INSIGHTS

COM242 - Sistemas Distribuídos

Documentação Insights

Divulgação automática de produtos de investimento

Carlos Henrique Souza Silva - 2019015979
João Lucas Ribeiro - 2019005856
Lucas Tiense Blazzi - 2018003310
Marcelo Cavalca Filho - 33161
Robson de Arruda Silva - 2019013624

Professor: Dr. Rafael de Magalhães Dias Frinhani

1. INTRODUÇÃO

O objetivo deste documento é registrar e explicar o processo de desenvolvimento do trabalho final da disciplina COM242 - Sistemas Distribuídos. O projeto final foi intitulado Insights e foi baseado no cenário 3 conforme descrito abaixo:

> ***Cenário 3: Divulgação Automática***
>
> *"Uma corretora especializada em carteiras de investimentos deseja melhorar o formato de ofertas para seus clientes. Atualmente ofertas de investimentos são feitas apenas por indicação direta de um analista para um cliente, conforme seu e interesses. Entretanto a diretoria tem observado que esta abordagem não está sendo eficiente uma vez que são mais de 5000 clientes para um corpo de 200 analistas. A corretora precisa de uma solução que automatiza o processo de envio de informações sobre produtos de investimento com base nos desejos de cada cliente (ex. segmento de mercado, produto, valor)."*

A partir deste cenário, desenvolveu-se uma adaptação de contexto, que foi definido em sequência e segue descrito abaixo:

> ***Adaptação de contexto***
>
> *"Uma casa de research possui um conjunto de clientes com interesses em oportunidades de investimento que atendam seus requisitos básicos em relação às características dos produtos ofertados.*
> *A empresa quer enviar notificações relacionadas a ofertas promissoras que entram no escopo de recomendações e que ao mesmo tempo se encaixam nos requisitos de interesse dos clientes.*
> *A empresa atualmente gasta muito com assessores de investimento e deseja automatizar a proposta de ativos a partir dos produtos em destaque disponibilizados por corretoras de investimento."*

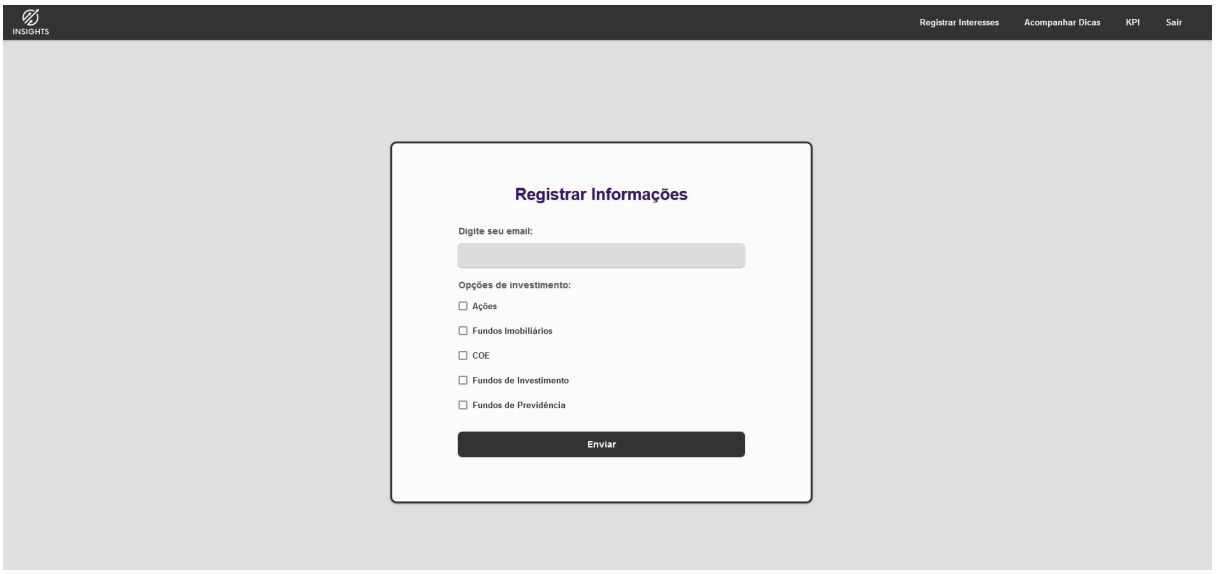
2. DESCRIÇÃO DOS MÓDULOS

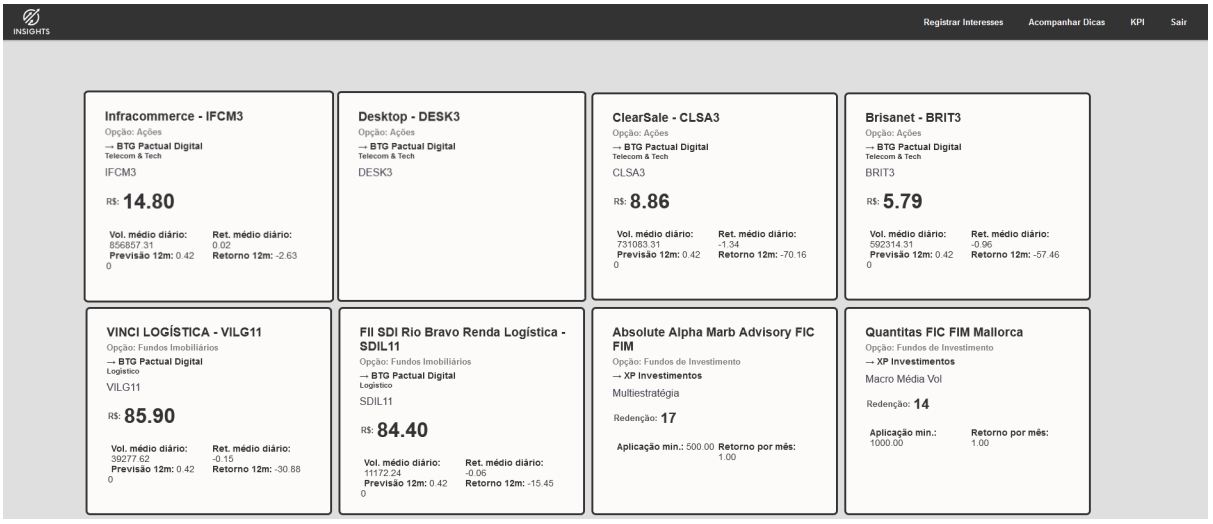
A partir da definição do problema e seu respectivo contexto, a elaboração deste trabalho propõe um modelo de negócio, dividindo o projeto em 7 módulos principais, que serão descritos nesta sessão.

2.1 MÓDULO DO CLIENTE

Este módulo é responsável pela interface de comunicação entre o usuário e a plataforma, sendo feito, portanto, o registro dos filtros de interesses do cliente em relação aos produtos de investimento que serão distribuídos na plataforma.

Além disso, nessa interface, o cliente pode também acessar as notificações, que são atualizadas em tempo real, assim como o acesso à análise de distribuição de produtos pelo sistema.





2.2 MÓDULO DE COLETA

Para este módulo, atribui-se as responsabilidades pela busca de produtos de investimentos nas áreas de destaque de corretoras de investimento (utilizados BTG Pactual e XP Investimentos). Esse módulo realizará a coleta e normalização de produtos para o modelo próprio do sistema, nesse caso, serão coletados produtos das classes: Fundos de Investimento, Ações, Fundos de Investimento Imobiliários, COE e Fundos de Previdência. No caso do produto ser um ativo de renda variável, o módulo de coleta se comunicará com o módulo de métricas (descrito a seguir) para adicionar dados de mercado do produto. Após a coleta desses dados, o módulo fará a publicação das informações no módulo de notificação, que fará o filtro e a distribuição dos dados.

2.3 MÓDULO DE MÉTRICAS

O cálculo de métricas de mercado relacionadas a um produto de renda variável solicitado é atribuído a este módulo, que realizará contato constante com o módulo de preços (descrito a seguir), onde solicitará a série histórica de preços de fechamento e volume de mercado. A partir dessas informações será calculado métricas como: dividend yield, retorno médio diário, volume médio diário e retorno total nos últimos 12 meses.

2.4 MÓDULO DE SÉRIES

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.

2.5 MÓDULO DE NOTIFICAÇÃO

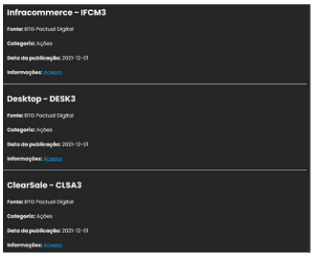
Responsável pela distribuição dos produtos de investimento entre os diversos serviços conectados. Esse módulo faz a divulgação dos produtos de acordo com a conexão estabelecida pelo serviço cliente, assim ele também aplica os filtros de acordo com o perfil do usuário registrado antes do envio da divulgação.

2.6 MÓDULO DE ANÁLISE

Responsável pelo agrupamento dos dados divulgados pelo serviço de notificação. Esse módulo é conectado ativamente ao módulo de notificação e recebe todas as informações divulgadas por ele. Desse modo, o serviço faz a contagem do número de produtos divulgados, de acordo com a classificação e de acordo com o produto em si. A partir disso, é disponibilizado ao módulo do cliente as informações de quantidade de produtos divulgados por classe de ativo, assim como os TOP 10 produtos recomendados.

2.7 MÓDULO DE NOTIFICAÇÃO POR E-MAIL

Responsável pela distribuição de email com os produtos de investimento filtrados de acordo com o interesse registrado pelo usuário na plataforma. A distribuição de emails é realizada a todos os usuários toda vez que o módulo de coleta realiza o registro de novos produtos.



3. DESCRIÇÃO TÉCNICA

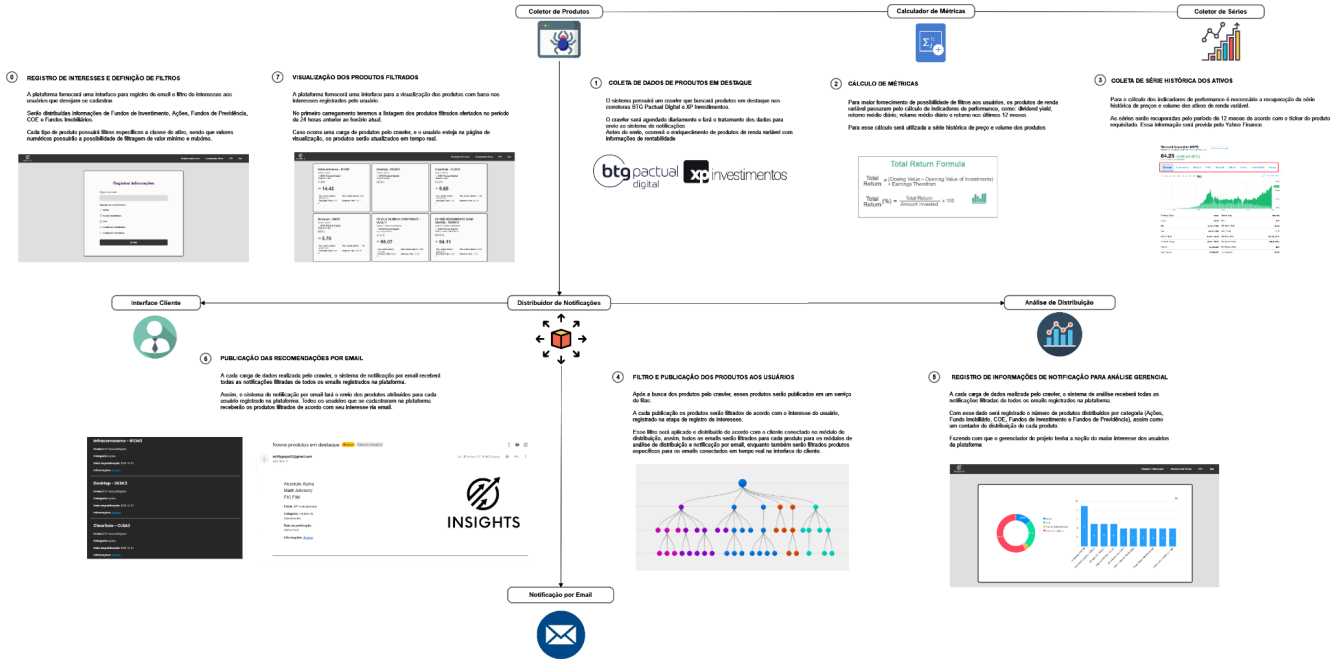
Nesse tópico serão descritas informações técnicas do projeto, acompanhando os diagramas implementados para a projeção e desenvolvimento da arquitetura, assim como a descrição de aspectos técnicos dos serviços elaborados.

3.1 DIAGRAMAS

A seguir estão listados os diagramas elaborados para o desenvolvimento do projeto. Os diagramas podem ser acessados através dos arquivos contidos no diretório para uma melhor visualização. Todos os processos documentados pelos diagramas foram descritos detalhadamente nos módulos 2. Descrição dos módulos e 3.2 Implementação, onde teremos a descrição do funcionamento do sistema, comunicação entre os serviços e bibliotecas utilizadas.

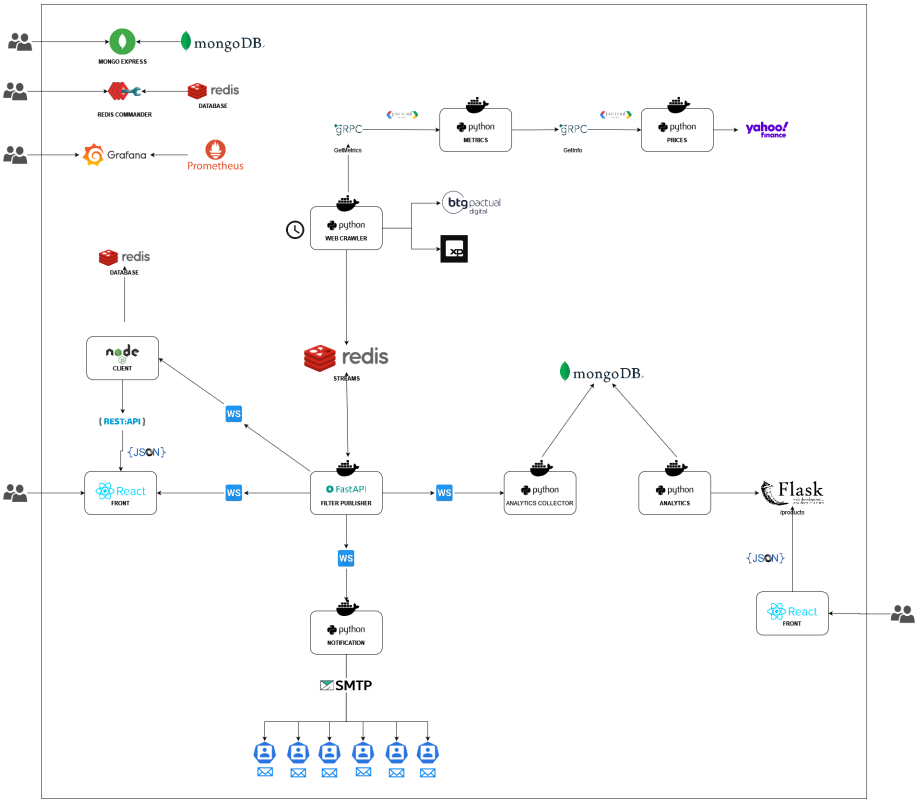
3.1.1 Modelo de Negócio

A seguir, temos a visualização da proposta de modelo de negócio, todos os módulos do sistema foram descritos detalhadamente no item anterior 2. Descrição dos módulos.



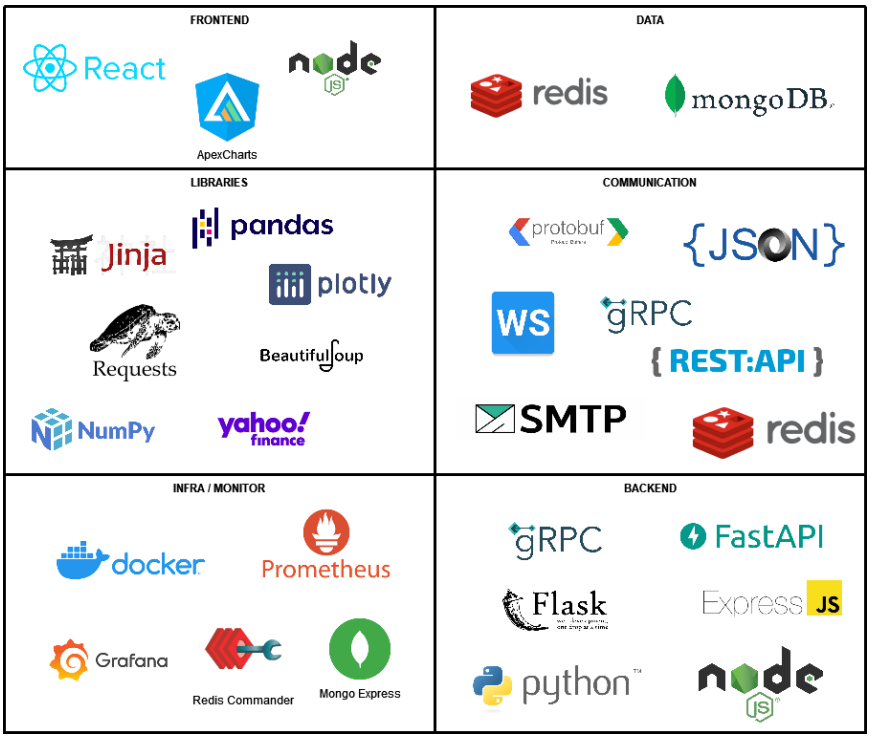
3.1.2 Arquitetura

Abaixo temos a visualização dos componentes que compõem a arquitetura do sistema, todos os componentes e serviços foram descritos detalhadamente no item 3.2 Implementação.

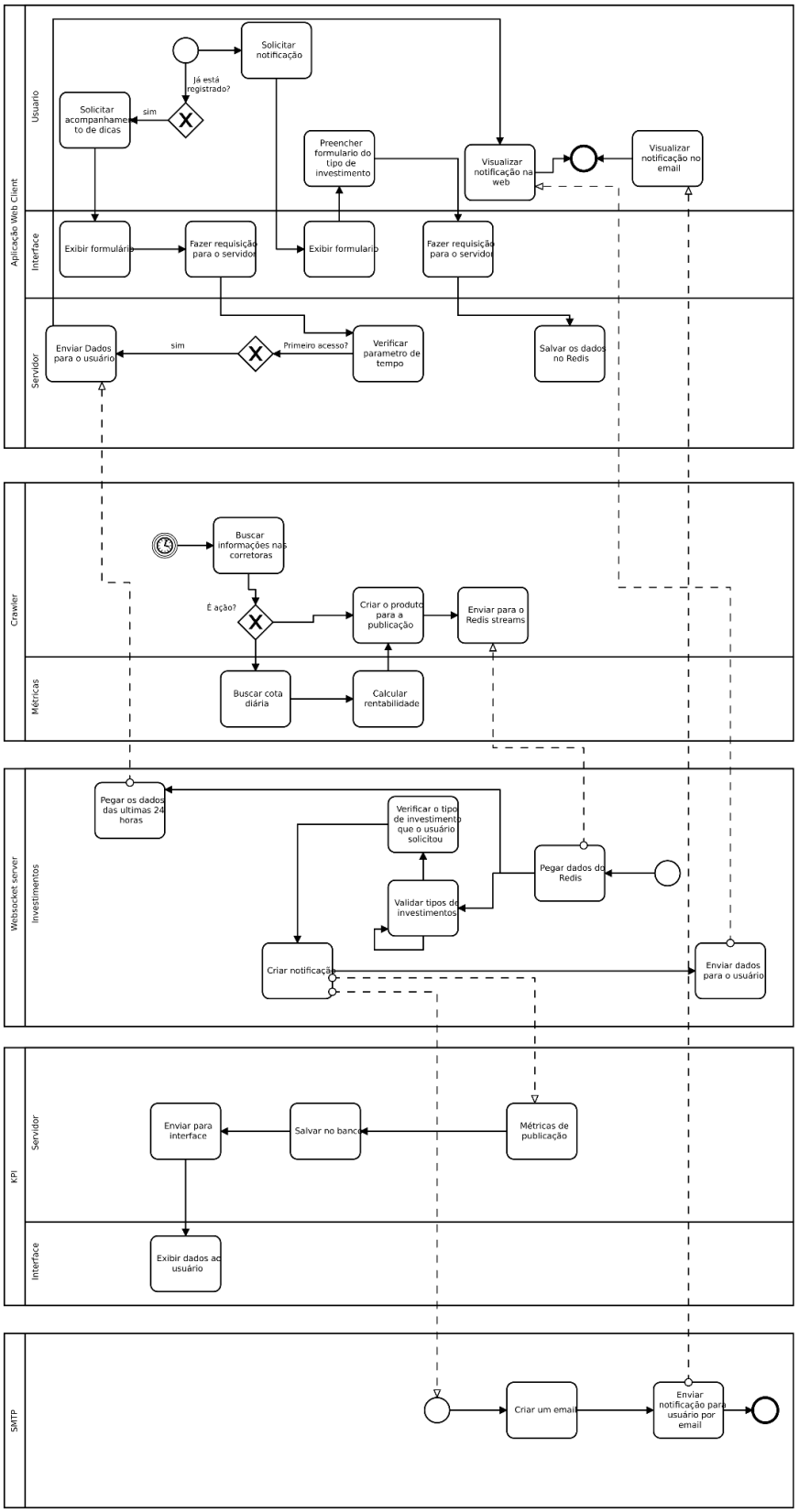


3.1.3 Tecnologias Utilizadas

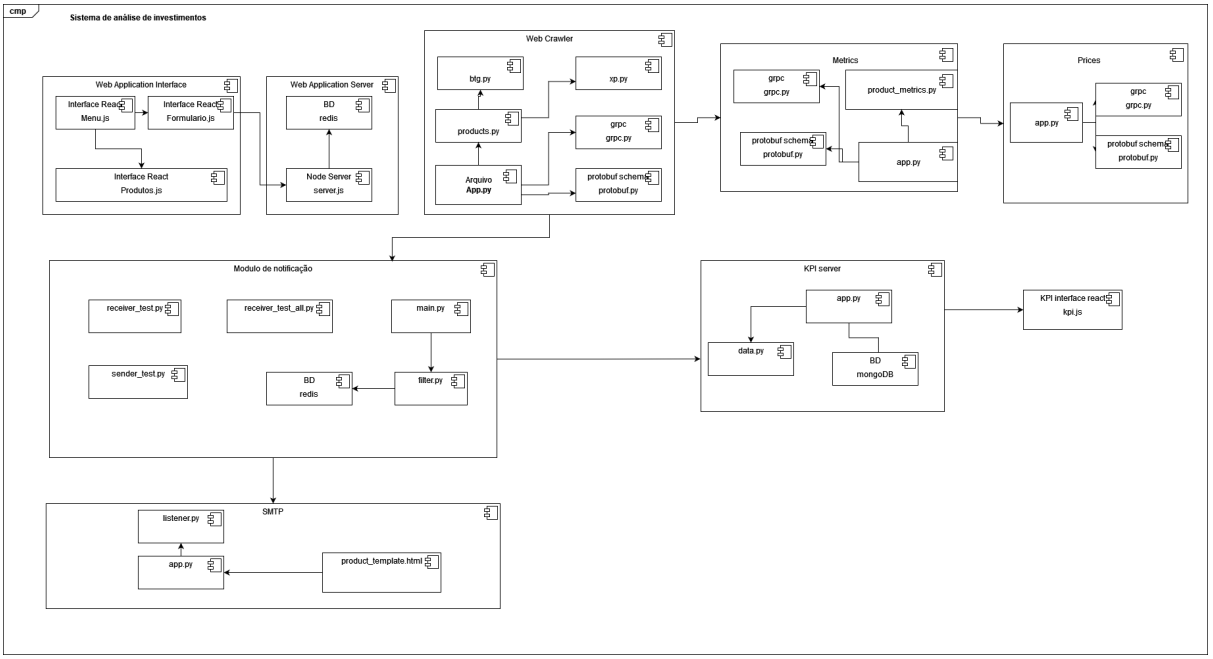
Abaixo temos a visualização das bibliotecas, tecnologias e bases de dados utilizados no projeto, todos esses aspectos da arquitetura e suas implementações foram descritos detalhadamente no item 3.2 Implementação.



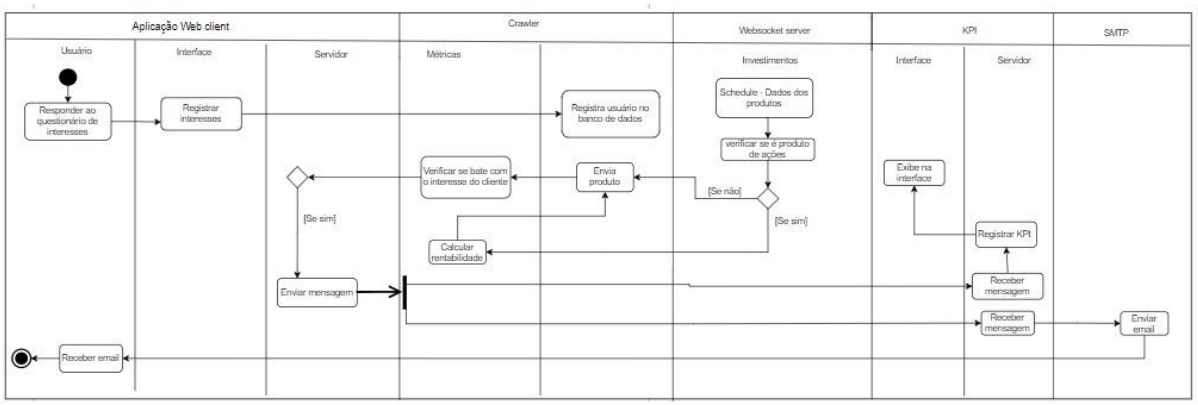
3.1.4 BPMN


Aplicação Web Client
Usuário
Solicitar notificação
Já está registrado?
Solicitar acompanhamento de dicas
sim
Preencher formulario do tipo de investimento
Visualizar notificação na web
Visualizar notificação no email
Interface
Exibir formulário
Fazer requisição para o servidor
Exibir formulario
Fazer requisição para o servidor
Servidor
Enviar Dados para o usuário
sim
Primeiro acesso?
Verificar parametro de tempo
Salvar os dados no Redis
Crawler
Buscar informações nas corretoras
É ação?
Criar o produto para a publicação
Enviar para o Redis streams
Métricas
Buscar cota diária
Calcular rentabilidade
Websocket server
Investimentos
Pegar os dados das ultimas 24 horas
Verificar o tipo de investimento que o usuário solicitou
Pegar dados do Redis
Validar tipos de investimentos
Criar notificação
Enviar dados para o usuário
KPI
Servidor
Enviar para interface
Salvar no banco
Métricas de publicação
Interface
Exibir dados ao usuário
SMTP
Criar um email
Enviar notificação para usuário por email

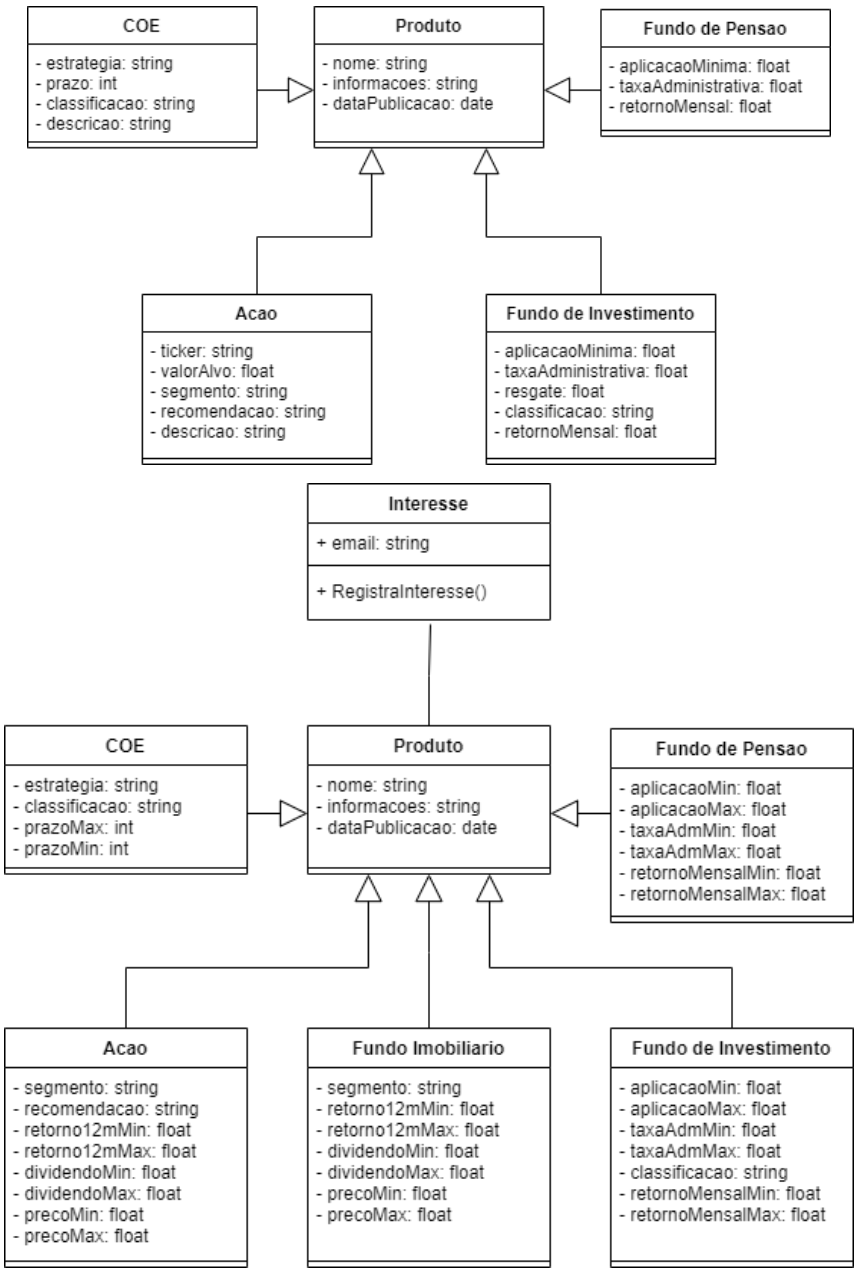
3.1.5 Diagrama de Componentes



3.1.6 Diagrama de Atividades



3.1.7 Diagrama de Classes



3.2 IMPLEMENTAÇÃO

3.2.1 Serviço *Crawler*

O serviço *Web Crawler* foi desenvolvido em Python. O *crawler* de produtos foi realizado nas corretoras BTG Pactual e XP Investimentos, conforme descrito anteriormente. Para esse processo foram utilizadas as bibliotecas *Beautiful Soup* e *Requests*. Com a realização do *crawl* os produtos são agrupados de acordo com suas categorias, e os dados são normalizados de acordo com a definição do diagrama de classes. No caso de produtos de renda variável (Fundos Imobiliários e Ações) o serviço faz a solicitação (como *client stub*) de cálculo de dados de mercado ao serviço de métricas, para isso, foi utilizado o gRPC e *Protocol Buffers*, o que aumentou a velocidade do processo, através da função definida abaixo:

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

Assim, também temos a definição da requisição e resposta, já que a definição dos dados nesse caso é estruturada:

```
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;
}
```

A partir da resposta retornada pelo serviço de métricas, os dados do produto serão enriquecidos com as devidas informações. Após todos os produtos serem coletados e normalizados, esse sistema publicará os dados no *Redis Streams*, na fila */products*. Para a demonstração do serviço foi desenvolvido um *endpoint* HTTP REST utilizando a biblioteca *Flask* para ativação do mesmo.

3.2.2 Serviço *Metrics*

O serviço de métricas é um *servicer* gRPC que é ativado pelo serviço *Web Crawler* conforme demonstrado acima. A partir da solicitação da função *GetMetrics*, o serviço de métricas fará uma solicitação ao serviço de preços utilizando também o gRPC, conforme a função e as estruturas abaixo:

```
service Info {
      rpc GetInfo(InfoRequest) returns (InfoResponse);

message InfoRequest {
     string ticker = 1;
     string start_date = 2;
     string end_date = 3;
}

message InfoResponse {
     string name = 1;
```

```
    repeated Value prices = 2;
    repeated Value volumes = 3;
    repeated Value dividends = 4;
}
```

Com os dados retornados o serviço utilizará a biblioteca *Pandas*, *Numpy* e *Scipy* para a realização dos cálculos de retorno, volume e *dividend yield*, retornando os dados de acordo com a estrutura *MetricsResponse* ao solicitante.

Código de definição de um servicer gRPC

```
        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()
```

3.2.3 Serviço *Prices*

O serviço de séries fornece um *servicer* gRPC para comunicação aos solicitantes. Nesse projeto ele é ativado pelo serviço de métricas de acordo com a estrutura de *Protocol Buffers* descrita acima. O *Protocol Buffers* foi importante nesse caso já que fornece um volume menor durante o tráfego dos dados, reduzindo a volumetria de dados trafegadas na rede, como neste caso buscamos séries históricas, o volume de dados tende a ser grande. Para o acesso aos preços foi utilizada a biblioteca *Yahoo Finance*, onde foram filtradas as séries de volume, preço de fechamento e dividendos distribuídos por um determinado ativo.

3.2.4 Serviço *Publisher*

O serviço *publisher* é o que possui maior responsabilidade no projeto, já que ele é responsável por distribuir todos os produtos que são publicados na fila *products* do *Redis Streams* (citada no serviço *crawler*). O serviço utiliza da biblioteca *Fast* API para fornecer a possibilidade de uma comunicação através de *websockets*, já que o tráfego de dados não pode ser feito via request response, dado o fato de que os produtos devem chegar em tempo real ao destino. O *Fast* API se encaixou bem nesse cenário por prover um serviço assíncrono em relação às requisições, assim o servidor de websocket roda em paralelo, assim como as requisições ao Redis, tornando o serviço totalmente escalável.

No projeto, temos o tratamento do websocket quando conectados aos endereços */products/?email={email},* */products/?email=all* e */products/?email=analytics (*onde *products* faz referência a fila do Redis que será distribuída pelo *websocket*). No caso do email ser all ou analytics, o publisher irá escanear o redis em busca de todos os registros de usuários e fará o filtro dos produtos para todos esses emails, associando os “n” produtos publicados na fila aos “n” e-mails registrados no Redis. Já no caso do email ser definido especificamente, teremos a busca apenas dos filtros do usuário que possui o e-mail definido, retornando só os produtos compatíveis com esse usuário.

Além disso, também é possível inserir como variável na conexão websocket, o atributo *last_ms={ms}*. Esse atributo fará com que o publisher busque nos dados armazenados na fila */products* os últimos “n” milissegundos de registros, distribuindo essa informação após a filtragem.

Setup do servidor websocket assíncrono

```
@app.websocket("/stream/{stream}")
async def proxy_stream(
    ws: WebSocket,
    stream: str,
    latest_id: str = None,
    past_ms: int = None,
    last_n: int = None,
```

```
    max_frequency: float = None,
    email: str = None,
):
    await ws.accept()
    redis = await aioredis.create_redis("redis://redis_db:6379")

    while True:
        to_read_id = latest_id
        if max_frequency is not None and latest_id is not None:
            ms_to_wait = 1000 / (max_frequency or 1)
            ts = int(latest_id.split("-")[0])
            to_read_id = f"{ts + max(0, round(ms_to_wait))}"

        messages: List[Message]
        try:
            messages = await read_from_stream(redis, stream, to_read_id, past_ms, last_n)
        except Exception as e:
            logger.info(f"read timed out for stream {stream}, {e}")
            return
```

Filtragem inicial assíncrona dos produtos

```
    async def filter_products(self):
        results = list()
        emails = list()
        cur = b'0'
        if self.emails in send_all:
            while cur:
                cur, keys = await self.redis.scan(cur, match='user:*')
                if keys:
                    emails.append(keys[0].decode().split(":")[-1])
        else:
            emails = self.emails
        logger.info(emails)

        for email in emails:
            user_filter = json.loads(await self.redis.get(f"user:{email}"))
            products = ast.literal_eval(self.messages[0]["payload"].get("products", []))
            for msg in products:
                _filter_type = _filter_map[msg["category"]]
                is_desired = user_filter.get(_filter_type, False)
                if isinstance(is_desired, bool):
                    continue
                result = self.filter_data(msg, user_filter[_filter_type])
                if result:
                    result["email"] = email
                    results.append(result)
```

```
        logger.info(f"----- {len(results)} FILTRADOS PARA O EMAIL {email} -----")
    return results
```

3.2.5 Serviço *Notifier Products*

O serviço notifier faz a distribuição dos emails aos usuários registrados. Conforme citado anteriormente, para receber os dados de todos os produtos filtrados associados a todos os emails registrados na base, o notifier se comunica com o serviço publisher via websocket, nesse caso ele escuta ativamente como cliente no path */products/?email=all* do *publisher*. Cada mensagem recebida nesse websocket ativa uma função que faz o agrupamento dos produtos de acordo com o email associado a esse produto. Assim, com esse agrupamento, é enviado um pacote de produtos ao usuário via email. Para o envio do email foi utilizado SMTP por meio da smtplib, assim como o Jinja para a composição do template que será encaminhado.

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
```

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)
```

3.2.5 Serviço *Collector*

Esse serviço possui dois componentes: o *listener* e a API. O *listener*, assim como o *notifier*, escuta ativamente ao publisher através do *websocket*, nesse caso, no path */products/?email=analytics*. As mensagens recebidas são agrupadas por classe de ativo e por produto individualmente, assim, é possível contar a quantidade de produtos distribuídos nesses dois cenários. Com essas contagens, é recuperado as informações de contagem de produtos e classes, que estão armazenadas no MongoDB, assim essas informações são somadas com as que acabaram de ser processadas nas mensagens, gerando os dados brutos para futura visualização.

Código de contagem dos produtos

```
def collect_frequency(messages):
    try:
        df = get_data_store("frequency")
        new_df = pd.DataFrame(messages)
        category_count=new_df["category"].value_counts().rename_axis("category").res
        et_index(name="count")
        category_count["publication_date"] = messages[0]["publication_date"]
        merged =pd.concat([df, category_count]).groupby(['publication_date',
        'category']).sum().reset_index()
        logger.info(merged)
        save_data_store(merged, "frequency", "category")
    except Exception as e:
        print(e)
        logger.error(e)

        def collect_top(messages):
            try:
                df = get_data_store("top10")
                new_df = pd.DataFrame(messages)
                assets_count=new_df["name"].value_counts().rename_axis("name"
                ).reset_index(name="count")
```

```
        merged = pd.concat([df, assets_count]).groupby(['name']) .sum().reset_index()
          save_data_store(merged, "top10", "name")
        except Exception as e:
          print(e)
          logger.error(e)
```

Já a api fará a disponibilização dos dados já processados ao front-end, através de uma API HTTP REST por meio da biblioteca Flask. Assim, quando solicitado (através das rodas */analytics/top10* e */analytics/frequency*), é feita uma consulta no MongoDB, retornando os dados na forma de JSON.

Exemplo de disponibilização dos dados via API

```
@app.route("/analytics/frequency")
@cross_origin()
def products_frequency():
  graphs["c"].inc()
  start = time.time()
  try:
    data = get_data_store("frequency").sort_values("publication_date", ascending=True)
    logger.info(data)
    products = data.to_dict(orient="records")
    response = jsonify(products)
    graphs["h"].observe(time.time() - start)
    return response
  except Exception as e:
    graphs["e"].inc()
    logger.error(e)
    return str(e)
```

3.2.6 Serviço *Notification Profile*

O serviço *Notification Profile*, é um servidor que recebe os dados enviados pelo usuário, através de um formulário na *web*, por requisição REST. Ele registra o interesse do usuário, de acordo com os investimentos escolhidos. Esse servidor foi desenvolvido com NodeJS e Express (Linguagem Javascript). Ao receber os dados na rota */registration,* por meio de uma requisição POST, é realizado verificações para remover possíveis campos vazios. Depois dessa filtragem, os dados serão salvos no Redis, utilizando o email como chave e assim confirmando o registro.

Depois que o registro foi feito, o servidor espera o envio do email na rota */recommendation*, por meio de uma requisição POST. Ao receber o email do usuário registrado, é feito conexão com o servidor de *websocket* que vai permitir que o usuário possa receber dados das últimas 24 horas, caso ele não tenha visualizado

as notificações anteriormente. Os dados serão enviados por meio de resposta no formato JSON.

Conexão com servidor websocket, recebimento e envio dos dados

```
routes.post('/recommendation', async (req, res) => {
  const WebSocket = require('ws');
  try {
    // Socket to emit to publisher server
    if (req.body.email) {
      const url =

`ws://localhost:8001/stream/products?email=${req.body.email}&past_ms=86400000`
      const connection = new WebSocket(url)
      connection.onmessage = e => {
        res.status(201).json({message: JSON.parse(e.data)})
      }
      setTimeout(() => {
        connection.close();
      }, 6000)
    } else {
      res.status(400).json({ message: 'Problem in the email' });
    }
  } catch (error) {
    console.error(error)
  }
})
```

3.2.7 Serviço *Front Notification*

O serviço Front Notification, é um *client* onde o usuário consegue registrar seus interesses de investimentos através de um formulário na web, e ao registrar as informações, elas são enviadas para o servidor via requisição REST, para isso foi utilizado um *fetch* padrão do Javascript com o método POST.

Assim que registrado as informações, o usuário poderá acessar a aba de acompanhar dicas, onde lá deverá preencher o campo de e-mail para capturar as dicas associadas ao e-mail. Depois de preenchido, o *client* enviará essa informação via requisição POST para o servidor, e obterá como resposta um JSON com as dicas de investimentos das últimas 24 horas associadas ao e-mail.

Logo que enviado o e-mail para o servidor também será aberta uma conexão via *websocket* (nativo do Javascript) com a FastAPI, em que irá receber as informações via JSON para poder atualizar as dicas de investimentos associada ao e-mail solicitado em tempo real na tela do usuário. As opções de investimentos serão mostradas em formato de *cards* na tela do usuário.

No *client* também será possível verificar as métricas (na aba de *KPI*) correspondentes ao total de cada tipo de investimento solicitado, e também será

mostrado os dez melhores investimentos até o momento. Para a criação das métricas da tela de *KPI* foi utilizado uma biblioteca *open source* chamada ApexCharts, com isso, foi possível fazer a criação de gráficos para a melhor visualização do usuário. Os dados dos gráficos foram obtidos do servidor em Flask no formato JSON via requisição GET.

O *client* foi criado com o framework ReactJS, utilizando nenhuma outra biblioteca além do ApexCharts.

4. DESCRIÇÃO TÉCNICA - DEVOPS

Essa etapa fará a descrição de recursos utilizados para gerenciamento e manutenção da aplicação, seja em nível de fluxo de dados na rede, execução do projeto e manutenção das bases de dados.

4.1.1 Utilização de *Containers Docker*

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 no container apenas, 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.

4.1.2 Utilização do *Docker-Compose*

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 mongoDB), 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
```

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.

Exemplo de banco de dados 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.

4.1.2 Utilização de Gerenciadores de Banco de Dados

Conforme descrito anteriormente foram utilizados os bancos de dados Redis e MongoDB. Para o gerenciamento dessas bases de dados foram utilizadas as imagens Docker dos serviços: *redis-commander* e *mongo-express*. Ambas foram instaladas e linkadas com os bancos através do docker-compose, podemos ver abaixo um exemplo dessa definição com o redis-commander.

Exemplo de 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
```

4.1.3 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.

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")
```

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:

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: