

# APLICAÇÃO DE GERÊNCIA DE FUNDOS IMOBILIÁRIOS UTILIZANDO CONCEITOS DE SISTEMAS DISTRIBUÍDOS

**Rafael José Camargo Bekhauser<sup>1</sup>**

<sup>1</sup>Graduando Bacharelado em Ciência da Computação - IFC Rio do Sul

rafaelcamargo.inf@gmail.com

**Abstract.** *This paper presents the development of a distributed application in order to manage real estate funds data, then are presented greater explanations about concepts of distributed systems. In addition, addressing the development of the application, using the Docker tool to manage database instances, MongoDB as distributed database software and the Python language for coding.*

**Key-words:** *Distributed systems; endpoints; Layered application.*

**Resumo.** *Este presente trabalho apresenta o desenvolvimento de uma aplicação distribuída com o intuito de gerir dados de Fundos Imobiliários, logo são apresentadas maiores explicações sobre conceitos de sistemas distribuídos. Ademais abordando o desenvolvimento da aplicação, utilizando a ferramenta Docker para gerir instâncias de bancos de dados, o MongoDB como software de banco de dados distribuído e a linguagem Python para codificação.*

**Palavras-chave:** *Sistemas distribuídos; FIIs; Aplicação em camadas.*

## 1. Introdução

Nos últimos anos, temos observado um crescimento significativo no número de investidores interessados em Fundos de Investimento Imobiliário (FIIs) na Bolsa de Valores brasileira (B3). Os FIIs são uma alternativa atrativa para investidores que buscam diversificação em seu portfólio, acesso ao mercado imobiliário e distribuição de rendimentos.

Daniel Chinzarian (2022), analista de fundos de investimentos da corretora de valores XP, nos anos de 2019 a 2022, levanta que segundo dados da B3 houve um aumento de 310%, 182%, 132% e 127%, respectivamente no número de investidores em FIIs.

Com o crescente número de investidores em FIIs na B3, surge a necessidade de desenvolver soluções de gerenciamento que atendam às demandas desses investidores.

Ademais, à medida que o mercado se expande, surgem desafios relacionados à distribuição geográfica dos investidores, necessidade de acesso em tempo real às informações e segurança das transações.

Diante desse contexto, o desenvolvimento de uma aplicação de gerenciamento de FIIs que utilize conceitos de sistemas distribuídos se torna relevante, visando proporcionar uma experiência eficiente, confiável e segura para os investidores, ao mesmo tempo em que lida com a crescente demanda desse mercado em constante expansão.

Portanto, neste presente trabalho abordaremos em um primeiro momento maiores explicações sobre sistemas distribuídos e suas técnicas, e em um segundo momento o desenvolvimento da aplicação.

## **2. Objetivos**

### **2.1. Objetivo Geral**

O objetivo geral deste trabalho é desenvolver uma aplicação simples de gerenciamento de fundos imobiliários que utilize conceitos de sistemas distribuídos para proporcionar escalabilidade, confiabilidade e distribuição.

### **2.2. Objetivos específicos**

1. Realizar um levantamento das necessidades e requisitos inerentes a aplicação.
2. Projetar uma arquitetura distribuída para a aplicação, levando em consideração a escalabilidade, a redundância e a tolerância a falhas.
3. Implementar os componentes da aplicação utilizando tecnologias apropriadas para sistemas distribuídos.
4. Realizar testes e avaliações na aplicação.

## **3. Levantamento de requisitos**

O levantamento de requisitos é uma etapa fundamental no processo de desenvolvimento de software, pois tem como objetivo compreender e documentar as necessidades e as funcionalidades desejadas para o sistema a ser desenvolvido.

Levamos em consideração o contexto, de uma aplicação de gerenciamento de Fundos de Investimento Imobiliário utilizando conceitos de sistemas distribuídos, onde devemos persistir os dados em um banco de dados distribuídos, com a possibilidade de integrar a leitura de dados por meio de um arquivo formato TXT,

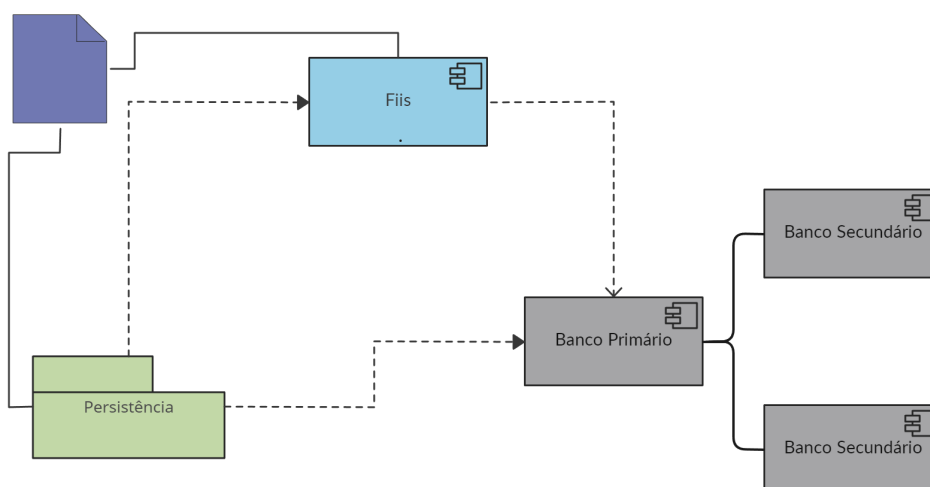
Portanto, como forma de melhor demonstrar visualmente as necessidades do projeto utilizamos o diagrama de componentes.

### **3.1. Diagrama de componentes**

O diagrama de componentes auxilia na identificação das dependências entre os diferentes componentes e módulos, e na definição da arquitetura de software. Sendo especialmente aplicado em sistemas distribuídos, onde os componentes podem estar espalhados por diferentes servidores ou até mesmo em diferentes empresas.

Conforme contexto supracitado anteriormente definimos um diagrama de componentes que pode ser verificado na imagem 1. Onde temos a presença do

componente principal os 'FIIs' que realizam persistência em um banco de primário, que por sua vez replica aos secundário.



**Imagem 1 – Diagrama de componentes.**

## 4. Sistemas distribuídos

Sistemas distribuídos são sistemas de computação compostos por vários componentes ou nós interconectados, localizados em diferentes máquinas físicas ou lógicas, que trabalham em conjunto para realizar uma tarefa ou fornecer um serviço. Esses componentes podem incluir computadores, servidores, dispositivos móveis, sensores, entre outros (COULOURIS, DOLLIMORE E KINDBERG, 2001).

A principal característica dos sistemas distribuídos é a comunicação e coordenação entre os diferentes nós, que ocorre por meio de troca de mensagens ou compartilhamento de recursos. Essa comunicação permite que os nós cooperem, troquem informações, coordenem atividades e realizem tarefas em conjunto, formando um sistema único e integrado.

### 4.1. Arquitetura em camadas

A arquitetura em camadas é um modelo de desenvolvimento de software que divide a aplicação em diferentes camadas, cada uma responsável por um conjunto específico de tarefas. É comumente utilizado em projetos de grande porte, onde a complexidade do sistema exige uma organização mais estruturada e modularizada.

Essa abordagem oferece benefícios como a possibilidade de isolamento de funcionalidades, facilitando a manutenção e evolução do sistema, além de garantir a escalabilidade do sistema, sem afetar as outras camadas. Com isso, é possível ter um sistema mais robusto, flexível e adaptável às mudanças de demanda.

## **4.2. Banco de Dados Distribuído**

Definimos um Banco de Dados Distribuído (BDD) como sendo uma coleção de nós (termo utilizado para se referir a um dispositivo ou uma entidade que faz parte da rede distribuída), em qual cada nó é um sistema de banco de dados independente e autônomo, sendo capaz de armazenar e gerenciar seus próprios dados. Porém, os nós estão ligados e cooperam uns com os outros, portanto os nós trabalham juntos para fornecer um acesso unificado aos dados distribuídos (DATE, 2004).

Segundo Korth (1999), cada nó de um BDD mantém um grau de controle sobre os dados armazenados localmente, o que permite autonomia e tomada de decisões individuais sobre a gestão desses dados. Logo, se um nó vier a ter uma falha crítica, os demais nós terão capacidade de operar em modo normal.

## **4.3. Chamada de procedimento remota**

O RPC (Remote Procedure Call) é uma técnica de programação utilizada para permitir a execução de procedimentos em computadores remotos através de uma rede. Através de RPC, um programa pode chamar funções em um computador remoto como se essas funções estivessem sendo executadas localmente.

O uso de RPC permite que a programação distribuída seja realizada de forma mais simples e eficiente, possibilitando a criação de sistemas distribuídos mais complexos.

## **5. Tecnologias e ferramentas**

Para o desenvolvimento da aplicação prática do presente trabalho, foram escolhidas tecnologias e ferramentas voltadas para o contexto de sistemas distribuídos, visando os contextos supracitados anteriormente, logo, como linguagem de programação utilizamos o Python, como sistema de banco de dados distribuído utilizaremos o MongoDB, para hospedar e gerir as instâncias de banco de dados utilizamos o Docker.

### **5.1. Python**

Python é uma linguagem de programação criada pelo holandês Guido van Rossum em 1991. Ela é uma linguagem interpretada de alto nível, suporta múltiplos paradigmas de programação como imperativo, orientado a objetos e funcional, possui tipagem dinâmica e forte, ademais possui gerenciamento automático de memória.

Suas estruturas de alto nível, juntamente com sua tipagem dinâmica a tornam muito atrativa para o desenvolvimento de aplicações de grande porte. Outro ponto de destaque se diz respeito a ampla gama de módulos disponíveis desenvolvidos para a linguagem.

O Python foi escolhido para este presente projeto pelo motivo de ser considerado uma linguagem de desenvolvimento rápido de aplicações (RAD), pela sua simplicidade e facilidade e pelo seu suporte a diversas bibliotecas e frameworks, sendo esta uma linguagem adequada para o desenvolvimento de sistemas distribuídos.

## **5.2. Lumi**

Lumi é um nano framework desenvolvido para a linguagem de programação Python. Tem por objetivo implementar um conversor direto de funções em uma API REST, por meio da aplicação dos conceitos de RPC juntamente com as especificações da arquitetura REST.

O framework opera realizando um mapeamento da assinatura da função escrita em Python, em uma rota da API REST. Caso existam parâmetros instanciados na função, os mesmos são dados como requeridos no body da requisição enviada pelo cliente.

## **5.3. MongoDB**

O MongoDB é um banco de dados NoSQL amplamente adotado que cresceu em popularidade nos últimos anos, especialmente em aplicativos modernos e escaláveis. Ao contrário dos bancos de dados relacionais tradicionais que usam um modelo de tabela fixa, o MongoDB adota uma abordagem flexível baseada em documentos para armazenamento de dados.

No MongoDB, os dados são organizados em documentos BSON (Binary JSON), que são estruturas de dados semelhantes a JSON e permitem uma representação rica e flexível dos dados. Essa estrutura de documento facilita a modelagem de dados complexos e a evolução dos esquemas, tornando o MongoDB adequado para cenários em que os requisitos de dados estão em constante mudança (POLITOWSKI E MARAN, 2014).

## **5.4. Docker**

Docker é uma plataforma de virtualização leve e de código aberto que permite a criação, o empacotamento e a execução de aplicativos em ambientes isolados, conhecidos como contêineres. Com base em tecnologias de virtualização a nível de sistema operacional, o Docker fornece uma abordagem eficiente e portátil para a implantação de aplicativos e serviços.

Os contêineres do Docker encapsulam todos os componentes necessários para um experimento, incluindo o sistema operacional, bibliotecas, dependências e o próprio aplicativo em um pacote único, que pode ser facilmente replicado e utilizado para a criação de outros containers (GOMES E SOUZA, 2015).

## 6. Aplicação prática

Na presente seção serão apresentados os detalhes referentes a implantação da aplicação. Logo, será configurado o ambiente de banco de dados distribuídos, seguidamente será colocado o desenvolvimento do código fonte, tendo em vista a aplicação da técnica de arquitetura em camadas, juntamente com o modelo de comunicação RPC.

### 6.1. Modelagem inicial

Com base no contexto apresentado para o desenvolvimento, podemos usar o diagrama de classes para representar as classes, suas necessidades e relacionamentos. Para reduzir o acoplamento e aplicar adequadamente os conceitos da arquitetura em camadas, é importante definir a necessidade das classes BO (Business Object), DTO (Data Transfer Object) e DAO (Data Access Object). Ademais, devido à aplicação do RPC, é necessário ter uma classe para controlar as requisições, bem como um servidor.

Portanto definimos no diagrama de classe, as classes FiiBO, FiiDAO, FiiDTO e Server, conforme imagem 2.

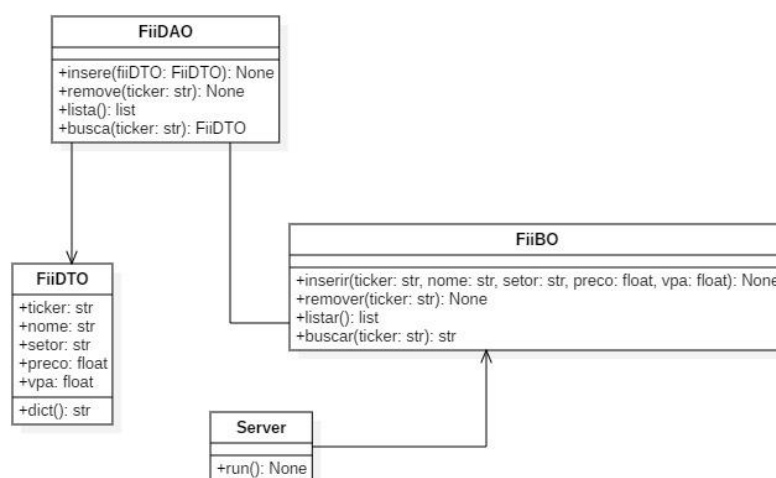


Imagem 2 – Diagrama de classes.

### 6.2. Criação do container do MongoDB

Para realizar a criação e subida do container no Docker utilizaremos o arquivo 'docker-compose.yml', este qual é um arquivo de configuração que contém a descrição dos serviços, redes e volumes necessários para executar um aplicativo Docker Compose. Ele usa uma sintaxe YAML (YAML Ain't Markup Language) para definir os componentes do aplicativo.

Definimos no arquivo 'docker-compose.yml' a imagem do MongoDB em três instâncias, cada uma em uma porta de serviço separada, e todas com a configuração inicial de replicação de dados (definido pelo comando '--replSet' e o pelo nome da replicação 'rs01'), conforme imagem 3.

```

1  version: "3.8"
2
3  services:
4    mongo1:
5      image: mongo:5
6      container_name: mongo1
7      command: ["--replSet", "rs01", "--bind_ip_all", "--port", "30001"]
8      volumes:
9        - ./data/mongo-1:/data/db
10     ports:
11       - 30001:30001
12
13    mongo2:
14      image: mongo:5
15      container_name: mongo2
16      command: ["--replSet", "rs01", "--bind_ip_all", "--port", "30002"]
17      volumes:
18        - ./data/mongo-2:/data/db
19     ports:
20       - 30002:30002
21
22    mongo3:
23      image: mongo:5
24      container_name: mongo3
25      command: ["--replSet", "rs01", "--bind_ip_all", "--port", "30003"]
26      volumes:
27        - ./data/mongo-3:/data/db
28     ports:
29       - 30003:30003

```

**Imagem 3 - Arquivo de configuração das instâncias.**

Para realizar a criação dos containers, devemos acessar a pasta ou local de onde salvamos o arquivo 'docker-compose.yml', e executar o seguinte comando no terminal 'docker-compose up -d' (como na imagem 4), por meio deste o Docker irá realizar o download das imagens do MongoDB e irá configurar três instâncias do banco de dados conforme definimos no arquivo de configuração.

```

PS C:\dev\IFC - BCC\Sistemas Distribuidos\BD_distribuido> docker-compose up -d
[+] Running 4/4
✓ Network bd_distribuido_default      Created
✓ Container mongo3                    Started
✓ Container mongo1                     Started
✓ Container mongo2                     Started
PS C:\dev\IFC - BCC\Sistemas Distribuidos\BD_distribuido>

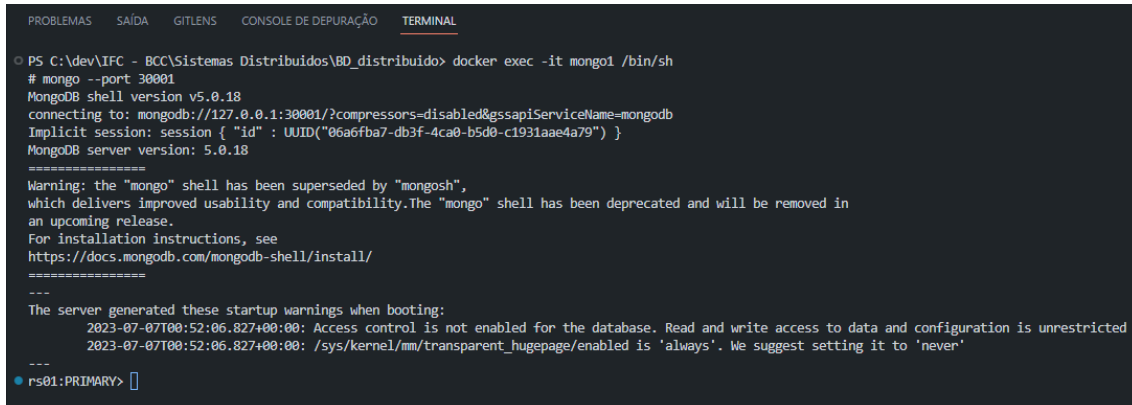
```

**Imagem 4 - Comando de inicialização do container.**

### 6.3. Configuração da replicação

Para de fato iniciarmos a replicação entre as três instâncias é necessário definir um banco principal, que será o transacional, e dois secundários que irão operar fazendo

a réplica dos dados. Portanto, devemos acessar uma das instâncias e realizar a definição de papéis de cada um, para isso utilizamos o comando (no terminal) ‘docker exec -it mongo1 /bin/sh’ que irá acessar o console do Docker no container informado, no caso o 1. Após inserimos o comando que nos permitirá acessar o console do MongoDB, ‘mongo --port 30001’. Como podemos ver na imagem 5. Se tudo ocorrer bem, irá aparecer o nome da definição da réplica, no caso ‘rs01’.



```

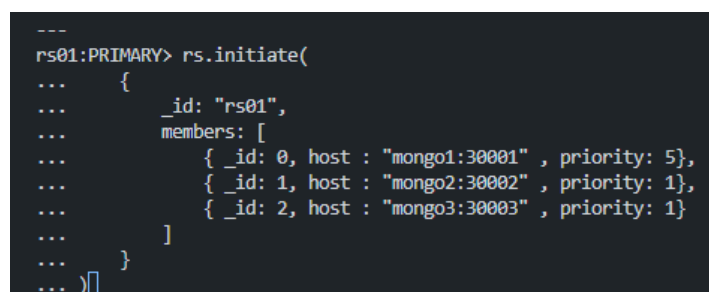
PS C:\dev\IFC - BCC\Sistemas Distribuidos\BD_distribuido> docker exec -it mongo1 /bin/sh
# mongo --port 30001
MongoDB shell version v5.0.18
connecting to: mongodb://127.0.0.1:30001/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("06a6fba7-db3f-4ca0-b5d0-c1931aae4a79") }
MongoDB server version: 5.0.18

=====
Warning: the "mongo" shell has been superseded by "mongosh",
which delivers improved usability and compatibility. The "mongo" shell has been deprecated and will be removed in
an upcoming release.
For installation instructions, see
https://docs.mongodb.com/mongodb-shell/install/
=====
---
The server generated these startup warnings when booting:
  2023-07-07T00:52:06.827+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
  2023-07-07T00:52:06.827+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
---
rs01:PRIMARY>

```

**Imagem 5 - Acesso ao console do banco de dados.**

Para definirmos quem será o banco primário e quem será secundário, utilizaremos a propriedade de prioridade (valor numérico, quanto mais baixo mais alta é a prioridade), logo quem tiver a maior prioridade será o primário. Logo, executaremos um script no console do MongoDB, que irá definir como vai ocorrer a replicação nas três instâncias, conforme podemos verificar na imagem 6.



```

---
rs01:PRIMARY> rs.initiate(
...   {
...     _id: "rs01",
...     members: [
...       { _id: 0, host: "mongo1:30001", priority: 5},
...       { _id: 1, host: "mongo2:30002", priority: 1},
...       { _id: 2, host: "mongo3:30003", priority: 1}
...     ]
...   }
... )

```

**Imagem 6 - Comando de inicialização da replicação.**

Após executado o script, será retornada uma mensagem informando o status da execução, caso tudo ocorra como esperado, aparecerá uma mensagem contendo “ok”: 0’.

#### 6.4. Definição das classes principais



Conforme o diagrama de classe apresentado anteriormente a aplicação é dividida em três camadas sendo DTO (Data Transfer Object), DAO (Data Access Object) e BO (Business Object).

A classe de objeto de transferência de dados (DTO) é um objeto que possui os dados necessários a serem transmitidos entre as demais classes. De acordo com Martin Fowler (2002), "DTOs são objetos simples que carregam dados entre processos remotos. Eles não devem conter lógica de negócios ou outras informações que pertençam a outro domínio." Conforme imagem 7, construímos um objeto em Python que possui como base um construtor e os atributos necessários de acordo com o cenário previamente descrito.

```

1  class FiiDTO:
2      def __init__(self, ticker: str, nome: str, setor: str, preco: float, vpa: float):
3          self._ticker = ticker
4          self._nome = nome
5          self._setor = setor
6          self._preco = preco
7          self._vpa = vpa
8
9      def __str__(self):
10         return f"Fii [" \
11             f"Ticker: {self._ticker} | " \
12             f"Nome: {self._nome} | " \
13             f"Setor: {self._setor} | " \
14             f"Preço: {self._preco} | " \
15             f"VPA: {self._vpa} | "
16
17     def dict(self):
18         return {
19             "ticker": self._ticker,
20             "nome": self._nome,
21             "setor": self._setor,
22             "preco": self._preco,
23             "vpa": self._vpa
24         }

```

**Imagem 7 - Classe DTO.**

A classe do objeto de acesso a dados (DAO), segundo Martin Fowler (2002), "A principal função de um objeto DAO é fornecer uma interface para uma camada de serviço de negócios com operações que exigem acesso a dados.". Outrossim tem por responsabilidade abstrair os detalhes de acesso ao banco de dados, fornecendo métodos que permitam realizar operações de inserção, busca e exclusão de informações do banco de dados.

Para tal, aplicamos o modelo de persistência no banco de dados MongoDB. Logo, definimos uma classe com métodos de inserção, busca e exclusão, segundo exemplo abaixo na imagem 8.

```

1  from DTO.fiiDTO import FiiDTO
2
3  class FiiDAO:
4      def __init__(self, collection):
5          # onde sera armazenado os dados
6          self.collection = collection
7
8          # Metodos para inserir, remover, listar e buscar fiis
9      def insere(self, fiiDTO: FiiDTO) -> None:
10         try:
11             print(fiiDTO.dict())
12             self.collection.insert_one(fiiDTO.dict())
13         except Exception as e:
14             print(e)

```

Imagem 8 - Classe DAO.

Conforme podemos observar, o método 'insere', recebe como parâmetro a própria classe juntamente com o objeto DTO a ser persistido. Sobre o objeto DTO é aplicado um método próprio que realiza a exportação do objeto no tipo dict com os atributos organizados em formato JSON.

BO, ou objeto de negócio, pode ser definido como um objeto que implementa a lógica de negócio em um sistema, sendo responsável por realizar validações, cálculos e demais regras de negócio (FOWLER, 2002).

Em concordância com o escopo da aplicação, podemos definir a classe BO como na imagem 9, onde temos uma chamada da instância da classe DAO, bem como os métodos que aplicam as regras de negócio (inserir, deletar e buscar). O método 'inserir', por exemplo, têm definido os parâmetros necessários para a realização da determinada tarefa, após é instanciado o objeto de transferência e por fim chamamos o método próprio da classe DAO que realiza a tarefa de persistência.

```

1  from DAO.fiiDAO import FiiDAO
2  from DAO.conn import Connection
3  from DTO.fiiDTO import FiiDTO
4
5  # Instancia do DAO
6  fiiDAO = FiiDAO(Connection().collection)
7
8  # Metodos para inserir, remover, listar e buscar fiis, utilizando o DAO
9
10 def inserir(ticker: str, nome: str, setor: str, preco: float, vpa: float):
11     fiiDTO = FiiDTO(ticker, nome, setor, preco, vpa)
12     fiiDAO.insere(fiiDTO)
13
14 def remover(ticker: str):
15     fiiDAO.remove(ticker)
16
17 def listar() -> list:
18     return fiiDAO.lista()
19
20 def buscar(ticker: str) -> str:
21     return fiiDAO.busca(ticker)
22

```

Imagem 9 - Classe BO.

## 6.5. Server

Seguindo o diagrama de classe e os requisitos da aplicação RPC, se faz necessário o uso de um web server (em suma, responsável por processar as solicitações HTTP enviadas pelo cliente) para que haja a devida comunicação entre os meios.

O nano framework Lumi, possui em seu escopo um web server embutido. Como o propósito do presente projeto não envolve o detalhamento do desenvolvimento do web server, optamos por abstrair esses detalhes e focar no uso do framework Lumi para atender aos requisitos da aplicação.

Para implementar a aplicação RPC, definimos a classe Server e, em seu construtor, criamos uma instância do Lumi. Em seguida, chamamos o método 'register' que recebe como parâmetro obrigatório a assinatura do método que desejamos registrar no web server e, como opcional, a rota pela qual o cliente fará a solicitação. Para isso, passamos como parâmetro as assinaturas da classe BO, de acordo com a imagem 10, linhas 7 a 15.

Definimos o método 'run' que tem como parâmetro a própria classe, tal realiza a chamada do método 'runServer' que por sua vez executa a aplicação web server, no host e porta indicados (imagem 10, linhas 18 e 19).

```

1  from lumi import Lumi
2  import BO.fiiBO as fbo
3
4  class Server:
5
6      def __init__(self):
7          # Cria uma instancia do servidor
8          self.app = Lumi()
9
10         # Registra as funcoes do BO no servidor para que possam ser acessadas remotamente
11         self.app.register(fbo.inserir, route='/inserir')
12         self.app.register(fbo.remover, route='/remover')
13         self.app.register(fbo.listar, route='/listar')
14         self.app.register(fbo.buscar, route='/buscar')
15
16         # Inicia o servidor
17         def run(self):
18             self.app.runServer(host='localhost', port=8000)
19
20 if __name__ == '__main__':
21     app = Server()
22     app.run()

```

Imagem 10 - Classe Server.

## 6.6. Integração com txt

Conforme apontado no levantamento de requisitos da aplicação, desenvolvemos um integrador, tal qual receberá um arquivo no formato TXT contendo 1 ou mais FIIs que serão inseridos no banco de dados. Logo, para tal definimos o padrão dos dados tendo como separador '|' portanto teremos, 'ticker|nome|setor|preco|vpa', como podemos ver na imagem 11.

A presente integração é um recurso que visa proporcionar agilidade e eficiência em gravar lotes numerosos de registros.

```

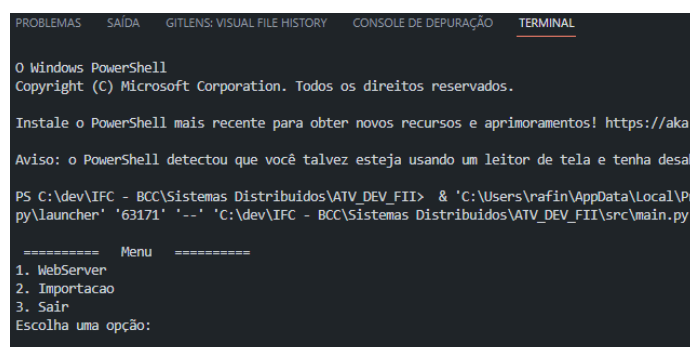
1  import B0.fiiB0 as fbo
2
3  class Import:
4
5      def __init__(self, filename):
6          self.filename = filename
7
8      def run(self):
9          with open(self.filename, 'r') as file:
10             for line in file:
11                 data = line.split('|')
12                 fbo.inserir(data[0], data[1], data[2], data[3], data[4])
13             print('Importação concluída com sucesso!')
14

```

**Imagem 11 - Classe de integração.**

## 7. Análise do fluxo de funcionamento e testes

Ao inicializar a aplicação o usuário deverá informar pelo terminal qual será o meio desejado, Server (para inicializar o serviço do RPC) ou Importação (para realizar a importação de um arquivo TXT pelo terminal). De acordo com a imagem 12.



```

PROBLEMAS  SAÍDA  GITLENS: VISUAL FILE HISTORY  CONSOLE DE DEPURACÃO  TERMINAL

O Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Instale o PowerShell mais recente para obter novos recursos e aprimoramentos! https://aka.ms/powershell

Aviso: o PowerShell detectou que você talvez esteja usando um leitor de tela e tenha desabilitado a verificação de segurança. Para obter mais informações, consulte https://aka.ms/powershell-security

PS C:\dev\IFC - BCC\Sistemas Distribuidos\ATV_DEV_FII> & 'C:\Users\rafin\AppData\Local\Programs\Python\Python39\python.exe' 'C:\dev\IFC - BCC\Sistemas Distribuidos\ATV_DEV_FII\src\main.py'

===== Menu =====
1. WebServer
2. Importacao
3. Sair
Escolha uma opção:

```

**Imagem 12 - Menu de interação via console.**

Caso o usuário opte pela primeira opção o webserver será inicializado e estará apto a realizar as chamadas de procedimentos anteriormente definidas. Como quesito de teste do webserver, faremos uma requisição HTTP sobre o mesmo, chamado a operação de inserção, logo passando os parâmetros necessários à solicitação. Conforme vemos na imagem 13, estamos realizando uma requisição ao endereço do servidor e passado dados de um FII como parâmetro, podemos conferir no terminal da imagem o campo 'status\_code: 200' que significa que a chamada ocorreu com êxito.

```

1  import requests
2  import json
3
4  # URL de acesso ao servidor
5  url = 'http://localhost:8000/inserir'
6
7  # Headers para indicar que o conteudo da requisicao eh um JSON
8  headers = {'content-type': 'application/json'}
9
10 # Payload com os dados da requisicao
11 payload = {
12     'ticker': 'HGLG11',
13     'nome': 'Cshg Logística',
14     'setor': 'Logística',
15     'preco': 200.00,
16     'vpa': 100.00
17 }
18
19 # Faz a requisicao POST para o servidor com os dados
20 response = requests.post(url, data=json.dumps(payload), headers=headers)
21
22 # Tenta converter a resposta para JSON e imprime
23 try:
24     print(response.json())
25 except Exception as e:
26     print(response)
27

```

PROBLEMAS SAÍDA GITLENS: VISUAL FILE HISTORY CONSOLE DE DEPURACÃO TERMINAL Code

[Running] python -u "c:\dev\IFC - BCC\Sistemas Distribuidos\ATV\_DEV\_FII\client.py"

{'exit\_code': 0, 'status\_code': 200, 'result': '', 'error': ''}

[Done] exited with code=0 in 0.45 seconds

Imagem 13 - Exemplo de requisição no webserver.

Para verificar se o dado foi devidamente inserido no banco de dados podemos realizar uma nova solicitação, mas chamando o procedimento ‘listar’, que trará todos os registros da coleção do banco de dados. Como vemos na imagem 14, tivemos o retorno no terminal do FII antes inserido.

```

1  import requests
2  import json
3
4  # URL de acesso ao servidor
5  url = 'http://localhost:8000/listar'
6
7  # Headers para indicar que o conteudo da requisicao eh um JSON
8  headers = {'content-type': 'application/json'}
9
10 # Payload com os dados da requisicao
11 payload = {
12 }
13
14
15 # Faz a requisicao POST para o servidor com os dados
16 response = requests.post(url, data=json.dumps(payload), headers=headers)
17
18 # Tenta converter a resposta para JSON e imprime
19 try:
20     print(response.json())
21 except Exception as e:
22     print(response)
23

```

PROBLEMAS SAÍDA GITLENS: VISUAL FILE HISTORY CONSOLE DE DEPURACÃO TERMINAL Code

[Done] exited with code=0 in 0.269 seconds

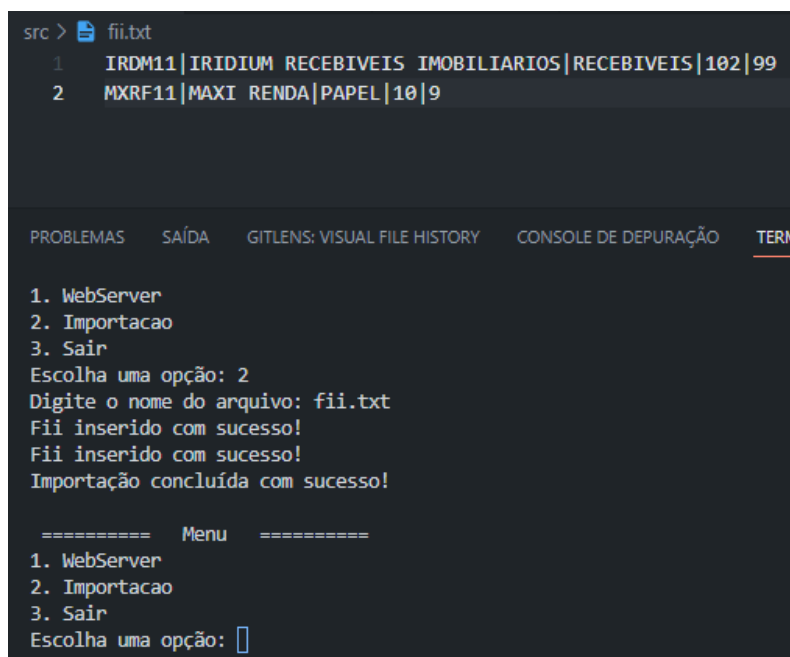
[Running] python -u "c:\dev\IFC - BCC\Sistemas Distribuidos\ATV\_DEV\_FII\clientrequest.py"

{'exit\_code': 0, 'status\_code': 200, 'result': ['Fii [Ticker: HGLG12 | Nome: Cshg Logística | Setor: Logística | Preço: 200.0 | VPA: 100.0 | ', 'error': '']}

[Done] exited with code=0 in 0.254 seconds

Imagem 14 - Exemplo de requisição no webserver.

Caso o usuário opte pela segunda opção, o terminal irá solicitar o nome do arquivo que se deseja importar. Como quesito de teste do definimos um arquivo TXT com dois FIIs no padrão previamente estabelecido, e passamos tal informação ao terminal. O qual deu continuidade na chamada do processo e realizou a inserção dos dados no banco de dados, como exemplificado na imagem 15.



```
src > fii.txt
1 IRDM11|IRIDIUM RECEBIVEIS IMOBILIARIOS|RECEBIVEIS|102|99
2 MXRF11|MAXI RENDA|PAPEL|10|9

PROBLEMAS  SAÍDA  GITLENS: VISUAL FILE HISTORY  CONSOLE DE DEPURAÇÃO  TERMINAL

1. WebServer
2. Importacao
3. Sair
Escolha uma opção: 2
Digite o nome do arquivo: fii.txt
Fii inserido com sucesso!
Fii inserido com sucesso!
Importação concluída com sucesso!

===== Menu =====
1. WebServer
2. Importacao
3. Sair
Escolha uma opção: 
```

Imagem 15 - Exemplo de requisição no webserver.

## 8. Considerações finais

Durante o processo de instalação e configuração das ferramentas supracitadas anteriormente, podemos perceber uma certa facilidade em realizar os processos. Sem dúvidas o uso do Docker simplifica o processo de instalação e configuração do MongoDB. Em vez de ter de precisarmos interagir com a instalação manual do MongoDB e suas dependências, podemos simplesmente executar um comando para baixar a imagem do MongoDB e iniciar um contêiner pronto para uso, economizando assim tempo e evitando possíveis conflitos de dependências.

Outrossim, o uso de técnicas de sistemas distribuídos como a arquitetura em camadas e o RPC proporcionam um melhor desempenho e escalabilidade para o MongoDB em ambientes de produção. A arquitetura em camadas permite distribuir a carga de trabalho entre diferentes servidores, aumentando a capacidade de processamento e reduzindo a sobrecarga em um único servidor. Além disso, o uso do Remote Procedure Call (RPC) facilita a comunicação entre os diferentes componentes do sistema distribuído, tornando mais eficiente a execução das operações no MongoDB.

A utilização de conceitos de sistemas distribuídos na gerência de fundos imobiliários é de extrema importância para a eficiência, escalabilidade e confiabilidade desse tipo de aplicação. A natureza complexa e dinâmica dos fundos imobiliários demanda uma abordagem distribuída, que permite lidar com grandes volumes de dados com um processamento intensivo.

Os aspectos presentes em uma aplicação distribuída, são cruciais em um ambiente financeiro complexo como o dos fundos imobiliários, onde a precisão e a confiabilidade são fundamentais para o sucesso e a reputação do negócio.

## Referências

BARRETO, José Victor Souza. **Fundos de investimento imobiliário no Brasil: as características que explicam o desempenho**. 2016. Tese de Doutorado.

DATE, C. J. **Introdução a sistemas de banco de dados**. 8 ed. Rio de Janeiro, Campus, 2004.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas distribuídos**. Madrid: Addison Wesley, 2001.

FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2002.

LUMI. Lumi Oficial, 2022. Disponível em <<https://github.com/Lumi-Official/lumi>>.

GOMES, Rafael; SOUZA, Rodrigo. **Docker-Infraestrutura como código, com autonomia e replicabilidade**. Superintendência de Tecnologia da Informação - Universidade Federal da Bahia (UFBA), Salvador, Ba, p. 1-4, 2015.

KORTH, H.F, SUDARSHAN S & SILBERSCHATZ, A. **Sistemas de banco de dados**. 3 ed., SP: Makron Books, 1999.

ÖZSU M.; VALDURIEZ P. **Principles of Distributed Database Systems**. Nova Jersey: PrenticeHall, 1999. 185p.

POLITOWSKI, Cristiano; MARAN, Vinicius. **Comparação de performance entre postgresql e mongodb**. Escola Regional de Banco de Dados. São Francisco do Sul, Brasil, 2014.

PYTHON. Python: The Python Language Reference. Disponível em: <<https://docs.python.org/3/reference/index.html>>.

VILHEGAS, Viviani. **UTILIZAÇÃO DE ARQUITETURA EM CAMADAS BASEADA NO MODEL VIEW CONTROLLER, EM APLICAÇÕES WEB**. ETIC-ENCONTRO DE INICIAÇÃO CIENTÍFICA-ISSN 21-76-8498, v. 7, n. 7, 2011.

VON AH, Bruno Eduardo. *Desenvolvimento para arquitetura distribuída utilizando Remote Procedure Call (RPC)*. 2013.