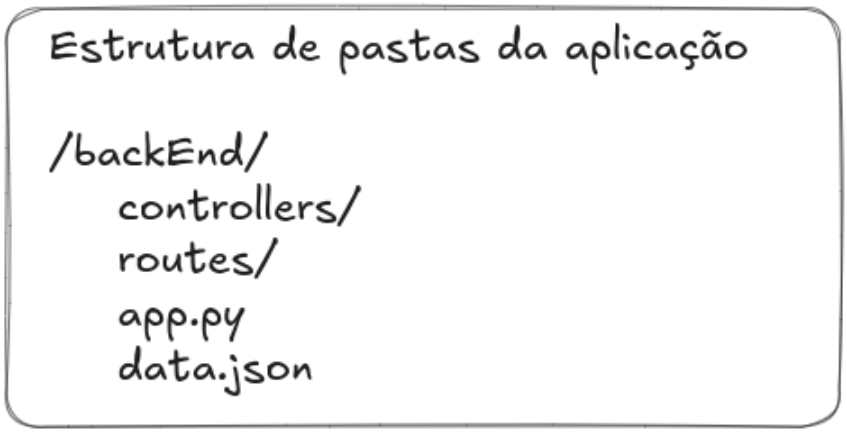
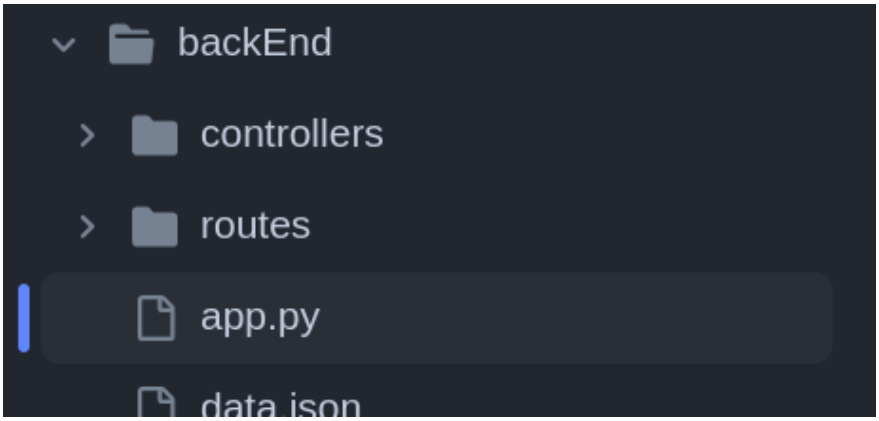
Python backEnd + JavaScript frontEnd

1. **Estrutura de pastas**

Comece construindo a nova estrutura de pastas da aplicação





2. App.py

```python
# Importa a classe Flask do módulo flask. A classe Flask é usada para criar uma instância do aplicativo web.
from flask import Flask

# Importa o blueprint user_routes do módulo user_routes que está dentro do diretório routes.
# Blueprints permitem dividir o aplicativo em componentes menores e reutilizáveis.
from routes.user_routes import user_routes

# Cria uma instância do aplicativo Flask. A variável app agora é o aplicativo Flask.
app = Flask(__name__)

# Registra o blueprint user_routes no aplicativo Flask.
# Isso permite que todas as rotas definidas em user_routes sejam adicionadas ao aplicativo principal.
app.register_blueprint(user_routes)

# Verifica se o script está sendo executado diretamente (e não importado como um módulo).
# Se for o caso, inicia o servidor web Flask. O parâmetro debug=True ativa o modo de depuração,
# que fornece feedback detalhado sobre erros e recarrega automaticamente o servidor quando há mudanças no
código.
if __name__ == "__main__":
    app.run(debug=True)
```

3. routes/user_routes.py

```python
# Importa a classe Blueprint do módulo flask. Blueprints são usados para criar grupos de rotas e funções associadas.
from flask import Blueprint

# Importa as funções do módulo user_controller que lidam com operações de CRUD para usuários.
# Estas funções serão usadas nas rotas definidas neste blueprint.
from controllers.user_controller import (
    get_all_users,
    get_user_by_id,
    get_user_by_name,
    add_user,
    update_user,
    delete_user,
)

# Cria uma instância do blueprint chamada "user_routes".
# O segundo parâmetro __name__ ajuda o Flask a localizar os recursos estáticos e templates associados a este blueprint.
user_routes = Blueprint("user_routes", __name__)

# Define uma rota para o endpoint "/usuarios" que responde a solicitações GET.
# Quando acessado, chama a função get_all_users para obter todos os usuários.
@user_routes.route("/usuarios", methods=["GET"])
def get_users():
    return get_all_users()

# Define uma rota para o endpoint "/usuarios/<int:id>" que responde a solicitações GET.
# Quando acessado com um ID de usuário, chama a função get_user_by_id para obter o usuário com o ID fornecido.
@user_routes.route("/usuarios/<int:id>", methods=["GET"])
def get_user(id):
    return get_user_by_id(id)

# Define uma rota para o endpoint "/usuarios/nome=<nome>" que responde a solicitações GET.
# Quando acessado com um nome de usuário, chama a função get_user_by_name para obter o usuário com o nome fornecido.
@user_routes.route("/usuarios/nome=<nome>", methods=["GET"])
def get_user_by_name_route(nome):
    return get_user_by_name(nome)

# Define uma rota para o endpoint "/usuarios" que responde a solicitações POST.
# Quando acessado, chama a função add_user para adicionar um novo usuário.
@user_routes.route("/usuarios", methods=["POST"])
def add_user_route():
    return add_user()

# Define uma rota para o endpoint "/usuarios/<int:id>" que responde a solicitações PUT.
# Quando acessado com um ID de usuário, chama a função update_user para atualizar o usuário com o ID fornecido.
@user_routes.route("/usuarios/<int:id>", methods=["PUT"])
def update_user_route(id):
    return update_user(id)

# Define uma rota para o endpoint "/usuarios/<int:id>" que responde a solicitações DELETE.
# Quando acessado com um ID de usuário, chama a função delete_user para excluir o usuário com o ID fornecido.
@user_routes.route("/usuarios/<int:id>", methods=["DELETE"])
def delete_user_route(id):
    return delete_user(id)
```

4.1 controllers/user_controller.py

```python
# Importa o módulo json para manipulação de arquivos JSON.
import json

# Importa as funções jsonify e request do Flask.
# jsonify é usado para criar respostas JSON a partir de objetos Python,
# e request é usado para acessar dados da solicitação HTTP.
from flask import jsonify, request


# Define a função para obter todos os usuários do arquivo JSON.
def get_all_users():
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Retorna os dados como uma resposta JSON.
    return jsonify(data)


# Define a função para obter um usuário específico pelo ID.
def get_user_by_id(id):
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Percorre a lista de usuários para encontrar o usuário com o ID fornecido.
    for item in data["usuarios"]:
        if item["id"] == id:
            # Retorna o usuário encontrado como uma resposta JSON.
            return jsonify(item)
    # Retorna uma mensagem de erro se o usuário não for encontrado.
    return jsonify({"message": "Não foi possível encontrar esse usuário!"})


# Define a função para obter um usuário específico pelo nome.
def get_user_by_name(nome):
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Cria uma lista para armazenar os usuários encontrados pelo nome.
    users = []
    # Percorre a lista de usuários para encontrar aqueles com o nome fornecido.
    for item in data["usuarios"]:
        if item["nome"].lower() == nome.lower():
            users.append(item)
    # Retorna o primeiro usuário encontrado como uma resposta JSON.
    if users:
        return jsonify(users[0])
    # Retorna uma mensagem de erro se nenhum usuário for encontrado.
    else:
        return jsonify({"error": "User not found"}), 404
```

4.2 controllers/user_controller.py

```python
# Define a função para adicionar um novo usuário.
def add_user():
    # Obtém os dados JSON da solicitação POST.
    new_data = request.get_json()
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Atribui um novo ID ao usuário, que é o próximo número na sequência.
    new_data["id"] = len(data["usuarios"]) + 1
    # Adiciona o novo usuário à lista de usuários.
    data["usuarios"].append(new_data)
    # Abre o arquivo "data.json" com codificação UTF-8 para escrita.
    with open("data.json", "w", encoding="utf-8") as json_file:
        # Salva os dados atualizados de volta no arquivo JSON.
        json.dump(data, json_file, indent=4)
    # Retorna uma mensagem de sucesso com o status HTTP 201 (Criado).
    return jsonify({"message": "Cadastrado com sucesso!"}), 201


# Define a função para atualizar um usuário existente.
def update_user(id):
    # Obtém os dados JSON da solicitação PUT.
    updated_data = request.get_json()
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Percorre a lista de usuários para encontrar o usuário com o ID fornecido.
    for item in data["usuarios"]:
        if item["id"] == id:
            # Atualiza os dados do usuário com as novas informações.
            item.update(updated_data)
            # Abre o arquivo "data.json" com codificação UTF-8 para escrita.
            with open("data.json", "w", encoding="utf-8") as json_file:
                # Salva os dados atualizados de volta no arquivo JSON.
                json.dump(data, json_file)
            # Retorna uma mensagem de sucesso.
            return jsonify({"message": "Usuário atualizado com sucesso!"})
    # Retorna uma mensagem de erro se o usuário não for encontrado.
    return jsonify({"message": "Usuário não encontrado!"}), 404


# Define a função para excluir um usuário.
def delete_user(id):
    # Abre o arquivo "data.json" com codificação UTF-8 para leitura.
    with open("data.json", encoding="utf-8") as json_file:
        # Carrega os dados JSON do arquivo para uma variável Python.
        data = json.load(json_file)
    # Percorre a lista de usuários para encontrar o usuário com o ID fornecido.
    for item in data["usuarios"]:
        if item["id"] == id:
            # Remove o usuário da lista.
            data["usuarios"].remove(item)
            # Abre o arquivo "data.json" com codificação UTF-8 para escrita.
            with open("data.json", "w", encoding="utf-8") as json_file:
                # Salva os dados atualizados de volta no arquivo JSON.
                json.dump(data, json_file)
            # Retorna uma mensagem de sucesso.
            return jsonify({"message": "Usuário deletado com sucesso!"})
    # Retorna uma mensagem de erro se o usuário não for encontrado.
    return jsonify({"message": "Usuário não encontrado!"}), 404
```

Descrição da Aplicação
Esta aplicação é uma API RESTful construída com Flask, destinada a gerenciar um conjunto de dados de usuários armazenados em um arquivo JSON. A API oferece operações básicas de CRUD (Create, Read, Update, Delete) para manipular informações sobre usuários. Aqui está um resumo das funcionalidades e benefícios da aplicação:

Funcionalidades
Obter Todos os Usuários

Endpoint: /usuarios (GET)
Descrição: Recupera uma lista de todos os usuários armazenados no arquivo JSON e retorna os dados em formato JSON.
Obter Usuário por ID

Endpoint: /usuarios/<int:id> (GET)
Descrição: Recupera as informações de um usuário específico com base no ID fornecido. Se o usuário for encontrado, os dados são retornados em formato JSON; caso contrário, é retornada uma mensagem de erro.
Obter Usuário por Nome

Endpoint: /usuarios/nome=<nome> (GET)
Descrição: Recupera as informações de um usuário com base no nome fornecido. O nome é comparado de forma insensível a maiúsculas/minúsculas. Se o usuário for encontrado, os dados são retornados em formato JSON; caso contrário, é retornada uma mensagem de erro.
Adicionar Novo Usuário

Endpoint: /usuarios (POST)
Descrição: Adiciona um novo usuário ao arquivo JSON. O usuário deve ser enviado como dados JSON na solicitação. O novo usuário é atribuído a um ID único automaticamente. A operação retorna uma mensagem de sucesso.
Atualizar Usuário

Endpoint: /usuarios/<int:id> (PUT)
Descrição: Atualiza as informações de um usuário existente com base no ID fornecido. Os dados de atualização devem ser enviados como JSON na solicitação. Se o usuário for encontrado e atualizado, é retornada uma mensagem de sucesso; caso contrário, é retornada uma mensagem de erro.
Excluir Usuário

Endpoint: /usuarios/<int:id> (DELETE)
Descrição: Remove um usuário específico com base no ID fornecido. Se o usuário for encontrado e removido, é retornada uma mensagem de sucesso; caso contrário, é retornada uma mensagem de erro.
Divisão e Benefícios
Blueprints: A aplicação utiliza a funcionalidade de blueprints do Flask para organizar as rotas relacionadas aos usuários em um componente separado. Isso ajuda a modularizar o código e facilita a manutenção e expansão da aplicação.

Separação de Preocupações: O código é dividido em três componentes principais:

Roteamento (user_routes): Define as rotas e métodos HTTP para cada operação CRUD.
Controle (user_controller): Contém a lógica de negócios para manipular dados de usuários, como leitura, escrita e atualização de informações.
Dados (data.json): Armazena os dados dos usuários em um formato JSON, permitindo fácil acesso e modificação.
Benefícios:

Organização: A separação em blueprints e módulos permite uma estrutura mais limpa e organizada, facilitando o entendimento e a manutenção do código.
Reusabilidade: O uso de blueprints permite que as rotas relacionadas a usuários sejam reutilizadas e facilmente integradas em outros aplicativos Flask.
Escalabilidade: A modularização facilita a adição de novas funcionalidades e endpoints no futuro.
Facilidade de Desenvolvimento: As operações CRUD são implementadas de maneira clara e direta, aproveitando o Flask para simplificar o desenvolvimento da API.