Desenvolvimento de um Sistema de Chat e Gestão com CRUD usando Flask e Tkinter

Prof. Pedro Capelari

17 de dezembro de 2024

Sumário

1 Introdução

Este material aborda a implementação de um sistema que combina um **chat em tempo real** com um **sistema de gestão CRUD (Create, Read, Update, Delete)** usando a linguagem de programação **Python**. Utilizaremos as seguintes ferramentas:

- **Flask**: Framework web leve para APIs RESTful.
- **Flask-SocketIO**: Extensão do Flask para comunicação em tempo real via WebSocket.
- **Tkinter**: Biblioteca padrão do Python para interfaces gráficas (GUIs).
- **Socket.IO**: Protocolo para comunicação bidirecional contínua entre cliente e servidor.
- **Requests**: Biblioteca para realizar requisições HTTP.

2 Arquitetura Cliente-Servidor

2.1 Funcionamento da Arquitetura Cliente-Servidor

A arquitetura **cliente-servidor** é um modelo de comunicação em que o processamento é dividido entre dois tipos de entidades:

- **Cliente**: É a parte que faz solicitações ao servidor e consome os serviços fornecidos por ele. Neste projeto, o cliente é uma aplicação desenvolvida com Tkinter.
- **Servidor**: É responsável por processar as solicitações dos clientes e retornar as respostas. O servidor gerencia dados e a lógica de negócio. Neste projeto, o servidor é implementado com Flask e Flask-SocketIO.

3 Teoria dos Protocolos HTTP e APIs RESTful

3.1 Protocolos HTTP

O protocolo HTTP (Hypertext Transfer Protocol) é a base para a comunicação na web. Ele funciona como um protocolo de requisição-resposta entre um cliente (navegador web) e um servidor. As principais características do HTTP incluem:

- **Métodos HTTP**: As operações principais do HTTP são realizadas por métodos como GET, POST, PUT e DELETE.
- **Status Codes**: Os códigos de status HTTP indicam o resultado da requisição, como 200 (OK), 404 (Not Found), 500 (Internal Server Error).
- **Headers**: Os cabeçalhos HTTP fornecem informações adicionais sobre a requisição ou resposta, como content-type, user-agent, etc.

3.2 APIs RESTful

REST (Representational State Transfer) é um estilo de arquitetura para projetar APIs. As APIs RESTful seguem seis princípios básicos:

- **Interface Uniforme**: As interações são feitas por meio de uma interface uniforme (recursos identificados por URLs).
- **Stateless**: Cada requisição do cliente para o servidor deve conter todas as informações necessárias para entender e processar a requisição.
- **Cacheable**: As respostas devem ser definidas como cacheáveis ou não.
- **Layered System**: A arquitetura pode ser composta de camadas hierárquicas.
- **Code on Demand (opcional)**: Servidores podem fornecer código executável para clientes quando necessário.
- **Client-Server**: A separação de responsabilidades entre cliente e servidor.

APIs RESTful utilizam métodos HTTP para realizar operações CRUD:

- **GET**: Recupera dados de um servidor.
- **POST**: Envia dados para serem processados pelo servidor.
- **PUT**: Atualiza dados existentes no servidor.
- **DELETE**: Remove dados do servidor.

4 Implementação do Projeto

4.1 Servidor com Flask e Flask-SocketIO

O servidor é responsável por gerenciar as operações CRUD e a comunicação em tempo real via WebSocket.

Listing 1: Código do servidor em Flask com Flask-SocketIO

```
from flask import Flask, request, jsonify
from flask_socketio import SocketIO, emit

app = Flask(__name__)
socketio = SocketIO(app, cors_allowed_origins="*")

# Simula  o de banco de dados em mem ria
users = []

# Rota CRUD para criar e listar usu rios
@app.route('/users', methods=['GET', 'POST'])
def manage_users():
```

```
    if request.method == 'POST':
        data = request.get_json()
        users.append(data)
        return jsonify({"message": "Usu rio criado com
            sucesso!"}), 201
    return jsonify(users), 200

# Rota para atualizar ou excluir usu rios
@app.route('/users/<int:user_id>', methods=['PUT', 'DELETE'
    ])
def handle_user(user_id):
    if user_id >= len(users):
        return jsonify({"error": "Usu rio n o encontrado"
            }), 404

    if request.method == 'PUT':
        data = request.get_json()
        users[user_id].update(data)
        return jsonify({"message": "Usu rio atualizado!"}),
             200

    if request.method == 'DELETE':
        users.pop(user_id)
        return jsonify({"message": "Usu rio exclu do!"}),
            200

# Evento de chat em tempo real
@socketio.on('send_message')
def handle_message(data):
    emit('receive_message', data, broadcast=True)

if __name__ == '__main__':
    socketio.run(app, debug=True)
```

4.2 Explicação do Servidor

- Rotas CRUD:
  - ‘GET /users‘: Retorna a lista de usuários.
  - ‘POST /users‘: Cria um novo usuário.
  - ‘PUT /users/¡id¿‘: Atualiza um usuário específico.
  - ‘DELETE /users/¡id¿‘: Exclui um usuário específico.
- WebSocket:
  - send message: Evento que recebe uma mensagem do cliente e retransmite a todos os clientes conectados.
  - receive message: Evento que todos os clientes escutam para receber mensagens em tempo real.

4.3 Cliente com Tkinter e Socket.IO

O cliente permite enviar mensagens em tempo real e realizar operações CRUD.

Listing 2: Código do cliente com Tkinter e Socket.IO

```
import tkinter as tk
from tkinter import messagebox
import socketio
import requests

# Conectar ao servidor Socket.IO
sio = socketio.Client()
sio.connect('http://localhost:5000')

# Fun   o para enviar mensagem ao chat
def send_message():
    message = entry_message.get()
    sio.emit('send_message', {'message': message})
    entry_message.delete(0, tk.END)

# Fun   o para criar usu rio
def create_user():
    name = entry_name.get()
    age = entry_age.get()
    response = requests.post('http://localhost:5000/users',
        json={'name': name, 'age': age})
    if response.status_code == 201:
        messagebox.showinfo("Sucesso", "Usu rio criado com
            sucesso!")
    entry_name.delete(0, tk.END)
    entry_age.delete(0, tk.END)

# Interface Tkinter
root = tk.Tk()
root.title("Chat e Gest o de Usu rios")

# Se   o de Chat
tk.Label(root, text="Chat").pack()
entry_message = tk.Entry(root)
entry_message.pack()
btn_send = tk.Button(root, text="Enviar Mensagem", command=
    send_message)
btn_send.pack()

# Se   o CRUD
tk.Label(root, text="Nome").pack()
entry_name = tk.Entry(root)
entry_name.pack()
tk.Label(root, text="Idade").pack()
entry_age = tk.Entry(root)
```

```
entry_age.pack()
btn_create = tk.Button(root, text="Criar Usu rio", command=
    create_user)
btn_create.pack()

# Loop da interface
root.mainloop()
```

4.4 Explicação do Cliente

- Função ‘send message‘: - Captura a mensagem da entrada de texto e a envia ao servidor via WebSocket.
- Função ‘create user‘: - Captura os dados de nome e idade e os envia ao servidor via uma requisição HTTP POST para criar um novo usuário.
- Interface Tkinter: - Contém campos de entrada e botões para o chat e o CRUD.

5 Conclusão

Neste material, vimos como construir um sistema de chat em tempo real integrado a um sistema de gestão CRUD usando:

- **Flask** para o servidor e APIs RESTful.
- **Flask-SocketIO** para comunicação em tempo real via WebSocket.
- **Tkinter** para a interface gráfica do cliente.
- **Requests** para requisições HTTP.

5.1 Expansões e Melhorias

Este projeto pode ser expandido para incluir autenticação de usuários, armazenamento em bancos de dados como MongoDB e interfaces mais avançadas. Algumas ideias para melhorias incluem:

- **Autenticação de Usuários**: Adicionar login e registro de usuários.
- **Banco de Dados**: Utilizar um banco de dados relacional ou NoSQL para armazenamento persistente de dados.
- **Interface Gráfica Avançada**: Melhorar a interface do usuário com bibliotecas como ‘ttk‘ e adicionar mais funcionalidades.
- **Deploy**: Fazer o deploy do aplicativo em um serviço de hospedagem, como Heroku ou AWS.

6 Referências

Para mais informações sobre as tecnologias utilizadas, consulte as seguintes referências:

- Flask: `https://flask.palletsprojects.com/`
- Flask-SocketIO: `https://flask-socketio.readthedocs.io/`
- Tkinter: `https://docs.python.org/3/library/tkinter.html`
- Socket.IO: `https://socket.io/`
- Requests: `https://docs.python-requests.org/`