



Instituto Politécnico
de Viana do Castelo

Licenciatura em ENGENHARIA INFORMÁTICA / Degree INFORMATICS ENGINEERING

INTEGRAÇÃO DE SISTEMAS

Trabalho Prático Final

Entrega Final

29950 – Diogo Oliveira;

Orientador(es):

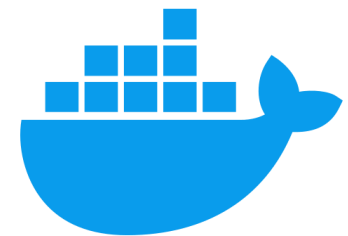
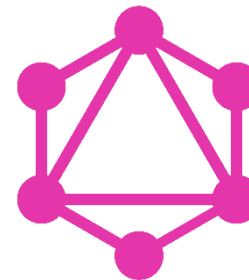
• Professor Jorge Ribeiro, Professor Leonardo Magalhães

■ Índice

- 1. Introdução e Objetivos
- 2. Estrutura do Dataset e Campos
- 3. Containers Docker
- 4. Servidor gRPC
 - 4.1 Dependências do Projeto
 - 4.2 Protótipo de serviços
 - 4.3 Ficheiro main.py
 - 4.4 Testes Postman
- 5. Worker rabbitMQ
 - 5.1 Definição do worker
 - 5.2 Ligação com a BD PostgreSQL
- 6. RestAPI em django
 - 6.1 Views
 - 6.2 URLs
- 7. GraphQL
 - 7.1 Definição de Schema e Models
 - 7.2 Testes Postman
- 8. Frontend
 - 8.1 Página Inicial do Mapa
 - 8.2 Upload do Ficheiro csv
 - 8.3 Filtros XPath
- 9. Bibliografia e Referências Web

■ 1. Introdução e Objetivos

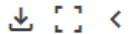
Este trabalho prático, realizado no âmbito da Unidade Curricular de Integração de Sistemas (de Informação e de Tecnologias), tem como objetivo introduzir e desenvolver a habilidade de integrar diversas tecnologias de modo que seja possível a comunicação entre elas. Em concreto, neste trabalho prático, vão ser exploradas as tecnologias de **gRPC**, **PostgreSQL**, **django**, **GraphQL** e **rabbitMQ**. Para a harmonização e facilidade de uso, todos os componentes do trabalho estarão separados em *containers Docker*.



■ 2. Estrutura do Dataset e Campos

Dataset: <https://www.kaggle.com/datasets/sudalairajkumar/daily-temperature-of-major-cities>

city_temperature.csv (140.6 MB)



Detail Compact Column

8 of 8 columns ▾

Region	Country	State	City	# Month	# Day	# Year	# AvgTempe...
--------	---------	-------	------	---------	-------	--------	---------------

Region

Region

North America

54%

Valid ■

2.91m 100%

Mismatched ■

0 0%

Europe

13%

Missing ■

0 0%

Other (967656)

33%

Unique

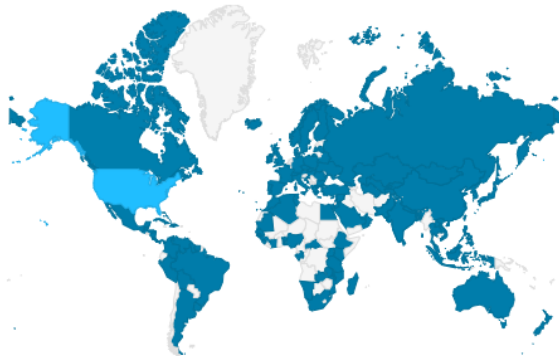
7

Most Common

North Amer... 54%

Country

Country



Valid ■

2.91m 100%

Mismatched ■

0 0%

Missing ■

0 0%

Unique

125

Most Common

US 50%

2. Estrutura do Dataset e Campos

Dataset: <https://www.kaggle.com/datasets/sudalairajkumar/daily-temperature-of-major-cities>

State

State

[null]

Texas

Other (1325626)

City

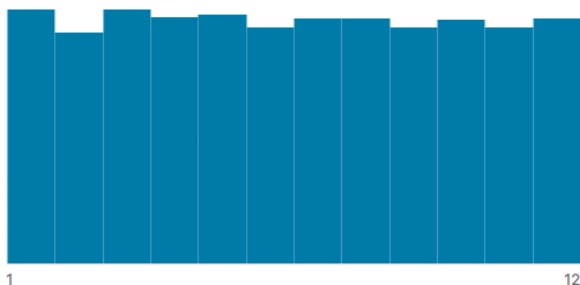
City of observation

321

unique values

Month

Month of observation



50%	Valid	1.46m	50%
	Mismatched	0	0%
4%	Missing	1.45m	50%
46%	Unique	52	
	Most Common	Texas	4%

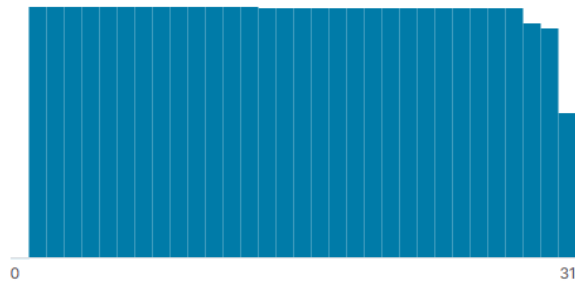
	Valid	2.91m	100%
	Mismatched	0	0%
	Missing	0	0%
	Unique	321	
	Most Common	Washington	1%

	Valid	2.91m	100%
	Mismatched	0	0%
	Missing	0	0%
	Mean	6.47	
	Std. Deviation	3.46	
	Quantiles	1	Min
		12	Max

2. Estrutura do Dataset e Campos

Day

Day of observation



Valid	2.91m	100%
Mismatched	0	0%
Missing	0	0%
Mean	15.7	
Std. Deviation	8.8	
Quantiles	0	Min
	31	Max

Year

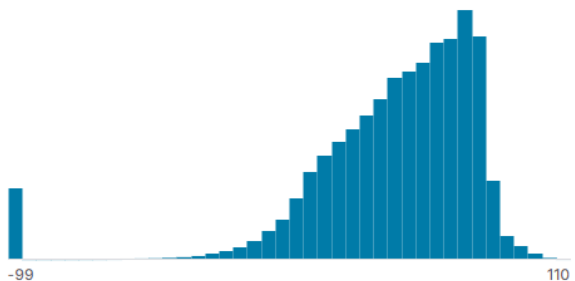
Year of Observation



Valid	2.91m	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.01k	
Std. Deviation	23.4	
Quantiles	200	Min
	2020	Max

AvgTemperature

Average temperature on the given day



Valid	2.91m	100%
Mismatched	0	0%
Missing	0	0%
Mean	56	
Std. Deviation	32.1	
Quantiles	-99	Min
	110	Max

3. Containers Docker

Para esta parte do trabalho prático foram adicionadas 2 novos containers Docker aos 5 existentes. Ficou então associado cada container ao Servidor gRPC, Base de Dados PostgreSQL, REST API django, Servidor rabbitMQ, Worker rabbitMQ, API GraphQL e Frontend em Next.js.

```
grpc-server:
  build: ./grpc-server
  container_name: grpc-server
  ports:
    - "50051:50051"
  volumes:
    - grpc-server:/app/media

  environment:
    GRPC_SERVER_PORT: 50051
    MAX_WORKERS: 10
    MEDIA_PATH: /app/media
    DBNAME: mydatabase
    DBUSERNAME: postgres
    DBPASSWORD: postgres
    DBHOST: db
    DBPORT: 5432
    PYTHONBUFFERED: 1
    RABBITMQ_HOST: rabbitmq
    RABBITMQ_PORT: 5672
    RABBITMQ_USER: guest
    RABBITMQ_PW: guest

  depends_on:
    - db
```

Figura 1 – Container gRPC

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

  volumes:
    grpc-server:
    pgdata:
```

Figura 2 – Container db e volumes

```
rest-api-server:
  build: ./rest_api_server
  container_name: rest_api_server
  ports:
    - "8000:8000"
  environment:
    - GRPC_PORT: 50051
    - GRPC_HOST: grpc-server
    - DBNAME: mydatabase
    - DBUSERNAME: postgres
    - DBPASSWORD: postgres
    - DBHOST: db
    - DBPORT: 5432

  depends_on:
    - grpc-server
    - db
```

Figura 3 – Container RestAPI

■ 3. Containers Docker

```
rabbitmq:
  image: rabbitmq:3.9-management
  ports:
    - "5672:5672"
    - "15672:15672"
  environment:
    RABBITMQ_DEFAULT_USER: guest
    RABBITMQ_DEFAULT_PASS: guest
  restart: always
```

Figura 4 – Container rabbitMQ

```
frontend:
  build: ./frontend
  ports:
    - "3000:3000"
  environment:
    REST_API_BASE_URL: http://rest-api-server:8000
    GRAPHQL_API_BASE_URL: http://graphql-server:8001
    NEXT_PUBLIC_URL: http://frontend:3000
    CSRF_COOKIE: xQjKRQfI8PMAubfyUwElWsp6k7r9BLaE
  depends_on:
    - rest-api-server
    - graphql-server
    - grpc-server
    - rabbitmq
    - db
```

Figura 7 – Container Frontend

```
worker:
  build: ./worker-rabbit-csv
  ports:
    - "8003:8003"
  environment:
    RABBITMQ_HOST: rabbitmq
    RABBITMQ_PORT: 5672
    RABBITMQ_USER: guest
    RABBITMQ_PW: guest
    DBNAME: mydatabase
    DBUSERNAME: postgres
    DBPASSWORD: postgres
    DBHOST: db
    DBPORT: 5432
  depends_on:
    - rabbitmq
    - db
    - grpc-server
```

Figura 5 – Container worker rabbitMQ

```
graphql-server:
  build: ./graphql_server
  container_name: graphql-server
  ports:
    - "8001:8001"
  environment:
    GRPC_PORT: 50051
    GRPC_HOST: grpc-server
    DBNAME: mydatabase
    DBUSERNAME: postgres
    DBPASSWORD: postgres
    DBHOST: db
    DBPORT: 5432
  depends_on:
    - grpc-server
    - db
```

Figura 6 – Container API GraphQL

3. Containers Docker

A estrutura dos containers e volumes fica assim:

is_tp_final				
frontend-1	c50a87a0b8a6	is_tp_final-frontend	3000:3000	
worker-1	004209bb4799	is_tp_final-worker	8003:8003	
rest_api_server	5972df827f21	is_tp_final-rest-api-serve	8000:8000	
graphql-server	2205af571e2e	is_tp_final-graphql-serve	8001:8001	
grpc-server	dfa7f223958d	is_tp_final-grpc-server	50051:50051	
rabbitmq-1	03adf2b8b179	rabbitmq:3.9-manageme	15672:15672	Show all ports (2)
postgres-db	109d2377a44e	postgres:latest	5432:5432	

Figura 8 – Estrutura de containers

Name ↑	Created
0d7eae08961df0894d8e153e75ce287095c76bdd659b4e6c2f554b649faa97d1	3 minutes ago
is_tp_final_grpc-server	2 days ago
is_tp_final_pgdata	2 days ago

Figura 9 – Estrutura volumes

■ 3. Containers Docker

Estrutura do projeto:

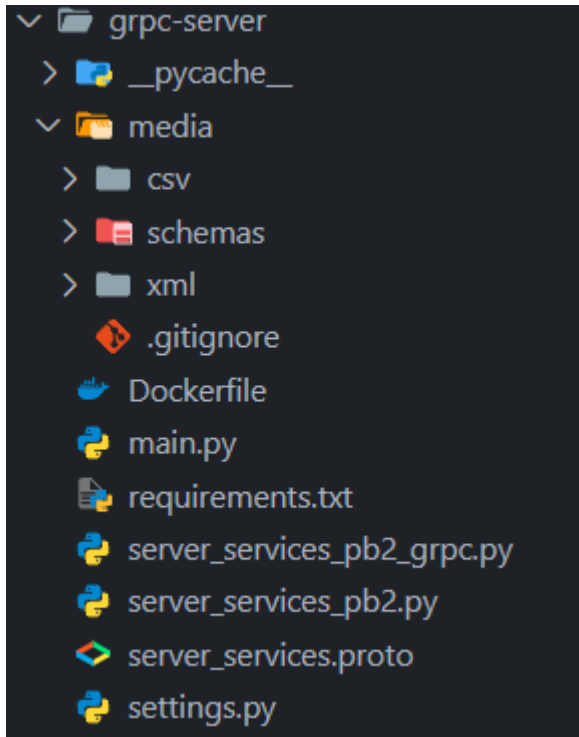


Figura 10 – gRPC-server

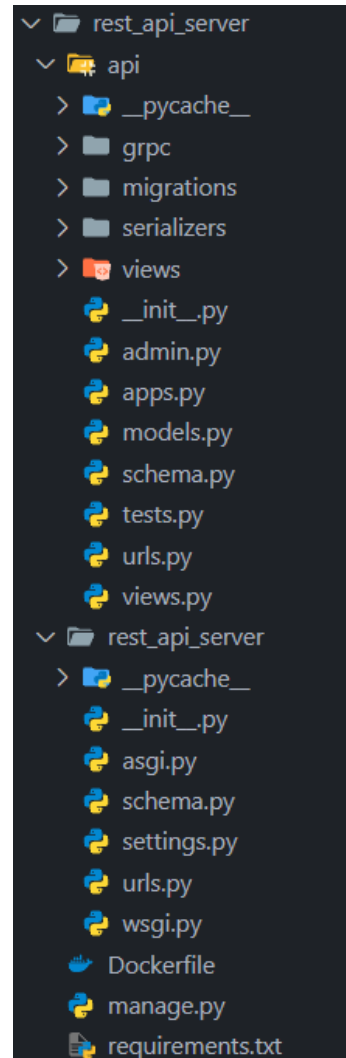


Figura 11 – rest_api_server

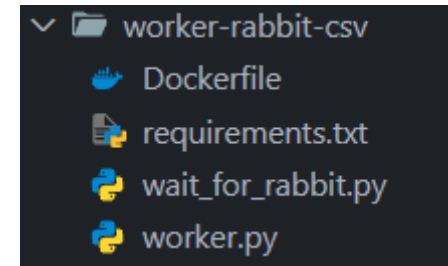


Figura 12 – worker-rabbit-csv

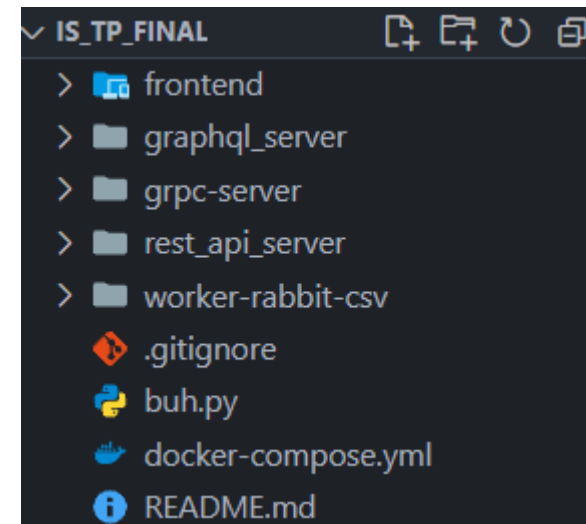
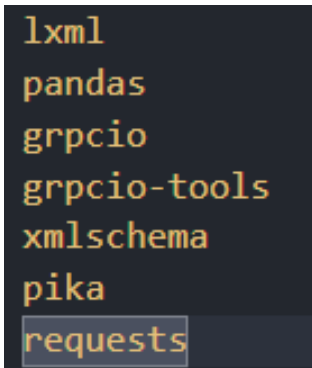


Figura 13 – projeto

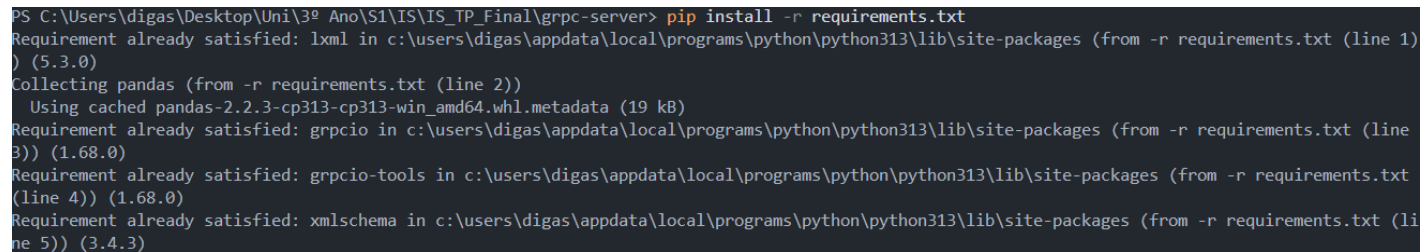
■ 4.1 Servidor gRPC – Dependências do Projeto

Para desenvolver um servidor gRPC em **Python3** é necessário instalar os pacotes **grpcio** e **grpcio-tools**. Para definir as dependências de um projeto em **Python** e instalá-las de maneira rápida, é possível listar os pacotes necessários num ficheiro **requirements.txt** e instalar os pacotes com o seguinte comando: “**pip install –r requirements.txt**”



```
lxml
pandas
grpcio
grpcio-tools
xmlschema
pika
requests
```

Figura 14 – requirements.txt



```
PS C:\Users\digas\Desktop\Uni\3º Ano\S1\IS\IS_TP_Final\grpc-server> pip install -r requirements.txt
Requirement already satisfied: lxml in c:\users\digas\appdata\local\programs\python\python313\lib\site-packages (from -r requirements.txt (line 1)) (5.3.0)
Collecting pandas (from -r requirements.txt (line 2))
  Using cached pandas-2.2.3-cp313-cp313-win_amd64.whl.metadata (19 kB)
Requirement already satisfied: grpcio in c:\users\digas\appdata\local\programs\python\python313\lib\site-packages (from -r requirements.txt (line 3)) (1.68.0)
Requirement already satisfied: grpcio-tools in c:\users\digas\appdata\local\programs\python\python313\lib\site-packages (from -r requirements.txt (line 4)) (1.68.0)
Requirement already satisfied: xmlschema in c:\users\digas\appdata\local\programs\python\python313\lib\site-packages (from -r requirements.txt (line 5)) (3.4.3)
```

Figura 15 – Execução do comando

■ 4.2 Servidor gRPC – Protótipo de serviços

Os serviços de um servidor gRPC podem ser definidos num ficheiro .proto onde são definidos os serviços, requests e responses. Neste ficheiro temos os serviços ImporterService, que trata do upload do ficheiro CSV e conversão para XML, e GroupByService que trata da consulta de dados sobre o XML com queries XPATH.

```
service ImporterService {  
    rpc UploadCSV (FileUploadRequest) returns (FileUploadResponse);  
    rpc UploadCSVChunks (stream FileUploadChunksRequest) returns (FileUploadChunksResponse);  
}
```

Figura 16 – Definição serviço ImporterService

```
service GroupByService {  
    rpc FilterXML (FilterRequest) returns (FilterResponse);  
    rpc SearchXML (SearchRequest) returns (SearchResponse);  
    rpc GroupXML (GroupRequest) returns (GroupResponse);  
    rpc OrderXML (OrderRequest) returns (OrderResponse);  
}
```

Figura 17 – Definição serviço GroupByService

■ 4.2 Servidor gRPC – Protótipo de serviços

No serviço *ImporterService*, existem chamadas *rpc* para **UploadCSV** e **UploadCSVChunks**. A chamada **UploadCSV** realiza a importação de um ficheiro CSV enviado integralmente, ou seja, caso o ficheiro seja grande, o método torna-se lento e ineficiente, pois a execução só prossegue e a resposta só é enviada ao cliente após a importação completa do ficheiro. Já na chamada **UploadCSVChunks**, o ficheiro é dividido em partes(chunks) e enviado a um *worker* (que será detalhado posteriormente) para que este realize as operações necessárias no CSV. Este método é mais eficiente, pois permite que o cliente receba uma resposta sem precisar esperar pela conclusão das operações sobre o ficheiro.

```
message FileUploadRequest {  
    bytes file = 1; // DTD file as bytes  
    string file_mime = 2;  
    string file_name = 3;  
}  
  
message FileUploadResponse {  
    bool success = 1;  
}
```

Figura 18 – Estrutura do pedido e resposta do UploadCSV

```
message FileUploadChunksRequest {  
    bytes data = 1;  
    string file_name = 2;  
}  
  
message FileUploadChunksResponse {  
    bool success = 1;  
    string message = 2;  
}
```

Figura 19 – Estrutura do pedido e resposta do UploadCSVChunks

■ 4.2 Servidor gRPC – Protótipo de serviços

No serviço *GroupByService*, estão definidas as chamadas *rpc* para consultar o XML gerado pelo serviço anterior. Este serviço inclui as seguintes chamadas:

- **FilterXML**: permite realizar uma pesquisa básica com uma *query* definida pelo cliente.
- **SearchXML**: busca qualquer ocorrência do valor especificado no pedido.
- **GroupXML**: agrupa os dados do XML com base nos atributos indicados no pedido.
- **OrderXML**: ordena os elementos do XML consultado de acordo com o atributo especificado no pedido.

```
message FilterRequest {
    string file_name = 1;
    string xpath_query = 2;
}

message FilterResponse {
    string query_result = 1;
}
```

Figura 20 – FilterXML

```
message SearchRequest {
    string file_name = 1;
    string search_term = 2;
}

message SearchResponse {
    repeated string matching_nodes = 1;
}
```

Figura 21 – SearchXML

```
message GroupRequest {
    string file_name = 1;
    repeated string group_by_xpaths = 2;
}

message GroupResponse {
    map<string, int32> grouped_data = 1;
}
```

Figura 22 – GroupXML

```
message OrderRequest {
    string file_name = 1;
    string order_by_xpath = 2;
    bool ascending = 3;
}

message OrderResponse {
    repeated string ordered_nodes = 1;
}
```

Figura 23 – OrderXML

■ 4.3 Servidor gRPC – Ficheiro main.py

No ficheiro **main.py** é onde está o inicializador do servidor e a definição da funcionalidade dos serviços.

```
# ===== Functions =====  
> def csv_to_xml(csv_file, xml_file, objname):...  
> def fill_empty_fields(xml_file):...  
> def validate_xml(xml_file, xsd_file):...  
  
cache = {}  
  
> def get_lat_lon_from_city(city, country):...  
  
# ===== gRPC Services =====  
  
You, 2 days ago | 1 author (You)  
> class ImporterService(server_services_pb2_grpc.ImporterServiceServicer):...  
  
You, 19 hours ago | 1 author (You)  
> class GroupByService(server_services_pb2_grpc.GroupByServiceServicer):...  
  
# ===== main =====  
  
> def serve():...  
  
> if __name__ == '__main__':...
```

Figura 24 – main.py

4.3 Servidor gRPC – Ficheiro main.py

```
def UploadCSVChunks(self, request_iterator, context):
    try:
        # rabbitmq connection
        rabbit_connection = pika.BlockingConnection(
            pika.ConnectionParameters(
                host=RABBITMQ_HOST,
                port=RABBITMQ_PORT,
                credentials=pika.PlainCredentials(RABBITMQ_USER, RABBITMQ_PW),
                heartbeat=600
            )
        )
        rabbit_channel = rabbit_connection.channel()
        rabbit_channel.queue_declare(queue='csv_chunks')

        file_name = None
        file_chunks = []

        # reads the chunks
        for chunk in request_iterator:
            if not file_name:
                file_name = chunk.file_name

            # appends recieved chunk to the list
            file_chunks.append(chunk.data)

            # sends the chunk to the queue
            rabbit_channel.basic_publish(exchange='', routing_key='csv_chunks', body=chunk.data)

        # marks the end of the file
        rabbit_channel.basic_publish(exchange='', routing_key='csv_chunks', body="__EOF__")

        file_content = b"".join(file_chunks)

        file_path = os.path.join(MEDIA_PATH, "./csv/", file_name)

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

    # XML conversion and validation
    # =====
    xml_path = os.path.join(MEDIA_PATH, "./xml/")
    csv_to_xml(file_path, os.path.join(xml_path, file_name.split(".")[0] + ".xml"), "Temp")
    xml_file = os.path.join(xml_path, file_name.split(".")[0] + ".xml")

    fill_empty_fields(xml_file)

    if not validate_xml(xml_file, os.path.join(MEDIA_PATH, "./schemas/", "schema.xsd")):
        logger.error(f"XML Validation Failed: {xml_file}")
    else:
        logger.info(f"XML Validation Success: {xml_file}")
    # =====

    return server_services_pb2.FileUploadChunksResponse(
        success=True,
        message=f'File {file_name} was imported.'
    )
except Exception as e:
    logger.error(f"Error: {str(e)}", exc_info=True)
    return server_services_pb2.FileUploadChunksResponse(success=False, message=str(e))
```

Figura 25 – UploadCSVChunks

```
# XML conversion and validation
# =====
xml_path = os.path.join(MEDIA_PATH, "./xml/")
csv_to_xml(file_path, os.path.join(xml_path, file_name.split(".")[0] + ".xml"), "Temp")
xml_file = os.path.join(xml_path, file_name.split(".")[0] + ".xml")

fill_empty_fields(xml_file)

if not validate_xml(xml_file, os.path.join(MEDIA_PATH, "./schemas/", "schema.xsd")):
    logger.error(f"XML Validation Failed: {xml_file}")
else:
    logger.info(f"XML Validation Success: {xml_file}")
# =====

return server_services_pb2.FileUploadChunksResponse(
    success=True,
    message=f'File {file_name} was imported.'
)
except Exception as e:
    logger.error(f"Error: {str(e)}", exc_info=True)
    return server_services_pb2.FileUploadChunksResponse(success=False, message=str(e))
```

Figura 26 – UploadCSVChunks

ImporterService – UploadCSVChunks:

esta função processa partes de um ficheiro CSV, envia-as para uma queue rabbitMQ, reconstrói e guarda o CSV para convertê-lo em XML e validá-lo.

4.3 Servidor gRPC – Ficheiro main.py

Helper Functions:

```
def csv_to_xml(csv_file, xml_file, objname):
    try:
        df = pd.read_csv(csv_file)
    except FileNotFoundError:
        raise FileNotFoundError(f"Arquivo {csv_file} não encontrado.")

    # creates empty lat and lon columns
    df['latitude'] = None
    df['longitude'] = None

    # populates lat and lon
    for idx, row in df.iterrows():
        if 'City' in row:
            lat, lon = get_lat_lon_from_city(row['City'], row['Country'])
            df.at[idx, 'latitude'] = lat
            df.at[idx, 'longitude'] = lon

    root = Element('root')

    for _, row in df.iterrows():
        obj_element = SubElement(root, objname)
        for col_name, value in row.items():
            col_element = SubElement(obj_element, col_name)
            col_element.text = str(value) if pd.notna(value) else ''

    tree = etree.ElementTree(root)
    tree.write(xml_file, pretty_print=True, xml_declaration=True, encoding='UTF-8')

def fill_empty_fields(xml_file): ...

def validate_xml(xml_file, xsd_file):
    try:
        schema = xmlschema.XMLSchema(xsd_file)
        return schema.is_valid(xml_file)
    except Exception as e:
        logger.error(f"Could not validate XML: {str(e)}")
        return False
```

Figura 27 – Funções de conversão e validação

```
cache = {}

def get_lat_lon_from_city(city, country):
    try:
        # check cache map
        if city in cache:
            logger.info(f"Cache hit for city: {city}")
            return cache[city]['lat'], cache[city]['lon']

        logger.info(f"Cache miss for city: {city}. Fetching data...")
        url = 'https://nominatim.openstreetmap.org/search'
        params = { ... }
        headers = { ... }

        # to respect rate limit
        time.sleep(1)
        response = requests.get(url, params=params, headers=headers)
        response.raise_for_status()
        data = response.json()

        if data:
            lat, lon = data[0]['lat'], data[0]['lon']
            logger.info(f"Latitude: {lat}, Longitude: {lon}")
            cache[city] = {'lat': lat, 'lon': lon}
            return lat, lon
        else:
            # if data from city fails, try with data from country
            params2 = { ... }
            response = requests.get(url, params=params2, headers=headers)
            response.raise_for_status()
            data = response.json()

            if data:
                lat, lon = data[0]['lat'], data[0]['lon']
                logger.info(f"Latitude: {lat}, Longitude: {lon}")
                cache[city] = {'lat': lat, 'lon': lon}
                return lat, lon
            # if no data from country, return 0
            else:
                return 0, 0

    except Exception as e: ...
```

Figura 28 – API para lat e lon

■ 4.3 Servidor gRPC – Ficheiro main.py

FilterXML: Executa uma query XPATH no ficheiro XML especificado no pedido. Retorna os dados da consulta em string.

```
def FilterXML(self, request, context):
    try:
        # validate request body
        if not request.file_name or not request.xpath_query:
            logger.error("Missing body parameters: file_name and/or xpath_query")
            raise ValueError("Missing body parameters: file_name and/or xpath_query")

        xml_path = os.path.join(MEDIA_PATH, "./xml/")
        xml_file = os.path.join(xml_path, request.file_name)

        # validate file exists
        if not os.path.exists(xml_file):
            logger.error(f"File not found: {xml_file}")
            raise FileNotFoundError(f"File not found: {xml_file}")

        # xpath query
        tree = etree.parse(xml_file)
        root = tree.getroot()
        results = root.xpath(request.xpath_query)

        # convert results to string
        str_results = [
            etree.tostring(result, encoding='unicode', method='xml').strip()
            for result in results
        ]

        logger.info(f"XPath query executed successfully: {len(results)} results")
        return server_services_pb2.FilterResponse(query_result=' '.join(str_results))

    except Exception as e:
        logger.error(f"Error: {str(e)}", exc_info=True)
        return server_services_pb2.FilterResponse(query_result=str(e))
```

Figura 29 – FilterXML

■ 4.3 Servidor gRPC – Ficheiro main.py

SearchXML: Realiza uma busca por um termo específico em um ficheiro XML. Ela valida os parâmetros de entrada, verifica a existência do ficheiro, executa uma consulta XPATH para encontrar elementos que contenham o termo, converte os resultados em texto e retorna os nós correspondentes. Em caso de erro, retorna uma lista vazia.

```
def SearchXML(self, request, context):
    try:
        # validates request body
        if not request.file_name or not request.search_term:
            logger.error("Missing body parameters: file_name and/or search_term")
            raise ValueError("Missing body parameters: file_name and/or search_term")

        xml_path = os.path.join(MEDIA_PATH, "./xml/")
        xml_file = os.path.join(xml_path, request.file_name)

        # validates file exists
        if not os.path.exists(xml_file):
            logger.error(f"File not found: {xml_file}")
            raise FileNotFoundError(f"File not found: {xml_file}")

        tree = etree.parse(xml_file)
        root = tree.getroot()

        # apply search with xpath
        results = root.xpath(f"//*[contains(text(), '{request.search_term}')]")

        # convert results to string
        matching_nodes = [
            etree.tostring(el, encoding='unicode', method='xml').strip()
            for el in results
            # filter out empty elements
            if el.getparent() is not None
        ]

        logger.info(f"Search executed successfully: {len(matching_nodes)} matches")
        return server_services_pb2.SearchResponse(matching_nodes=matching_nodes)

    except Exception as e:
        logger.error(f"Error: {str(e)}", exc_info=True)
        return server_services_pb2.SearchResponse(matching_nodes=[])
```

Figura 30 – SearchXML

■ 4.3 Servidor gRPC – Ficheiro main.py

GroupXML: Agrupa elementos de um ficheiro XML com base em valores extraídos por caminhos XPATH fornecidos. Valida os parâmetros de entrada e a existência do ficheiro, percorre os elementos do XML, calcula as chaves de agrupamento, conta as ocorrências de cada grupo e retorna os dados agrupados.

```
def GroupXML(self, request, context):
    try:
        # validate request body
        if not request.file_name or not request.group_by_xpaths:
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details('file_name and group_by_xpaths are required.')
            return server_services_pb2.GroupResponse()

        xml_path = os.path.join(MEDIA_PATH, "./xml/")
        xml_file = os.path.join(xml_path, request.file_name)

        # validate file exists
        if not os.path.exists(xml_file):
            context.set_code(grpc.StatusCode.NOT_FOUND)
            context.set_details(f'File not found: {request.file_name}')
            return server_services_pb2.GroupResponse()

        tree = etree.parse(xml_file)
        root = tree.getroot()

        grouped_data = {}

        # extract values from xml root
        for element in root:
            # extract values from group_by_xpaths
            key = "-".join(
                element.xpath(xpath)[0].text if element.xpath(xpath) else "N/A"
                for xpath in request.group_by_xpaths
            )
            # increment key counter
            grouped_data[key] = grouped_data.get(key, 0) + 1

        return server_services_pb2.GroupResponse(grouped_data=grouped_data)
    except Exception as e:
        context.set_code(grpc.StatusCode.INTERNAL)
        context.set_details(str(e))
        return server_services_pb2.GroupResponse()
```

Figura 31 – GroupXML

■ 4.3 Servidor gRPC – Ficheiro main.py

OrderXML: Ordena elementos de um ficheiro XML com base em um caminho XPath especificado.

```
def OrderXML(self, request, context):
    try:
        # validates request body
        if not request.file_name or not request.order_by_xpath:
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details('file_name and order_by_xpath are required.')
            return server_services_pb2.OrderResponse()

        xml_path = os.path.join(MEDIA_PATH, "./xml/")
        xml_file = os.path.join(xml_path, request.file_name)

        if not os.path.exists(xml_file):
            context.set_code(grpc.StatusCode.NOT_FOUND)
            context.set_details(f'File not found: {request.file_name}')
            return server_services_pb2.OrderResponse()

        tree = etree.parse(xml_file)
        root = tree.getroot()

        # selects all Temp elements
        elements = root.findall('./Temp')

        # orders the elements based on the order_by_xpath
        elements_sorted = sorted(
            elements,
            key=lambda el: el.find(request.order_by_xpath).text if el.find(request.order_by_xpath) is not None else "",
            reverse=not request.ascending # inverts order if ascending is False
        )

        # converts the ordered elements to string
        ordered_nodes = [
            etree.tostring(el, encoding='unicode', method='xml').strip()
            for el in elements_sorted
        ]

        return server_services_pb2.OrderResponse(ordered_nodes=ordered_nodes)
    except Exception as e:
        context.set_code(grpc.StatusCode.INTERNAL)
        context.set_details(str(e))
        return server_services_pb2.OrderResponse()
```

Figura 32 – OrderXML

■ 4.3 Servidor gRPC – Ficheiro main.py

```
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_ImporterServiceServicer_to_server(ImporterService(), server)  
    server_services_pb2_grpc.add_GroupByServiceServicer_to_server(GroupByService(), server)  
  
    server.add_insecure_port(f'[::]:{GRPC_SERVER_PORT}')    server.start()  
  
    print(f'Server running at {GRPC_SERVER_PORT}...')    server.wait_for_termination()
```

Figura 33 – Starter do servidor gRPC

4.4 Servidor gRPC – Testes Postman

Após iniciar o servidor podemos testar os serviços no Postman. Para verificar se o serviço foi executado com sucesso podemos verificar o status devolvido pelo teste e também podemos verificar se o ficheiro enviado foi criado no caminho de **MEDIA** definido nas variáveis de ambiente.

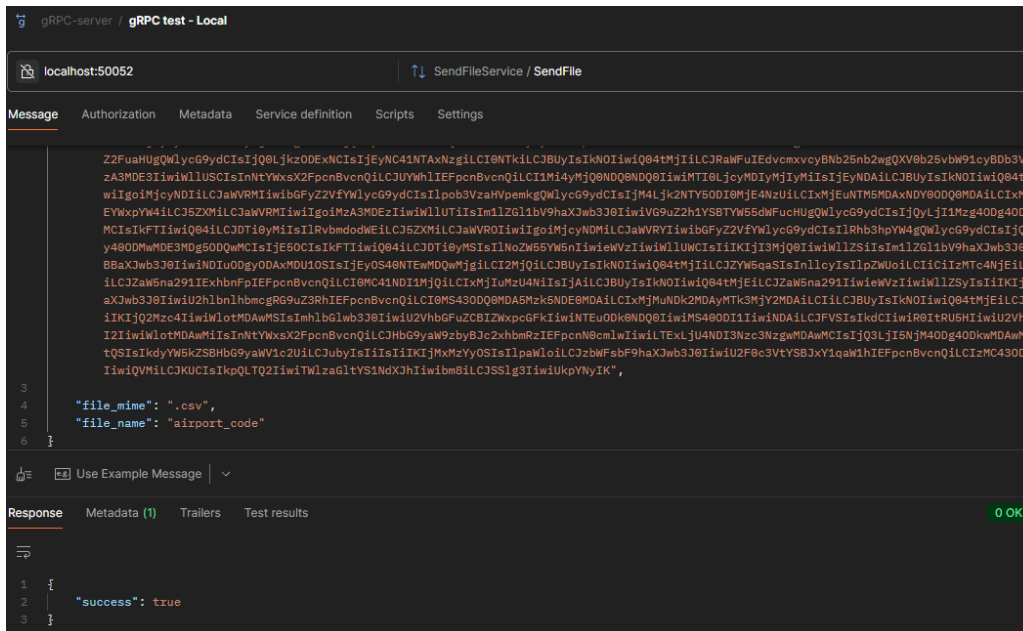


Figura 34 – Teste do serviço no Postman

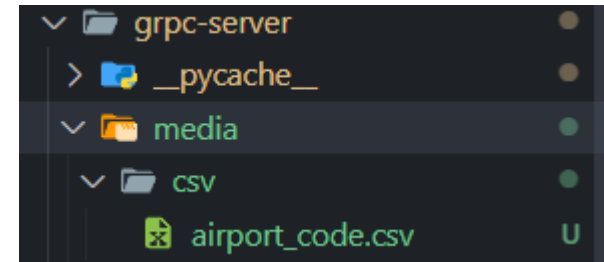


Figura 35 – Ficheiro CSV criado

■ 4.4 Servidor gRPC – Testes Postman

Também é possível testar as chamadas ao serviço de GroupBy.

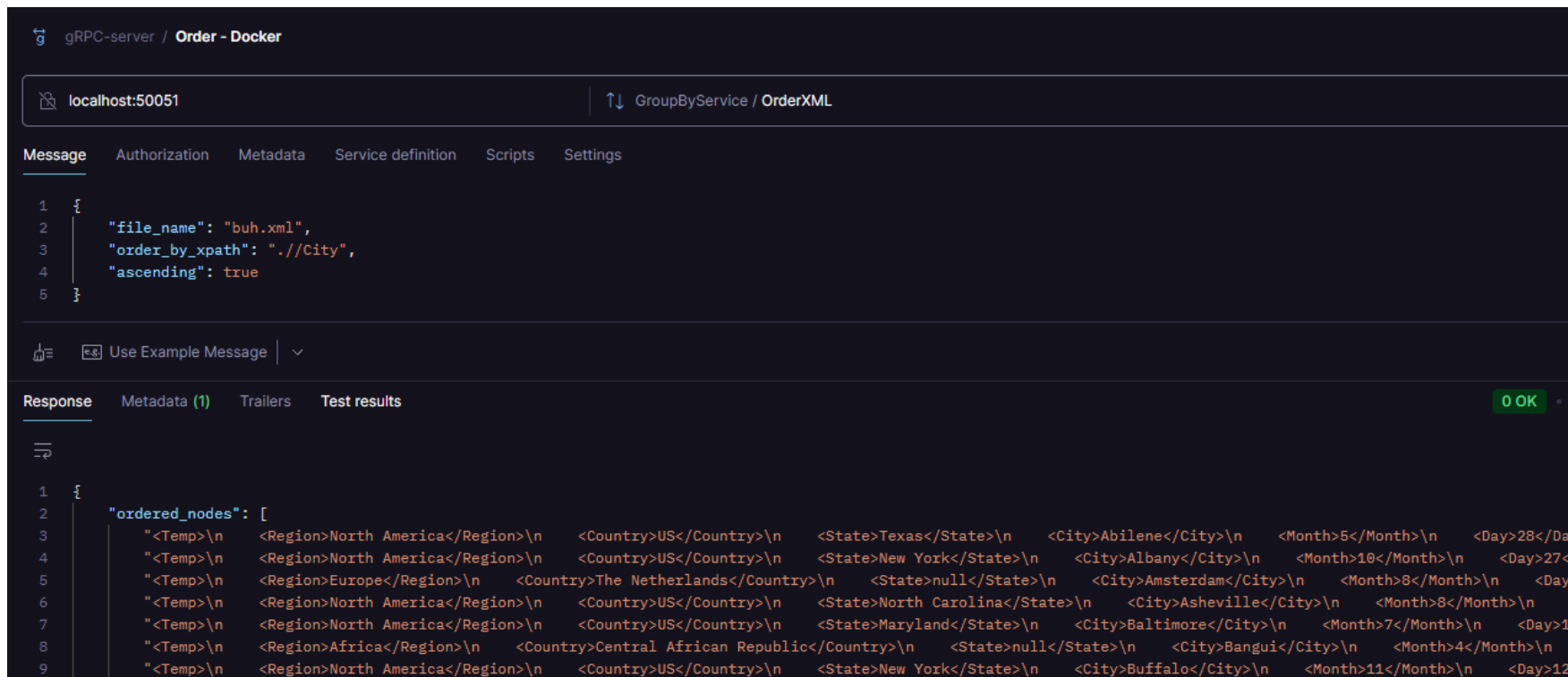


Figura 36 – Teste GroupXML

■ 5.1 Worker rabbitMQ – Definição do Worker

Neste projeto temos um worker rabbitMQ definido para inserir os dados recebidos dos chunks numa base de dados PostgreSQL.

```
def process_message(ch, method, properties, body):
    str_stream = body.decode('utf-8')

    if str_stream == "__EOF__":

        logger.info("EOF marker received. Finalizing...")

        try:
            file_content = b"".join(reassembled_data)
            csvfile = StringIO(file_content.decode('utf-8'))

            for df_chunk in pd.read_csv(csvfile, chunksize=10000):
                logger.info(f"Processing chunk of size: {df_chunk.shape}")
                insert_to_db(df_chunk)

        except Exception as e:
            logger.error(f"Error processing final message: {e}")
        finally:
            reassembled_data.clear()
    else:
        reassembled_data.append(body)

def main():
    credentials = pika.PlainCredentials(RABBITMQ_USER, RABBITMQ_PW)
    connection = pika.BlockingConnection(pika.ConnectionParameters(
        host=RABBITMQ_HOST,
        port=RABBITMQ_PORT,
        credentials=credentials,
        heartbeat=600
    ))

    channel = connection.channel()
    channel.queue_declare(queue=QUEUE_NAME)
    channel.basic_consume(queue=QUEUE_NAME,

        on_message_callback=process_message, auto_ack=True)
    logger.info(f"Waiting for messages...", exc_info=True)
    channel.start_consuming()

if __name__ == "__main__":
    main()
```

Figura 37 – Definição do worker e função process message

■ 5.2 Worker rabbitMQ – Ligação com a BD PostgreSQL

Definição dos modelos das tabelas que serão criadas e preenchidas com os dados do CSV.

```
# DB table models
You, 2 days ago | 1 author (You)
class Country(Base):
    __tablename__ = 'countries'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(255), nullable=False, unique=True)

You, 2 days ago | 1 author (You)
class TemperatureData(Base):
    __tablename__ = 'data'

    id = Column(Integer, primary_key=True, autoincrement=True)
    Region = Column(String(255), nullable=False)
    Country_id = Column(Integer, ForeignKey('countries.id'), nullable=False)
    State = Column(String(255), nullable=True)
    City = Column(String(255), nullable=False)
    Month = Column(Integer, nullable=False)
    Day = Column(Integer, nullable=False)
    Year = Column(Integer, nullable=False)
    AvgTemperature = Column(Float, nullable=False)
    Latitude = Column(Float, nullable=False)
    Longitude = Column(Float, nullable=False)

    # Relationship to the Country table
    country = relationship("Country")

    __table_args__ = (
        UniqueConstraint('Region', 'Country_id', 'State', 'City', 'Month', 'Day', 'Year',
                        name='uq_temperature_data'),
    )
```

Figura 38 – Modelos das tabelas da BD

■ 5.2 Worker rabbitMQ – Ligação com a BD PostgreSQL

Declaração da string de conexão à base de dados e gestor de sessões para controlar as transações feitas.

```
# DB connection
def get_db_connection_string(
    host=DBHOST,
    database=DBNAME,
    user=DBUSERNAME,
    password=DBPASSWORD,
    port=DBPORT
):
    return f"postgresql+pg8000://{user}:{password}@{host}:{port}/{database}"

# DB session
@contextmanager
def db_session_scope(engine):
    Session = sessionmaker(bind=engine)
    session = Session()
    try:
        yield session
        session.commit()
    except Exception as e:
        session.rollback()
        logger.error(f"Error on database transaction: {e}")
        raise
    finally:
        session.close()
```

Figura 39 – Connection string e Session Manager

■ 5.2 Worker rabbitMQ – Ligação com a BD PostgreSQL

Função principal de inserção de dados do ficheiro CSV na base de dados. Usa também a API Nominatim para guardar os valores de latitude e longitude na base de dados.

```
def insert_to_db(df, batch_size=1000):
    try:
        connection_string = get_db_connection_string()
        engine = create_engine(
            connection_string,
            pool_size=10,
            max_overflow=20,
            pool_timeout=30,
            pool_recycle=1800
        )

        # create tables
        Base.metadata.create_all(engine)

        # creates empty lat and lon columns
        df['Latitude'] = None
        df['Longitude'] = None

        # populates lat and lon
        for idx, row in df.iterrows():
            if 'City' in row and 'Country' in row:
                lat, lon = get_lat_lon_from_city(row['City'], row['Country'])
                df.at[idx, 'Latitude'] = lat
                df.at[idx, 'Longitude'] = lon

        data_to_insert = df.to_dict('records')
```

Figura 40 – Cria e preenche as colunas lat e lon do dataframe

■ 5.2 Worker rabbitMQ – Ligação com a BD PostgreSQL

Introduz primeiro os dados na tabela de países para depois mapear os ids para a tabela principal *data*.

```
with db_session_scope(engine) as session:
    # insert countries
    countries = {row['Country'] for row in data_to_insert}
    country_map = {}

    for country in countries:
        stmt = insert(Country).values(name=country).on_conflict_do_nothing()
        session.execute(stmt)

    session.commit()

    # retrieve countries to map later to temps
    country_map = {
        country.name: country.id
        for country in session.query(Country).all()
    }

    # map countries to ids in temp data
    for row in data_to_insert:
        row['Country_id'] = country_map[row.pop('Country')]

    # insert temps
    for i in range(0, len(data_to_insert), batch_size):
        batch = data_to_insert[i:i+batch_size]

        stmt = insert(TemperatureData).values(batch)
        stmt = stmt.on_conflict_do_nothing(
            index_elements=[
                'Region', 'Country_id', 'State', 'City',
                'Month', 'Day', 'Year'
            ]
        )
        session.execute(stmt)

    logger.info(f"Data inserted successfully: {len(data_to_insert)} records")
    return len(data_to_insert)

except Exception as e:
    logger.error(f"Error inserting data: {e}")
    raise
```

Figura 41 – Introdução dos dados na BD

■ 6.1 RestAPI em django – Views

Na pasta views, irão ser criados os ficheiros correspondentes ao endpoints da RestAPI.

Para este projeto foram criadas views para a visualização dos dados *raw* como estão na base de dados. Foi também criada uma view para o upload do ficheiro CSV. E uma view para consultar o XML com queries XPATH.

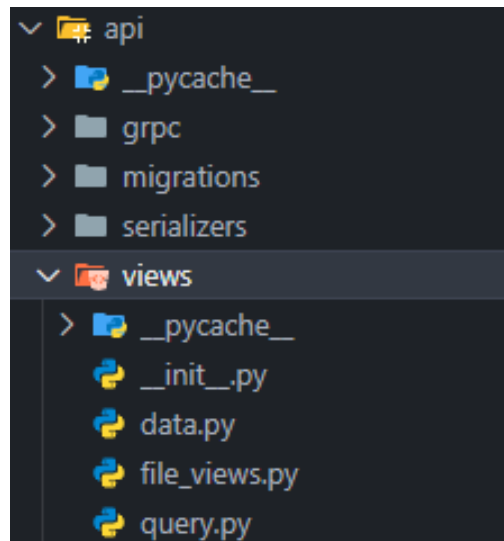


Figura 42 – Ficheiros de views criados

6.1 RestAPI em django – Views

A view data tem apenas um método GET para fazer uma query à base de dados.

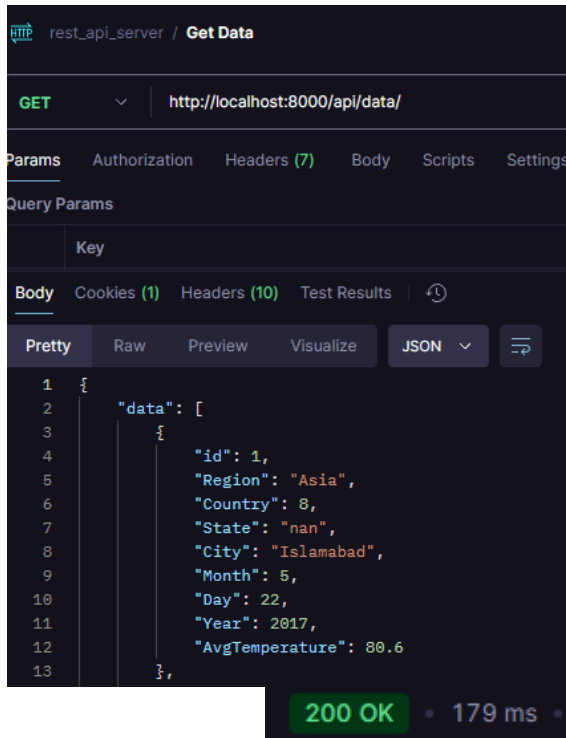


Figura 43 – Resultado do pedido

```

from rest_framework.views import APIView...

You, 2 minutes ago | 1 author (You)
class GetAllData(APIView):
    def get(self, request):
        with connection.cursor() as cursor:
            cursor.execute("SELECT * FROM data")
            result = cursor.fetchall()

            data = [
                {
                    "id": row[0],
                    "Region": row[1],
                    "Country_Id": row[2],
                    "State": row[3],
                    "City": row[4],
                    "Month": row[5],
                    "Day": row[6],
                    "Year": row[7],
                    "AvgTemperature": row[8],
                    "Latitude": row[9],
                    "Longitude": row[10],
                }
                for row in result
            ]

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

Figura 44 – View data.py

6.1 RestAPI em django – Views

A view `file_views` tem os métodos para lidar com o upload dos ficheiros csv. Eles recebem o pedido POST e chamam o servidor gRPC para operar sobre o ficheiro recebido no `request.body`.

```
class FileUploadChunksView(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)

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

            def generate_file_chunks(file, file_name, chunk_size=(64 * 1024)):

                try:
                    while chunk := file.read(chunk_size):
                        yield server_services_pb2.FileUploadChunksRequest(data=chunk, file_name=file_name)

                except Exception as e:
                    print(f"Error reading file: {e}")
                    raise

            try:
                response = stub.UploadCSVChunks(generate_file_chunks(file, file.name, (64 * 1024)))

                if response.success:
                    return Response ({
                        "file_name": file.name,
                    }, status=status.HTTP_201_CREATED)

                return Response ({ "error": f": {response.message}" }, status=status.HTTP_500_INTERNAL_SERVER_ERROR)

            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)
```

Figura 46 – FileUploadChunks da view `file_views.py`

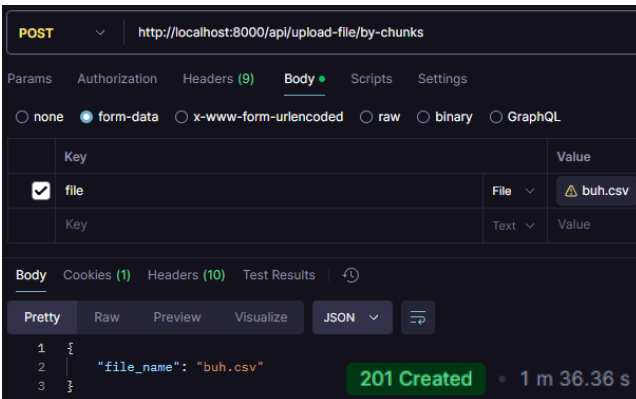


Figura 45 – Resultado do pedido

■ 6.1 RestAPI em django – Views

A view query contem os pedidos POST que vão chamar os serviços gRPC que fazem as consultas ao XML criado.

```
from rest_framework.views import APIView...

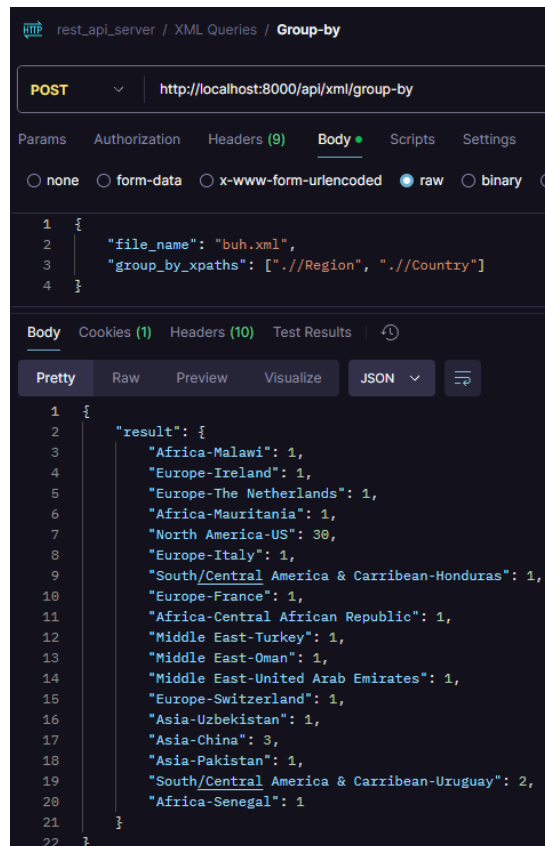
You, 16 seconds ago | 1 author (You)
class XPathFilterBy(APIView):
    def post(self, request):...

You, 16 seconds ago | 1 author (You)
class XPathOrderBy(APIView):
    def post(self, request):...

You, 16 seconds ago | 1 author (You)
class XPathGroupBy(APIView):
    def post(self, request):...

You, 16 seconds ago | 1 author (You)
class XPathSearch(APIView):
    def post(self, request):...
```

Figura 47 –view query.py

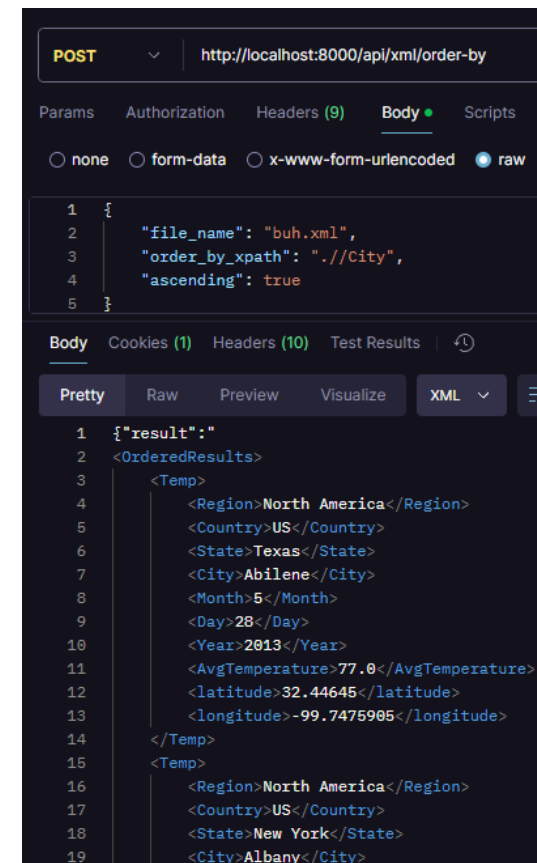


```
POST http://localhost:8000/api/xml/group-by

{
  "file_name": "buh.xml",
  "group_by_xpaths": [".//Region", ".//Country"]
}
```

```
{
  "result": {
    "Africa-Malawi": 1,
    "Europe-Ireland": 1,
    "Europe-The Netherlands": 1,
    "Africa-Mauritania": 1,
    "North America-US": 30,
    "Europe-Italy": 1,
    "South/Central America & Carribean-Honduras": 1,
    "Europe-France": 1,
    "Africa-Central African Republic": 1,
    "Middle East-Turkey": 1,
    "Middle East-Oman": 1,
    "Middle East-United Arab Emirates": 1,
    "Europe-Switzerland": 1,
    "Asia-Uzbekistan": 1,
    "Asia-China": 3,
    "Asia-Pakistan": 1,
    "South/Central America & Carribean-Uruguay": 2,
    "Africa-Senegal": 1
  }
}
```

Figura 48 – Resultado do pedido group-by



```
POST http://localhost:8000/api/xml/order-by

{
  "file_name": "buh.xml",
  "order_by_xpath": ".//City",
  "ascending": true
}
```

```
<?xml version="1.0"?>
<result>
  <OrderedResults>
    <Temp>
      <Region>North America</Region>
      <Country>US</Country>
      <State>Texas</State>
      <City>Abilene</City>
      <Month>5</Month>
      <Day>28</Day>
      <Year>2013</Year>
      <AvgTemperature>77.0</AvgTemperature>
      <latitude>32.44645</latitude>
      <longitude>-99.7475905</longitude>
    </Temp>
    <Temp>
      <Region>North America</Region>
      <Country>US</Country>
      <State>New York</State>
      <City>Albany</City>
    </Temp>
  </OrderedResults>
</result>
```

Figura 49 – Resultado do pedido order-by

■ 6.2 RestAPI em django – URLs

Na pasta api o ficheiro **urls.py** tem a definição dos endpoints da api.

```
from django.urls import path
from .views.file_views import FileUploadView, FileUploadChunksView
from .views.data import GetAllData
from .views.query import XPathFilterBy, XPathGroupBy, XPathSearch, XPathOrderBy

urlpatterns = [
    path('upload-file/', FileUploadView.as_view(), name='upload-file'),
    path('upload-file/by-chunks', FileUploadChunksView.as_view(), name='upload-file-by-chunks'),
    path('data/', GetAllData.as_view(), name='data'),
    path('xml/filter-by', XPathFilterBy.as_view(), name='xml-filter-by'),
    path('xml/group-by', XPathGroupBy.as_view(), name='xml-group-by'),
    path('xml/search-by', XPathSearch.as_view(), name='xml-search-by'),
    path('xml/order-by', XPathOrderBy.as_view(), name='xml-order-by'),
]
```

Figura 50 – api/urls.py

Na pasta do projeto “rest_api_server”, o ficheiro **urls.py** define os endpoints globais da api.

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),
]
```

Figura 51 – rest_api_server/urls.py

■ 7.1 GraphQL – Definição de Schema e Models

Para implementar o GraphQL primeiro é necessário definir os modelos dos dados com que vai interagir. Neste caso ele vai consultar/mutar as Temperaturas e os Países.

```
from django.db import models

...

class Temperature(models.Model):
    Region = models.CharField(max_length=255)
    Country_id = models.ForeignKey(
        'Country',
        on_delete=models.CASCADE,
        db_column='Country_id'
    )
    State = models.CharField(max_length=255)
    City = models.CharField(max_length=255)
    Month = models.IntegerField()
    Day = models.IntegerField()
    Year = models.IntegerField()
    AvgTemperature = models.FloatField()
    Latitude = models.FloatField()
    Longitude = models.FloatField()

    ...

    class Meta:
        db_table = "data"

    def __str__(self):
        return f"{self.Region}, {self.Country_id}, {self.State},

...

class Country(models.Model):
    name = models.CharField(max_length=255)

    ...

    class Meta:
        db_table = "countries"

    def __str__(self):
        return self.name
```

Figura 52 – Modelos definidos

■ 7.1 GraphQL – Definição de Schema e Models

No schema é onde são definidas as operações possíveis de efetuar sobre os dados dos modelos.

Neste caso temos o CRUD para as temperaturas e o Create e Read para os países.

```
import graphene ...

...
class CountryType(DjangoObjectType): ...

...
class TemperatureType(DjangoObjectType): ...

...
class CreateCountry(graphene.Mutation): ...

...
class CreateTemperature(graphene.Mutation): ...

...
class UpdateTemperature(graphene.Mutation): ...

...
class DeleteTemperature(graphene.Mutation): ...

...
class Query(graphene.ObjectType):
    all_temperatures = graphene.List(TemperatureType)
    all_countries = graphene.List(CountryType)
    temperature_by_id = graphene.Field(TemperatureType, id=graphene.Int(required=True))
    temperatures_by_country = graphene.List(TemperatureType, country_id=graphene.Int(required=True))

    def resolve_all_temperatures(self, info): ...

    def resolve_all_countries(self, info): ...

    def resolve_temperature_by_id(self, info, id): ...

    def resolve_temperatures_by_country(self, info, country_id): ...

...
class Mutation(graphene.ObjectType):
    create_temperature = CreateTemperature.Field()
    create_country = CreateCountry.Field()
    update_temperature = UpdateTemperature.Field()
    delete_temperature = DeleteTemperature.Field()

schema = graphene.Schema(query=Query, mutation=Mutation)
```

Figura 53 – Schema definido

7.2 GraphQL – Testes Postman

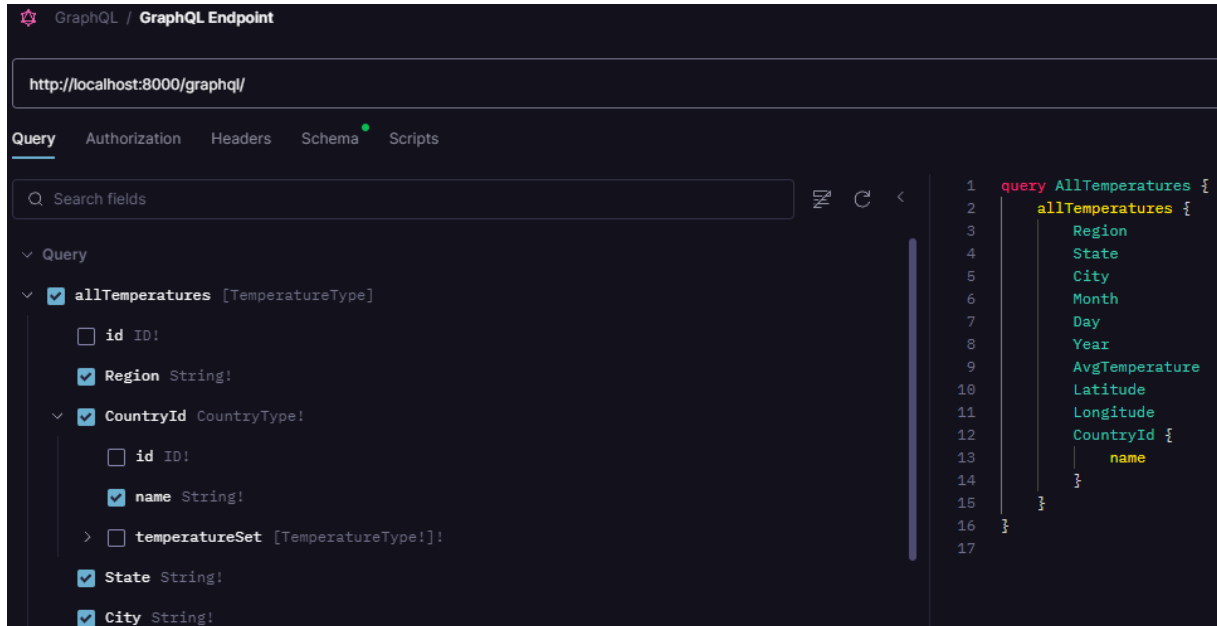


Figura 54 – Query a todas as entradas Temperature

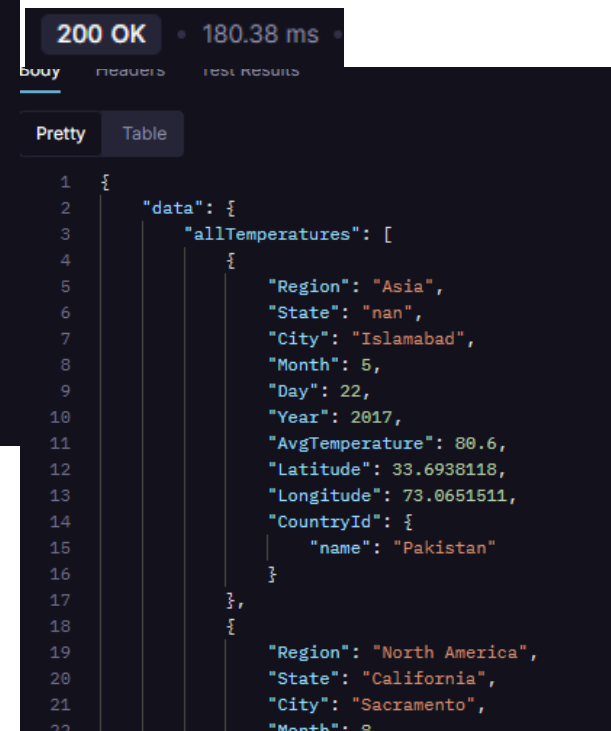


Figura 55 – Resultado

■ 8.1 Frontend – Página Inicial do Mapa

Para carregar os dados guardados na base de dados PostgreSQL através do servidor GraphQL foi necessário criar um tipo City no schema do GraphQL e fazer com que seja possível ler e atualizar objetos desse tipo.

```
class CityType(graphene.ObjectType):
    nome = graphene.String()
    latitude = graphene.Float()
    longitude = graphene.Float()
    id = graphene.ID()
```

Figura 56 – CityType

```
cities = graphene.List(CityType, nome=graphene.String())

def resolve_cities(self, info, nome=None):
    query = Temperature.objects.values('City', 'Latitude', 'Longitude', 'id').distinct('Latitude', 'Longitude')
    if nome:
        query = query.filter(City__icontains=nome)
    return [
        CityType(
            nome=city['City'],
            latitude=city['Latitude'],
            longitude=city['Longitude'],
            id=city['id']
        ) for city in query
    ]
```

Figura 58 – Ler cidades

```
class UpdateCity(graphene.Mutation):
    You, last week | 1 author (You)
    class Arguments:
        id = graphene.Int(required=True)
        nome = graphene.String()
        latitude = graphene.Float()
        longitude = graphene.Float()

    city = graphene.Field(CityType)

    def mutate(self, info, id, nome=None, latitude=None, longitude=None):
        try:
            temperature = Temperature.objects.get(id=id)
            if latitude is not None:
                temperature.Latitude = latitude
            if longitude is not None:
                temperature.Longitude = longitude
            temperature.save()

            return UpdateCity(city=CityType(
                id=temperature.id,
                nome=temperature.City,
                latitude=temperature.Latitude,
                longitude=temperature.Longitude
            ))
        except Temperature.DoesNotExist:
            raise Exception("City not found")
```

Figura 57 – Atualizar cidade

■ 8.1 Frontend – Página Inicial do Mapa

Depois na base de código do frontend em React e Next.js foi necessário alterar o URL da request e verificar se o body do pedido ao servidor estava de acordo com o esperado pelo servidor GraphQL.

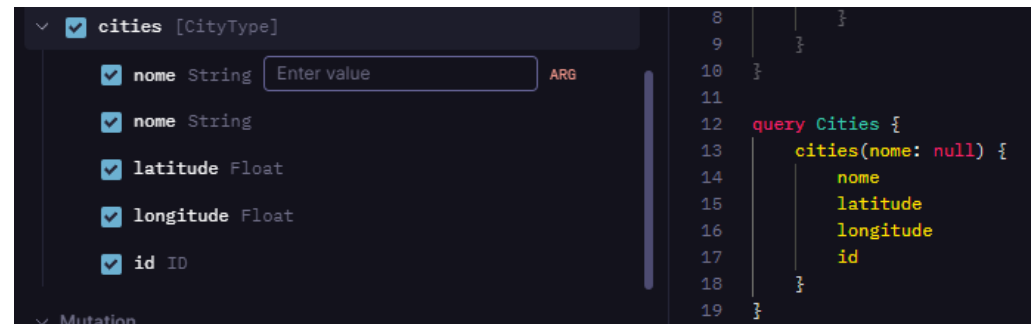
```
export async function POST(req: NextRequest) {

  const requestBody = {
    query: `query Cities {
      cities${city.length > 0 ? `(nome: "${city})"` : ``} {
        nome
        latitude
        longitude
        id
      }
    }`
  }

  const options = { ... }

  try{
    const promise = await fetch(`${process.env.GRAPHQL_API_BASE_URL}/graphql/`, options)
  }
```

Figura 59 – Código route.ts da request



```
8 | }
9 | }
10 | }
11 | }
12 | query Cities {
13 |   cities(nome: null) {
14 |     nome
15 |     latitude
16 |     longitude
17 |     id
18 |   }
19 | }
```

Figura 60 – Estrutura da query GraphQL

■ 8.1 Frontend – Página Inicial do Mapa

Depois na base de código do frontend em React e Next.js foi necessário alterar o URL da request e verificar se o body do pedido ao servidor estava de acordo com o esperado pelo servidor GraphQL.

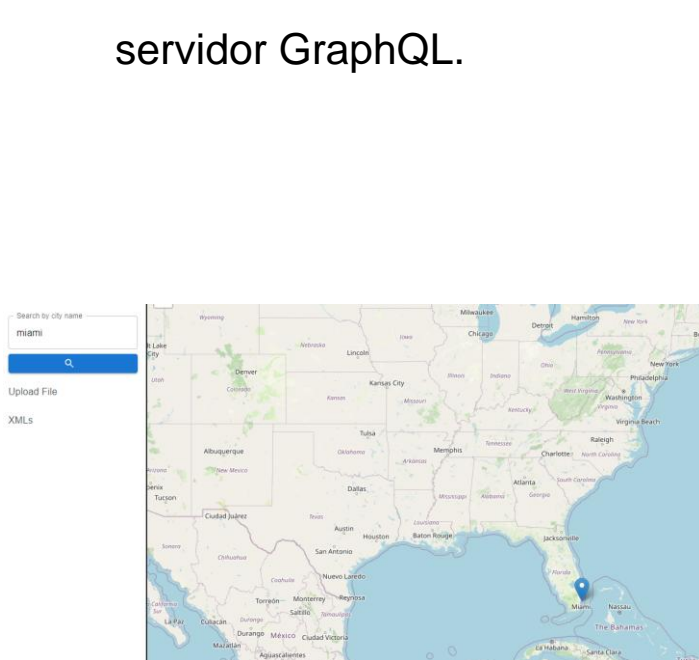


Figura 61 – Página inicial sem pesquisa

Figura 62 – Exemplo pesquisa “Miami”

■ 8.1 Frontend – Página Inicial do Mapa

Para aplicar guardar as
mudanças feitas a alterar um
pin de sítio utiliza-se também
o servidor GraphQL.

```
const requestBody = {
  query: `mutation UpdateCity {
    updateCity(id: ${id}, latitude: ${request_body.latitude}, longitude: ${request_body.longitude}) {
      city {
        nome
        latitude
        longitude
        id
      }
    }
  }`
}

const options = { ...
}

try{
  const promise = await fetch(`${process.env.GRAPHQL_API_BASE_URL}/graphql/`, options)
```

Figura 63 – Request de mutação

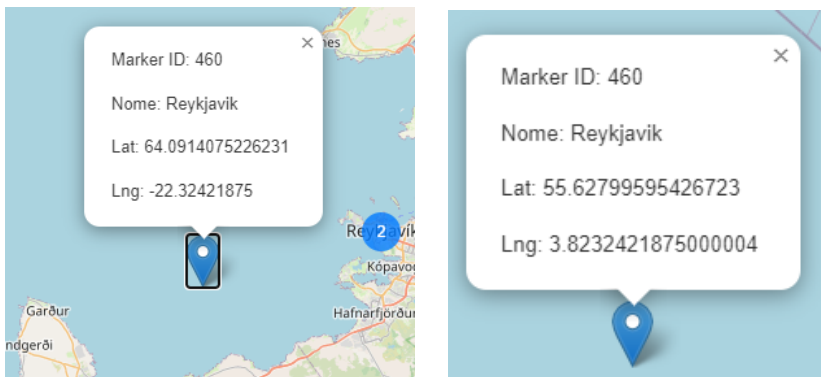


Figura 64 – Exemplo mutação (antes/depois)

■ 8.2 Frontend – Upload do Ficheiro csv

Para dar upload do ficheiro
csv é utilizado o URL criado
na rest_api.

```
export async function POST(req: NextRequest) {
  const formData = await req.formData()
  const body = Object.fromEntries(formData)
  const file = (body.file as Blob) || null
  const dtd_file = (body.dtd_file as Blob) || null

  if(!file || !dtd_file){
    return NextResponse.json({status: 500, message: 'Files not sent!'}, { status: 500 })
  }

  const formdata = new FormData()

  formdata.append("file", file)
  formdata.append("file_dtd", dtd_file)

  const requestOptions = {
    method: "POST",
    body: formdata
  }

  try{
    const promise = await fetch(`${process.env.REST_API_BASE_URL}/api/upload-file/by-chunks`, requestOptions)
```

Figura 65 – Código da request

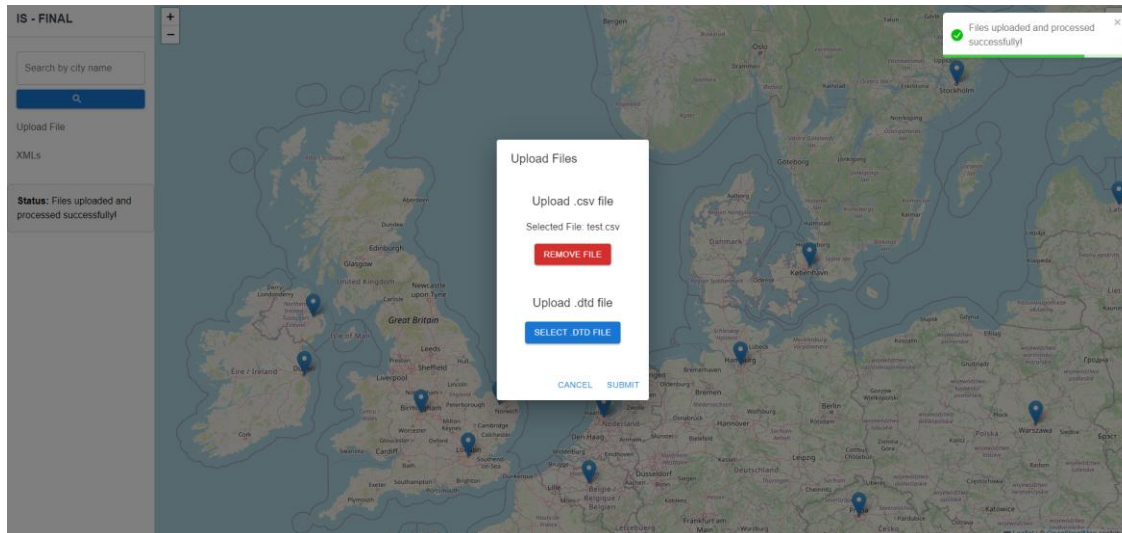


Figura 66 – Exemplo ficheiro uploaded

■ 8.3 Frontend – Filtros XPath

Para navegar o ficheiro XML gerado com os filtros Xpath, são utilizados os endpoints criados na rest_api.

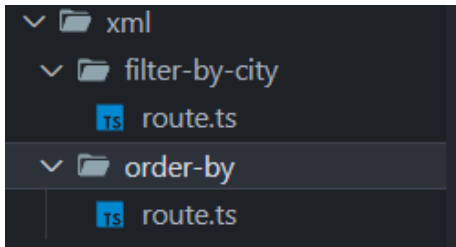


Figura 67 – Estrutura de pedidos

```
const requestOptions = {
  method: "POST",
  body: JSON.stringify({
    "file_name": "tiny.xml",
    "xpath_query": `//Temp[contains(City, '${city}')]`
  }),
  headers: {
    'content-type': 'application/json'
  }
};

try {
  const response = await fetch(`${process.env.REST_API_BASE_URL}/api/xml/filter-by`, requestOptions);
}
```

Figura 68 – Código da request “filter-by”

```
const requestOptions = {
  method: "POST",
  body: JSON.stringify({
    "file_name": "tiny.xml",
    "order_by_xpath": `.${}/${expression}`,
    "ascending": `${ascending}`
  }),
  headers: {
    'content-type': 'application/json'
  }
};

try {
  const response = await fetch(`${process.env.REST_API_BASE_URL}/api/xml/order-by`, requestOptions);
}
```

Figura 69 – Código da request “order-by”

■ 8.2 Frontend – Upload do Ficheiro csv

XML Viewer

SEARCH BY CITY ORDER-BY

Search by city name

New York

Q

```
<root>
  <Temp>
    <Region>North America</Region>
    <Country>US</Country>
    <State>New York</State>
    <City>New York City</City>
    <Month>8</Month>
    <Day>30</Day>
    <Year>2019</Year>
    <AvgTemperature>78.2</AvgTemperature>
    <latitude>40.7127281</latitude>
    <longitude>-74.0060152</longitude>
  </Temp>
</root>
```

CANCEL

Figura 70 – Resultado da query filter-by

XML Viewer

```
<Temp>
  <Region>Middle East</Region>
  <Country>Saudi Arabia</Country>
  <State>null</State>
  <City>Riyadh</City>
  <Month>6</Month>
  <Day>11</Day>
  <Year>2015</Year>
  <AvgTemperature>97.8</AvgTemperature>
  <latitude>18.0107143</latitude>
  <longitude>-15.955326</longitude>
</Temp>
<Temp>
  <Region>North America</Region>
  <Country>US</Country>
  <State>Arizona</State>
  <City>Phoenix</City>
  <Month>9</Month>
  <Day>1</Day>
  <Year>2019</Year>
  <AvgTemperature>97.7</AvgTemperature>
  <latitude>33.4484367</latitude>
  <longitude>-112.074141</longitude>
</Temp>
<Temp>
  <Region>North America</Region>
  <Country>US</Country>
  <State>Arizona</State>
  <City>Phoenix</City>
  <Month>6</Month>
  <Day>11</Day>
```

CANCEL

Figura 71 – Resultado da query order-by por “AvgTemperature”

■ 9 Bibliografia e Referências Web

[1] Dataset Kaggle: <https://www.kaggle.com/datasets/sudalairajkumar/daily-temperature-of-major-cities/data>

[2] Docker Desktop Download: <https://www.docker.com/products/docker-desktop/>

[3] Tutorial de execução do Trabalho Final:

https://elearning.ipvc.pt/ipvc2024/pluginfile.php/74854/mod_resource/content/1/is-final-tutorial-1-PT.pdf

[4] Tutorial de implementação do GraphQL: <https://medium.com/simform-engineering/empowering-your-django-backend-with-graphql-a-powerful-combination-764babd30bb0>

[5] Base de Código/template do frontend: <https://gitlab.com/leo99fernandes/is-frontend-template>

o teu • de partida



www.ipvc.pt