CONGRESSO CATARINENSE DE
CIÊNCIA DA COMPUTAÇÃO
2022. Ano 9, Vol. 9. ISSN: 2319-0655

RELATÓRIO DE DESENVOLVIMENTO APLICAÇÃO RPC

Rafael José Camargo Bekhauser[1]
[1]Graduando Bacharelado em Ciência da Computação - IFC Rio do Sul
rafaelcamargo.inf@gmail.com

Abstract. *This paper presents the development and application of the RPC technique, using as a base scenario a system for employee registration, where it should manage and maintain the basic data of an employee, thus allowing the inclusion of registration, name, e-mail, position and salary. As a strategy for developing the RPC technique, REST software architecture modeling will be adopted, applying the concepts of the layered modeling architecture.*
Key-words: *RPC; REST; Software architecture; Layered modeling.*

Resumo. *Este trabalho apresenta o desenvolvimento e aplicação da técnica RPC, utilizando como cenário base um sistema para cadastro de funcionário, onde o mesmo deverá gerir e manter os dados básicos de um funcionário, logo, permitindo a inclusão de matrícula, nome, e-mail, cargo e salário. Como estratégia de desenvolvimento da técnica RPC, será adotado a modelagem de arquitetura de software REST, aplicando os conceitos da arquitetura de modelagem em camadas.*
Palavras-chave: *RPC; REST; Arquitetura de software; Modelagem em camadas.*

1. Introdução

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.

O presente relatório técnico tem a proposta de desenvolvimento e aplicação da técnica RPC, utilizando como cenário base um sistema para cadastro de funcionário, onde o mesmo deverá gerir e manter os dados básicos de um funcionário, logo, permitindo a inclusão de matrícula, nome, e-mail, cargo e salário.

Como estratégia de desenvolvimento da técnica RPC, será adotado a modelagem de arquitetura de software REST, aplicando os conceitos da arquitetura de modelagem em camadas (apresentação, objeto de negócio e acesso aos dados).

2. Arquitetura REST

REST (Representational State Transfer) é um modelo de arquitetura para desenvolvimento de softwares distribuídos, sendo este baseado no protocolo de comunicação HTTP (HyperText Transfer Protocol). Em suma, o REST tem como proposta o acesso aos recursos de um determinado sistema, por meio da utilização de

URLs (Uniform Resource Locators), aplicando sobre estes operações com base nos métodos HTTP (GET, POST, DELETE, dentre outros).

Nesta abordagem o cliente realiza uma requisição ao servidor web, para acessar ou alterar determinado recurso através do envio de um pedido HTTP, por sua vez o servidor web, retorna ao cliente uma resposta com as informações condizentes ao resultado da requisição. As requisições e respostas aplicadas no REST, podem assumir diferentes formatos, como XML, JSON, HTML, dentre outros.

O conceito de stateless, que significa "sem estado" em inglês, é fundamental no estilo arquitetural REST, permitindo que cada solicitação feita ao servidor contenha todas as informações necessárias para seu processamento, sem depender de informações armazenadas anteriormente no servidor. Logo, o servidor pode atender a uma grande quantidade de solicitações de maneira eficiente e escalável, sem a necessidade de manter um estado de conexão com o cliente entre as solicitações.

Devido a estas características a arquitetura REST, sendo considerada como uma arquitetura com alta escalabilidade e flexibilidade, permitindo fácil adaptação às necessidades específicas da aplicação desenvolvida. Outrossim REST possui um baixo acoplamento permitindo que os componentes da aplicação sejam independentes entre si, proporcionando a modularização.

3. 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. Modelagem e representação

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 REST, é necessário ter uma classe para controlar as requisições, bem como um servidor.

Portanto definimos no diagrama de classe, as classes FuncionarioBO, FuncionarioDAO, FuncionarioDTO e Server, conforme imagem abaixo. De forma a padronizar os possíveis cargos atribuídos aos funcionários incluímos o enum Cargo

(enum, ou enumeração é um tipo de dado abstrato, cujos valores são atribuídos a exatamente um elemento de um conjunto finito de identificadores).

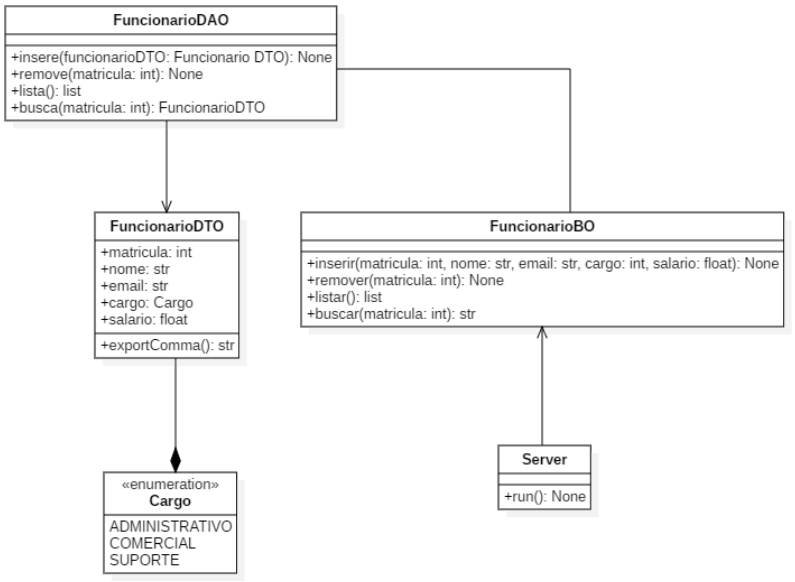

Imagem 1 – Diagrama de classe.

4.1. Representação do cliente

O desenvolvimento do cliente que irá realizar a solicitação ao servidor foi abstraído da modelagem anteriormente apresentada, pois o mesmo pode assumir diferentes modelos e meios de desenvolvimento e aplicação. Entretanto seguindo os padrões e convenções de boas práticas de desenvolvimento, se faz necessário a definição do stub.

Um stub é um componente do lado cliente, são fragmentos de algoritmos que tem por objetivo prover a abstração da chamada do método, isolando desta forma os detalhes referentes a comunicação através da rede. Tal qual contém as assinaturas dos métodos, os tipos de dados necessários e atributos.

Com base no diagrama de classe apresentado, abstraímos a classe FuncionarioBO, onde estão contidos os procedimentos que serão chamados remotamente, juntamente com os tipos de dados necessários e os retornos esperados de cada um respectivamente, conforme imagem 2.

```
1 def inserir(matricula: int, nome: str, email: str, cargo: int, salario: float)
-> None: ...
2 def remover(matricula: int) -> None: ...
3 def listar() -> list: ...
4 def buscar(matricula: int) -> str: ...
```

Imagem 2 – Abstração dos métodos de chamada remota.

5. Desenvolvimento

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, podemos interpretar visualmente conforme imagem 3. Caso existam parâmetros instanciados na função, os mesmos são dados como requeridos no body da requisição enviada pelo cliente.

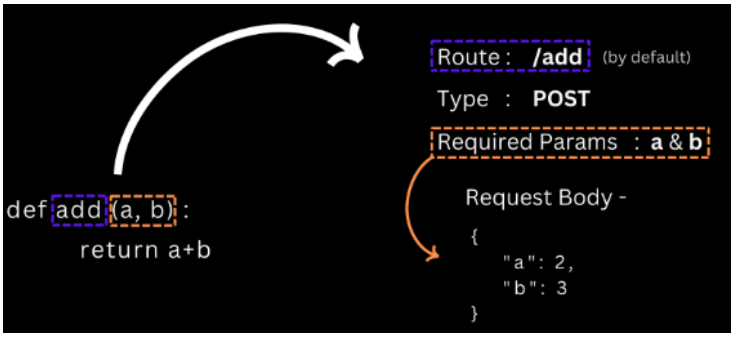

Imagem 3 – Exemplo de conversão do framework Lumi.[1]

[1] Fonte: Lumi Oficial, 2022. Disponível em <https://raw.githubusercontent.com/Tanmoy741127/cdn/main/lumi/function-api-map.png>.

5.3. Definição das classes

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 4, construímos um objeto em Python que possui como base um construtor e os atributos necessários de acordo com o cenário previamente descrito.

```
11  # Classe DTO para representar um funcionario
12  class FuncionarioDTO:
13      def __init__(self, matricula: int, nome: str, email: str, cargo: Cargo, salario: float):
14          self._matricula = matricula
15          self._nome = nome
16          self._email = email
17          self._cargo = cargo
18          self._salario = salario
19
```

Imagem 4 – Desenvolvimento da 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 um um arquivo formato TXT, com os dados separados por vírgula. Logo, definimos uma classe com métodos de inserção, busca e exclusão em um arquivo textual, segundo exemplo abaixo.

```python
class FuncionarioDAO:
    def __init__(self):
        # Nome do arquivo que armazena os dados dos funcionarios
        self.filename = 'funcionarios.txt'

    """
    Metodo para inserir um funcionario no arquivo

    :param funcionarioDTO: FuncionarioDTO
    :return: None
    """
    def insere(self, funcionarioDTO: FuncionarioDTO) -> None:
        try:
            with open(self.filename, 'a') as f:
                f.write(funcionarioDTO.exportComma() + '\n')
        except Exception as e:
            print(e)
```

Imagem 5 – Exemplo da 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 string com os atributos separados por vírgula.

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 6, 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.

```python
# Instancia do DAO
funcionarioDAO = FuncionarioDAO()

# Metodos para inserir, remover, listar e buscar funcionarios, utilizando o DAO


def inserir(matricula: int, nome: str, email: str, cargo: int, salario: float):
    funcionarioDTO = FuncionarioDTO(matricula, nome, email, Cargo(cargo), salario)
    funcionarioDAO.insere(funcionarioDTO)


def remover(matricula: int):
    funcionarioDAO.remove(matricula)


def listar() -> list:
    return funcionarioDAO.lista()


def buscar(matricula: int) -> str:
    return funcionarioDAO.busca(matricula).__repr__()
```

Imagem 6 – Classe BO.

5.4. Server

Seguindo o diagrama de classe e os requisitos da aplicação REST, 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 REST.

Para implementar a aplicação REST, 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 7, 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 7, linhas 18 e 19).

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

Imagem 7 – Classe Server.

5.4. Client

A construção do cliente pode-se dar de diferentes formas, visto que seu desenvolvimento não tem interferência direta sobre o servidor e demais classes internas,

cada solicitação realizada é independente e não compartilha informações de estado com outras solicitações.

Esse modelo proporciona uma independência para o cliente, permitindo que seja desenvolvido em diferentes linguagens de programação e plataformas, possuindo como obrigatoriedade a compatibilidade com o protocolo de comunicação HTTP. Se faz necessário destacar que, para a correta construção do cliente deve-se seguir as especificações e requisitos da aplicação REST.

Para tal construímos um cliente em Python de acordo com o stub anteriormente colocado. O cliente desenvolvido tem por objetivo acessar o recurso de 'inserir' do servidor de controle de cadastro de funcionários, logo, definimos como body da requisição os dados necessários, em seguida realizamos o requisição POST ao servidor, conforme imagem 8.

```python
import requests
import json

# URL de acesso ao servidor
url = 'http://localhost:8000/inserir'

# Headers para indicar que o conteudo da requisicao eh um JSON
headers = {'content-type': 'application/json'}

# Payload com os dados da requisicao
payload = {
    'matricula': 123,
    'nome': 'Rafael',
    'email': 'rafael@mail.com',
    'cargo': 2,
    'salario': 1000.0
}

# Faz a requisicao POST para o servidor com os dados
response = requests.post(url, data=json.dumps(payload), headers=headers)

# Tenta converter a resposta para JSON e imprime
try:
    print(response.json())
except Exception as e:
    print(response)
```

Imagem 8 – Exemplo de requisição da parte cliente.

6. Conclusão

No decorrer deste projeto, foi possível aplicar a técnica RPC utilizando a arquitetura REST, tal qual permitiu uma separação clara das camadas de apresentação, objeto de negócio e acesso aos dados. O uso do framework Lumi, aliado ao Python, permitiu a criação de um servidor que disponibiliza as funcionalidades para as chamadas remotas.

A aplicação da técnica RPC permitiu a independência entre o cliente e o servidor, que por sua vez possibilita o desenvolvimento de diferentes clientes em diferentes linguagens de programação e plataformas, desde que estes sigam as especificações e requisitos da aplicação REST.

Em suma, a modelagem de arquitetura de software REST, aliada ao uso da técnica RPC, se mostrou uma estratégia eficiente para o desenvolvimento deste projeto, permitindo a criação de um sistema de gerenciamento de dados de funcionários de forma organizada e independente.

Referências

FERREIRA, Willian Ottoni; DE OLIVEIRA KNOP, Igor. **Estruturação de Aplicações Distribuídas com a Arquitetura REST**. Caderno de Estudos em Sistemas de Informação, v. 3, n. 1, 2017.

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

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

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

SILVA, LUCAS RAFAEL DA. NECESSIDADE DE SE UTILIZAR BOAS PRÁTICAS ARQUITETURAIS E PADRÕES DE PROJETO NO DESENVOLVIMENTO DE WEB SERVICE BASEADO NA ARQUITETURA RESTFUL. 2016.

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

TEIXEIRA, Mário A. M. **Aplicações Distribuídas em Windows 95. Apoiadas por Ferramenta de Geração Automática de Stubs**. Dissertação de Mestrado, USP-ICMC, 1997.