



## Atividade Prática: Criando um To-Do List Monolítico

 **Objetivo:** Criar um sistema simples de cadastro de tarefas com funcionalidades de **adicionar, editar, excluir e marcar como concluído**.

 **Tecnologias recomendadas (depende do nível dos alunos)**

- **Stack básica:** HTML, CSS, JavaScript (para frontend) + Python (Flask) ou Node.js (Express) para o backend.
- **Banco de dados:** SQLite (simples e não precisa de configuração extra).

---

## Passo 1: Configurar o Ambiente

Antes de começar, os alunos devem garantir que têm o **Python** instalado. Eles podem verificar isso com:

```
python --version
```

Se Python estiver instalado, deve retornar algo como:

```
Python 3.x.x
```

Caso contrário, o Python pode ser baixado em: <https://www.python.org/downloads/>.

---

## Passo 2: Criar um Ambiente Virtual

É uma boa prática criar um **ambiente virtual** para o projeto. Isso evita conflitos entre bibliotecas. Execute os comandos:

```
# Criar um ambiente virtual
python -m venv .venv

# Ativar o ambiente virtual (Windows)
.venv\Scripts\activate

# Ativar o ambiente virtual (Linux/Mac)
source .venv/bin/activate
```

O terminal mostrará algo como `(.venv)` indicando que o ambiente virtual está ativo.

---

## Passo 3: Instalar as Dependências

Dentro do ambiente virtual, instale o Flask:

```
pip install flask
```

Opcionalmente, se quiser salvar as dependências em um arquivo `requirements.txt`, use:

```
pip freeze > requirements.txt
```

E para instalar a partir dele em outro ambiente:

```
pip install -r requirements.txt
```

---

## Passo 4: Criar o Banco de Dados

Crie um script para definir a estrutura do banco.

Crie o arquivo `init_db.py` e adicione:

```
import sqlite3

con = sqlite3.connect("database.db")
cur = con.cursor()
cur.execute("""
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    task TEXT NOT NULL,
    completed INTEGER DEFAULT 0
)
""")
con.commit()
con.close()

print("Banco de dados criado com sucesso!")
```

Agora execute:

```
python init_db.py
```

Isso criará um arquivo `database.db` com a tabela `tasks`.

---

## Passo 5: Criar a Aplicação Flask

Agora, crie o arquivo `app.py` com o seguinte código:

```
from flask import Flask, render_template, request, redirect
import socket, os

app = Flask(__name__)

import sqlite3

def connect_db():
    return sqlite3.connect("database.db")

@app.route('/oi')
def hello():
    hostname = socket.gethostname()
    port = os.environ.get('PORT', '5000') # Pega a porta do ambiente ou
    usa 5000 como padrão
    return f"Hello from {hostname} on port {port}!\n"

@app.route('/')
def index():
    con = connect_db()
    cur = con.cursor()
    cur.execute("SELECT * FROM tasks")
    tasks = cur.fetchall()
    con.close()
    return render_template("index.html", tasks=tasks)

@app.route('/add', methods=['POST'])
def add_task():
    task = request.form['task']
    con = connect_db()
    cur = con.cursor()
    cur.execute("INSERT INTO tasks (task, completed) VALUES (?, ?)", (task,
0))
    con.commit()
    con.close()
    return redirect('/')

@app.route('/delete/<int:id>')
def delete_task(id):
    con = connect_db()
    cur = con.cursor()
    cur.execute("DELETE FROM tasks WHERE id=?", (id,))
    con.commit()
    con.close()
    return redirect('/')

@app.route('/complete/<int:id>')
def complete_task(id):
    con = connect_db()
    cur = con.cursor()
    cur.execute("UPDATE tasks SET completed = 1 WHERE id=?", (id,))
    con.commit()
```

```
con.close()
return redirect('/')

if __name__ == "__main__":
    app.run(debug=True)
```

Esse código cria uma API simples com as seguintes rotas:

- / → Exibe a lista de tarefas.
- /add → Adiciona uma nova tarefa.
- /delete/<id> → Exclui uma tarefa.
- /complete/<id> → Marca uma tarefa como concluída.

---

## Passo 6: Criar o Frontend

Crie uma pasta chamada `templates` e dentro dela um arquivo `index.html`:

```
<!DOCTYPE html>
<html lang="pt">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Lista de Tarefas</title>
  <style>
    body { font-family: Arial, sans-serif; max-width: 500px; margin:
auto; text-align: center; }
    ul { list-style: none; padding: 0; }
    li { padding: 10px; border: 1px solid #ddd; margin: 5px 0; display:
flex; justify-content: space-between; }
    .completed { text-decoration: line-through; color: gray; }
  </style>
</head>
<body>
  <h1>Lista de Tarefas</h1>
  <form action="/add" method="POST">
    <input type="text" name="task" required>
    <button type="submit">Adicionar</button>
  </form>
  <ul>
    {% for task in tasks %}
      <li class="{% if task[2] == 1 %}completed{% endif %}">
        {{ task[1] }}
        <a href="/complete/{{ task[0] }}">✓</a>
        <a href="/delete/{{ task[0] }}">✗</a>
      </li>
    {% endfor %}
  </ul>
</body>
</html>
```

Esse frontend:

- ✓ Exibe as tarefas
- ✓ Permite adicionar novas
- ✓ Permite marcar como concluídas
- ✓ Permite excluir

---

## Passo 7: Executar a Aplicação

Para rodar o servidor Flask, execute:

```
python app.py
```

Agora acesse no navegador:

👉 <http://127.0.0.1:5000/>

---

Para escalar sua aplicação Flask monolítica, você pode adotar estratégias que melhorem o desempenho e permitam que mais usuários acessem simultaneamente. Vou dividir as soluções em **escalabilidade vertical** e **escalabilidade horizontal**, além de outras otimizações.

---

## 1 Escalabilidade Vertical (Aumentar Recursos do Servidor)

---

A escalabilidade **vertical** significa **melhorar o servidor** onde sua aplicação está rodando. Isso inclui:

- ✓ **Usar um servidor mais potente** (mais CPU, RAM)
- ✓ **Usar um banco de dados externo** (ex: PostgreSQL no RDS da AWS)
- ✓ **Configurar um WSGI mais eficiente**, como **Gunicorn**

### Usando Gunicorn

WSGI (Web Server Gateway Interface) é um padrão para servidores web e aplicações Python se comunicarem. **Gunicorn** é um servidor WSGI que pode melhorar a performance do Flask. Usar o Gunicorn traz os seguintes benefícios:

- ✓ Diferente do servidor de desenvolvimento do Flask, que processa apenas uma requisição por vez, o Gunicorn pode lidar com várias conexões simultâneas, tornando a aplicação mais eficiente.
- ✓ O Gunicorn pode ser executado atrás de um proxy reverso como o Nginx ou Apache, que pode servir arquivos estáticos e lidar com tarefas de balanceamento de carga.

Em produção, ao invés de rodar `python app.py`, use **Gunicorn** para melhorar a performance:

```
pip install gunicorn
gunicorn -w 4 -b 0.0.0.0:5000 app:app
```

Isso inicia 4 "workers", permitindo que várias requisições sejam processadas ao mesmo tempo.

- ✓ Se houver mais de 4 requisições simultâneas, elas entram na fila e esperam um worker ficar disponível.
- ✓ Se o servidor tiver mais CPU/RAM, você pode aumentar o número de workers.
- ✓ Se o número de requisições for muito alto e os workers demorarem para processar, a fila pode ficar sobrecarregada, causando lentidão ou erros 502/504 (Bad Gateway, Timeout).
- ✓ O Nginx pode ajudar a gerenciar conexões e servir arquivos estáticos, reduzindo a carga do Gunicorn.

Se a aplicação faz muitas operações de entrada e saída (consultas SQL, chamadas HTTP externas), aumentar os workers melhora o desempenho. Fórmula Geral:  $2 \times \text{CPUs} + 1$  (ex: 4 CPUs → 9 workers).

---

## 2 Escalabilidade Horizontal (Múltiplas Instâncias)

---

A escalabilidade **horizontal** significa rodar várias cópias da aplicação para distribuir o tráfego.

### Usando Load Balancer

Se o tráfego aumentar, você pode rodar **múltiplas instâncias** e usar um **Load Balancer** para distribuir as requisições.

- ♦ No **Railway, Render ou Heroku**, basta aumentar as "instâncias" na configuração do serviço.
- ♦ Se estiver em um **VPS (AWS, DigitalOcean, Brdrive)**, pode usar o **NGINX** como proxy reverso.

Exemplo de configuração NGINX para distribuir o tráfego entre 2 instâncias Flask:

```
upstream flask_app {  
    server 127.0.0.1:5000;  
    server 127.0.0.1:5001;  
}  
  
server {  
    listen 80;  
    location / {  
        proxy_pass http://flask_app;  
    }  
}
```

Isso envia requisições para múltiplas instâncias Flask rodando nas portas **5000** e **5001**.

---

## 3 Usando Containers (Docker e Kubernetes)

---

Outra maneira de escalar é usar **Docker** e **Kubernetes** para gerenciar múltiplas réplicas.

### Criando um Dockerfile

Crie um arquivo **Dockerfile** na raiz do projeto:

```
# Usa a versão leve do Python (Alpine)
FROM python:3.9-alpine
# Define o diretório de trabalho dentro do contêiner
WORKDIR /app
# Copia os arquivos necessários para o contêiner
COPY . .
# Instala as dependências necessárias (usa --no-cache para evitar arquivos desnecessários)
RUN pip install --no-cache-dir -r requirements.txt
# Expõe a porta 5000 para acesso externo
EXPOSE 5000
# Comando para iniciar a aplicação usando Gunicorn
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:5000", "app:app"]
```

Agora, **crie e rode o container**:

```
docker build -t flask-app .
docker run -p 5000:5000 flask-app
```

Se quiser escalar com **Kubernetes**, pode usar **kubectl scale** para rodar **múltiplas cópias** do container.

---

## 4 Banco de Dados Externo

---

Usar **SQLite** em produção não é recomendado. Para escalar, use um banco como:

- ✓ **PostgreSQL** (ex: AWS RDS, Supabase, Railway, Render)
- ✓ **MySQL** (ex: PlanetScale)

### Exemplo de Conexão PostgreSQL

1 Instale o driver:

```
pip install psycopg2
```

2 Atualize **connect\_db()** no **app.py**:

```
import psycopg2
def connect_db():
    return psycopg2.connect(
        dbname="meubanco",
        user="usuario",
        password="senha",
```

```
        host="servidor.externo.com",  
        port=5432  
    )
```

Isso melhora a escalabilidade, pois várias instâncias Flask podem acessar o mesmo banco.

---

## 5 Cache para Melhorar Performance

---

Usar um **cache** evita que consultas repetidas sobrecarreguem o banco.

✓ **Redis**: Ótimo para armazenar resultados de consultas frequentes.

✓ **Memcached**: Boa opção para melhorar tempos de resposta.

### Usando Redis

Instale a biblioteca Python:

```
pip install redis
```

No `app.py`, adicione um cache:

```
import redis  
cache = redis.Redis(host='localhost', port=6379, db=0)  
  
@app.route('/')  
def index():  
    tasks = cache.get('tasks')  
    if not tasks:  
        con = connect_db()  
        cur = con.cursor()  
        cur.execute("SELECT * FROM tasks")  
        tasks = cur.fetchall()  
        con.close()  
        cache.set('tasks', str(tasks), ex=30) # Expira em 30s  
    return render_template("index.html", tasks=eval(tasks))
```

Isso reduz a carga no banco de dados.

---

## 6 Otimizações de Código

---

Vamos fazer mudanças para testar a escalabilidade da aplicação.

Database Init



```
mkdir data
```

```
import sqlite3

con = sqlite3.connect("data/database.db")
cur = con.cursor()
cur.execute("""
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    task TEXT NOT NULL,
    completed INTEGER DEFAULT 0
)
""")
con.commit()
con.close()

print("Banco de dados criado com sucesso!")
```

## app.py

```
from flask import Flask, render_template, request, redirect
import socket
import os
import logging
import sqlite3

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@app.before_request
def log_request_info():
    hostname = socket.gethostname()
    port = os.environ.get('PORT', '5000') # Pega a porta do ambiente ou
    usa 5000 como padrão
    logger.info(f"Requisição recebida em {hostname} na porta {port}")

def connect_db():
    # return sqlite3.connect("database.db")
    db_path = '/app/db/database.db'
    conn = sqlite3.connect(db_path)
    conn.row_factory = sqlite3.Row
    return conn

@app.route('/env')
def env():
    return str(os.environ)
```

```
@app.route('/oi')
def hello():
    hostname = socket.gethostname()
    port = os.environ.get('PORT', '5000') # Pega a porta do ambiente ou
    usa 5000 como padrão
    logger.info(f"Requisição recebida em {hostname} na porta {port}")
    return f"Hello from {hostname} on port {port}!\n"

@app.route('/')
def index():
    con = connect_db()
    cur = con.cursor()
    cur.execute("SELECT * FROM tasks")
    tasks = cur.fetchall()
    con.close()
    return render_template("index.html", tasks=tasks)

@app.route('/add', methods=['POST'])
def add_task():
    task = request.form['task']
    con = connect_db()
    cur = con.cursor()
    cur.execute("INSERT INTO tasks (task, completed) VALUES (?, ?)", (task,
0))
    con.commit()
    con.close()
    return redirect('/')

@app.route('/delete/<int:id>')
def delete_task(id):
    con = connect_db()
    cur = con.cursor()
    cur.execute("DELETE FROM tasks WHERE id=?", (id,))
    con.commit()
    con.close()
    return redirect('/')

@app.route('/complete/<int:id>')
def complete_task(id):
    con = connect_db()
    cur = con.cursor()
    cur.execute("UPDATE tasks SET completed = 1 WHERE id=?", (id,))
    con.commit()
    con.close()
    return redirect('/')

if __name__ == "__main__":
    app.run(debug=True)
```

## Nginx

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    upstream flask_app {
        server app1:5000 max_fails=3 fail_timeout=30s; # Tolerância a
falhas
        server app2:5001 max_fails=3 fail_timeout=30s;
        server app3:3000 backup;
    }

    server {
        listen 80;
        server_name _;

        location / {
            proxy_pass http://flask_app;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            proxy_connect_timeout 10;
            proxy_send_timeout 10;
            proxy_read_timeout 10;

            client_max_body_size 10M;
        }
    }
}
```

## Docker Compose

```
services:
  app1:
    build:
      context: .
      dockerfile: Dockerfile
```

```
    container_name: app1
    environment:
      - PORT=5000
    volumes:
      - ./data:/app/db # Monta ./data do host em /app/db no container

app2:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: app2
  environment:
    - PORT=5001
  volumes:
    - ./data:/app/db # Monta ./data do host em /app/db no container

app3:
  image: busybox:latest
  container_name: app3
  volumes:
    - ./backup:/var/www # Monta o diretório app3 como raiz do servidor
  command: ["httpd", "-f", "-p", "3000", "-h", "/var/www"] # Inicia o
httpd na porta 3000
nginx:
  image: nginx:latest
  container_name: nginx
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  ports:
    - "80:80"
  depends_on:
    - app1
    - app2
    - app3
```

## Dockerfile

```
# Usa a versão leve do Python (Slim)
FROM python:3.9-slim
# Define o diretório de trabalho dentro do contêiner
WORKDIR /app

# Instala as dependências necessárias
RUN apt update && apt install -y net-tools bash

# Copia os arquivos necessários para o contêiner
COPY . .
# Instala as dependências necessárias (usa --no-cache para evitar arquivos
desnecessários)
RUN pip install --no-cache-dir -r requirements.txt
# Comando para iniciar a aplicação usando Gunicorn
```

```
CMD ["sh", "-c", "gunicorn --workers 2 --bind 0.0.0.0:${PORT:-5000}  
app:app"]
```

## Comandos

```
docker ps  
docker compose ps  
docker compose up -d --build  
docker compose down  
docker compose logs -f app1  
docker compose logs -f app2  
docker compose logs -f nginx  
docker exec -it app1 bash -c "netstat -tuln | grep 5000"  
docker exec -it app2 bash -c "netstat -tuln | grep 5001"  
curl http://localhost/oi  
for i in {1..10}; do curl http://localhost/oi; done
```

## Para parar um container

```
docker stop app1  
docker stop app2  
# assim testamos o backup
```