

TAREFA:

Enviar até o dia 21/09/2024:

Um arquivo PDF contendo as **capturas das telas** (somente) do *backend* e do *frontend* em execução, conforme o final do passo 7.

(Incluir no arquivo o nome do grupo, os nomes dos integrantes e suas respectivas matrículas).

Localização do link de envio no AVA:

"ENVIAR | Registros de Evidências do Levantamento de Necessidades".

(Basta que um único integrante do grupo faça o envio).

Observações:

1. **Não será permitida a adoção de aplicativos/ferramentas diferentes das descritas aqui**, salvo em situações extremamente necessárias consentidas pelo professor.
2. **Esse guia é apenas uma referência.** Cabe ao aluno/grupo fazer as pesquisas complementares para solucionar eventuais problemas naturais, ou adequações ao funcionamento esperado.

Guia Didático para a Construção de uma Base de Aplicação Web Simples com Docker

Visão Geral

Você está prestes a construir a sua primeira (ou não) aplicação web! O sistema será dividido em duas partes:

1. *Backend* (Parte do Servidor): Onde a lógica da aplicação e o banco de dados ficam.
2. *Frontend* (Parte do Cliente): A interface que o usuário vê e interage.

Passo 1: Ferramentas utilizadas no exercício

Cada ferramenta terá um papel específico no desenvolvimento da aplicação.

O **Docker** e **Docker Compose** garantirão que o ambiente seja consistente. O Docker é uma ferramenta que facilita o processo de desenvolvimento e execução de aplicações, empacotando tudo o que sua aplicação precisa para rodar em um *contêiner*, que é como uma caixinha virtual.

O **Flask** e **Werkzeug** irão gerenciar o *backend*; o **Node.js** e **React** vão lidar com o *frontend*; e o **MongoDB** poderá armazenar os dados com persistência. Bibliotecas como **axios** e **Flask-CORS** podem facilitar a comunicação entre *frontend* e *backend*, e o **Python** será a linguagem base para o desenvolvimento do *backend*.

Passo 2: Configurando o Ambiente de Desenvolvimento

2.1. Instalando o Docker

Antes de começar, você precisa instalar o Docker no seu computador:

- Windows e Mac: Baixe o Docker Desktop do site oficial:

<https://www.docker.com/products/docker-desktop>

- Linux: Siga as instruções específicas para a sua distribuição no site oficial do Docker.

Após a instalação, você pode verificar se tudo está funcionando abrindo o terminal (ou prompt de comando) e digitando:

```
docker --version
```

Você deve ver algo como Docker version 20.10.7, build f0df350.

Passo 3: Estrutura do Projeto

Agora, vamos organizar nossos arquivos. A estrutura do projeto será assim:

```
project/
├── backend/      Parte do servidor
├── frontend/     Parte do cliente
└── docker-compose.yml  Arquivo que orquestra todo o projeto
```

3.1. *Backend* (Parte do Servidor)

O *backend* é onde ficam as regras do jogo. Aqui, lidamos com o banco de dados, a lógica dos negócios e a API que conecta o *frontend* ao *backend*.

Observação:

API significa "Application Programming Interface". É uma forma de permitir que dois sistemas diferentes se comuniquem. No nosso caso, o *frontend* e o *backend* se comunicarão através de uma API REST (um tipo de API que segue algumas regras simples para troca de informações).

3.2. *Frontend* (Parte do Cliente)

O *frontend* é o que o usuário vê e interage. Aqui, usaremos React (uma biblioteca JavaScript) para construir uma página simples.

Passo 4: Criando o Backend com Docker

4.1. Criando o Dockerfile do Backend

Sua pasta backend/ deverá ter esta estrutura:

```
backend/
├── app.py
├── Dockerfile
└── requirements.txt
```

4.1.1. Arquivo: backend/app.py

```
from flask import Flask, jsonify, request

# Inicializa a aplicação Flask
app = Flask(__name__)
```

```

# Simulação de banco de dados em memória (lista de projetos)
projects = []

# Rota para retornar todos os projetos (GET)
@app.route('/projects', methods=['GET'])
def get_projects():
    return jsonify(projects), 200

# Rota para adicionar um novo projeto (POST)
@app.route('/projects', methods=['POST'])
def add_project():
    new_project = request.get_json() # Pega os dados enviados no
    corpo da requisição
    projects.append(new_project)
    return jsonify(new_project), 201

# Ponto de entrada para rodar a aplicação
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)

```

Roda o *backend* usando Flask. Este exemplo cria uma pequena API RESTful que responde a requisições GET e POST.

O que está acontecendo aqui:

a) Importações:

- Flask : Framework que cria o servidor web.
- jsonify : Função para transformar dados em JSON, formato usado nas APIs REST.
- request : Utilizado para lidar com dados de requisições POST, como a criação de novos projetos.

b) Inicializa a aplicação (app = Flask(__name__)) :

- A função Flask() inicializa o aplicativo web.

c) Rota GET para /projects :

- @app.route('/projects', methods=['GET']) define uma rota que responde a requisições HTTP GET, devolvendo a lista de projetos cadastrados (nesse caso, armazenados em uma lista em memória).

d) Rota POST para /projects :

- @app.route('/projects', methods=['POST']) define uma rota que aceita requisições POST. Ela recebe um projeto em formato JSON e o adiciona à lista de projetos.

e) Ponto de entrada (if __name__ == '__main__') :

- Garante que a aplicação Flask rodará quando você executar o arquivo `app.py`. O servidor escuta em todas as interfaces de rede (0.0.0.0) e na porta 5000 (como configurado no Docker).

4.1.2. Arquivo: backend/Dockerfile

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "app.py"]
```

Um Dockerfile é um arquivo que contém um conjunto de instruções para construir a imagem do Docker. Pense nele como uma receita.

O que está acontecendo aqui:

- a) `FROM python:3.10-slim`: Estamos dizendo ao Docker para começar com uma versão leve do Python 3.10.
- b) `WORKDIR /app`: Aqui, definimos o diretório de trabalho para `/app` dentro do contêiner.
- c) `COPY requirements.txt requirements.txt`: Estamos copiando o arquivo de dependências do Python para o contêiner.
- d) `RUN pip install --no-cache-dir -r requirements.txt`: Instalamos as dependências do Python no contêiner.
- e) `COPY . .`: Copiamos todos os arquivos do diretório atual para dentro do contêiner.
- f) `CMD ["python", "app.py"]`: Finalmente, dizemos ao Docker para rodar o arquivo `app.py` usando Python quando o contêiner iniciar.

4.1.3. Arquivo: backend/requirements.txt

```
Flask==2.3.3
Flask-RESTful==0.3.9
pymongo==4.1.1
```

- a) *Flask: Framework* usado para construir o servidor web no *backend*. Ele serve como base para criar as rotas da API, que permitirá a comunicação entre o *frontend* (React) e o *backend*.
- b) *Flask-RESTful*: Extensão do Flask que simplifica a definição de rotas e as operações associadas, facilitando a estruturação da API REST, como requisições GET e POST, essencial para que o *frontend* e o *backend* possam se comunicar corretamente.

- c) *pymongo*: Biblioteca usada para conectar a aplicação Flask ao banco de dados MongoDB, que é um dos serviços listados no `docker-compose.yml`. Permite que a aplicação interaja com o MongoDB, possibilitando operações de leitura e gravação de dados no banco.

Passo 5: Criando o Frontend com Docker

5.1. Criando o Dockerfile do Frontend

Similar ao *backend*, criamos um Dockerfile para o *frontend*. Sua pasta `frontend/` deve ter esta estrutura:

```
frontend/
├── Dockerfile
├── package.json
├── package-lock.json
├── public/
│   └── index.html
└── src/
    ├── App.js
    └── index.js
```

5.1.1. Arquivo: `frontend/Dockerfile`

```
# Usar uma imagem leve do Node.js
FROM node:18-alpine

# Definir o diretório de trabalho dentro do contêiner
WORKDIR /app

# Copiar os arquivos de dependências para o contêiner
COPY package.json package-lock.json ./

# Instalar as dependências
RUN npm install

# Copiar todos os arquivos do projeto para o contêiner
COPY . .

# Construir a versão otimizada da aplicação
RUN npm run build

# Expor a porta 3000 para o servidor
EXPOSE 3000
```

```
# Comando para rodar a aplicação
CMD ["npm", "start"]
```

O que está acontecendo aqui:

- a) FROM node:18-alpine : Começamos com uma versão leve do Node.js.
- b) WORKDIR /app : Definimos o diretório de trabalho para /app.
- c) COPY package.json package-lock.json ./ : Copiamos os arquivos de configuração do Node.js.
- d) RUN npm install : Instalamos as dependências necessárias.
- e) COPY . . : Copiamos todos os arquivos para o contêiner.
- f) RUN npm run build : Construimos o projeto.
- g) EXPOSE 3000 : Informamos ao Docker que nossa aplicação estará disponível na porta 3000.
- h) CMD ["npm", "start"] : Iniciamos a aplicação quando o contêiner rodar.

5.1.2. Arquivo: frontend/package.json

```
{
  "name": "frontend",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1"
  },
  "devDependencies": {},
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Após adicionar essas dependências, rode o comando:

```
npm install
```

Isso instalará as dependências mencionadas e gerará o arquivo `package-lock.json`.

5.1.3. Arquivo: `frontend/public/index.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1" />
    <meta name="theme-color" content="#000000" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this
app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Esse arquivo é essencial para que o React construa a aplicação, pois ele é o ponto de entrada para o *frontend*.

5.1.4. Arquivo: `frontend/src/App.js`

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Bem-vindo à aplicação de exemplo!</h1>
      <p>Este é um exemplo de uma Single Page Application (SPA)
usando React.</p>
    </div>
  );
}

export default App;
```

Este código renderiza uma página simples que exibe uma mensagem de boas-vindas e pode ser facilmente adaptado para fazer chamadas à API do *backend* no futuro.

O que está acontecendo aqui:

- a) `import React from 'react';` : Importa a biblioteca React, que é necessária para criar componentes no *frontend*.
- b) `function App()` : Define o componente principal da aplicação chamado App. Um componente é uma função que retorna um pedaço da interface de usuário.
- c) `return` : Dentro da função `App()`, o `return` define o que será exibido na página. Aqui estamos usando HTML simples (JSX no React) para mostrar um título e um parágrafo.
- d) `export default App;` : Torna o componente App disponível para ser importado em outros arquivos, como o arquivo principal que renderiza a aplicação (normalmente `index.js`).

5.1.5. Arquivo: `frontend/src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css'; // Se você não tiver um arquivo CSS, remova
esta linha
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

O `index.js` é o ponto de entrada da aplicação React e é responsável por renderizar o componente principal (geralmente `App.js`) dentro do HTML.

O que está acontecendo aqui:

- a) `import React from 'react';` : Importa a biblioteca React, que é necessária para criar interfaces.
- b) `import ReactDOM from 'react-dom/client';` : Importa a função ReactDOM para renderizar o React no DOM.
- c) `import App from './App';` : Importa o componente principal App, que será renderizado na interface.
- d) `document.getElementById('root')` : O HTML (`index.html`) possui um elemento com o ID `root`, e é dentro dele que o React irá renderizar a aplicação.

Passo 6: Orquestrando Tudo com Docker Compose

6.1. Criando arquivo `docker-compose.yml`

Docker Compose é uma ferramenta que permite definir e rodar aplicações Docker multi-contêiner. Usaremos o `docker-compose.yml` para rodar o *backend*, o *frontend* e o banco de dados MongoDB juntos.

6.1.1. Arquivo: `docker-compose.yml`

```
version: '3.8'

services:
  backend:
    build: ./backend
    ports:
      - "5000:5000"
    environment:
      - MONGO_URI=mongodb://mongo:27017/ongdb
    depends_on:
      - mongo

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend

  mongo:
    image: mongo:5.0
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:
```

O que está acontecendo aqui:

- a) `services` : Definimos três serviços: *backend*, *frontend* e *mongo* (o banco de dados).
- b) `build` : Especifica o diretório de onde o Docker deve construir o serviço.
- c) `ports` : Define as portas que serão expostas para acesso externo.
- d) `depends_on` : Define dependências, para garantir que um serviço esteja rodando antes do outro.
- e) `volumes` : Cria um volume para persistir os dados do MongoDB.

6.1.2. Orquestrar a aplicação

1. Abra o terminal na pasta do projeto.

2. Execute o comando abaixo para iniciar todos os serviços:

```
docker-compose up --build
```

3. O Docker irá construir as imagens e rodar os contêineres para o *backend*, *frontend* e banco de dados MongoDB.

4. Se quiser, você pode forçar a reconstrução das imagens do zero sem usar o cache anterior, o que pode resolver problemas de cópia de arquivos.












```
docker-compose build --no-cache
```

Passo 7: Executando a Aplicação

Inicie os contêineres com o comando:

```
docker-compose up
```

No Docker Desktop, vá até a aba "Containers/Apps". Lá você verá os contêineres rodando, provavelmente com nomes como *backend* e *frontend*. Ao lado de cada contêiner, você verá as portas que estão sendo expostas. Por exemplo:

<input type="checkbox"/>	<div> project</div>		Running (3/3)	0.58%	2 minutes ago				
<input type="checkbox"/>	<div> mongo-1 9a621bdfaaf5 </div>	mongo:5.0	Running	27017:27017 	0.41%	2 minutes ago			
<input type="checkbox"/>	<div> backend-1 95fe3d48c5f4 </div>	project-backend:<none>	Running	5000:5000 	0.13%	2 minutes ago			
<input type="checkbox"/>	<div> frontend-1 94aba1761eb2 </div>	project-frontend:<none>	Running	3000:3000 	0.04%	2 minutes ago			

Isso confirma que suas portas locais estão mapeadas para os contêineres, e que `localhost:3000` e `localhost:5000` estão acessíveis.

O `localhost` que você usa no navegador está mapeado para os contêineres Docker.

- Acesse o *frontend* no seu navegador: `http://localhost:3000`

- A API do *backend* estará disponível em: `http://localhost:5000/projects`

Se você estiver rodando o projeto no terminal com `docker-compose up`, você pode simplesmente pressionar **CTRL + C**.

Se você já saiu do terminal ou deseja garantir que todos os contêineres do projeto sejam parados, você pode executar:

```
docker-compose down
```

Isso irá parar e remover os contêineres criados pelo Docker Compose, mas sem apagar as imagens ou volumes persistentes.

Anexo 1: Resumo das ferramentas utilizadas no exercício

A1.1. Docker

- **Descrição:** Uma plataforma que permite criar, empacotar e executar aplicações em contêineres, isolando o ambiente de desenvolvimento do sistema operacional.
- **Utilidade no exercício:** Garante que tanto o *backend* (Flask) quanto o *frontend* (React) rodem em ambientes consistentes e isolados, independentemente da máquina do desenvolvedor, empacotando todas as dependências e configurações necessárias para que a aplicação rode de forma previsível.

A1.2. Docker Compose

- **Descrição:** Ferramenta que permite definir e gerenciar aplicações multi-contêiner, usando um arquivo YAML para especificar como os contêineres devem interagir.
- **Utilidade no exercício:** Orquestra o *backend* (Flask), o *frontend* (React), e o banco de dados (MongoDB) em contêineres separados, facilitando a execução e o gerenciamento de todos os serviços da aplicação ao mesmo tempo.

A1.3. Flask

- **Descrição:** Um microframework web escrito em Python, usado para construir aplicações web e APIs. Ele fornece o mínimo necessário para criar um *backend*, como roteamento e renderização de *templates*.
- **Utilidade no exercício:** Construir o *backend* da aplicação, onde são definidas as rotas e a lógica de negócios. Ele também expõe uma API REST que o *frontend* (React) consome.

A1.4. Werkzeug

- **Descrição:** Trata-se de um servidor web embutido que acompanha o Flask. Usado principalmente para desenvolvimento e oferece funcionalidades como depuração e recarregamento automático.

- **Utilidade no exercício:** Ele é usado automaticamente pelo Flask como servidor durante o desenvolvimento. Ele serve a aplicação Flask localmente e ajuda no desenvolvimento com seus recursos de *debugging*.

A1.5. Node.js

- **Descrição:** Node.js é um ambiente de execução de JavaScript no lado do servidor, que permite executar código JavaScript fora do navegador. Ele também é a base para a ferramenta de gerenciamento de pacotes npm.

- **Utilidade no exercício:** É necessário para o *frontend* da aplicação, já que usamos o React (que é baseado em Node.js). Ele também gerencia pacotes JavaScript, como axios, que são usados para fazer requisições HTTP do *frontend* para o *backend*.

A1.6. React

- **Descrição:** O React é uma biblioteca JavaScript usada para construir interfaces de usuário (UI) dinâmicas e responsivas, permitindo a criação de Single Page Applications (SPA).

- **Utilidade no exercício:** Utilizado para construir o *frontend* da aplicação, lida com a interface que o usuário vê e interage, além de fazer chamadas para a API do *backend* para obter e exibir dados.

A1.7. Python

- **Descrição:** Linguagem de programação de propósito geral, conhecida por sua simplicidade e versatilidade. Flask é baseado em Python.

- **Utilidade no exercício:** É a linguagem base para o desenvolvimento do *backend* da aplicação, com o Flask fornecendo as funcionalidades de framework para a construção de APIs e lógica de negócio.

A1.8. MongoDB

- **Descrição:** É um banco de dados NoSQL que armazena dados em formato de documentos JSON. Ele é escalável e flexível para armazenar dados não estruturados.

- **Utilidade no exercício:** Utilizado como o banco de dados para o *backend* da aplicação. Ele armazena os dados dos projetos e quaisquer outros dados que precisem ser persistidos pela API do Flask.

A1.9. npm (Node Package Manager)

- **Descrição:** npm é o gerenciador de pacotes do Node.js, usado para instalar e gerenciar bibliotecas e dependências JavaScript.

- **Utilidade no exercício:** Usado para instalar pacotes no *frontend*, como axios (para fazer chamadas HTTP) e as dependências do React.

A1.10. axios

- **Descrição:** Biblioteca JavaScript usada para fazer requisições HTTP. Ela permite interagir com APIs e obter ou enviar dados de/para o *backend*.
- **Utilidade no exercício:** Utilizado no *frontend* (React) para fazer requisições à API do Flask. Ele envia dados para o *backend* (POST) e recupera dados da API (GET) para exibição no *frontend*.

A1.11. Flask-CORS

- **Descrição:** Extensão para Flask que permite configurar o CORS (Cross-Origin Resource Sharing), necessário para permitir ao *frontend* fazer requisições para o *backend* em domínios ou portas diferentes.
- **Utilidade no exercício:** Usado para habilitar o CORS na API Flask, permitindo que o *frontend* em `http://localhost:3000` faça requisições para o *backend* em `http://localhost:5000` sem ser bloqueado por questões de segurança.

Anexo 2: Testando a API do exercício

É possível fazer o *frontend* exibir o conteúdo dos projetos enviados via JSON para o *backend*. Para isso, você pode usar o React no *frontend* para fazer uma requisição GET para a API `/projects` e exibir a lista de projetos que foi armazenada na memória (ou futuramente no banco de dados).

Aqui está um passo a passo de como fazer isso:

Passo 1: Pare os contêineres, caso estejam em execução.

Passo 2: Instale o *axios*, uma biblioteca para fazer requisições HTTP, como as que estamos fazendo para o *backend*.

Opção 1. Acrescente a seguinte dependência no arquivo `package.json`:

```
"dependencies": {  
  "axios": "^0.21.1",
```

Após adicionar essa dependência, rode o comando a seguir dentro do diretório `backend/`:

```
npm install
```

Opção 2. Execute o seguinte comando na pasta frontend/:

```
npm install axios
```

Passo 3: Atualizar o App.js no *frontend*

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [projects, setProjects] = useState([]); // Estado para
  armazenar os projetos

  // useEffect é chamado quando o componente é montado
  useEffect(() => {
    // Fazendo uma requisição GET para o backend na rota /projects
    axios.get('http://localhost:5000/projects')
      .then(response => {
        setProjects(response.data); // Armazena os projetos no
        estado
      })
      .catch(error => {
        console.error("Houve um erro ao buscar os projetos!",
        error);
      });
  }, []); // [] garante que o useEffect seja executado apenas uma
  vez

  return (
    <div>
      <h1>Bem-vindo à aplicação de exemplo!</h1>
      <p>Este é um exemplo de uma Single Page Application (SPA)
      usando React.</p>

      { /* Exibindo os projetos apenas se houver algum */ }
      <h2>Lista de Projetos da ONG</h2>
      { projects.length === 0 ? (
        <p>Nenhum projeto foi adicionado ainda.</p>
      ) : (
        <ul>
          { projects.map((project, index) => (
            <li key={index}>
              <strong>Nome:</strong> {project.name} <br />
              <strong>Descrição:</strong> {project.description}
            </li>
          )) }
        </ul>
      ) }
    </div>
  );
}

export default App;
```

- a) `useState`: Define um estado (`projects`) que vai armazenar a lista de projetos recebida do *backend*.
- b) `useEffect`: Executa uma requisição GET para buscar os projetos assim que o componente React é montado. A resposta do *backend* é armazenada no estado `projects`.

Passo 4: A API Flask precisa permitir que o *frontend* faça requisições de diferentes origens. Vamos adicionar uma configuração de CORS (*Cross-Origin Resource Sharing*) no Flask. No arquivo `requirements.txt` do *backend*, adicione:

```
Flask-CORS==3.0.10
```

Passo 5: Modifique o arquivo `app.py` para ativar o CORS:

```
from flask import Flask, jsonify, request
from flask_cors import CORS    Importa o CORS

app = Flask(__name__)
CORS(app)    Habilita o CORS para o Flask

projects = []

@app.route('/projects', methods=['GET'])
def get_projects():
    return jsonify(projects), 200

@app.route('/projects', methods=['POST'])
def add_project():
    new_project = request.get_json()
    projects.append(new_project)
    return jsonify(new_project), 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Passo 6: Reconstrua e reinicie o *backend* com:

```
docker-compose build --no-cache
docker-compose up
```

Passo 7: Agora você pode rodar a aplicação e testar a exibição dos projetos. Você pode enviar novos projetos para a API com uma requisição POST para `http://localhost:5000/projects`, enviando um JSON com um novo projeto.

Execute o comando a seguir no PowerShell:

```
Invoke-RestMethod -Uri http://localhost:5000/projects -Method Post -
Headers @{ "Content-Type" = "application/json" } -Body '{"name":
"Projeto ONG", "description": "Um projeto simples"}'
```


O que está acontecendo aqui:

- a) `Invoke-RestMethod` : Este cmdlet faz requisições HTTP no PowerShell, equivalente ao `curl`.
- b) `Uri` : O *endpoint* da API.
- c) `Method Post` : Define que o método HTTP será POST.
- d) `Headers @{ "Content-Type" = "application/json" }` : Define o cabeçalho da requisição para indicar que o corpo da mensagem será em formato JSON.
- e) `Body` : O corpo da requisição, onde você passa os dados em JSON.

No exemplo do `app.py`, a API que você está usando não está conectada a um banco de dados real, então quando você envia o JSON via POST:

- Armazena os dados em uma lista na memória:
 - Quando você envia uma requisição POST para a rota `/projects` com um JSON, os dados são adicionados a uma lista em memória chamada `projects`.
 - Essa lista existe apenas enquanto o servidor Flask estiver rodando.
- Retorna o objeto enviado:
 - A API retorna o mesmo JSON que você enviou, mas com uma confirmação de sucesso (status 201), indicando que o projeto foi criado com sucesso.
 - Esse retorno apenas exibe o que foi enviado, mas não há transformação ou processamento adicional além de armazenar os dados na lista e devolvê-los como resposta.

Anexo 3: Reflexão do desenvolvimento web "ontem" e "hoje"

A3.1. O "antigamente": *Frontend* e *Backend* mais simples

No início, o desenvolvimento web era mais direto:

- *Frontend*: Criávamos HTML, CSS e JavaScript diretamente no Notepad, e essas tecnologias eram suficientes para criar páginas estáticas com algumas interações.

- *Backend*: Utilizávamos PHP, MySQL e servidores como Apache para fazer sites dinâmicos. Tudo era feito em um único servidor, e as tecnologias eram monolíticas e simples.

Você controlava quase todo o desenvolvimento no mesmo ambiente (servidor web local ou compartilhado) e o ciclo de desenvolvimento era rápido. Muitas vezes, um único arquivo `.php` cuidava de tudo, incluindo o HTML, lógica e manipulação de banco de dados.

A3.2. O "hoje": *Frontend* e *Backend* mais sofisticados

Atualmente, o desenvolvimento web mudou drasticamente. Surgiram novas ferramentas, metodologias e arquiteturas que tornaram as coisas aparentemente mais complexas. Mas por que isso aconteceu?

A3.2.1. *Frontend*

- *Aplicações mais interativas*: As expectativas dos usuários mudaram. As páginas são mais dinâmicas e interativas, muitas vezes parecendo com "aplicativos" em vez de simples páginas web, o que exige frameworks como React, Vue.js, e Angular para lidar com atualizações em tempo real, roteamento no *frontend* e estados complexos.
- *SPA (Single Page Application)*: Agora é comum que as aplicações sejam SPA, onde toda a navegação e manipulação de dados ocorre sem recarregar a página. Isso exige uma arquitetura mais elaborada com chamadas API, roteamento no *frontend*, manipulação de estados, etc.
- *Ferramentas de build*: Ferramentas como Node.js (com npm ou yarn), Webpack, Babel surgiram para ajudar a gerenciar dependências, compilar código e otimizar o desempenho de grandes projetos. Isso aumenta a complexidade, mas facilita a manutenção e escalabilidade dos projetos.

A3.2.2. *Backend*

- *APIs RESTful e microserviços*: Antes, tudo era gerenciado em uma única aplicação (PHP ou outra linguagem). Hoje, é comum separar o *backend* em APIs RESTful ou microserviços. Isso melhora a modularidade, escalabilidade e permite que diferentes partes da aplicação sejam desenvolvidas separadamente, em linguagens diferentes (por exemplo, Python/Flask para uma API e Node.js para outra).
- *Escalabilidade e performance*: Aplicações modernas precisam lidar com uma quantidade enorme de usuários e tráfego, o que levou à popularização de tecnologias como Docker e Kubernetes. Elas permitem que você crie, distribua e gerencie contêineres, oferecendo maior controle sobre o ambiente em que o *backend* e o *frontend* rodam.
- *DevOps e CI/CD*: Com o aumento da complexidade, surgiram as práticas de DevOps e CI/CD (Integração Contínua/Entrega Contínua), para automatizar *deploys* e testes. Isso requer ferramentas e configurações adicionais, mas garante que as aplicações possam ser lançadas rapidamente e de forma confiável.
- *Segurança*: Ferramentas como CORS, autenticação OAuth, e criptografia avançada surgiram em resposta à necessidade de segurança em uma web mais interativa e acessível. Isso adiciona mais camadas ao desenvolvimento.

A3.3. Por que tudo parece mais difícil?

A complexidade aumentou porque o tipo de experiências web que os usuários e empresas esperam também cresceu. O que antes era uma página simples, hoje é uma aplicação complexa com interatividade avançada, responsividade, segurança e escalabilidade.

- Escalabilidade: Usando Docker e outras ferramentas modernas, você pode facilmente escalar sua aplicação para milhares ou milhões de usuários.
- Manutenção e modularidade: Ao dividir seu projeto em microserviços, *frontends* separados, e APIs, fica mais fácil manter o código e expandi-lo ao longo do tempo.
- Experiência do usuário: Com frameworks modernos como React, você pode oferecer uma experiência de usuário mais rica e fluida, semelhante a aplicativos nativos.

A3.4. Então, a abordagem "antiga" morreu?

A abordagem mais simples de "HTML + CSS + JS" e *backend* monolítico ainda existe e é usada, especialmente para projetos pequenos, sites simples ou até blogs pessoais. Há muitos cenários onde simples é melhor. Mas para aplicações maiores, com milhares de usuários, lógica complexa e integração com múltiplos sistemas, as novas ferramentas e abordagens são essenciais.

Anexo 4: Roteamento no frontend e estados complexos

A4.1. O que é roteamento no *frontend*

Dentro do contexto do desenvolvimento web, o termo **roteamento** se refere ao processo de definir como uma aplicação web responde a diferentes URLs ou requisições HTTP. O roteamento direciona o tráfego dentro da aplicação, ligando URLs específicas a funcionalidades ou componentes de código que devem ser executados.

Tradicionalmente, o roteamento era feito no *backend*. Cada vez que um usuário clicava em um link ou acessava uma URL, o servidor processava a solicitação e enviava de volta uma nova página HTML. Isso funcionava bem para páginas estáticas ou com poucas interações dinâmicas.

Frontend sem roteamento (antigo):

- Um usuário acessava `www.exemplo.com/pagina1`.

- O servidor enviava um arquivo HTML correspondente à página 1.
- Se o usuário clicasse em um link para `www.exemplo.com/pagina2`, o servidor processava a nova solicitação e enviava o HTML da página 2.

Com o advento das *Single Page Applications* (SPA), o conceito de roteamento passou a ser gerenciado no *frontend*. Isso significa que o navegador carrega apenas uma única página HTML e, em vez de fazer uma nova solicitação ao servidor a cada mudança de URL, o JavaScript do *frontend* manipula a mudança de "páginas" sem recarregar o navegador.

Roteamento no *frontend* (moderno):

- Um usuário acessa `www.exemplo.com/`.
- A página é carregada apenas uma vez (um arquivo HTML e um script JavaScript).
- Se o usuário clicar em um link para `www.exemplo.com/pagina1`, o JavaScript (normalmente usando bibliotecas como React Router, Vue Router ou Angular Router) muda a URL e exibe o conteúdo correspondente à página 1 sem recarregar a página.
- A navegação entre "páginas" no *frontend* é rápida e suave, pois a página HTML não é recarregada do servidor.

No nosso exemplo:

- **CORS:** Um mecanismo de segurança que controla as permissões para que o *frontend* faça requisições ao *backend* em diferentes domínios. Ele não faz roteamento no *frontend*.
- `app.py` (*backend*): Faz o roteamento no *backend*, definindo como a API responde às requisições.
- **Frontend (React):** Para fazer roteamento no *frontend*, podemos utilizar bibliotecas como React Router para React, Vue Router para Vue.js, ou Angular Router para Angular, que permitem navegar entre diferentes páginas ou componentes sem recarregar a página.

A4.2. O que são estados complexos no *frontend*

No desenvolvimento *frontend* moderno, especialmente em SPAs, o conceito de estado refere-se aos dados e informações que estão sendo mantidos e manipulados na aplicação enquanto o usuário interage com ela. Um estado simples pode ser algo básico, como armazenar se um botão foi clicado ou não.

Em uma aplicação maior, o estado pode se tornar muito mais complexo. Em vez de armazenar um único valor, você pode estar lidando com:

- Autenticação de usuários (o usuário está logado ou não?).
- Dados carregados de uma API (uma lista de produtos, projetos, etc.).

- Interações dinâmicas do usuário (filtros aplicados, itens selecionados em um carrinho de compras).
- Componentes interligados: Vários componentes que precisam compartilhar dados ou reagir a mudanças no estado de outros componentes.

Imagine que você tem uma aplicação de e-commerce, onde você precisa armazenar:

- Dados do usuário (se ele está logado ou não).
- Itens no carrinho de compras.
- Filtros aplicados a uma lista de produtos.
- A página atual em que o usuário está navegando.