# CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

## DEPARTAMENTO DE COMPUTAÇÃO


**DOCKERIZAÇÃO DE APLICAÇÃO MULTICAMADA PARA TREINAMENTO PERSONALIZADO COM IA**
**DISCIPLINA: DEVOPS**
**PROFESSOR: DELANO MEDEIROS BEDER**


**EDUARDO HENRIQUE SPINELLI – RA 800220**



**SÃO CARLOS – SP**
**2025**

## 1. Introdução

Este projeto tem como objetivo aplicar os conceitos de conteinerização com Docker na disciplina de DevOps, transformando uma aplicação multicamada real em uma estrutura baseada em containers. A aplicação escolhida foi a *PlannerRun*, uma plataforma para geração de planos personalizados de corrida utilizando inteligência artificial.

A solução foi dividida em três contêineres distintos:

- **Backend**: API Flask responsável por manipulação de dados, integração com Stripe e envio de e-mails.
- **Frontend**: Aplicação Next.js para interação com o usuário.
- **Banco de Dados**: PostgreSQL com script de inicialização automatizado.

---

## 2. Visão Geral da Aplicação

A *PlannerRun* é uma aplicação web que permite a usuários informarem seus dados pessoais (altura, peso, objetivo de treino, etc.) e recebem um plano de treino adaptado, com base em lógica de IA. O sistema também realiza integração com o serviço de pagamentos Stripe e armazena dados em PostgreSQL.

**Funcionalidades:**

- Cadastro de cliente e preferências de treino.
- Integração com Stripe para checkout.
- Consulta da contagem total de clientes.
- Envio automático de e-mail pós-pagamento.
- Interface web responsiva construída em Next.js.

---

## 3. Estrutura de Containers

### 3.1 Backend (Flask)

- Baseado em `python:3.10-slim`

- Executa `app.py`
- Configurado para expor a porta 5000
- Habilitado para CORS (inclusive via container)
- Comunica-se com o banco de dados por nome de host `db`

#### 3.2 Frontend (Next.js)

- Baseado em `node:18`
- Executa `npm run dev` na porta 3000
- Recebe a variável de ambiente `NEXT_PUBLIC_API_URL=http://backend:5000/api`
- Realiza chamadas para a API Flask para salvar dados e recuperar métricas

#### 3.3 Banco de Dados (PostgreSQL)

- Imagem baseada em `postgres:15`
- Executa script `init.sql` para criação automática da tabela `clientes`
- Utiliza volume `db_data` persistente para manter dados entre execuções

---

## 4. Dockerfiles e docker-compose

**Arquivos obrigatórios:**

- `Dockerfile` do backend
- `Dockerfile` do frontend
- `Dockerfile` do banco (utiliza `COPY init.sql`)
- `docker-compose.yml` responsável por orquestrar os serviços

**docker-compose.yml (trecho):**

```
Unset
volumes:
  db_data:

services:
  db:
    ...
    volumes:
      - db_data:/var/lib/postgresql/data
```

---

## 5. Execução da aplicação

**Requisitos:**

- Docker e Docker Compose instalados

**Passos:**

```
Shell
git clone <repo>
cd DevOps
docker-compose up --build
```

Acesse:

- Frontend: http://localhost:3000
- Backend: http://localhost:5000/api

---

## 6. Observações Técnicas

- O volume `db_data` impede perda de dados no banco ao reiniciar containers.
- A variável de ambiente `NEXT_PUBLIC_API_URL` é usada no frontend via `process.env`.
- Foi necessário ajustar CORS no backend para permitir comunicações entre containers.

## 7. Conclusão

O trabalho evidenciou na prática a importância da conteinerização em ambientes DevOps modernos. A separação dos serviços em containers independentes facilita o deploy, escalabilidade e manutenção da aplicação, atendendo aos princípios de microserviços.