

MÓDULO

document_color_meter

Documentación técnica

Descripción: Este documento contiene información técnica del módulo de medición de color de documentos

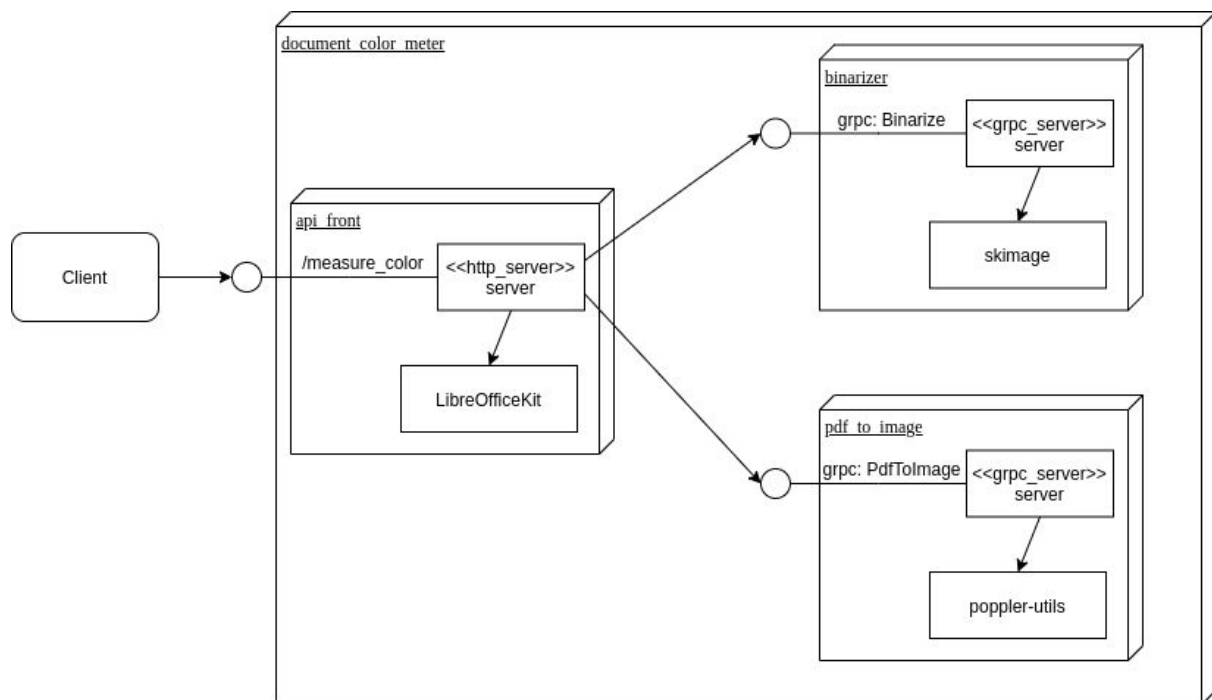
1. Arquitectura:

El módulo está compuesto de tres (3) microservicios.

Su interfaz principal es el endpoint json-rest “/measure_color”.

El microservicio “api_front” se comunica con los microservicios “binarizer” y “pdf_to_image” mediante [gRPC](#).

El siguiente diagrama muestra los componentes del sistema.



2. Requisitos del sistema

El sistema puede correr mediante docker por lo que solo es necesario tener docker y docker-compose para poder ejecutarlo. Así que podrá correrlo bajo linux, windows o mac.

Se recomiendan las siguientes versiones:

docker versión ≥ 19

docker-composer versión ≥ 1.21

Para desarrollo sin docker se recomienda usar Ubuntu. Fue desarrollado usando Ubuntu 19.10 x86_64 como entorno local de desarrollo.

3. Módulos:

3.1. api_front:

Servicio que orquesta las operaciones. Es el que expone la funcionalidad del módulo para que un cliente pueda utilizarlo. Usa libreofficekit para la conversión de los documentos Office a PDF.

Detalles del servicio:

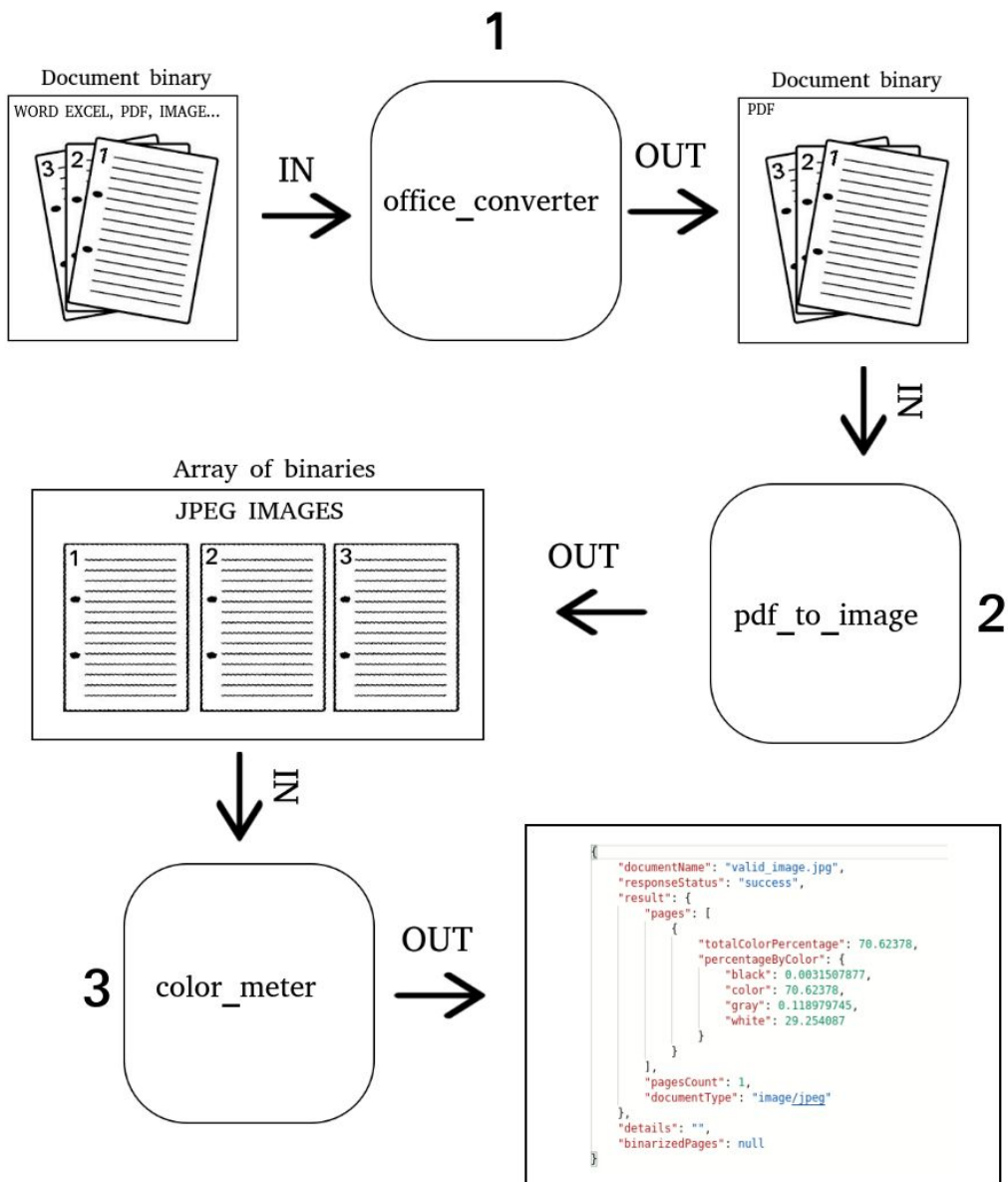
- Lenguaje: Golang versión 1.13. (gotest para test unitarios)
- Framework: gin-gonic/gin
- Sistema de versiones: go modules
- Imagen base de docker: ubuntu:19.10 (la razón es que la versión de libre office que es compatible con LibreOfficeKit solo funcionó de forma estable con esta distro)
- Dependencias del sistema: libreoffice libreofficekit-dev golang-1.13 golang-1.13-go

Endpoint	/measure_color
Descripción	Permite medir la cantidad de color que contiene cada página de un documento
Request	
Campos	<p>binarize (bool, opcional): Indica si se desea retornar las imágenes binarizadas. Si es false o no se indica este parámetro entonces no se retorna la imagen binarizada en formato base 64.</p> <p>whiteThreshold (int, opcional): Indica el umbral de blanco.</p> <p>blackThreshold (int, opcional): Indica el umbral de negro</p> <p>grayThreshold (int, opcional): Indica la máxima desviación para el umbral de gris.</p> <p>file (file): Fichero del documento.</p> <p>Se aceptan los siguientes tipos de archivos:</p> <ul style="list-style-type: none">- imágenes: jpeg, png, tiff, bmp- documentos: doc, docx, ppt, xls, xlsx, pdf, odt
Protocolo	HTTP

Formato	form-data
Response	
Formato	JSON
Campos	<p>documentName: Nombre del fichero del documento indicado en el request.</p> <p>responseStatus: Indica el status de la operación. Los posibles status son los siguientes</p> <ul style="list-style-type: none"> - bad_request: El request no es válido - bad_file: Archivo de formato inválido - empty_file_file: El fichero tiene tamaño cero. - server_error: Algún error desconocido del lado del servidor. - processing_error: Error procesando el archivo - binarizing_error: Error durante la binarización - success: Operación exitosa <p>result: objeto que contiene el resultado de la operación. Dentro de él tiene los siguientes campos:</p> <p>result->pages: Es el resultado de la medición de color para cada página. El índice de cada respuesta estará asociado al número de la página. Por ejemplo: El resultado de la página 1 estará en la primera posición del arreglo.</p> <p>result->pages->totalColorPercentage: Indica el porcentaje de color dentro de la página.</p> <p>result->pages->percentageByColor: Contiene información sobre el porcentaje por cada tipo de color.</p> <p>details: Cuando ocurre un error este campo indica las razones del error.</p> <p>binarizedPages: Imágenes binarizadas en formato JPEG y codificadas en base 64</p>

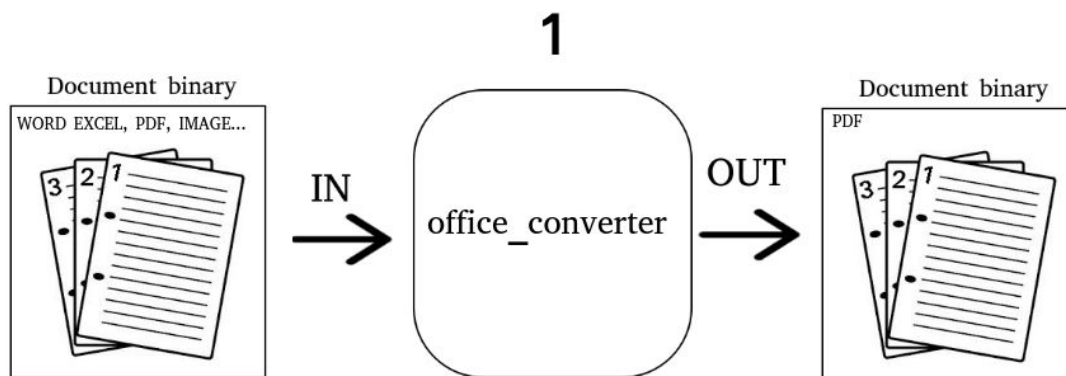
3.1.1. Algoritmo para la medición del color en una página

Para conseguir medir el color de cada página se diseñó un algoritmo de 3 etapas que se describirán a continuación y que pueden ser visualizadas de forma gráfica en el siguiente diagrama:



Etapas 1: Conversión del documento a una PDF.

Puesto que los documentos pueden venir en diversos formatos (office, pdf, etc), la primera etapa consiste en convertir dichos documentos a documentos PDF. Se utiliza el el formato PDF puesto que es el que más implementaciones de rasterización tiene disponible. Lo que facilita cambiar en caso de cambios futuros en el API.



Esta conversión se realiza en el submódulo:

api_front/core/office_converter

Su interfaz principal recibe un documento y retorna otro en formato PDF.

```
type OfficeConverter interface {  
    Convert(doc []byte, outFormat DocumentFormat) ([]byte, error)  
}
```

Para la conversión de documentos Office/OpenOffice a PDF se utiliza el siguiente wrapper de libreofficekit para go:

github.com/dveselov/go-libreofficekit

Se utiliza esta librería puesto que utiliza las librerías nativas de C++ de Libre Office usando CGo.

Nota:

En caso de que el documento sea de tipo imagen o PDF entonces no se realiza esta etapa.

Etapla 2: Conversión de PDF a Imágenes

El submodulo encargado de esta taréa es “pdf_to_image_converter”.

En el está contenido una librería cliente para comunicarse con el servicio gRPC “pdf_to_image”.

El método remoto invocado es “Convert” y el servicio es “PdfToImageService”. El resultado es un arreglo de imágenes en formato JPEG.

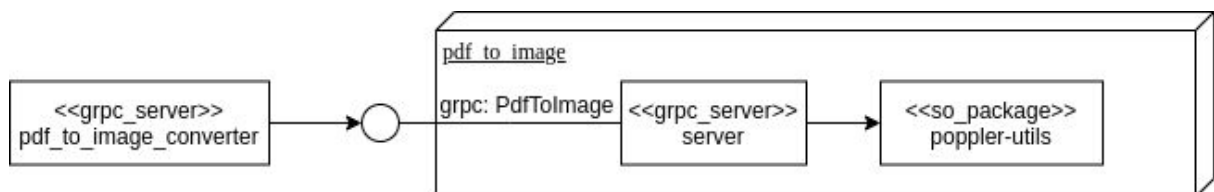
El código descriptor en protocol buffer es:

```
enum ImageFormat {  
    UNKNOWN = 0;  
    JPEG = 1;  
    PNG = 2;  
    SVG = 3;  
}
```

```
message PdfToImageRequest {  
    bytes file = 1;  
    ImageFormat format = 2;  
}
```

```
message PdfToImageResponse {  
    repeated bytes pages = 1;  
}
```

```
service PdfToImageService {  
    rpc Convert (PdfToImageRequest) returns (PdfToImageResponse) {}  
}
```



[illegible]

Rangos de color:

Se definen 4 rangos de color que se evalúan en el siguiente orden:

Blanco (White):

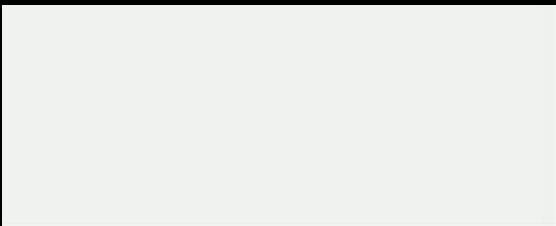
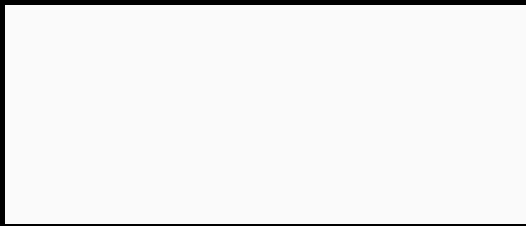
Si la sumatoria de R, G y B es **mayor** que el valor de 'White Threshold' entonces se dice que el píxel es blanco.

White Threshold: Umbral de blanco.

Valor Por defecto: $240+240+240=720$

Puede setear este valor asignando

Ejemplos de color blanco con el 'White Threshold' = 720 (por defecto)

Ejemplo 1: (240,242,240) 722	Ejemplo 2: (250,250,250) 750
	

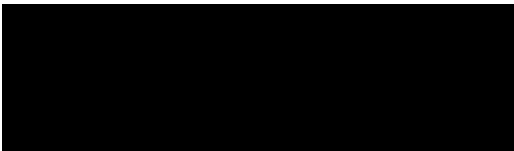
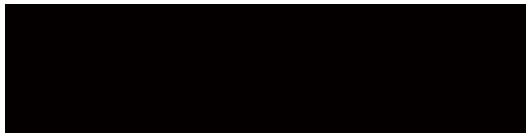
Negro (Black):

Si la sumatoria de R, G y B es **menor** que el valor de BlackThreshold entonces se dice que el píxel es negro.

Black Threshold: Umbral de negro.

Valor Por defecto: $5+5+5=15$

Ejemplos de color blanco con el 'Black Threshold' = 15 (por defecto)

Ejemplo 1: (0,0,0) 722	Ejemplo 2: (5,5,5) 750
	

Gris (Gray):

Si la diferencia entre el mayor y menor valor entre R, G o B es menor que el valor de GrayThreshold entonces se dice que el píxel es gris.

Técnicamente un píxel es gris cuando los valores de sus componentes de colores son idénticos entre ellos.

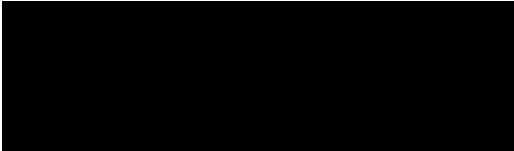
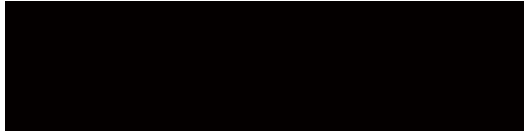
Por ejemplo (50,50,50) o (70,70,70)

Los valores negro absoluto (0,0,0) y blanco (255,255,255) son técnicamente grises, pero primero se evalúa si entran entre los rangos de negros y blancos, por lo que quizás el algoritmo no los detecte como tal (depende de los umbrales definidos).

Gray Threshold: Umbral de gris. Se puede definir como la tolerancia.

Valor Por defecto: $5+5+5= 15$

Ejemplos de color blanco con el 'Black Threshold' = 15 (por defecto)

Ejemplo 1: (0,0,0) 722	Ejemplo 2: (5,5,5) 750
	

Asignación de los umbrales por defecto:

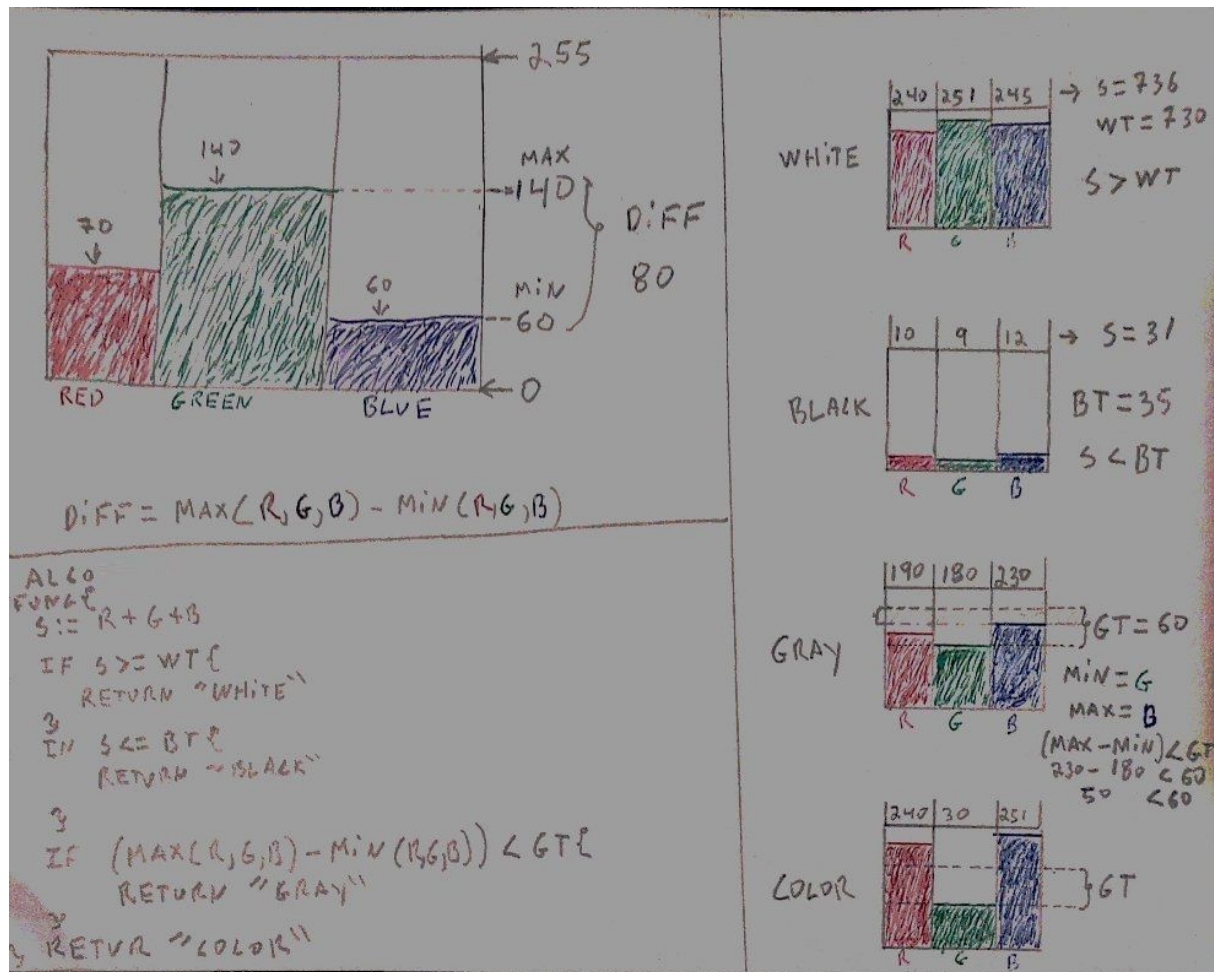
Para cambiar el valor de los umbrales por defecto puedes setear estas variables de entorno en el servicio api_front:

White Threshold: WHITE_THRESHOLD

Black Threshold: BLACK_THRESHOLD

Gray Threshold: GRAY_THRESHOLD

El siguiente dibujo muestra ejemplos de la aplicación del algoritmo en 4 casos posibles.



Binarizado

Para la binarización se utiliza el servicio gRPC "binarizador".

Hace uso de la librería sklearn la cual tiene primitivas nativas. Esta fue la razón de la elección de python sobre c++ para el microservicio de proceso de binarizado.

El servicio usa por defecto la técnica SAUVOLA de binarización, pero se puede configurar para usar los siguientes:

SAUVOLA: Binarizado local (con ventana de valor 25 y $k = 0.8$)

NIBLACK: Binarizado local (con ventana de valor 25 y $k = 0.8$)

OTSU: Binarizado por promedios globales.

4. Probando el servicio

4.1. Compilando y corriendo los servicios

Primero debe clonar o descomprimir el proyecto

Una vez dentro de él debe abrir una terminal y ejecutar:

\$ docker-compose build

Debería ver algo como lo siguiente:

```
---> Using cache
---> 0633b3b6afba
Step 15/16 : ENV PYTHONPATH /opt/${NAME}
---> Using cache
---> 910fb1fae47a
Step 16/16 : CMD ["/usr/bin/python3", "app/server.py"]
---> Using cache
---> 0dcd01c28e85
Successfully built 0dcd01c28e85
Successfully tagged document_color_meter_binarizer:latest
```

Esto puede demorar un tiempo considerable, puesto se construirán las 3 imágenes de los servicios con lo que ello implica (descargar dependencias, compilar, construir la imagen docker, etc)
Este paso sólo debe hacerlo una vez, las veces siguientes usará el caché de docker.

Luego debe ejecutar:

\$ docker-compose up

Con esto se ejecutan los 3 servicios. Debería ver una salida como esta:

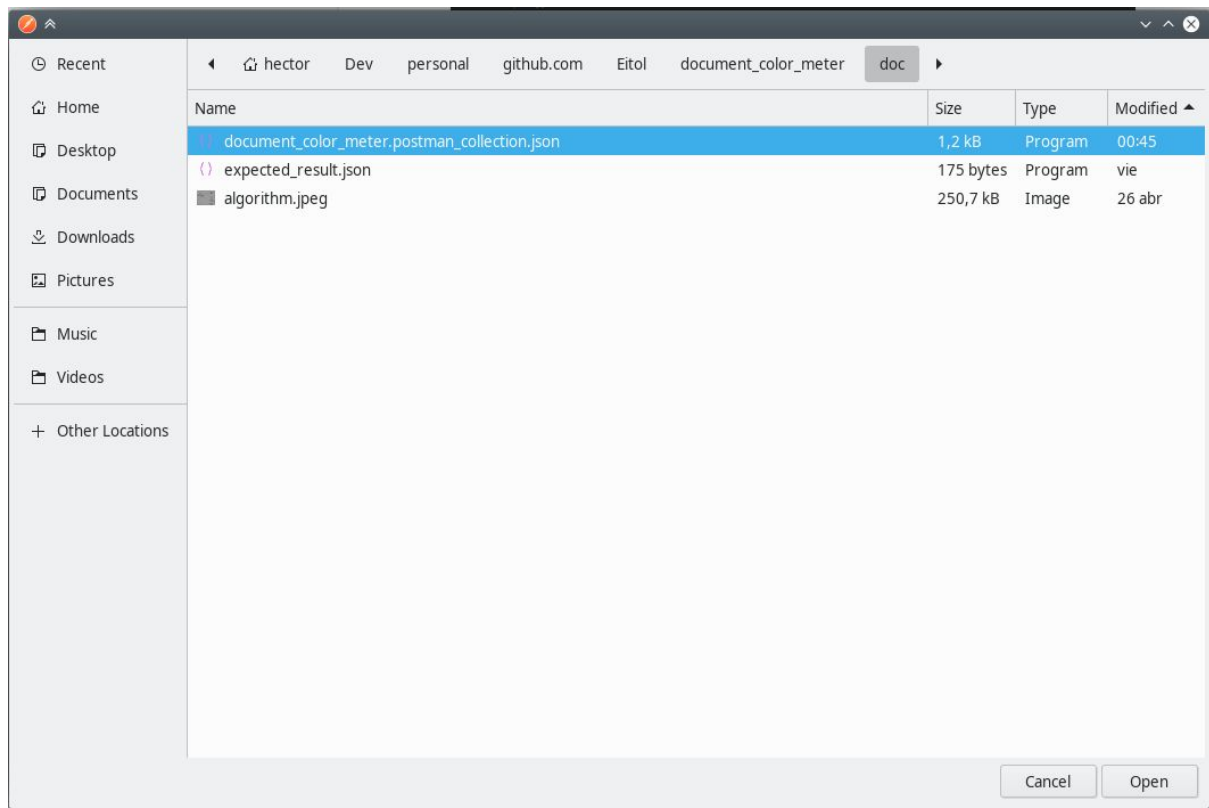
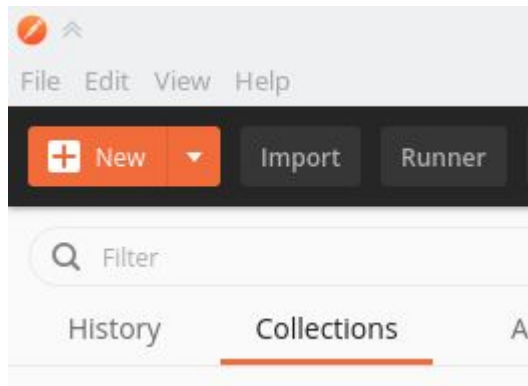
```
hector@hector-VivoBook-ASUSLaptop-X580GD-X580GD:~/Dev/personal/github.com/Eitol/document_color_meter$ docker-compose up
Starting document_color_meter_binarizer_1    ... done
Recreating document_color_meter_api_front_1  ... done
Starting document_color_meter_pdf_to_image_1 ... done
Attaching to document_color_meter_binarizer_1, document_color_meter_pdf_to_image_1, document_color_meter_api_front_1
```

4.2 Probando los servicios con postman

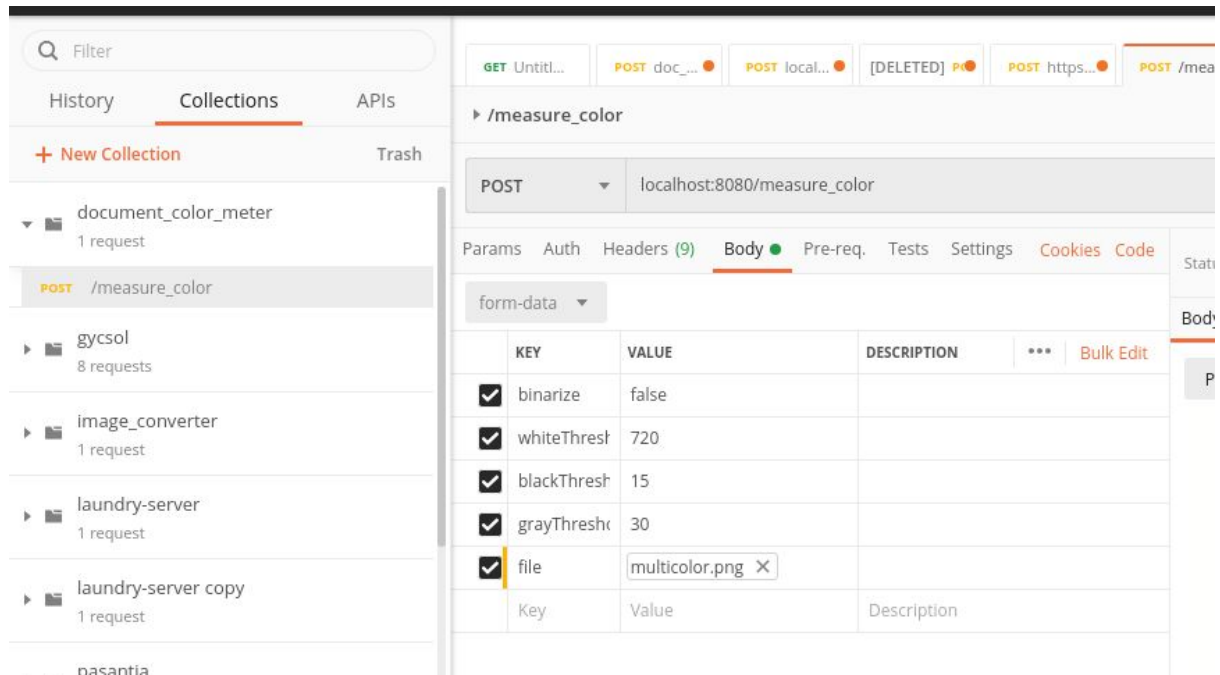
Debe descargar el Postman (<https://www.postman.com/>)

En la carpeta “doc” del proyecto se adjunta una colección de postman que contiene un request ya armado.

Debe ir a postman y dar clic al botón import y seleccionar el fichero “document_color_meter.postman_collection.json”



Luego seleccione el endpoint “/measure_color”



Luego seleccione la imagen “doc/multicolor.png” para el campo “file”

Para finalizar presione el botón “Send” y debería ver la siguiente respuesta:

