O EventEmitter é uma classe do módulo events no Node.js que facilita a implementação de um padrão de eventos e ouvintes (listeners). Ele é usado para criar, emitir e manipular eventos assíncronos ou síncronos, permitindo que diferentes partes de uma aplicação se comuniquem de forma eficiente.

**Como funciona?**

* Um objeto de EventEmitter pode emitir eventos com o método .emit(eventName, [...args]).
* Listeners podem ser registrados para esses eventos usando .on(eventName, listener) ou .once(eventName, listener) (para escutar apenas uma vez).
* Você pode remover listeners usando .removeListener(eventName, listener) ou .off(eventName, listener).

Essa funcionalidade é útil para lidar com tarefas baseadas em eventos, como interações de usuário, mensagens, fluxos de dados, ou manipulação de dados assíncronos.

**Exemplo Prático**

Vamos criar um sistema de monitoramento de um pedido de comida. Quando o pedido é criado, um evento será disparado, e ouvintes diferentes reagirão de acordo:

**Código:**

javascript

Copiar código

const EventEmitter = require('events');

// Criando a classe Pedido que estende EventEmitter

class Pedido extends EventEmitter {

criarPedido(pedido) {

console.log('Pedido criado:', pedido);

this.emit('novoPedido', pedido);

}

}

// Instância do Pedido

const pedido = new Pedido();

// Adicionando ouvintes

pedido.on('novoPedido', (pedido) => {

console.log(`🚚 Enviando pedido para entrega: ${pedido.item}`);

});

pedido.on('novoPedido', (pedido) => {

console.log(`📦 Atualizando o estoque para o item: ${pedido.item}`);

});

// Emitindo um evento

pedido.criarPedido({ id: 1, item: 'Pizza Grande', quantidade: 1 });

**Saída:**

plaintext

Copiar código

Pedido criado: { id: 1, item: 'Pizza Grande', quantidade: 1 }

🚚 Enviando pedido para entrega: Pizza Grande

📦 Atualizando o estoque para o item: Pizza Grande

**Explicação:**

1. **Criação de Eventos**: A classe Pedido estende EventEmitter e pode emitir eventos personalizados.
2. **Listeners**: Foram adicionados dois ouvintes para o evento 'novoPedido' que realizam ações específicas quando o evento é emitido.
3. **Emissão de Evento**: Quando o método criarPedido é chamado, o evento 'novoPedido' é emitido, e todos os ouvintes são notificados.

Esse padrão torna o código modular e fácil de manter, permitindo que novas funcionalidades sejam adicionadas apenas registrando novos ouvintes.

@@@@@@@@@@@@@@@@@@@@@@@

* **Agora adicionando mais eventos!!!!!**

const EventEmitter = require('events');

// Criando a classe Pedido que estende EventEmitter

class Pedido extends EventEmitter {

criarPedido(pedido) {

console.log('✅ Pedido criado:', pedido);

this.emit('novoPedido', pedido);

}

confirmarPagamento(pedido) {

console.log('💳 Pagamento confirmado para o pedido:', pedido.id);

this.emit('pagamentoConfirmado', pedido);

}

prepararPedido(pedido) {

console.log('🍳 Pedido em preparo:', pedido.item);

setTimeout(() => {

console.log('🍽️ Pedido pronto:', pedido.item);

this.emit('pedidoPreparado', pedido);

}, 2000); // Simula tempo de preparo

}

notificarCliente(pedido) {

console.log('📲 Notificando cliente sobre o pedido:', pedido.id);

this.emit('clienteNotificado', pedido);

}

}

// Instância do Pedido

const pedido = new Pedido();

// Adicionando ouvintes

pedido.on('novoPedido', (pedido) => {

console.log(`🚚 Enviando pedido para o sistema de entrega: ${pedido.item}`);

});

pedido.on('novoPedido', (pedido) => {

console.log(`📦 Atualizando estoque para o item: ${pedido.item}`);

pedido.emit('estoqueAtualizado', pedido);

});

pedido.on('pagamentoConfirmado', (pedido) => {

pedido.prepararPedido(pedido);

});

pedido.on('pedidoPreparado', (pedido) => {

pedido.notificarCliente(pedido);

});

pedido.on('clienteNotificado', (pedido) => {

console.log(`🎉 Cliente notificado. Pedido ${pedido.id} concluído com sucesso!`);

});

// Simulação do fluxo de pedidos

const novoPedido = { id: 1, item: 'Pizza Grande', quantidade: 1 };

// Fluxo do sistema

pedido.criarPedido(novoPedido);

setTimeout(() => pedido.confirmarPagamento(novoPedido), 1000); // Simula tempo para confirmação de pagamento

**Saída Esperada:**

plaintext

Copiar código

✅ Pedido criado: { id: 1, item: 'Pizza Grande', quantidade: 1 }

🚚 Enviando pedido para o sistema de entrega: Pizza Grande

📦 Atualizando estoque para o item: Pizza Grande

💳 Pagamento confirmado para o pedido: 1

🍳 Pedido em preparo: Pizza Grande

🍽️ Pedido pronto: Pizza Grande

📲 Notificando cliente sobre o pedido: 1

🎉 Cliente notificado. Pedido 1 concluído com sucesso!

**Explicação:**

1. **Eventos Adicionais**:
   * pagamentoConfirmado: Disparado quando o pagamento do pedido é confirmado.
   * pedidoPreparado: Emitido quando o preparo do pedido é concluído.
   * clienteNotificado: Finaliza o fluxo de pedido, notificando o cliente.
2. **Encadeamento de Eventos**:
   * Cada evento dispara o próximo passo do fluxo, criando um pipeline lógico para o processamento do pedido.
3. **Simulação de Tempo**:
   * O setTimeout é usado para simular atrasos reais, como tempo de pagamento ou preparo.
4. **Reusabilidade**:
   * A modularidade permite que outros componentes sejam integrados facilmente, como mais ouvintes ou lógica de negócios.

Esse fluxo representa um exemplo robusto de como o EventEmitter pode ser utilizado para criar sistemas reativos e organizados!

@@@@@@@@@@@@@@@

Os quatro tipos fundamentais de streams no Node.js são:

1. **Writable Stream**:
   * São streams para os quais você pode **escrever dados**. Elas possuem métodos como .write() para enviar dados e .end() para sinalizar o fim da gravação.
   * Exemplo: fs.createWriteStream() (para gravar em arquivos).
2. **Readable Stream**:
   * São streams das quais você pode **ler dados**. Elas possuem métodos como .read() para ler dados e eventos como data, end, e error para lidar com os dados e a conclusão.
   * Exemplo: fs.createReadStream() (para ler arquivos).
3. **Duplex Stream**:
   * São streams que permitem **ler e escrever dados**. Ou seja, elas combinam as características dos streams legíveis e graváveis.
   * Exemplo: sockets de rede ou streams de comunicação bidirecional.
4. **Transform Stream**:
   * São uma forma especial de **Duplex Stream**, mas com a capacidade de **modificar** ou **transformar** os dados enquanto eles são lidos e gravados. Isso ocorre com base em uma transformação definida pelo usuário, como compressão ou criptografia.
   * Exemplo: zlib.createGzip() (para comprimir dados).

Esses tipos de streams ajudam a gerenciar operações assíncronas de leitura e escrita de dados de forma eficiente, permitindo o processamento de grandes volumes de dados sem bloquear o programa.