**Relatório de Aula Prática - Desenvolvimento**

**Com Framework Para Node.js**

**Aluno: Gabriel Neves**

**CONSTRUINDO UM SERVIDOR WEB BÁSICO**

**Descrição da Atividade**

O objetivo desta atividade foi criar um servidor HTTP básico usando Node.js, capaz de responder a diferentes rotas com mensagens específicas. A atividade permite entender como configurar um servidor em Node.js e responder a solicitações em diversas URLs.

**Passo a Passo da Implementação**

Segui o roteiro do portfólio para construir o projeto e desenvolvi o servidor HTTP conforme as instruções. Abaixo, compartilho o código completo do arquivo servidor.js:

*const http = require('http');*

*const server = http.createServer((req, res) => {*

*res.setHeader('Content-Type', 'text/plain');*

*switch (req.url) {*

*case '/':*

*res.statusCode = 200;*

*res.end('Hello, world');*

*break;*

*case '/sobre':*

*res.statusCode = 200;*

*res.end('Página Sobre');*

*break;*

*case '/contato':*

*res.statusCode = 200;*

*res.end('Página de Contato');*

*break;*

*default:*

*res.statusCode = 404;*

*res.end('Página não encontrada');*

*break;*

*}*

*});*

*const PORT = 3000;*

*server.listen(PORT, () => {*

*console.log(`Servidor rodando em http://localhost:${PORT}`);*

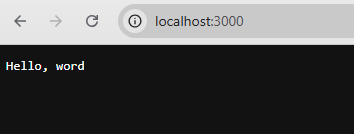
*});*

**Teste do Projeto**

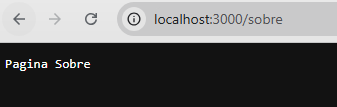
Realizei testes para cada uma das rotas configuradas. Os resultados foram verificados no navegador, e as respostas foram exibidas corretamente.

**Evidências de Execução**

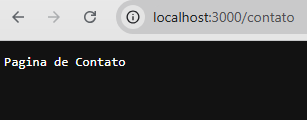
Acessando http://localhost:3000/: Exibe "Hello, world":



Acessando http://localhost:3000/sobre: Exibe "Página Sobre"



Acessando http://localhost:3000/contato: Exibe "Página de Contato"



**IMPLEMENTAÇÃO E DEPURAÇÃO UTILIZANDO O NODE JS**

**Testando E Depurando Aplicações Node.Js**

**Passo a Passo da Configuração**

1. Criei um diretório para o projeto e naveguei até ele no terminal:

*mkdir soma*

*cd soma*

2. Executei o comando para inicializar o projeto com um arquivo package.json padrão:

*npm init -y*

3. Instalei o Mocha como dependência de desenvolvimento:

*npm install mocha --save-dev*

4. Na raiz do projeto, criei um arquivo math.js para a implementação da função de soma:

*function soma(a, b) {*

*return a + b;*

*}*

*module.exports = { soma };*

5. Criei um diretório test e, dentro dele, um arquivo math.test.js com os casos de teste para a função soma:

*const assert = require('assert')*

*const { soma } = require('../math');*

*describe('Testes de Soma do Servidor', () => {*

*it('should return 5 when adding 2 and 3', (done) => {*

*assert.strictEqual(soma(2, 3), 5);*

*done();*

*});*

*it('should return -1 when adding -2 and 1', (done) => {*

*assert.strictEqual(soma(-2, 1), -1);*

*done();*

*});*

*it('should return 0 when adding 0 and 0', (done) => {*

*assert.strictEqual(soma(0, 0), 0);*

*done();*

*});*

*});*

6. Adicionei um script no package.json para rodar o Mocha:

"scripts": {

"test": "mocha"

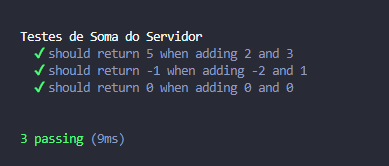
}

7. Execução dos Testes:

*npm test*

8. Esse foi o resultado:

Os testes passaram com sucesso, validando a funcionalidade da função soma em diferentes cenários.



**INTERFACE E SEGURANÇA NO NODE.JS**

**Desenvolvimento De Interfaces De Usuário Com Node.Js**

1. Configuração do Projeto

Para começar, criei uma pasta onde o projeto ficaria armazenado. Fiz isso no terminal com os comandos:

mkdir validacao-cpf

cd validacao-cpf

2. Criação dos Arquivos

Dentro da pasta do projeto, criei três arquivos principais: cpf.html, cpf.css, e cpf.js. Esses arquivos são responsáveis, respectivamente, pelo layout HTML, pelos estilos CSS, e pela lógica JavaScript de validação.

Para criar esses arquivos pelo terminal, usei:

*touch cpf.html cpf.css cpf.js*

3. Estruturação do Código

Agora eu adicionei o código em cada um desses arquivos.

**HTML (cpf.html)**

No cpf.html, criei um formulário simples que inclui um campo para inserir o CPF e uma área para exibir a mensagem de validação. O código ficou assim:

*<!DOCTYPE html>*

*<html lang="pt-BR">*

*<head>*

*<meta charset="UTF-8">*

*<meta name="viewport" content="width=device-width, initial-scale=1.0">*

*<title>Validação de CPF</title>*

*<link rel="stylesheet" href="cpf.css">*

*</head>*

*<body>*

*<div class="container">*

*<h2>Validação de CPF</h2>*

*<form id="form-cpf">*

*<label for="cpf">CPF:</label>*

*<input type="number" id="cpf" maxlength="11" placeholder="Digite o CPF (somente números)">*

*<p id="mensagem-validacao"></p>*

*<button class="button-validate" type="button" onclick="validaCpf()">Validar CPF</button>*

*</form>*

*</div>*

*<script src="cpf.js"></script>*

*</body>*

*</html>*

**CSS (cpf.css)**

Para estilizar a página, utilizei o arquivo cpf.css, onde defini algumas regras de estilo para diferenciar visualmente as mensagens. A cor verde indica quando o CPF é válido, e a cor vermelha quando é inválido. O código ficou assim:

*body {*

*font-family: Arial, sans-serif;*

*display: flex;*

*justify-content: center;*

*align-items: center;*

*height: 100vh;*

*margin: 0;*

*background-color: #EBF0F5;*

*}*

*.container {*

*text-align: center;*

*background-color: #FFF;*

*padding: 20px;*

*border-radius: 8px;*

*box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);*

*width: 300px;*

*}*

*input {*

*width: 100%;*

*padding: 10px;*

*margin-top: 10px;*

*border: 1px solid #ccc;*

*border-radius: 4px;*

*box-sizing: border-box;*

*}*

*.button-validate{*

*width:100%;*

*padding: 12px 12px 12px 12px;*

*background: #004ECC;*

*border-radius: 8px;*

*outline:none !important;*

*color: #FFF;*

*border: none !important;*

*cursor: pointer;*

*}*

*#mensagem-validacao {*

*margin-top: 10px;*

*font-weight: bold;*

*}*

*.sucesso {*

*color: green;*

*}*

*.erro {*

*color: red;*

*}*

**JavaScript (cpf.js)**

Por último, escrevi o código JavaScript no cpf.js para validar o CPF. Quando o CPF é inserido, a função verifica se ele está correto e exibe a mensagem correspondente. Usei o seguinte código:

*function validaCpf(){*

*const cpf = document.getElementById("cpf").value;*

*const mensagemValidacao = document.getElementById("mensagem-validacao");*

*if (cpf.length === 11) {*

*if (validarCPF(cpf)) {*

*mensagemValidacao.textContent = "CPF válido!";*

*mensagemValidacao.className = "sucesso";*

*} else {*

*mensagemValidacao.textContent = "CPF inválido!";*

*mensagemValidacao.className = "erro";*

*}*

*} else {*

*mensagemValidacao.textContent = "CPF inválido!";*

*mensagemValidacao.className = "erro";*

*}*

*}*

*function validarCPF(cpf) {*

*if (/^(\d)\1{10}$/.test(cpf)) return false;*

*let soma = 0;*

*let resto;*

*for (let i = 1; i <= 9; i++) {*

*soma += parseInt(cpf.substring(i - 1, i)) \* (11 - i);*

*}*

*resto = (soma \* 10) % 11;*

*if (resto === 10 || resto === 11) resto = 0;*

*if (resto !== parseInt(cpf.substring(9, 10))) return false;*

*soma = 0;*

*for (let i = 1; i <= 10; i++) {*

*soma += parseInt(cpf.substring(i - 1, i)) \* (12 - i);*

*}*

*resto = (soma \* 10) % 11;*

*if (resto === 10 || resto === 11) resto = 0;*

*if (resto !== parseInt(cpf.substring(10, 11))) return false;*

*return true;*

*}*

A função validarCPF utiliza uma fórmula de cálculo específica para verificar os dígitos verificadores do CPF. Ao inserir o CPF no campo apropriado, o JavaScript chama essa função automaticamente para determinar se o número é válido.

4. Testando a Aplicação

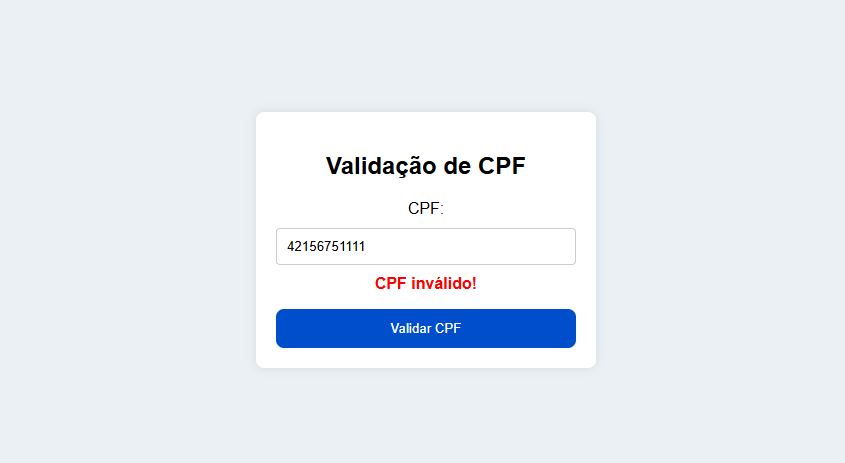
Depois de escrever os códigos, testei a aplicação abrindo o arquivo cpf.html no navegador. No campo de CPF, inseri valores para verificar se o código estava funcionando. A mensagem "CPF válido!" aparece em verde quando o CPF está correto, e "CPF inválido!" aparece em vermelho caso contrário.

Conclusão

Este projeto proporciona uma verificação de CPF diretamente no navegador, com um feedback visual imediato, ajudando o usuário a saber rapidamente se o CPF inserido é válido ou inválido.

Evidencias:





**TESTES UTILIZANDO NODE.JS**

**Estratégias De Testes**

1. Configuração do Projeto

Criei uma nova pasta chamada servidor-http e acessei o diretório:

*mkdir servidor-http*

*cd servidor-http*

*npm init -y*

2. Instalação das Dependências

Instalei as dependências necessárias para o servidor e para os testes automatizados:

*npm install express mocha chai chai-http --save-dev*

3. Criação do Servidor HTTP

Para configurar o servidor HTTP, criei um arquivo chamado server.js na raiz do projeto. Esse arquivo utiliza o Express para responder a diferentes rotas, permitindo que o servidor receba requisições e responda de acordo com a URL solicitada.

touch server.js

Dentro de server.js, escrevi o código para criar o servidor e definir as rotas:

*const express = require('express');*

*const app = express();*

*app.use(express.json());*

*app.get('/', (req, res) => {*

*res.send('Hello World');*

*});*

*app.post('/data', (req, res) => {*

*const data = req.body;*

*res.json({ message: 'Sucesso', data });*

*});*

*const PORT = 3000;*

*app.listen(PORT, () => {*

*console.log(`Servidor rodando em http://localhost:${PORT}`);*

*});*

*module.exports = app;*

4. Estrutura do Projeto

Para organizar o código e adicionar funcionalidades ao servidor, criei o arquivo math.js, onde implementei funções matemáticas básicas, como uma função de soma:

*touch math.js*

*function soma(a, b) {*

*return a + b;*

*}*

*module.exports = { soma };*

5. Escrever Testes de Integração

Para validar o funcionamento do servidor e suas rotas, configurei os testes de integração usando Chai e Chai-HTTP:

*const chai = require('chai');*

*const chaiHttp = require('chai-http');*

*const server = require('../server');*

*chai.use(chaiHttp);*

*const { expect } = chai;*

*describe('Testes de Integração do Servidor', () => {*

*it('Deve retornar "Hello World" na rota GET /', (done) => {*

*chai.request(server)*

*.get('/')*

*.end((err, res) => {*

*expect(res).to.have.status(200);*

*expect(res.text).to.equal('Hello World');*

*done();*

*});*

*});*

*it('Deve retornar JSON com mensagem de sucesso na rota POST /data', (done) => {*

*const data = { nome: 'Teste' };*

*chai.request(server)*

*.post('/data')*

*.send(data)*

*.end((err, res) => {*

*expect(res).to.have.status(200);*

*expect(res.body).to.be.an('object');*

*expect(res.body).to.have.property('message').equal('Sucesso');*

*expect(res.body).to.have.property('data').eql(data);*

*done();*

*});*

*});*

*});*

6. Organização dos Testes

Criei uma pasta test para organizar os testes, e dentro dela, adicionei o arquivo integration.test.js para armazenar os testes de integração:

*mkdir test*

*touch test/integration.test.js*

7. Configuração do Script de Teste

Adicionei um script no package.json para executar o Mocha com os testes:

*"scripts": {*

*"test": "mocha"*

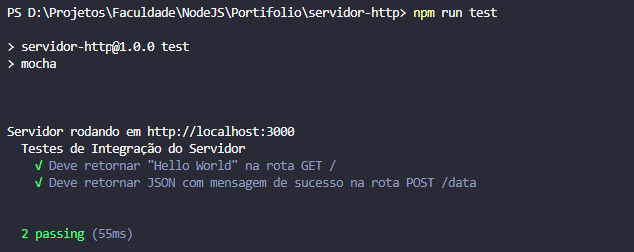
*}*

8. Executando os Testes

Executei os testes para validar o funcionamento da aplicação:

npm test

9. Evidencias



**Todo o código está no repositório do GitHub:**

https://github.com/gabriel3dev/portifolio-nodejs