CCF 355 - Sistemas Distribuídos e Paralelos

Trabalho prático - parte 5

Infinity Deck

Isabella Ramos (3474), Ricardo Spínola (3471)

5 de agosto de 2022

Conteúdo

1 Introdução

O trabalho prático parte 5 consiste na implementação do middleware Web Service no sistema já em desenvolvimento, Infinity Deck.

O desenvolvimento do trabalho utilizando conceito de serviços foi empregado com a arquitetura *REST*, possibilitando a criação de uma *API* para nosso sistema.

2 Instalação

2.1 Utilização - Aplicação

- **Linux** → Google Drive ou One Drive
- **Windows** → Google Drive ou One Drive

Para instalação em linux, o arquivo é disponibilizado no formato .deb

- **sudo dpkg -i infinity-deck_web.deb**

Para inicial a aplicação:

- **infinity-deck –no-sandbox**

2.2 Utilização - Servidor

Para o servidor, mantém-se os mesmos procedimentos. Os arquivos são encontrados dentro da pasta Server.

- **python3 init.py** *Inicialização do banco (reset)*
- **python3 server.py** *Inicialização do servidor em 127.0.0.1 - 23123*
- **python3 server.py *host port*** *Inicialização custom*

3 Implementação

A visão de serviços foi construída utilizando o *Flask* para o servidor, e a biblioteca *request* no lado do cliente.

A biblioteca Flask utilizada no servidor, o mesmo implementado em Python, fornece mecanismo de criação de serviços.

Criando uma classe WebService, disponibilizamos API's para realização de operações no banco de dados. Anteriormente utilizávamos apenas um método, de modo que, todas as mensagens recebidas eram direcionadas e tratadas por ele.

Entretanto, ao criarmos serviços, operações no banco deveriam ser disponibilizadas como API, de forma que cada operação deva ter um método equivalente.

3.1 Servidor

3.1.1 Versão Socket

```
msg = cliente.recv(1024).decode()
resp = self.controller.executeOperation(msg)
if(resp):
    cliente.sendall(resp.encode())
```

Figura 1: Método unificado

```
def executeOperation(self,msg):
    banco = BD()

    api, body = msg.split('$')[1:3]

    if(api == 'lu'):
        sts = banco.logarUser(body['login'],body['password'])
        resp = {'status':True if len(sts) else False, 'dados':sts}
```

Figura 2: Distinção de métodos pela classe *controller*

3.1.2 Versão WebService

```
class WebService:
    def __init__(self):
        self.app = Flask(__name__)
        self.controller = Controller()

        @self.app.route('/logarUser', methods = ['POST'])
        def logarUser():
            try:
                data = request.json
            except:
                return {"status":False,"mensage":"Dados (JSON) ausentes, Method Post"}
            resp = self.controller.logarUser(data)
            return resp
```

Figura 3: Classe WebService

```
class Controller:


    def logarUser(self,body):
        banco = BD()
        error = reportError(body,['login','password'])
        if(error):
            return error

        sts = banco.logarUser(body['login'],body['password'])
        resp = {
            'status':True if sts else False,
            'dados':sts if sts else "Dados incorretos, não foi possível logar"
        }

        banco.save()
        banco.close()
        return JSON.string(resp)
```

Figura 4: Classe Controller

3.1.3 Tratamento de erros

Métodos *POST* em *REST* necessitam de parâmetros passados no *body*. Adicionamos tratamentos para estes parâmetros como, ausência do *body*, ou mesmo de alguns campos.

```
def reportError(body,data):
    msg = "Campos obrigatórios - "
    err = False
    for prop in data:
        try:
            temp = body[prop]
        except:
            err = True
            msg = msg + "{} ".format(prop)
    return False if not err else JSON.string({'status':False,'mensagem':msg})
```

Figura 5: Verificação de erros

```
{
  "status": false,
  "mensagem": "Campos obrigatórios - login password "
}
```

Figura 6: Erro método de logar

3.2 Aplicação

A utilização do middleware WebService na aplicação foi feita com o auxílio da biblioteca *request*. A mesma possibilita fazer requisições *HTTP/REST*

```
function requestServer(method,api,body){
  if(method=='post')
    return new Promise(resolve=>{
        request.post(`http://${HOST}:${PORT}/${api}`,{json:JSON.parse(body)},(err,resp,body)=>{
          resolve(body)
        })
  })
```

Figura 7: Requisição feita pela aplicação

3.3 API

Como mencionado, foram criados métodos separados como *endpoint's*. Seus métodos divergem de modo que, para operações que não alterarem o banco de dados , as mesmas serão tratadas como *GET*, do contrário, serão *POST*

```
@self.app.route('/')
def index():···
@self.app.route('/logarUser', methods = ['POST'])
def logarUser():···
@self.app.route('/createUser', methods = ['POST'])
def createUser():···
@self.app.route('/getCartas')
def getCartas():···
@self.app.route('/getUsers')
def getUsers():···
@self.app.route('/getInventory', methods=['POST'])
def getInventory():···
@self.app.route('/addCarta', methods=['POST'])
def addCarta():···
@self.app.route('/createProposta', methods=['POST'])
def createProposta():···
@self.app.route('/getProposta', methods=['POST'])
def getProposta():···
@self.app.route('/aceitaProposta', methods=['POST'])
def aceitaProposta():···
@self.app.route('/rejeitaProposta', methods=['POST'])
def rejeitaProposta():···
```

Figura 8: Métodos criados

Métodos *GET* podem ser feitos diretamente pelo navegador.

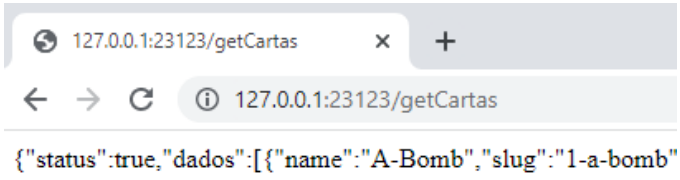

Figura 9: Teste navegador

3.3.1 Restauração de funcionalidades

No módulo anterior, havíamos removido a possibilidade de se alterar a porta ou ip da aplicação. Essa remoção foi devido há complicações que o gRPC causara.

Com a utilização de serviços, tanto a aplicação quanto o servidor podem ter *ip's* ou portas alteradas dinamicamente.

- **server**: ***python3 server.py host port***

Para a aplicação que nativamente iniciara em *127.0.0.1:23123* terão duas áreas para alteração.***Configurações*** ou ***tela de erro****(apresentada quando servidor está desconectado).*

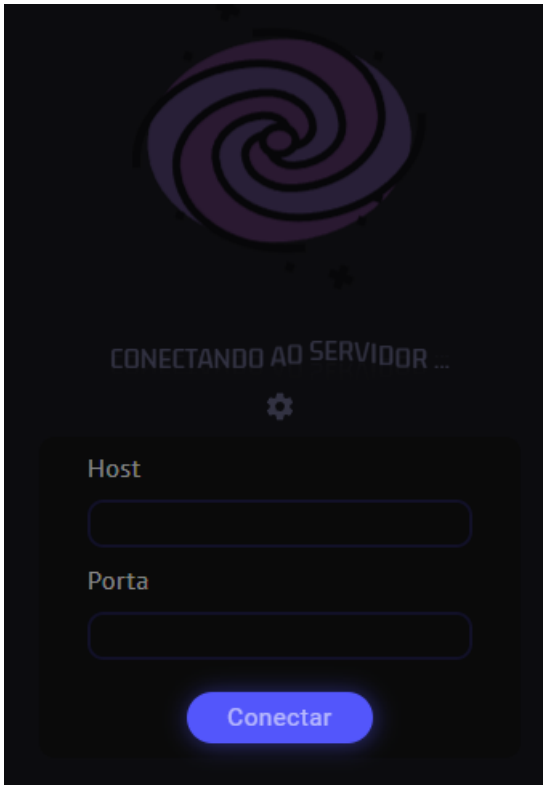

Figura 10: Alteração de destino na aplicação

4 Conclusão

De modo geral, todas implementações foram realizadas com êxito. Entretanto, cada fase teve suas particularidades quanto aos aspectos de implementação.

Socket's fora uma implementação custosa quanto ao tempo dedicado. Necessitava de tratamento de mensagens e concorrência -com utilização *threads*, o que o tornara na fase de desenvolvimento um ponto de desvantagem. A vantagem encontrada em Socket's se dá pelo acesso a baixo nível de implementação, o que nos proporciona facilidade para utilização em diferentes ambientes.

O desenvolvimento com o middleware gRPC foi satisfatório quanto ao aprendizado na utilização de objetos remotos. Entretanto, a baixa disponibilidade de conteúdo na web sobre a utilização do mesmo em diferentes ambientes gerou empecilhos quanto ao desenvolvimento. Contrapondo as dificuldades encontradas na sua implementação, o gRPC foi fácil de ser utilizado para casos simples.

O middleware de WebService (REST) por ser mais popular dentre as implementações, detém maior facilidade quanto a implementação em diferentes ambientes. A utilização do mesmo demandava maior modularização dos métodos dispostos no sistema.

Referências

[1] *Build a Python Web Server with Flask.* URL: https://projects.raspberrypi.org/en/projects/python-web-server-with-flask (acedido em 05/08/2022).

[2] Gregor Martynus. *Simplified HTTP request client.* https://github.com/request/request. (Acedido em 05/08/2022).