

FIAP - Faculdade de Informática e Administração Paulista
Architecture Analytics And NoSQL

ISABELLA FRANCISCO HEDER
RM 561300

CHECKPOINT 2
Case Docker

TURMA: 1TSCPV

São Paulo, SP
2025

ENUNCIADO:

1) Criar um Dockerfile para a aplicação Flask:

Crie um arquivo chamado Dockerfile para construir a imagem Docker da aplicação Flask. ✓

A imagem deverá ser baseada na imagem oficial do Python. ✓

O Dockerfile deve instalar o Flask, SQLAlchemy e a biblioteca psycopg2 para conexão com o banco de dados PostgreSQL. ✓

O Dockerfile deve definir a variável de ambiente FLASK_APP ✓ e o comando para iniciar a aplicação (flask run) ✓

2) Criar a aplicação Flask com SQLAlchemy:

Crie um arquivo chamado app.py que implementa uma aplicação Flask simples com SQLAlchemy. ✓

A aplicação deve ter uma rota / que **insira um novo usuário no banco de dados e outra rota /users que retorne todos os usuários armazenados.** ✓ (adicionei read, update e delete a partir da busca ID)

A aplicação deve se conectar ao banco de dados PostgreSQL e armazenar os usuários com um modelo simples que contenha id, name e email. ✓

3) Criar um docker-compose.yml para orquestrar os containers:

Crie um arquivo docker-compose.yml para orquestrar a aplicação e o banco de dados. ✓

O arquivo docker-compose.yml deve definir dois serviços:

app: O serviço que rodará a aplicação Flask. ✓

db: O serviço que rodará o banco de dados PostgreSQL. ✓

Defina a rede entre os containers para permitir a comunicação entre o container da aplicação e o do banco de dados (as portas). ✓

Utilize variáveis de ambiente no docker-compose.yml para configurar a conexão com o banco de dados (por exemplo, POSTGRES_USER, POSTGRES_PASSWORD e POSTGRES_DB). ✓

4) Criar o banco de dados PostgreSQL: (CRUD)

Configure o banco de dados PostgreSQL para inicializar automaticamente com uma tabela chamada users que tenha as colunas id, name e email. ✓ (arquivo .sql)

Utilize um script SQL para criar essa tabela ou configure a aplicação para criá-la automaticamente ao ser iniciada. ✓ (roda o arquivo .sql automaticamente a partir do código no arquivo .yml)

5) Construir e rodar os containers:

Utilize o comando docker-compose build para construir os containers. ✓

Após a construção, utilize docker-compose up para iniciar a aplicação e o banco de dados. ✓

A aplicação Flask deve ser acessível na URL <http://localhost:5000/algo> ✓

6) Testar a aplicação:

Abra o navegador e acesse <http://localhost:5000/users>. Isso deverá retornar uma lista de usuários armazenados no banco de dados. ✓

Utilize uma ferramenta como o Postman ou cURL para enviar uma requisição POST para <http://localhost:5000/> com um payload JSON que contenha o nome e o email de um novo usuário. Após a requisição, o novo usuário deve ser adicionado ao banco de dados e aparecer na lista retornada pela rota /users. ✓

Dicas:

Docker e Docker Compose devem estar instalados em sua máquina ✓

Criar os seguintes arquivos:

Dockerfile (com a configuração da aplicação Flask). ✓

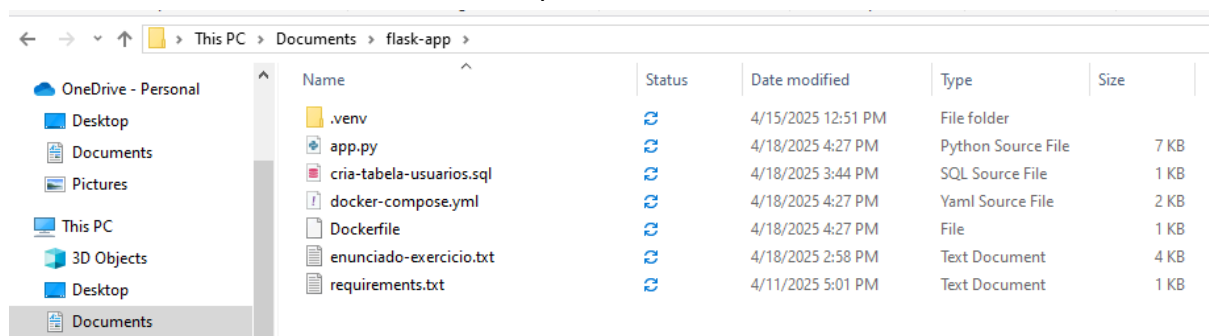
app.py (com a aplicação Flask e a integração com o banco de dados PostgreSQL). ✓
docker-compose.yml (para orquestrar a aplicação e o banco de dados). ✓
Script SQL para criar a tabela ou configuração automática da tabela no banco. ✓

RESOLUÇÃO:

PS: bd - banco de dados

DIRETÓRIO:

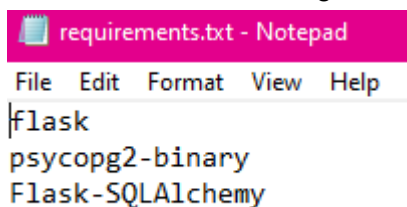
- ❖ O primeiro passo foi a criação de um diretório novo, para que nele, ficassem disponíveis todos os arquivos do CP
- ❖ `.venv` foi criado automaticamente pelo *vscode*



Name	Status	Date modified	Type	Size
.venv		4/15/2025 12:51 PM	File folder	
app.py		4/18/2025 4:27 PM	Python Source File	7 KB
cria-tabela-usuarios.sql		4/18/2025 3:44 PM	SQL Source File	1 KB
docker-compose.yml		4/18/2025 4:27 PM	Yaml Source File	2 KB
Dockerfile		4/18/2025 4:27 PM	File	1 KB
enunciado-exercicio.txt		4/18/2025 2:58 PM	Text Document	4 KB
requirements.txt		4/11/2025 5:01 PM	Text Document	1 KB

REQUIREMENTS.TXT:

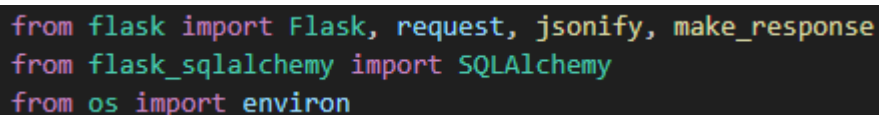
- ❖ A partir do apresentado no enunciado elaborado pelo professor e as aulas que tivemos, adicionei os seguintes requerimentos para o CP:



```
requirements.txt - Notepad
File Edit Format View Help
flask
psycopg2-binary
Flask-SQLAlchemy
```

APP.PY:

1. IMPORTAÇÕES



```
from flask import Flask, request, jsonify, make_response
from flask_sqlalchemy import SQLAlchemy
from os import environ
```

- ❖ *Flask* = para criar a aplicação *web*
- ❖ *request* = trabalha com requisições HTTP
- ❖ *jsonify* = formatar em *json*

- ❖ *make_response* = customizar respostas HTTP
- ❖ *sqlalchemy* = manipular banco de dados
- ❖ *environ* = acessar variáveis de ambiente

2. FLASK E BANCO DE DADOS

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = environ.get('DB_URL')
db = SQLAlchemy(app)

if not app.config['SQLALCHEMY_DATABASE_URI']:
    raise RuntimeError("A variável de ambiente 'DB_URL' não está configurada.")
```

- ❖ Configura a URL para conectar com o banco de dados (usando a variável DB_URL que tem o *link* do bd dentro dela)
- ❖ Liga a aplicação flask (app) com o SQLAlchemy (que vai gerenciar o bd)
- ❖ *if*: verifica se DB_URL tá configurada antes de rodar 'app'

3. FORMATO TABELA DO BD:

```
11
12 class User(db.Model):
13     __tablename__ = 'usuarios'
14
15     id = db.Column(db.Integer, primary_key=True)
16     usuario = db.Column(db.String(80), unique=True, nullable=False)
17     email = db.Column(db.String(120), unique=True, nullable=False)
18
19     def json(self):
20         return {'id': self.id, 'usuario': self.usuario, 'email': self.email}
21
22 with app.app_context():
23     db.create_all()
```

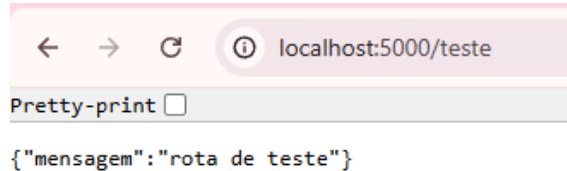
- ❖ Define classe *User* para a tabela 'usuarios', contendo o que cada usuário deve ter em seu 'cadastro'
- ❖ **id** = cria coluna do banco de dados com *integer* (inteiros), sendo uma chave primária (indispensável e obrigatória)
- ❖ **usuario** = cria coluna do bd com *string* (pode conter até 80 caracteres), sendo obrigatória, única (sem usuários iguais) e não podendo ser nulo
- ❖ **email** = mesma coisa que usuário só que com limite de 120 caracteres
- ❖ Faz com que o retorno desses dados sejam em formato JSON
- ❖ Inicia o bd criando um contexto para ele e criando a tabela

4. ROTAS E SERVIÇOS DA API (CRUD):

ROTA TESTE:

```
# ROTA TESTE
@app.route('/teste', methods=['GET']) # não precisa especificar o método, mas é bom para deixar claro o que a rota faz
def teste():
    return make_response(jsonify({'mensagem': 'rota de teste'}), 200) # retorna em json com o status 200 (ok)
```

- ❖ Cria a rota teste dada como “<http://localhost:5000/teste>”
- ❖ Esta é uma rota para conferir se tudo está funcionando; se tudo estiver rodando corretamente, essa mensagem vai aparecer no navegador



```
flask_app | 172.18.0.1 - - [18/Apr/2025 15:52:55] "GET /teste HTTP/1.1" 200 -
```

SERVIÇO CRIAÇÃO DE USUÁRIOS:

```
# CRIANDO USUÁRIO
@app.route('/create-usuarios', methods=['POST']) # rota para criar um usuário, usando o método POST
def criar_usuario():
    try:
        dados = request.get_json() # pega os dados do usuário em formato json
        # verifica se os dados estão corretos
        if not dados or 'usuario' not in dados or 'email' not in dados:
            return make_response(jsonify({'mensagem': 'dados inválidos ou incompletos'}), 400)
        # se os dados estiverem corretos, cria o usuário
        novo_usuario = User(usuario=dados['usuario'], email=dados['email']) # cria um novo usuário em formato json com os dados: usuario e email
        db.session.add(novo_usuario) # adiciona o usuário ao banco de dados
        db.session.commit() # salva as alterações no banco de dados
        return make_response(jsonify({'mensagem': 'usuário criado com sucesso', 'user': novo_usuario.json()}), 201) # retorna em json com o status 201 (criado)
    except Exception as e: # caso de erro
        # caso não seja erro de dados inválidos, retorna erro 500
        print(e) # mostra o erro
        return make_response(jsonify({'mensagem': 'erro ao criar usuário'}), 500)
```

- ❖ Cria a rota “/create-usuarios”, com o método *POST* (só aceita requisições *POST*) - o método se escolhe e se aplica no *Postman* (mais pra frente no documento eu explico melhor essa plataforma)
- ❖ Esta rota vai criar os usuários do banco de dados a partir da leitura dos dados enviados em formato JSON (na var *dados*)
- ❖ Os dados a serem lidos precisam ser enviados em formato JSON (ENTRADA) antes de rodar o site
- ❖ Note que: o *id* é criado automaticamente na SAÍDA
- ❖ *IF*: ele garante que as informações escritas na ENTRADA estejam corretas, exemplo: campos nulos; caso não, no terminal retornará o *status* 500 e na tela, será impressa a mensagem de erro criada através do *make_response*;
- ❖ Define o que o usuário e *email* vão ser fornecidos e vão formar o *User*
- ❖ Após a coleta dos dados é feita a criação do usuário (*.add*) que é salvo no bd (*.commit*)
- ❖ Se tudo estiver correto com os dados, o *make_response* (importado nesse mesmo arquivo) retornará a mensagem de sucesso na tarefa + os dados do novo usuário; além disso, no terminal retornará o *status* 201
- ❖ Caso algo, além da falta de dados nas áreas, estiver errado, o erro será impresso na tela, e junto dele uma mensagem de erro criada através do *make_response*; no terminal retornará o *status* 500

- ❖ Note que a rota muda e é sempre colocada no fim do link padrão “localhost:5000” começando com uma barra;

```
flask_app | 172.18.0.1 - - [18/Apr/2025 17:48:46] "POST /create-usuarios HTTP/1.1" 201 -
flask_app | 172.18.0.1 - - [18/Apr/2025 17:49:27] "POST /create-usuarios HTTP/1.1" 201 -
```

localhost:5000/create-usuarios

POST localhost:5000/create-usuarios

Body

```
1 {
2   "usuario": "isabella",
3   "email": "isabella.f.heder@gmail.com"
4 }
```

ENTRADA

201 CREATED • 44 ms • 293 B

Body

```
1 {
2   "mensagem": "usuário criado com sucesso",
3   "user": {
4     "email": "isabella.f.heder@gmail.com",
5     "id": 1,
6     "usuario": "isabella"
7   }
8 }
```

SAÍDA

localhost:5000/create-usuarios

POST localhost:5000/create-usuarios

Body

```
1 {
2   "usuario": "leticia",
3   "email": "leticia@gmail.com"
4 }
```

201 CREATED • 19 ms • 283 B

Body

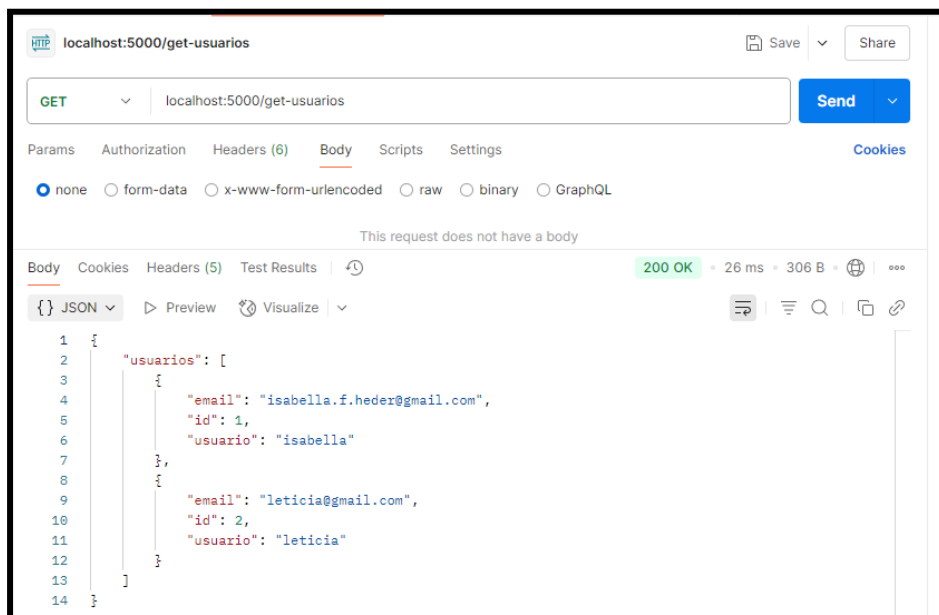
```
1 {
2   "mensagem": "usuário criado com sucesso",
3   "user": {
4     "email": "leticia@gmail.com",
5     "id": 2,
6     "usuario": "leticia"
7   }
8 }
```

SERVIÇO LISTAR TODOS OS USUÁRIOS:

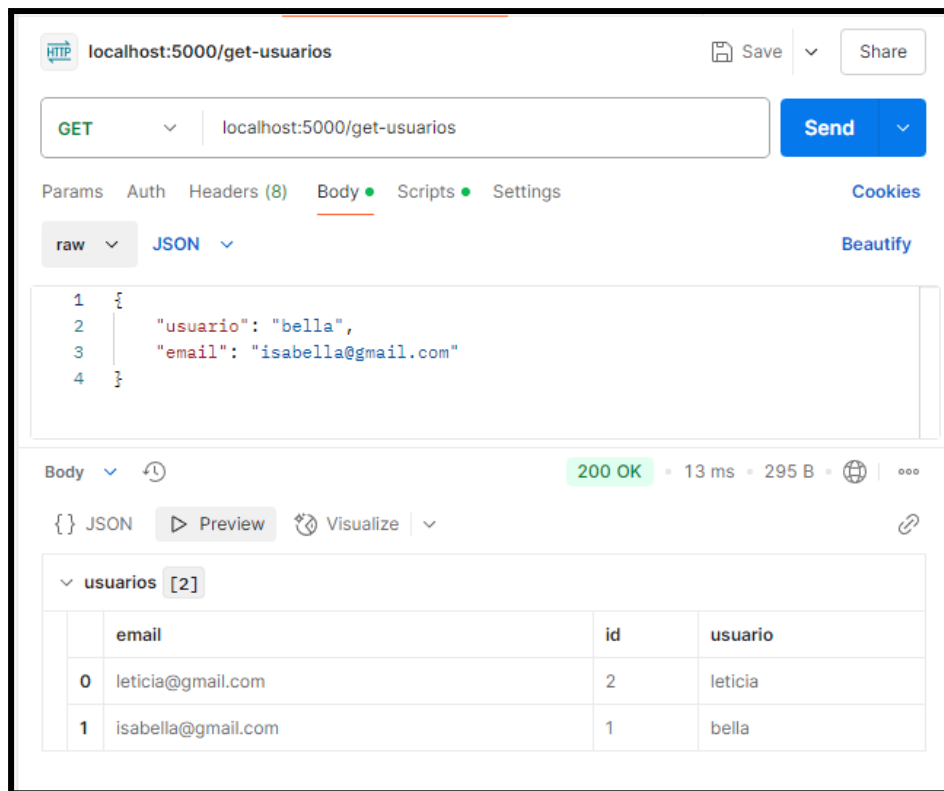
```
# PEGANDO TODOS OS USUÁRIOS
@app.route('/get-usuarios', methods=['GET']) # rota para pegar todos os usuários, usando o método GET
def pegar_usuarios():
    try:
        usuarios = User.query.all() # pega todos os usuários do banco de dados
        # verifica se a lista está vazia
        if not usuarios: # se não tiver nenhum usuário
            return make_response(jsonify({'mensagem': 'Nenhum usuário encontrado'}), 204)
        return make_response(jsonify({'usuarios': [usuario.json() for usuario in usuarios]}), 200) # retorna em json com o status 200 (ok)
    except Exception as e: # caso de erro
        print(e) # mostra o erro
        return make_response(jsonify({'mensagem': 'erro ao pegar usuários'}), 500)
```

- ❖ Cria a rota “/get-usuarios”, utilizando o método *GET* (pegar) para listar todos os usuários existentes no banco de dados
- ❖ Obtém todos os usuários do banco de dados
- ❖ Caso não tenha nenhum usuário, o código deve retornar uma mensagem avisando que não encontrou nenhum usuário e no terminal retornar o *status* 204
- ❖ Caso haja usuários, o código deverá retornar a lista de usuários salvos no banco de dados + o *status* 200 no terminal
- ❖ Caso algo, além da falta de usuários, estiver errado, o erro será impresso na tela, e junto dele uma mensagem de erro criada através do *make_response*

```
flask_app | 172.18.0.1 - - [18/Apr/2025 17:50:02] "GET /get-usuarios HTTP/1.1" 200 -
flask_app | 172.18.0.1 - - [18/Apr/2025 17:50:12] "GET /get-usuarios HTTP/1.1" 200 -
```



- ❖ O PRÓXIMO *GET* FOI FEITO **APÓS UMA ATUALIZAÇÃO DO USUÁRIO** (outro serviço a ser explicado na resolução), por isso os dados estão diferentes do que foi criado:

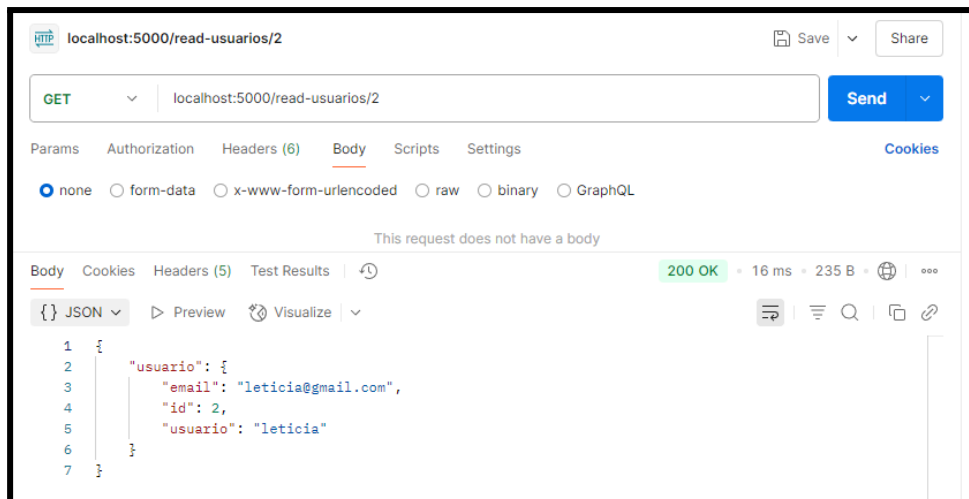


SERVIÇO IDENTIFICAR USUÁRIO POR ID:

```
# PEGANDO USUÁRIO PELO ID
@app.route('/read-usuarios/<int:id>', methods=['GET']) # rota para pegar um usuário pelo id, usando o método GET
def pegar_usuario(id):
    try:
        usuario = User.query.filter_by(id=id).first() # pega UM usuário pelo id
        # verificando se o usuário existe antes de retornar
        if usuario:
            return make_response(jsonify({'usuario': usuario.json()}), 200) # retorna o usuário em json com o status 200 (ok)
            return make_response(jsonify({'mensagem': 'usuário não encontrado'}), 404) # se o usuário não for encontrado, retorna erro 404 (não encontrado)
    except Exception as e: # caso de erro
        print(e) # mostra o erro
        return make_response(jsonify({'mensagem': 'erro ao pegar usuário'}), 500)
```

- ❖ Cria a rota “/read-usuarios/id”, utilizando o método *GET* (pegar) para pegar o usuário que tenha o *id* citado
- ❖ Busca um usuário filtrado pelo seu *id*
- ❖ Se for um usuário: retorna as informações do usuário e no terminal retorna o *status* 200
- ❖ Caso contrário: retorna mensagem de erro e no terminal o *status* 404
- ❖ Caso o erro seja outro, retorna mensagem de erro e *status* 500 no terminal

```
flask_app | 172.18.0.1 - - [18/Apr/2025 17:51:55] "GET /read-usuarios/2 HTTP/1.1" 200 -
```

SERVIÇO ATUALIZANDO USUÁRIO:

```
# ATUALIZANDO USUÁRIO
@app.route('/update-usuarios/<int:id>', methods=['PUT']) # rota para atualizar um usuário pelo id, usando o método PUT
def atualizar_usuario(id):
    try:
        usuario = User.query.filter_by(id=id).first()
        if usuario:
            dados = request.get_json() # pega os dados do usuário em formato json
            usuario.usuario = dados['usuario'] # atualiza o nome do usuário
            usuario.email = dados['email'] # atualiza o email do usuário
            db.session.commit() # salva as alterações no banco de dados
            return make_response(jsonify({'mensagem': 'usuário atualizado com sucesso', 'user': usuario.json()}), 200) # retorna em json com o status 200 (ok)
        return make_response(jsonify({'mensagem': 'usuário não encontrado'}), 404) # se o usuário não for encontrado, retorna erro 404 (não encontrado)
    except Exception as e:
        print(e) # mostra o erro
        return make_response(jsonify({'mensagem': 'erro ao atualizar usuário'}), 500)
```

- ❖ Cria a rota “/update-usuarios/id”, utilizando o método *PUT* para alterar dados do usuário (identificado pelo *id*)
- ❖ Igual à rota de criação de usuário, essa rota necessita de dados enviados em formato *JSON* (ENTRADA) antes de rodar o *site*
- ❖ Para averiguar meu teste, note que, no momento em que criei o meu usuário, eu criei ele utilizando meu nome completo (isabella) e meu *email* completo (isabella.f.heder@gmail.com)
- ❖ A entrada é escrita em *JSON* da mesma maneira que foi feita a criação (*create*) e o código identifica qual usuário está sendo atualizado a partir do *ID*
- ❖ O código lê os dados *JSON* e atualiza; depois ele salva as alterações (*commit*)
- ❖ Caso o *update* seja um sucesso, com o *make_response* é impressa uma resposta na tela e no terminal o *status* 200
- ❖ Usuário não encontrado (*id* não encontrado), mensagem de erro + *status* 404 no terminal
- ❖ Outros erros: mensagem de erro + *status* 500

```
flask_app | 172.18.0.1 - - [18/Apr/2025 17:53:23] "PUT /update-usuarios/1 HTTP/1.1" 200 -
```

localhost:5000/create-usuarios

POST localhost:5000/create-usuarios

Body

```
1 {
2   "usuario": "isabella",
3   "email": "isabella.f.hedex@gmail.com"
4 }
```

ENTRADA

201 CREATED • 44 ms • 293 B

Body

```
1 {
2   "mensagem": "usuário criado com sucesso",
3   "user": {
4     "email": "isabella.f.hedex@gmail.com",
5     "id": 1,
6     "usuario": "isabella"
7   }
8 }
```

SAÍDA

localhost:5000/update-usuarios/1

PUT localhost:5000/update-usuarios/1

Body

```
1 {
2   "usuario": "bella",
3   "email": "isabella@gmail.com"
4 }
```

ENTRADA

200 OK • 18 ms • 281 B

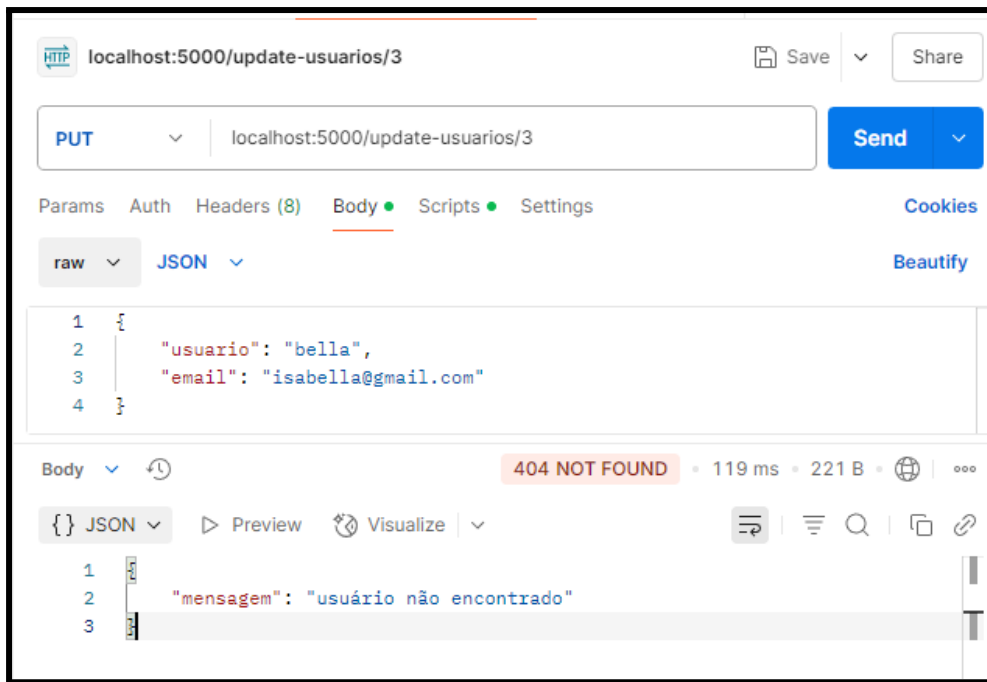
Body

```
1 {
2   "mensagem": "usuário atualizado com sucesso",
3   "user": {
4     "email": "isabella@gmail.com",
5     "id": 1,
6     "usuario": "bella"
7   }
8 }
```

SAÍDA

❖ FORÇANDO MENSAGEM DE ERRO (ID NÃO EXISTENTE):

flask_app | 172.18.0.1 - - [20/Apr/2025 04:06:07] "PUT /update-usuarios/3 HTTP/1.1" 404 -



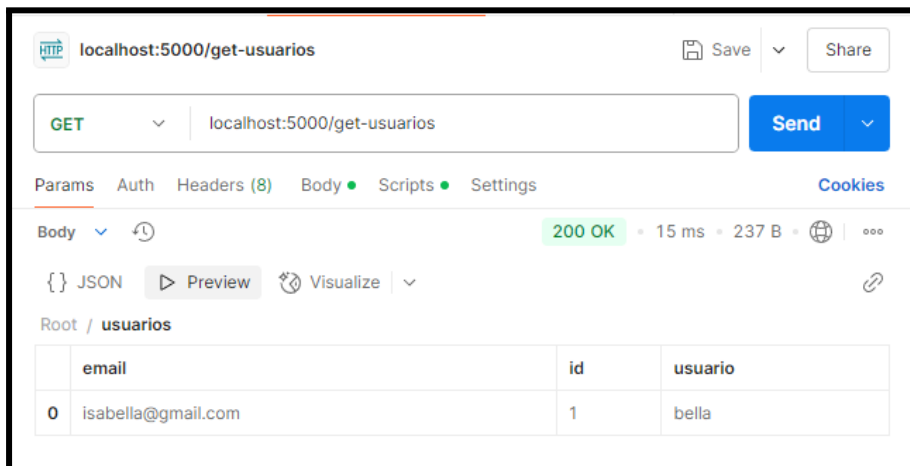
SERVIÇO DELETAR USUÁRIO:

```
# DELETANDO USUÁRIO
@app.route('/delete-usuarios/<int:id>', methods=['DELETE']) # rota para deletar um usuário pelo id, usando o método DELETE
def deletar_usuario(id):
    try:
        usuario = User.query.filter_by(id=id).first() # pega o usuário pelo id
        if usuario:
            db.session.delete(usuario) # deleta o usuário
            db.session.commit() # salva as alterações no banco de dados
            return make_response(jsonify({'mensagem': 'usuário deletado com sucesso'}), 200) # retorna em json com o status 200 (ok)
        return make_response(jsonify({'mensagem': 'usuário não encontrado'}), 404) # se o usuário não for encontrado, retorna erro 404 (não encontrado)
    except Exception as e:
        print(e)
        return make_response(jsonify({'mensagem': 'erro ao deletar usuário'}), 500)
```

- ❖ Cria a rota “/delete-usuarios/id”, utilizando o método *DELETE* para apagar o usuário que tenha o *id* citado
- ❖ Busca um usuário filtrado pelo seu *id*
- ❖ Se for um usuário: deleta as informações do usuário, salva a alteração e no terminal retorna o *status* 200
- ❖ Usuário não encontrado (*id* não encontrado), mensagem de erro + *status* 404 no terminal
- ❖ Outros erros: mensagem de erro + *status* 500



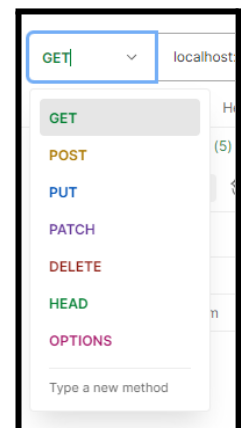
```
flask_app | 172.18.0.1 - - [20/Apr/2025 04:10:24] "DELETE /delete-usuarios/2 HTTP/1.1" 200 -
```



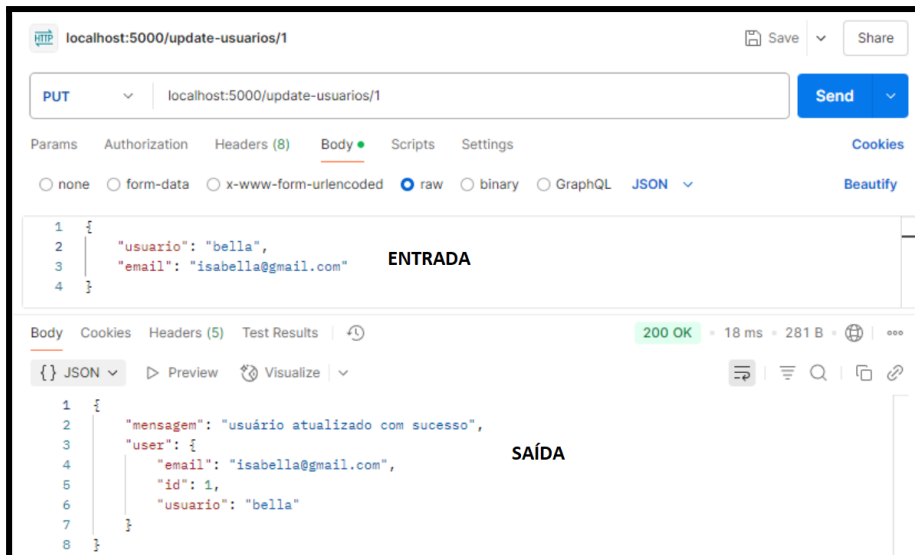
POSTMAN:

É DE **EXTREMA IMPORTÂNCIA** QUE O MÉTODO DE REQUISIÇÃO (GET, PUT, POST, DELETE) ESTEJA CORRETO, SE NÃO, NÃO VAI FUNCIONAR!!!

- ❖ *Postman* foi a plataforma utilizada por mim para testar a minha resolução do CP2; Praticamente tudo foi testado no *Postman*;
- ❖ PS: a rota de teste foi feita no próprio navegador
- ❖ Ele, como o nome diz, funciona como um carteiro: cuida das informações, recebe e traz elas para você.
- ❖ O postman funciona a partir de métodos que você escolhe (os que eu utilizei foram *GET*, *PUT*, *POST* e *DELETE*);
 - *GET*: serve para pegar dados e mostrá-los na tela (meu serviço *get* e *read*)
 - *PUT*: serve para inserir dados novos em dados já existentes (meu serviço *update*)
 - *POST*: serve para a criação de dados novos (meu serviço *create*)
 - *DELETE*: deleção de dados (meu serviço *delete*)



IMPORTANTE: nos métodos *POST* e *PUT*, como anteriormente mostrado neste documento, é necessário a inscrição de dados no ambiente de entrada;



DOCKERFILE:

```
Dockerfile > ...
1 FROM python:3.10-slim-buster
2
3 # escolhendo o diretório de trabalho
4 WORKDIR /app
5
6 COPY requirements.txt .
7
8 # instalando as dependências
9 RUN pip install -r requirements.txt
10
11 # copiar o resto do código
12 COPY . .
13
14 # escolhendo a porta que o flask vai rodar
15 EXPOSE 5000
16
17 # rodando o flask
18 CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
19
```

Nesse arquivo:

- A imagem foi definida (*FROM*) baseada na imagem oficial do *Python* (a que eu usei é *slim-buster*, otimizada para ser mais leve e inclui o *python 3.10*)
- *WORKDIR* define o diretório de trabalho (dentro do contêiner) como */app* (todos os comandos depois disso vão ser executados nesse diretório)
- *COPY*: copia o arquivo de requerimentos (local) pra dentro do */app*
- *RUN*: faz a instalação das dependências listadas dentro do arquivo de requerimentos
- *COPY . .*: copia todos os arquivos do local para o de trabalho */app*
- *EXPOSE*: deixa claro que o contêiner vai usar a porta 5000 pra fazer a comunicação (mas aqui ainda não faz a conexão com o *host* - isso é feito no *.yaml*)
- *CMD*: aqui é definido o comando que vai ser executado quando iniciar o contêiner (*flask run*: inicia o *Flask*; - *--host=0.0.0.0*: fica acessível externamente; - *--port=5000*: define a porta onde o *Flask* vai rodar)

run -d --help for more information

```
PS C:\Users\isabe\OneDrive\Documents\flask-app> docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
da5f28de6d72	isabella/flask_app_img:1.0.0	"flask run --host=0..."	23 minutes ago	Exited (137) 7 minutes ago	0.0.0.0:5000->5000	flask_app

DOCKER-COMPOSE.YML:

```

>Run All Services
services:
  >Run Service
  flask_app: # aplicação flask
    container_name: flask_app # nome do container
    image: isabella/flask_app_img:1.0.0 # imagem usada

    build: . # imagem vai ser criada localmente nesse diretório (que tem o dockerfile)
    ports: # mapeia as portas container:host
      - "5000:5000"

    environment: # define variáveis de ambiente
      - DB_URL=postgresql://postgres:password@flask_db:5432/postgres
    depends_on:
      - flask_db # flask_app depende do db, iniciando primeiro o flask_db e dps o app

  >Run Service
  flask_db: # banco de dados postgres
    container_name: flask_db
    image: postgres # imagem do postgres
    ports:
      - "5432:5432"

    environment:
      - POSTGRES_USER=postgres # usuário
      - POSTGRES_PASSWORD=postgres # senha
      - POSTGRES_DB=postgres # nome do db

    volumes:
      - ./cria-tabela-usuarios.sql:/docker-entrypoint-initdb.d/cria-tabela-usuarios.sql
      - pgdata:/var/lib/postgresql/data
volumes:
  pgdata: {}

```

FLUXO:

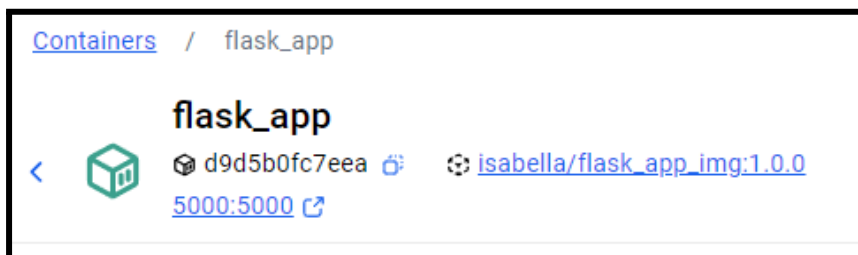
- ❖ o *flask_db* é iniciado primeiro, configurando o *postgreSQL* com as credenciais
- ❖ os dados são armazenados no volume *pgdata*
- ❖ o *flask_app* só é iniciado depois do *flask_db*, por conta do *depends_on*
- ❖ o *flask_app* usa o *DB_URL* para se conectar no *postgres* (*flask_db*)
- ❖ a api *flask* vai poder ser acessada na URL *http://localhost:5000*
- ❖ o bd *PostgreSQL* vai estar disponível na porta 5432

CÓDIGO:

- ❖ *flask_app* e *flask_db* são os dois serviços que vão ser executados no *docker compose*

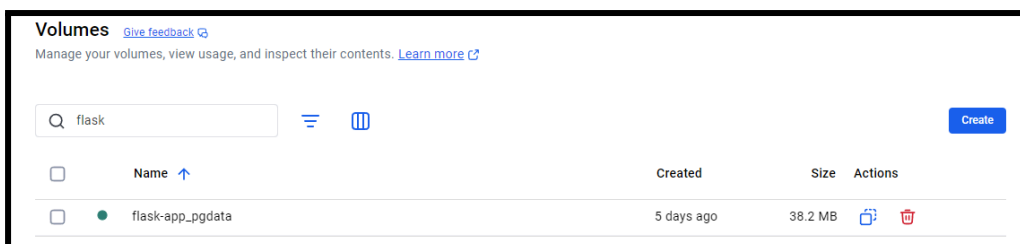
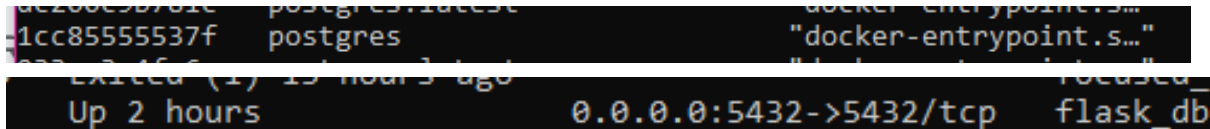
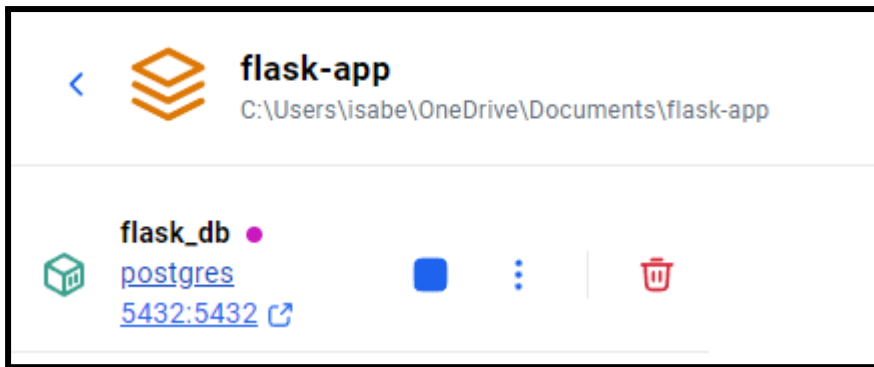
FLASK_APP:

- ❖ representa a aplicação Flask
- ❖ a imagem vai ser criada localmente a partir do diretório que está (.) - que tem o arquivo *dockerfile*
- ❖ *ports*: aqui cria a conexão da porta do contêiner (5000) com a porta do *host* (5000), que faz com que a api possa ser acessada pelo navegador ou pelo *postman* (acesso externo)
- ❖ as variáveis de ambiente foram definidas (*DB_URL* = faz conexão com *postgreSQL*):
`postgresql://usuario:senha@host:porta/bd`
`DB_URL=postgresql://postgres:password@flask_db:5432/postgres`
- ❖ *depends_on*: faz com que o *flask_db* seja iniciado antes do *flask_app*

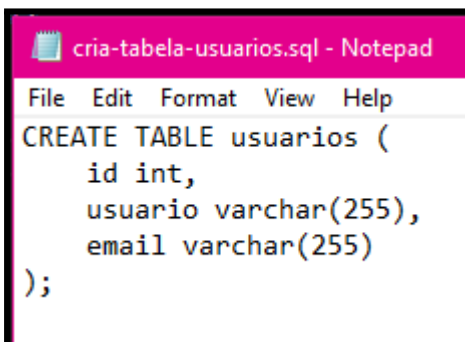


FLASK_DB:

- ❖ serviço que representa o bd *PostgreSQL*
- ❖ a imagem do contêiner é a oficial do *postgres*
- ❖ *ports*: faz a conexão da porta do contêiner (5432) para a porta do *host* (5432)
- ❖ variáveis de ambiente (credenciais): *POSTGRES_USER* (nome do usuário administrador do bd), *POSTGRES_PASSWORD* (senha desse usuário) e *POSTGRES_DB* (nome do bd que é criado automaticamente)
- ❖ volumes:
- ❖ conexão do banco de dados com o *script SQL* para fazer com que o *postgres* inicialize executando automaticamente o arquivo *.sql* (coloco o arquivo local *cria-tabela-usuarios.sql* dentro do diretório */docker-entrypoint-initdb.d/* - que tá dentro do contêiner)
- ❖ já na segunda linha eu crio o volume *pgdata* para que os dados do bd fiquem salvos mesmo que o contêiner seja removido



CRIA-TABELA-USUARIOS.SQL:



Este arquivo:

- ❖ Tem um comando SQL que cria uma tabela chamada usuarios
- ❖ Cada linha representa uma coluna (id, usuario, *email*)
- ❖ Cada coluna armazena um tipo de dado: *id* - *integer* (inteiros), usuario e *email* - *varchar* (*string* variável, com um máximo de 255 caracteres)
- ❖ PORQUE QUE ESSE *CREATE TABLE* NÃO SE CHOCA COM O *CREATE_ALL* DO *APP.PY* (levando em consideração que ambas as tabelas se chamam usuarios)?
 - O *create_all* confere se já existe uma tabela com esse nome. Caso já exista ele não faz a criação novamente.

- ❖ Para saber se este arquivo rodou da maneira correta, eu tinha que entrar no *postgres* e descobrir se havia uma tabela dentro (para isso utilizei o CMD):

```
C:\Users\isabe>docker exec -it flask_db bash
root@1cc8555537f:/# psql -U postgres -d postgres
psql (17.4 (Debian 17.4-1.pgdg120+2))
Type "help" for help.

postgres=# \dt
          List of relations
Schema | Name   | Type  | Owner
-----+-----+-----+-----
public | usuarios | table | postgres
(1 row)
```

- ❖ *-it* faz com que tenha interação com o terminal do contêiner
- ❖ *bash* cria um terminal *Bash* dentro do contêiner
- ❖ *psql* faz com que eu consiga interagir diretamente com o *postgres*
- ❖ *-u* é o nome do usuário (*postgres*) definido no *.yaml*
- ❖ *-d* é o banco de dados que eu quero me conectar (*postgres*) definido no *.yaml*
- ❖ *\dt* lista todas as tabelas no bd

```
postgres=# \dt+ usuarios
          List of relations
Schema | Name   | Type  | Owner   | Persistence | Access method | Size  | Description
-----+-----+-----+-----+-----+-----+-----+-----
public | usuarios | table | postgres | permanent   | heap           | 8192 bytes |
(1 row)

postgres=# SELECT * FROM usuarios;
 id | usuario | email
-----+-----+-----
  1 | bella   | isabella@gmail.com
(1 row)
```

- ❖ *\dt+ usuarios* = mostra as informações sobre a tabela usuarios (inclui tamanho da tabela, persistência - que foi feito no volume *pgdata* etc.)
- ❖ *SELECT **: comando SQL para selecionar todas as colunas
- ❖ *FROM usuarios*: especifica qual a tabela

ANTES DE FAZER QUALQUER TESTE E EXECUTAR QUALQUER ROTA:

COM O DOCKER DESKTOP ABERTO

```
PS C:\Users\isabe\OneDrive\Documents\flask-app> docker compose up -d flask_db
time="2025-04-15T14:43:20-03:00" level=warning msg="C:\\Users\\isabe\\OneDrive\\Documents\\flask-app\\docker-compose.yml: the a
ed, please remove it to avoid potential confusion"
[+] Running 15/15
  ✓ flask_db Pulled                                     31.8s
[+] Running 3/3
  ✓ Network flask-app_default Created                   0.2s
  ✓ Volume "flask-app_pgdata" Created                  0.0s
  ✓ Container flask_db Started                         3.2s
PS C:\Users\isabe\OneDrive\Documents\flask-app>
```

```

PS C:\Users\isabe\OneDrive\Documents\flask-app> docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
acb966d6bcd4   postgres      "docker-entrypoint.s..." 2 minutes ago  Up 2 minutes  0.0.0.0:5432->5432/tcp   flask_db
ec85242043db   isabella      "python execdockerfl..." 3 days ago    Exited (0) 3 days ago                                practical_swirles
PS C:\Users\isabe\OneDrive\Documents\flask-app>

```

```

PS C:\Users\isabe\OneDrive\Documents\flask-app> docker compose build
Compose can now delegate builds to bake for better performance.
To do so, set COMPOSE_BAKE=true.
[*] Building 35.3s (11/11) FINISHED
-> [flask_app internal] load build definition from Dockerfile
-> => transferring dockerfile: 395B
-> [flask_app internal] load metadata for docker.io/library/python:3.10-slim-buster
-> [flask_app internal] load .dockerignore
-> => transferring context: 2B
-> [flask_app 1/5] FROM docker.io/library/python:3.10-slim-buster@sha256:37aa274c2d001f09b14828450d903c55f821c90f225fdd08c5180fcca77b3f
-> => resolve docker.io/library/python:3.10-slim-buster@sha256:37aa274c2d001f09b14828450d903c55f821c90f225fdd08c5180fcca77b3f
-> => sha256:6e7d39ff1535e0249529c91607c3e93f107c500b603a0ad75e90c85e4cf43b6 3.37MB / 3.37MB
-> => sha256:b2697057e25fe078e5f17bdb522b6a913fc3ce2a041d34d2e4771e0f3ddd3c3f 242B / 242B
-> => sha256:6903bcecb9721c4ee87e4188b263a0a392a16b2e2813c7a0ed0f9c4416194734 11.50MB / 11.50MB
-> => sha256:824416e234237961c9c5d4f41dfe5b295a3c35a671ee52889bf08d8e257ec4c 2.78MB / 2.78MB
-> => sha256:8b91b88d557765cd8c680266875a3f6dc4337b6ce15a17e4857139e5fc964f3 27.14MB / 27.14MB
-> => extracting sha256:8b91b88d557765cd8c680266875a3f6dc4337b6ce15a17e4857139e5fc964f3
-> => extracting sha256:824416e234237961c9c5d4f41dfe5b295a3c35a671ee52889bf08d8e257ec4c
-> => extracting sha256:6903bcecb9721c4ee87e4188b263a0a392a16b2e2813c7a0ed0f9c4416194734
-> => extracting sha256:b2697057e25fe078e5f17bdb522b6a913fc3ce2a041d34d2e4771e0f3ddd3c3f
-> => extracting sha256:6e7d39ff1535e0249529c91607c3e93f107c500b603a0ad75e90c85e4cf43b6
-> [flask_app internal] load build context
-> => transferring context: 42.64MB
-> [flask_app 2/5] WORKDIR /app
-> [flask_app 3/5] COPY requirements.txt .
-> [flask_app 4/5] RUN pip install -r requirements.txt
-> [flask_app 5/5] COPY . .
-> [flask_app] exporting to image
-> => exporting layers
-> => exporting manifest sha256:df76e7e496366f3a2b788b5deb5c0281375dd3d701c1a1067f8aacde851403f
-> => exporting config sha256:a18ac2af496953ea8eddf8eeeee57ae7c4651f0063f06c9c9623c31211d13ef
-> => exporting attestation manifest sha256:e207b6a1b0e94c0be2fc885fb59f6282baf8bb307e5e9df4d6dd50e45bb18b7
-> => exporting manifest list sha256:e1e71c48720a1e04e986d57684c049a47fa7c7d7ce45727f7ceb0af14119e6d5
-> => naming to docker.io/isabella/flask_app_img:1.0.0
-> => unpacking to docker.io/isabella/flask_app_img:1.0.0
-> [flask_app] resolving provenance for metadata file
[*] Building 1/1
✓ flask_app Built
PS C:\Users\isabe\OneDrive\Documents\flask-app>

```

```

PS C:\Users\isabe\OneDrive\Documents\flask-app>
PS C:\Users\isabe\OneDrive\Documents\flask-app> docker compose up --build flask_app
Compose can now delegate builds to bake for better performance.
To do so, set COMPOSE_BAKE=true.
[*] Building 10.6s (11/11) FINISHED
-> [flask_app internal] load build definition from Dockerfile
-> => transferring dockerfile: 397B
-> [flask_app internal] load metadata for docker.io/library/python:3.10-slim-buster
-> [flask_app internal] load .dockerignore
-> => transferring context: 2B
-> [flask_app 1/5] FROM docker.io/library/python:3.10-slim-buster@sha256:37aa274c2d001f09b14828450d903c55f821c90f225fdd08c5180fcca77b3f
-> => resolve docker.io/library/python:3.10-slim-buster@sha256:37aa274c2d001f09b14828450d903c55f821c90f225fdd08c5180fcca77b3f
-> [flask_app internal] load build context
-> => transferring context: 181.82kB
-> CACHED [flask_app 2/5] WORKDIR /app
-> CACHED [flask_app 3/5] COPY requirements.txt .
-> CACHED [flask_app 4/5] RUN pip install -r requirements.txt
-> [flask_app 5/5] COPY . .
-> [flask_app] exporting to image
-> => exporting layers
-> => exporting manifest sha256:c29b739e36f68d7b9f532e84cc164c3a2a2966cd5a394f5e79be8fbc918ff
-> => exporting config sha256:d41a546b2024f092923180a114a2652483625be1f51c75c99bb4c9d8036697be
-> => exporting attestation manifest sha256:1b1d73687d99a6c529bd14bf15f16728b44f4d812dc6ced0638fc42db7fd7c69
-> => exporting manifest list sha256:66a7228926911a2ce9bbf693b6579d254ee662d9ea9d00f7aa35e7db532ce7b
-> => naming to docker.io/isabella/flask_app_img:1.0.0
-> => unpacking to docker.io/isabella/flask_app_img:1.0.0
-> [flask_app] resolving provenance for metadata file
[*] Running 3/3
✓ flask_app Built
✓ Container flask_db Running
✓ Container flask_app Recreated
Attaching to flask_app
flask_app | * Debug mode: off
flask_app | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
flask_app | * Running on all addresses (0.0.0.0)
flask_app | * Running on http://127.0.0.1:5000
flask_app | * Running on http://172.18.0.3:5000
flask_app | Press CTRL+C to quit
flask_app | 172.18.0.1 - - [15/Apr/2025 18:58:14] "GET /teste HTTP/1.1" 200 -
flask_app | 172.18.0.1 - - [15/Apr/2025 18:58:14] "GET /favicon.ico HTTP/1.1" 404 -

```

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage ⓘ

0.04% / 400% (4 CPUs available)

Container memory usage ⓘ

83.67MB / 3.73GB

[Show charts](#)

Q Search



☒ Only show running containers

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	flask-app	-	-	-	0.04%	2 hours ago	
<input type="checkbox"/>	flask_db	1cc85555537f	postgres	5432:5432	0.01%	2 hours ago	
<input type="checkbox"/>	flask_app	d9d5b0fc7eea	isabella/flask_app_img:1.0.0	5000:5000	0.03%	2 hours ago	

Images [Give feedback](#)

View and manage your local and Docker Hub images. [Learn more](#)

Local Docker Hub repositories

1.96 GB / 3.7 GB in use 4 Images

Last refresh: 2 hours ago

Q Search



<input type="checkbox"/>	Name	Tag	Image ID	Created	Size	Actions
<input type="checkbox"/>	isabella	latest	f3570f53505c	9 days ago	1.71 GB	
<input type="checkbox"/>	postgres	latest	fe3f571d128e	2 months ago	620.68 MB	
<input type="checkbox"/>	isabella/flask_app_img	1.0.0	179c02f3879e	15 hours ago	316.2 MB	

```
PS C:\Users\isabe\OneDrive\Documents\flask-app>
PS C:\Users\isabe\OneDrive\Documents\flask-app> docker compose up
[+] Running 2/2
 ✓ Container flask_db    Running
 ✓ Container flask_app   Created
```