Sistemas Distribuídos - TP5 - Rock Album vol. 1 - Implementação utilizando REST API

Leandro Lázaro Araújo Vieira - 3513[1], Mateus Pinto da Silva - 3489[2]

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-caf) – Florestal – MG – Brasil

(leandro.lazaro[1], mateus.p.silva[2])@ufv.br

Resumo. *Implementamos a nossa especificação de sistema distribuído de álbum de figurinhas, o Rock Album vol. 1, utilizando API REST. Optamos por fazer isso da forma mais simplificada o possível, utilizando a linguagem de programação Python, o banco de dados SQLite3, o framework Flask de API REST e a biblioteca Requests para fazer requisições HTTP. Muito código dos trabalhos práticos de implementação via Socket e Middleware RMI foi reaproveitado. Não fizemos interface gráfica nem tratamentos para segurança. Como resultado, conseguimos implementar toda a nossa própria especificação de forma bastante concisa, e aprendemos bastante sobre as utilidades que os APIs REST oferecem. Também descrevemos rapidamente as diferenças percebidas entre esta e as outras duas implementações que fizemos. Além disso, nosso trabalho é facilmente reproduzível, bastando que se instale o interpretador Python 3 e um ambiente Virtualenv para executá-lo, visto que o framework Flask e a biblioteca Request não são nativos da linguagem, embora o banco de dados SQLite3 seja.*

1. Informações importantes

Nossos objetivos com este trabalho são aprender o máximo possível do conteúdo da disciplina de forma prática - ou seja, entender o funcionamento dos APIs REST, como utilizá-los e principalmente quando utilizá-los - e também conseguir uma boa nota. Como interface gráfica e tratamentos para segurança não serão avaliados, optamos por não fazê-los. Embora pareça uma decisão evasiva e até preguiçosa de nossa parte, isso nos permitiu um foco completo em realizar a troca de mensagens via a API REST Flask, além de nos poupar tempo que foi gasto também no conteúdo de outras disciplinas. Assim, toda a documentação também seguirá este foco, deixando de lado alguns detalhes sobre como lidar com a linguagem Python, SQL, etc para detalhar e explicitar as decisões sobre a API REST, porquê as tomamos e porquê acreditamos ter sido boas escolhas. Além disso, assume-se que o leitor desta documentação já tenha lido as documentações dos trabalhos práticos anteriores feitos via Sockets (cujo título é "Sistemas Distribuídos - TP3 - Rock Album vol. 1 - Implementação utilizando socket") e via Middleware RMI (cujo título é "Sistemas Distribuídos - TP4 - Rock Album vol. 1 - Implementação utilizando Middleware RMI"), portanto partes reaproveitadas de código não serão reexplicadas.

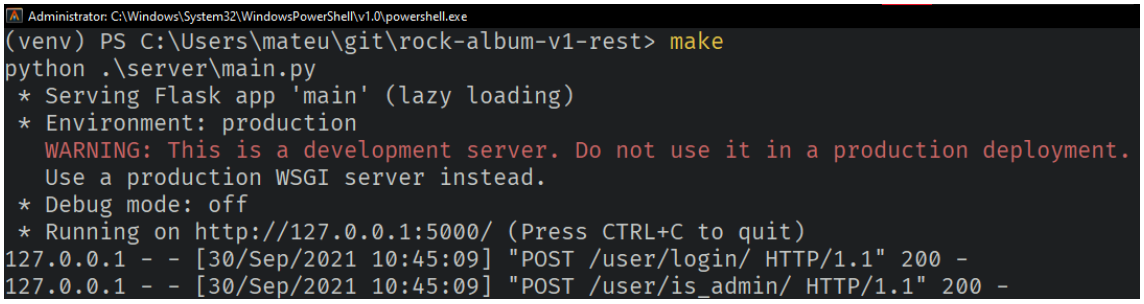

Figura 1. Processo servidor sendo executado, recebendo algumas requisições POST.

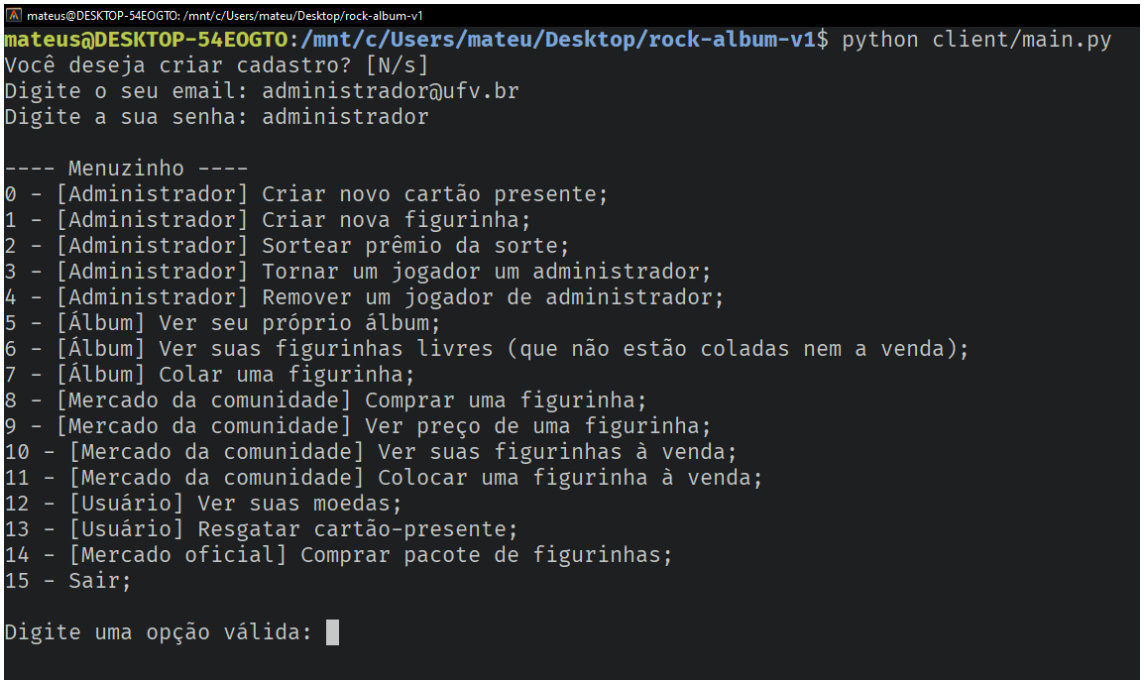

Figura 2. Processo cliente sendo executado com a efetuação do login de um administrador.

2. Como executar

Para executar os processos do sistema distribuído, é necessário ter o interpretador Python 3 e o gerenciador de ambientes Python 3 Virtualenv. É possível executá-los em quaisquer sistemas operacionais que tenham implementações do interpretador citado e a API de Sockets. O trabalho foi testado no Windows 10 e no Linux Ubuntu. Além disso, é preciso que a porta 5000 esteja aberta no Firewall do sistema operacional. Caso não seja possível abrí-la por quaisquer motivos, é possível escolher outra através do arquivo **config.json** que está na raiz do projeto.

Antes da execução, é necessário criar um ambiente virtual Python 3 Virtualenv. Para isso, execute o comando **virtualenv venv**. Para ativar o ambiente virtual no Linux, utilize o comando **source venv/bin/activate** e no Windows, utilize o comando **.\venv\Scripts\activate.ps1** no PowerShell para fazer o mesmo. Depois disso, utilize

o comando **pip install -r requirements.txt** para instalar o Flask e a biblioteca requests nesse ambiente. Para a correta execução e teste do sistema distribuídos, duas instâncias do terminal com o ambiente virtual ativado são necessárias, então o comando de ativação (seja em sua versão Linux ou Windows) precisa ser repetido duas vezes, uma em cada janela de terminal.

Com os pré-requisitos, finalmente será possível executar o sistema distribuído. O processo servidor pode ser iniciado em um terminal criado acima com o comando **make**. O processo cliente pode ser iniciado no último terminal criado acima disponível com o comando **make run**. Embora seja possível registrar no sistema pelo próprio cliente, criamos os seguintes usuários para facilitar a avaliação da professora/monitor:

Email	**Senha**	**É administrador?**
administrador@ufv.br	administrador	Sim
a@ufv.br	a	Não
b@ufv.br	b	Não

A versão do banco de dados é exatamente igual as dos dois últimos trabalhos práticos, e toda a interface do cliente também é idêntica, a fim de poupar-nos retrabalho e ajudar na correção.

3. Decisões de implementação

Como optamos utilizar a arquitetura de rotas desde o trabalho prático via API de Sockets e o REST API também o utiliza, bastou definirmos qual o método de cada uma das rotas. Utilizamos apenas o PUT e o POST, pois são os mais comuns para sistemas distribuídos que não rodam em navegadores. A diferença entre eles é que o PUT é usado para operações idempotentes, enquanto o POST é usado para operações que alteram o banco de dados. Assim, bastou que olhássemos quais rotas alteravam o banco de dados para classificá-las corretamente.

Rota	Método
/admin/create_giftcard/	POST
/admin/create_stickers/	POST
/admin/draw_lucky_prize/	POST
/admin/op/	POST
/admin/unop/	POST
/album/get_album/	PUT
/album/get_free_stickers/	PUT
/album/paste_sticker/	POST
/community_market/buy_sticker/	POST
/community_market/get_sticker_price/	PUT
/community_market/get_stickers_waiting_for_sale/	PUT
/community_market/put_sticker_to_sell/	POST
/user/get_coins/	PUT
/user/is_admin/	PUT
/user/login/	PUT
/user/register/	POST
/user/retrieve_giftcard/	POST
/official_market/buy_sticker_pack/	POST

Tabela 1. Rotas do sistema distribuído e seus métodos correspondentes.

4. O banco de dados

Não houve nenhuma alteração no banco de dados, nem quanto às suas tabelas nem ao driver utilizado. Assim, todo o modelo, os motivos das decisões de implementação são iguais as dos trabalhos anteriores.

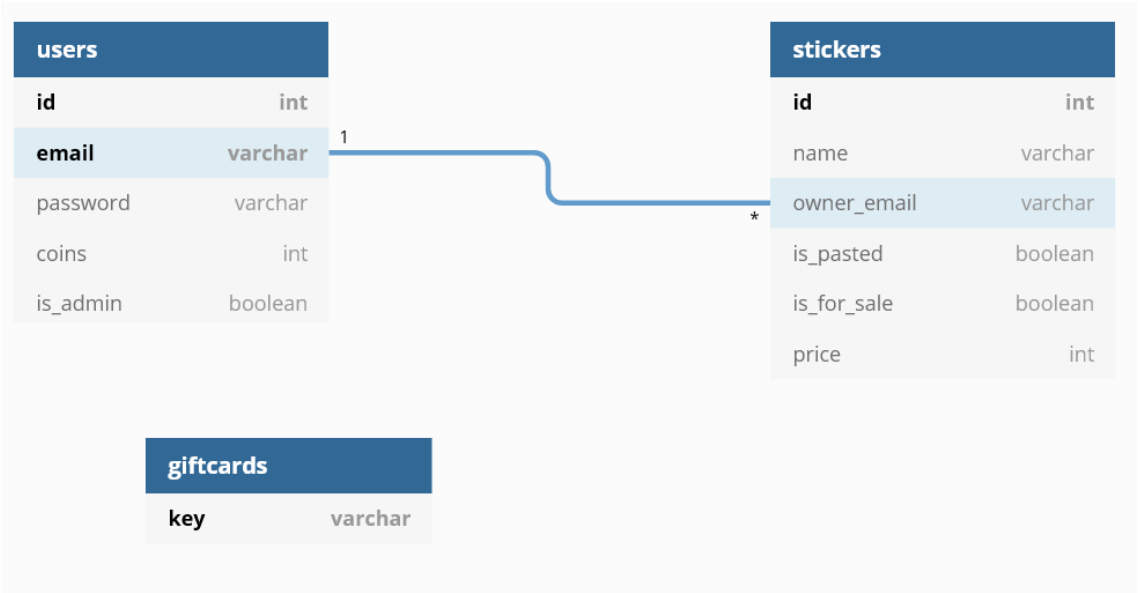

Figura 3. Diagrama entidade-relacionamento do banco de dados.

5. O processo servidor

O processo servidor do nosso sistema distribuído funciona, em alto nível, como um servidor qualquer feito em API REST, recebendo requisições em JSON, fazendo operações com banco de dados, e retornando um JSON de resposta para o cliente. Todas essas operações são feitas em um laço indeterminado, que só para através de interrupção do processo. Assim, o processo servidor pode ser visto como um grande receptor e cuspidor de JSONs. Note que essa foi a forma exata empregada no trabalho prático via API de Sockets, porém toda a serialização e desserialização da representação externa de dados JSON é feita implicitamente pelo Flask.

```
127.0.0.1 - - [30/Sep/2021 10:46:23] "POST /album/get_album/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:27] "POST /user/get_coins/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:29] "PUT /official_market/buy_sticker_pack/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:30] "PUT /official_market/buy_sticker_pack/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:31] "PUT /official_market/buy_sticker_pack/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:36] "POST /album/get_free_stickers/ HTTP/1.1" 200 -
127.0.0.1 - - [30/Sep/2021 10:46:43] "PUT /album/paste_sticker/ HTTP/1.1" 200 -
```

Figura 4. Processo servidor em execução.

5.1. A API REST

Todo o processo servidor foi escrito em apenas um arquivo, pois ele ficou muito simples e dividi-lo apenas pioraria a legibilidade do código. Nele, há apenas funções expostas pelo Flask para as rotas públicas e funções não expostas para as rotas privadas. Duas das funções não expostas são feitas para iniciar e encerrar uma conexão ao banco de dados. Sobre a inicialização do servidor, é chamado o método **run**() do framework, passando o endereço do host e a porta a serem utilizados.

```
1  from flask import Flask, request, g
2  import json
3  import sqlite3
4
5  CFG = json.load(open('config.json'))
6  DATABASE_FILENAME = CFG['SQLITE_FILE']
7  APP = Flask(__name__)
8
9
10 def get_database():
11     DATABASE = getattr(g, '_database', None)
12     if DATABASE is None:
13         DATABASE = g._database = sqlite3.connect(DATABASE_FILENAME)
14     return DATABASE
15
16
17 @APP.teardown_appcontext
18 def close_connection(_):
19     DATABASE = getattr(g, '_database', None)
20     if DATABASE is not None:
21         DATABASE.close()
22
23 [...]
24
25 if __name__ == '__main__':
```

```
26      APP.run(host=CFG['HOST'], port=CFG['PORT'])
```

Fragmento de código 1. Código do servidor feito em Flask. São mostradas as funções que iniciam e encerram a conexão com banco de dados, além do início do próprio servidor.

5.2. As rotas

Como já dito, todas as rotas foram implementadas via funções do servidor. Assim, as rotas auxiliares são funções não expostas e podem retornar diferentes tipos, enquanto as rotas públicas são funções expostas pelo Flask que retornam JSONs. Rotas auxiliares são úteis para fazer verificações e aumentar a redigibilidade e reuso de código, enquanto as rotas públicas são de fato as chamadas pelos clientes. Em relação à entrada, todas são em JSONs, pois é a representação externa de dados do sistema distribuído, ou seja, a informação enviada pelo cliente através de uma requisição é sempre um JSON.

```
1 def __is_this_sticker_waiting_for_sale():
2     DB_CUR = get_database().cursor()
3     return 1 if list(DB_CUR.execute(f'SELECT COUNT(*) FROM stickers
    WHERE id="{request.form["sticker_id"]}" AND is_for_sale=1'))[0][0]
    == 1 else 0
```

Fragmento de código 2. Exemplo de rota auxiliar implementada via função não exposta.

```
1 @APP.route('/user/login/', methods=['POST'])
2 def user__login():
3     DB_CUR = get_database().cursor()
4     if list(DB_CUR.execute(f'SELECT COUNT(*) FROM users WHERE email="{
    request.form["email"]}" AND password="{request.form["password"]}"')
    )[0][0] == 1:
5         return {'error': 0}
6     else:
7         return {'error': 1, 'error_message': 'Usuario nao encontrado!'}
```

Fragmento de código 3. Rota de login do processo servidor implementada via função exposta.

6. O processo cliente

O processo cliente do nosso sistema distribuído funciona, em alto nível, recebendo informações do terminal, enviando requisições HTTP para o servidor em JSON, recebendo respostas HTTP em JSON e tratando-as a fim de exibi-las de forma amigável ao usuário. O processo cliente desta implementação ficou bastante parecida com a da implementação via API de Sockets. Para lidar com as requisições e respostas HTTP, usamos a biblioteca **requests**.

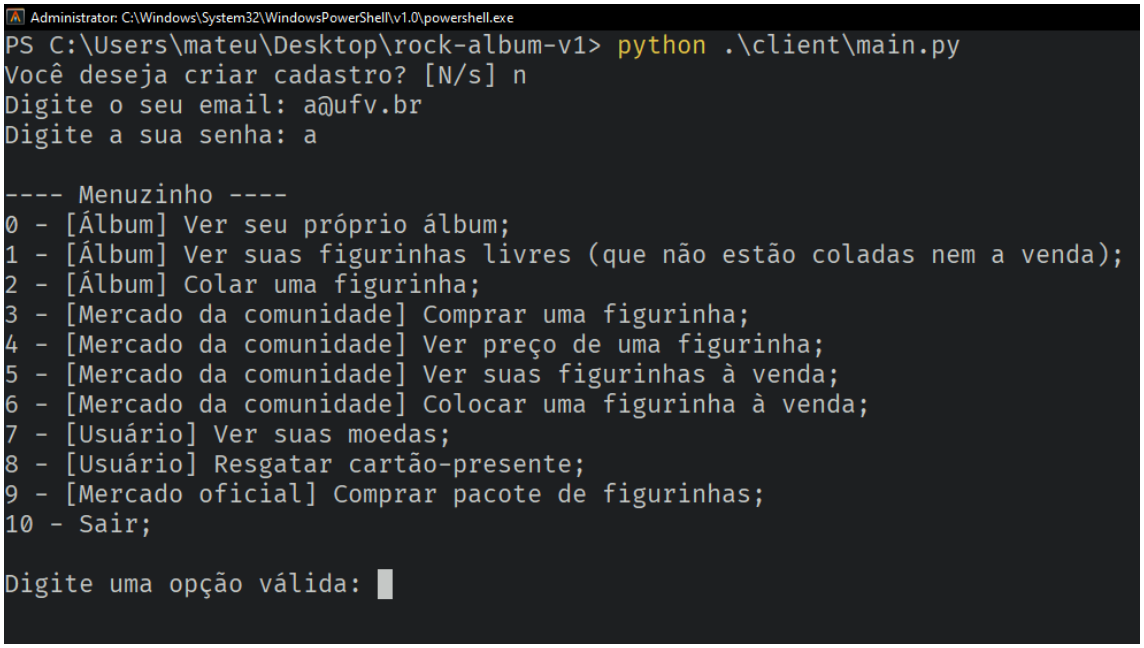

Figura 5. Processo cliente em execução.

6.1. A API REST

Toda a API REST foi abstraída utilizando as funções **put()** e **post()** da biblioteca **requests**. Essas funções funcionam de forma bastante parecida com a **contact_server()** da implementação de API de Sockets, recebendo a requisição JSON como método e o endereço (com a porta) do servidor e devolvendo a resposta em JSON.

```
RESPONSE = requests.put(SERVER_URL + 'album/get_album/', data=REQUEST).
    json()
```

Fragmento de código 4. Exemplo de uso da biblioteca requests.

6.2. As rotas

Com a possibilidade de enviar e receber JSONs via HTTP, o único passo que falta é criar esses JSONs de envio e exibir de forma amigável os JSONs de resposta. Para isso, utilizamos uma função de roteamento que escolhe a rota certa para criar o JSON certo coletando os dados de forma amigável do usuário para criar a requisição e exibindo de forma amigável sua resposta. Além disso, a função cria um menu bonito, porém resolvemos não mostrar isso na documentação por não ser o foco.

```
def router(EMAIL, IS_ADMIN):
    ADMIN = [
        ('[Administrador] Criar novo cartao presente', routes.
    admin__create_giftcard),
        ('[Administrador] Criar nova figurinha', routes.
    admin__create_stickers),
        ('[Administrador] Sortear premio da sorte', routes.
    admin__draw_lucky_prize),
        ('[Administrador] Tornar um jogador um administrador', routes.
    admin__op),
        ('[Administrador] Remover um jogador de administrador', routes.
    admin__unop),
```

```
    ]

    NON_ADMIN = [
        ('[Album] Ver seu proprio album', routes.album__get_album),
        ('[Album] Ver suas figurinhas livres (que nao estao coladas nem a venda)', routes.album__get_free_stickers),
        ('[Album] Colar uma figurinha', routes.album__paste_sticker),
        ('[Mercado da comunidade] Comprar uma figurinha', routes.community_market__buy_sticker),
        ('[Mercado da comunidade] Ver preco de uma figurinha', routes.community_market__get_sticker_price),
        ('[Mercado da comunidade] Ver suas figurinhas a venda', routes.community_market__get_stickers_waiting_for_sale),
        ('[Mercado da comunidade] Colocar uma figurinha a venda', routes.community_market__put_sticker_to_sell),
        ('[Usuario] Ver suas moedas', routes.user__get_coins),
        ('[Usuario] Resgatar cartao-presente', routes.user__retrieve_giftcard),
        ('[Mercado oficial] Comprar pacote de figurinhas', routes.official_market__buy_sticker_pack),
        ('Sair', lambda *_: quit())
    ]

    ROUTES = ADMIN + NON_ADMIN if IS_ADMIN else NON_ADMIN
```

Fragmento de código 5. O roteador que escolhe o método certo a ser chamado do processo cliente.

Todas as rotas do processo cliente seguem o mesmo padrão, necessitado do email do usuário para ser enviado juntamente com a requisição e não retornam nada. Seu funcionamento interno também é padronizado. Primeiro, é criada a requisição coletando os dados do usuário, envia essa requisição para o servidor através de HTTP via biblioteca **requests**, e exibe a resposta de forma amigável.

```
def admin__op(EMAIL):
    REQUEST = {
        'email': EMAIL,
        'target_email': input('Insira o email a ser tornado administrador: '),
    }

    RESPONSE = requests.post(SERVER_URL + 'admin/op/', data=REQUEST).json()

    if RESPONSE['error'] == 1:
        print(f'Erro! {RESPONSE["error_message"]}')
    else:
        print('Feito!')
```

Fragmento de código 6. Método de tornar um usuário administrador do processo cliente.

7. Considerações finais

Através deste trabalho, pudemos perceber o empoderamento que as APIs REST oferecem. O único problema encontrado foi o menor desempenho em comparação à implementação

via API de Sockets. A produtividade foi a maior das três implementações e o código foi o mais legível. Além disso, diferentemente da implementação via Middleware RMI, a implementação via API REST não necessitou de conversão de código, e pudemos usar programação funcional tranquilamente.

Entretanto, o framework Flask não nos agradou muito. Um de seus problemas é impossibilidade de definir os tipos de cada um dos campos da requisição JSON do cliente. O desempenho deste framework é bastante reduzido em comparação a outros de outras linguagens que não foram permitidas, como o Node e o Deno (ambos da linguagem Javascript).

Com relação a especificação do sistema distribuído em si, pudemos implementar toda a nossa própria especificação do Rock Album vol 1 sem maiores problemas. Criamos um sistema distribuído de processos portáteis e reproduzíveis, capaz de ser testado em diferentes sistemas operacionais e sem muitos pré-requisitos.