estágio  
Desenvolvimento Full Stack

Aluna de Engenharia de Software

linha horizontal

# 16 e 17 de setembro de 2024

## 1. Enunciado do Exercício

Criar um backend com um cron job que realiza as seguintes operações:

1. **Buscar Dados:** Buscar todos os registros de uma tabela (tabela1) em um banco de dados (db1).
2. **Copiar Dados:** Copiar todos esses registros para outra tabela (tabela2) em um banco de dados diferente (db2).
3. **Tecnologias:** Utilizar JavaScript com Express e SQLite.
4. **Nome do Projeto:** cron-job-backend.

Um cron job é basicamente uma tarefa agendada que é executada automaticamente em intervalos de tempo específicos, sem precisar de intervenção humana. Ele é muito usado em servidores para realizar tarefas repetitivas, como backups de dados ou envio de relatórios. No seu caso, o cron job será responsável por buscar dados de uma tabela e copiá-los para outra tabela, de maneira automática.

## Fase 1: Configuração do projeto

**Criação da pasta do projeto e inicialização do Node.js**: Aqui, você cria um novo projeto Node.js, o que serve como uma base para o desenvolvimento de aplicações em JavaScript.

* mkdir cron-job-backend: Cria uma nova pasta para o seu projeto.
* npm init -y: Inicializa o projeto e cria um arquivo package.json, que guarda informações do seu projeto e suas dependências.

**Instalação das dependências**: As bibliotecas que você vai usar:

npm install express sqlite3 node-cron

* express: Um framework para criar servidores web com facilidade.
* sqlite3: Para trabalhar com bancos de dados SQLite.
* node-cron: Biblioteca que permite criar cron jobs em Node.js, ou seja, tarefas que podem ser agendadas para rodar automaticamente.

### Passo 2: Crie a estrutura de arquivos

Na raiz do projeto, crie:

* index.js: para configurar o servidor e cron job.

### Passo 3. Código Atualizado para index.js

### código dentro do arquivo index.js. Este código realiza a tarefa de copiar dados de tabela1 para tabela2, limpando a tabela de destino antes de inserir os novos dados.

const express = require('express');

const sqlite3 = require('sqlite3').verbose();

const cron = require('node-cron');

const app = express();

// Configurar o banco de dados SQLite

const db1 = new sqlite3.Database('./db1.sqlite');

const db2 = new sqlite3.Database('./db2.sqlite');

// Criar tabelas exemplo no banco de dados 1 e 2

db1.run('CREATE TABLE IF NOT EXISTS tabela1 (id INTEGER PRIMARY KEY, nome TEXT)');

db2.run('CREATE TABLE IF NOT EXISTS tabela2 (id INTEGER PRIMARY KEY, nome TEXT)');

// Função para copiar dados de tabela1 (db1) para tabela2 (db2)

const copiarDados = () => {

// Limpar tabela2

db2.run('DELETE FROM tabela2', (err) => {

if (err) {

console.error('Erro ao limpar tabela2:', err);

return;

}

// Copiar dados de tabela1 para tabela2

db1.all('SELECT \* FROM tabela1', (err, rows) => {

if (err) {

console.error('Erro ao buscar dados:', err);

return;

}

rows.forEach((row) => {

db2.run('INSERT INTO tabela2 (id, nome) VALUES (?, ?)', [row.id, row.nome], (err) => {

if (err) {

console.error('Erro ao inserir dados:', err);

} else {

console.log(`Dados copiados: ${row.id} - ${row.nome}`);

}

});

});

});

});

};

// Agendar cron job para rodar a cada minuto

cron.schedule('\* \* \* \* \*', () => {

console.log('Executando cron job...');

copiarDados();

});

// Servidor básico

app.get('/', (req, res) => {

res.send('Cron job está rodando!');

});

app.listen(3000, () => {

console.log('Servidor rodando na porta 3000');

});

### Explicação Detalhada

#### **1. Importação das Bibliotecas**

const express = require('express');

const sqlite3 = require('sqlite3').verbose();

const cron = require('node-cron');

**express**: Biblioteca para criar o servidor web.

**sqlite3**: Biblioteca para interagir com bancos de dados SQLite. verbose() é usado para obter logs mais detalhados. **.verbose()**: Ativa o modo detalhado de depuração para o módulo sqlite3.

**node-cron**: Biblioteca para agendar tarefas que serão executadas periodicamente.

#### **2. Inicialização do Servidor Express**

const app = express();

**app**: Cria uma instância do aplicativo Express, que será usada para definir rotas e iniciar o servidor.

#### **3. Configuração dos Bancos de Dados**

const db1 = new sqlite3.Database('./db1.sqlite');

const db2 = new sqlite3.Database('./db2.sqlite');

**db1 e db2**: Instâncias dos bancos de dados SQLite. db1 é usado para tabela1 e db2 para tabela2.

#### **4. Criação das Tabelas**

db1.run('CREATE TABLE IF NOT EXISTS tabela1 (id INTEGER PRIMARY KEY, nome TEXT)');

db2.run('CREATE TABLE IF NOT EXISTS tabela2 (id INTEGER PRIMARY KEY, nome TEXT)');

**db1.run e db2.run**: Executa comandos SQL para criar tabelas se elas não existirem. tabela1 e tabela2 têm colunas id e nome.

#### **5. Função copiarDados**

const copiarDados = () => {

// Limpar tabela2

db2.run('DELETE FROM tabela2', (err) => {

if (err) {

console.error('Erro ao limpar tabela2:', err);

return;

}

// Copiar dados de tabela1 para tabela2

db1.all('SELECT \* FROM tabela1', (err, rows) => {

if (err) {

console.error('Erro ao buscar dados:', err);

return;

}

rows.forEach((row) => {

db2.run('INSERT INTO tabela2 (id, nome) VALUES (?, ?)', [row.id, row.nome], (err) => {

if (err) {

console.error('Erro ao inserir dados:', err);

} else {

console.log(`Dados copiados: ${row.id} - ${row.nome}`);

}

});

});

});

});

};

**copiarDados**: Função responsável por copiar dados de tabela1 para tabela2.

* **Limpar tabela2:** Remove todos os registros da tabela2 antes de inserir novos dados.
* **Buscar dados de tabela1:** Seleciona todos os registros de tabela1.
* **Inserir dados em tabela2:** Insere cada registro da tabela1 na tabela2. Se ocorrer um erro, ele é exibido no console; caso contrário, o sucesso é registrado.

### Explicação Linha a Linha

1. **Definição da Função**

const copiarDados = () => {

Define uma função chamada copiarDados usando uma função de seta (arrow function). Esta função será responsável por realizar a cópia dos dados.

1. Limpar tabela

db2.run('DELETE FROM tabela2', (err) => {

**db2.run('DELETE FROM tabela2', ...): Executa um comando SQL para remover todos os registros da tabela tabela2 no banco de dados db2.**

**(err) => { ... }: Função de callback que lida com possíveis erros ao executar o comando. O parâmetro err será null se não houver erros.**

1. **Tratamento de Erros ao Limpar** tabela2

if (err) {

console.error('Erro ao limpar tabela2:', err);

return;

}

**if (err) { ... }**: Verifica se houve um erro ao tentar limpar tabela2.

**console.error('Erro ao limpar tabela2:', err);**: Exibe uma mensagem de erro no console, indicando que houve um problema ao limpar a tabela.

**return;**: Se houver um erro, a função copiarDados é interrompida e não prossegue para as próximas etapas.

1. **Buscar Dados de** tabela1

db1.all('SELECT \* FROM tabela1', (err, rows) => {

**db1.all('SELECT \* FROM tabela1', ...)**: Executa um comando SQL para buscar todos os registros da tabela tabela1 no banco de dados db1.

**(err, rows) => { ... }**: Função de callback que lida com possíveis erros e retorna os resultados da consulta. O parâmetro rows contém os registros retornados.

1. **Tratamento de Erros ao Buscar Dados**

if (err) {

console.error('Erro ao buscar dados:', err);

return;

}

**if (err) { ... }**: Verifica se houve um erro ao buscar dados de tabela1.

**console.error('Erro ao buscar dados:', err);**: Exibe uma mensagem de erro no console, indicando que houve um problema ao buscar os dados.

**return;**: Se houver um erro, a função copiarDados é interrompida e não prossegue para a inserção dos dados.

1. **Inserir Dados em tabela2**

rows.forEach((row) => {

db2.run('INSERT INTO tabela2 (id, nome) VALUES (?, ?)', [row.id, row.nome], (err) => {

**rows.forEach((row) => { ... });**: Itera sobre cada registro (linha) retornado da consulta SELECT em tabela1.

**rows**: É um array de objetos que contém todos os registros que foram buscados da tabela1.

**forEach((row) => { ... })**: É um método que percorre cada item no array rows. Para cada item (registro), o código dentro do forEach será executado.

**row**: Representa um registro individual da tabela1 que foi retornado pela consulta SQL.

**db2.run('INSERT INTO tabela2 (id, nome) VALUES (?, ?)', [row.id, row.nome], (err) => { ... });**: Executa um comando SQL para inserir cada registro na tabela2. Os valores id e nome são passados como parâmetros para evitar problemas de injeção de SQL.

**db2.run**: É um método que executa um comando SQL no banco de dados db2.

**'INSERT INTO tabela2 (id, nome) VALUES (?, ?)'**: É o comando SQL que diz ao banco de dados para inserir um novo registro na tabela2. O ? é um marcador de posição que será substituído pelos valores reais.

**[row.id, row.nome]**: É um array com os valores que serão inseridos na tabela2. Esses valores vêm do objeto row, que é um registro da tabela1. row.id e row.nome correspondem às colunas id e nome da tabela2.

**(err) => { ... }**: É uma função de callback que é chamada depois que o comando SQL é executado. Ela lida com possíveis erros e realiza ações após a execução do comando.

1. **Tratamento de Erros ao Inserir Dados**

if (err) {

console.error('Erro ao inserir dados:', err);

} else {

console.log(`Dados copiados: ${row.id} - ${row.nome}`);

}

**if (err) { ... }**: Verifica se houve um erro ao tentar inserir um registro em tabela2.

**console.error('Erro ao inserir dados:', err);**: Exibe uma mensagem de erro no console se houver um problema ao inserir os dados.

**console.log(Dados copiados: ${row.id} - ${row.nome});**: Exibe uma mensagem de sucesso no console indicando que o registro foi copiado com sucesso.

1. **Fechamento da Função**

});

Fecha a função de callback db2.run e a função rows.forEach.

#### **6. Agendamento do Cron Job**

cron.schedule('\* \* \* \* \*', () => {

console.log('Executando cron job...');

copiarDados();

});

**cron.schedule('\* \* \* \* \*', ...)**: Configura um cron job que executa a função copiarDados a cada minuto. O formato '\* \* \* \* \*' é uma expressão cron que especifica a frequência da tarefa.

#### **7. Configuração do Servidor Express**

app.get('/', (req, res) => {

res.send('Cron job está rodando!');

});

app.listen(3000, () => {

console.log('Servidor rodando na porta 3000');

});

**app.get('/', ...)**: Define uma rota para o endpoint /. Quando acessado, o servidor responde com a mensagem "Cron job está rodando!".

**app.listen(3000, ...)**: Inicia o servidor na porta 3000 e exibe uma mensagem no console confirmando que o servidor está rodando.

### Resumo do código index.js

* **Importação:** Bibliotecas necessárias são carregadas.
* **Inicialização:** Servidor e bancos de dados são configurados.
* **Criação de Tabelas:** Tabelas são criadas se não existirem.
* **Função copiarDados:** Limpa tabela2, copia dados de tabela1 para tabela2.
* **Cron Job:** Executa a função copiarDados a cada minuto.
* **Servidor Express:** Configura um servidor básico que confirma seu funcionamento.

## 4. Explicação do Banco de Dados

### Configuração do Banco de Dados

* **Banco de Dados 1 (db1):** Utilizado para armazenar a tabela tabela1, que contém os dados originais que serão copiados.
* **Banco de Dados 2 (db2):** Utilizado para armazenar a tabela tabela2, que receberá os dados copiados de tabela1.

### Estrutura das Tabelas

* **Tabela tabela1 em db1:**
  + **Colunas:**
    - id: Identificador único do registro (inteiro).
    - nome: Nome associado ao identificador (texto).
* **Tabela tabela2 em db2:**
  + **Colunas:**
    - id: Identificador único do registro (inteiro).
    - nome: Nome associado ao identificador (texto).

### Operações Realizadas pelo Código

1. **Criação das Tabelas:**
   * As tabelas tabela1 e tabela2 são criadas se ainda não existirem nos respectivos bancos de dados.
2. **Limpeza da Tabela de Destino:**
   * Antes de copiar os novos dados, a tabela tabela2 é limpa com a instrução DELETE FROM tabela2. Isso garante que os dados antigos sejam removidos.
3. **Cópia dos Dados:**
   * Todos os registros da tabela tabela1 são lidos e inseridos na tabela tabela2.
4. **Agendamento do Cron Job:**
   * O cron job é configurado para executar a função copiarDados a cada minuto, garantindo que os dados em tabela2 sejam atualizados regularmente

## 5. Teste e Verificação

1. **Inicie o Servidor:**

node index.js

**Verifique no Navegador:**

* Acesse http://localhost:3000/ para confirmar que o servidor está rodando.

**Verifique no DB Browser para SQLite:**

* Abra db2.sqlite e verifique se os dados da tabela1 foram copiados para tabela2.

**17 de setembro de 2024**

#### **Enunciado Novo**

O novo comando foi para modificar o cron job existente, de forma que ele não exclua mais os dados da tabela de destino antes de copiar novos dados. Em vez disso, deve realizar um UPDATE ou INSERT de maneira dinâmica, sem precisar criar uma instrução SQL específica para cada tabela.

### Passos e Alterações Feitas

1. **Problema Inicial**
   1. O cron job anterior estava deletando a tabela de destino (tabela2) antes de inserir novos dados.
   2. Além disso, a função que copiava dados entre as tabelas não estava preparada para lidar com UPDATE e INSERT de forma dinâmica.
2. **Objetivo das Alterações**
   1. Alterar o cron job para usar INSERT com ON CONFLICT para fazer UPDATE de forma dinâmica e eficiente.

**Código Atualizado**:

**Modificação da Função copiarDados**

* **Problema**: O código não estava configurado para realizar INSERT ou UPDATE dinâmico.
* **Correção**: Utilizar INSERT INTO ... ON CONFLICT(id) DO UPDATE SET ... para inserir novos dados ou atualizar registros existentes com base no id.

### Detalhamento do Código Atualizado

1. **Criação e Configuração das Tabelas**:

db1.run('CREATE TABLE IF NOT EXISTS tabela1 (id INTEGER PRIMARY KEY, nome TEXT)');

db2.run('CREATE TABLE IF NOT EXISTS tabela2 (id INTEGER PRIMARY KEY, nome TEXT)');

Cria as tabelas tabela1 e tabela2 se não existirem. Estas tabelas têm colunas id e nome.

**Função copiarDados**:

* **Obtenção das Colunas**:

db1.all(`PRAGMA table\_info(${nomeTabela1})`, (err, columns) => {

// Processa a informação das colunas

});

* **Preparação das Strings SQL**:

const colunas = columns.map(col => col.name).join(', '); // Ex: "id, nome"

const placeholders = columns.map(() => '?').join(', '); // Ex: "?, ?"

const updatePlaceholders = columns.map(col => `${col.name} = excluded.${col.name}`).join(', '); // Ex: "id = excluded.id, nome = excluded.nome"

**colunas**: Lista de colunas para o comando INSERT.

**placeholders**: Marcadores de posição para os valores a serem inseridos.

**updatePlaceholders**: Atualizações a serem feitas em caso de conflito de id.

**Consulta e Inserção/Atualização**:

db1.all(`SELECT \* FROM ${nomeTabela1}`, (err, rows) => {

rows.forEach((row) => {

const values = columns.map(col => row[col.name]); // Obtemos os valores dinamicamente

db2.run(

`INSERT INTO ${nomeTabela2} (${colunas}) VALUES (${placeholders}) ON CONFLICT(id) DO UPDATE SET ${updatePlaceholders}`,

values,

(err) => {

if (err) {

console.error('Erro ao inserir ou atualizar dados:', err);

} else {

console.log(`Dados copiados: ${row.id}`);

}

}

);

});

});

**rows**: Dados da tabela nomeTabela1 são obtidos e iterados.

**values**: Valores para cada linha são extraídos e utilizados para preencher o comando SQL.

**db2.run**: Insere ou atualiza dados na tabela nomeTabela2 com base no id.

***Essas mudanças garantem que o cron job atualize ou insira dados de maneira dinâmica, sem precisar escrever SQL específico para cada tabela.***

😀😀😀😀😀😀😀😀😀😀