Desenvolvendo uma API RESTful com Node.js e Express

O que é uma API RESTful?

API (Interface de Programação de Aplicações) é um conjunto de regras que permite que diferentes aplicações se comuniquem entre si.

REST (Representational State Transfer) é um estilo arquitetural que utiliza o protocolo HTTP para comunicação de dados.

- Não é um **protocolo** nem uma **tecnologia**, mas sim um **estilo arquitetural** para construção de sistemas distribuídos.
- Muito usado na web porque se baseia em **HTTP**.

O que é uma API RESTful?

Uma API RESTful segue os princípios REST:

Cliente-Servidor:

Separa os interesses do cliente (interface do utilizador) dos interesses do servidor (armazenamento de dados), garantindo que as mudanças num dos lados não afetem o outro.

Sem Estado (Stateless):

Cada solicitação do cliente para o servidor deve conter toda a informação necessária para ser compreendida e processada, sem depender de qualquer estado armazenado no servidor.

Sistema em Camadas:

A arquitetura pode ser composta por camadas hierárquicas. Cada componente não tem conhecimento da estrutura das outras camadas, o que melhora a escalabilidade e a manutenção da aplicação, como no caso do padrão MVC (Modelo-Visão-Controlador).

Representatividade:

Os recursos podem ser representados em diferentes formatos (JSON, XML, HTML, etc).

Cliente-Servidor

O cliente (frontend) e o servidor (backend) são separados, permitindo evolução independente.

API REST na Prática

Considere um sistema de gerenciamento de uma biblioteca:

Recursos

Entidades importantes do seu sistema

- Livros
- Autores
- Usuários

Endpoints

URLs que representam recursos

- /livros
- /autores
- /usuarios

Ações

Operações sobre recursos

- Obter todos os livros
- Adicionar um novo autor
- Atualizar dados de usuário

Uma API bem projetada é intuitiva e fácil de entender!

Entendendo CRUD

CRUD representa as quatro operações básicas que podem ser realizadas em qualquer recurso persistente:

1

Create (Criar)

Adiciona novos dados ao sistema usando o método HTTP **POST**

Exemplo: Cadastrar um novo livro na biblioteca

2

Read (Ler)

Recupera dados existentes usando o método HTTP **GET**

Exemplo: Visualizar informações de todos os livros ou de um livro específico

3

Update (Atualizar)

Modifica dados existentes usando os métodos HTTP **PUT** ou **PATCH**

Exemplo: Atualizar o número de páginas de um livro

4

Delete (Excluir)

Remove dados usando o método HTTP **DELETE**

Exemplo: Remover um livro do catálogo

Estrutura do Projeto: Arquitetura MVC

Para organizar o código da nossa API de forma eficiente, adotaremos o padrão arquitetural MVC (Model-View-Controller). Ele promove a separação de responsabilidades, tornando o desenvolvimento mais modular e fácil de manter.

1

Model (Modelo)

Responsável pela lógica de dados e interação com o banco de dados (ou dados em memória). Define a estrutura dos dados e as operações possíveis.

- Responsável pela **lógica de dados**.
- Representa as **entidades** do sistema (ex.: Usuário, Livro, Pedido).
- Se conecta ao **banco de dados**.

2

View (Visualização)

Em APIs REST, a "View" lida com a apresentação dos dados ao cliente, geralmente formatando a resposta em JSON ou XML. É como os dados são "vistos" pelo consumidor da API.

- Responsável pela **interface com o usuário**.
- Em aplicações web tradicionais (HTML, CSS, templates).
- Na APIs REST a "view" geralmente é a **resposta JSON** enviada ao cliente.

3

Controller (Controlador)

Recebe as requisições, coordena a lógica de negócio, interage com o Modelo e prepara os dados para a Visualização. Atua como o intermediário.

- Responsável por **receber as requisições** e decidir o que fazer.
- Faz a ponte entre **Model** e **View**.
- Contém a lógica de **negócio e controle**.

Estrutura do Projeto: Arquitetura MVC

Essa separação nos ajuda a manter o código limpo, testável e escalável.

- Facilitar a **manutenção** e **evolução** do sistema.
- Melhorar a **organização** do código.
- Permitir que **equipes diferentes** trabalhem em paralelo (ex.: front-end cuida das views, back-end cuida do model).
- Aumentar o **reuso** de código.

Fluxo do MVC

1. O **usuário** faz uma requisição
2. O **Controller** recebe a requisição e consulta o **Model**.
3. O **Model** busca/processa os dados.
4. O **Controller** envia a resposta para a **View**.
5. O cliente recebe os dados formatados (JSON, HTML, etc).

Fluxo resumido dentro dessa estrutura

1. **O cliente faz a requisição**
2. A **rota** em `routes/exemploRoutes.js` captura a URL.
3. A rota chama a função no **controller** (`exemploController.js`).
4. O controller pede dados ao **model** (`exemploModel.js`).
5. O model retorna os dados → controller organiza a resposta.
6. A resposta é enviada (pela **view** ou direto com `res.json`).

Estrutura de um Projeto

Vamos organizar nosso projeto de forma simples e escalável:

```
api/
  ├── node_modules/
  ├── src/
  │    ├── controllers/
  │    │    └── tarefasController.js
  │    ├── routes/
  │    │    └── tarefasRoutes.js
  │    ├── models/
  │    │    └── tarefasModel.js
  │    ├── view/
  │    │    └── formularioTarefas.html
  │    └── app.js
  ├── package.json
  └── server.js
```

Criando o Servidor

Vamos começar criando nosso arquivo `server.js` na raiz do projeto:

```
// server.js
const app = require('./src/app');

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
});
```

Criando o Servidor

E agora, o arquivo `app.js` dentro da pasta `src`:

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

const methodOverride = require('method-override');
app.use(methodOverride('_method'))

const path = require('path');
app.use(express.static(path.join(__dirname, 'view')));

// Configurando middlewares
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

module.exports = app;
```

Criando o Modelo de Tarefas

Para começar, vamos criar um modelo simples para nossas tarefas em `src/models/taskModel.js`:

```
let tarefas = [
 { id: 2, titulo: Estudar', feita: 0},
];

// Funções para manipular as tarefas
const getTodasTarefas  = () => tarefas;

const getTarefaId = (id) => tarefas.find(task => task.id === id);

const getFeitas = () => {
 return tarefa.find(item => item.feito === 1)
};

const criarTarefa = (taskData) => {
 const newTask = {
 id: tasks.length > 0 ? Math.max(...tasks.map(t => t.id)) + 1 : 1,
 title: taskData.title,
 completed: taskData.completed || false
 };
 tasks.push(newTask);
 return newTask;
};
```

```
module.exports = {
 getTodasTarefas ,
 getTarefaId ,
 getFeitas ,
 criarTarefa
};
```

Criando o Controlador

Agora vamos criar nosso controlador em `src/controllers/taskController.js`:

```
const taskModel = require('../models/taskModel');

// GET /tarefas - Listar todas as tarefas
const getTarefas = (req, res) => {
 const tasks = taskModel.getTodasTarefas();
 res.json(tarefas);
};

// GET /tarefas/:id - Obter uma tarefa específica
const getTarefasId = (req, res) => {
 const id = parseInt(req.params.id);
 const tarefa = taskModel.getTarefaId(id);

 if (!tarefa) {
 res.status(404).json({ erro: 'Tarefa não encontrada' });
 }
 res.json(tarefas);
};
```

```
// GET /tasks - Listar as tarefas Feitas
const getTarefasFeitas = (req, res) => {
 const tasks = taskModel.getFeitas ();
 res.json(tarefas);
};

const paginaInicial  =  (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'form.html'))
};

// POST /tasks - Criar uma nova tarefa
const criarTarefa = (req, res) => {
 const nova = taskModel.criarTarefa(req.body);
 res.status(201).json(nova);
};

module.exports = {
 getTarefas ,
 getTarefasId,
 getTarefasFeitas,
 criarTarefa
};
```

Definindo as Rotas

Agora vamos criar nossas rotas em `src/routes/taskRoutes.js`:

```
const express = require('express');
const router = express.Router();
const taskController = require('../controllers/taskController');

// Definindo as rotas para as operações CRUD
router.get('/tarefas', tarefaController.getTarefas);
router.get('/tarefas/:id', tarefaController.getTarefasId);
router.get('tarefasFeitas/', tarefaController.getTarefasFeitas );
router.post('tarefas/', tarefaController.criarTarefa );
router.get('/', tarefaController.paginaInicial);

module.exports = router;
```

Em seguida, atualizamos o `app.js` para incluir estas rotas:

```
const tarefasRoutes = require('./routes/tarefasRoutes');
app.use('/tarefas', tarefasRoutes);
```

CRUD em Ação: Operação CREATE

Criando uma Nova Tarefa (POST)

Para criar um novo registro usando o método POST:

```
// Requisição POST para /tasks
{
  "titulo": "Estudar Express",
  "feito": 0
}
```

O controlador chama *criarTarefa* do modelo, que adiciona a tarefa ao "banco de dados" e retorna o novo objeto com status 201 (Created).

CRUD em Ação: Operação READ

Existem duas maneiras de ler dados da nossa API:

Listar Todas as Tarefas

Método HTTP: **GET**

URL: `/tarefas`

Resposta: Array com todas as tarefas

```
[
  { id: 1, titulo: "Aprender Node.js", feito: 0},
  { id: 2, titulo: "Criar uma API REST", feito: 1},
  ...
]
```

Obter Uma Tarefa Específica

Método HTTP: **GET**

URL: `/tarefas/:id` (ex: `/tasks/1`)

Resposta: Objeto da tarefa ou erro 404

```
{ id: 1, titulo: "Aprender Node.js", feito: 1}
```

O controlador chama *getTarefas* ou *getTarefaId* do modelo, que retorna todas as tarefas ou a tarefa com id identificado (ocasionando o erro 404 se o id não existir).

CRUD em Ação: Operação UPDATE

Atualizando uma Tarefa (PUT)

Para atualizar um registro existente:

O controlador chama `editarTarefa` do modelo, que atualiza a tarefa no "banco de dados" e retorna o objeto atualizado.

Se a tarefa não existir, retornamos um erro 404.

O método PUT geralmente substitui todo o recurso, enquanto o PATCH (não implementado neste exemplo) permite atualizações parciais.

CRUD em Ação: Operação DELETE

Removendo uma Tarefa (DELETE)

Para excluir um registro existente:

Método HTTP: **DELETE**

URL: `/tarefa/:id` (ex: `/tarefa/2`)

Resposta: Código 204 (No Content) se bem-sucedido, ou 404 se a tarefa não existir.

O controlador chama `deletarTarefa` do modelo, que remove a tarefa do "banco de dados" e não retorna conteúdo na resposta.

Seguindo as convenções REST, uma requisição DELETE bem-sucedida não deve retornar corpo de resposta.

Boas Práticas: Estruturação e Nomeação

Seguir boas práticas torna sua API mais profissional, consistente e fácil de usar:

Nomeação de Recursos

- Use substantivos no plural para coleções (/tarefas)
- Use substantivos no singular ou IDs para itens específicos (/tarefa/1)
- Evite verbos nas URLs (/getTasks)

Versionamento

- Use prefixo de versão (/api/v1/tarefas)
- Mantenha compatibilidade com versões anteriores
- Documente mudanças entre versões

Códigos de Status

- 200: OK (sucesso)
- 201: Created (recurso criado)
- 400: Bad Request (erro do cliente)
- 404: Not Found (recurso não existe)
- 500: Internal Server Error (erro do servidor)

Boas Práticas: Segurança e Validação

A segurança é fundamental para qualquer API em produção:

Validação de Dados

- Valide TODOS os dados de entrada
- Defina tipos e limites para cada campo
- Use bibliotecas como Joi ou express-validator
- Trate erros de validação de forma clara

Documentação da API

Uma boa documentação é crucial para que outros desenvolvedores possam usar sua API com facilidade:

Swagger/OpenAPI

Padrão para documentar APIs RESTful:

- Descrição de endpoints, parâmetros e respostas
- Geração de documentação bonita e funcional