Trabado de Fin de Grado

GRADO EN INFORMÁTICA

Facultad de Matemáticas e Informática

Universidad de Barcelona

Implementación de una plataforma web para la visualización y manipulación de ecografías intravasculares

---

Autor: Maria Roman Martin

Director: Dr. Simone Balocco
Realitzado en: Departamento de Matemáticas e Informática

Barcelona, June 12, 2023

Resum

Per tal d'augmentar la precisió de la computació de contorns, aquest Treball de Fi de Grau proposa una aplicació per a l'administració d'ultrasons intravasculars. Un mètode de diagnòstic d'imatge no invasiu utilitzat per avaluar els trastorns vasculars és l'ecografia intravascular. Prendre decisions clíniques significatives requereix un maneig precís de les troballes. L'aplicació es crea a Django utilitzant el llenguatge de programació Python, amb emmagatzematge de fitxers al núvol proporcionat per Google Cloud Storage, i s'implementa a Azure App Service. L'instrument busca augmentar l'eficàcia de l'administració d'ultrasò intravascular i oferir troballes més precises per donar suport a la interpretació i el diagnòstic a través de les funcionalitats desenvolupades durant el projecte. Entre elles i la més important en aquest treball ha estat la capacitat de poder dibuixar la túnica mitjana i íntima en una capa d'una ecografia intravascular. A més, s'ha aconseguit la incorporació d'un altre projecte, que utilitza tècniques d'aprenentatge automàtic (machine learning) per calcular contorns a les ecografies intravasculars. Aquesta funcionalitat addicional ha millorat encara més la precisió i la capacitat d'anàlisi de l'instrument i ha permès una interpretació més detallada i un diagnòstic més fiable.

Resumen

Con el fin de aumentar la precisión de la computación de contornos, este Trabajo de Fin de Grado propone una aplicación para la administración de ultrasonidos intravasculares. Un método de diagnóstico de imagen no invasivo utilizado para evaluar los trastornos vasculares es la ecografía intravascular. Tomar decisiones clínicas significativas requiere un manejo preciso de los hallazgos. La aplicación se crea en Django utilizando el lenguaje de programación Python, con almacenamiento de archivos en la nube proporcionado por Google Cloud Storage, y se implementa en Azure App Service. El instrumento busca aumentar la eficacia de la administración de ultrasonido intravascular y ofrecer hallazgos más precisos para apoyar la interpretación y el diagnóstico a través de las funcionalidades desarrolladas durante el proyecto. Entre ellas y la más importante en este trabajo ha sido la capacidad de poder dibujar la túnica media e íntima de una capa de una ecografía intravascular. Además, se ha logrado la incorporación de otro proyecto, que utiliza técnicas de aprendizaje automático (machine learning) para calcular contornos en las ecografías intravasculares. Esta funcionalidad adicional ha mejorado aún más la precisión y la capacidad de análisis del instrumento, permitiendo una interpretación más detallada y un diagnóstico más confiable.

Abstract

In order to increase the precision of contour computation, this bachelor's thesis proposes an application for the administration of intravascular ultrasounds. A non-invasive imaging diagnostic method used to assess vascular disorders is intravascular ultrasounds. Making significant clinical decisions requires accurate handling of the findings. The application is created in Django using the Python programming language, with cloud file storage provided by Google Cloud Storage, and being deployed in Azure App Service. The instrument seeks to increase the effectiveness of intravascular ultrasound administration and provide more accurate findings to support interpretation and diagnosis through the functionalities developed during the project. Among them and the most important in this work has been the ability to be able to draw the middle and intimate tunic of a layer of an intravascular ultrasound. In addition, the incorporation of another project has been achieved, which uses machine learning techniques to calculate contours in intravascular ecographs. This additional functionality has further improved the accuracy and analytical capability of the instrument, allowing for more detailed interpretation and a more reliable diagnosis.

Contents

1 Introducción **1**

1.1 Objetivos 1

1.2 Motivación 1

1.3 Estructura de la Memoria 1

1.4 Planificación 3

2 Marco teórico **4**

2.1 Medicina 4

2.2 Informática 4

3 Tecnologías **6**

3.1 Marco de trabajo 6

3.2 Base de datos 6

3.3 Almacenamiento en la nube 7

3.4 Despliegue 7

3.5 Lenguaje de programación 9

3.5.1 Backend 9

3.5.2 Frontend 10

4 Antecedentes **12**

4.1 Funcionalidades 12

4.2 Base de Datos 13

4.3 Interfaz 14

5 Estructura Código **17**

5.1 Funcionamiento Django 17

5.2 Modelo de datos 18

5.3 Vistas 18

5.4 Plantillas 19

5.5 Enrutamiento 20

5.6 Formularios 21

6 Implementación **23**

6.1 Base de datos 23

6.2 Almacenamiento en la nube 24

6.3 Funcionalidades 26
6.4 Modificaciones Frontend 35

7 Costes **37**
7.1 Google Cloud Storage 37
7.2 Heroku 37
7.3 Azure App Service 38

8 Conclusiones **40**
8.1 Resultados 40
8.2 Aprendizaje Personal 40
8.3 Trabajo Futuro 41
8.4 Documentos de despliegue 43

Introducción

1.1 Objetivos

Actualmente, el área de la medicina se basa en gran medida en el uso de *imágenes médicas* para el diagnóstico y la terapia de enfermedades. Los expertos médicos emplean con frecuencia imágenes *DICOM* en este contexto para el diagnóstico y tratamiento de diversas enfermedades. El objetivo de este Trabajo de Fin de Grado es la continuidad del desarrollo de una aplicación que manipula y visualiza archivos *DICOM*.

Para ello, será necesaria la revisión y actualización las *librerías* para, posteriormente, *desplegar* la página web en una *plataforma de nube* y funcionalidades, en particular, el dibujado de capas, para mejorar la experiencia del usuario y la precisión del diagnóstico.

Además, se propone implementar un proyecto externo de otra compañera en el que se usa un algoritmo de *machine learning* encargado de obtener contornos de una capa, lo que permitirá una mayor precisión y rapidez en el análisis de las *imágenes médicas*.

1.2 Motivación

La medicina y la informática han estado interrelacionadas durante décadas. La computación se ha convertido en una herramienta esencial en la investigación, el diagnóstico y el tratamiento de enfermedades. También ha permitido la gestión y análisis de grandes cantidades de datos médicos, lo que permite una mejor comprensión de los patrones de enfermedades, la efectividad de los tratamientos y la prevención de enfermedades.

Estos procesos, que sin la informática, tendrían un tiempo de ejecución más largo, ya que son los seres humanos los que lo realizarían, pueden automatizarse y optimizarse para facilitar una ayuda a este colectivo, lo cual podría permitir a la vez una agilización en una parte del sistema sanitario.

Por ello, considero que este proyecto es una excelente oportunidad para aplicar mis conocimientos y habilidades en informática en un contexto relevante y con impacto directo en la salud de las personas.

1.3 Estructura de la Memoria

Durante la realización de la memoria se ha ido reestructurando continuamente hasta acabar con un guión definitivo, debido a que, a medida que se ha ido redactando, han surgido cambios y modificaciones.

A continuación se explicará brevemente la estructura final de esta memoria con cada apartado.

Primero nos encontramos con una introducción, donde se explica cuáles han sido los objetivos principales de este Proyecto IVUS, de que trata y la motivación por la cual se ha escogido este trabajo. Dentro de este apartado también encontramos la planificación a través de una *metodología agile* y revisiones periódicas que se ha seguido durante la realización de este Trabajo de Fin de Grado.

En la siguiente sección tenemos el marco teórico del proyecto. En él se definen conceptos que no se explican durante la redacción de otros puntos de la memoria, tanto en el ámbito médico como en el informático.

Seguidamente, se explican las tecnologías que se han usado durante el desarrollo de la aplicación. Dentro se menciona Django, el marco de trabajo usado junto con la base de datos, el almacenamiento en la nube usado, las plataformas de despliegue en la nube y finalmente, los lenguajes que se han usado tanto para la parte de *backend* como de *frontend*.

En el siguiente apartado tenemos los antecedentes de este Proyecto IVUS, ya que este Trabajo de Fin de Grado fue heredado por el alumno Daniel Ruiz. En él, se mencionan y se explican brevemente las distintas funcionalidades ya implementadas, la estructura de la base de datos y la interfaz usada.

Acto seguido se encuentra la estructura del código del proyecto. En este, se explica el funcionamiento básico del *framework* de Django y cada uno de sus componentes a detalle. Estos son el modelo de datos, las vistas, las plantillas, el enrutamiento y los formularios. Para cada uno, se muestra ejemplos usados dentro del Proyecto IVUS.

En el antepenúltimo capítulo tenemos la implementación del proyecto. Primero de todo se explica la nueva base de datos con los cambios realizados respecto al año pasado. De la misma forma se hace para el almacenamiento en la nube y su estructura. Seguidamente, se explican todas las funcionalidades implementadas, tanto las nuevas como las que se han modificado. Finalmente, se mencionan los cambios de *frontend* más importantes que se han realizado en la aplicación.

En el siguiente capítulo, se muestran todos los costes derivados de la aplicación. Están divididos en tres apartados: los costes del almacenamiento en la nube de Google Cloud Storage y los de las dos plataformas de Azure y Heroku para el despliegue de la aplicación. También se muestran los precios que supondría el coste de poner el Proyecto IVUS en producción.

En la penúltima sección, en las conclusiones, se detallan los resultados obtenidos al finalizar el Trabajo de fin de Grado y el trabajo futuro que podría ser realizado para mejorar el Proyecto IVUS.

Finalmente se encuentra un anexo en el que se mencionan los documentos de despliegue realizados durante este trabajo y la bibliografía con los enlaces usados para la realización de esta memoria.

1.4 Planificación

La toma de decisión de la realización del proyecto fue en septiembre. Con el Dr. Balocco, tuvimos una reunión donde me explico cuál era la temática del proyecto y discutir brevemente cuáles serían mis tareas dentro del trabajo y mencionarme cuál sería el *framewok* de desarrollo, el cual sería Django siguiendo con el proyecto IVUS anterior. Gracias a la asignatura de Ingeniería de Software, pude conocer con profundidad con este *framework* y conocer la estructura de un proyecto basado en Django. En diciembre, tuvimos la segunda reunión para decidir cuáles serían mis puntos de partida. Acordamos que la primera tarea era familiarizarme y poder ejecutar en mi ordenador el proyecto del año anterior. Para ello dediqué un mes y medio y gracias a la ayuda de Dani Ruiz, el autor del código, pude apropiarme del proyecto y sentirme cómoda.

Para el desarrollo del proyecto, establecimos con el Dr. Balocco mantener una *metodología Agile* a través de varios *sprints* durante todo el proceso de elaboración, en los cuales se mostraba el trabajo realizado del sprint anterior y se presentaban y discutían las propuestas de futuras funcionalidades o cambios en el proyecto. Además de *sprints*, a través de correo electrónico, nos comunicábamos cuando tenía alguna duda o problema que no me dejara avanzar en el desarrollo de la aplicación para poder solucionarlo.

Por último, mencionar que también, para cada *sprint*, realice un documento en el que se detallaba el trabajo realizado y el proceso llevado a cabo. También se explicaba las funcionalidades o cambios en el código para tener constancia del trabajo realizado y en especial, para elaborar esta memoria posteriormente.

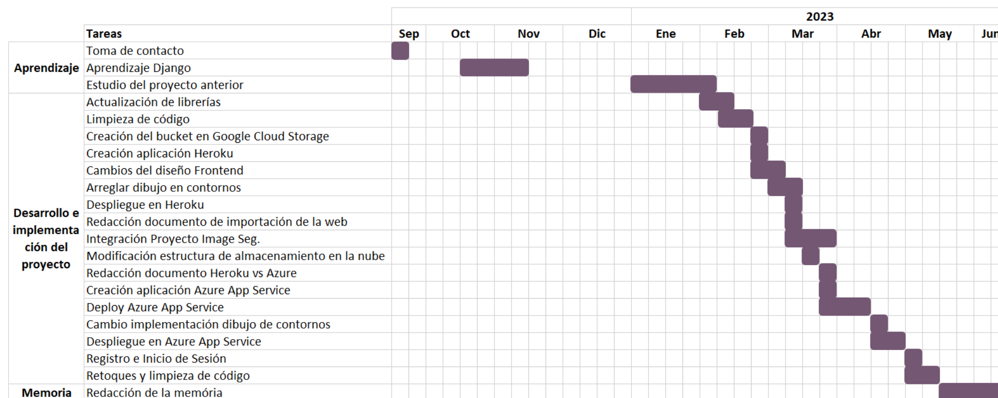


Figure 1: Diagrama de Gantt

Marco teórico

El marco teórico esta compuesto de dos secciones principales, informática y medicina, debido a que en esta memoria se han citado términos que pertenecen a a los dos tópicos y dentro del Proyecto IVUS, la informática y la medicina han mantenido una estrecha relación de colaboración.

2.1 Medicina

DICOM: Un archivo *DICOM* (Digital Imaging and Communications in Medicine) es un formato estándar utilizado para almacenar, transmitir y procesar imágenes médicas y datos relacionados en entornos clínicos. Se utiliza mayoritariamente en el campo de la radiología y otras disciplinas médicas, como la resonancia magnética (RM) y la ecografía. Los archivos *DICOM* contienen tanto los datos de la imagen médica (píxeles, dimensiones, profundidad de bits, etc.) como información adicional asociada, como metadatos del paciente, detalles del estudio, configuraciones de adquisición y datos de contexto clínico.

Imágenes médicas: Son representaciones visuales de estructuras internas del cuerpo humano o de procesos biológicos, adquiridas a través de diversas tecnologías de diagnóstico por imágenes. Desempeñan un papel fundamental en el campo de la medicina, ya que permiten visualizar y analizar el interior del cuerpo para su diagnóstico, tratamiento y seguimiento de enfermedades y condiciones médicas.

Túnica intima: La capa más interna de una arteria o vena se denomina tuncia intima. Las células endoteliales forman su capa única, que es respaldada por una lamina elástica interna. El flujo sanguíneo está en contacto directo con las células endoteliales.

Túnica media: La capa media de una ateria o vena, conocida como el túncia media, está compuesta de células musculares lisas, tejido elástico y colágeno. Está situado entre la túnica externa y la túnica intima. Debido a su color y la forma en que sus fibras están dispuestas transversalmente, se puede identificar desde la capa interna.

2.2 Informática

Librería: En la programación, una *librería* (también conocida como biblioteca) es un conjunto de código predefinido y reutilizable que proporciona funciones, clases y utilidades para realizar tareas específicas. Las l*librerías* son componentes clave en el desarrollo de software, puesto que ahorran tiempo al utilizar el código existente. desplegar.

Framework: En informática es un conjunto de herramientas, *librerías*, componentes y reglas que proporcionan una estructura para las aplicaciones. Un *framework* facilita el proceso de desarrollo al ofrecer un conjunto de funcionalidades predefinidas y estandarizadas, lo que permite a los programadores enfocarse en la

lógica específica de su aplicación en lugar de tener que crear todo desde cero.

Metodología Agile: Es un enfoque de gestión y desarrollo de proyectos que se basa en principios ágiles y valores colaborativos que se enfoca en la adaptabilidad, la colaboración y la entrega incremental de software de alta calidad. Sus objetivos son la satisfacción del cliente, la respuesta rápida a los cambios y la entrega continua de valor.

Sprint: En la *metodología Agile*, es un período de tiempo predefinido y fijo durante el cual se desarrolla un conjunto de tareas específicas para cumplir con los objetivos del proyecto. Un *sprint* típico dura de 1 a 4 semanas y se utiliza para producir un incremento de trabajo potencialmente entregable, como una nueva funcionalidad o un conjunto de características mejoradas.

URL: Una URL es utilizada para identificar la ubicación de cualquier sitio web u otro recurso en línea. Estos tres componentes – esquema, anfitrión y camino – ayudan a su navegador web a localizar y mostrar la información necesaria.

Plataforma como servicio: Es una arquitectura de computación en la nube que ofrece a los desarrolladores una plataforma y un entorno para diseñar, implementar y gestionar aplicaciones sin preocuparse por la infraestructura de soporte.

Machine Learning: La inteligencia artificial (IA) tiene un área llamada machine learning o aprendizaje automático que se centra en la creación de modelos y algoritmos que permiten a los ordenadores aprender y hacer predicciones o juicios sin ser explícitamente programados. Implica la creación de modelos matemáticos y estadísticos capaces de descifrar y analizar grandes volúmenes de datos con el fin de detectar tendencias, predecir el futuro o tomar medidas.

Frontend y Backend: En el desarrollo web, los términos frontend y backend se utilizan para denotar diversos componentes de una aplicación o sistema en línea. El *backend* se refiere a la parte de una aplicación web que gestiona el procesamiento de datos, el almacenamiento y la lógica empresarial, mientras que el *frontend* hace referencia a la porción de un sitio web con la que los usuarios interactúan directamente.

Renderizar:El término renderizar en la programación describe el acto de producir la salida, generalmente en forma de una representación visual o textual, de las entradas o datos.

HTTP: Un protocolo o colección de directrices llamado HTTP (Hypertext Transfer Protocol) describe cómo los datos se envían entre un cliente y un servidor a través de Internet.

Clave Foránea: En un contexto de bases de datos, una clave foránea es aquella que se relaciona con la superclave de otra tabla o entidad para establecer una relación o enlace entre las dos.

Sueprclave: En un contexto de bases de datos, una superclave es aquella o aquellas que identifican inequivocamente una entidad o table.

Tecnologías

3.1 Marco de trabajo

Para el desarrollo del Proyecto IVUS, el *framework* de Django ha sido el utilizado gracias a su facilidad para crear aplicaciones en línea. Se adhiere al patrón arquitectónico de diseño modelo-visualización-controlador (MVC), haciendo hincapié en la reutilización y la accesibilidad de los componentes. Django ofrece una colección de herramientas y módulos que se encargan de muchas tareas típicas de desarrollo web, liberando a los desarrolladores de tratar con detalles tediosos y permitiéndoles concentrarse en construir la lógica única de sus aplicaciones.

Este consta de siete componentes y conceptos, explicados con detalle en el punto 4.3, los cuales son los siguientes:

- **Models**: Representa la estructura de datos y define los campos y relaciones de las entidades.
- **Vistas**: Maneja las respuestas y peticiones, procesa los datos de los modelos y lo envía a lasl plantillas para su representación.
- **Templates**: Genera la salida que los usuarios visualizan combinando HTML con variables, iteraciones y condicionales.
- **Enrrutamiento**: Mapeo de las *URLs* a las vistas específicas, proporcionando una estructura de *URL* flexible y entendible.
- **Forms**: Simplifica en manejo de los forms tradicionales de HTML, incluyendo el manejo de entradas, validaciones y guardado de datos en la base de datos.
- **Autenticación y Autorización**: Proporciona un sistema robusto de autenticación de usuario, administración de sesión y control de permisos.
- **Abstracción de la base de datos**: Proporciona una abstracción de las interacciones de la base de datos usando una capa de mapeo relativo a objetos (ORM), que admite múltiples bases de datos.

3.2 Base de datos

La base de datos usada en el Proyecto IVUS ha sido SQLite, un sistema de gestión de bases de datos relacionales que es ligero, sin servidor y de código abierto. Dado que el motor de base de datos se implementa como una biblioteca dentro del propio programa, SQLite no necesita una configuración de servidor separada o un laborioso proceso de instalación como lo hacen otros sistemas de bases de datos.

En el comienzo del Proyecto IVUS de Daniel Ruiz, se decidió usar SQLite porque no requería una instalación de servidor separado, se minimizaba la latencia de acceso a la base da datos al llamar a funciones y estas son más eficaces en la comunicación entre procesos.

Además, la base de datos se almacena en el ordenador local como un solo archivo, incluyendo todas las definiciones, tablas, índices y datos. Para una base de datos como la del Proyecto IVUS, la cual es bastante simple porque consta de tres entidades, hace que sea beneficioso durante el desarrollo, pero, en un futuro, a

medida que la aplicación se expanda y el número de usuarios aumente, puede no ser suficiente para gestionar la carga adicional. El procesamiento de datos a gran escala o el tráfico pesado generalmente no están destinados a ser utilizados con ordenadores personales, lo que puede causar problemas de rendimiento e incluso accidentes.

3.3 Almacenamiento en la nube

La funcionalidad principal del Proyecto IVUS es la gestión de documentos *DICOM*, por lo que es necesario tener una plataforma donde poder almacenar estos archivos. Para ello, y siguiendo con el proyecto anterior, Google Cloud Storage ha sido el almacenamiento en la nube que se ha usado durante el desarrollo de la aplicación.

Este es un servicio ofrecido por Google Cloud Platform para almacenamiento en la nube en el que los usuarios pueden acceder y guardar datos de forma segura de una manera escalable y duradera.

Dentro de los servicios ofrecidos por Google Cloud Storage, se ha usado un contenedor o Bucket, usado para organizar y manejar almacenamiento de objetos, como ficheros, imágenes, videos, documentos o cualquier tipo de datos. Google Cloud Storage ofrece un sistema de credenciales en los que, a través de un fichero JSON, se pueden conectar al *framework* de Django para poder actualizar información dentro de la aplicación. Esto es gracias a la integración proporcionada por Django a través de la *librería django-storage* y , que permite acceder de una manera sencilla a GCS como *backend* de almacenamiento para la aplicación de Django y *google-cloud-storage*, que proporciona una interfaz para interactuar con el servicio de Google Cloud Storage.

3.4 Despliegue

El despliegue del Proyecto IVUS ha tenido lugar en dos plataformas: Heroku y Azure. Anteriormente, se hizo un despliegue en Heroku, una *plataforma en la nube como servicio* (PaaS). Esta fue escogida, ya que facilita la administración, el despliegue y la escalabilidad.

Las aplicaciones en Heroku se ejecutan utilizando una arquitectura basada en contenedores llamada *dynos*. Los dinos son entornos compactos y asilados que dan los recursos necesarios, como la CPU o memoria para ejecutar un programa. Es posible ajustar la cantidad de dinos para manejar el tráfico en función de las necesidades de la aplicación.

También, gracias a *GitHub*, simplifican el despliegue de la aplicación, ya que Heroku nos ofrece conectar a nuestra aplicación nuestro propio *repositorio*. Esto permite realizar despliegues continuos cada vez que se ejecuta un cambio en *GitHub*, automatizando así el proceso de despliegue de la aplicación.

Las modificaciones en el proyecto de Django son mínimas. Solo es necesario cambiar los *ALLOWED_HOST*, que es la lista que especifica qué nombres de host están permitidos, en el fichero de *settings.py*, en el que se debe añadir la *URL*

proporcionada por Heroku para visualizar la página web como se muestra en la figura 2.

```
ALLOWED_HOSTS = ['tfg-mariaroman.herokuapp.com', '127.0.0.1', '0.0.0.0']
```

Figure 2: Modificación de la configuración de ALLOWED_HOST en Django para depliegue en Heroku

A pesar de todas sus ventajas, tiene una gran limitación, el tamaño de *Slug* permitido. El slug es un paquete que contiene la aplicación web y todos su recursos necesario para ejecutarse. Debido que dentro del Proyecto IVUS se usan *librerías* muy pesadas como es la de *Tensorflow*, el tamaño del *slug* del proyecto es mayor que 500Mb, el tamaño máximo permitido por Heroku para hacer el despliegue de una aplicación. Por ello, se ha tenido que buscar una alternativa para poder realizar el despliegue de la página sin necesidad de dividirla en dos aplicaciones, es decir, una para la gestión de documentos y otra para el cálculo de contornos de capas de un documento *DICOM*.

Para solucionar este problema, se ha usado Azure App Service, un herramienta proporcionada por la plataforma de Azure. Este es un servicio de *plataforma com servicio* (PaaS) que permite la creación, implementación y escalabilidad sencilla de una aplicación web y móviles en la nube. A diferencia de Heroku, el slug máximo permitido en Azure App Service es de 4GB, por lo que ha sido posible el despliegue de la aplicación sin hacer ninguna división de esta.

En términos de escalabilidad, permite escalar automáticamente la capacidad de las aplicaciones en función de la demanda mediante ajustes verticales (cambiando la cantidad de recursos asignados) u horizontalmente (agregando más instancias) para manejar un mayor volumen de tráfico. Esto la hace muy atractiva para cuando la demanda del Proyecto IVUS aumente.

Al igual que Heroku, Azure nos permite asociar nuestro repositorio de GitHub para poder hacer el despliegue de la aplicación de manera sencilla. Adicionalmente, las modificaciones dentro del proyecto Django son simples. Solo es necesario aplicar los mismos cambios que en Heroku, es decir, añadir en el campo de *ALLOWED_HOST* en el archivo de *settings.py* la *URL* que nos proporciona nuestra aplicación de Azure App Service, como se muestra en la figura 3.

```
ALLOWED_HOSTS = ['tfg-mariaroman.azurewebsites.net', '127.0.0.1', '0.0.0.0']
```

Figure 3: Modificación de la configuración de ALLOWED_HOST en Django para despliegue en Azure

3.5 Lenguaje de programación

3.5.1 Backend

El lenguaje de programación usado en le Proyecto IVUS para la parte de *backend* ha sido Python. Es un lenguaje de alto nivel de programación usado para el desarrollo del *framework* de Django. Este *lenguaje* se centra en proporcionar una sintaxis simple y clara y prioriza la legibilidad del código.

Una de sus grandes ventajas y por lo que se ha escogido este lenguaje es que puede ser incorporado dentro de las plantillas de HTLM, el lenguaje usado en la parte de *frontend*, para controlar la *renderización* de los datos.

Algunas de las razones principales para las que Django usa Python son:

- **Simplicidad y legibilidad**: Como se ha mencionado anteriormente, Python tiene una sintaxis que facilita a la creación de código claro y comprensible.
- **Productividad para los desarrolladores**: La filosofía de diseño se centra en la simplificación de las actividades de desarrollo complicadas.
- **Comunidad de desarrolladores grande y vibrante**: Python tiene una comunidad de desarrolladores sana que apoya su expansión y construye un ecosistema enriquecido de *librerías* y *frameworks*.
- **Escalabilidad y adaptación**: Debido a la adaptabilidad de Python, Django puede utilizarse para una variedad de proyectos de desarrollo web. Tanto las aplicaciones de pequeña escala como los sitios web grandes y con un nombre de usuarios amplio pueden ejecutarse en Django.
- **Soporte para bases de datos**: Python proporciona soporte de primer nivel para una amplia gama de bases de datos, incluyendo opciones como PostgreSQL, MySQL y la usada en este proyecto, SQLite.
- **Desarrollo rápido**: El desarrollo rápido es posible por la productividad de las abstracciones de alto nivel de Django y la simplicidad de Python. La funcionalidad integrada de Django, incluyendo autenticación, procesamiento de formularios e interfaces administrativas, acelera el desarrollo.

Dentro de Python, para el desarrollo del proyecto, se han usado distintas *librerías*. Python nos ofrece un amplio ecosistema de bibliotecas, las cuales cubren un amplio rango de funcionalidades que nos ayudan a no empezar de cero. Algunas de las más populares y usadas en este Proyecto IVUS han sido las siguientes:

- **Numpy**: Una *librería*, para computación científica y el trabajo con arrays y matrices.
- **Matplotlib**: Una biblioteca para crear visualizaciones estáticas, animadas e interactivas. Esta se ha sido una de las más usadas en el proyecto para visualizar las capas de los archivos *DICOM*.
- **TensorFlow**: Esta *librería* para *machine learing* y *deep learning*, usada ampliamente para la creación y entrenamiento de redes neuronales. En el proyecto, se ha usado en la implementación de cálculo de contornos en una capa.
- **Pickle**: Es una biblioteca usada para la serialización de objetos. Nos ayuda a guardar y cargar objetos en un formato binario. Dentro del proyecto, se ha

utilizado para cargar archivos *DICOM* locales.

3.5.2 Frontend

Para la parte de *frontend*, se han dos lenguajes: HTML y JavaScript.

HTML es un lenguaje estándar usado para estructurar y presentar contenido en una web. Define la estructura y el *layout* de una página web usando una colección de etiquetas o tags, que son elementos que se utilizan para marcar y estructurar el contenido como encabezados, párrafos, imágenes, enlaces, tablas, y atributos. El código HTML es interpretado por los navegadores web, que luego lo muestran como una página de formato gráfico. Un ejemplo de HTML dentro del proyecto está en la figura 4, donde se muestra como se visualiza la información de un paciente a través del tag **<div>**. Todos los tags van acompañados por un <> al inicio y un </> al final.

```
<div class="row">
    <div class="col">
        <h2 style="margin-bottom: 20px">Información del paciente</h2>
        <p><b>Nombre y apellidos:</b> Pedro Sanchez</p>
    </div>
</div>
<div class="row">
    <div class="col">
        <p><b>Domicilio:</b> Gran Via de les Corts Catalanes 109, 1o 1a</p>
    </div>
</div>
<div class="row">
    <div class="col">
        <p><b>DNI:</b> 39479594L</p>
    </div>
</div>
<div class="row">
    <div class="col">
        <p><b>Número de capas del documento:</b> {{ capas_totales }}</p>
    </div>
</div>
```

Figure 4: Ejemplo Código en HTML del Proyecto IVUS

JavaScript es un lenguaje de programación de alto nivel. Es ejecutar por el navegador web del usuario ya que es un código del lado del cliente. Puede modificar elementos de HTML, manejar eventos, verificar formularios, generar animaciones y realizar una variedad de otras tareas. Ofrece la lógica y las funcionalidades necesarias para hacer que las páginas web sean dinámicas. Un ejemplo del uso de JavaScript dentro de la aplicación es el dibujo de una capa de un documento *DICOM*. En la figura 5 se muestra la función connectTheDots que conecta los puntos a medida que se van dibujando en la capa.

```
<script>
        const canvas = document.getElementById("myCanvas");
        const ctx = canvas.getContext("2d");

        // Cargar la imagen
        const img = new Image();
         img.crossOrigin =  'anonymous'
        img.src = document.getElementById("myImage").src;
        const points = []

       // Función para conectar los puntos
        no usages   mariroman99
        function connectTheDots() {
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            ctx.drawImage(img, 0, 0);

            ctx.setLineDash([5, 5]); // ponemos la linea con puntos
            ctx.beginPath();
            ctx.moveTo(points[0].x, points[0].y);
            for (let i = 1; i < points.length; i++) {
              ctx.lineTo(points[i].x, points[i].y);
            }
            ctx.stroke();
            ctx.setLineDash([]);
          }
</script>
```

Figure 5: Ejemplo Código en JavaScript del Proyecto IVUS

HTML y JavaScript se combinan para crear una experiencia de web atractiva e interactiva. La estructura de la página y contenidos es proporcionada por HTML mientras que JavaScript añade interacción entre el usuario y la aplicación.

Antecedentes

4.1 Funcionalidades

Como se ha mencionado anteriormente en esta memoria, el Proyecto IVUS fue heredado por el alumno Daniel Ruiz, el cual realizo varias funcionalidades dentro de la aplicación para poder gestionar los documentos *DICOM*. Estas eran las siguientes:

- **Subir un documento**: Esta funcionalidad permite al usuario subir un archivo a la sección de archivos para su posterior visualización. Solo están permitidos los archivos *DICOM*. En el caso de que el usuario quiera subir otro tipo de archivo se le denegará a través de una *HttpResponseBadRequest*. Después de la comprobación, el archivo es procesado y se crea un objeto Documento, el cual es guardado en la base de datos a través del método *save()*.
- **Eliminar un documento**: Como su nombre indica, la funcionalidad de eliminar documento elimina el archivo seleccionado de la base de datos a través de su id, el cual es pasado en la URL *eliminardocumento/<id>*. El objeto es eliminado a través de la función *delete()*.
- **Visualizar un documento por capas**: La visualización de un documento por capas se realiza a partir de la llamada a la *URL visualizardicom/<id>/<capa>*, la cual ejecutando la función de *descargardicom*, accede a la base de datos para recuperar dicho doumento a través del id. Mediante *pydicom dcmread*, lee el documento recuperado y con el método *dumps* de la librería *pickle*, lo serializa para ser visualizado. La función *descargardicom* apunta a la *URL visualizardicom/<capa>/<capas_totales>/<ruta_archivo_local>/<id>*, la cual tendrá la información necesaria para que se puedan generar las visualizaciones. Para la visualización de una capa, se usa la función *imshow* de la *librería matploblib*, la cual recibe una matriz de píxeles extraída del archivo *DICOM*. Posteriormente, la visualización es guardada en un archivo HTML con la función *guardar_fig_en_html*, usando la librería *mpld3*.
- **Generar la sección de un documento**: Los datos de todas las capas de un documento *DICOM* se interpolan para formar la sección, que es una representación longitudinal. El documento se divide en bloques. La sección se genera utilizando un índice vertical que evita la exhibición de componentes desde el eje central. Se generan columnas amarillas para indicar la posición de la capa visualizada, y se añaden las capas no correspondientes a la que se quiere visualizar a la lista. Luego, se traspone la matriz de columnas para visualizarla horizontalmente y se usa la función *imshow* para visualizarla y el método *guardar_fig_en_html* para guardarla en un archivo HTML.
- **Dibujar sobre una capa**: Para esta funcionalidad se uso la librería de *matplotlib*, la cual permite usar ventanas interactivas. Gracias a la función *connect:* de esta librería, que permite conectar un evento de interacción del usuario, como el clic del ratón, se pudo realizar el dibujo dentro de la capa y capturar los puntos dibujados. A través de líneas, estos eran conectados entre

si y guardados en el directorio *media/<id>/contours/*, dentro de un archivo de tipo texto.

- **Descargar los contornos de una capa**: Esta funcionalidad, como su nombre indica, era capaz de descargar los contornos dibujados por un usuario en forma de fichero *.txt* en el que se visualizaban los puntos dibujados de una capa.
- **Descargar los contornos de todas las capas**: Similar a la funcionalidad anterior, esta permitía al usuario descargar en formato *.zip* todos los ficheros *.txt* de las capas dibujadas.

4.2 Base de Datos

Adicionalmente, creo y estructuró la base de datos, la cual tiene un diseño sencillo, ya que, dentro del Proyecto IVUS, solamente existen dos entidades: *Documentos* y *Capa*, que es una entidad débil, ya que sus atributos no la identifican completamente, sino que necesita información de Documentos para poder ser reconocida.

Documento, como se muestra en la figura 6 tiene como *superclave* un *id*, el que lo identifica inequívocamente, mientras que Capa, al ser una entidad débil, tiene como atributo *superclave* el *id* y una *clave foránea* que es *documento.*

La relación entre Capa y Documento siempre será de *1:N*, es decir, de uno a muchos, donde un documento puede tener muchas capas y una Capa pertenece a un Documento, ya que cuando una existe una entidad débil, siempre tendremos esta relación.

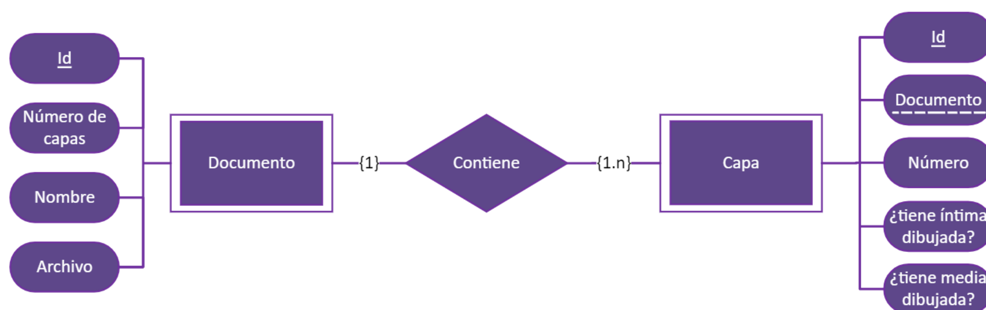


Figure 6: Diagrama antiguo entidad relación Proyecto IVUS

Entrando más en detalle de los distintos atributos de las dos entidades, tenemos primero el Documento, la entidad. Esta se compone como, se ha mencionado anteriormente, de un id, el cual es la *superclave*, que es creada por Django aleatoriamente. Seguidamente, tenemos el *Archivo*, que es el fichero *DICOM* que el usuario sube a la aplicación. También tenemos un atributo Capas, que nos indica el número de capas que hay dentro del Documento para facilitar la navegación al visualizarlo, y finalmente tenemos el atributo *Nombre*, el cual, como su nombre indica, es el nombre del archivo subido.

La entidad Capa, consta también de una *superclave id*, la cual, de la misma forma que en la entidad Documento es creada por Django de manera aleatoria. También, como se ha mencionado anteriormente, tenemos un atributo Documento, que es la clave foránea que une una Capa al Documento que pertenece. Seguidamente, tenemos el atributo *Número de la capa*, que nos indica la posición de la Capa dentro del Documento. Finalmente, tenemos dos atributos booleanos que son *¿Tiene íntima dibujada?* y *¿Tiene media dibujada?*. Estos nos indican si el usuario ha dibujado un contorno de tipo *íntima* o *media* en la capa.

4.3 Interfaz

El Proyecto IVUS que realizo Daniel Ruiz se cimentaba sobre el conjunto de principios y leyes de percepción de la Gestalt, las cuales describen como percibimos y organizamos la información visual. Estos se basan en la hipótesis de que la mente humana tiende naturalmente a organizar los componentes visuales en patrones significativos y cohesivos. Las leyes de la Gestalt que se usaron y siguen aplicándose en este Proyecto IVUS son:

- **Ley de la proximidad**: Los elementos que están cerca unos de otros a menudo se ven como un grupo o una unidad.
- **Ley de la similitud**: Los objetos que se agrupan visualmente a menudo tienen atributos similares, como forma, tamaño o color
- **Ley de la continuidad**: La mente tiende a percibir formas como líneas o curvas continuas en lugar de cambios rápidos o bruscos.
- **Ley de la clausura**: Para crear un todo lógico, la mente se esfuerza por rematar formas o figuras incompletas.

Debido a que el objetivo principal del Proyecto IVUS era la optimización de la utilidad de las distintas funcionalidades y no enfocarse en un diseño muy elaborado, se decidió crear un *Frontend* minimalista y que redujera el número de llamadas al servidor. Para ello, se usaron distintos *templates* en *HTML* y *JavaScript* para el desarrollo de la página web.

Inicio y Gestión de documentos

La página de inicio o *landing page* y la de gestión de documentos constaban de un diseño muy sencillo, en el que, en la página de inicio, como se observa en la figura 7 se encontraban dos elementos: Un logotipo de la página donde clicando nos dirigía a la página de gestión de archivos, y una barra de navegación con un único botón, que nos llevaba al inicio del portal.

Seguidamente, nos encontrábamos con la sección de gestión de documentos, donde se visualizaba en la parte izquierda, como se ve en la figura 8, un listado que contenía en una columna los distintos archivos *DICOM*, otra con un botón para poderlos visualizar y otra para eliminarlos. También, en la párte derecha, se hallaba con un componente que permitía al usuario poder subir al portal archivos *DICOM*.

Figure 7: figure
Página Inicio Antigua Proyecto IVUS

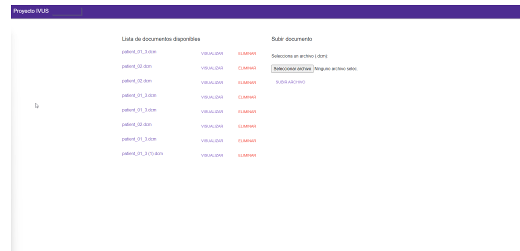
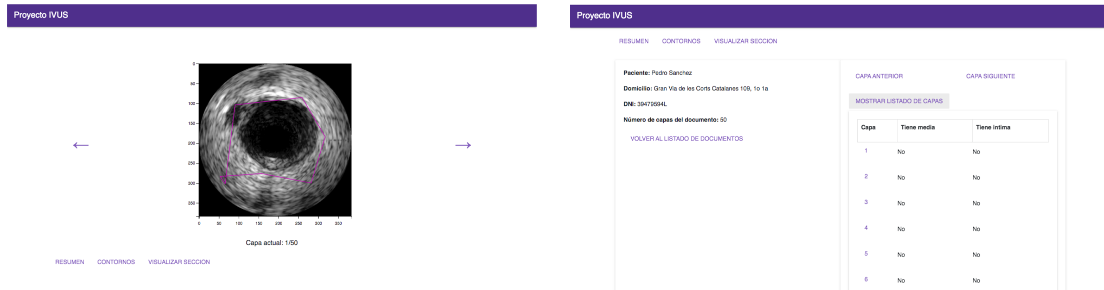


Figure 8: figure
Página Gestión Archivos Proyecto IVUS

Visualización de capas de un documento

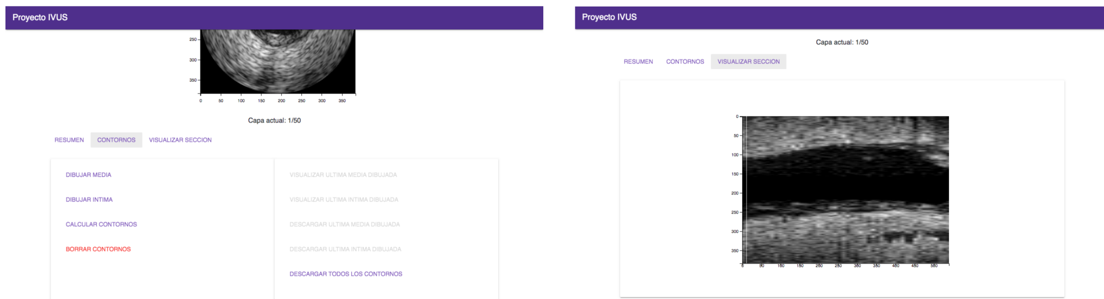
La página de visualización de capas de un documento, como se visualiza en la figura 9a, al igual que la de gestión de documentos, tiene un diseño simple, ya que siempre se impuso el correcto funcionamiento de las distintas funcionalidades ante el diseño del portal web. En esta se visualizaba la capa del documento. En la parte inferior de la visualización de la capa, había un menú horizontal con tres secciones: un resumen, en la figura 9b, con la información del paciente; los contornos, como se ve en la figura 10a en los que se le daba la opción al usuario a dibujar y visualizar los contornos de una capa y por último, visualizar sección, como se observa en la figura 10b en la que se observaba la sección completa del documento.



(a) Página de visualización de capas de un documento

(b) Página de resumen en visualización de capas

Figure 9: Páginas de visualización de capas de un documento



(a) Página de contornos en visualización de capas

(b) Página de sección en visualización de capas

Figure 10: Páginas de visualización de capas de un documento

Estructura Código

5.1 Funcionamiento Django

El funcionamiento de Django se basa en un contexto de interacción de usuario, enrutamiento de *URLs*, vistas, modelos y plantillas. Cada elemento será explicado con más detalle posteriormente pero, para tener una idea general de como funciona la arquitectura de Django, en la figura 11 podemos visualizar todo su proceso.

Este proceso empieza con la interacción del usuario con el navegador, accediendo a una *URL* concreta o sometiendo un formulario. Esta petición pasa por el patrón de *URLs*, que mapea la *URL* recibida a una vista en particular. El gestor de eventos, también conocido como *Resolver de URL* actúa como gestor que examina los patrones de *URL* y determina qué función de vista o clase debe manejar la solicitud. Una vez pasada la petición a la vista, esta contiene la lógica para procesar la solicitud, interactuar con los modelos y realizar operaciones con la base de datos.

Después del procesamiento de la petición e interacción con los modelos, la vista prepara los datos para ser *renderizados* en la respuesta. Estos datos son enviados a las plantillas, ficheros HTML que definen la estructura y presentación de la respuesta. El resultado del procesamiento de los datos *renderizados* en las plantillas es una representación en HTML de la respuesta.

Finalmente, esta respuesta es enviada al navegador del usuario, el cual puede visualizar la salida de su petición inicial.

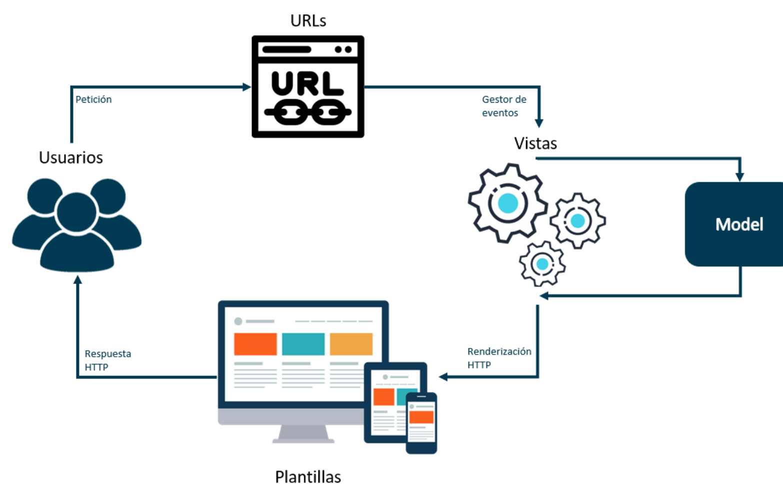


Figure 11: Funcionamiento del framework de Django

5.2 Modelo de datos

En Django, los modelos son componentes fundamentales que definen la estructura y el comportamiento de nuestros datos almacenados en la base de datos. Un modelo en Django es una clase de Python heredada por *django.db.models.Model*. Cada uno de los atributos representa un campo en la base de datos, en el que se especifica el tipo de dato y cualquier información adicional. En la figura 12 se muestra la entidad Documentos con sus atributos. Cada uno está especificado por un tipo. Entre ellos, podemos encontrar *IntegerField* o *TextField*.

```
class Document(models.Model):
    docfile = models.FileField(upload_to='media')
    capas = models.IntegerField(default=1, null=True)
    nombre = models.TextField(null=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE, default=1)
```

Figure 12: Ejemplo Modelo Documento Proyecto IVUS

Dentro del modelo, también se definen las relaciones entre otros modelos. Estas relaciones se establecen para conectar y asociar distintos modelos entre ellos.

Las principales relaciones que tiene Django son:

- **ForeignKey**: Esta define una relación de uno a varios. Representada por un campo *ForeignKey*. En esta relación, un modelo tiene una *clave foránea* que apunta a la clave primaria de otro modelo.
- **ManyToManyField**: La relación de varios a varios es representada por el campo *ManyToMayField*. En esta relación múltiples instancias de un modelo pueden ser asociadas a múltiples instancias de otro.
- **OneToOneField**: Esta es usada para definir la relación de uno a uno. Es representada por el campo *OneToOneField*. Una instancia de un modelo esta asociada a exactamente otra instancia de otro modelo, y viceversa.

5.3 Vistas

Las vistas en Django o views son clases o métodos de Python que aceptan solicitudes *HTTP* y devuelven respuestas *HTTP*. Manejan la lógica de la aplicación y eligen qué datos obtener o alterar y cómo entregarlo al usuario.

Las vistas sirven como un puente entre la base de datos y el navegador web del usuario, lo que le permite manejar las entradas de los usuarios, procesarlas y *renderizarlas* en las plantillas para producir el resultado HTML final.

Las vistas pueden ser definidas en Django usando funciones o clases:

- **Vistas basadas en funciones**: Estas se basan en funciones simples de Python, las cuales tienen por parámetro una petición y retornan un objeto respuesta. Un ejemplo de una vista basada en funciones dentro del Proyecto

IVUS es la *views.py* dentro de la carpeta *members* como se puede observar en la figura 13.

- **Clase basada en vistas**: Estas clases heredan de la clase *View* o de sus subclases de Django y se encargan de la gestión de distintos metodos *HTTP* como *GET, POST*, etc. Ofrecen mayor flexibilidad y organización del código comparado con vistas basadas en funciones. Como, dentro del Proyecto IVUS, ninguna función se limita solo a una petición *GET o POST*, no tenemos una vista basada en clase, sino que todas las funciones tienen como petición un objeto.

```python
from members.forms import RegisterUserForm

def login_user(request):
    if request.method == "POST":
        username = request.POST["username"]
        password = request.POST["password"]
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            print("Success")
            return redirect('index')
            # Redirect to a success page.
        else:
            # Return an 'invalid login' error message.
            messages.success(request, ('Ha habido un error al iniciar sesion. Prueba de nuevo'))
            return redirect('login')
    else:
        return render(request, 'authenticate/login.html', {})

def logout_user(request):
    print(request.user)
    logout(request)
    messages.success(request, ('LogOut Success'))
    return redirect('index')
```

Figure 13: Vista basada en funciones en el Proyecto IVUS

5.4 Plantillas

Las plantillas en Django son ficheros que especifican la organización y diseño de las páginas HTML que produce la aplicación web. Permiten desconectar la lógica de la aplicación de la presentación visual, lo que mejora la reutlización y mantenimiento del código. El motor de plantillas en Django ofrece un mecanismo fuerte y adaptable para producir páginas web dinámicas.

Algunos aspectos importantes de las plantillas de Django son los siguientes:

- **HTML usando Tags de plantillas**: La mayoría de las plantillas de Django están escritas en HTML y son capaces de incluir cualquier código HTML válido. Sin embargo, contienen etiquetas de plantilla, que son construcciones únicas rodeadas por los delimitadores *{% %}* o *{{ }}*. En la figura 14 podemos ver una parte de la plantilla *archivos.html* en la que se visualiza le uso de los delimitadores.
- **Elementos dinámicos**: Las plantillas permiten incluir elementos dinámicos desde la vista, tales como variables. Estas pueden ser enviadas desde la vista a la plantilla usando un contexto *renderizado*, el cual es un diccionario de

objetos. Dentro de la vista, como se muestra en la figura 15 con el nombre del documento, es necesario usar *{{ }}*.

- **Herencia de templates**: Las plantillas the Django tienen una gran ventaja y es que pueden heredar de otras. Esto permite crear una plantilla base para toda la aplicación y heredarla en el resto de plantillas. Dentro del Proyecto IVUS, se ha usado este aspecto para la barra de navegación, la cual esta implementada en el fichero *base.html* y mediante *{% extends 'base.html' %}*, como se observa al principio de la figura 14.

```
{% extends 'base.html' %}
{% load static %}

<html>
<head>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
    <meta charset="utf-8">
    <title>Archivos</title>
</head>
{% block body %}
    <body>
    <div class="container">
    <h1 style="...">Documentos</h1>
        <form id="form">
            <input type="search" id="myInput" onkeyup="searchTable()" placeholder="Search documents" style="...">
         <button style="background: #5e478a; color: white; border-radius: 10px; border:none; width: 80px"
                onclick="clearInput()">Reset</button>
        </form>
```

Figure 14: Plantilla de HTML en el Proyecto IVUS

```
<h6 style="...">Todos los documentos disponibles</h6>
<table id="myTable" style="...">
    {% if documents %}
                {% for document in documents %}
                    <tr id="{{ document.nombre }}" style="...">
                        <td style="..." >
                            <form action="{% url 'descargardicom' document.id  1 %}" method="post"
                                     enctype="multipart/form-data">
                                    {% csrf_token %}
                                    <input type="submit" class="btn btn-text btn-primary" style="..." value={{ document.nombre }}
                                          />
                                </form>
                        </td>
```

Figure 15: Variables en plantillas en el Proyecto IVUS

5.5 Enrutamiento

En Django, el enrutamiento se refiere al proceso de mapear las *URLs* hacia las funciones o las clases de las vistas que se encargan de manejar estas *URLs*. Este proceso es imprescindible dentro de las aplicaciones web ya que se determina como las peticiones serán procesadas y que vistas son responsables de generar las respuestas apropiadas.

Dentro de Django, los dos componentes principales que se encargan del enrutamiento son las vista, explicadas en el punto 5.3, y un patrón de *URLs*.

Este patrón de urls está definido en el Proyecto IVUS como *urls.py* tanto en la carpeta *rekProject*, que es el archivo principal que redirige a los otros archivos *urls.py*, como en las apps de *rek* y *members*. En la figura 16 podemos visualizar el archivo *urls.py* de la carpeta *rekProject* , donde se redirige a los archivos *urls.pyrek* y *members*.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include("rek.urls")),
    path('members/', include('django.contrib.auth.urls')),
    path('members/', include('members.urls')),
```

Figure 16: Archivo urls.py de rekProject en el Proyecto IVUS

También, en la figura 16 se visualiza como todos los patrones usan *path()*. Esta sintaxis es ofrecida por Django y es la más común para definir un patrón de *URL* gracias al modulo *django.urls*. Esta incluye tres componentes imprescindibles:

- **Ruta**:Este primer argumento corresponde a la ruta, que es representada por un cadena que describe el patrón de URL. Puede contener componentes estáticos, definidos con / / o con componentes dinámicos, definidos con $< >$. Estos dos se observan en la figura 17.
- **Vistas**: El siguiente argumento son las funciones o clases que deben manejar las peticiones. Importando el fichero de las vistas, podemos llamar a la función pertinente para que se ejecute la petición y nos devuelva la respuesta. Podemos observar su implementación en la figura 17.
- **Nombres**: El último elemento dentro de la función *path()* son los nombres. Estos nos permiten poder llamar el patrón de url dentro de los templates para que se puedan ejecutar las peticiones. También, pueden ser usadas en las vistas para poder referenciar los patrones de *URLs*. Como en los dos puntos anteriores, podemos observar su implementación en la figura 17.

```
urlpatterns = [
    path('', views.index, name='index'),
    url('archivos', views.list, name='archivos'),
    url('logout', views.logout, name='logout'),
    path('download-folder/<id>', views.download_folder, name='download_folder'),
    path('visualizardicom/<id>/<capa>', views.descargardicom, name='descargardicom'),
    path('eliminardocumento/<id>', views.eliminardocumento, name='eliminardocumento'),
```

Componente estático
Componente dinámico
Vista
Nombre

Figure 17: Componentes archivo urls.py de rek en el Proyecto IVUS

5.6 Formularios

Los formularios de Django ofrecen un enfoque práctico para gestionar las entradas de los usuarios, verificar los datos y producir componentes de formularios HTML. El sistema de formularios en Django ofrece una interfaz de alto nivel para interactuar con los formularios HTML y está construido sobre las clases de Python.

A continuación está una visión general de cómo funcionan los formularios en Django:

- **Definición de formularios**: Es necesario crear una clase de Python que herede de la clase *django.forms* para poder construir un formulario en Django.

Los formularios son subclases de esta. En los formularios, se pueden definir campos y especificar el tipo de dato y cualquier propiedad adicional. En la figura 18 se observa el formulario de registro, el cual usa el formulario predefinido *UserCreationForm* de *django.contrib.auth.forms*.

- **Renderización de formularios**: Los formularios en Django pueden producir componentes de formulario HTML automáticamente. Enviando una instancia de la clase formulario a la plantilla, se *renderiza* ese formulario. Este generará de manera automática la información necesaria en HTML para que el usuario pueda visualizarlo. Como se puede observar en la figura 19, el formulario *form* es el que se pasa por contexto desde la vista a la plantilla y se *renderiza* a HTML.
- **Validación de formularios**: Cuando un usuario tramita un formulario, Django automáticamente realiza una validación basada en la definición del formulario. Comprueba si la información proporcionada por el usuario es del mismo tipo que la definida en los campos del formulario. Si la información no es válida, Django mostrará un mensaje de error en el campo correspondiente.
- **Procesamiento de información de formularios**: Cuando el usuario ha tramitado la información y se ha validado que corresponde con el tipo de datos, puede ser procesada en la vista creando instancias de la clase formulario. Esta clase proporciona métodos y atributos para obtener los datos validados o limpiar el campo. También pueden ser útiles para subir documentos.

```
class RegisterUserForm(UserCreationForm):
    email = forms.EmailField(widget=forms.EmailInput(attrs={'class': 'form-control'}))
    first_name = forms.CharField(max_length=50, widget=forms.TextInput(attrs={'class': 'form-control'}))
    last_name = forms.CharField(max_length=50, widget=forms.TextInput(attrs={'class': 'form-control'}))

    mariroman99
    class Meta:
        model = User
        fields = ('username', 'first_name', 'last_name', 'email', 'password1', 'password2')
```

Figure 18: Formulario de registro en el Proyecto IVUS

```
<form action="{% url 'register_user' %}" method=POST>
    {% csrf_token %}
    {% for field in form %}
    <div style="...">
      <label for="{{ field.id_for_label }}" class="custom-label"
             style="font-size: 20px; margin-top: 20px">
          {{ field.label }}</label>
      {{ field }}
    </div>
  {% endfor %}

    <br/><br/>
  <div style="...">
      <button type="submit" style="...">Submit</button>
    </div>
 </form>
```

Figure 19: Formulario de registro renderizado en la plantilla register_user.html en el Proyecto IVUS

Implementación

6.1 Base de datos

Durante el desarrollo de la base de datos del Proyecto IVUS, no ha habido grandes cambios respeto al año pasado. Como bien se ha mencionado en el punto 4.2, la base de datos utilizada para la aplicación ha sido SQLite.

Debido a que se ha implementado un sistema de registro y autentificación en la página web, funcionalidad explicada posteriormente, se ha añadido una nueva entidad llamada Usuario. Esta consta de un *username*, que es la *superclave* que identifica inequívocamente a un usuario, el *correo electrónico*, un *Nobmre* y *Apellido* por separado y finalmente una *contraseña*, para poder identificarse cuando inicia sesión.

Esta modificación en la base de datos ha provocado un cambio en la antigua. La implementación del registro y la autenticación ha supuesto que cuando un usuario intenta acceder a la aplicación, no puede realizar ninguna funcionalidad sin haber iniciado sesión previamente. Esto supone un cambio en la entidad de documento, dado que la inexistencia de un usuario implica necesariamente la inexistencia de un documento asociado.

Por ello, la entidad *Documento* ha sido transformada a una entidad débil que debe ser respaldada por un *Usuario*.

También, entre las entidades *Usuario* y *Documento*, como se observa en la figura 20, se ha creado una relación de cero a varios. Esta describe una conexión en la que una entidad puede tener cero o muchas relaciones con otras entidades.

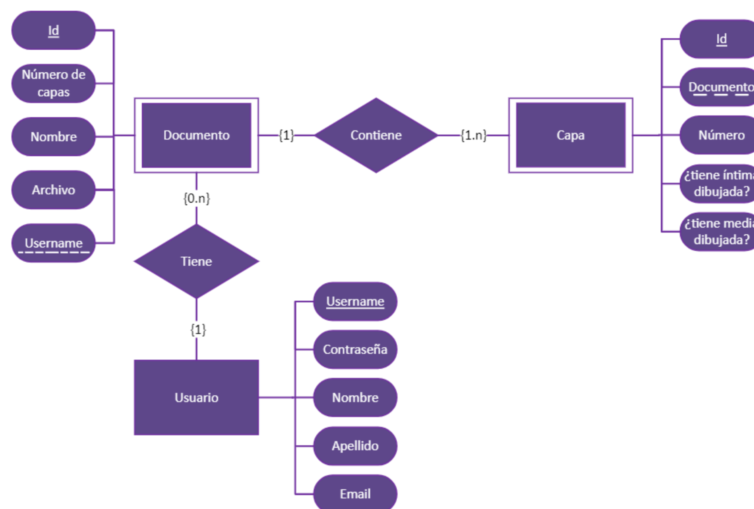


Figure 20: Diagrama entidad relación Proyecto IVUS

En un extremo de la línea de relación hay un signo "uno", y en el otro, hay un símbolo "cero o muchos". El signo "uno" indica que debe haber al menos una relación entre las entidades primarias y secundarias, pero el símbolo "cero o muchos" denota

que puede haber cero o varios enlaces entre los entes principales y secundarios.

Finalmente, para que la entidad Documento sea una entidad débil de *Usurio*, ha sido necesario añadir la *clave foránea username*, la cual identifica a la entidad Usuario y hará que el documento, junto a su *id* sean únicos.

6.2 Almacenamiento en la nube

Una de las partes más importantes dentro del Proyecto IVUS ha sido el almacenamiento en la nube de los ficheros *DICOM*. Como se ha mencionado en el punto 3.3, se ha usado el servicio Bucket de Google Cloud Storage, en el que, además de los documentos, se encuentran todos los archivos estáticos como son los archivos *css* o imágenes como el logo de la página web o de botones.

Al principio del desarrollo de la aplicación, la estructura dentro del Bucket tenía un diseño simple, en el que todos los archivos *DICOM* eran almacenados en una carpeta llamada documentos.

Durante la implementación de la aplicación, la organización de los ficheros dentro del Bucket ha sido modificada para proporcionar robustez y una mejor gestión de todos los archivos. Para ello, se ha creado una estructura donde, la carpeta llamada *media* almacena para cada documento una carpeta independiente, identificada con su id. Otro de los motivos por los que se ha creado un directorio para cada documento es que, debido a que actualmente la aplicación guarda todas las imágenes dibujadas y sus ficheros de coordenadas en la nube, era necesario una organización para la comprensión y legibilidad dentro de las funciones de la vista.

En la figura 21 se visualiza la estructura interna del Bucket. Dentro de las sub carpetas que tiene la carpeta de *contours* se almacenan las imágenes dibujadas y los ficheros de texto con los puntos del dibujo de las capas.

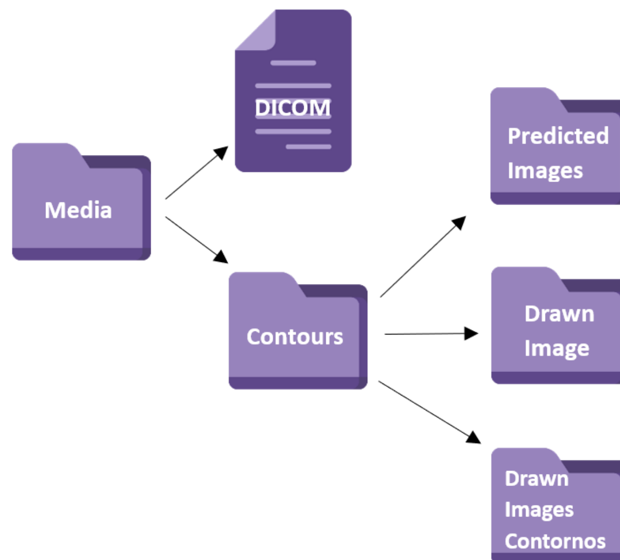


Figure 21: Esturctura interna del Bucket de Google Cloud Storage

Cuando un usuario sube un documento a la aplicación, a la vez que se crea la entidad Documento, se crea y se sube la carpeta junto al documento al Bucket, a través de la función *upload_to_path* dentro del fichero *models.py*. De la misma forma, cuando un usuario dibuja sobre una capa, los ficheros resultantes, es decir, la imagen dibujada y el archivo de puntos, son guardados en el Bucket.

Para poder realizar estos cambios y subir los distintos archivos desde el *framework* de Django ha sido necesaria la importación de las *librerías* mencionadas en el punto 3.3. También, era imprescindible crear un fichero de credenciales para acceder automáticamente al Bucket desde el proyecto.

Dentro del código, han sido imprescindibles cambios y adiciones respecto a las que se realizaron en el proyecto inicial, en la que se añadió en las apps instaladas *storages* en el archivo *settings.py*, ya que esto indica a Django que incluya la funcionalidad proporcionada por *django-storages*, la *librería* que nos permite usar diferentes servicios de almacenamiento en la nube.

Entre ellos ha estado la cambio de la variable *GS_BUCKET_NAME* en el archivo *settings.py*. Esta define el nombre del bucket de Google Cloud Storage y, debido al cambio de cuenta y la creación de un nuevo bucket para seguir con el Proyecto IVUS, ha sido modificada con el nuevo nombre del bucket, como se observa en la figura 22.

```
from google.oauth2 import service_account
GS_CREDENTIALS = service_account.Credentials.from_service_account_file(os.path.join(BASE_DIR, 'credentials.json'))
DEFAULT_FILE_STORAGE = 'storages.backends.gcloud.GoogleCloudStorage'
GS_BUCKET_NAME = 'tfg-mariaroman'
STATIC_URL = "https://storage.googleapis.com/{}/".format(GS_BUCKET_NAME)

STATICFILES_DIRS = (os.path.join(BASE_DIR, "static"), )
STATICFILES_STORAGE = 'storages.backends.gcloud.GoogleCloudStorage'
```

Figure 22: Configuración Django para el uso de Google Cloud Storage

A diferencia del Proyecto IVUS inicial, se ha usado el almacenamiento en la nube no solo para almacenar los archivos *DICOM*, sino también para guardar ficheros como las imágenes dibujadas y sus coordenadas. Por ello, ha sido necesario añadir en las vistas tanto la clase *storage* del módulo *google.cloud*, la cual proporciona una interfaz que interactúa con los servicios de almacenamiento en la nube de Google, como la primera línea que se muestra en la figura 23, que indica y carga el fichero de credenciales de autenticación JSON para acceder a Google Cloud Storage.

```
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "credentials.json"
from google.cloud import storage
```

Figure 23: Configuración en views.py para interactuar con Google Cloud Storage

Este almacenamiento de todos los archivos en la nube actúa como una copia de seguridad. Si ocurre algún problema en la aplicación, los documentos seguirán estando seguros y accesibles en la nube. También aporta mayor seguridad, ya que los proveedores de servicios en la nube generalmente implementan medidas de seguridad más robustas para la protección de dichos archivos.

6.3 Funcionalidades

Dibujar una capa

Como bien se ha mencionado en el apartado 4.1, la funcionalidad de dibujar una capa fue implementada en el Proyecto IVUS anterior, en la que se usaba la *librería matplotlib* para de desplegar una ventana interactiva para poder dibujar sobre una capa. Esto funcionaba de manera local, pero cuando se tuvo que hacer el despliegue tanto en Azure como en Heroku, no funcionó en ninguna de las plataformas debido a que estas no soportan el uso de ventanas interactivas en aplicaciones. Por ello, se tuvo que buscar una alternativa y rehacer la funcionalidad desde cero.

Mencionar que esta función permite dibujar tanto la capa *íntima* como la *media*. También es posible dibujar encima de una capa en la que se han calculado los contornos. Para ello, cuando el usuario está visualizando una capa, en el apartado de contornos, como se muestra en la figura 24, puede escoger que capa quiere dibujar y si quiere dibujar encima de una capa sin o con cálculo de contornos.

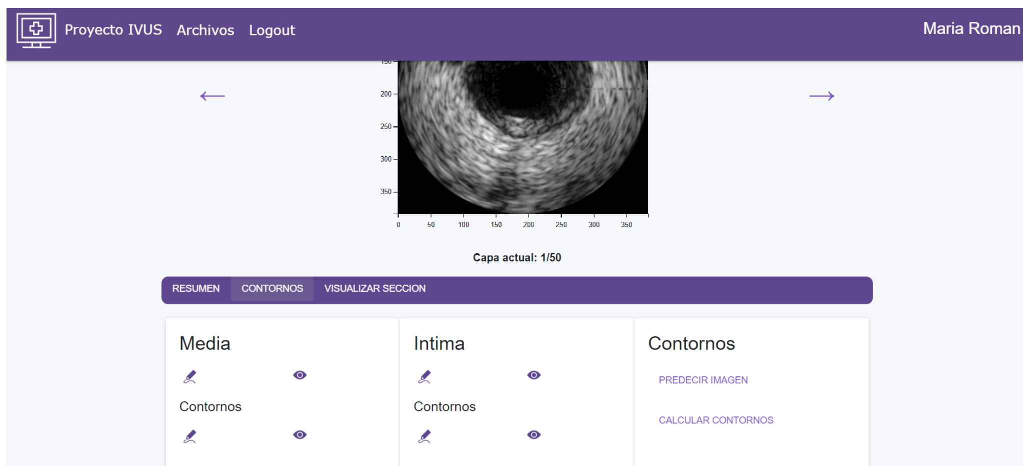


Figure 24: Página de visualización de las opciones de dibujo de una capa del Proyecto IVUS

Esta funcionalidad ha sido realizada a través de JavaScript, en concreto con el elemento canvas de HTML, que se utiliza para dibujar gráficos dinámicos y animaciones en el navegador. Cuando un usuario se dispone a dibujar una capa, se le redirige a la *URL habilitardibujadodicom/capa/capas_totales/ruta_archivo_local/media_o_intima/id/*. El método de la vista al cual el patrón de *URL* dirige la petición es el llamado *habilitar_dibujo*. Primero, a través de la *librería pickle*, leemos el archivo. Luego, miramos qué tipo de capa es la que se debe dibujar, una con contornos o sin contornos. Este paso es esencial porque, dependiendo si la capa tiene contornos calculados o no, será guardada en el Bucket en un directorio u otro. Por eso, debemos tratar los dos tipos de capas de manera distinta.

En los dos casos, se generará una imagen tipo *png*, debido a que dentro de un elemento canvas no podemos añadir directamente la capa de un documento *DICOM*,

sino que debe tener un formato de imagen. Esta imagen se guardará en el Bucket para su posterior recuperación en la plantilla. Para ello, se ha usado el paquete *google.cloud.storage*, que permite configurar un objeto cliente de Google y llamar a un Bucket en concreto de ese cliente.

Cuando el Bucket está recuperado, a través de la función *blob*, creamos un objeto con el directorio donde debe ser guardada la imagen. Llegados a este punto, debemos hacer la diferenciación entre una capa con contornos calculados y sin calcular, ya que el directorio de una y de otra será distinto. Para las imágenes sin cálculo de contornos, el directorio donde se guardará la imagen es *DrawnImages* y para las imágenes con contorno será *DrawnImagesContonros*. Una vez la imagen a dibujar se ha subido, nos dirigirá a la *URL* mencionada, donde se creará un elemento canvas con la imagen creada en el Bucket.

El funcionamiento de dibujar en un canvas se encuentra en la plantilla *dibujarcapa.html*. En esta, se desarrollan un seguido de funciones para poder capturar los clics (*addPoint*), dibujar un círculo (*drawCicrle*) y conectarlos entre sí (*connectTheDots*). También, se guardan las coordenadas de cada punto en un *array* de puntos, para que el usuario pueda visualizarlo posteriormente. Adicionalmente, como se muestra en la figura 25, el usuario puede borrar un punto o todo el dibujo de la capa. También, puede subir a un documento de puntos que se dibujarán en la capa visualizada.

Figure 25: Página de dibujo de una capa del Proyecto IVUS

Una vez el usuario ha acabado de dibujar, debe guardar el dibujo, porque si no lo hace, al salir de la página de dibujo, se le notificará que no ha guardado el dibujo.

Una vez el usuario guardé el dibujo, el patrón de *URL* llamará a la función de *habilitar_dibujo*, en la que, a través del valor "pintado" obtenido por parámetro de la solicitud *HTTP POST*, él cuál nos indicará si se ha dibujado sobre una capa, ejecutará la función *upload_image_to_google* que a través de la obtención los datos de la capa dibujada mandados por parámetro, igual que el valor "pintado", hace una decodificación de estos a base64 y crea un objeto imagen. Después el proceso de subir la imagen al Bucket es el mismo que cuando no está dibujada.

Finalmente, de la misma manera que se guarda la imagen dibujada en el Bucket, también se sube el archivo de las coordenadas donde se han dibujado los distintos puntos. Para ello, se crea un documento de tipo texto que se rellena con la lista de puntos que se ha pasado por parámetro de la solicitud *HTTP POST*. Este fichero es luego subido al mismo directorio en el que se ha subido la imagen dibujada.

Visualizar capa dibujada

Para la visualización de una capa dibujada, como su nombre indica, es necesario que el usuario haya dibujado una capa. En caso contrario, no le aparecerá ninguna imagen ni las opciones de descargar los archivos relacionados con esta.

En caso de que si se haya dibujado en la capa, se encontrará con la página que se visualiza en la figura 26.

Figure 26: Página de visualización de una capa dibujada del Proyecto IVUS

Cuando el usuario, desde la visualización de capa, se dirige a la visualización de capa dibujada, se le redirige a la *URL visualizardicomdibujada/capa/capas_totales/ruta_archivo_local/media_o_intima/id/*, en el caso de que la capa no tenga contornos o a la *URL visualizardicomdibujadacontornos /capa/capas_totales/ruta_archivo_local/ media_o_intima/id/* en el caso que sí. La diferencia entre una y otra es que las funciones que ejecutan en la vista tienen directorios distintos, ya que como se ha mencionado anteriormente, los dos tipos de imágenes se almacenan en carpetas distintas. Sin embargo, el funcionamiento de los métodos es el mismo.

La recuperación del fichero de coordenadas de puntos dibujados se realiza a través de la función *download_as_string().decode('utf-8')* de la librería *google.cloud.storage*, donde se descarga el archivo del Bucket de Google Cloud Storage como una cadena de texto para luego, pasarlo por parámetro a la plantilla. En el caso de la imágen, es más sencillo. Lo único que se hace es crear el directorio donde está la imagen guardada con los parámetros de ruta recibidos en la *URL*, especificamente el id, el número de capa, y la media_o_intima. Seguidamente, se

valida que el directorio existe dentro del Bucket. En caso afirmativo, devuleve por parámetro dicho directorio, él cuál, a través de la expresión de *{% static filePathName %}*, tomará el valor de la variable filePathName y lo combinará con la configuración del directorio de archivos estáticos definida en Django para generar la URL completa del archivo estático.

Descargar capa dibujada

Dentro de esta funcionalidad, hay dos tipos de archivos los cuales el usuario puede descargar de una capa dibujada. Una es la imagen en formato *png* dibujada y otra el fichero tipo texto con las coordenadas de los puntos dibujados. También, el usuario tiene dos sitios en la página web para descargar estos documentos.

La primera es durante el dibujado de una capa. Cuando un usuario está dibujando una capa, puede descargar en cualquier momento tanto la imagen como el fichero tipo texto. Para descargar la imagen, se hace a través de la función de JavaScript de *downloadCanvas*, que a través de obtener el elemento canvas que se muestra en la pantalla y convertirlo en una *URL* que contiene la representación de la imagen, se descarga en el dispositivo local del usuario. Para el fichero de coordenadas, se crea un documento Blob que respresenta un archivo de texto con el contenido de los puntos. Un Blob es una parte de la API de JavaScript y se utiliza para representar datos binarios o de texto en bruto en forma de objeto. Para que sea de tipo texto, le especificamos el tipo "text/plain", el cual es un archivo de texto sin formato. Seguidamente, creamos una URL única para el objeto Blob, para poderlo descargar en un dispositivo local.

La segunda opción que tiene el usuario para descargar la capa dibujada es a través de la visualización de esta, como se muestra en la figura 26. En esta pantalla se encuentran, en la parte inferior derecha, dos iconos para descargar tanto las coordenadas como la imagen dibujada. Como bien se ha explicado anteriormente, los puntos son recuperados al visualizar la capa dibujada y el proceso para descargarlos es el mismo que cuando el usuario se encuentra en la primera situación, donde está dibujando la capa, al visualizar una capa, el parámetro es una lista de puntos.

Eliminar documento

Esta funcionalidad no se ha implementado de cero, simplemente se ha modificado para su correcto funcionamiento. En el Proyecto IVUS anterior, cuando un usuario eliminaba un documento, se llamaba a la función de *eliminardocumento* de la vista y se eliminaba solo de la base de datos. Esto hacía que, tanto en la carpeta *media/id* que se crea para guardar el documento *DICOMO* en local como en el Bucket no se eliminara dicho documento. Por ello, este problema ha sido solucionado para proporcionar un correcto funcionamiento de esta funcionalidad. Cuando un usuario quiere eliminar un documento, se obtiene el parámetro de ruta id que se pasa en la *URL eliminardocumento/id*. Primero, se comprueba si dicho documento existe en la base de datos, a través de obtenerlo de la base de datos. En caso afirmativo, se elimina y seguidamente se borra la carpeta *media/id* relacionada con este archivo.

Una vez eliminada la carpeta, se elimina del Bucket de Google Cloud Storage. Para ello, se obtiene el Bucket a través de la *librería google.cloud.storage*. Dentro del Bucket, se busca la carpeta coincidente con la ruta *media/id*. Si esta existe, se itera sobre todos los objetos del directorio y se elimina uno a uno.

Finalmente, cuando la carpeta que contiene el documento y sus archivos adicionales se ha eliminado, el usuario es redirigido a la plantilla *archivos.html*, donde se visualizan todos los archivos del usuario.

Calcular y visualizar contornos de una capa

Para el cálculo de los contornos de una capa, se implementó dentro del Proyecto IVUS el proyecto de Irene Garacia, en el que calcula contornos de una capa a través de *machine learing*.

Cuando un usuario quiere calcular los contornos de una capa en la sección de contornos de una capa, como se visualiza en la figura 24, se la redirige a la *URL visualizarcontornodicom/capa/capas_totales/<ruta_archivo_local/id/*. A través del patrón de *URLs*, se llama a la función *calcular_contornos* de la vista. En esta se crea el directorio donde se va a guardar la imagen con los contornos calculados. Seguidamente, se llama a la función *leer_puntos_algoritmo* del fichero *contornos.py*. En este fichero se calculan los contornos de una capa a través de *machine learning* con las *librerías* de *tensorflow*. El color de la *túncia media* y de la *túncia íntima* es diferente para que el usuario pueda diferenciar entre una y otra.

El uso de las *librerías* de *tensorflow* junto con las que se usan en el Proyecto IVUS hizo que, el año pasado, no se pudiera desplegar la aplicación en Heroku, como se ha mencionado anteriormente.

Una vez se han calculado los contornos de la capa, se retornará a la vista, la cual guardará la figura en HTML a través de la funcion *fig_to_html* de la *librería mpld3* para ser renderizada en la plantilla *visualizardicom.html*. De esta forma, el usuario podrá visualizar la capa con los contornos dibujados, al igual que una capa normal, como se visualiza en la figura 27.

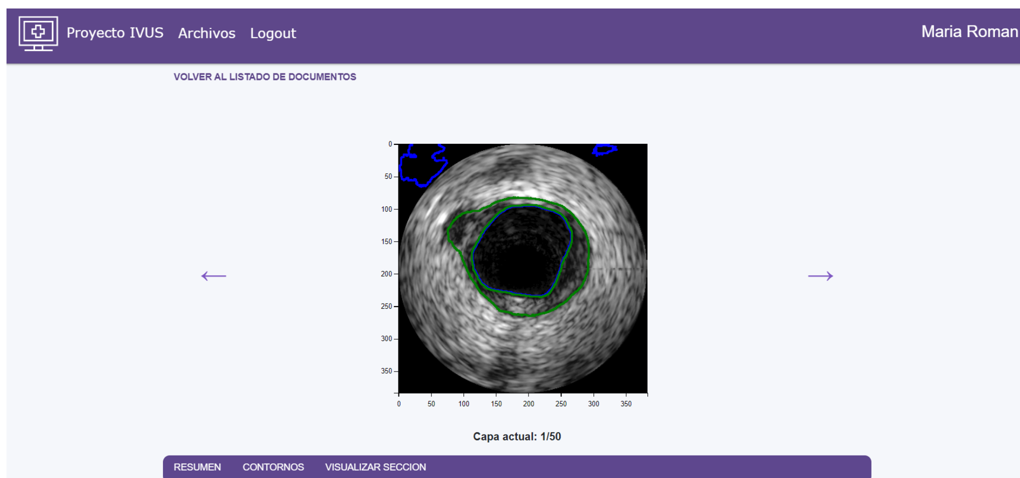


Figure 27: Página de visualización de una capa con contornos del Proyecto IVUS

Segmentación y visualización de una capa

Además de calcular los contornos, como se muestra en la funcionalidad anterior en el proyecto de Irene Garcia, se implementó un clasificador de imágenes, él cuál, a partir de una imagen y a través de *machine learning* y las librerías de *tensorflow*, podía segmentar las imágenes para, en el caso de una un documento *DICOM* poder diferenciar las distintas capas.

Este proyecto fue el que se subió a Heroku, ya que se hizo de manera independiente sin ser adaptado al Proyecto IVUS, por lo que no se utilizaban el resto de *librerías* usadas en el proyecto y el tamaño de la aplicación comprimida era menor. Solo se segmentaban las imágenes que se subían a través de un elemento *input file* de HTML a la aplicación en concreto.

Sin embargo, ha sido posible implementar la segmentación al Proyecto IVUS para que el usuario pueda tener otra visión de la capa y sus contornos.

Para ello, como se muestra en la figura 24 en la parte inferior derecha, está el botón predecir imagen. Cuando el usuario clica en este, se le redirige a la *URL predictedImages/capa/capas_totales/ruta_archivo_local/id/*, la cual llama a la función *habilitar_prediccion*. Dentro de esta, se lee y se carga el archivo *DICOM* a través de la *librería pickle*.

Seguidamente, ya que para segmentar es necesaria una imagen de tipo *.png*, será imprescindible la generación de una imagen del archivo *DICOM*. Para ello, se crea una figura a través del archivo leído con la *librería* de *matplotlib* y mediante la función *savefig* de la misma *librería*, se guarda la imagen en local. Una vez generada la imagen, gracias a la función *predictImageBucket*, se calcula la segmentación de la imagen generada con *machine learning*. La imagen resultante es subida al Bucket de Google Cloud Storage. Esta función también genera el directorio en el cual se guarda la imagen para, luego, *renderizarlo* en la plantilla *visualizarcapacalculada.html*. Dentro de esta, se visualizará a través de la expresión *{% static filePathName %}*, donde el filePathName será el directorio que se ha pasado por parámetro, como se

visualiza en la 28.

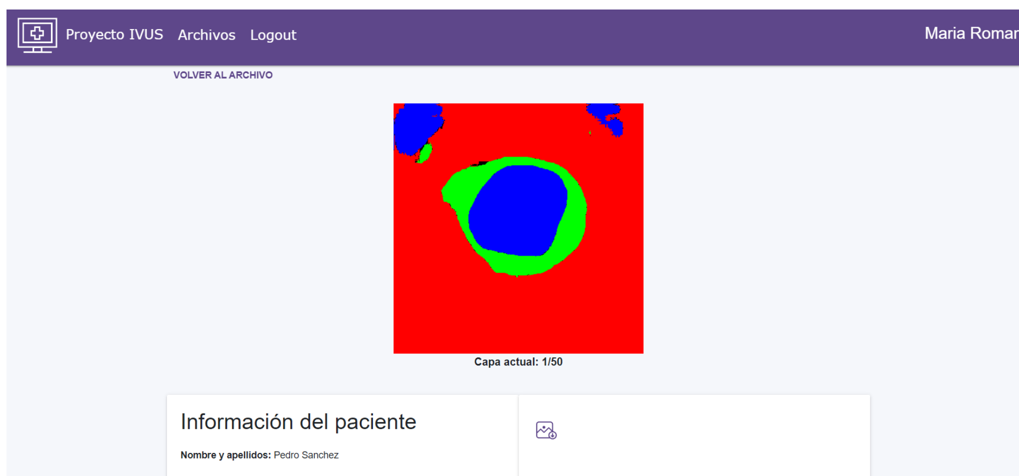


Figure 28: Página de visualización de una capa segmentada Proyecto IVUS

Descargar carpeta entera

En esta funcionalidad, un usuario puede descargar todos los ficheros relacionados con un documento, es decir, tanto el documento *DICOM*, como las imágenes dibujadas y las coordenadas de los puntos dibujados. Para ello, en el menú principal de visualizar una capa, como se muestra en la figura 29, a través del botón "Descargar carpeta", el patrón de *URLs* llamará a la función *download_folder*, la cual a través de la *librería zipfile*, generará una carpeta *.zip* con todos los documentos y carpetas asociadas al documento con el id enviado por parámetro. Finalmente, crearemos una respuesta y le añadiremos el fichero para que pueda ser descargado.

Figure 29: Página de visualización de una capa Proyecto IVUS

Buscar documentos

Cuando un usuario está visualizando todos los documentos, tiene la opción de encontrarlo a través del buscador que se encuentra en la parte superior de la página, como se visualiza en la figura 30.

Esta funcionalidad se ha implementado a través de JavaScript, en la que se itera sobre todos los documentos de un usuario y se muestran los documentos con el mismo nombre que se ha introducido en el buscador.

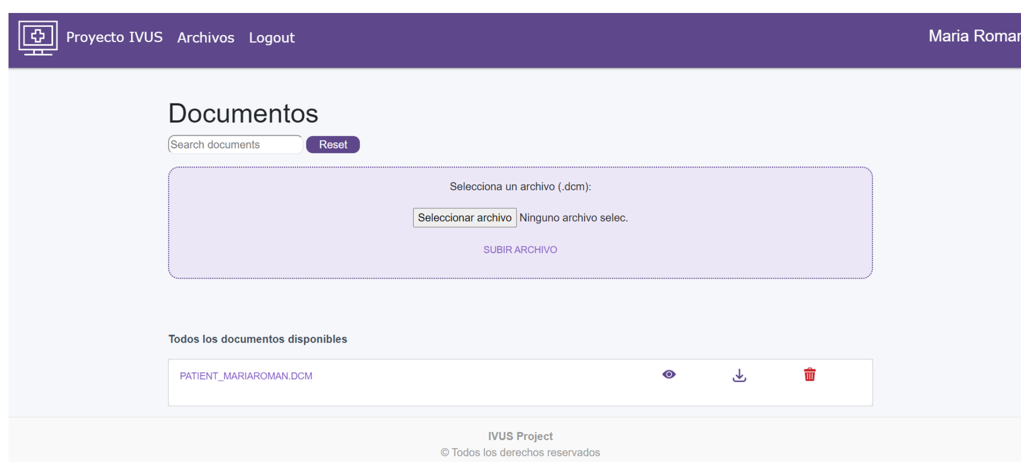


Figure 30: Página con todos los archivos de un usuario del Proyecto IVUS

Registro

El registro de un usuario era una funcionalidad atractiva dentro de la aplicación, ya que permite que cada cliente de la página web pueda tener una cuenta propia en la que tenga sus propios documentos *DICOM*. Para la implementación de esta funcionalidad, se ha creado una nueva aplicación dentro del Proyecto IVUS llamada *members*, en la que se encuentran las implementaciones de esta función y la de inicio de sesión.

Con la ejecución de esta funcionalidad, un usuario que entra por primera vez a la aplicación debe registrarse para poder usarla. Para ello, deberá dirigirse a la barra de navegación horizontal que está en la parte superior y clicar el botón de registro. Esto redirige al usuario a la *URL /members/register_user*, donde aparecerá un formulario con los campos que aparecen en la figura 31. Al rellenar los parámetros del formulario, se le notificará al usuario cuando alguno de estos no tenga el formato correspondiente. En concreto son el campo de email, el cual debe contener un formato de correo electrónico, y la contraseña, que debe tener una mayúscula y minúscula, un número y un símbolo para ser válida.

Figure 31: Registro de la aplicación del Proyecto IVUS

Este formulario es recibido en las *views*, donde se comprueba si es válido. En caso afirmativo, se crea un formulario de registro de usuarios a través de la clase *UserCreationForm* del módulo *django.contrib.auth.forms* con los campos proporcionados por esta clase que son el nombre de usuario, una contraseña y una contraseña de confirmación. Adicionalmente, para tener más información de un usuario, se añade al usuario un email, con un tipo *EmailField*, un nombre y un apellido con tipo *CharField*.

Cuando el usuario ha sido creado, gracias a la función *authenticate* del módulo *auth* de Django, se autentifica a un usuario en el sistema y con la función *login* del mismo módulo se inicia la sesión.

Iniciar Sesión

A diferencia de la funcionalidad anterior, en esta el usuario ya se ha registrado y posee una cuenta en la aplicación. Por ello, cuando llega a la *landing page*, se encuentra con la opción de *Login* en la barra de navegación superior. El usuario es redirigido al formulario que se encuentra en la *URL /members/login_user*. En este punto, el usuario visualiza un formulario como el que se muestra en la figura 32.

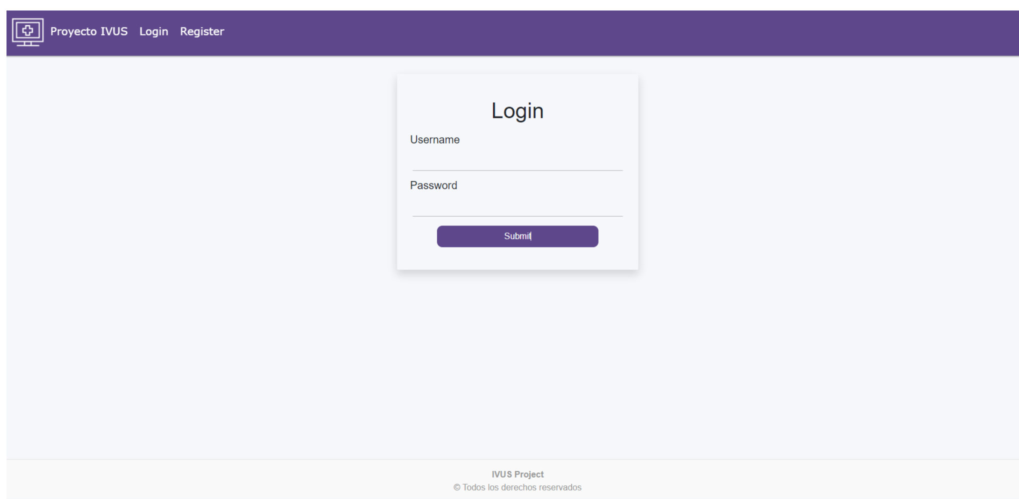


Figure 32: Inicio sesión de la aplicación del Proyecto IVUS

En este punto, se envía una petición *POST* a la función *login_user* de la vista, la cual obtiene el nombre de usuario y su contraseña y con la función *authenticate*, mencionada anteriormente, que autentifica al usuario. Si este usuario existe, de la misma forma que en registro, se usará la función *login* para iniciar sesión con el usuario validado. En el caso de que el usuario no exista, se volverá a recargar la página donde se encuentra el formulario de inicio de sesión con el mensaje de "Ha habido un error al iniciar sesión. Prueba de nuevo", para que el usuario vuelva a iniciar sesión.

Una vez el usuario haya iniciado sesión con éxito, se le redirigirá a la *landing page* en la que podrá visualizar sus archivos.

Logout

Esta funcionalidad permite al usuario cerrar sesión en la aplicación. Para ello, debe haber iniciado sesión, ya que si no lo le aparecerá la opción. Para ejecutar esta función, es necesario dirigirse a la barra superior en la que se encuentra el botón de *Logout*. Al clicarlo, se ejecutará la función de *logout_user* de las vistas, la cual a través de la función *logut* del módulo *auth* de Django, se hará el cierre de sesión y devolverá al usuario a la página principal.

6.4 Modificaciones Frontend

Dentro de *frontend* se han realizado distintos cambios para que se le facilite la navegación a un usuario. El primero es en la página principal. Anteriormente, cuando un usuario entraba, no quedaba claro donde tenía que clicar para poder visualizar los documentos *DICOM*. Para ello, se ha implementado en la barra de navegación un botón llamado "Archivos", él cuál redirige al usuario a la página donde se visualizan todos los documentos. Este cambio hace que se cumpla la ley de Hicks, que nos dice que el tiempo se invierte para tomar una decisión incrementa con

el número de opciones y la complejidad disponibles. Con esto, el usuario dispone de un botón fácil y comprensible.

Otro de los cambios que se ha realizado es en la visualización de documentos. Se han añadido tres botones con icono, como se ve en la figura 33, para cada documento, en los que el usuario tiene tres opciones posibles: Visualizarlo, descargarlo o eliminarlo. Aplicando el principio de reconocimiento de patrones, los iconos visualmente claros y comprensibles facilitan la visualización y comprensión del usuario.

Figure 33: Iconos de un documento de la aplicación del Proyecto IVUS

Costes

Para el desarrollo del Proyecto IVUS, han sido necesarias tres plataformas para poder llevar a cabo el funcionamiento y testeo de la aplicación. Gracias a las distintas subscripciones gratuitas y de estudiantes que ofrecen estas plataformas, no ha habido ningún gasto económico ni parte del alumno ni el tutor. El posterior funcionamiento y producción de la aplicación conllevará unos gastos que, a continuación, se detallaran para los distintos servicios usados.

7.1 Google Cloud Storage

El uso de almacenamiento en la nube de Google Cloud Storage utilizado durante el desarrollo del proyecto, ha tenido coste cero gracias a la suscripción gratuita de 90 días y $200 de crédito que se le proporciona a un nuevo usuario. Para la puesta en producción del Proyecto IVUS, los costes que comportarían los servicios de Google Cloud Storage con una ubicación en Madrid(europe-southwest1) serían los siguientes:

Almacenamiento estándar (por GB al mes)	**Nearline Storage** (por GB al mes)	**Coldline Storag** (por GB al mes)	**Almacenamiento de archivos** (por GB al mes)
$0.023	$0.013	$0.006	$0.0025

Figure 34: Tabla Costes Google Cloud Storage

7.2 Heroku

Heroku, igual que Google Cloud Storage, nos ha ofrecido una suscripción de estudiante, la cual nos ha permitido poder usar esta plataforma que comporte ningún gasto para la realización del Proyecto IVUS. Esta suscripción ofrece un crédito de $13 por mes durante un año. Esto ha permitido poder usar la plataforma sin coste alguno, tanto para la aplicación de gestión y visualización de archivos como para la de cálculo de contornos.

Para el testeo de la aplicación, se han usado dos planes Eco Dynos. El coste total que ha supuesto de los créditos ofrecidos ha sido de $15,07.

Desplegar y poner en producción el Proyecto IVUS, siempre teniendo en cuenta que es necesario tener dos aplicaciones por separado, sería necesario usar el plan Standard 2X que proporciona Heroku para ejecutar aplicaciones comerciales en producción. También, para proporcionar una base de datos SQL administrada, sería necesario solo para la aplicación de administración y visualización de ficheros un plan Standard 1X especifico para apps comerciales en producción. Este supondría unos costes como los que se reflejan en la Figura 35.

Heroku Pricing Estimate

Created: May 5, 2023
Available until: November 11, 2023

Est. monthly cost $250
2 apps

App 1	my-app-estimate-1		$150
	Dynos		
	Standard 2X	2 instances $50 per plan	$100
	Data Services		
	Postgres Standard 0	1 instance	$50
App 2	my-app-estimate-2		$100
	Dynos		
	Standard 2X	2 instances $50 per plan	$100

Included

- Up to 5 Heroku Teams with 1-5 members each (additional cost for larger Teams)
- Heroku Pipelines and Heroku Review Apps
- Standard support during business hours with 1+ day response

Estimated monthly cost
$250

Figure 35: Costes Heroku

7.3 Azure App Service

La alternativa que se ha usado a Heroku para hacer el testeo y despliegue de la aplicación ha sido Azure App Service. Esta, al igual que las otras plataformas, proporciona una suscripción gratuita de 12 meses con un crédito de $200. Durante el desarrollo del Proyecto IVUS, ha sido la plataforma usada con más frecuencia para las distintas pruebas de funcionamiento, ya que solo ha sido necesario tener una aplicación en despliegue. Por ello, ha tenido unos costes de $87,42 dentro del crédito ofrecido. Para poder llevar a producción el proyecto, necesitaríamos un plan Estándar S1 con un plan de base de datos SQL de uso general. Esto comportaría unos gastos como los que se encuentran en la Figura 36.

App Service | Nivel Estándar; 1 S1 (1 núcleos, 1.75 GB de RAM, 50... | Por adelantado: 0,00 ... | Mensualmente: 70,68...

App Service

Región: West Europe
Sistema operativo: Linux
Nivel: Estándar

Estándar

INSTANCIA: S1: Núcleos: 1, 1.75 GB de RAM, 50 GB de almacenamiento, 0,095 US$

1 Instancias × 744 Horas = 70,68 US$

Conexiones SSL

Si costo inicial 0,00 US$
Costo mensual 70,68 US$

Soporte

SOPORTE: Incluido 0,00 US$

Seleccione su programa u oferta

PROGRAMA DE LICENCIAS: Contrato de cliente de Microsoft (MCA)

Mostrar Precios De Desarrollo Y Pruebas

Costo inicial estimado 0,00 US$
Coste mensual estimado 70,68 US$

Figure 36: Costes Azure App Service

Para el plan de Data Service, no existe un coste determinado, sino que, Azure, tiene costes estimados y sus precios dependen de los Gb por mes que se usen. El plan de Uso General que se muestra en la Figura 37. Sería el idóneo, ya que es un plan de servicio de nivel básico y medio que es adecuado para cargas de trabajo de bases de datos de tamaño mediano.

Azure SQL Database

Plan de database	**Memória mínima**	**Memoria máxima**	**Precio**	**Almacenamiento localmente redundante**	**Almacenamiento de copia de seguridad**	**Retención a largo plazo**
Serie estándar (Gen 5)	2,02 GB	240 GB	$0,57/hora de núcleo virtual	$0,14/GB/mes	$0,119/GB/mes	$0,0298/GB/mes

Figure 37: Costes Azure Database Service

Conclusiones

8.1 Resultados

Mirando los objetivos del Trabajo de fin de Grado en el capítulo de introducción, uno de ellos, él cuál era fundamental su correcto funcionamiento, era poder dibujar en una capa. A pesar de que se ha invertido más tiempo del esperado, ya que antes de que se implementara esta función con JavaScript, se estuvieron probando distintas formas de que funcionara a través de una ventana interactiva tanto en Azure como en Heroku, ha sido finalmente posible sacarla adelante y adicionalmente poder añadir opciones como un botón de borrar o el dibujo de puntos a través de la subida de un fichero.

También, otro de los objetivos principales de este trabajo era la subida a producción de la aplicación web. A pesar de que no ha sido posible usar Heroku, se ha buscado otra solución, Azure, sin comportar coste alguno, que ha hecho posible que se haya podido desplegar el proyecto completo. Este objetivo también ha hecho posible que el proyecto de machine learning para calcular contornos de una capa haya podido ser añadido al Proyecto IVUS como una funcionalidad más.

A pesar de que no estuviera contemplado en los objetivos, pero si en el trabajo futuro que mencionó Daniel Ruiz en su memoria, se ha podido implementar un proceso de autenticación para que los usuarios de la aplicación puedan tener su espacio personal con sus propios documentos *DICOM*.

En conclusión, todos los objetivos de este trabajo han sido realizados con éxito y adicionalmente, se han podido realizar otras tareas que no estaban previstas para este trabajo, pero que, para el Proyecto IVUS, eran realmente atractivas a la par que necesarias.

8.2 Aprendizaje Personal

Desde una perspectiva personal, en este Trabajo de Fin de Grado he tenido la ocasión de conocer el *framework* de Django y aprender su funcionamiento. También me ha dado la oportunidad de desarrollarme como programadora y aprender a enfrentarme a problemas y resolverlos de forma propia.

El hecho de haber realizado esta aplicación de manera individual me ha permitido tener una visión general del funcionamiento de esta y un control total sobre la misma que nunca había experimentado y que, antes de inmersionarme en el mundo laboral, creo que ha sido una experiencia gratamente beneficiosa para mi crecimiento tanto personal como laboral.

Agradecer al Dr. Simone Balocco por la ayuda proporcionada y la tutoría realizada durante todo el proceso de la implementación del Proyecto IVUS.

8.3 Trabajo Futuro

Debido a que no ha habido tiempo suficiente para acabar de implementara algunas funcionalidades y que, al invertir los primeros meses del Trabajo de Fin de Grado en aprender las tecnologías que se usan en el Proyecto IVUS y el código previo, estas son algunas de las mejoras o nuevas funcionalidades que creo, a nivel personal, que mejorarían la aplicación actual para que fuera más robusta y segura:

- Actualmente, el inicio de sesión de registro está implementado para que un usuario pueda iniciar sesión, registrarse y cerrar sesión. Por ello, una de las posibles mejoras sería añadir opciones de modificación de perfil como puede ser cambiar la contraseña o recuperarla. Esto mejoraría la seguridad de la aplicación y permitiría al usuario tener un control mayor de su cuenta dentro del Proyecto IVUS.
- Otra mejora que podría realizarse y que se mencionó en el Trabajo de Fin de grado anterior es poder recuperar la información de los documentos *DICOM* como la información del paciente al que pertenece. Debido a que se han usado archivos de prueba y por motivos de privacidad de datos, durante la realización de la aplicación no ha sido posible implementar esta mejora. En el caso que la aplicación llegara a ponerse en producción, sería interesante añadir, en la entidad de documentos dentro de la base de datos, ciertos campos que identifiquen al paciente
- Finalmente, una funcionalidad nueva que se podría añadir es poder hacer zoom dentro de una capa, para que el usuario pueda tener una vista más detallada de esta. Esto podría aplicarse tanto a una capa sin dibujar como una con dibujo. No solo sería útil para visualizar la capa, sino que, a la hora de pintar, tanto el dibujo como las coordenadas que se dibujan podrían ser más precisas para un posterior diagnóstico.

Anexo

8.4 Documentos de despliegue

Durante la realización del Proyecto IVUS y como se ha comentado anteriormente, en el comienzo de la aplicación, Heroku fue la plataforma usada para realizar el despliegue. A pesar de que no se pudiera realizar una aplicación conjunta, en esta plataforma, del proyecto de gestión de documentos *DICOM* y del cálculo de contornos a través de *machine learning*, se crearon dos documentos de despliegue, uno para cada aplicación, para poder facilitar la continuación del proyecto. De la misma forma, se hizo otro documento en el que se detallaba como continuar con esta aplicación, usando la plataforma en la que se encuentra actualmente el Proyecto IVUS, Azure.

En estos documentos se detalla la instalación de dependencias dentro del *framework* de Django, la creación de un Bucket en Google Cloud Storage y el despliegue, tanto en Azure como en Heroku.

Bibliografia

[1] ¡Aplica psicología al diseño UI! Leyes de Gestalt | EDteam. (s. f.). https://ed.team/blog/aplica-psicologia-al-diseno-ui-leyes-de-gestalt

[2] Canvas tutorial—Web APIs | MDN. (2023, febrero 19). https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial

[3] Cephalin. (2023, marzo 20). Tutorial: Deploy a Python Django or Flask web app with PostgreSQL - Azure App Service. https://learn.microsoft.com/en-us/azure/app-service/tutorial-python-postgresql-app

[4] Creating Apps from the CLI | Heroku Dev Center. (s. f.). https://devcenter.heroku.com/articles/creating-apps

[5] Django. (s. f.). Django Project. https://docs.djangoproject.com/en/4.2/topics/class-based-views/

[6] Fácil, P. (2023, abril 19). [DJANGO] El uso de plantillas. Programación Fácil Blog. https://programacionfacil.org/blog/django-el-uso-de-plantillas/

[7] Google Cloud Storage—Django-storages 1.13.2 documentation. (s. f.). https://django-storages.readthedocs.io/en/latest/backends/gcloud.html

[8] Free Icons and Stickers—Millions of images to download. Flaticon. https://www.flaticon.com/https%3A%2F%2Fwww.flaticon.com%2F

[9] Ley de Hick y Ley de Fitts aplicadas al diseño UX. uiFromMars. https://www.uifrommars.com/principios-ux-ley-hick-y-fitts/

[10] Matplotlib: Python plotting—Matplotlib 3.1.2 documentation. (s. f.). https://matplotlib.org/3.1.1/index.html

[11] Página principal | Microsoft 365. (s. f.). https://www.office.com/?acctsw=1&auth=2

[12] Reconocimiento de patrones (psicología). (2022). En Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Reconocimiento_de_patrones