. Se encarga de recibir las peticiones HTTP siguiendo la estructura de API RESTful, además de devolver las respuestas al cliente y llamar a los servicios.

- Servicios. Es donde se encuentra toda la lógica de negocio. Si es necesario, retorna las respuestas al controlador y se comunica con la capa de persistencia.

En la tabla 7 se pueden ver los métodos el controlador y en la figura 3 se puede un diagrama que muestra la composición del back-end.

Recursos	URI	Métodos
Modelo IA	/api/model/table	POST (Query:type)
		GET (Query:id)
	/api/model/table/ids	GET
	/api/model/test	GET

Tabla 7

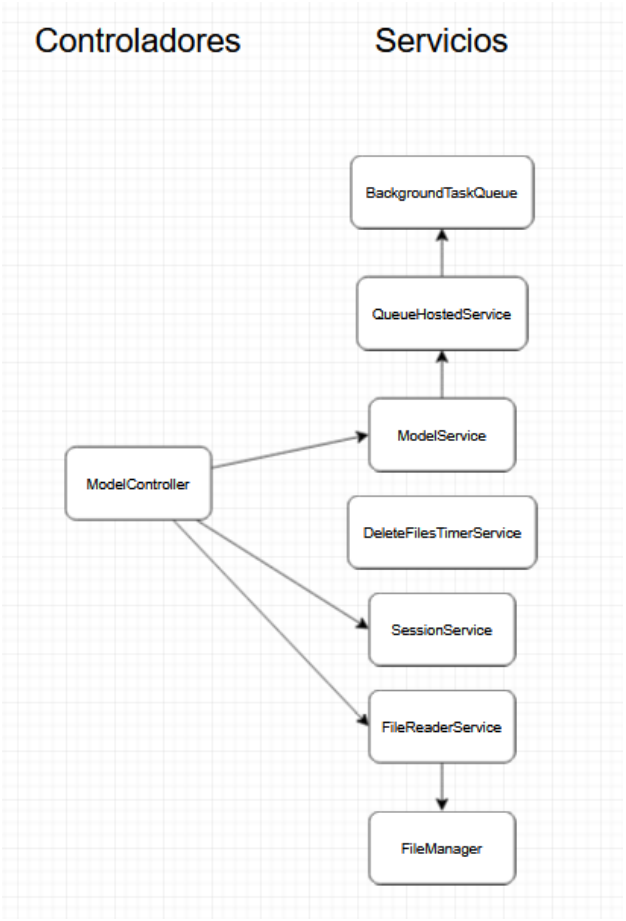


Figura 3

Los servicios API RESTful suelen incluir una capa de datos, capa que se encarga de la comunicación con la base de datos, en esta aplicación no existe debido a que los datos (imágenes y respuestas del modelo de IA) son temporales y se guardan en carpetas del servidor.

7.4. Session .NET

HTTP es un protocolo sin estado. Por defecto, las solicitudes HTTP son mensajes independientes que no retienen datos del usuario. Debido a que es necesario recordar

que imágenes ha subido el usuario es necesario incluir algo que permita preservar datos.

De las opciones disponibles que presenta .NET, en la aplicación se hace uso del middleware de sesiones. La sesión está pensada para durar hasta que el navegador se cierre, sin *cookies*, perfecto para la idea del proyecto.

La sesión de .NET funciona mediante el uso de un identificador:

1. El usuario hace una petición HTTP.
2. El servidor comprueba si hay un ID de sesión. En caso de no existir el ID, el servidor genera un nuevo ID y lo incluye en la respuesta que llega al usuario.
3. El servicio web incluye el ID en cada solicitud que realiza.

En caso de no influir el identificador en la cabecera de la solicitud, se creará una nueva sesión en cada solicitud.

8. Desarrollo del Proyecto.

8.1. Prototipos de Testeo.

Una vez terminada la fase de informase, tocaba comenzar con la implementación y desarrollo de código del proyecto. Antes de comenzar, se realizó algunos programas de consola en Python y .NET para el testeo de los modelos y funcionamiento, así como encontrar el modelo IA que se va a usar para el producto final.

Tanto la aplicación de Python como la de .NET son muy básicas.

8.1.1. Python y Pytorch.

Durante el comienzo del proyecto, se realizaron un par de pequeñas aplicaciones de consolar en Python con tres objetivos en mente:

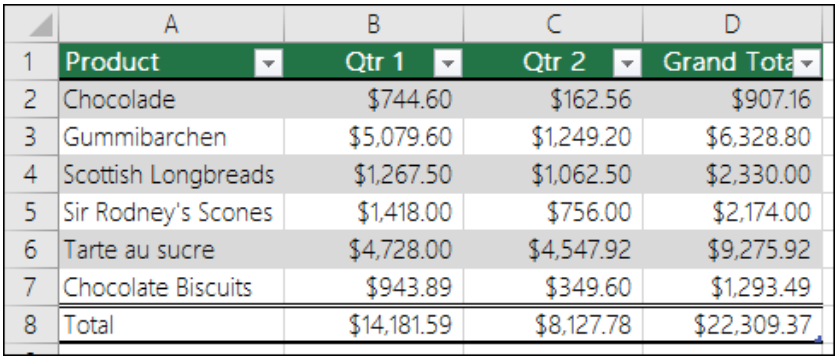
- Aprender a usar e integrar los modelos de inteligencia artificial en Python.
- Comprobar el funcionamiento correcto de los modelos de IA dentro de mi equipo.
- Encontrar un modelo de IA que se acerque o que haga el objetivo deseado.

Durante este tiempo de prototipos de testeo, se encontró el modelo IA que sería usado para el proyecto, el modelo Phi3 de Microsoft, exactamente, la versión de 128K de longitud de contexto (en tokens) y “Vision”, la cual permite añadir imágenes al prompt y trabajar con ellas. Además, este modelo ya presentaba una versión en formato ONNX, lo cual facilita su integración de .NET.

Este modelo se puede encontrar en el repositorio de Hugging Face en https://huggingface.co/microsoft/Phi-3-vision-128k-instruct-onnx.

La aplicación de Python, la cual sigue uno de los tutoriales proporcionados por ONNX, pregunta por la ruta donde se encuentra la imagen en el equipo y también la acción que se quiere que se realice. A continuación, se muestra un ejemplo.

1. Cuando la aplicación lo pida, pasamos la ruta de la imagen, en este caso la imagen es una captura de una tabla de Excel, como la de la figura 4.

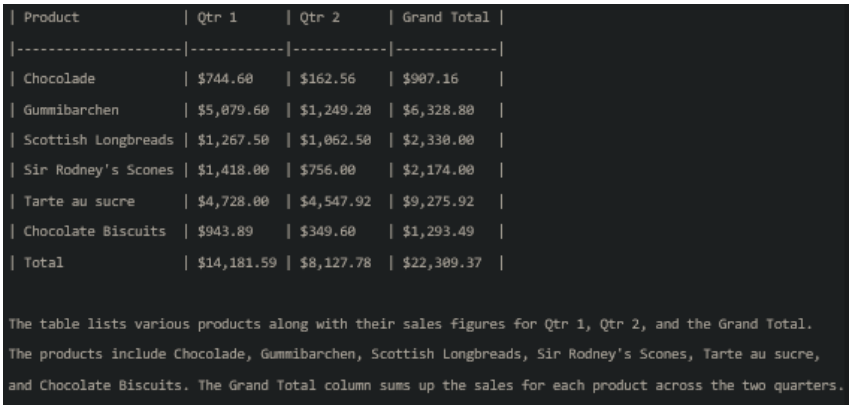


	A	B	C	D
1	Product	Qtr 1	Qtr 2	Grand Tota
2	Chocolade	$744.60	$162.56	$907.16
3	Gummibarchen	$5,079.60	$1,249.20	$6,328.80
4	Scottish Longbreads	$1,267.50	$1,062.50	$2,330.00
5	Sir Rodney's Scones	$1,418.00	$756.00	$2,174.00
6	Tarte au sucre	$4,728.00	$4,547.92	$9,275.92
7	Chocolate Biscuits	$943.89	$349.60	$1,293.49
8	Total	$14,181.59	$8,127.78	$22,309.37

Figura 4

2. Escribimos la acción que queremos realizar. En este caso será “Convert this image to markdown format”.
3. Esperamos a que el modelo realice la tarea.

El resultado acaba siendo la imagen de la tabla que le hemos pasado al modelo en formato markdown como la de la figura 5.



```
| Product             | Qtr 1      | Qtr 2      | Grand Total |
|---------------------|------------|------------|-------------|
| Chocolade           | $744.60    | $162.56    | $907.16     |
| Gummibarchen        | $5,079.60  | $1,249.20  | $6,328.80   |
| Scottish Longbreads | $1,267.50  | $1,062.50  | $2,330.00   |
| Sir Rodney's Scones | $1,418.00  | $756.00    | $2,174.00   |
| Tarte au sucre      | $4,728.00  | $4,547.92  | $9,275.92   |
| Chocolate Biscuits  | $943.89    | $349.60    | $1,293.49   |
| Total               | $14,181.59 | $8,127.78  | $22,309.37  |

The table lists various products along with their sales figures for Qtr 1, Qtr 2, and the Grand Total.
The products include Chocolade, Gummibarchen, Scottish Longbreads, Sir Rodney's Scones, Tarte au sucre,
and Chocolate Biscuits. The Grand Total column sums up the sales for each product across the two quarters.
```

Figura 5

8.1.2. Microsoft .NET.

Una vez terminadas todas las pruebas dentro de Python y Pytorch, se decidió la realización de pruebas dentro del entorno de .NET con dos principales objetivos:

- Aprender a usar e integrar los modelos de inteligencia artificial en .NET.
- Comprobar el correcto funcionamiento de los modelos de inteligencia artificial en formato ONNX dentro de .NET.

Al igual que en la aplicación de testeo de Python, en .NET también se ha realizado una pequeña aplicación de consola para comprobar los objetivos nombrados. Para aprender

a integrarlo se ha seguido un pequeño tutorial que se puede encontrar en los Dev Blogs de Microsoft. Esta aplicación se trata de un pequeño chatbot por consola.

8.2. Prototipo .NET

El objetivo de este paso del desarrollo era el de transformar la aplicación de testeo de Python a una aplicación, también de consola, en .NET haciendo uso de ONNX, y si era posible, reutilizar este prototipo en la aplicación web.

Este prototipo es una recreación uno a uno del código de Python previamente desarrollado. Para poder implementar los modelos de Phi de Microsoft en .NET, es necesario hacer uso de los paquetes nuget *Microsoft.ML.OnnxRuntime*. Estos paquetes permiten todo lo necesario para utilizar el modelo de ONNX como, cargar el modelo, la creación del token y del prompt y la ejecución del modelo.

8.3. Desarrollo del Back-end

En este apartado se tratarán el desarrollo del back-end de la aplicación, así como decisiones y cambios que han surgido.

8.3.1. Capa de Negocio / Servicios

En este apartado se va a tratar la capa de negocio de la aplicación. En la capa de negocio es donde podemos encontrar toda la lógica del programa.

8.3.1.1. ModelService

Uno de los objetivos del prototipo de .NET era el de poder reutilizar el código de la aplicación de consola en la lógica de la aplicación web. En la figura 6 podemos ver la lógica principal para utilizar el modelo IA haciendo uso del nuget mencionado anteriormente. Además, este método devuelve un Task, ya que es necesario añadirlo a la cola FIFO en caso de haber múltiples solicitudes seguidas.

```
private ValueTask GetTableTask(string imagePath, string type, string id, string modelPath,
CancellationToken token)
{

     StringBuilder sb = new StringBuilder();
     using Microsoft.ML.OnnxRuntimeGenAI.Model model = new
Microsoft.ML.OnnxRuntimeGenAI.Model(modelPath);

     _logger.LogInformation("Loading model");
     _logger.LogInformation("Model Loaded");
     using var processor = new MultiModalProcessor(model);
     using var tokenizer = processor.CreateStream();

     string[] images = new string[1];
     images[0] = imagePath;                    //Newest version of OnnxRuntimeGenAI accepts
multiple images.
     Images image = Images.Load(images);
     var fullPrompt = "<|user|>\n<|image_1|>\nDo only convert this image to " + type + " format,
do not add any description or summary of what you just did, do not add any more comments apart
from the conversion. Do not add empty lines.<|end|>\n<|assistant|>\n";

     using var generatorParams = new GeneratorParams(model);
     using var input = processor.ProcessImages(fullPrompt, image);
     generatorParams.SetInputs(input);
     generatorParams.SetSearchOption("max_length", 3072);

     using var generator = new Generator(model, generatorParams);
     _logger.LogInformation("Start printing");
     while (!generator.IsDone())
     {
          generator.ComputeLogits();
          generator.GenerateNextToken();
          var outputTokens = generator.GetSequence(0);
          var output = tokenizer.Decode(outputTokens[^1]);        //Last element of the token
          sb.Append(output);
     }
     writeToFile(sb.ToString(), id);
     image.Dispose();
     File.Delete(imagePath);     //The image is deleted when the AI model finishes
     _logger.LogInformation("Finised printing");
     tokenizer.Dispose();
     processor.Dispose();
     model.Dispose();
     return ValueTask.CompletedTask;
}
```

Figura 6

El método coge la imagen asociada al identificador, el cual se encuentra en una de las carpetas del servidor y que será eliminado cuando el método termine. Por otra parte, el output del modelo se guarda en un string el cual se guardará en un fichero dentro de otra carpeta del servidor.

También, está el método, figura 7, al que llama el controlador para inicial al modelo, este método se encarga de que todos los parámetros sean correctos y de añadir la tarea a la cola FIFO. El método lanza error si alguno de los parámetros es nulo o está vacío.

```
public void GetTable(string imagePath, string type, string id)
{

     if (string.IsNullOrEmpty(imagePath))
     {
          _logger.LogInformation("No image");
          throw new ArgumentException(imagePath, "Image path is null or empty");
     }

     if (string.IsNullOrEmpty(type))
     {
          _logger.LogInformation("No type");
          throw new ArgumentException(imagePath, "Type is null or empty");
     }

     if (string.IsNullOrEmpty(id))
     {
          _logger.LogInformation("No Id");
          throw new ArgumentException(imagePath, "Id is null or empty");
     }

     string modelPath = ${_model.ModelPath}";

     if (string.IsNullOrWhiteSpace(modelPath))
     {
          _logger.LogInformation("Error: Model path is not provided. Exiting
program...");
          throw new ArgumentException(modelPath, "Model path is not provided");
     }

     _taskQueue.QueueBackgroundWorkItemAsync((token) => GetTableTask(imagePath, type,
id, modelPath, token));
}
```

Figura 7

8.3.1.2. FIFO Queue

Para la implementación de la cola FIFO de tareas, es necesario el desarrollo de dos clases que se estarán ejecutando en segundo plano desde que comienza el servicio web.

Por una parte, está la clase de *BackgroundTaskQueue*. Esta clase hace uso de la clase Channel para la creación de una cola FIFO, en el que se añaden Tasks, que en este caso son solo la lógica del modelo de IA. Cuando se crea la cola se elige el tamaño de esta y se elige su funcionamiento cuando llega al límite. La clase solo incluye dos métodos, añadir una tarea a la cola y quitar una tarea de la cola.

Los métodos de la clase *BackgroundTaskQueue se pueden ver en la figura 8.*

```
public BackgroundTaskQueue(int capacity)
{
     BoundedChannelOptions options = new(capacity)
     {
          FullMode = BoundedChannelFullMode.Wait
     };
     _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>(options);
}

public async ValueTask QueueBackgroundWorkItemAsync(
     Func<CancellationToken, ValueTask> workItem)
{
     ArgumentNullException.ThrowIfNull(workItem);

     await _queue.Writer.WriteAsync(workItem);
}

public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
     CancellationToken cancellationToken)
{
     Func<CancellationToken, ValueTask>? workItem =
          await _queue.Reader.ReadAsync(cancellationToken);

     return workItem;
}
```

Figura 8

Por otra parte, está la clase de *QueueHostedService*. Esta clase se encarga, constantemente, de comprobar la cola y cuando hay una tarea disponible en la cola, elimina la tarea de la cola haciendo uso del método de *DequeueAsync* y ejecuta la tarea en segundo plano. Los métodos se pueden ver en la figura 9.

```
protected override Task ExecuteAsync(CancellationToken stoppingToken)
{

     return ProcessTaskQueueAsync(stoppingToken);
}

private async Task ProcessTaskQueueAsync(CancellationToken stoppingToken)
{
     while (!stoppingToken.IsCancellationRequested)
     {
          try
          {
               Func<CancellationToken, ValueTask>? workItem =
                    await TaskQueue.DequeueAsync(stoppingToken);

               await workItem(stoppingToken);
          }
          catch (OperationCanceledException)
          {
               // Prevent throwing if stoppingToken was signaled
          }
          catch (Exception ex)
          {
               _logger.LogError(ex, "Error occurred executing task work item.");
          }
     }
}

public override async Task StopAsync(CancellationToken stoppingToken)
{
          _logger.LogInformation(
          $"{nameof(QueuedHostedService)} is stopping.");

     await base.StopAsync(stoppingToken);
}
```

Figura 9

En resumen, el funcionamiento de la ejecución del modelo de IA consiste en que la tarea se añade a la cola haciendo uso del método *QueueBackgroundWorkItemAsync*. Cuando se detecta una nueva tarea, está se elimina de la cola y se ejecuta en segundo plano. Una vez finalizado, el ciclo se repite.

8.3.1.3. DeleteFilesTimerService

Como ya se ha mencionado anteriormente, no existe una base de datos, las imágenes están pensadas para ser eliminadas cuando el modelo de IA termina el proceso, y la respuesta del modelo para dentro de un tiempo determinado.

Para realizar el borrado de los ficheros con la salida del modelo de IA, se ha creado una nueva clase que se estará ejecutando en segundo plano y que hace uso de una programación asíncrona. Los métodos se pueden ver en la figura 10.

```
protected override Task ExecuteAsync(CancellationToken stoppingToken)
{
     return DeleteFilesAsync(stoppingToken);
}

private async Task DeleteFilesAsync(CancellationToken stoppingToken)
{
     while (!stoppingToken.IsCancellationRequested)
     {
          try
          {
               if (Directory.Exists(textFolder))
               {
                    _logger.LogInformation("Start deleting files");
                    DateTime now = DateTime.Now;
                    string[] files = Directory.GetFiles(textFolder);

                    foreach (string file in files)
                    {
                         DateTime age = File.GetLastWriteTime(file);
                         if ((now - age) > fileAge)
                         {
                              File.Delete(file);
                              _logger.LogInformation("Deleting file");
                         }
                    }
               }
          }
          catch (Exception)
          {
               _logger.LogError("Error while deleting files");
          }
          await Task.Delay(waitTime, stoppingToken);
     }
}
```

Figura 10

Cuando la clase se crea, se ejecuta el método de *DeleteFilesAsync*, el cual es un método que ejecuta en bucle el borrado de los ficheros. En su primera iteración no borra ningún fichero, pero se queda en espera con el método *Delay* de la clase Task. Hay dos variables importantes:

- *waitTime*. Esta variable indica en minutos cada cuanto tiempo se va a comprobar los ficheros.
- *fileAge*. Esta variable indica en hora, el tiempo de vida máximo del fichero.

Cuando se vuelve a ejecutar la tarea, comprueba todos los ficheros de texto del output del modelo y si superan la edad máxima, se borran. El toquen de cancelación sirve para que se deje de ejecutar el bucle si se decide bloquear el borrado de ficheros.

8.3.1.4. Session

Para la gestión de las sesiones dentro de la aplicación, se ha implementado una nueva clase. Esta clase tiene como objetivo principal almacenar y recuperar los datos asociados a la sesión del usuario que, en este caso, se tratan de los IDs de las imágenes subidas.

La clase hace uso de la interfaz *IHttpContextAccessor*, que permite acceder a la propiedad de *Session*, sesión e información que dura el tiempo elegido dentro del fichero *Program.cs*.

SessionService presenta dos métodos, uno para añadir IDs a la lista y otro método para devolver la lista de IDs:

- *AddToSessionList* (Figura 11). Este método se encarga de añadir el nuevo ID de la imagen a la lista asociada a la sesión pasada como parámetro. En el caso de tratarse de la primera imagen, primero se crea una lista vacía y después se añade el ID. Antes de acceder a la información de la sesión, es necesario deserializarla. La *seassonKey* sirve para identificar la información de la sesión a la que se quiere identificar, aunque en este caso, la sesión solo se utiliza para almacenar la lista de IDs. Este método lanza excepción en caso de no existir la sesión o de que el identificador de la imagen sea nulo o esté vacío.
- *SessionList* (Figura 12). Este método se encarga de devolver la lista de la sesión asociada. Nuevamente, se hace uso de la *sessionKey* para identificar la información de la sesión deseada. Este método lanza excepción en caso de no existir la sesión.

```
public void AddToSessionList(string sessionKey, string id)
{
     var session = _httpContextAccessor.HttpContext?.Session;
     if (session is null)
     {
          throw new InvalidOperationException("Session not available");
     }

     if (string.IsNullOrEmpty(id))
     {
          throw new ArgumentException("ID null or empty");
     }

     List<string>? list;
     var sessionData = session.GetString(sessionKey);
     if (string.IsNullOrEmpty(sessionData))
     {
          list = new List<string>();
     }
     else
     {
          list =
System.Text.Json.JsonSerializer.Deserialize<List<string>>(sessionData);
          if (list is null)
          {
               list = new List<string>();
          }
     }

     list.Add(id);
     session.SetString(sessionKey, System.Text.Json.JsonSerializer.Serialize(list));
}
```

Figura 11

```
public List<string> SessionList(string sessionKey)
{

     var session = _httpContextAccessor.HttpContext?.Session;
     if (session is null)
     {
          throw new InvalidOperationException("Session not available");
     }

     var sessionData = session.GetString(sessionKey);

     if (string.IsNullOrEmpty(sessionData))
     {
          return new List<string>();
     }

     List<string>? list =
System.Text.Json.JsonSerializer.Deserialize<List<string>>(sessionData);

     if (list is null)
     {
          return new List<string>();
     }

     return list;
}
```

Figura 12

8.3.1.5. *FileReaderService*

Para juntar y eliminar toda la lógica de lectura de ficheros se ha creado la clase de *FileReaderService*, con el propósito de eliminar líneas de código del controlador y

facilitar la etapa de testeo del controlador. Esta clase se encarga, en su totalidad, de la lectura de ficheros de texto, que en esta aplicación sería la respuesta del modelo de inteligencia artificial.

La clase solo presenta un único método, *ReadFileFromPath*, figura 14, el cual se encarga de, a través de la ruta de un fichero, obtener su contendido y retornarlo como un string. Para ello, el método hace uso de la clase *SteamReader*, añadiendo cada línea del fichero a un *StringBuilder*, junto a un final de línea.

En este caso, no se accede directamente a la clase *SteamReader*, ya que no es posible realizar mocks de este. En su lugar, se ha creado una nueva clase, denominada *IFileManager*, figura 13, la cual expone le método *SteamReader*, permitiendo realizar el mock y testear la clase.

```
public class FileManager : IFileManager
{
    public StreamReader StreamReader(string path)
    {
        return new StreamReader(path);
    }
}
```

Figura 13

```
public string ReadFileFromPath(string path)
{

    if (string.IsNullOrWhiteSpace(path))
    {
        throw new ArgumentException(path, "File path is null or empty");
    }

    StringBuilder sb = new StringBuilder();

    try
    {
        using (var sr = _fileManager.StreamReader(path))
        {
            while (sr.Peek() >= 0)
            {
                sb.Append(sr.ReadLine());
                sb.Append(Environment.NewLine);
            }
        }
    }
    catch (Exception ex)
    {
        throw new IOException("Failed reading the file", ex);
    }
    return sb.ToString();
}
```

Figura 14

8.3.2. Capa de presentación / Controlador

En este apartado se tratarán los controladores, o capa de presentación, del sistema, la cual se encarga de las peticiones HTTP del usuario.

Este proyecto solo cuenta con una única clase de controlador, *ModelController*, que se encarga de la subida de imágenes y las devoluciones del output del modelo de IA del usuario.

Para comenzar, la clase cuenta con el uso de inyección de dependencias de los servicios tratados anteriormente en el apartado de capa de negocio. En forma de resumen, los servicios son:

- *IModelService*. Servicio que incluye toda la lógica del Modelo de IA.
- *IFileSystem*. Servicio perteneciente al paquete no oficial, perteneciente a TestableIO, de TestableIO.System.IO.Abstractions.Wrappers. Necesario para la realización de pruebas de los métodos estáticos de System.IO.
- *ISessionService.* Servicio que incluye el manejo de la sesión del usuario.
- *IFileReaderService*. Servicio que incluye la lectura de ficheros de texto.

En el siguiente apartado se describe el propósito y código de los diferentes métodos HTTP de la tabla.

8.3.2.1. POST /api/model/table

Este método permite al usuario subir una única imagen junto al formato al formato al que lo quiere transformar. El formato se recibe en forma de query.

Primero, se comprueba que todos los parámetros sean correctos y en caso de ser incorrectos, se retorna un error de tipo 400. De los parámetros se comprueba:

- Que se haya enviado la imagen junto a la petición.
- Que realmente se trate de una imagen de extensión jpeg, png o jpg.
- Que el formato al que se quiere transformar la imagen sea uno de los aceptados.

Después se asigna al archivo de la imagen un identificador de tipo GUID. Es este el valor que se usará para poder recuperar la respuesta del modelo cuando el usuario lo solicite. GUID es la implementación de Microsoft de UUID [12], el cual es un número de 16 bytes generado aleatoriamente siguiendo la estructura de una palabra de cuatro bytes, tres palabras de dos bytes y un bloque de seis bytes.

Seguido de ello, la imagen se guarda en una de las carpetas del servidor con el nombre del GUID haciendo uso del método *CopyTo* de la interfaz de *IFormFile*. Después, se comprueba que el fichero existe y que se ha creado correctamente. En caso de ocurrir un error durante la creación del fichero, se devuelve un error de tipo 500.

Posteriormente, se hace uso del servicio para poder añadir el string GUID de la imagen creada y añadirlo a la lista de la sesión asociada.

Para finalizar, se hace uso del servicio del modelo que inicializa el proceso de añadir la tarea el modelo de IA a la cola FIFO. En caso de enviar un error antes de añadir la tarea a la cola, se devuelve un error de tipo 500.

El método se puede ver en la figura 15.

```
[HttpPost("Table")]
public IActionResult PostImage([FromForm] IFormFile image, [FromQuery]
string type = "Markdown")
{
     // No image uploaded
     if (image == null || image.Length == 0)
     {
          return BadRequest("Error; No file.");
     }

     // File is not an image
     if (image.ContentType != "image/jpeg" && image.ContentType !=
"image/png" && image.ContentType != "image/jpg")
     {
          return BadRequest("Error; File is not jpeg, png or jpg.");
     }

     // Type is not accepted
     if (!types.Contains(type))
     {
          return BadRequest("Error; Type is not accepted");
     }

     string Id = Guid.NewGuid().ToString();
     string fileName = $"{Id}";
     string filePath = Path.Combine(imageFolder, fileName);

     using (var stream = new FileStream(filePath, FileMode.Create))
     {
          image.CopyTo(stream);
     }


     // Check if image was saved
     if (!System.IO.File.Exists(filePath) || new
FileInfo(filePath).Length == 0)
     {
          return StatusCode(StatusCodes.Status500InternalServerError,
"Image not saved");
     }

     //Session to store the IDs of the session.
     _session.AddToSessionList(sessionKey, Id);

     try
     {
          _model.GetTable(filePath, type, Id);
     }
     catch (Exception)
     {
          return StatusCode(StatusCodes.Status500InternalServerError,
"Error with internal model");
     }

     return Ok(new
     {
          message = "Upload successful!",
          id = Id
     });
}
```

Figura 15

8.3.2.2. GET /api/model/table

Este método llamado *GetMessage*, figura 16, permite obtener el texto generado por el modelo de IA a partir del GUID de la imagen. El ID se recibe en forma de query.

Primero, se comprueba que todos los parámetros sean correctos y en caso de ser incorrectos, se retorna un error de tipo 400. En este método solo se recibe un único parámetro, el id de la imagen, y se comprueba:

- Que el id no sea nulo o esté vacío.
- Que el id sea válido y exista un fichero de texto asociado al id. Se hace uso del servicio de *IFileSystem* para juntar la ruta de la carpeta con el id y después, comprobar la existencia del fichero.

Seguido de ello, se utiliza el servicio de *IFileReaderService* para obtener en forma de string, el contenido del fichero de texto asociado al ID pasado como parámetro, para finalmente, retornarlo al usuario con código 200. En caso de ocurrir un error durante la lectura del fichero, se devuelve un error 500.

```
[HttpGet("Table")]
public IActionResult GetMessage([FromQuery] string id)
{
     // No id
     if (string.IsNullOrWhiteSpace(id))
     {
          return BadRequest("No id provided");
     }

     string imagePath = _fileSystem.Path.Combine(textFolder, $"{id}.txt");

     // No file with that id
     if (!_fileSystem.File.Exists(imagePath))
     {
          return BadRequest("NO image uploaded with that id");
     }

     string text = "";
     try
     {
          text = _fileReader.ReadFileFromPath(imagePath);
     }
     catch (Exception)
     {
          return StatusCode(StatusCodes.Status500InternalServerError, "Failed reading
the text");
     }

     return Content(text, "text/plain");
}
```

Figura 16

8.3.2.3. GET /api/model/table/Ids

Este método, llamado *GetUploadedIds*, figura 17, se encarga de devolver la lista de IDs asociada a la sesión del usuario actual. Este método no recibe ningún parámetro, ya que la identificación del usuario la realiza a través de la propiedad *Session*. Este método se utiliza para que el usuario no tenga que recordar o guardar los IDs.

Primero crea una lista de strings para seguidamente utilizar el servicio de *ISessionService* y utilizar el método que permite obtener la lista de IDs. En caso de ocurrir algún error durante el proceso, se devuelve un error de tipo 500.

```
[HttpGet("Table/Ids")]
public IActionResult GetUploadedIds()
{
    List<string> list;
    try
    {
        list = _session.SessionList(sessionKey);
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError,
"Error when obtaining list");
    }
    return Ok(list);
}
```

Figura 17

8.4. Desarrollo del Front-end

En este apartado se tratarán el desarrollo del front-end de la aplicación, así como decisiones y cambios que han surgido.

8.4.1. Servicios

En este apartado se va a tratar los servicios de la web. En los servicios es donde podemos encontrar toda la lógica del programa. Durante gran parte del proyecto, la lógica del front-end se encontraba en las páginas, debido a la falta de experiencia en el desarrollo de front-end, hasta moverlo a un servicio e inyectarlo en las páginas.

8.4.1.1. UploadImageService

Este servicio se encarga de realizar la comunicación con el servidor API y de la subida de los ficheros.

El servicio de *UploadImageService* presenta dos inyecciones de dependencias:

- *_http*. Necesario para realizar las llamadas HTTP al servidor API. La URI del cliente HTTP se declara en el fichero *Program.cs.*
- *_toastService*. Servicio necesario para mostrar las notificaciones informativas al usuario.

El servicio solo presenta un único método, *UploadImageAsync*, figura 18, el cual recibe el fichero imagen que se quiere subir junto al formato al que se quiere transformar la imagen. El método devuelve la respuesta del servidor.

Primero, se crea una variable de tipo *MultipartFormDataContent*, el cual incluirá solo la imagen del usuario. El primer *try* se encarga de añadir esta imagen del usuario pasada por parámetro, a la variable *content* y, en caso de ocurrir algún error durante el proceso, se lanza un error y se notifica al usuario mediante un toast.

Después, se comienza con la creación de la petición HTTP, para ello se crea una instancia de *HttpRequestMessage* que incluye el método HTTP, en este caso se trata de POST, y la URI a la que se quiere enviar, en este caso es /api/Model/Table?type=,

añadiendo el formato como query. Además, para que funcione la sesión de .NET es necesario enviar el identificador de sesión. Finalmente se añade el contenido, que es la imagen, a la petición.

En el segundo *try* se encuentra el envío de la petición y el procesamiento de la respuesta del servidor API. Para enviar la solicitud al cliente HTTP, se utiliza el método *SendAsync*, y es este el motivo por el que el método es asíncrono, ya que el método no asíncrono de *Send* no funciona correctamente. Una vez recibida la respuesta del servidor, se comprueba que el código de la respuesta es 200, en caso contrario se lanza un error y se comunica al usuario, mediante una notificación, que ha ocurrido algún problema con la API. Si la respuesta es correcta, se comprueba que la respuesta del servidor sea adecuada, es decir, que el ID devuelto no sea nulo o esté vacío y lanzar un error y notificar al usuario si fuese necesario. Para retornar y comprobar la respuesta, se ha creado una clase *ResponseDto*, figura 19, que es necesaria para mapear in información.

```
public async Task<ResponseDto> UploadImageAsync(IBrowserFile file, string type)
{
     bool upload = false;
     long maxFileSize = 1024 * 1024 * 10;         //10MB

     using var content = new MultipartFormDataContent();
     ResponseDto? responseJson = null;

     try
     {
          var fileContent = new StreamContent(file.OpenReadStream(maxFileSize));

          fileContent.Headers.ContentType = new MediaTypeHeaderValue(file.ContentType);

          content.Add(content: fileContent, name: "\"image\"", fileName: file.Name);
          upload = true;

     }
     catch (Exception)
     {
          _toastService.ShowError("Error when uploading image", 3);
          throw new InvalidOperationException("Error when uploading image");
     }

     if (upload)
     {

          var request = new HttpRequestMessage(HttpMethod.Post,
$"/api/Model/Table?type={type}");
          request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
          request.Content = content;

          try
          {
               var response = await _http.SendAsync(request);

               if (response.IsSuccessStatusCode)
               {
                    using var responseStream = response.Content.ReadAsStream();
                    responseJson =
JsonSerializer.Deserialize<ResponseDto>(responseStream);

                    if (responseJson is not null)
                    {
                         string tmp = responseJson.id;
                         if (string.IsNullOrWhiteSpace(tmp))
                         {
                              _toastService.ShowError("No ID returned from API", 3);
                              throw new InvalidOperationException("No ID returned from
API");
                         }
                         else
                         {
                              _toastService.ShowSuccess("Upload Successful", 3);
                              return responseJson;
                         }
                    }
                    else
                    {
                         _toastService.ShowError("Error with API response", 3);
                         throw new InvalidOperationException("Error with API response");
                    }
               }
               else
               {
                    _toastService.ShowError("Error from API, API code is not 200", 3);
                    throw new InvalidOperationException("Error from API, API code is
```

Figura 18

```
public class ResponseDto
{
    public string? message { get; set; }
    public string? id { get; set; }
}
```

Figura 19

8.4.1.2. ToastService

Este servicio se encarga de mostrar las notificaciones toast. Esta implementación de los toast en Blazor está basada en el artículo de C# Corner escrito por Rijwan Asnari.

El servicio describe dos eventos, *OnShow* y *OnHide*. Cuando se llama al método de *ShowToast*, activa el evento de *OnShow*, pasando el mensaje que se quiere mostrar en la notificación, el tiempo de notificación y el tiempo que se quiere que dure en pantalla. Cuando se llama al método de *HideToast*, se activa el evento de *OnHide.* Aparte de esto, se han creado cinco métodos más para facilitar su implementación en otros servicios de la web. La implementación de la clase se puede ver en la figura 20.

```
public class ToastService
{
    public event Action<string, string, int> OnShow;
    public event Action OnHide;
    public void ShowToast(string message, string type = "info", int dismissAfter = 3)
    {
        OnShow?.Invoke(message, type, dismissAfter);
    }
    public void HideToast()
    {
        OnHide?.Invoke();
    }

    public void ShowSuccess(string message, int dismissAfter = 3) =>
ShowToast(message, "success", dismissAfter);
    public void ShowError(string message, int dismissAfter = 3) => ShowToast(message,
"failure", dismissAfter);
    public void ShowWarning(string message, int dismissAfter = 3) =>
ShowToast(message, "warning", dismissAfter);
    public void ShowInfo(string message, int dismissAfter = 3) => ShowToast(message,
"info", dismissAfter);
    public void ShowAlert(string message, int dismissAfter = 3) => ShowToast(message,
"alert", dismissAfter);
}
```

Figura 20

8.4.2. Paginas / Layouts

Las páginas representan la vista completa que verá el usuario cuando la solicite a través de una URL. En esta web aparecen dos páginas:

- *Upload*. El usuario accede a esta página para poder seleccionar el formato y subir la imagen.
- *Output*. El usuario accede a esta página para poder obtener la respuesta del modelo de IA.

8.4.2.1. *Upload*

En la página de *Upload*, figura 21, se permite al usuario seleccionar el formato y subir la imagen al servidor. En esta página se realizan dos inyecciones de dependencias:

- *_logger*. Servicio que permite realizar escrituras en la consola durante el desarrollo para seguir el correcto funcionamiento.
- *_imageService*. Servicio que permite subir la imagen al servidor mediante comunicaciones API.

La selección del formato se realiza mediante un *select* y un *foreach*. Mediante el uso del *foreach*, se recorre la lista completa de formatos y se imprimen en forma de una lista en la que le usuario elige el que quiere, evitando errores al escribir formatos no aceptados en una caja de texto. Por otra parte, con razor se puede hacer uso de *@bind*, permite relacionar una de las variables, *type* en este caso, con lo elegido por el usuario, haciendo el código más sencillo y legible.

Para la elección de ficheros, se utiliza un *InputFile* que en el momento que se detecta un cambio (El usuario selecciona una imagen), se ejecuta el método de *OnInputImageChange*.

```
<body>
    <h1 style="margin-bottom: 20px;">Upload your image</h1>

    <label style="margin-bottom: 10px;">Choose the output:</label>

    <select @bind=type class="form-select" style="width: 20%;">
        @foreach (var t in types)
        {
            <option value="@t">@t</option>
        }
    </select>
    <div class="mt-4">
        <label>
                Upload up to @maxAllowedImages file:
                <InputFile OnChange="OnInputImageChange" accept=".jpg,.jpeg,.png"/>
        </label>
    </div>
</body>
```

Figura 21

En la sección de código de la página, figura 23, solo hay un único método, *OnInputImageChange*. Este método se llama cuando el usuario selecciona una imagen. El único parámetro que recibe se trata de un *InputFileChangeEventArgs* que es la imagen que el usuario ha subido. El método recorre todas las imágenes, aunque en este caso está limitado a una única imagen, para después llamar al método de *UploadImageAsync* del servicio de *_imageService*.

Al final de todo el proceso se permite a la pagine renderizar. La vista de la página en el navegador se puede ver en la figura 22.

Figura 22

```
@code {
    private List<string> types = new() { "Json", "Markdown", "Csv", "SQL create
table", "Html" };
    private string type = "Json";
    private string returnId = "";
    private int maxAllowedImages = 1;
    private bool shouldRender = true;

    protected override bool ShouldRender() => shouldRender;

    public async Task OnInputImageChange(InputFileChangeEventArgs e)
    {
        shouldRender = false;

        using var content = new MultipartFormDataContent();

        foreach (var file in e.GetMultipleFiles(maxAllowedImages))
        {
            await _imageService.UploadImageAsync(file, type);
        }

        shouldRender = true;
    }

}
```

Figura 23

8.4.2.2. Output

En la página de *Output*, figura 24, se permite al usuario seleccionar de una lista, una de las imágenes que ha subido, para después solicitar el output del modelo de IA, si el proceso ha terminado.

La página está formada por una lista desplegable, esta lista contiene las imágenes que el usuario ha subido durante esa sesión. Esta lista se realiza mediante el uso del *select* para crear la lista y un *for* para poder rellenar la lista con los IDs. Con la opción de *@bind* y *value* es posible relacionar la elección del usuario mostrando un texto diferente en la lista al usuario.

Por otra parte, la página presenta dos botones, un botón para refrescar las imágenes disponibles para elegir en la lista y otro botón para obtener el output del modelo de IA. El botón de refresco activa el método de *GetList*, mientras que el botón de obtener output llama al método de *GetText*.

```
<PageTitle>Output</PageTitle>

<h1>Output</h1>

<div style="display: flex; align-items: center; gap: 1rem;">
    <select @bind=id class="form-select" style="width: 20%;">
        @for (int i = 0; i < list.Count; i++)
        {
            <option value="@list[i]">Image @i</option>
        }
    </select>
    <button class="btn btn-primary" @onclick="GetText">Get Output</button>
    <button class="btn btn-primary" @onclick="GetList">Refresh List</button>
</div>
<div class="mt-4">
<textarea @bind="text" style="width: 70%; height: 80vh;"></textarea>
</div>
```

Figura 24

En el apartado de *@code*, hay dos métodos:

- *GetList*, figura 26. Este método se encarga de obtener los IDs de las imágenes asociadas a la sesión actual.
- *GetText*, figura 27. Este método se encarga de recibir la respuesta del método de IA asociada a un ID.

A diferencia de la página de *Upload*, no hay servicios que contengan la lógica ni comunicaciones con la API.

El método de *GetList* comienza con la creación de la solicitud indicando el método, un GET, y la URI. Además, se incluye las credenciales que permiten hacer uso de las sesiones de .NET. El *try* incluye la comunicación con la API, en caso de ocurrir un error, se muestra una notificación al usuario. Seguido, se comprueba que la respuesta del servidor sea de tipo 200, mostrando error en caso contrario. Después, se intenta obtener la lista de strings de la respuesta del servidor, si se obtiene la lista de forma, la lista visible se actualiza con el uso del *@bind* y se le da el valor del primer ID de la lista a la variable *id*.

El método de *GetText* sigue los mismos pasos que le método de *GetList* con la diferencia que la respuesta esperada se trata de un único string en vez de una lista de strings.

A continuación, se la vista de la página en el navegador en la figura 25.

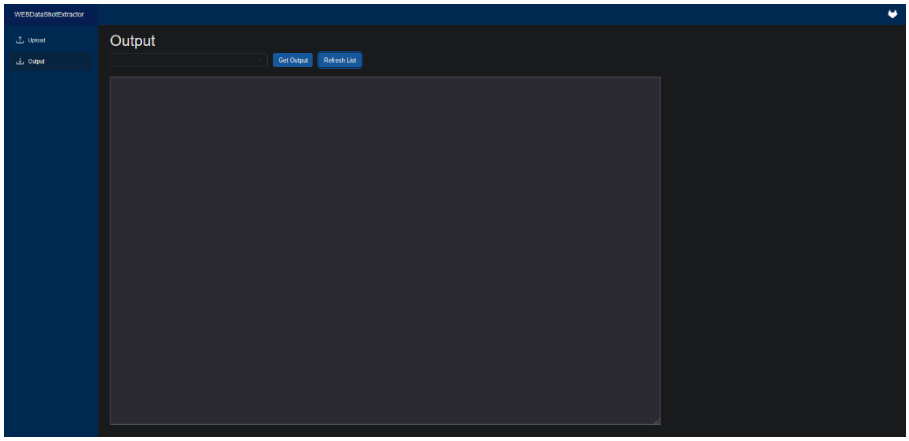


Figura 25

```
public async Task GetList()
{
     var request = new HttpRequestMessage(HttpMethod.Get, "/api/Model/Table/Ids");
     request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);

     try{
          var response = await _http.SendAsync(request);

          if(response.IsSuccessStatusCode)
          {
               list = await response.Content.ReadFromJsonAsync<List<string>>();
               if(list is null)
               {
                    _toastService.ShowError("Error when reading response", 3);
               }
               else if(list.Count > 0)
               {
                    id = list[0];
               }
          }
          else
          {
               _toastService.ShowError("Error from API, API code is not 200", 3);
               list = new List<string>();
          }
     }
     catch
     {
          _toastService.ShowError("No communication with API", 3);
     }
}
```

Figura 26

```
public async Task GetText()
{
     var request = new HttpRequestMessage(HttpMethod.Get, "/api/Model/Table?id=" +
id);
     request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);

     try
     {
          var response = await _http.SendAsync(request);
          if(response.IsSuccessStatusCode)
          {
               text = await response.Content.ReadAsStringAsync();
               if(string.IsNullOrWhiteSpace(text))
               {
                    _toastService.ShowError("Empty response from API", 5);
               }

          }
          else
          {
               _toastService.ShowError("Error from API response, ID might not exist",
3);
          }
     }
     catch
     {
          _toastService.ShowError("No communication with API", 3);
     }
}
```

Figura 27

9. Pruebas

En el siguiente apartado se va a tratar los resultados y test que se han realizado al servicio web.

9.1. Back-end Unitarias

Para realizar las pruebas unitarias del back-end se ha usado tanto NUnit. También, se ha hecho uso del paquete Moq el cual permite realizar mocks, o instancias falsas, de las clases de las que dependen la clase *sut* sobre la que estamos realizando las pruebas. Además, se ha hecho uso de Swagger para realizar las pruebas de forma manual.

A continuación, se muestran las pruebas realizadas en NUnit.

- Método *PostImage* de *ModelController*

ID	Test	Resultado
UT00	Imagen y formato validos	OK 200 y se llama a *GetTable*
UT01	Imagen y formato válidos, fallo al guardar imagen	Error 500
UT02	Imagen es nulo	Error 400
UT03	Imagen no presenta extensión valida	Error 400
UT04	Formato no valido	Error 400
UT05	Formato vacío/nulo	Error 400
UT06	Imagen y formato válidos. *ModelService* lanza error.	Error 500

Tabla 8

- Método *GetMessage* de *ModelController*

ID	Test	Resultado
UT00	ID valido y fichero existente	OK 200, contenido correcto
UT01	ID vacío	Error 400
UT02	ID nulo	Error 400
UT03	ID valido, pero no existe fichero	Error 400
UT04	ID valido y fichero existente, pero error al leer fichero	Error 500

Tabla 9

- Método *GetUploadIds* de *ModelController*

ID	Test	Resultado
UT00	Todo correcto	OK 200
UT01	*SessionList* lanza error	Error 500

Tabla 10

- Método *GetTable* de *ModelService*

ID	Test	Resultado
UT00	Ruta de imagen y modelo, formato e id validos	No se lanzan errores y se llama a *QueueBackgroundWorkItemAsync*
UT01	Ruta de imagen es nula	Se lanza *ArguementException*
UT02	Ruta de imagen está vacía	Se lanza *ArguementException*
UT03	Formato es nulo	Se lanza *ArguementException*
UT04	Formato está vacío	Se lanza *ArguementException*
UT05	Id es nulo	Se lanza *ArguementException*
UT06	ID está vacío	Se lanza *ArguementException*
UT07	Ruta del modelo está vacía	Se lanza *ArguementException*
UT08	Ruta del modelo es nulo	Se lanza *ArguementException*

Tabla 11

- Método *ReadFileFromPath* de *FileReaderService*

ID	Test	Resultado
UT00	Ruta correcta con fichero existente	String con el contenido de texto esperado
UT01	Ruta es nula	Se lanza *ArguementException*
UT02	Ruta está vacía	Se lanza *ArguementException*
UT03	*StreamReader* lanza excepción	Se lanza *IOExpection*

Tabla 12

A continuación, se muestra parte del código de las pruebas, en concreto, la configuración y la prueba de éxito en cada uno de los métodos.

```
[SetUp]
public void Setup()
{
        _mockModelService = new Mock<IModelService>();
    _mockFileSystem = new Mock<IFileSystem>();
    _mockSessionService = new Mock<ISessionService>();
    _mockFileReaderService = new Mock<IFileReaderService>();
    sut = new ModelController(_mockModelService.Object,_mockSessionService.Object,
_mockFileReaderService.Object, _mockFileSystem.Object);

}

[Test]
public async Task PostImage_ValidImageAndType_ReturnsOkAndSavesImage()
{
    // Arrange
    var mockFile = CreateMockFormFile("test.jpg", "image/jpeg", true);

    // Act
    var result = sut.PostImage(mockFile.Object, "Markdown");

    // Assert
    ClassicAssert.NotNull(result);
    ClassicAssert.IsInstanceOf<OkObjectResult>(result);
    _mockModelService.Verify(mock => mock.GetTable(It.IsAny<string>(),
It.IsAny<string>(), It.IsAny<string>()), Times.Once());
}
```

Figura 28

```
[Test]
public void GetMessage_ValidId_ReturnOkAndStream()
{
// Arrange
_mockFileSystem.Setup(f => f.File.Exists(It.IsAny<string>())).Returns(true);
_mockFileSystem.Setup(f => f.Path.Combine(It.IsAny<string>(),
It.IsAny<string>())).Returns("Test");
_mockFileReaderService.Setup(f =>
f.ReadFileFromPath(It.IsAny<string>())).Returns("AAAA");

// Act
var result = sut.GetMessage("Test") as ContentResult;

// Assert
ClassicAssert.NotNull(result);
ClassicAssert.AreEqual(result.Content, "AAAA");
ClassicAssert.AreEqual(result.ContentType, "text/plain");
}
```

Figura 29

```
[SetUp]
public void Setup()
{
_mockBackgroundTaskQueue = new Mock<IBackgroundTaskQueue>();
_mockLogger = new Mock<ILogger<ModelService>>();
_mockModel = new Mock<IOptions<ModelSettings>>();
ModelSettings s = new ModelSettings() { ModelPath = "path" };
_mockModel.Setup(f => f.Value).Returns(s);
sut = new ModelService(_mockBackgroundTaskQueue.Object, _mockLogger.Object,
_mockModel.Object);
}

[Test]
public void GetTable()
{

// Act
sut.GetTable("path", "type", "id");

// Assert
_mockBackgroundTaskQueue.Verify(mock =>
mock.QueueBackgroundWorkItemAsync(It.IsAny<Func<CancellationToken, ValueTask>>()),
Times.Once);
}
```

Figura 30

```
[SetUp]
public void Setup()
{
_mockFileManager = new Mock<IFileManager>();
sut = new FileReaderService(_mockFileManager.Object);
}

[Test]
public void ReadFileFromPath_ReturnsText()
{

// Arrange
string fakeFileContents = "Hello world";
byte[] fakeFileBytes = Encoding.UTF8.GetBytes(fakeFileContents);
MemoryStream fakeMemoryStream = new MemoryStream(fakeFileBytes);
_mockFileManager.Setup(fileManager =>
fileManager.StreamReader(It.IsAny<string>())).Returns(() => new
StreamReader(fakeMemoryStream));

// Act
string result = sut.ReadFileFromPath("path");

// Assert
ClassicAssert.IsNotNull(result);
ClassicAssert.AreEqual(result, "Hello world" + Environment.NewLine);

}
```

Figura 31

9.2. Front-end

Las pruebas del front-end del servicio web se han realizado probando la aplicación después de ser desplegada. Estas pruebas se han realizado para comprobar que no se lanza ningún error que no esté siendo manejado, evitando que la página deje de funcionar, además de que siempre se envíe una notificación al usuario cuando surge un error.

En la tabla 13 se pueden ver las pruebas realizadas con sus resultados.

ID	Test	Resultado
T00	Subir imagen	La imagen se ha subido al servidor y se notifica al usuario
T01	Se sube fichero que no es imagen (.png, .jpg, .jpeg)	Error que se notifica al usuario
T02	Refrescar lista sin haber subido imágenes	La aplicación sigue funcionando
T03	Refrescar lista habiendo subido imágenes	La lista se actualiza con las imágenes subidas.
T04	Obtener output cuando la imagen está siendo procesada o en cola	No se imprime nada en la caja de texto y se notifica al usuario.
T05	Obtener output cuando la imagen ya ha sido procesada.	Se imprime la salida del modelo en la caja de texto y se notifica al usuario.

Tabla 13

9.3. Pruebas de Aceptación

Antes del resultado final de la página web, el proceso de subir la imagen y obtener la respuesta del modelo era diferente, ya que no existía las sesiones ni la lista desplegable para seleccionar la imagen. El flujo del usuario antes era más incómodo:

1. El usuario selecciona el formato.
2. El usuario sube la imagen.
3. El usuario guarda o copia el GUID de la imagen devuelto por el servidor.
4. El usuario se dirige a la pestaña de output.
5. El usuario escribe el GUID en la caja de texto.
6. El usuario presiona el botón de obtener output.

Durante el proceso de pruebas de aceptación, uno de los problemas más grandes del programa resultó ovio al momento. El hecho de tener que guardar el GUID en un editor de texto o en cualquier otro lugar del ordenador del usuario, resulta bastante incomodo. Este problema solo aumenta cuantas más imágenes se suban al servidor.

A la conclusión que se llegó durante el desarrollo es que era necesario eliminar el proceso de la gestión y guardado de las GUID por parte del usuario. Para eliminar esta responsabilidad sin añadir una base de datos con inicio de sesión, se decidió usar o *cookies* o *session* de .NET. Debido a que no se quería añadir *cookies* a la página web, se decidió usar las sesiones.

En la página de *output* se añadió la lista desplegable de imágenes. Para implementar la lista fue necesario añadir una nueva llamada API para obtener la lista completa de IDs del usuario asociada a la sesión.

Después de todos los cambios realizados, se volvió a realizar las pruebas de aceptación y el co-director acabo contento con los resultados.

10. Docker

Para el proyecto se ha realzado un Docker el cual permite un fácil y rápido despliegue del servidor de forma local, el cual era una de las ideas importantes del proyecto. Para el despliegue del servicio es necesario realizar tres pasos:

1. Descargar el modelo de inteligencia artificial del repositorio de Hugging Face
2. El despliegue del servidor API.
3. El despliegue del servicio WEB.

La creación del Docker permite que cualquier persona, aunque no tenga idea de servidores, sea capaz de desplegar el servicio simplemente descargando o clonando el repositorio donde se guarda el proyecto.

A continuación, se muestra el código del Docker de la WEB, figura 33, y de la API, figura 32, los cuales generan las imágenes de ambos, y finalmente el código del Docker, figura 34, que genera el contenedor de la aplicación.

```
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

# Set the working directory inside the container
WORKDIR /app

# copy csproj and restore as distinct layers
COPY APIDataShotExtractor.csproj ./
RUN dotnet restore "APIDataShotExtractor.csproj"

# Copy the rest of the application files to the container
COPY . .

# Get model
RUN apt-get update
RUN apt-get install -y git-lfs
RUN git lfs install
RUN git clone https://huggingface.co/microsoft/Phi-3-vision-128k-instruct-onnx
WORKDIR /app/Phi-3-vision-128k-instruct-onnx
RUN git lfs pull

# Build the application in Release mode and output the build to /app/build
WORKDIR /app
RUN dotnet build "APIDataShotExtractor.csproj" -c Release -o /app/build

# Use the build stage to publish the application
FROM build AS publish

# Publish the application in Release mode to /app/publish
RUN dotnet publish "APIDataShotExtractor.csproj" -c Release -o /app/publish

# Use the ASP.NET runtime image for running the application
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS final

# Set the working directory inside the container
WORKDIR /app

# Copy the published application from the publish stage
COPY --from=publish /app/publish .
COPY --from=build /app/Phi-3-vision-128k-instruct-onnx ./Phi-3-vision-128k-instruct-
onnx

# Set the environment variable to configure the application to listen on port 80
ENV ASPNETCORE_URLS=http://+:5296

# Expose port 5296 to allow external access to the application
EXPOSE 5296

# Set the entry point to run the application
ENTRYPOINT ["dotnet", "APIDataShotExtractor.dll"]
```

Figura 32

```
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

# Set the working directory inside the container
WORKDIR /app

# Copy the project file into the container
COPY WEBDataShotExtractor.csproj ./
RUN dotnet restore "WEBDataShotExtractor.csproj"

# Copy the rest of the source code into the container
COPY . .

# Build the application in Release mode and output the build to /app/build
RUN dotnet build "WEBDataShotExtractor.csproj" -c Release -o /app/build

# Use the build stage to publish the application
FROM build AS publish

# Publish the app in Release mode to the 'out' folder
RUN dotnet publish "WEBDataShotExtractor.csproj" -c Release -o /app/publish

# Use the Nginx base image to serve the published app
FROM nginx AS final

# Set a working directory (optional, for consistency)
WORKDIR /app

# Copy the published application from the publish stage
COPY --from=publish /app/publish .

# Expose port 80 (the default port Nginx listens on)
EXPOSE 80

# Copy the published Blazor files to the Nginx web root
COPY --from=publish /app/publish/wwwroot /usr/share/nginx/html

CMD ["nginx", "-g", "daemon off;"]
```

Figura 33

```
services:
  api: # Service for the API
    build: ./API # Path to the Dockerfile for building the API
    environment:
      - ASPNETCORE_ENVIRONMENT=Development # Set the environment to Development
    ports:
      - "5296:5296" # Map port 5296 on the host to port 5296 in the container
    networks:
      - datashot-net # Connect this service to the 'datashot-net' network

  web: # Service for the WEB (frontend)
    build: ./WEB # Path to the Dockerfile for building the WEB
    environment:
      - ASPNETCORE_ENVIRONMENT=Development # Set the environment to Development
      - ApiUrl=http://localhost:5296/api/Model/ # URL for the API
    ports:
      - "5053:80" # Map port 5053 on the host to port 80 (nginx default port) in the
container
    depends_on:
      - api # Ensure the API service starts before this service
    networks:
      - datashot-net # Connect this service to the 'datashot-net' network

networks:
  datashot-net: # Define a custom network named 'datashot-net'
```

Figura 34

11. Conclusión y futuro trabajo

En resumen, se ha conseguido completar los requisitos de la aplicación. El objetivo del proyecto era el aprender algunas de las tecnologías más populares, además de aprender a manejar e implementar modelos de inteligencia artificial, no solo en Python, si no también en otros entornos de desarrollo. También, se ha conseguido desarrollar una aplicación full-stack y conseguir desplegarla en un entorno de Docker. Otro objetivo importante era el de aprender a organizar, por mi parte, un proyecto de varios meses de duración, aprendiendo a organizarme y calcular adecuadamente la carga de trabajo.

Por otra parte, una de las tareas deseadas a implementar era la integración del modelo de IA con GPU y no solo con CPU, pero debido a un fallo de calculo de la carga de trabajo de algunas tareas y de retrasar las tareas, no se ha podido implementar. Otra cosa con la que no estoy completamente contento es con el diseño de la capa de presentación del front-end, aunque tampoco estaba dentro de mi interés, pero tengo pensado aprender más de ello a futuro, empezando por mejorar el diseño de este TFG más adelante.

Durante el desarrollo se han implementado todas las pruebas del back-end utilizando las técnicas aprendidas durante la carrera, pero no se ha podido hacer los del front-end. Estuve mirando la librearía de BUnit para hacer pruebas de Blazor, pero para mi parecer, hace falta más documentación y guías, además de que no llegue a entender como mockear la subida de ficheros.

Al final, se ha conseguido completar la aplicación, menos con lo mencionado anteriormente, pero estoy muy contento con el desarrollo y resultado final y siento que he aprendido bastante. Por otra parte, estoy descontento conmigo mismo por la falta de organización por mi parte, ya que el objetivo era entregar este TFG durante la convocatoria de junio, tarea que era más que factible además de haber añadido el objetivo de la GPU mencionado antes.

Por otro lado, quiero añadir que tengo pensado traducir el front-end a React, el cual es bastante usado hoy en día, además de añadir una base de datos, inicios de sesión, autorizaciones y persistencia atemporal con el objetivo de aprender y seguir practicando. También, tengo el interés de entrenar este modelo de IA (phi3) para realizar esta actividad mejor, aunque este es menos probable que lo intente.

12. Bibliografía

[1] Python, «What is Python? Executive Summary,» [En línea]. [Último acceso: 18 Junio 2025].

[2] «PyTorch documentation,» Pytorch, [En línea]. Available: https://docs.pytorch.org/docs/stable//index.html#pytorch-documentation. [Último acceso: 18 junio 2025].

[3] «https://onnx.ai/about.html,» Open Neural Network Exchange, [En línea]. Available: https://onnx.ai/about.html. [Último acceso: 18 junio 2025].

[4] R. A. y. S. L. Daniel Roth, «Overview of ASP.NET Core,» ASP.NET Core, 21 abril 2025. [En línea]. Available: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-9.0.

[5] Microsoft, «Launch your idea to the web fast with Blazor,» Blazor, [En línea]. Available: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor. [Último acceso: 19 junio 2025].

[6] Microsoft, «What is Git?,» Git, 28 noviembre 2022. [En línea]. Available: https://learn.microsoft.com/en-us/devops/develop/git/what-is-git.

[7] Docker, Inc., Docker, [En línea]. Available: https://docs.docker.com/get-started/docker-overview/. [Último acceso: 19 junio 2025].

[8] Anaconda, «Conda Documentation,» Conda, [En línea]. Available: https://anaconda.org/anaconda/conda. [Último acceso: 19 junio 2025].

[9] Anaconda, «Miniconda,» Miniconda, [En línea]. Available: https://www.anaconda.com/docs/getting-started/miniconda/main. [Último acceso: 19 junio 19].

[10] «What Is NUnit?,» NUnit, [En línea]. Available: https://nunit.org/. [Último acceso: 20 junio 2025].

[11] IBM, «IBM Product Master Documentación,» 14 enero 2025. [En línea]. Available: https://www.ibm.com/docs/es/product-master/12.0.0?topic=processes-defining-use-cases.

[12] Wikipedia, «Universally unique identifier,» [En línea]. Available: https://en.wikipedia.org/wiki/Universally_unique_identifier. [Último acceso: 29 junio 2025].

[13] B. Pazur, «What Is DeepSeek? Everything to Know About the New Chinese AI Tool,» 31 Enero 2025. [En línea]. Available: https://www.cnet.com/tech/services-and-software/what-is-deepseek-everything-to-know-about-the-new-chinese-ai-tool/.

[14] OpenAI, «About Us,» [En línea]. Available: https://openai.com/about/. [Último acceso: 1 Julio 2025].