# Projeto Final (Tutorial) - Insights COM241 - Administração e Gerência de Redes de Computadores

**Carlos Henrique Souza Silva[1], Leonardo Rodrigo de Sousa[1], Lucas Tiense Blazzi[1], Marcelo Cavalca Filho[1]**

[1]Universidade Federal de Itajubá (UNIFEI)
Itajubá – MG – Brazil

***Resumo.*** *Este documento descreve o desenvolvimento de uma plataforma para backtest de investimento e proposição de carteiras à clientes por assessores. O documento desenvolve detalhadamente as caracterísitcas de cada serviço desenvolvido no projeto. O objetivo arquitetural do trabalho foi elaborar elaborar os métodos de comunicação utilizando gRPC por meio de protocol buffers, fazendo a utilização de API's REST para comunicação com o frontend quando necessário. Além disso, para o escopo de gerenciamento de redes foi abordado o a utilização de ferramentas de monitoramento de rede como Prometheus e Grafana.*

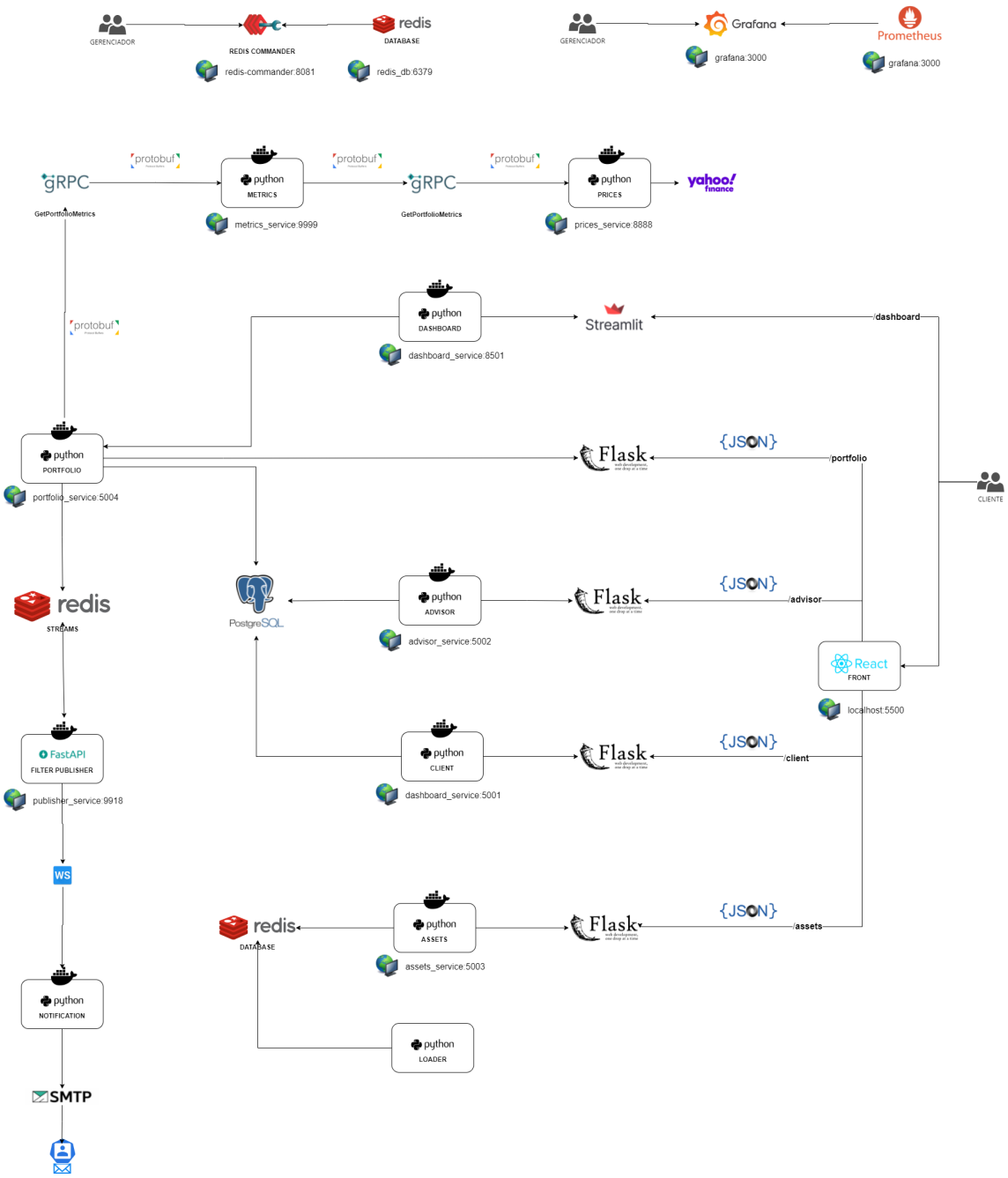
## 1. Introdução

O objetivo deste documento é registrar e descrever detalhadamente, sob a forma de tutorial, o processo de desenvolvimento do trabalho final da disciplina COM241 - Administração e Gerência de Redes de Computadores.

O projeto final foi intitulado Insights e se baseia na implementação de uma infraestrutura que permita a um assessor de investimentos fazer o controle de registro dos seus clientes e realizar a recomendação de portfólios de investimento. Para realizar a análise e composição de um portfólio, o assessor terá acesso a métricas de investimento de risco e retorno, como sharpe, retorno acumulado, volatilidade, fronteira eficiente de Markowitz, dentre outros. A proposta do trabalho é que o fluxo de pedido dessas informações e cálculos sejam realizados através de chamadas gRPC utilizando Protocol Buffers para comunicação. Após o desenvolvimento do portfólio o assessor poderá realizar o backtest de um período, onde serão geradas visualizações para as medidas previamente listadas. Após o backtest e a concretização do portfólio, será possível recomendar a carteira. Nesse processo de recomendação o cliente deverá receber um email com as alocações dos produtos propostos. Em nível arquitetural, o projeto buscou elaborar um ambiente em containers utilizando Docker, assim como a utilização de ferramentas que monitoram a saúde do ambiente.

A partir da proposta de projeto descrita no parágrafo anterior, a equipe formulou a arquitetura representada pela figura abaixo. Nela temos cada serviço separado em um componente, sendo que cada serviço será abordado superficialmente a seguir, para depois ser detalhado em nível de desenvolvimento.

**Arquitetura do sistema Insights**




## 2. Descrição dos Serviços

A partir da definição da arquitetura na imagem acima, iremos descrever superficialmente a função de cada serviço desenvolvido:

**Módulo Advisor:** responsável pela manutenção dos cadastros dos assessores (registro, edição, visualização e remoção).

**Módulo Client:** responsável pela manutenção dos cadastros dos clientes (registro, edição, visualização e remoção).

**Módulo Assets:** responsável pela listagem dos ativos disponíveis na

plataforma, assim como as informações necessárias para gerar uma visualização individual do ativo.

**Módulo Loader:** responsável pela carga das séries históricas de preço de fechamento dos ativos, assim como informações cadastrais dos mesmos, na base de dados. Esses dados foram utilizados para exibir o resumo dos ativos. A plataforma utilizada para o acesso a essas informações foi o Yahoo Finance.

**Módulo Metrics:** responsável pelo cálculo de métricas de mercado relacionadas a um portfólio solicitado. Esse serviço realizará contato constante com o módulo de preços (descrito a seguir), onde solicitará a série histórica de preços de fechamento dos ativos que compõem o portfólio. A partir dessas informações serão calculados métricas como: retorno, retorno acumulado, sharpe, volatilidade, drawdown, correlação, métricas anualizadas e fronteira eficiente de Markowitz.

**Módulo Prices:** responsável pela busca e retorno de informações históricas de um determinado ativo de renda variável. A plataforma utilizada para o acesso a essas informações foi o Yahoo Finance.

**Módulo Portfólio:** responsável pela manutenção dos portfólios (criação, remoção, edição, visualização), recomendação (publicação das recomendações no serviço de stream) e preparação dos portfólios para comunicação com o serviço de métricas, que retornará as medidas de risco e retorno

**Módulo Dashboard:** responsável pela exibição das visualizações geradas pelos cálculos das métricas de um determinado portfólio.

**Módulo Publisher:** responsável pela conexão entre as recomendações publicadas pelo módulo portfólio e o serviço de notificações de email. Esse serviço faz o controle de todas as notificações publicadas no serviço de streams e as direciona para o respectivo cliente solicitante.

**Módulo Notification:** responsável pela distribuição de emails com a composição do portfólio recomendado pelo advisor a um ou mais clientes desejados.

## 3. Tutorial - Tecnologias da Arquitetura

Nesse tópico serão descritas informações técnicas que desenvolverão o tutorial de implementação de recursos arquiteturais do projeto. Assim, o foco do tutorial é demonstrar como desenvolver os recursos utilizados na plataforma. Dessa forma, iremos mostrar como implementar recursos da arquitetura como composição de serviços utilizando docker, build de aplicação através do docker-compose, utilização de bancos de dados através de imagens, implementação e integração de gerenciadores dos bancos de dados e monitoramento de recursos de redes utilizando Prometheus e Grafana.

### 3.1. Implementação dos Dockerfiles

Como foi dito anteriormente, a equipe buscou elaborar o ambiente separando os serviços em containers isolados com o Docker. Para isso precisamos elaborar um arquivo Dockerfile que acompanhará cada serviço implementado. Esse arquivo será utilizado para o build do ambiente, por meio do docker-compose (demonstrado posteriormente), onde serão definidos as configurações do serviço.

Todo o backend do sistema foi encapsulado em containers docker, assim, temos toda a instalação dos pacotes e geração de imagens independentes no sistema, abaixo temos um exemplo de configuração Dockerfile aplicada em um serviço que utiliza gRPC. Nele vemos que não precisamos instalar as dependências e compilar os arquivos protocol buffers no momento de execução, já que as configurações do Docker já executarão esse processo e subirão uma imagem atualizada.

**Exemplo de Dockerfile**

```
FROM python:3.8-slim

RUN mkdir /service
COPY proto/ /service/proto/
COPY metrics/ /service/metrics/

WORKDIR /service/metrics

RUN python -m pip install --upgrade pip
RUN python -m pip install --no-cache-dir -r requirements.txt
RUN python -m grpc_tools.protoc -I ../proto --python_out=. \
          --grpc_python_out=. ../proto/insights.proto

EXPOSE 9999

ENTRYPOINT [ "python", "-u", "app.py" ]
```

Nessa configuração fazemos a definição da linguagem utilizada na primeira linha. Criamos o diretório onde nossa imagem armazenará os recursos. Assim, copiamos o nosso arquivo .proto para esse diretório, evitando que ele apareça várias vezes no nosso projeto. A partir disso, podemos instalar nossas dependências e fazer a compilação de nosso arquivo Protocol Buffers, que gerará dois arquivos apenas no container, economizando espaço no repositório. Além disso, utilizamos o comando EXPOSE para expor a porta do container, sendo possível acessá-lo da rede local. Por fim, basta indicar o código para executar seu script no ENTRYPOINT.

### 3.1.1. Passo a passo - Implementação dos Dockerfiles

- Declarar no topo do arquivo a imagem da linguagem a ser utilizada no container (python:3.8-slim)
- Definir o diretório em que o projeto será executado
- Copiar os arquivos do diretório atual para o que será utilizado no container
- Definir o diretório principal de atuação do container
- Instalar as dependências e bibliotecas que serão utilizadas
- Definir a exposição de porta, caso o serviço precise ser acessado na rede
- Definir o entrypoint de código, que será executado no build do container Docker

### 3.2. Utilização do Docker-Compose nos serviços

A partir da geração dos Dockerfile para cada serviço, conforme descrito acima, podemos gerar um arquivo docker-compose.yaml que definirá todos os recursos utilizados pela aplicação que serão subidos para a execução do projeto. O docker-compose.yaml possui declarações de rede e comunicação entre os serviços, nesse caso ele também foram utilizadas imagens de banco de dados para utilização no projeto (como redis e PostgreSQL), assim, não foi necessário uma instalação local dos bancos, mas sim das imagens através do Docker.

**Exemplo de serviço no docker-compose**

```
metrics_service:
    build:
        context: .
        dockerfile: metrics/Dockerfile
    environment:
        PRICES_HOST: prices_service
    image: metrics
    networks:
        - insights-network
    ports:
        - 9999:9999

volumes:
    redis-volume:
    postgres-volume:
    prometheus-data:
    grafana-data:
    front-portfolio:
    mongo-volume:

networks:
    insights-network:
        driver: bridge
```

Para a definição de um serviço no docker-compose fazemos a definição do build pelo path onde está localizado o Dockerfile que criamos anteriormente. Além disso, podemos definir as variáveis de ambiente com environment, caso necessário. No caso do projeto utilizamos a definição de uma rede do docker (insights-network) que permitirá a comunicação entre os diversos serviços, assim como uma resolução de domínio a partir do nome dos containers.

Além do serviço, também definimos outros dois recursos do docker-compose, as networks e os volumes. A network foi definida para que os containers se comuniquem em uma rede comum durante a execução do projeto. Ela faz uma resolução de domínio dos containers de acordo com o nome especificado para o serviço, assim podemos acessar o serviço de métricas, por exemplo, através do endereço metrics_service:9999.

Já os volumes são declarados para que os dados armazenados durante a execução do projeto não existam só em tempo de execução, armazenando no diretório declarado nos volumes. Assim, teremos os dados persistidos mesmo após o encerramento dos containers.

### 3.2.1. Passo a passo - Utilização do Docker-Compose nos serviços

- Declarar o nome do serviço (metrics_service)
- No item buid, definir o diretório onde está localizado o Dockerfile que será utilizado para compor a aplicação
- No item environment, definir as variáveis de ambiente que serão utilizadas pelo container
- No item image, definir a imagem que será utilizada (caso já exista, como no caso dos bancos de dados) ou definir um nome para sua imagem customizada
- No item networks devemos declarar a rede Docker que será utilizada no projeto
- Definir a porta em que o serviço ficará hospedado
- Definir o entrypoint de código, que será executado no build do container Docker
- Definir os volumes que armazenarão os dados da aplicação durante a execução dos containters, para que eles não sejam perdidos após o encerramento dos containers
- Definir a rede que será utilizada pelo projeto, o modo bridge foi especificado para a comunicação interna entre os containers

## 3.3. Utilização do Docker-Compose para os bancos de dados

Conforme definido anteriormente foram utilizados os bancos Redis e PostgreSQL para o desenvolvimento do projeto. Para melhorar a portabilidade do projeto, os bancos de dados foram integrados ao projeto por meio do docker-compose, não necessitando instalação local dos mesmos, já que serão utilizadas imagens Docker que serão instaladas durante o build da aplicação. Abaixo temos a declaração do Redis:

**Definição do redis no docker-compose**

```
redis_db:
    container_name: redis_db
    command: bash -c "redis-server --appendonly yes"
    image: redis
    ports:
        - "6379:6379"
    volumes:
        - ./redis-volume:/data
    networks:
        - insights-network
```

Para a definição do banco utilizamos uma imagem externa, nesse caso o redis. Definimos as portas padrão utilizadas pelo banco de dados, no caso 6379. Um

aspecto importante para os bancos de dados no docker é a definição dos volumes, que permitirão que os dados fiquem salvos mesmo após a finalização dos containers, assim, todos os dados gerados em tempo de execução ficarão salvos. Adicionamos também esse recurso na rede (insights-network) para comunicação com os outros recursos. Seguindo o mesmo estilo de declaração do redis, temos o postgreSQL:

**Definição do PostgreSQL no docker-compose**

```
postgres_db:
    image: postgres
    environment:
        POSTGRES_PASSWORD: "senha_trabalho_unifei"
    ports:
        - "15432:5432"
    volumes:
        - ./postgres-volume:/var/lib/postgresql/data
    networks:
        - insights-network
```

### 3.3.1. Passo a passo - Utilização do Docker-Compose para os bancos de dado

- Declarar o nome do serviço de base de dados (utilizado como host para o acesso na rede)
- Definir a senha do banco através da determinada variável de ambiente especificada na documentação
- Declarar a imagem do banco de dados utilizada
- Definir a porta em que o banco de dados ficará hospedado, alguns bancos utilizam portas padrão
- No item volumes, definir o diretório onde serão persistidos os dados armazenados pelo gerenciado do banco de dados
- Declarar a rede em que o banco estará localizado

## 3.4. Utilização de Gerenciadores de Banco de Dados

Conforme descrito anteriormente foram utilizados os bancos de dados Redis e PostgreSQL. Para o gerenciamento dessas bases de dados foram utilizadas as imagens Docker dos serviços: redis-commander e pg-admin. Ambas foram instaladas e linkadas com os bancos através do docker-compose, podemos ver abaixo a definição com o redis-commander.

**Integração e build redis / redis-commander**

```
redis-commander:
    image: rediscommander/redis-commander:latest
    depends_on:
        - redis_db
    environment:
        - REDIS_HOSTS=local:redis_db:6379
```

```
    ports:
        - 8081:8081
    networks:
        - insights-network
```

Devemos nos atentar para a declaração do parâmetro depends_on que determinará a inicialização do redis antes do redis_commander, evitando problemas de conexão. Além disso, nesse caso, temos também a definição da variável de ambiente (REDIS_HOSTS), que determinará o link de comunicação entre esses dois recursos. Abaixo temos, de forma semelhante, a utilização do pgadmin:

**Integração e build redis / redis-commander**

```
pgadmin:
    image: dpage/pgadmin4
    environment:
        PGADMIN_DEFAULT_EMAIL: "email@hotmail.com"
        PGADMIN_DEFAULT_PASSWORD: "senha"
    ports:
        - "16543:80"
    depends_on:
        - postgres_db
    networks:
        - insights-network
```

No caso do pgadmin temos a integração automática com o postgreSQL pela porta padrão (15432), necessitando apenas da declaração de usuário e senha que serão utilizados para o acesso ao painel de gerenciamento.

Após o pgadmin estar configurado com os passos definidos podemos criar um servidor portgreSQL e o banco de dados em que a aplicação será executada. Para isso execute os seguintes passos:

1. Acesse o pgadmin em localhost:16543
2. Realize o login e senha com as variáveis de ambiente definidas no docker-compose.yaml
3. Clique em create server
4. Defina um nome para o seu servidor na aba General
5. Na aba Connection, o Host deve ser preenchido com o nome do seu serviço postgreSQL (no caso, postgres_db)
6. Preencha o campo Port com a porta do postgres_db (no caso, 5432)
7. Definia o database e o Username a partir do nome da imagem (no caso, postgres)
8. Finalize o preenchimento do campo Password com o valor definido na variável de ambiente do docker-compose
9. Salve o servidor e crie o banco de dados a partir da estrutura .sql.

### 3.5. Utilização do Prometheus e Grafana para Monitoramento dos Serviços

Para o monitoramento da rede e dos serviços foi utilizado o Prometheus, que faz a coleta de métricas definidas em nível de código. O Grafana utilizado em associação ao Prometheus para a disponibilização de uma interface mais amigável e para a composição de dashboards. Ambos foram instalados por imagens definidas no docker-compose.

**Definição dos serviços a serem monitorados pelo Prometheus**

```
global:
  scrape_interval: 15s
  scrape_timeout: 10s

rule_files:
  - alert.yml

scrape_configs:
  - job_name: prometheus
      static_configs:
        - targets:
              - 'prometheus:9090'

  - job_name: apis
      static_configs:
        - targets:
              - 'client_service:5001'
              - 'advisor_service:5002'
              - 'assets_service:5003'
              - 'portfolio_service:5004'
              - 'collector_service:5005'

  - job_name: metrics
      static_configs:
        - targets:
              - 'metrics_service:9998'
              - 'prices_service:8889'
```

Conforme mostrado acima fazemos a definição do escopo de monitoramento no arquivo prometheus.yaml. Nele definimos o intervalo de tempo de captura das métricas, assim como os targets (alvos da captura de dados). Como definimos uma rede docker, conforme citado anteriormente, os targets fazem referência ao nome do serviço e a porta utilizada.

**Passos para a definição do Prometheus:**

1. Instale a imagem no docker-compose conforme o arquivo
2. Crie o arquivo prometheus.yml dentro da pasta prometheus para configurações de scraping

3. Crie a pasta prometheus-data dentro da pasta prometheus para a persistência dos dados
4. No arquivo prometheus.yml defina as variaveis globais screape_interval (intervalo entre as capturas de métricas) e scrape_timeout (definição de tempo para serviço inativo)
5. Defina os jobs a serem analisados a partir da definição do scrape_configs, cada job acompanha um job_name que o caracteriza, assim como um conjunto de serviços definidos pelo campo targets. Os targets devem ser definidos com host:port.

Também devemos definir as métricas a serem capturadas pelo Prometheus dentro do código do serviço a ser monitorado, sendo que esse serviço também deve fornecer uma rota de captura de métricas ao Prometheus (normalmente /metrics), abaixo vemos um exemplo de um contador de execução em uma função de API.

**Exemplo de disponibilização de métricas para o Prometheus**

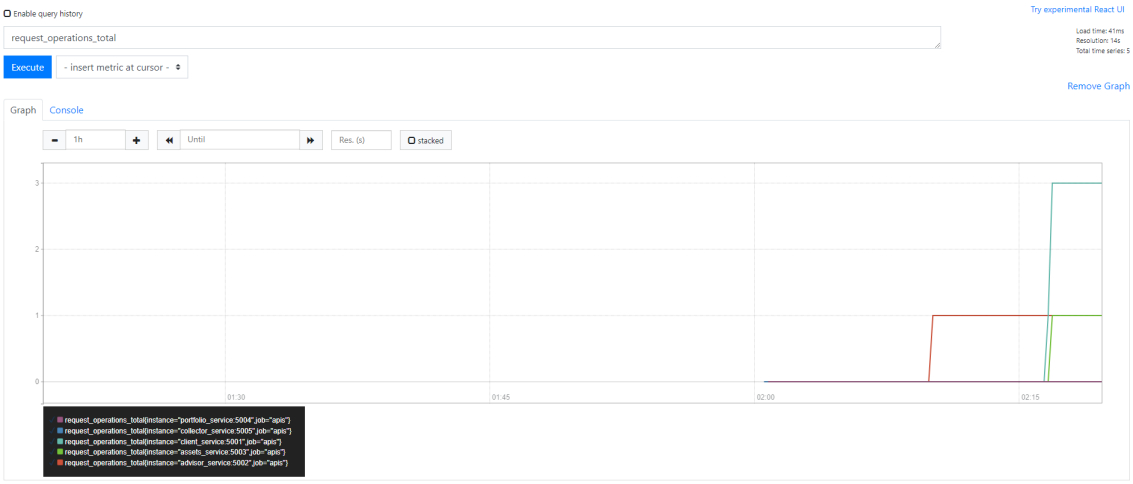
```
from prometheus_client import Counter, Histogram
graphs = dict()
graphs["c"] = Counter("request_operations_total", "Total number of
    ↪ processed requests")

@app.route("/")
@cross_origin()
def hello():
    graphs["c"].inc()
    return {"status": 200}


@app.route("/metrics")
def prometheus_metrics():
    res = []
    for k, v in graphs.items():
        res.append(prometheus_client.generate_latest(v))
    return Response(res, mimetype="text/plain")
```

**Passos para a captura de métricas para o Prometheus:**

1. Encontre a biblioteca adequada do prometheus para o serviço utilizado
2. Defina os nomes dos objetos de captura que serão utilizados para as métricas
3. Adicione as operações associadas a esses objetos e realize suas funções no código desejado
4. Disponibilize uma rota de acesso no serviço no path /metrics, para disponibilização da rota de scraping utilizada pelo Prometheus.

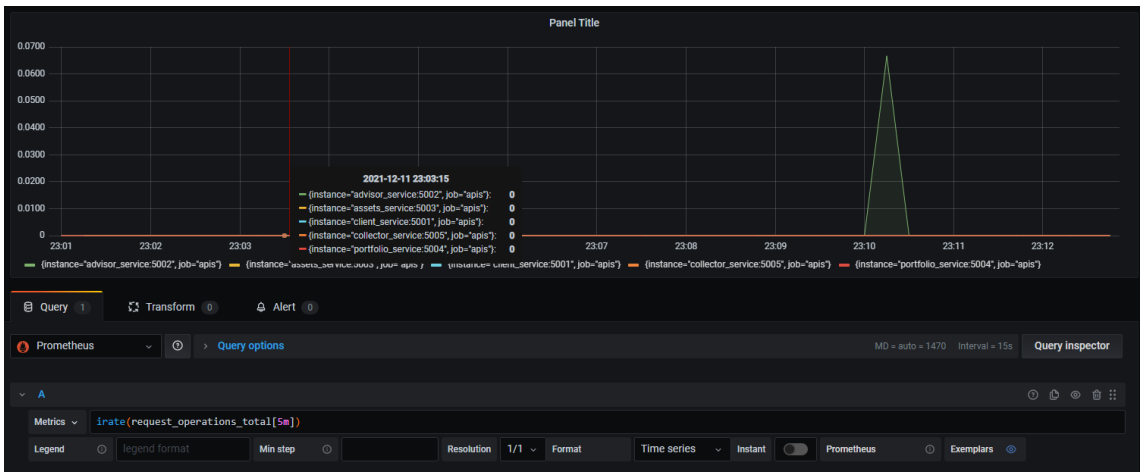
Foi definido um contador com nome "request_operations_total" onde é incrementado toda vez que a função hello() é executada. Conforme citado, devemos expor uma rota /metrics para a captura das informações pelo Prometheus. A seguir, vemos a execução deste contador no Prometheus:

## Exemplo de query Prometheus




Após a definição das métricas dentro dos serviços, podemos utilizar essas mesmas métricas para a composição de um dashboard no Grafana através da interface com o Prometheus. Abaixo temos um exemplo de query semelhante ao listado acima, assim como um dashboard de exemplo:

## Exemplo de query Prometheus no Grafana



**Exemplo de dashboard Grafana**


## 4. Tutorial - Tecnologia dos Serviços

Nesse tópico serão descritas informações técnicas que desenvolverão o tutorial do projeto. Assim, o foco do tutorial é demonstrar como desenvolver os recursos utilizados na plataforma. Dessa forma, iremos mostrar como implementar recursos da arquitetura e dos serviços. No caso de soluções redundantes, como a elaboração das APIs utilizando Flask, será utilizado um serviço de exemplo.

### 4.1. Implementação do gRPC (Metrics / Prices)

O gRPC é um método de chamada de procedimento remoto desenvolvido pelo Google, ele foi utilizado no projeto com o objetivo de ganho de performance e para a manutenção de dados estruturados fortemente tipados, já que utiliza como transporte, dados no formato Protocol Buffers.

Para a explicação do gRPC precisamos primeiro entender como montar um schema Protocol Buffers, onde serão definidas nossas mensagens e as chamadas gRPC. O schema protobuf é desenvolvido em um arquivo com extensão .proto. Abaixo temos um exemplo de estrutura:

**Exemplo de schema protobuf**

```
service Metrics {
    rpc GetMetrics(MetricsRequest) returns (MetricsResponse);

message Metric {
    string name = 1;
    float value = 2;
}

message MetricsResponse {
    string name = 1;
    repeated Metric metrics = 2;
}
```

```
message MetricsRequest {
    string base_date = 1;
    string ticker = 2;
}
```

Acima, podemos ver como é declarado um request response de gRPC, para isso, definimos um service. O service possuirá os modelos de chamada que serão utilizados, esses modelos são definidos por: rpc <NomeDaFunção> (<MensagemRequisição>) returns <MensagemResposta>, sendo que as mensagens são as estruturas fortemente tipadas.

Essas mensagens definirão o formato do dado que será trafegado. Seu formato é: <tipo> <nomeDaVariável> = <número>, sendo que o número costuma ser sequencial e iniciado em 1, esse número definirá a posição da variável no objeto serializado. Além disso, podemos ter listas de dados e estruturas compostas, como o exemplo: repeated Metric metrics = 2, nesse caso repeated indica uma lista de dados e Metric representa o tipo (do tipo mensagem, ou seja, outra estrutura).

Após a definição do .proto devemos compilar esse arquivo para a linguagem desejada, assim teremos um método de importar os recursos do gRPC / protobuf dentro do código. Para a compilação do arquivo do projeto em python temos o seguinte código (a instalação da biblioteca grpcio-tools é requerida - pip install grpcio-tools):

```
python -m grpc\_tools.protoc -I ../proto --python\_out=. \ --grpc\
    ↪ _python\_out=. ../proto/insights.proto
```

Após a execução teremos dois arquivos como resultado. Um será o arquivo responsável pela serialização e deserialização dos dados no formato protobuf, o outro possuirá as definições para as stubs e servicers gRPC delarados no .proto. A stub nada mais é que o client do gRPC, enquanto o servicer é o servidor que faz as declarações das funções.

Após essa etapa podemos desenvolver o nosso servidor e nosso cliente. Abaixo teremos a definição do servidor (servicer), que será explicado em seguida.

**Código de definição de um servicer gRPC**

```
import insights_pb2_grpc as grpc_service
from insights_pb2 import MetricsResponse

def server_setup():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10),
        ↪ interceptors=(PromServerInterceptor(
        ↪ enable_handling_time_histogram=True),))
    grpc_service.add_MetricsServicer_to_server(MetricsServicer(),
        ↪ server)
    server.add_insecure_port(f"metrics_service:{metrics_port}")
    start_http_server(9998)
    try:
```

```
        server.start()
        logger.info(f"Server is running on metrics_service:{
            ↪ metrics_port}")
        server.wait_for_termination()
    except KeyboardInterrupt:
        logger.info("Stopping metrics service")
        server.stop(0)

class MetricsServicer(grpc_service.MetricsServicer):
    def GetMetrics(self, request, context):
        data = get_prices(request)
        if data.ByteSize():
            df = compose_df(data)
            if df.empty:
                return MetricsResponse()
            logger.info(f"Calculating metrics: {request.ticker}")
            metrics = Metrics(request).get_metrics(df)
            return response_builder(metrics, data.name)
        else:
            return MetricsResponse()

if __name__ == "__main__":
    print(f"Starting metrics server on port {metrics_port}", file=sys
        ↪ .stderr)
    server_setup()
```

Para a definição do servidor, primeiro precisamos definir o que nossa função declarada no .proto executará, fazemos isso com a definição de uma classe com o mesmo nome do servicer declarado no .proto e que herdará o servicer (grpc_service.MetricsServicer) definido anteriormente, sobrescrevendo assim, as funções. Assim, podemos definir o código da função, que deverá possuir o mesmo nome para que o método seja sobrescrito. É importante que o método retorne e receba as mesmas estruturas de dados definidas no .proto, caso contrário, o serviço não funcionará.

Após realizada a definição do servicer devemos subir nosso servidor gRPC. Fazemos isso com a definição do server (server = grpc.server()) associando-o ao servicer previamente definido conforme o código: grpc_service.add_MetricsServicer_to_server(MetricsServicer(), server). Assim, basta associar nosso serviço a um host e porta: server.add_insecure_port(f"host:porta"). E por fim, iniciar o servidor: server.start().

Com a definição do servidor pronta, podemos criar as interfaces dos clientes que se comunicarão com o servidor definido anteriormente. Abaixo temos um exemplo dessa comunicação:

```
from insights_pb2 import InfoRequest
```

```
from insights_pb2_grpc import InfoStub

def get_prices(request):
    with grpc.insecure_channel(f"prices_service:{prices_port}") as
        ↪ channel:
        stub = InfoStub(channel)
        start_date = get_12m(request.base_date)
        request = InfoRequest(ticker=f"{request.ticker}.SA",
            ↪ start_date=start_date, end_date=request.base_date)
        prices = stub.GetInfo(request)
        return prices
```

Acima temos uma chamada para o servicer GetInfo, começamos nossa conexão abrindo um canal grpc com o host e porta em que o servicer está localizado. A partir disso, declaramos a stub por meio do canal definido. Depois, basta compor nossa requisição seguindo o schema .proto que definimos anteriormente (o schema deve ser seguido para que não ocorra erros de mapeamento no servicer). Por fim, basta fazer a requisição com a execução da função (mesmo nome definido no servicer) a partir da stub. O retorno será do tipo definido no modelo request response do .proto.

#### 4.1.1. Passo a passo - Implementação do gRPC (Metrics / Prices)

- Criação e desenvolvimento do arquivo .proto, onde será mantido o schema do protobuf e a declaração das chamadas gRPC
- Compilação do .proto na linguagem desejada, gerando dois arquivos (servicer/stub gRPC e serialização protobuf)
- Definição do cliente (stub) que fará as requisições a uma função mantida pelo servidor
- Definição do servidor (servicer) e exposição do server em determinada porta
- Declaração da função gRPC no servidor que desempenhará determinada tarefa

### 4.2. Implementação da API Flask (Advisor / Clients / Assets)

Para a comunicação do backend com o frontend, foram utilizadas api's em python utilizando a biblioteca Flask, fornecendo endpoints HTTP REST que trafegam objetos no formato JSON. Abaixo temos um exemplo de declaração de um endpoint utilizando Flask, esse modelo pode ser replicado para qualquer endpoint a ser desenvolvido, mantendo a estrutura base e mudando o conteúdo executado pela função.

```
@app.route("/assets/info", methods=["POST"])
@cross_origin()
def get_assets_info():
    try:
        symbols = request.get_json().get("symbols", [])
        assets_info = get_symbols(symbols)
        response = jsonify(assets_info)
```

```
        return response
    except Exception as e:
        return str(e)
```

Alguns pontos chave devem ser levados em consideração durante a implementação dos endpoints. Um endpoint Flask é determinado por um decorator como: @app.route("/assets/info", methods=["POST"]), assim, temos o formato @app.route("<path>", methods=[<metodos HTTP>]). Essa será a rota que poderá ser acessada pelas requisições. Para o retorno dos dados podemos utilizar o jsonify, que garantirá que o objeto retornado seja do tipo JSON.

Um aspecto importante percebido durante o desenvolvimento, foi a necessidade das definições de CORS (Cross-Origin Resource Sharing) na API, isso porque o frontend está hospedado em outro endereço, e a não definição das permissões CORS pode levar a um erro de comunicação entre o front e a API, que bloqueará os recursos pela restrição, não permitindo que uma página recupere os recursos de um outro domínio, fora do domínio da aplicação. Assim, podemos utilizar uma extensão do Flask para definir a política de CORS conforme o exemplo abaixo:

```
from flask_cors import CORS, cross_origin

app = Flask(__name__)
cors = CORS(app)
```

Após essa definição basta adicionar o decorator @cross_origin() nos endpoints da API. Por fim, podemos iniciar nossa API Flask definindo o arquivo da nossa api pelo comando export FLASK_APP=<arquivo.py> e executando-o com: flask run –host=host –port=porta

#### 4.2.1. Passo a passo - Implementação da API Flask (Advisor / Clients / Assets)

- Definição das variáveis de ambiente e porta onde o servidor será hospedado
- Definição das rotas declarando explicitamente os protocólos de comunicação
- Definição da função desempenhada por cada rota especificada
- Declaração das rotas que possuirão permissão de CORS

### 4.3. Implementação do Servidor WebSocket (Publisher)

Conforme definido anteriormente, foi utilizado o Redis Streams associado a um servidor websocket para fazer o controle de recomendações de portfólio para envio de email. Para a publicação das mensagens no redis, iniciamos uma sessão e utilizamos o comando xadd, conforme o exemplo abaixo (o host foi definido como redis_db devido a utilização da rede Docker - explicada posteriormente):

```
def send_notifications(portfolio, clients):
    try:
        client = redis.Redis(host="redis_db", port=6379)
        _portfolio = ast.literal_eval(portfolio)
```

```
        for c in clients:
            _portfolio["email"] = c["email"]
            _portfolio["name"] = c["name"]
            client.xadd("portfolio", message_encoder(_portfolio))
    except Exception as e:
        raise e
```

Conforme visto, além da sessão Redis aberta, utilizamos o comando xadd para publicar a mensagem seguindo o seguinte formato: <sessão>.xadd("<tópico>", <mensagem>)

Após o envio da mensagem ao Redis Streams, teremos um servidor que escutará todas as mensagens que chegam no tópico registrado. Para definição desse servidor utilizamos a biblioteca Fast API do python. A seguir temos a base para a definição do servidor websocket utilizando Fast API:

```
@app.websocket("/stream/{stream}")
async def proxy_stream(
    ws: WebSocket,
):
    await ws.accept()

        try:
                await ws.send_json(results)
        except (ConnectionClosed, WebSocketDisconnect):
                logger.info(f"{ws} disconnected from stream {stream}")
                return
```

### 4.3.1. Passo a passo - Implementação do Servidor WebSocket (Publisher)

- Definir a comunicação com o banco de dados utilizando o endereço do serviço e a porta declarados no docker-compose
- Publicação das mensagens na fila utilizando o comando XADD e definindo explicitamente a fila alvo
- Definição do servidor websocket a partir do decorator @app.websocket
- Abrir o servidor para comunicação com o comando ws.accept() no path declarado no decorator

## 4.4. Implementação do Client WebSocket (Notification)

Conforme citado anteriormente, para receber as recomendações de portfólio disparadas quando um assessor solicita, temos um serviço que se comunica com o serviço publisher via websocket, nesse caso ele escuta ativamente como cliente no path /portfoliol do publisher. Para cada mensagem recebida nesse websocket, direcionamos a mensagem para uma função que constrói e envia um email ao cliente. Para o envio do email foi utilizado SMTP por meio da smtplib, assim como o Jinja para a composição do template que será encaminhado. Abaixo temos a definição do cliente websocket:

**Código de conexão cliente ao websocket**

```
def on_message(ws, messages):
    messages = json.loads(messages)
    send_email(messages)

def on_error(ws, error):
    print(f"--- Websocket Error ---\nMessage:{error}")

def on_close(ws, close_status_code, close_msg):
   print(f"--- Connection Closed ---\nStatus:{close_status_code}\
      ↪ nMessage:{close_msg}")

if __name__ == "__main__":
    websocket.enableTrace(True)
    ws = websocket.WebSocketApp("ws://publisher_service:8001/stream/
       ↪ portfolio",
                                on_message=on_message,
                                on_error=on_error,
                                on_close=on_close)

    while True:
        try:
            ws.run_forever()
        except:
            pass
```

Como vemos acima, iniciamos o WebSocketApp da biblioteca websocket do Python, definindo o host e porta do servidor, assim como o path a ser conectado. Após isso, temos três tratativas para as conexões: mensagem, erro, fechamento de conexão; onde cada um determina uma função a ser executada. No caso de mensagem (on_message) iremos tratar a mensagem e fazer o envio do email. Para o envio do email temos o exemplo abaixo:

**Envio de email através do servidor SMTP**

```
def send_email(messages):
    try:
        with smtplib.SMTP("smtp.gmail.com", 587) as server:
            server.starttls(context=context)
            server.login(source_email, source_password)
            for message in messages:
                email_message = build_products_message(message)
                logger.info(f"Sending products email to {message[0]['
                   ↪ email']}")
                server.sendmail(source_email, message[0]["email"],
                   ↪ email_message.as_string())
    except Exception as e:
        logger.error(e)
```

Abrimos a conexão SMTP com smtplib através do servidor gmail (servidor de email utilizado pela equipe), definimos o tls para protocolo de segurança na comunicação do processo de envio na rede. Após isso, logamos no servidor com nossa conta Gmail, e por fim fazemos o envio da mensagem (sendemail) após a composição do template.

### 4.4.1. Passo a passo - Implementação do Client WebSocket (Notification)

- Definir o cliente websocket no host e portas alvo definidos no docket-compose
- Declarar estrutura de funções on_message, on_error, on_close
- Definir a ação das funções declaradas acima
- Escutar o servidor websocket através do comando ws.run_forever()
- Estabecer a conexão SMTP com o servidor de email utilizado (no caso gmail)
- Realizar o login com as credenciais utilizando o tls para comunicação segura
- Fazer o envio do email através do comando server.sendmail definindo a mensagem e o email alvo

## 4.5. Consumo da API pelo front-end

A transmissão de informações entre o back-end e o front-end foi feito através de requisições à API, que retornava objetos JSON. As requisições foram feitas através do Axios (cliente HTTP baseado em promises para o node.js). As requisições feitas pela nossa aplicação foram do tipo GET e POST, que possuem estrutura bem parecidas, pelo Axios (demonstradas no exemplo de código abaixo). As Mudanças principais: o tipo do método através do atributo "method", a url do endpoint, a presença ou não do atributo data (que corresponde a uma string com valores no corpo da requisição POST).

**Código de requisição pelo Axios**

```
var config = {
  method: 'post',
  url: 'localhost:5003/assets/info',
  headers: {
    'Content-Type': 'application/json'
  },
  data : data
};

axios(config)
.then(function (response) {
  console.log(JSON.stringify(response.data));
})
.catch(function (error) {
  console.log(error);
});
```

Os dados são retornados no uso do then, que terá uma função com a resposta. A resposta retornará não só os dados, quanto outras informações do HTTP que podem ser úteis, inclusive no tratamento de erros por parte do front-end. Mas nos casos onde tudo ocorre corretamente, o "response.data" trará as informações que eram desejadas ao realizar a requisição e podem ser associadas à uma variável e exploradas posteriormente.

## 5. Execução do Projeto

- Baixe o docker seguindo as orientações pelo link, caso necessário
- Execute o comando docker-compose up no diretório base do projeto, esse comando irá buildar o backend e os serviços de banco de dados / gerenciamento
- Na pasta front-portfolio, execute npm install e em seguida npm start para iniciar o frontend da aplicação
- Acesse o frontend da aplicação pelo endereço localhost:5500