2024/2025

Instituto Politécnico de Viana do Castelo

Escola Superior de Tecnologia e Gestão

TRABALHO FINAL – TUTORIAL PARTE 1

INTEGRAÇÃO DE SISTEMAS

JORGE RIBEIRO E LEONARDO MAGALHÃES

ENGENHARIA INFORMÁTICA

Instituto Politécnico de Viana do Castelo

Escola Superior de Tecnologia e Gestão

ENGENHARIA INFORMÁTICA
Enunciado do trabalho final

Integração de Sistemas (de Informação)

Ano letivo 2023/2024

Conteúdo

Requisitos para fazer o tutorial 2
Estrutura de Pastas do Projeto.....3
gRPC Server.....3
gRPC e conexão com a base de dados 9
RestAPI 12

Figura 1 - Arquitetura do sistema 2

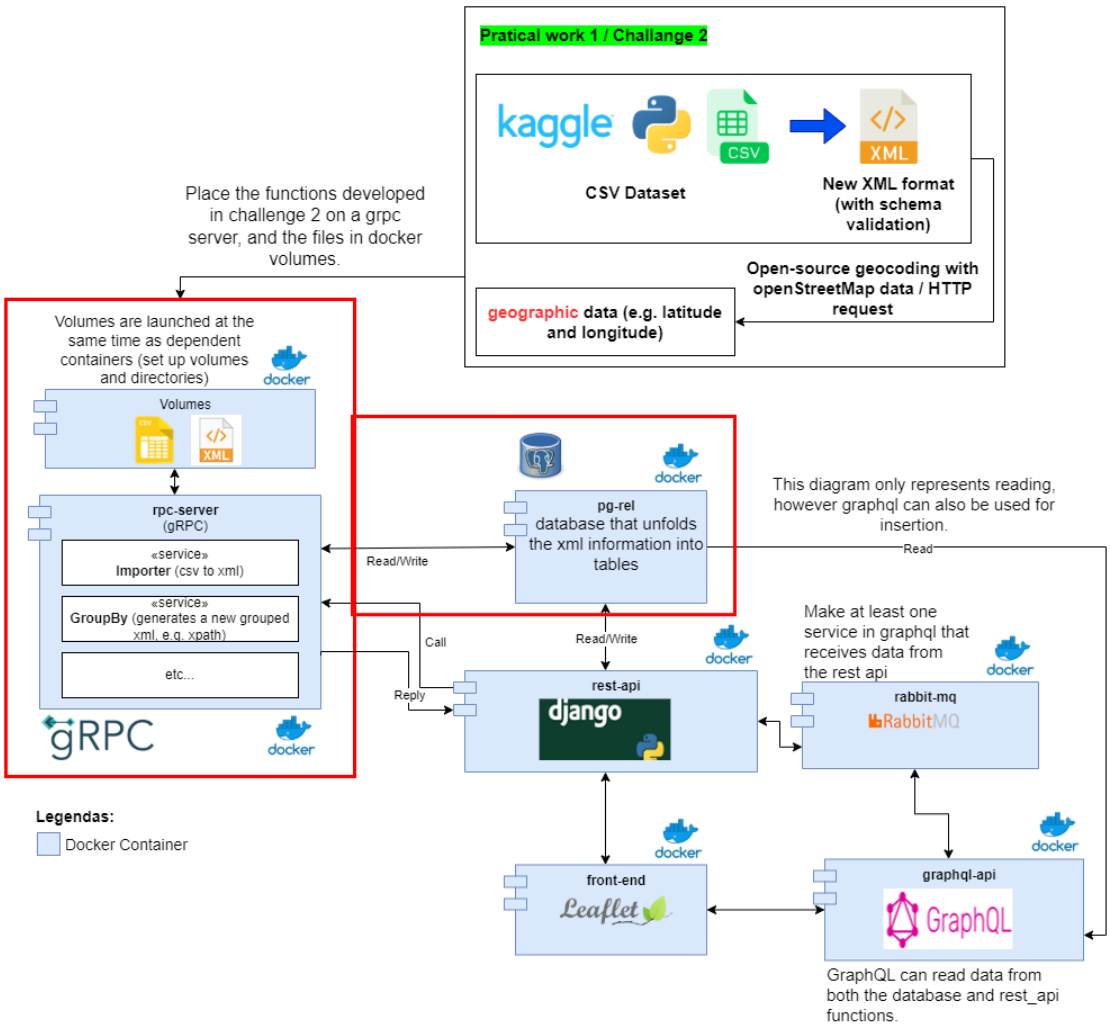


Figura 1 - Arquitetura do sistema

Na primeira parte do tutorial do trabalho prático final, vamos focar nos servidores gRPC e REST API.

Requisitos para fazer o tutorial

- Python devidamente instalado
- Pip
- Docker
- Postaman

Estrutura de Pastas do Projeto

- Is-final
  - grpc-server
  - rest_api_server

gRPC Server

Mais informações acerca de gRPC https://www.velotio.com/engineering-blog/grpc-implementation-using-python

1. Trabalhar no diretório "is-final/ grpc-server"
2. Definir as dependências do projeto, este ficheiro pode ser alterado ao longo do projeto. Criar o ficheiro "requirements.txt"

```
grpcio
grpcio-tools
```

3. Executar o comando "**pip install -r requirements.txt**" para instalar as dependências definidas no ficheiro.

4. Criar o ficheiro .proto (“**server_services.proto**”) (pode ser atualizado ao longo do projeto)

```
syntax = "proto3";
package server_services;
// Request message
message SendFileRequestBody {
  bytes file = 1;       // DTD file as bytes
  string file_mime = 2;
  string file_name = 3;
}
// Response message
message SendFileResponseBody {
  bool success = 1;
}
// Service definition
service SendFileService {
  rpc SendFile (SendFileRequestBody) returns (SendFileResponseBody);
}
```

No código acima foi criado um serviço chamado **SendFileService** que consiste num conjunto de serviços. Neste exercício em questão foi apenas definido o serviço **SendFile.** Este serviço recebe um tipo de mensagem definido (SendFileRequestBody) e retorna uma mensagem do tipo SendFileResponseBody.

5. Após o ficheiro .proto ser devidamente definido, deverá ser executado este comando para gerar as classes necessárias para a implementação destes serviços no gRPC server “**python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. server_services.proto**”. Este comando vai gerar a classe **server_services_pb2_grpc.py** que é responsável pela definição das classes dos serviços gRPC, e vai gerar a classe **server_services_pb2.py** que será responsável por gerar as classes das mensagem dos serviços gRPC.

6. Criar o ficheiro "**settings.py**", que vai ser responsável por ler as variáveis passadas do sistema operativo (ponto importante para quando a aplicação por lançada num container Docker, já que esse container será o sistema operativo). Como na máquina local essas variáveis não estão definidas no sistema operativo, a aplicação vai obter os valores *default* definidos.

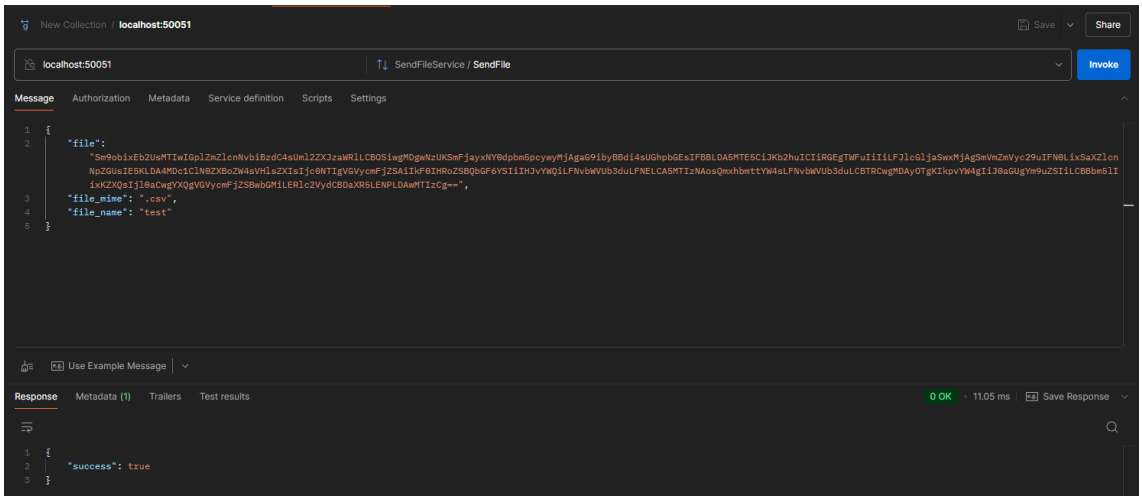
```
import os

# Server Configuration

# Default value '50051'

GRPC_SERVER_PORT = os.getenv('GRPC_SERVER_PORT', '50051')

MAX_WORKERS = int(os.getenv('MAX_WORKERS', '10'))

#Media Files

MEDIA_PATH=os.getenv('MEDIA_PATH', f'{os.getcwd()}/app/csv')
```

7. Criar o ficheiro "main.py", que vai ser responsável por executar o servidor gRPC.

```
from concurrent import futures
from settings import GRPC_SERVER_PORT, MAX_WORKERS, MEDIA_PATH
import os
import server_services_pb2_grpc
import server_services_pb2
import grpc


#Consult the file "server_services_pb2_grpc" to find out the name of the Servicer class of the "SendFileService" service
class SendFileService(server_services_pb2_grpc.SendFileServiceServicer):
  def __init__(self, *args, **kwargs):
    pass


  def SendFile(self, request, context):
    os.makedirs(MEDIA_PATH, exist_ok=True)
    file_path = os.path.join(MEDIA_PATH, request.file_name + request.file_mime)


    ficheiro_em_bytes = request.file


    with open(file_path, 'wb') as f:
      f.write(ficheiro_em_bytes)
    #nome definido no proto para a resposta "SendFileResponseBody"
    return server_services_pb2.SendFileResponseBody(success=True)
```

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    # Consult the file "server_services_pb2_grpc" to see the name of the function generated to add the service to the server
    server_services_pb2_grpc.add_SendFileServiceServicer_to_server(SendFileService(), server)

    server.add_insecure_port(f'[::]:{GRPC_SERVER_PORT}')
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

8. Executar o comando "**python main.py**" para lançar o servidor.
9. Criar um **workspace** no **Postman**
    a. Criar uma Collection "gRPCserver" no workspace criado.
    b. Adicionar um **request** do tipo gRPC na collection criada (ex: sendFileService).
    c. Definir o URL "localhost:50051"
    d. Na aba "selecionar método", carregar o ficheiro .proto criado no projeto e selecionar o serviço que é pretendido testar.
    e. Definir a mensagem enviada para o serviço.
    f. Executar e testar o pedido.



Já que o serviço definido em como objetivo enviar um ficheiro para o servidor, o mesmo é enviado em base64. (para testar o envio de um ficheiro em base64 podem converter o mesmo https://base64.guru/converter/encode/file).

10. Criar o ficheiro "Dockerfile", de forma a executar o servidor num container Docker.

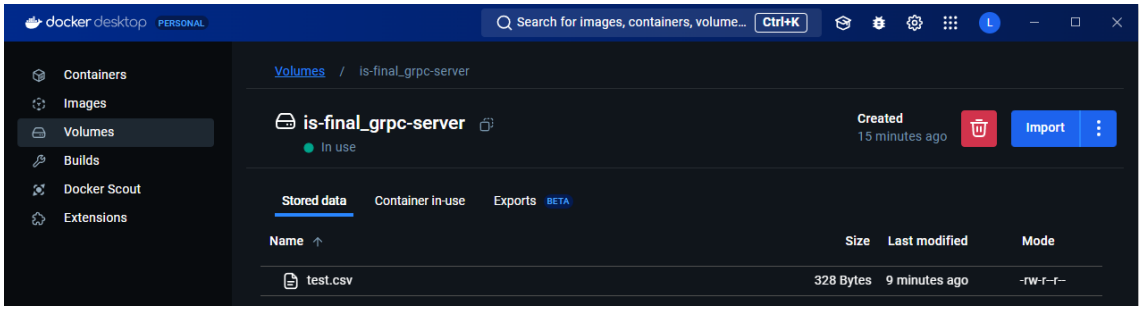
```
# Base image
FROM python:3.10-slim
# Set the working directory
WORKDIR /app
# Copy the requirements file and install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt
# Copy the application code
COPY . .
# Expose the gRPC port
EXPOSE 50051
# Start the gRPC server
CMD ["python", "main.py"]
```

11. Alterar o diretório de trabalho para "is-final" e criar o ficheiro "**docker-compose.yml**"

```
services:
  grpc-server:
    build: ./grpc-server
    container_name: grpc-server
    ports:
      - "50051:50051"
    volumes:
      - grpc-server:/app/media
    # Define the OS/Container variable environments needed by the grpc
server
    environment:
      - GRPC_SERVER_PORT=50051
      - MAX_WORKERS=10
      - MEDIA_PATH=/app/media
      - DBNAME=mydatabase
      - DBUSERNAME=myuser
      - DBPASSWORD=mypassword
      - DBHOST=db
      - DBPORT=5432
    depends_on:
      - db
```

```yaml
  db:
    image: postgres:latest
    container_name: postgres-db
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  grpc-server:
  pgdata:
```

12. Executar o comando "**docker-compose up**"
13. Após o comando ser executado, vão ser levantados os containers para a base de dados e para o gRPC server.
14. Voltar a testar o serviço no **Postman.**
15. Verificar se o ficheiro enviado pelo serviço foi colocado no volume docker definido.



gRPC e conexão com a base de dados

1. Alterar o ficheiro "requirements.txt" e adicionar a dependência "pg8000" para fazer a conexão à base de dados.
2. Instalar a dependência.
3. Ao configurar o ficheiro "**docker-compose.yml**" já foi configurado um container para a base de dados, é de notar já foram definidas as variáveis da base de dados no container "grpc-server".
4. Alterar o ficheiro "settings.py" na pasta "grpc-server" de forma a ler essas variáveis.

5. **Alterar** o valor default da variável **GRPC_SERVER_PORT** para 50052, já que o container deste servidor a partir de agora está a correr na porta 50051. Desta forma é possível executar o servidor na máquina local na porta 50052.

```
import os

# Server Configuration
GRPC_SERVER_PORT = os.getenv('GRPC_SERVER_PORT', '50052')
MAX_WORKERS = int(os.getenv('MAX_WORKERS', '10'))

#Media Files
MEDIA_PATH=os.getenv('MEDIA_PATH', f'{os.getcwd()}/app/csv')

#DB settings
DBNAME=os.getenv('DBNAME', 'mydatabase')
DBUSERNAME=os.getenv('DBUSERNAME', 'myuser')
DBPASSWORD=os.getenv('DBPASSWORD', 'mypassword')
DBHOST=os.getenv('DBHOST', localhost)
DBPORT=os.getenv('DBPORT', '5432')
```

6. Alterar o serviço gRPC no ficheiro “main.py” para fazer a conexão à base de dados.
    a. Importar a dependência “pg8000”;
    b. Importar a dependência “logging” e configurar a mesma, de forma à aplicação enviar logs de erros, etc.
    c. Importar as novas variáveis do ficheiro “settings.py”.
    d. Alterar o serviço, de forma a fazer a conexão à base de dados.

```
from concurrent import futures
from settings import GRPC_SERVER_PORT, MAX_WORKERS, MEDIA_PATH,
DBNAME, DBUSERNAME, DBPASSWORD, DBHOST, DBPORT
import os
import server_services_pb2_grpc
import server_services_pb2
import grpc
import logging
import pg8000

# Configure logging
LOG_FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
logger = logging.getLogger("FileService")

#Consult the file "server_services_pb2_grpc" to find out the name of
the Servicer class of the "SendFileService" service
```

```
class
SendFileService(server_services_pb2_grpc.SendFileServiceServicer):
    def __init__(self, *args, **kwargs):
        pass

    def SendFile(self, request, context):
        os.makedirs(MEDIA_PATH, exist_ok=True)
        file_path = os.path.join(MEDIA_PATH, request.file_name +
request.file_mime)

        ficheiro_em_bytes = request.file

        with open(file_path, 'wb') as f:
            f.write(ficheiro_em_bytes)

        logger.info(f"{DBHOST}:{DBPORT}", exc_info=True)
        # Establish connection to PostgreSQL
        try:
            # Connect to the database
            conn = pg8000.connect(user=f'{DBUSERNAME}',
password=f'{DBPASSWORD}', host=f'{DBHOST}', port=f'{DBPORT}',
database=f'{DBNAME}')
            cursor = conn.cursor()
            # SQL query to create a table
            create_table_query = """
            CREATE TABLE IF NOT EXISTS users (
                id SERIAL PRIMARY KEY,
                name VARCHAR(100),
                email VARCHAR(100) UNIQUE NOT NULL,
                age INT
            );
            """
            # Execute the SQL query to create the table
            cursor.execute(create_table_query)
            # Commit the changes (optional in this case since it's a
DDL query)
            conn.commit()
            #name defined in the proto for the response
"SendFileResponseBody"
            return
server_services_pb2.SendFileResponseBody(success=True)
        except Exception as e:
            logger.error(f"Error: {str(e)}", exc_info=True)
            context.set_details(f"Failed: {str(e)}")
```

```
            context.set_code(grpc.StatusCode.INTERNAL)
            return
server_services_pb2.SendFileResponseBody(success=False)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    # Consult the file "server_services_pb2_grpc" to see the name of
the function generated to add the service to the server
    server_services_pb2_grpc.add_SendFileServiceServicer_to_server(Sen
dFileService(), server)
    server.add_insecure_port(f'[::]:{GRPC_SERVER_PORT}')
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

7. Executar o servidor "**python main.py**" e testar o serviço no **Postman**, de relembrar que agora o servidor local está na porta 50052 e o servidor docker está na porta 50051.
8. Poderá ver a tabela criada (ex: utilizar pgAdmin para consultar as based de dados postgres).
9. Executar o comando "**docker-compose up --build**" e, posteriormente, testar o serviço no postman (porta 50051).

RestAPI

1. Executar o comando "pip install django".
2. Executar o comando "**django-admin startproject rest_api_server**" para criar um projeto django.
3. Dentro da pasta criada definir as dependências do projeto, este ficheiro pode ser alterado ao longo do projeto. Criar o ficheiro "requirements.txt".

```
django
djangorestframework
django-cors-headers
django_pg8000
grpcio
grpcio-tools
```

4. Executar o comando "**pip install -r requirements.txt**" para instalar as dependências definidas no ficheiro.

5. Executar o comando "**python manage.py startapp api**" para criar uma app dentro do projeto.
6. Editar a variável **INSTALLED_APPS** no ficheiro **settings.py** dentro da pasta rest_api_server e adicionar:

```
INSTALLED_APPS = [
    …,
    'rest_framework',
    'api'
]
```

7. Criar uma pasta "serializers" dentro da pasta "api" criada anteriormente, um serializer serve para validar o request body de um determinado endpoint.
8. Criar o ficheiro "__init__.py" (vazio) na pasta "serializers", de forma a poder importar este diretório como módulo.
9. Dentro da pasta "serializers" deve ser criado o ficheiro "file_serializer.py", como apenas vamos fazer um endpoint neste tutorial, apenas vai ter a verificação do request body desse endpoint.

```
from rest_framework import serializers

class FileUploadSerializer(serializers.Serializer):
    file = serializers.FileField()
```

10. Criar uma pasta "views" dentro da pasta "api", onde se vai colocar o código dos endpoints criados na rest-api, como neste tutorial vai ser apenas demonstrado um endoint, apenas vai ser criado um ficheiro. Para API's mais complexas deve-se criar uma estrutura de vários ficheiros por categoria de endpoints.
11. Criar o ficheiro "__init__.py" (vazio) na pasta "views", de forma a poder importar este diretório como módulo.
12. Criar o ficheiro "file_views.py" na pasta "views" e colocar o código do endpoint que queremos criar (ou no futuro endpoints que têm a ver com ficheiros):

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from ..serializers.file_serializer import FileUploadSerializer
import os

class FileUploadView(APIView):
    def post(self, request):
        serializer = FileUploadSerializer(data=request.data)
```

```
        if serializer.is_valid():
            file = serializer.validated_data['file']

            if not file:
                return Response({"error": "No file uploaded"},
status=400)

            # Get MIME type using mimetypes
            file_name, file_extension = os.path.splitext(file.name)

            return Response({
                "file_name": file_name,
                "file_extension": file_extension
            }, status=status.HTTP_201_CREATED)

        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

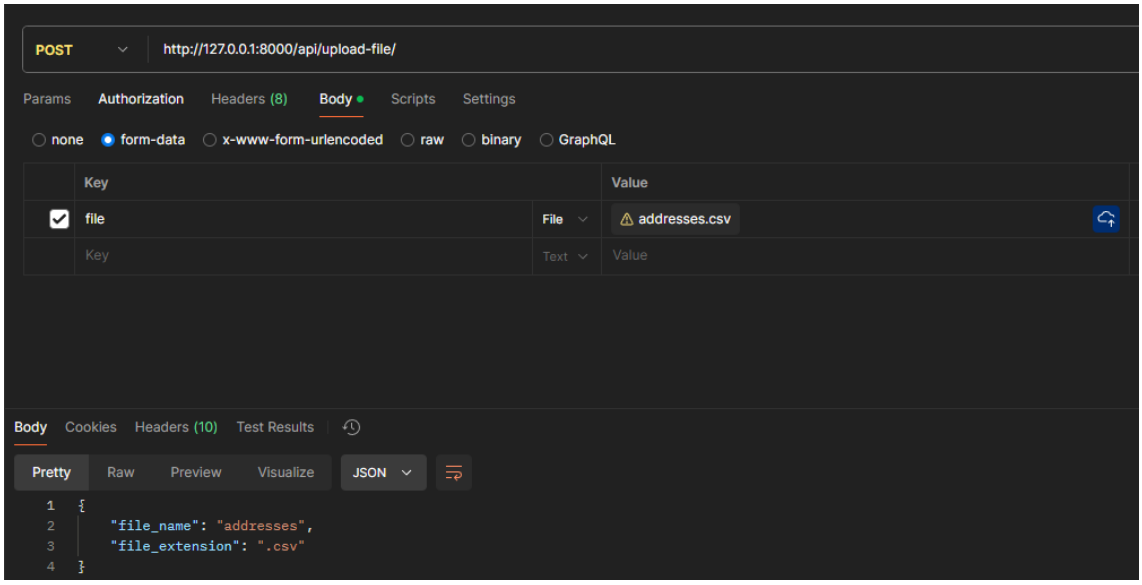
13. Criar o ficheiro "urls.py" na pasta "api". Este ficheiro vai ser responsável por definir as rotas para cada controller definido nos ficheiros presentes na pasta "views".

```
from django.urls import path
from .views.file_views import FileUploadView

urlpatterns = [
    path('upload-file/', FileUploadView.as_view(), name='upload-file')
]
```

14. Dentro da pasta "rest_api_server" alterar o ficheiro "urls.py", de forma a incluir as rotas previamente definidas na raiz do projeto.
15. Executar o comando "**python manage.py runserver**" para correr o servidor.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')), # Include app-level URLs
]
```

16. Criar uma collection no Postman "rest-api", criar um novo REST request "upload-file" e testar o endpoint criado:



17. No ficheiro “rest_api_server/settings.py” importar o modulo “os” e alterar a variável:

```
DATABASES = {
    'default': {
        'ENGINE': 'django_pg8000',
        'NAME': os.getenv('DBNAME', 'mydatabase'),
        'USER': os.getenv('DBUSERNAME', 'myuser'),
        'PASSWORD': os.getenv('DBPASSWORD', 'mypassword'),
        'HOST': os.getenv('DBHOST', 'localhost'),
        'PORT': os.getenv('DBPORT', '5432')
    }
}
```

18. Após configurar a conexão à base de dados está no container docker, criar um ficheiro “users.py” no diretório “api/views”, onde estarão os todos os controllers dos endpoints relacionados com a interação desta tabela.

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from django.db import connection

class GetAllUsers(APIView):
    def get(self, request):
        with connection.cursor() as cursor:
            cursor.execute("SELECT * FROM users")
            result = cursor.fetchall()
        # Convert the result to a list of Book instances if necessary
```

```
        users = [Book(id=row[0], name=row[1]) for row in result]

        return Response({"users": users}, status=status.HTTP_200_OK)
```

19. Adicionar uma rota para este controller no ficheiro “api/urls.py”

```
from django.urls import path
from .views.file_views import FileUploadView
from .views.users import GetAllUsers

urlpatterns = [
    path('upload-file/', FileUploadView.as_view(), name='upload-
file'),
    path('users/', GetAllUsers.as_view(), name='users')
]
```

20. Testar o novo endpoint no Postman.
21. Criar a pasta “grpc” no diretório “api”
22. Criar o ficheiro “__init__.py” (vazio) na pasta “grpc”, de forma a poder importar este diretório como módulo.
23. Copiar o ficheiro .proto definido no rRPC server nessa pasta.
24. Executar o comando “python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. server_services.proto” dentro desse diretório.
25. No ficheiro “server_services_pb2_grpc.py” gerado, alterar o “import server_services_pb2 as server__services__pb2” para “import api.grpc.server_services_pb2 as server__services__pb2”
26. Adicionar no ficheiro “rest_api_server/settings” as variáveis necessárias para a conexão ao servidor gRPC.

```
GRPC_HOST = os.getenv('GRPC_HOST', 'localhost')
GRPC_PORT = os.getenv('GRPC_PORT', '50051')
```

27. Alterar o endpoint de enviar o ficheiro, de forma a enviar esse ficheiro para o rpc server (“api/views/ file_views.py”).

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from ..serializers.file_serializer import FileUploadSerializer
import grpc
import api.grpc.server_services_pb2 as server_services_pb2
import api.grpc.server_services_pb2_grpc as server_services_pb2_grpc
import os
```

```
from rest_api_server.settings import GRPC_PORT, GRPC_HOST

class FileUploadView(APIView):
    def post(self, request):
        serializer = FileUploadSerializer(data=request.data)

        if serializer.is_valid():
            file = serializer.validated_data['file']

            if not file:
                return Response({"error": "No file uploaded"},
status=400)

            # Get MIME type using mimetypes
            file_name, file_extension = os.path.splitext(file.name)

            # Read the file content and convert it to base64
            file_content = file.read()

            # Connect to the gRPC service
            channel = grpc.insecure_channel(f'{GRPC_HOST}:{GRPC_PORT}'
)
            stub    =
server_services_pb2_grpc.SendFileServiceStub(channel)

            # Prepare gRPC request
            request = server_services_pb2.SendFileRequestBody(
                file_name=file_name,
                file_mime=file_extension,
                file=file_content
            )

            # Send file data to gRPC service
            try:
                response = stub.SendFile(request)

                return Response({
                    "file_name": file_name,
                    "file_extension": file_extension
                }, status=status.HTTP_201_CREATED)

            except grpc.RpcError as e:
                return Response({"error": f"gRPC call failed:
{e.details()}"}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

```
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

28. Criar o ficheiro "Dockerfile" na raiz do projeto rest-api.

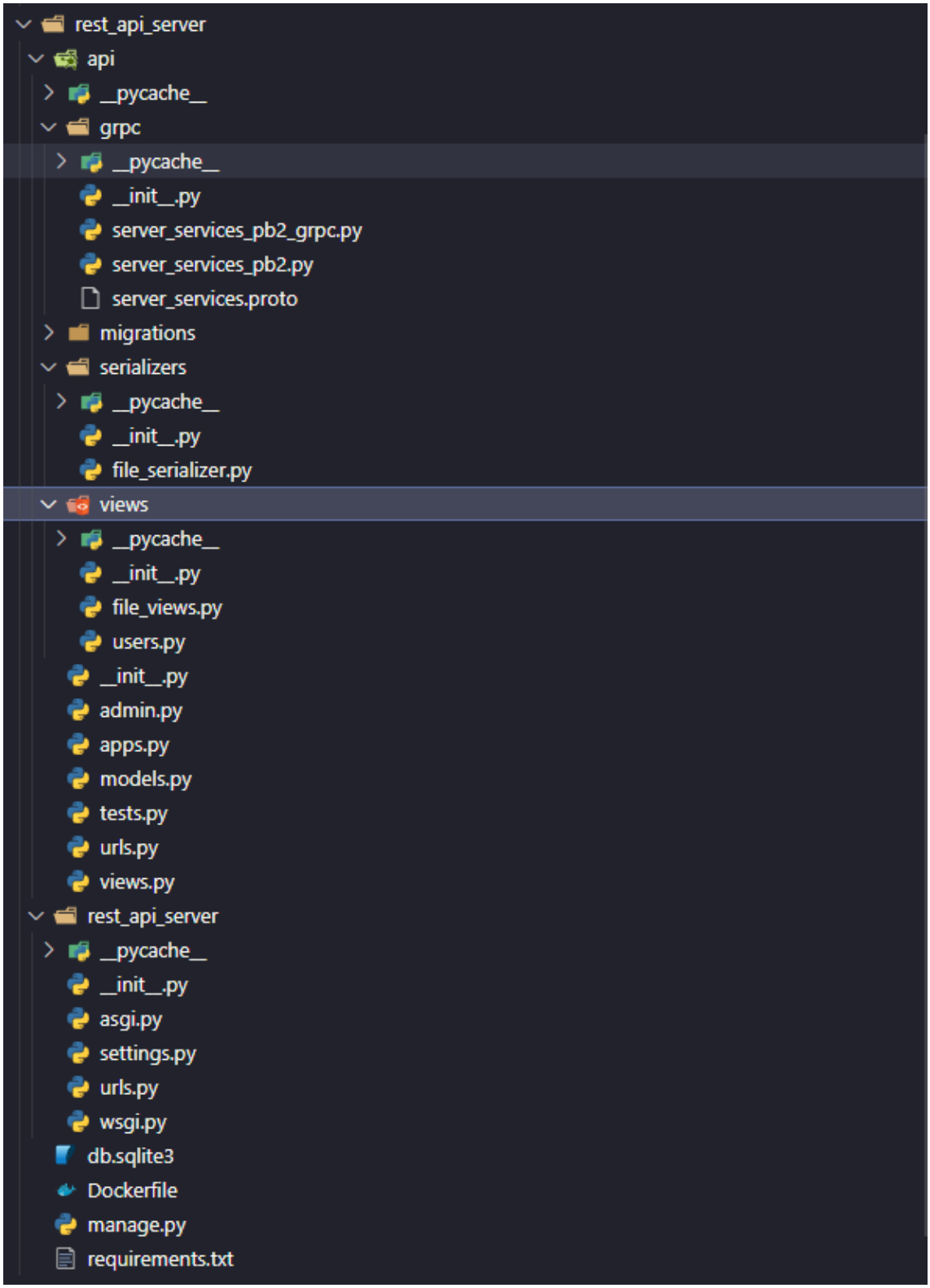
```
# Use an official Python runtime as a parent image
FROM python:3.9-slim
# Set the working directory in the container
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . /app/
# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Expose port 8000 for the Django app
EXPOSE 8000
# Command to run the application
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

29. Alterar o ficheiro "docker-compose.yml" de forma a lançar o container da aplicação

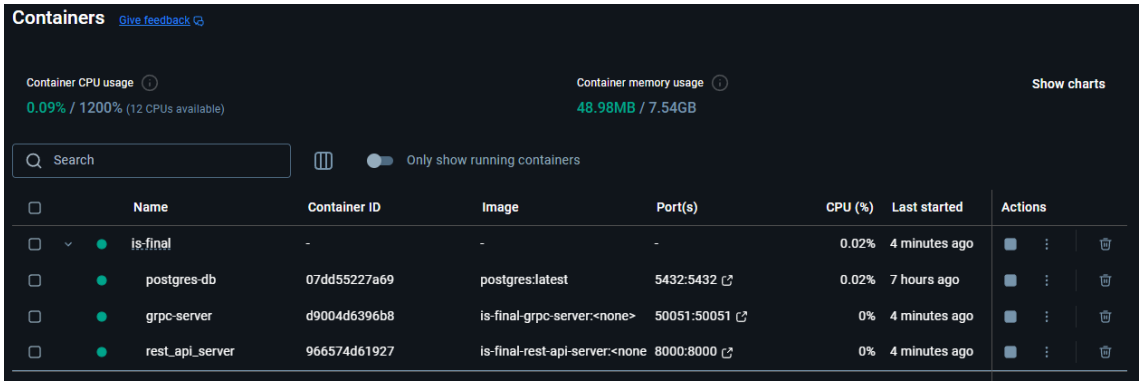
```
services:
  grpc-server:
    build: ./grpc-server
    container_name: grpc-server
    ports:
      - "50051:50051"
    volumes:
      - grpc-server:/app/media
    # Define the OS/Container variable environments needed by the grpc
server
    environment:
      - GRPC_SERVER_PORT=50051
      - MAX_WORKERS=10
      - MEDIA_PATH=/app/media
      - DBNAME=mydatabase
      - DBUSERNAME=myuser
      - DBPASSWORD=mypassword
      - DBHOST=db
      - DBPORT=5432
    depends_on:
      - db
```

```
  rest-api-server:
    build: ./rest_api_server
    container_name: rest_api_server
    ports:
      - "8000:8000"
    # Define the OS/Container variable environments needed by the grpc
server
    environment:
      - GRPC_PORT=50051
      - GRPC_HOST=grpc-server
      - DBNAME=mydatabase
      - DBUSERNAME=myuser
      - DBPASSWORD=mypassword
      - DBHOST=db
      - DBPORT=5432
    depends_on:
      - db
      - grpc-server

  db:
    image: postgres:latest
    container_name: postgres-db
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  grpc-server:
  pgdata:
```

30. A estrutura final do projeto “rest-api” deverá ser:



31. Executar o comando “docker-compose up -d --build”


Containers
Give feedback
Container CPU usage
0.09% / 1200% (12 CPUs available)
Container memory usage
48.98MB / 7.54GB
Show charts
Search
Only show running containers
Name
Container ID
Image
Port(s)
CPU (%)
Last started
Actions
is-final
-
-
-
0.02%
4 minutes ago
postgres-db
07dd55227a69
postgres:latest
5432:5432
0.02%
7 hours ago
grpc-server
d9004d6396b8
is-final-grpc-server:<none>
50051:50051
0%
4 minutes ago
rest_api_server
966574d61927
is-final-rest-api-server:<none
8000:8000
0%
4 minutes ago