

Material de apoio

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

Impresso por: JOÃO VITOR DE MELO FREITAS
Data: quinta-feira, 18 jul. 2024, 23:09

Índice

1. Back-end com Node.js

- 1.1. Introdução à Orientação a Objetos
- 1.2. Vídeo Aula
- 1.3. Classe em Node.js Continuação
- 1.4. Vídeo Aula
- 1.5. Herança em JavaScript
- 1.6. Vídeo Aula
- 1.7. Métodos Estáticos em Classes JavaScript
- 1.8. Vídeo Aula

1. Back-end com Node.js

Revisão e Continuação das Rotas

Na aula anterior, criamos um arquivo `produtos.js` dentro da pasta `rotas` e aprendemos a separar funções em arquivos distintos para melhorar a organização do código. Vimos como utilizar `module.exports` para exportar essas funções e importar no `server.js`. Hoje, vamos continuar configurando essas rotas e adicionar mais funcionalidades.

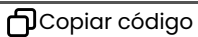
Reestruturação das Funções de Produtos

Vamos revisar e melhorar as funções de produtos, utilizando orientação a objetos para uma melhor organização.

Criando a Classe Produto

Vamos criar uma classe `Produto` para gerenciar nossos produtos.

javascript

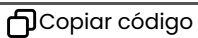


```
// rotas/produtos.js class Produto { constructor(id, nome, valor) { this.id = id; this.nome = nome; this.valor = valor; } static listarProdutos() { return JSON.stringify(produtos); } static adicionarProduto(produto) { produtos.push(produto); return JSON.stringify({ mensagem: 'Produto adicionado com sucesso' }); } static removerProduto() { produtos.pop(); return JSON.stringify({ mensagem: 'Produto removido com sucesso' }); } } const produtos = [ new Produto(1, 'Teclado', 50), new Produto(2, 'Mouse', 30) ]; module.exports = Produto;
```

Utilizando a Classe Produto no Servidor

Vamos ajustar o arquivo `server.js` para utilizar a classe `Produto`.

javascript



```
// server.js const http = require('http'); const Produto = require('./rotas/produtos'); const server = http.createServer((req, res) => { const { url, method } = req; if (url === '/' && method === 'GET') { res.statusCode = 200; res.setHeader('Content-Type', 'text/plain'); res.end('Servidor Node.js'); } else if (url === '/produtos' && method === 'GET') { res.statusCode = 200; res.setHeader('Content-Type', 'application/json'); res.end(Produto.listarProdutos()); } else if (url === '/produtos/adicionar' && method === 'POST') { const novoProduto = new Produto(3, 'Monitor', 800); res.statusCode = 201; res.setHeader('Content-Type', 'application/json'); res.end(Produto.adicionarProduto(novoProduto)); } else if (url === '/produtos/remover' && method === 'DELETE') { res.statusCode = 200; res.setHeader('Content-Type', 'application/json'); res.end(Produto.removerProduto()); } else { res.statusCode = 404; res.setHeader('Content-Type', 'text/plain'); res.end('Página não encontrada'); } }); const hostname = '127.0.0.1'; const port = 3000; server.listen(port, hostname, () => { console.log(`Servidor rodando em http://${hostname}:${port}/`); });
```

Testando as Rotas de Produtos

Para testar, utilize ferramentas como Postman ou curl para enviar requisições GET, POST e DELETE para as rotas configuradas.


1.1. Introdução à Orientação a Objetos

A orientação a objetos (OO) é um paradigma de programação que nos ajuda a organizar e estruturar nossos projetos de forma mais eficiente. Em JavaScript, podemos criar classes e objetos para representar entidades e suas funcionalidades.

Criando uma Classe Pessoa

Vamos criar uma classe `Pessoa` para entender melhor como funciona a orientação a objetos em JavaScript.

javascript

 Copiar código

```
// criando uma classe Pessoa class Pessoa { constructor(nome, cpf, idade) { this.nome = nome; this.cpf = cpf; this.idade = idade; } correr() { console.log(`${this.nome} está correndo.`); } dormir() { console.log(`${this.nome} está dormindo.`); } }  
// criando instâncias da classe Pessoa const pessoa1 = new Pessoa('José', '123.456.789-00', 30); const pessoa2 = new Pessoa('Maria', '987.654.321-00', 25); const pessoa3 = new Pessoa('João', '456.123.789-00', 28); // utilizando os métodos da classe Pessoa pessoa1.correr(); // José está correndo. pessoa2.dormir(); // Maria está dormindo. pessoa3.correr(); // João está correndo.
```


Explicação do Código

1. **Classe Pessoa:** Definimos a classe `Pessoa` com um construtor que recebe `nome`, `cpf` e `idade` como parâmetros.
2. **Métodos:** Adicionamos métodos `correr` e `dormir` para simular ações que uma pessoa pode realizar.
3. **Instâncias:** Criamos três instâncias da classe `Pessoa` (`pessoa1`, `pessoa2`, `pessoa3`) com diferentes valores.
4. **Utilização dos Métodos:** Chamamos os métodos `correr` e `dormir` nas instâncias criadas.

Executando o Código

Para executar o código acima, salve-o em um arquivo `classes.js` e execute no terminal:

bash

 Copiar código

```
node classes.js
```

Você verá a saída no console indicando as ações realizadas pelas instâncias da classe `Pessoa`.

Conclusão

Hoje aprendemos a aplicar conceitos de orientação a objetos em nosso projeto Node.js, criando e utilizando classes para organizar melhor nosso código.

Continuaremos a explorar mais nas próximas aulas e a aplicar esses conceitos em nosso projeto de back-end.

Espero que tenham gostado e até a próxima aula!

1.2. Vídeo Aula

dia 04 nodejs 1 converted



1.3. Classe em Node.js Continuação

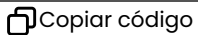
Vamos continuar com a nossa classe. Na aula passada, criamos a classe `Pessoa`, que é um molde de um objeto usando orientação a objetos em JavaScript.

Definimos os atributos, os dados dessa classe, e o construtor para inicializar esses atributos.

Além disso, criamos alguns métodos de classe, como `correr`, que exibe o nome seguido da ação de correr, e `dormir`, que também vamos adicionar agora.

Vamos revisar rapidamente e adicionar o método `dormir`:

javascript



```
class Pessoa { constructor(nome, cpf, idade) { this.nome = nome; this.cpf = cpf; this.idade = idade; } correr() { console.log(`${this.nome} está correndo.`); } dormir() { console.log(`${this.nome} está dormindo.`); } } const pessoa1 = new Pessoa('José', '123.456.789-00', 30); const pessoa2 = new Pessoa('Maria', '987.654.321-00', 25); const pessoa3 = new Pessoa('João', '456.123.789-00', 28);
```

Explicação Teórica

Vamos entender melhor como funciona a criação e manipulação de instâncias de classes em JavaScript.

Endereços de Memória e Instâncias

Quando criamos uma nova instância de `Pessoa`, como `const pessoa1 = new Pessoa('José', '123.456.789-00', 30);`, estamos alocando um espaço na memória para armazenar esses dados. A variável `pessoa1` aponta para esse endereço de memória.

Na memória, criamos um espaço para cada instância, onde os dados dessa instância são armazenados.

Manipulação de Instâncias

Vamos ver como a manipulação de instâncias funciona na prática.

javascript



```
const pessoaAuxiliar = pessoa1; pessoaAuxiliar.nome = 'Alice'; console.log(pessoa1.nome); // Alice console.log(pessoaAuxiliar.nome); // Alice
```

Explicação

Quando atribuímos `pessoa1` a `pessoaAuxiliar`, não estamos criando uma nova instância. Estamos apenas apontando `pessoaAuxiliar` para o mesmo endereço de memória de `pessoa1`. Portanto, qualquer alteração feita através de `pessoaAuxiliar` afetará `pessoa1`.

Implementação Prática

Vamos aplicar esse conhecimento em nosso projeto de produtos.

Criando a Classe Produto

Vamos criar uma classe `Produto` para gerenciar nossos produtos de forma mais estruturada.

javascript



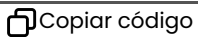
```
// rotas/produtos.js class Produto { constructor(id, nome, valor) { this.id = id; this.nome = nome; this.valor = valor; } static listarProdutos() { return JSON.stringify(produtos); } static adicionarProduto(produto) { produtos.push(produto); return JSON.stringify({ mensagem: 'Produto adicionado com sucesso' }); } static removerProduto() { produtos.pop(); return
```

```
JSON.stringify({ mensagem: 'Produto removido com sucesso' })); } } const produtos = [ new Produto(1, 'Teclado', 50), new  
Produto(2, 'Mouse', 30) ]; module.exports = Produto;
```

Utilizando a Classe Produto no Servidor

Vamos ajustar o arquivo `server.js` para utilizar a classe `Produto`.

javascript



```
// server.js const http = require('http'); const Produto = require('./rotas/produtos'); const server =  
http.createServer((req, res) => { const { url, method } = req; if (url === '/' && method === 'GET') { res.statusCode = 200;  
res.setHeader('Content-Type', 'text/plain'); res.end('Servidor Node.js'); } else if (url === '/produtos' && method ===  
'GET') { res.statusCode = 200; res.setHeader('Content-Type', 'application/json'); res.end(Produto.listarProdutos()); } else  
if (url === '/produtos/adicionar' && method === 'POST') { const novoProduto = new Produto(3, 'Monitor', 800); res.statusCode  
= 201; res.setHeader('Content-Type', 'application/json'); res.end(Produto.adicionarProduto(novoProduto)); } else if (url ===  
'/produtos/remover' && method === 'DELETE') { res.statusCode = 200; res.setHeader('Content-Type', 'application/json');  
res.end(Produto.removerProduto()); } else { res.statusCode = 404; res.setHeader('Content-Type', 'text/plain');  
res.end('Página não encontrada'); } }); const hostname = '127.0.0.1'; const port = 3000; server.listen(port, hostname, () =>  
{ console.log(`Servidor rodando em http://${hostname}:${port}/`); });
```

Testando as Rotas de Produtos

Para testar, utilize ferramentas como Postman ou curl para enviar requisições GET, POST e DELETE para as rotas configuradas.

Conclusão

Hoje, aprofundamos nosso conhecimento em orientação a objetos e aplicamos esses conceitos em nosso projeto Node.js, criando e utilizando classes para organizar melhor nosso código.

Continuaremos a explorar mais sobre OO nas próximas aulas e a aplicar esses conceitos em nosso projeto de back-end.

Espero que tenham gostado e até a próxima aula!

1.4. Vídeo Aula

dia 04 nodejs 2 converted



1.5. Herança em JavaScript

Olá pessoal, tudo bem? Na aula passada, vimos como criar classes e objetos em JavaScript. Hoje, vamos aprender sobre herança.

Vamos entender como uma classe pode herdar atributos e métodos de outra classe.

Conceito de Herança

A herança é um princípio fundamental da orientação a objetos, onde uma classe pode herdar atributos e métodos de outra classe.

Isso nos permite reutilizar código e criar hierarquias de classes. Por exemplo, uma classe `Pessoa` pode ser herdada por uma classe `Funcionario`, pois um funcionário é uma pessoa.


Vamos entender isso na prática.

Implementando Herança

Criando a Classe `Pessoa`

Primeiro, vamos criar a classe `Pessoa` com alguns atributos e métodos:

javascript

 Copiar código

```
class Pessoa { constructor(nome, cpf, dataDeNascimento) { this.nome = nome; this.cpf = cpf; this.dataDeNascimento = dataDeNascimento; } autenticar() { console.log(`${this.nome} está autenticado.`); } } const pessoa1 = new Pessoa('João', '123.456.789-00', '01/01/1990'); console.log(pessoa1); pessoa1.autenticar();
```

Criando a Classe `Gerente` Herdeira de `Pessoa`

Agora, vamos criar uma classe `Gerente` que herda de `Pessoa`:

javascript

 Copiar código

```
class Gerente extends Pessoa { constructor(nome, cpf, dataDeNascimento, departamento) { super(nome, cpf, dataDeNascimento); this.departamento = departamento; } autenticar() { super.autenticar(); console.log(`Gerente do departamento ${this.departamento} autenticado.`); } } const gerente1 = new Gerente('José', '987.654.321-00', '02/02/1980', 'Vendas'); console.log(gerente1); gerente1.autenticar();
```


Explicação

- Classe `Pessoa`:** Criamos uma classe `Pessoa` com os atributos `nome`, `cpf` e `dataDeNascimento`, além de um método `autenticar`.
- Classe `Gerente`:** Criamos a classe `Gerente` que herda de `Pessoa` usando a palavra-chave `extends`. No construtor de `Gerente`, usamos `super()` para chamar o construtor da classe `Pessoa` e inicializar os atributos herdados. Adicionamos um novo atributo `departamento`.
- Sobrescrita de Método:** O método `autenticar` é sobrescrito na classe `Gerente` para adicionar uma funcionalidade específica do gerente, mas ainda chama o método `autenticar` da classe `Pessoa` usando `super.autenticar()`.

Testando a Herança

Vamos testar nosso código para ver como a herança funciona na prática:

javascript

 Copiar código

```
const gerente1 = new Gerente('José', '987.654.321-00', '02/02/1980', 'Vendas'); console.log(gerente1); gerente1.autenticar(); // Output: José está autenticado. Gerente do departamento Vendas autenticado.
```

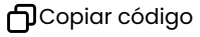
Explicação Detalhada da Memória

Quando criamos uma nova instância de `Gerente`, como `const gerente1 = new Gerente('José', '987.654.321-00', '02/02/1980', 'Vendas');`, alocamos um espaço na memória para armazenar esses dados.

A variável `gerente1` aponta para esse endereço de memória.

Manipulação de Referências

javascript



Copiar código

```
const gerenteAuxiliar = gerente1; gerenteAuxiliar.nome = 'Alice'; console.log(gerente1.nome); // Alice
console.log(gerenteAuxiliar.nome); // Alice
```

Quando atribuímos `gerente1` a `gerenteAuxiliar`, ambos apontam para o mesmo endereço de memória. Qualquer alteração feita através de `gerenteAuxiliar` afetará `gerente1`.

Conclusão

Hoje, aprendemos sobre herança em JavaScript. Vimos como uma classe pode herdar atributos e métodos de outra classe, facilitando a reutilização de código e a criação de hierarquias.

Na próxima aula, continuaremos a explorar mais conceitos de orientação a objetos e como aplicá-los em nosso projeto de back-end.

Espero que tenham gostado e até a próxima aula!

1.6. Vídeo Aula

dia 04 nodejs 3 converted



1.7. Métodos Estáticos em Classes JavaScript

Olá pessoal, tudo bem?

Hoje vamos continuar com nossa aula sobre orientação a objetos. Na aula passada, vimos como criar classes e objetos.

Agora, vamos explorar os métodos estáticos em classes JavaScript e entender como utilizá-los de maneira eficiente.

Revisão: Instanciando Classes

Para utilizar os atributos e métodos de uma classe, normalmente precisamos instanciar essa classe.

Ao instanciar, criamos um objeto que contém os dados e métodos definidos na classe.

javascript

 Copiar código

```
class Pessoa { constructor(nome, cpf, dataDeNascimento) { this.nome = nome; this.cpf = cpf; this.dataDeNascimento = dataDeNascimento; } autenticar() { console.log(`${this.nome} está autenticado.`); } } const pessoa1 = new Pessoa('João', '123.456.789-00', '01/01/1990'); console.log(pessoa1.nome); // João pessoa1.autenticar(); // João está autenticado.
```

Acesso Direto à Classe

Se tentarmos acessar diretamente um atributo ou método da classe sem instanciá-la, obteremos um erro ou um valor `undefined`.

javascript

 Copiar código

```
console.log(Pessoa.nome); // undefined Pessoa.autenticar(); // Erro
```

Métodos Estáticos


Métodos estáticos são aqueles que pertencem à classe em si, e não às instâncias da classe.

Isso significa que podemos chamar esses métodos diretamente na classe, sem precisar criar uma instância.

Definindo Métodos Estáticos

Vamos criar uma classe `Carro` com métodos estáticos:

javascript

 Copiar código

```
class Carro { static acelerar() { console.log('O carro está acelerando.')} static frear() { console.log('O carro está freando.')} static ligarSeta(direcao) { console.log(`Ligando seta para ${direcao}.`); } } Carro.acelerar(); // O carro está acelerando. Carro.frear(); // O carro está freando. Carro.ligarSeta('direita'); // Ligando seta para direita.
```

Utilidade dos Métodos Estáticos

Os métodos estáticos são úteis quando precisamos de funcionalidades que não dependem do estado das instâncias.

Por exemplo, podemos utilizá-los para operações de banco de dados, validações e outras funções utilitárias.

Exemplo com Banco de Dados

Vamos criar uma classe `BancoDeDados` com métodos estáticos para operações CRUD:

javascript

 Copiar código

```
class BancoDeDados { static criar(dado) { console.log(`Criando o dado: ${dado}`); } static atualizar(dado) { console.log(`Atualizando o dado: ${dado}`); } static deletar(dado) { console.log(`Deletando o dado: ${dado}`); } static
```

```
consultar(dado) { console.log(`Consultando o dado: ${dado}`); } } BancoDeDados.criar('Registro 1'); // Criando o dado:
Registro 1 BancoDeDados.atualizar('Registro 1'); // Atualizando o dado: Registro 1 BancoDeDados.deletar('Registro 1'); //
Deletando o dado: Registro 1 BancoDeDados.consultar('Registro 1'); // Consultando o dado: Registro 1
```

Conclusão

Os métodos estáticos em JavaScript são uma maneira poderosa de organizar e reutilizar código que não depende do estado das instâncias de uma classe.

Eles são particularmente úteis para operações utilitárias e funções de apoio.

Pratiquem criando suas próprias classes e métodos estáticos. Até a próxima aula!

1.8. Vídeo Aula

dia 04 nodejs 4 converted

