5.1 Introdução a Frameworks para APIs REST

APIs REST (Representational State Transfer) são a espinha dorsal de muitas aplicações web modernas, permitindo a comunicação eficiente entre clientes (como navegadores ou aplicativos móveis) e servidores. Desenvolver uma API REST do zero seria trabalhoso, exigindo gerenciamento de requisições HTTP, roteamento e formatação de respostas. Para simplificar esse processo, frameworks fornecem estruturas pré-construídas que aceleram o desenvolvimento, garantem consistência e seguem boas práticas. Neste capítulo, focaremos em dois frameworks gratuitos e open-source amplamente utilizados: **Node.js com Express** e **Flask (Python)**, com ênfase no Express para os exemplos práticos, mas incluindo equivalências no Flask para contextualizar.

O que é um Framework?

Um framework é uma biblioteca ou conjunto de ferramentas que abstrai tarefas comuns, como roteamento de URLs, parsing de requisições HTTP, manipulação de respostas JSON e integração com bancos de dados. Em APIs REST, frameworks ajudam a: - Definir endpoints (ex.: `/tarefas`) associados a métodos HTTP (GET, POST, etc.). - Processar corpos de requisições (ex.: JSON) e formatar respostas. - Implementar middlewares para autenticação, validação ou logging. - Gerenciar erros e códigos de status HTTP.

Node.js com Express

- **O que é?** Node.js é um ambiente de execução que permite rodar JavaScript no servidor, fora do navegador. Express é um framework minimalista para Node.js, projetado para criar APIs RESTful de forma rápida e eficiente.
- **Características**:
- Leve e flexível, com uma API simples.
- Suporte nativo para JSON e middlewares (ex.: para validação ou autenticação).
- Grande comunidade, com vasta documentação e pacotes adicionais (ex.: Joi para validação).
- **Vantagens**:
- Rápido para prototipagem e aplicações em produção.
- Integração direta com JavaScript, ideal para equipes que usam front-ends como React ou Vue.

- Amplo suporte em hospedagens como Vercel, Heroku (versão gratuita limitada) ou servidores próprios.
- **Casos de Uso**: APIs para aplicações web, microserviços, integração com bancos de dados NoSQL (ex.: MongoDB) ou SQL (ex.: PostgreSQL).
- **Exemplo Simples**: `javascript const express = require('express'); const app = express(); app.get('/hello', (req, res) => res.json({ mensagem: 'Olá, mundo!' })); app.listen(3000, () => console.log('Servidor rodando na porta 3000'));`

Flask (Python)

- **O que é?** Flask é um microframework para Python, projetado para ser simples, extensível e fácil de aprender. Ele é ideal para desenvolvedores familiarizados com Python ou que trabalham em projetos que integram ciência de dados.
- **Características**:
- Minimalista, permitindo configurações personalizadas.
- Suporte a JSON via biblioteca integrada (`jsonify`).
- Integração com bibliotecas Python, como SQLAlchemy (para bancos de dados) ou Pydantic (para validação).
- **Vantagens**:
- Curva de aprendizado suave, especialmente para iniciantes em Python.
- Flexibilidade para projetos pequenos ou protótipos rápidos.
- Comunidade ativa e suporte a extensões.
- **Casos de Uso**: APIs para aplicações web, protótipos, integração com ferramentas de machine learning.
- **Exemplo Simples**: `python from flask import Flask, jsonify app = Flask(__name__) @app.route('/hello', methods=['GET']) def hello(): return jsonify({'mensagem': 'Olá, mundo!'}) if __name__ == '__main__': app.run(port=3000)`

Por que Express para Este Capítulo?

Escolhemos o **Express** como framework principal para os exemplos devido à sua popularidade, simplicidade e integração com JavaScript, que é amplamente usado em aplicações web modernas. No entanto, incluiremos equivalências no Flask para desenvolvedores que preferem Python, garantindo que o conteúdo seja acessível a diferentes públicos. Ambos são open-source, gratuitos e amplamente suportados,

alinhando-se ao objetivo da apostila.

Outras Ferramentas Open-Source

Além de Express e Flask, outras opções open-source incluem: - **FastAPI (Python)**: Moderno, rápido e com suporte a validação automática via Pydantic. - **Koa (Node.js)**: Alternativa ao Express, com sintaxe mais moderna usando async/await. - **Spring Boot (Java)**: Ideal para sistemas corporativos, mas com curva de aprendizado mais íngreme. Para manter a simplicidade, focaremos em Express, com paralelos ao Flask.

5.2 Configuração do Ambiente de Desenvolvimento

Antes de desenvolver a API, é necessário configurar o ambiente de desenvolvimento com ferramentas open-source. Isso inclui instalar o Node.js, o Express, um editor de código e ferramentas para testar requisições HTTP. Abaixo, detalhamos o processo passo a passo.

Passo 1: Instalação do Node.js

Node.js é necessário para executar o Express. Siga estas etapas: 1. **Download**: - Acesse o site oficial (nodejs.org) e baixe a versão **LTS** (Long Term Support), recomendada para estabilidade. - Disponível para Windows, macOS e Linux. 2. **Instalação**: - Siga as instruções do instalador para seu sistema operacional. - No Windows, o instalador inclui o npm (gerenciador de pacotes do Node.js). - No Linux, use o gerenciador de pacotes (ex.: `sudo apt install nodejs npm` no Ubuntu). 3. **Verificação**: - Abra o terminal e execute: `bash node -v npm -v` - Isso retorna as versões instaladas (ex.: `v20.17.0` para Node.js e `10.8.1` para npm).

Alternativa para Flask

- Instale o Python (versão 3.8 ou superior) em python.org.
- Verifique a instalação: `bash python3 --version pip3 --version`
- Instale o Flask: `bash pip3 install flask`

Passo 2: Configuração do Projeto

Crie um projeto Node.js para a API: 1. **Crie um diretório**: `bash mkdir minha-api-rest cd minha-api-rest` 2. **Inicialize o projeto**: `bash npm init -y` - Isso cria um arquivo `package.json` com configurações padrão: `json { "name": "minha-api-rest", "version": "1.0.0", "main": "src/index.js", "scripts": { "start": "node src/index.js" } }` 3. **Instale o Express**: `bash npm install express` - Isso adiciona o Express como dependência no `package.json`.

Estrutura do Projeto

Crie a seguinte estrutura de diretórios:

```
minha-api-rest/
■■■ node_modules/
■■■ package.json
■■■ package-lock.json
■■■ src/
   ■■■ index.js
```

- O arquivo `src/index.js` será o ponto de entrada da API.

Para Flask

- Crie um diretório: `bash mkdir minha-api-flask cd minha-api-flask`
- Crie um arquivo `app.py` como ponto de entrada.

Passo 3: Ferramentas Adicionais

- **Editor de Código**: Use o **Visual Studio Code** (VS Code), que é gratuito, open-source e suporta extensões como:
- **JavaScript (ES6) code snippets**: Para atalhos de código.
- **REST Client**: Para testar requisições HTTP diretamente no editor.
- **Cliente HTTP**:
- **cURL**: Ferramenta de linha de comando incluída em Linux/macOS e disponível para Windows.
- **Postman**: Versão gratuita para testar APIs.
- **REST Client** (extensão do VS Code): Permite criar arquivos `.http` para requisições.
- **Gerenciador de Dependências**:

- npm para Node.js.
- pip para Python/Flask.

Passo 4: Configuração para Desenvolvimento

- **Nodemon** (opcional, para Node.js):
- Instale o Nodemon para reiniciar o servidor automaticamente durante o desenvolvimento: `bash npm install --save-dev nodemon`
- Adicione ao `package.json`: `json "scripts": { "start": "node src/index.js", "dev": "nodemon src/index.js" }`
- Execute com: `bash npm run dev`
- **Flask Development Server**:
- O Flask inclui um servidor de desenvolvimento embutido que reinicia automaticamente: `bash flask run`

5.3 Criação de uma API REST Básica com Endpoints GET

Agora que o ambiente está configurado, vamos criar uma API REST simples para gerenciar uma lista de livros, com dois endpoints GET: - `GET /livros`: Lista todos os livros. - `GET /livros/:id`: Retorna os detalhes de um livro específico.

Código da API (Express)

Crie o arquivo `src/index.js` com o seguinte conteúdo:

```
// Importa o Express
const express = require('express');
const app = express();

// Define a porta do servidor
const PORT = 3000;

// Dados fictícios para simular um banco de dados
const livros = [
 { id: 1, titulo: "Dom Quixote", autor: "Miguel de Cervantes", ano: 1605
 { id: 2, titulo: "1984", autor: "George Orwell", ano: 1949 },
 { id: 3, titulo: "O Senhor dos Anéis", autor: "J.R.R. Tolkien", ano: 195
];
```

```
// Middleware para parsear JSON (necessário para futuros capítulos)
app.use(express.json());

// Endpoint GET para listar todos os livros
app.get('/livros', (req, res) => {
 res.json(livros);
});

// Endpoint GET para obter um livro por ID
app.get('/livros/:id', (req, res) => {
 const id = parseInt(req.params.id);
 if (isNaN(id)) {
   return res.status(400).json({ erro: "ID deve ser um número" });
 }
 const livro = livros.find(l => l.id === id);
 if (!livro) {
   return res.status(404).json({ erro: "Livro não encontrado" });
 }
 res.json(livro);
});

// Inicia o servidor
app.listen(PORT, () => {
 console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Explicação do Código

1. **Importação do Express**:
2. `const express = require('express')`: Carrega o módulo Express.
3. `const app = express()`: Cria uma instância do aplicativo.
4. **Porta do Servidor**:
5. `const PORT = 3000`: Define a porta (pode ser alterada, evitando conflitos).
6. **Dados Fictícios**:
7. O array `livros` simula um banco de dados. No Capítulo 6, conectaremos a um banco real.
8. **Middleware**:

9. `app.use(express.json())`: Processa corpos JSON (necessário para POST/PUT no próximo capítulo).
10. **Endpoints GET**:
11. `/livros`: Retorna a lista completa de livros em JSON.
12. `/livros/:id`: Usa um parâmetro dinâmico (`:id`) para buscar um livro específico. Valida o ID e retorna erro 404 se não encontrado.
13. **Iniciar o Servidor**:
14. `app.listen(PORT, ...)`: Inicia o servidor na porta especificada.

Testando a API

1. **Inicie o servidor**: `bash node src/index.js` Ou, com Nodemon: `bash npm run dev`
2. **Teste com cURL**:
3. Listar todos os livros: `bash curl http://localhost:3000/livros` **Saída**: `json [ { "id": 1, "titulo": "Dom Quixote", "autor": "Miguel de Cervantes", "ano": 1605 }, { "id": 2, "titulo": "1984", "autor": "George Orwell", "ano": 1949 }, { "id": 3, "titulo": "O Senhor dos Anéis", "autor": "J.R.R. Tolkien", "ano": 1954 } ]`
4. Obter um livro específico: `bash curl http://localhost:3000/livros/2` **Saída**: `json { "id": 2, "titulo": "1984", "autor": "George Orwell", "ano": 1949 }`
5. Testar erro: `bash curl http://localhost:3000/livros/999` **Saída**: `json { "erro": "Livro não encontrado" }`
6. **Teste com REST Client** (VS Code):

   Crie um arquivo `test.http`: ``` GET http://localhost:3000/livros HTTP/1.1

   ###

   GET http://localhost:3000/livros/2 HTTP/1.1 ``` - Clique em “Send Request” na extensão REST Client para executar.

Equivalência com Flask

Para quem prefere Python, aqui está a API equivalente em Flask:

```
from flask import Flask, jsonify, request

app = Flask(__name__)

livros = [
   {"id": 1, "titulo": "Dom Quixote", "autor": "Miguel de Cervantes", "an
   {"id": 2, "titulo": "1984", "autor": "George Orwell", "ano": 1949},
   {"id": 3, "titulo": "O Senhor dos Anéis", "autor": "J.R.R. Tolkien", "
]

@app.route('/livros', methods=['GET'])
def get_livros():
   return jsonify(livros)

@app.route('/livros/<int:id>', methods=['GET'])
def get_livro(id):
   for livro in livros:
       if livro['id'] == id:
           return jsonify(livro)
   return jsonify({"erro": "Livro não encontrado"}), 404

if __name__ == '__main__':
   app.run(port=3000, debug=True)
```

- **Executar**: `bash flask run`
- **Testar**: `bash curl http://localhost:5000/livros`

Boas Práticas Aplicadas

- **Interface Uniforme**: URIs claras (`/livros`, `/livros/:id`) e uso de GET para leitura.
- **Sem Estado**: Cada requisição é independente, com dados fictícios no servidor.
- **Respostas JSON**: Formato padrão, conforme Capítulo 4.
- **Códigos de Status**: 200 (OK), 400 (Bad Request), 404 (Not Found).

5.4 Atividade Prática: Desenvolvimento de uma API para Listar Recursos

Nesta atividade, você desenvolverá uma API REST para gerenciar uma lista de tarefas, com três endpoints GET: - `GET /tarefas`: Lista todas as tarefas. - `GET /tarefas/:id`: Retorna uma tarefa específica. - `GET /tarefas/pendentes`: Lista tarefas não concluídas.

Requisitos

- Use **Node.js com Express**.
- Armazene dados em um array (sem banco de dados).
- Retorne respostas em JSON com códigos de status apropriados.
- Inclua validação básica (ex.: ID inválido).
- Adicione links HATEOAS nas respostas.

Solução

Crie o arquivo `src/index.js`:

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.json());

const tarefas = [
 { id: 1, titulo: "Estudar REST APIs", concluida: false, prioridade: "alt
 { id: 2, titulo: "Fazer compras", concluida: true, prioridade: "média" }
 { id: 3, titulo: "Correr 5km", concluida: false, prioridade: "baixa" }
];

// Endpoint GET para listar todas as tarefas
app.get('/tarefas', (req, res) => {
 const tarefasComLinks = tarefas.map(tarefa => ({
   ...tarefa,
   _links: {
     self: `/tarefas/${tarefa.id}`,
     pendentes: `/tarefas/pendentes`
   }
 }));
 res.json({
```

```
    tarefas: tarefasComLinks,
    meta: {
      total: tarefas.length,
      page: 1,
      limit: 10
    }
  });
});

// Endpoint GET para obter uma tarefa por ID
app.get('/tarefas/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const tarefa = tarefas.find(t => t.id === id);
  if (!tarefa) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  res.json({
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      all: `/tarefas`,
      pendentes: `/tarefas/pendentes`
    }
  });
});

// Endpoint GET para listar tarefas pendentes
app.get('/tarefas/pendentes', (req, res) => {
  const tarefasPendentes = tarefas.filter(t => !t.concluida).map(tarefa =>
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      all: `/tarefas`
    }
  }));
```

```
 res.json({
   tarefas: tarefasPendentes,
   meta: {
     total: tarefasPendentes.length,
     page: 1,
     limit: 10
   }
 });
});

// Inicia o servidor
app.listen(PORT, () => {
 console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Explicação

- **Endpoints**:
- `/tarefas`: Retorna todas as tarefas com metadados de paginação.
- `/tarefas/:id`: Retorna uma tarefa específica, com validação de ID.
- `/tarefas/pendentes`: Filtra tarefas não concluídas.
- **HATEOAS**: Cada resposta inclui links para recursos relacionados (`self`, `all`, `pendentes`).
- **Metadados**: Inclui informações de paginação (`total`, `page`, `limit`).
- **Validação**: Verifica se o ID é um número válido.
- **Códigos de Status**: Usa 200 (OK), 400 (Bad Request) e 404 (Not Found).

Testando

1. **Inicie o servidor**: `bash npm run dev`
2. **Testes com cURL**:
3. Listar todas as tarefas: `bash curl http://localhost:3000/tarefas` **Saída**: `json { "tarefas": [ { "id": 1, "titulo": "Estudar REST APIs", "concluida": false, "prioridade": "alta", "_links": { "self": "/tarefas/1", "pendentes": "/tarefas/pendentes" } }, ... ], "meta": { "total": 3, "page": 1, "limit": 10 } }`
4. Obter tarefa específica: `bash curl http://localhost:3000/tarefas/1` **Saída**: `json { "id": 1, "titulo": "Estudar REST APIs",`

"concluida": false, "prioridade": "alta", "_links": { "self": "/tarefas/1", "all": "/tarefas", "pendentes": "/tarefas/pendentes" } }

5. Listar tarefas pendentes: bash curl http://localhost:3000/tarefas/pendentes **Saída**: json { "tarefas": [{ "id": 1, "titulo": "Estudar REST APIs", "concluida": false, "prioridade": "alta", "_links": { "self": "/tarefas/1", "all": "/tarefas" } }, { "id": 3, "titulo": "Correr 5km", "concluida": false, "prioridade": "baixa", "_links": { "self": "/tarefas/3", "all": "/tarefas" } }], "meta": { "total": 2, "page": 1, "limit": 10 } }

Desafio Extra

Adicione um endpoint `GET /tarefas/prioridade/:nivel` que lista tarefas por prioridade (ex.: `alta`, `média`, `baixa`).

Solução

Adicione ao `index.js`:

```
app.get('/tarefas/prioridade/:nivel', (req, res) => {
 const nivel = req.params.nivel.toLowerCase();
 const niveisValidos = ['alta', 'média', 'baixa'];
 if (!niveisValidos.includes(nivel)) {
   return res.status(400).json({ erro: "Nível de prioridade inválido" });
 }
 const tarefasFiltradas = tarefas
   .filter(t => t.prioridade.toLowerCase() === nivel)
   .map(tarefa => ({
     ...tarefa,
     _links: {
       self: `/tarefas/${tarefa.id}`,
       all: `/tarefas`
     }
   }));
 res.json({
   tarefas: tarefasFiltradas,
   meta: {
```

```
      total: tarefasFiltradas.length,
      page: 1,
      limit: 10
    }
  });
});
```

- **Teste**: `bash curl http://localhost:3000/tarefas/prioridade/alta` **Saída**: `json { "tarefas": [ { "id": 1, "titulo": "Estudar REST APIs", "concluida": false, "prioridade": "alta", "_links": { "self": "/tarefas/1", "all": "/tarefas" } } ], "meta": { "total": 1, "page": 1, "limit": 10 } }`

Conclusão

Este capítulo introduziu o desenvolvimento de APIs REST com Node.js e Express, configurando o ambiente, criando endpoints GET e aplicando boas práticas, como HATEOAS e metadados de paginação. A atividade prática consolidou esses conceitos, criando uma API funcional para gerenciar tarefas. O Capítulo 6 expandirá essa API com métodos POST, PUT e DELETE, além de validação e tratamento de erros, conectando-se diretamente às bases teóricas dos Capítulos 3 e 4.

Dicas para Continuar

- Adicione mais campos às tarefas (ex.: `prazo`, `categoria`).
- Teste a API com Postman ou REST Client para explorar diferentes cenários.
- Pesquise sobre middlewares no Express (ex.: logging de requisições).

Conexão com o Próximo Capítulo

O Capítulo 6 continuará o desenvolvimento da API de tarefas, adicionando endpoints para criação, atualização e exclusão de recursos, além de implementar validação de dados com Joi e tratamento de erros robusto, aplicando os conceitos de modelagem de dados do Capítulo 4.