

## Material de apoio

Site: [Geração Tech](#)  
Curso: Formação em Desenvolvedor Web - Online  
Livro: Material de apoio

Impresso por: JOÃO VITOR DE MELO FREITAS  
Data: sexta-feira, 2 ago. 2024, 08:41

# Índice

## **1. Back-End com Node.js**

1.1. Implementação de REST API com Express

1.2. Vídeo Aula

1.3. Implementação de CRUD

1.4. Vídeo Aula

1.5. Implementação de CRUD

1.6. Vídeo Aula

# 1. Back-End com Node.js

## Revisão da Aula Anterior

Na última aula, aprendemos a:

- Criar classes e métodos estáticos em JavaScript.
- Implementar herança entre classes.
- Utilizar métodos HTTP para manipulação de dados (CRUD).

## 1.1. Implementação de REST API com Express

### Configuração Inicial

#### Instalando Dependências

Primeiro, precisamos instalar o Node.js, npm e o Express. Para isso, abra seu terminal e execute os seguintes comandos:

```
bash
```


 Copiar código

```
npm init -y npm install express
```

#### Estrutura do Projeto

Vamos criar a seguinte estrutura de pastas e arquivos para o nosso projeto:

```
go
```

 Copiar código

```
my-backend-project/ | ├── node_modules/ |── src/ |── controllers/ | | └── produtoController.js |── models/ | | └── produto.js |── routes/ | | └── produtoRoutes.js |── app.js |── server.js |── package.json |── package-lock.json
```

#### Configurando o Servidor

No arquivo `server.js`, configure o servidor básico do Express:

```
javascript
```

 Copiar código

```
const express = require('express'); const app = express(); const produtoRoutes = require('./routes/produtoRoutes'); app.use(express.json()); app.use('/produtos', produtoRoutes); const PORT = 3000; app.listen(PORT, () => { console.log(`Servidor rodando na porta ${PORT}`); });
```

#### Definindo Rotas

No arquivo `routes/produtoRoutes.js`, configure as rotas para a API de produtos:

```
javascript
```


 Copiar código

```
const express = require('express'); const router = express.Router(); const produtoController = require('../controllers/produtoController'); router.get('/', produtoController.listar); router.post('/', produtoController.adicionar); router.put('/:id', produtoController.editar); router.delete('/:id', produtoController.excluir); module.exports = router;
```

#### Criando Controladores

No arquivo `controllers/produtoController.js`, implemente a lógica para manipulação dos produtos:

```
javascript
```

 Copiar código

```
const Produto = require('../models/produto'); exports.listar = (req, res) => { res.status(200).json(Produto.listar()); }; exports.adicionar = (req, res) => { const novoProduto = req.body; Produto.adicionar(novoProduto); res.status(201).json({ message: 'Produto adicionado com sucesso' }); }; exports.editar = (req, res) => { const id = req.params.id; const dadosAtualizados = req.body; Produto.editar(id, dadosAtualizados); res.status(200).json({ message: 'Produto editado com sucesso' }); }; exports.excluir = (req, res) => { const id = req.params.id; Produto.excluir(id); res.status(200).json({ message: 'Produto excluído com sucesso' }); };
```

## Modelando a Classe Produto

No arquivo `models/produto.js`, defina a classe `Produto` com seus métodos estáticos:

javascript



```
class Produto { static produtos = []; static listar() { return this.produtos; } static adicionar(produto) {
this.produtos.push(produto); } static editar(id, dadosAtualizados) { const index = this.produtos.findIndex(p => p.id == id);
if (index !== -1) { this.produtos[index] = { ...this.produtos[index], ...dadosAtualizados }; } } static excluir(id) {
this.produtos = this.produtos.filter(p => p.id != id); } } module.exports = Produto;
```

## Testando a API Iniciando o Servidor

Para iniciar o servidor, execute o comando:

bash



```
node src/server.js
```

## Testando as Rotas

Utilize uma ferramenta como Postman ou Insomnia para testar as seguintes rotas:

- **GET /produtos:** Lista todos os produtos.
- **POST /produtos:** Adiciona um novo produto.
- **PUT /produtos/**  
: Atualiza o produto com o ID especificado.
- **DELETE /produtos/**  
: Remove o produto com o ID especificado.

## Conclusão

Nesta aula, aprendemos a configurar um servidor Express, definir rotas e controladores, e modelar uma classe para gerenciar produtos.

Utilizamos métodos estáticos para manipular dados de forma eficiente e organizada.

Pratiquem esses conceitos e explorem novas funcionalidades para aprimorar suas habilidades em desenvolvimento back-end.

Até a próxima aula!

## 1.2. Vídeo Aula

dia 6 express 1 converted



## 1.3. Implementação de CRUD

### Introdução

Olá pessoal, tudo bem?


Vamos continuar nossa aula de back-end utilizando o framework Express. Na aula passada, configuramos o Express e criamos rotas básicas para produtos.

Agora, vamos expandir nossa aplicação para incluir operações CRUD (Create, Read, Update, Delete) e organizar nosso código utilizando a arquitetura MVC (Model-View-Controller).

### Estrutura do Projeto

Para começar, vamos organizar a estrutura do nosso projeto conforme a arquitetura MVC. A estrutura será a seguinte:

go


 Copiar código

```
my-backend-project/ | ├── node_modules/ | ├── src/ | ├── controllers/ | | └── usuarioController.js | ├── models/ | | └── usuarioModel.js | ├── routes/ | | └── usuarioRoutes.js | ├── app.js | ├── server.js | ├── package.json | └── package-lock.json
```

### Configurando o Servidor

No arquivo `server.js`, configuramos o servidor básico do Express:

javascript


 Copiar código

```
const express = require('express'); const app = express(); const usuarioRoutes = require('./routes/usuarioRoutes'); app.use(express.json()); app.use('/usuarios', usuarioRoutes); const PORT = 3000; app.listen(PORT, () => { console.log(`Servidor rodando na porta ${PORT}`); });
```

### Definindo Rotas

No arquivo `routes/usuarioRoutes.js`, configuramos as rotas para a API de usuários:

javascript

 Copiar código

```
const express = require('express'); const router = express.Router(); const usuarioController = require('../controllers/usuarioController'); router.get('/', usuarioController.listar); router.get('/:id', usuarioController.consultarPorId); router.post('/', usuarioController.criar); router.put('/:id', usuarioController.atualizar); router.delete('/:id', usuarioController.excluir); module.exports = router;
```

### Criando Controladores

No arquivo `controllers/usuarioController.js`, implementamos a lógica para manipulação dos usuários:

javascript

 Copiar código

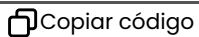
```
const Usuario = require('../models/usuarioModel'); class UsuarioController { static listar(req, res) { res.status(200).json(Usuario.listar()); } static consultarPorId(req, res) { const id = req.params.id; const usuario = Usuario.consultarPorId(id); if (usuario) { res.status(200).json(usuario); } else { res.status(404).json({ message: 'Usuário não encontrado' }); } } static criar(req, res) { const novoUsuario = req.body; Usuario.criar(novoUsuario); res.status(201).json({ message: 'Usuário criado com sucesso' }); } static atualizar(req, res) { const id = req.params.id; const dadosAtualizados = req.body; Usuario.atualizar(id, dadosAtualizados); res.status(200).json({ message: 'Usuário
```

```
atualizado com sucesso' }); } static excluir(req, res) { const id = req.params.id; Usuario.excluir(id);  
res.status(200).json({ message: 'Usuário excluído com sucesso' }); } } module.exports = UsuarioController;
```

## Modelando a Classe Usuario

No arquivo `models/usuarioModel.js`, definimos a classe `Usuario` com seus métodos estáticos:

javascript



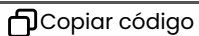
```
class Usuario { static usuarios = []; static listar() { return this.usuarios; } static consultarPorId(id) { return  
this.usuarios.find(usuario => usuario.id == id); } static criar(usuario) { this.usuarios.push(usuario); } static  
atualizar(id, dadosAtualizados) { const index = this.usuarios.findIndex(usuario => usuario.id == id); if (index !== -1) {  
this.usuarios[index] = { ...this.usuarios[index], ...dadosAtualizados }; } } static excluir(id) { this.usuarios =  
this.usuarios.filter(usuario => usuario.id !== id); } } module.exports = Usuario;
```

## Testando a API

### Iniciando o Servidor

Para iniciar o servidor, execute o comando:

bash



```
node src/server.js
```

### Testando as Rotas

Utilize uma ferramenta como Postman ou Insomnia para testar as seguintes rotas:

- **GET /usuarios:** Lista todos os usuários.
- **GET /usuarios/**

: Consulta um usuário por ID.

- **POST /usuarios:** Cria um novo usuário.
- **PUT /usuarios/**

: Atualiza o usuário com o ID especificado.

- **DELETE /usuarios/**

: Remove o usuário com o ID especificado.

## Conclusão

Nesta aula, configuramos um servidor Express, definimos rotas e controladores, e modelamos uma classe para gerenciar usuários.

Utilizamos a arquitetura MVC para organizar nosso código de forma eficiente e facilitar a manutenção.

Pratiquem esses conceitos e explorem novas funcionalidades para aprimorar suas habilidades em desenvolvimento back-end.

Até a próxima aula!



## 1.4. Vídeo Aula

dia 6 express 2 converted



## 1.5. Implementação de CRUD

### Introdução

Olá pessoal, tudo bem? Vamos continuar nossa aula de back-end utilizando o framework Express.


Na aula passada, criamos os models com métodos para listar, consultar por ID, criar, atualizar e deletar. Também implementamos os controllers, que puxam os models de usuários para realizar essas operações.

Agora, vamos alinhar melhor nosso código, trabalhar com métodos estáticos e simular uma base de dados para nossos usuários.

### Estrutura do Projeto

Para organizar nosso projeto conforme a arquitetura MVC (Model-View-Controller), a estrutura será a seguinte:

go


 Copiar código

```
my-backend-project/ | ├── node_modules/ | ├── src/ | ├── controllers/ | | └── usuarioController.js | ├── models/ | | └── usuarioModel.js | ├── routes/ | | └── usuarioRoutes.js | ├── app.js | ├── server.js | ├── package.json | └── package-lock.json
```

### Configurando o Servidor

No arquivo `server.js`, configuramos o servidor básico do Express:

javascript


 Copiar código

```
const express = require('express'); const app = express(); const usuarioRoutes = require('./routes/usuarioRoutes');
app.use(express.json()); app.use('/usuarios', usuarioRoutes); const PORT = 3000; app.listen(PORT, () => {
console.log(`Servidor rodando na porta ${PORT}`); });
```

### Definindo Rotas

No arquivo `routes/usuarioRoutes.js`, configuramos as rotas para a API de usuários:

javascript

 Copiar código

```
const express = require('express'); const router = express.Router(); const usuarioController =
require('../controllers/usuarioController'); router.get('/', usuarioController.listar); router.get('/:id',
usuarioController.consultarPorId); router.post('/', usuarioController.criar); router.put('/:id',
usuarioController.atualizar); router.delete('/:id', usuarioController.excluir); module.exports = router;
```

### Criando Controladores

No arquivo `controllers/usuarioController.js`, implementamos a lógica para manipulação dos usuários:

javascript

 Copiar código

```
const Usuario = require('../models/usuarioModel'); class UsuarioController { static listar(req, res) {
res.status(200).json(Usuario.listar()); } static consultarPorId(req, res) { const id = req.params.id; const usuario =
Usuario.consultarPorId(id); if (usuario) { res.status(200).json(usuario); } else { res.status(404).json({ message: 'Usuário
não encontrado' }); } } static criar(req, res) { const novoUsuario = req.body; Usuario.criar(novoUsuario);
res.status(201).json({ message: 'Usuário criado com sucesso' }); } static atualizar(req, res) { const id = req.params.id;
const dadosAtualizados = req.body; Usuario.atualizar(id, dadosAtualizados); res.status(200).json({ message: 'Usuário
```

```
atualizado com sucesso' }); } static excluir(req, res) { const id = req.params.id; Usuario.excluir(id);  
res.status(200).json({ message: 'Usuário excluído com sucesso' }); } } module.exports = UsuarioController;
```

## Modelando a Classe Usuario

No arquivo `models/usuarioModel.js`, definimos a classe `Usuario` com seus métodos estáticos:

javascript

 Copiar código


```
class Usuario { static usuarios = []; static listar() { return this.usuarios; } static consultarPorId(id) { return  
this.usuarios.find(usuario => usuario.id == id); } static criar(usuario) { this.usuarios.push(usuario); } static  
atualizar(id, dadosAtualizados) { const index = this.usuarios.findIndex(usuario => usuario.id == id); if (index !== -1) {  
this.usuarios[index] = { ...this.usuarios[index], ...dadosAtualizados }; } } static excluir(id) { this.usuarios =  
this.usuarios.filter(usuario => usuario.id !== id); } } module.exports = Usuario;
```

## Testando a API

### Iniciando o Servidor

Para iniciar o servidor, execute o comando:

bash

 Copiar código

```
node src/server.js
```

### Testando as Rotas

Utilize uma ferramenta como Postman ou Insomnia para testar as seguintes rotas:

- **GET /usuarios**: Lista todos os usuários.
- **GET /usuarios/**

: Consulta um usuário por ID.

- **POST /usuarios**: Cria um novo usuário.
- **PUT /usuarios/**

: Atualiza o usuário com o ID especificado.

- **DELETE /usuarios/**

: Remove o usuário com o ID especificado.

## Implementando Persistência de Dados

### Configurando Dados de Teste

Para simular uma base de dados, vamos criar uma lista estática de usuários no model:

javascript

 Copiar código

```
class Usuario { static usuarios = [ { id: 1, nome: 'Maria', login: 'maria' }, { id: 2, nome: 'João', login: 'joao' } ]; //  
métodos listados anteriormente... }
```

### Atualizando Métodos do Model

Os métodos do model são responsáveis por manipular os dados dos usuários:

javascript

 Copiar código

```
class Usuario { static usuarios = [ { id: 1, nome: 'Maria', login: 'maria' }, { id: 2, nome: 'João', login: 'joao' } ];
static listar() { return this.usuarios; } static consultarPorId(id) { return this.usuarios.find(usuario => usuario.id ==
id); } static criar(usuario) { usuario.id = this.usuarios.length + 1; this.usuarios.push(usuario); } static atualizar(id,
dadosAtualizados) { const index = this.usuarios.findIndex(usuario => usuario.id == id); if (index !== -1) {
this.usuarios[index] = { ...this.usuarios[index], ...dadosAtualizados }; } } static excluir(id) { this.usuarios =
this.usuarios.filter(usuario => usuario.id !== id); } } module.exports = Usuario;
```

## Testando o Método POST

Para testar a criação de usuários, siga estes passos:

1. Abra o Insomnia ou Postman.
2. Crie uma nova requisição POST para <http://localhost:3000/usuarios>.
3. No corpo da requisição, adicione um usuário no formato JSON, como mostrado abaixo:

json

 Copiar código

```
{ "nome": "Carlos", "login": "carlos" }
```

4. Envie a requisição. Você deve receber uma mensagem de sucesso.
5. Verifique se o usuário foi adicionado fazendo uma requisição GET para <http://localhost:3000/usuarios>.

## Conclusão

Nesta aula, configuramos um servidor Express, definimos rotas e controladores, e modelamos uma classe para gerenciar usuários.

Utilizamos a arquitetura MVC para organizar nosso código de forma eficiente e facilitar a manutenção.

Pratiquem esses conceitos e explorem novas funcionalidades para aprimorar suas habilidades em desenvolvimento back-end.

Até a próxima aula!

## 1.6. Vídeo Aula

dia 6 express 3 converted

