



UNIFACS

UNIVERSIDADE SALVADOR

LAUREATE INTERNATIONAL UNIVERSITIES*

Sistemas Distribuídos e Mobile

Relatório A3

Andrey Barroso Costa – 1272123464

Leonardo Carvalho de Oliveira – 1272121364

Pablo Henrique Carvalho dos Santos – 1272121863

Vitor Souza dos Santos – 1272121538

Yana Barreto Luiz Simina – 1272219964

SALVADOR – BA

2023

SUMÁRIO

1	Introdução.....	3
2	Objetivo.....	4
3	Justificativa da escolha da tecnologia	5
4	Iniciando o projeto	6
4.1	Bibliotecas utilizadas no projeto:.....	6
5	Detalhamento sobre a arquitetura, estratégia e algoritmos utilizados.....	8
5.1	Arquitetura do projeto:.....	8
5.2	Estratégia:	9
5.3	Algoritmos:	10
6	Postman	23
7	Considerações Finais.....	24
8	Referências:	25

1 Introdução

O mercado contemporâneo exige soluções tecnológicas eficientes para otimizar processos, melhorar a tomada de decisões e proporcionar uma experiência aprimorada aos clientes. Nesse contexto, o relatório apresenta o resultado do desenvolvimento de uma aplicação dedicada à simulação da captação de dados de vendas para uma rede de lojas. Este projeto visa proporcionar uma gestão integrada e eficaz, abrangendo desde o controle de estoque até a análise estatística.

O gerenciamento de clientes e estoque é realizado por operações CRUD (Create, Read, Update, Delete). Essas operações garantem uma interface intuitiva para manipular eficientemente os dados do cliente. Para o controle de vendas, estoque e recebimento de pedidos de compra, a aplicação oferece situações que otimizam o fluxo de trabalho no varejo. Essa abordagem busca proporcionar uma solução robusta e escalável, desenvolvida na plataforma Node.js.

A geração de relatórios estatísticos é um componente crucial desta aplicação, fornecendo insights valiosos para a equipe de gestão. Os relatórios incluem informações sobre os produtos mais vendidos, análises detalhadas do consumo por cliente, bem como relatórios de produtos com baixo estoque.

O banco de dados escolhido para esta aplicação é o SQLite, proporcionando uma estrutura sólida e confiável para armazenamento de dados. Essa escolha visa garantir a integridade e consistência dos dados, elementos essenciais para uma aplicação de gestão eficaz.

Este relatório abordará em detalhes cada aspecto do desenvolvimento da aplicação, desde a escolha da tecnologia até a implementação das funcionalidades específicas. Espera-se que este projeto contribua significativamente para a eficiência operacional e aprimoramento da gestão em ambientes varejistas, proporcionando uma solução tecnológica robusta e adaptável para a captação de dados de vendas em uma rede de lojas.

2 Objetivo

Este relatório tem como objetivo detalhar o desenvolvimento da aplicação de gestão de vendas para uma rede de lojas. Abordaremos as escolhas tecnológicas, como a linguagem JavaScript com Node.js, o modelo de comunicação (API REST) e o banco de dados relacional (SQLite), justificando suas seleções. Descreveremos as operações CRUD para gerenciamento de clientes, produtos, vendas e estoque, ressaltando sua eficiência e integração. A capacidade de receber pedidos de compra e otimizar o fluxo de trabalho no varejo será explorada.

Os relatórios estatísticos, incluindo análises de produtos mais vendidos, consumo médio por cliente e identificação de produtos com baixo estoque, serão apresentados, destacando seu papel na tomada de decisões estratégicas. Com esses objetivos, buscamos fornecer uma visão abrangente do desenvolvimento da aplicação, evidenciando sua eficácia na gestão de vendas para uma rede de lojas.

3 Justificativa da escolha da tecnologia

No panorama dinâmico e desafiador do desenvolvimento de software, a escolha criteriosa da tecnologia desempenha um papel crucial no sucesso de um projeto. Diante dessa consideração, optamos por adotar uma abordagem inovadora, escolhendo o Node.js como base para o desenvolvimento da API. A decisão fundamenta-se na eficiência e escalabilidade inerentes ao ambiente assíncrono e orientado a eventos proporcionado pelo Node.js.

A notável agilidade nas operações de I/O, potencializada pela extensa variedade de bibliotecas e frameworks, destaca-se como um diferencial significativo. O uso do Express, em particular, simplifica a arquitetura da API, permitindo uma construção modular e organizada. Essa abordagem não apenas favorece a fase inicial do desenvolvimento, mas também estabelece uma base sólida para a manutenção contínua do projeto.

Além disso, a escolha do Node.js é respaldada pela sua escalabilidade horizontal, uma característica vital para projetos que vislumbram expansão e crescimento futuro. A comunidade ativa e o suporte contínuo à tecnologia proporcionam um ambiente propício para atualizações e melhorias, garantindo que o Node.js permaneça não apenas relevante, mas também evoluído.

Assim, ao optarmos pela tecnologia Node.js para a construção da API, estamos investindo não apenas em eficiência e escalabilidade, mas também em uma comunidade engajada e em constante evolução. Essa escolha reflete nosso comprometimento com um desenvolvimento ágil e bem fundamentado, buscando alcançar os objetivos do projeto de maneira sólida e inovadora.

4 Iniciando o projeto

Para inicializar o projeto, é imprescindível possuir o Node.js instalado localmente. Posteriormente, abra o terminal e navegue até a pasta “código fonte” e execute o comando "npm install" para instalar as dependências do projeto. Em seguida, empregue o comando "npm start" para iniciar o projeto. Esse comando "npm start" corresponde a um script configurado para iniciar a aplicação usando a biblioteca nodemon, a qual reinicia automaticamente a aplicação em caso de detecção de alterações no código. Alternativamente, é possível utilizar o comando "node index.js" para iniciar o projeto.

Recomenda-se a utilização do Portman para a execução de requisições no sistema. O uso dessa ferramenta facilita testes e validações, permitindo uma interação intuitiva com a API.

4.1 Bibliotecas utilizadas no projeto:

Express: é um framework web para Node.js que simplifica o desenvolvimento de aplicativos web e APIs. Ele oferece uma estrutura mínima e flexível, permitindo a fácil criação de rotas, manipulação de requisições e respostas, e integração com diversos plugins. Sua abordagem leve facilita a construção rápida e eficiente de servidores web. O Express também suporta middlewares, o que permite a inclusão de funcionalidades adicionais nas etapas de processamento das requisições. Essa biblioteca é amplamente utilizada na construção de aplicações web escaláveis e robustas em Node.js.

Body-parser: é uma biblioteca Node.js que facilita o processamento de dados contidos no corpo das requisições HTTP. Ele é comumente utilizado em conjunto com o framework Express para analisar e formatar dados vindos de formulários web ou requisições JSON. O Body-parser simplifica a extração de informações do corpo da requisição, tornando mais fácil o acesso aos dados enviados pelos clientes. Essa biblioteca é crucial em

aplicações web para manipulação eficiente de dados provenientes de diferentes fontes, contribuindo para o desenvolvimento de APIs robustas em Node.js.

SQLite3: é uma biblioteca leve e eficiente para integração do SQLite com aplicações Node.js. Ele possibilita a criação e manipulação de bancos de dados SQLite diretamente no ambiente JavaScript, facilitando o armazenamento e recuperação de dados de forma local. Sua implementação é compatível com o SQL padrão, permitindo a execução de consultas e operações em bancos de dados SQLite de maneira simplificada.

Nodemon: é uma ferramenta popular para desenvolvedores Node.js, que automatiza o reinício automático de aplicações sempre que são feitas alterações no código fonte. Ele simplifica o processo de desenvolvimento, eliminando a necessidade manual de reiniciar o servidor a cada modificação no código. O Nodemon monitora os arquivos do projeto em tempo real, reiniciando automaticamente a aplicação para refletir as atualizações instantaneamente.

5 Detalhamento sobre a arquitetura, estratégia e algoritmos utilizados.

5.1 Arquitetura do projeto:

Na arquitetura do projeto, a implementação dos padrões de repository e controller visa promover uma separação eficiente de preocupações, otimizando a organização e a manutenção do código. Os repositories, como componentes especializados, operam como interfaces entre a lógica de negócios e o banco de dados. Sua responsabilidade primária reside na execução de consultas, proporcionando uma camada de abstração que isola a complexidade do acesso aos dados.

No contexto dos controllers, estabelece-se um ponto focal para a gestão do fluxo de operações e a criação de funcionalidades que atendam às requisições do sistema. Essas entidades desempenham um papel crucial na orquestração das interações entre os diversos componentes do aplicativo, garantindo uma abordagem modular e facilitando a manutenção e expansão do sistema.

Ao adotar essa abordagem, promovemos a flexibilidade e a escalabilidade do projeto, uma vez que as funcionalidades específicas das requisições são encapsuladas nos controllers, permitindo uma melhor organização do código e facilitando a integração de novas funcionalidades de forma coesa e estruturada. Essa arquitetura oferece benefícios significativos no desenvolvimento e evolução do sistema, contribuindo para a criação de um software mais robusto e de fácil manutenção.

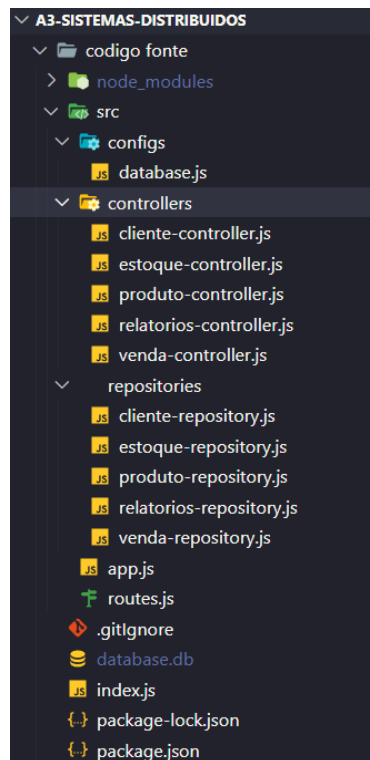


Figura 1 – arquitetura do projeto.

5.2 Estratégia:

A estratégia adotada neste projeto está fundamentada na eficiente segregação de responsabilidades e na promoção da modularidade por meio da implementação dos padrões de repository e controller. Ao designar aos repositórios a tarefa de interagir com o banco de dados, conseguimos estabelecer uma nítida separação entre a lógica de negócios e a camada de persistência, resultando em uma manutenção simplificada e favorecendo a escalabilidade do sistema.

Os controllers, por sua vez, desempenham um papel crucial na orquestração das operações, atuando como mediadores entre as solicitações do sistema e as funcionalidades específicas associadas. Essa abordagem não apenas simplifica o código, mas também facilita a incorporação de novas funcionalidades, proporcionando uma resposta ágil às demandas do projeto.

Ao adotar essa estratégia, buscamos criar uma arquitetura que permita a evolução contínua do software, oferecendo flexibilidade para ajustes e expansões futuras. A clara separação de responsabilidades entre os diversos componentes do sistema contribui para a manutenção da coesão e facilita a compreensão do código-fonte, estabelecendo uma base sólida para o desenvolvimento sustentável do projeto. Essa abordagem estratégica não apenas atende às demandas atuais, mas também prepara o terreno para futuras inovações e adaptações conforme as necessidades do ambiente de negócios evoluem.

5.3 Algoritmos:

As configurações iniciais do projeto são feitas em dois arquivos, o “index.js” e o “app.js”. O “app.js” é responsável por criar um servidor web usando o framework Express em Node.js. Primeiramente, são importados os módulos express, body-parser, e um conjunto de rotas definidas em um arquivo chamado "routes.js". Em seguida, uma instância do aplicativo Express é criada, e são configurados middlewares para tratar dados provenientes de requisições, como o middleware body-parser para analisar dados de formulários e o express.json() para processar dados no formato JSON. Por fim, as rotas definidas no arquivo mencionado são incorporadas ao aplicativo. Esse código, exportado como um módulo, serve como a base para a construção de um servidor web utilizando o framework Express, proporcionando um ambiente configurado para manipular requisições HTTP, processar dados e direcionar o fluxo do aplicativo de acordo com as rotas definidas.

Já no “index.js” é configurado um servidor HTTP utilizando o módulo http. Primeiramente, são importados os módulos http e o aplicativo Express definido no arquivo “app.js”. Em seguida, é definida uma porta para o servidor, que pode ser especificada através da variável de ambiente “process.env.PORT”, ou padrão para a porta 3001 caso não esteja definida. Posteriormente, é criado um servidor utilizando a função “http.createServer”, passando o “app” Express como argumento. Por fim, o servidor é configurado para escutar conexões na porta especificada, exibindo uma mensagem no console indicando que o servidor foi iniciado com sucesso. Dessa forma, este código

estabelece a infraestrutura básica para executar um servidor web que responde às requisições HTTP.



```
1 const http = require('http');
2 const app = require('./src/app');
3
4 const port = parseInt(process.env.PORT) || 3001;
5 const server = http.createServer(app);
6 server.listen(port, () => {
7   console.log('Server started on port http://localhost:${port}...');
8 });
```

Figura 2 – index.js



```
1 const express = require('express');
2 const bodyParser = require('body-parser');
3 const routes = require('./routes');
4
5 const app = express();
6 app.use(bodyParser.urlencoded({extended: false}));
7 app.use(express.json());
8 app.use(routes);
9
10 module.exports = app;
```

Figura 3 – app.js

A configuração do banco de dados é centralizada no arquivo database.js, onde a biblioteca "sqlite3" é importada. Neste arquivo, é criada uma constante denominada "db", representando uma nova instância de um banco de dados SQLite. Subsequentemente, a função "db.serialize()" é invocada, tomando como argumento uma função de retorno de chamada (callback). Dentro dessa função de retorno de chamada, é executado um script para a criação do banco de dados, das tabelas associadas, e também para a inserção de alguns registros iniciais.

É importante notar que esse script é executado apenas se os dados correspondentes ainda não existirem no banco, garantindo que seja gerado somente uma vez ao iniciar o projeto. Essa abordagem assegura a integridade e consistência do banco de dados, evitando redundâncias na criação de estruturas já existentes. O processo de configuração do banco de dados é, assim, efetuado de maneira automatizada, proporcionando uma inicialização suave do projeto com as estruturas necessárias previamente estabelecidas.

Figura 4 – database.js

```

1  curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
2
3    "url": "http://10.10.10.10:8080",
4
5    "method": "POST",
6
7    "body": "10.10.10.10:8080"
8  }'
9
10 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
11
12   "url": "http://10.10.10.10:8080",
13
14   "method": "POST",
15
16   "body": "10.10.10.10:8080"
17 }'
18
19 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
20
21   "url": "http://10.10.10.10:8080",
22
23   "method": "POST",
24
25   "body": "10.10.10.10:8080"
26 }'
27
28 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
29
30   "url": "http://10.10.10.10:8080",
31
32   "method": "POST",
33
34   "body": "10.10.10.10:8080"
35 }'
36
37 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
38
39   "url": "http://10.10.10.10:8080",
40
41   "method": "POST",
42
43   "body": "10.10.10.10:8080"
44 }'
45
46 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
47
48   "url": "http://10.10.10.10:8080",
49
50   "method": "POST",
51
52   "body": "10.10.10.10:8080"
53 }'
54
55 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
56
57   "url": "http://10.10.10.10:8080",
58
59   "method": "POST",
60
61   "body": "10.10.10.10:8080"
62 }'
63
64 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
65
66   "url": "http://10.10.10.10:8080",
67
68   "method": "POST",
69
70   "body": "10.10.10.10:8080"
71 }'
72
73 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
74
75   "url": "http://10.10.10.10:8080",
76
77   "method": "POST",
78
79   "body": "10.10.10.10:8080"
80 }'
81
82 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
83
84   "url": "http://10.10.10.10:8080",
85
86   "method": "POST",
87
88   "body": "10.10.10.10:8080"
89 }'
90
91 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
92
93   "url": "http://10.10.10.10:8080",
94
95   "method": "POST",
96
97   "body": "10.10.10.10:8080"
98 }'
99
100 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
101
102   "url": "http://10.10.10.10:8080",
103
104   "method": "POST",
105
106   "body": "10.10.10.10:8080"
107 }'
108
109 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
110
111   "url": "http://10.10.10.10:8080",
112
113   "method": "POST",
114
115   "body": "10.10.10.10:8080"
116 }'
117
118 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
119
120   "url": "http://10.10.10.10:8080",
121
122   "method": "POST",
123
124   "body": "10.10.10.10:8080"
125 }'
126
127 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
128
129   "url": "http://10.10.10.10:8080",
130
131   "method": "POST",
132
133   "body": "10.10.10.10:8080"
134 }'
135
136 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
137
138   "url": "http://10.10.10.10:8080",
139
140   "method": "POST",
141
142   "body": "10.10.10.10:8080"
143 }'
144
145 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
146
147   "url": "http://10.10.10.10:8080",
148
149   "method": "POST",
150
151   "body": "10.10.10.10:8080"
152 }'
153
154 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
155
156   "url": "http://10.10.10.10:8080",
157
158   "method": "POST",
159
160   "body": "10.10.10.10:8080"
161 }'
162
163 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
164
165   "url": "http://10.10.10.10:8080",
166
167   "method": "POST",
168
169   "body": "10.10.10.10:8080"
170 }'
171
172 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
173
174   "url": "http://10.10.10.10:8080",
175
176   "method": "POST",
177
178   "body": "10.10.10.10:8080"
179 }'
180
181 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
182
183   "url": "http://10.10.10.10:8080",
184
185   "method": "POST",
186
187   "body": "10.10.10.10:8080"
188 }'
189
190 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
191
192   "url": "http://10.10.10.10:8080",
193
194   "method": "POST",
195
196   "body": "10.10.10.10:8080"
197 }'
198
199 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
200
201   "url": "http://10.10.10.10:8080",
202
203   "method": "POST",
204
205   "body": "10.10.10.10:8080"
206 }'
207
208 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
209
210   "url": "http://10.10.10.10:8080",
211
212   "method": "POST",
213
214   "body": "10.10.10.10:8080"
215 }'
216
217 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
218
219   "url": "http://10.10.10.10:8080",
220
221   "method": "POST",
222
223   "body": "10.10.10.10:8080"
224 }'
225
226 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
227
228   "url": "http://10.10.10.10:8080",
229
230   "method": "POST",
231
232   "body": "10.10.10.10:8080"
233 }'
234
235 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
236
237   "url": "http://10.10.10.10:8080",
238
239   "method": "POST",
240
241   "body": "10.10.10.10:8080"
242 }'
243
244 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
245
246   "url": "http://10.10.10.10:8080",
247
248   "method": "POST",
249
250   "body": "10.10.10.10:8080"
251 }'
252
253 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
254
255   "url": "http://10.10.10.10:8080",
256
257   "method": "POST",
258
259   "body": "10.10.10.10:8080"
260 }'
261
262 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
263
264   "url": "http://10.10.10.10:8080",
265
266   "method": "POST",
267
268   "body": "10.10.10.10:8080"
269 }'
270
271 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
272
273   "url": "http://10.10.10.10:8080",
274
275   "method": "POST",
276
277   "body": "10.10.10.10:8080"
278 }'
279
280 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
281
282   "url": "http://10.10.10.10:8080",
283
284   "method": "POST",
285
286   "body": "10.10.10.10:8080"
287 }'
288
289 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
290
291   "url": "http://10.10.10.10:8080",
292
293   "method": "POST",
294
295   "body": "10.10.10.10:8080"
296 }'
297
298 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
299
300   "url": "http://10.10.10.10:8080",
301
302   "method": "POST",
303
304   "body": "10.10.10.10:8080"
305 }'
306
307 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
308
309   "url": "http://10.10.10.10:8080",
310
311   "method": "POST",
312
313   "body": "10.10.10.10:8080"
314 }'
315
316 curl -u admin -m 5000 -X POST -H 'Content-Type: application/json' -d '{
317
318   "url": "http://10.10.10.10:8080",
319
320   "method": "POST",
321
322   "body": "10.10.10.10:808
```

Figura 5 – routes.js

Os arquivos **repositories** são essencialmente centrados nas operações fundamentais de acesso a dados, como **findAll** para recuperar todos os registros, **findById** para encontrar um registro específico, **insert** para adicionar novos dados, **update** para modificar registros existentes e **deleteById** para excluir um registro com base no ID. Essas funções formam a base para interações cruciais com o banco de dados, garantindo uma gestão eficiente e eficaz dos dados armazenados.

A função **findAll** é uma função assíncrona que realiza uma consulta ao banco de dados para recuperar todos os registros de uma tabela (conforme exemplificado na figura 6). Utiliza uma **Promise** para encapsular a operação assíncrona, empregando a função **db.each** do **SQLite**. Durante a execução dessa função, os resultados da consulta são iterados, cada linha é adicionada ao array da entidade. Em caso de erro, a **Promise** é rejeitada com a indicação do erro. Quando a consulta é concluída, a **Promise** é resolvida com o array contendo todos os registros da tabela.

A screenshot of a code editor with a dark background and light-colored text. The code is written in JavaScript and implements an asynchronous function named `findAll`. The function uses a `Promise` to encapsulate the database query. It starts by initializing an empty array `produtos`. Then, it uses `db.each` to execute a SQL query: `'SELECT * FROM Produtos ORDER BY id'`. The `each` method takes a callback function that receives an error (`err`) and a row (`row`). If there is an error, it logs a message to the console and rejects the promise. If successful, it pushes the `row` into the `produtos` array. After the `each` loop finishes, it checks for any errors and either rejects the promise or resolves it with the `produtos` array. The code is numbered from 1 to 15.

```
1  async function findAll() {
2      return new Promise((resolve, reject) => {
3          const produtos = [];
4          db.each('SELECT * FROM Produtos ORDER BY id', (err, row) => {
5              if (err) {
6                  console.error('Ocorreu um erro ao localizar todos os produtos!');
7                  reject(err);
8              }
9              produtos.push(row);
10         }, (err, count) => {
11             if (err) reject(err);
12             resolve(produtos);
13         });
14     });
15 }
```

Figura 6 – função **findAll**.

A função **findById** busca um registro no banco de dados com base em um ID específico (conforme exemplificado na figura 7). Ela utiliza uma **Promise** para encapsular a operação assíncrona, utilizando a função **db.prepare** do **SQLite** para preparar uma instrução SQL parametrizada que seleciona um registro cujo ID corresponde ao parâmetro fornecido. A função **stmt.get** é então utilizada para executar a consulta e obter o resultado. Em caso de

erro durante a operação, a **Promise** é rejeitada com o erro correspondente. Se a consulta for bem-sucedida, a **Promise** é resolvida com a linha resultante contendo as informações do produto. Importante notar que a instrução é finalizada com **stmt.finalize()**, assegurando a liberação adequada dos recursos. Essa estrutura permite a busca eficiente de registros pelo ID de forma assíncrona.



```
1  async function findById(id) {
2      return new Promise((resolve, reject) => {
3          const stmt = db.prepare('SELECT * FROM Produtos WHERE id = ?', [id]);
4          stmt.get((err, row) => {
5              if (err) {
6                  console.error('Ocorreu um erro ao localizar produto pelo ID!');
7                  reject(err);
8              }
9              resolve(row);
10         });
11         stmt.finalize();
12     });
13 }
```

Figura 7 – função **findById**.

A função **insert** tem como objetivo inserir um novo produto no banco de dados. Utiliza uma **Promise** para encapsular a operação assíncrona, empregando a função **db.prepare** do **SQLite** para preparar uma instrução SQL de inserção. Os valores do novo produto são vinculados à instrução usando **stmt.bind**, e a inserção é executada com **stmt.run**. Em caso de erro durante a inserção, a **Promise** é rejeitada com o erro correspondente. Após a inserção bem-sucedida, a função continua a obter o ID gerado para o novo registro utilizando uma segunda instrução SQL. O ID é então utilizado para buscar o produto recém-inserido através da função **findById**. O resultado dessa busca é então passado para a resolução da **Promise**. Dessa forma, essa função não apenas insere um produto, mas também retorna as informações completas do produto inserido, facilitando o uso subsequente dessa informação no código. No final, as instruções são finalizadas para liberar os recursos associados.

```

1  async function insert(produto) {
2      return new Promise((resolve, reject) => {
3          const stmt = db.prepare('INSERT INTO Produtos(nome, descricao, preco) VALUES(?, ?, ?)');
4          stmt.bind([produto.nome, produto.descricao, produto.preco]);
5          stmt.run(err => {
6              if (err) {
7                  console.error('Ocorreu um erro ao inserir produto!');
8                  reject(err);
9              }
10         });
11         stmt.finalize();
12         const stmt2 = db.prepare('SELECT seq FROM sqlite_sequence WHERE name = "Produtos"');
13         stmt2.get((err, row) => {
14             resolve(findById(row ? row['seq'] + 1 : 1));
15         });
16         stmt2.finalize();
17     });
18 }

```

Figura 8 – função **insert**.

A função **update** tem como finalidade realizar a atualização de informações de um registro no banco de dados. Ela utiliza uma **Promise** para encapsular a operação assíncrona, empregando a função **db.prepare** do **SQLite** para preparar uma instrução SQL de atualização. Os valores do produto a serem atualizados são vinculados à instrução utilizando **stmt.bind**, e a atualização é executada com **stmt.run**. Em caso de erro durante a operação, a **Promise** é rejeitada com o erro correspondente. Se a atualização for bem-sucedida, a **Promise** é resolvida sem a necessidade de retornar dados específicos, indicando que a operação foi concluída com êxito. Finalmente, a instrução é finalizada para liberar os recursos associados.

```

1  async function update(produto) {
2      return new Promise((resolve, reject) => {
3          const stmt = db.prepare('UPDATE Produtos SET nome = ?, descricao = ?, preco = ? WHERE id = ?');
4          stmt.bind([produto.nome, produto.descricao, produto.preco, produto.id]);
5          stmt.run(err => {
6              if (err) {
7                  console.error('Ocorreu um erro na atualização do produto!');
8                  reject(err);
9              }
10             resolve();
11         });
12         stmt.finalize();
13     });
14 }

```

Figura 9 – função **update**.

A função **deleteById** tem como propósito excluir um produto do banco de dados com base no seu ID. Utiliza uma Promise para encapsular a operação assíncrona, empregando a função **db.prepare** do **SQLite** para preparar uma instrução SQL de exclusão. O ID do produto a ser removido é vinculado à instrução usando **stmt.bind**, e a exclusão é executada com **stmt.run**. Em caso de erro durante a operação, a **Promise** é rejeitada com o erro correspondente. Se a exclusão for bem-sucedida, a **Promise** é resolvida sem retornar dados específicos, indicando que a operação foi concluída com sucesso. Finalmente, a instrução é finalizada para liberar os recursos associados. Essa estrutura encapsula de forma eficaz a lógica de exclusão no banco de dados, fornecendo um código assíncrono claro e compreensível, facilitando a manutenção e compreensão do sistema.



```
1  async function deleteById(id) {
2    return new Promise((resolve, reject) => {
3      const stmt = db.prepare('DELETE FROM Produtos WHERE id = ?');
4      stmt.bind([id]);
5      stmt.run(err => {
6        if (err) {
7          console.error('Ocorreu um erro com ao deletar produto!');
8          reject(err);
9        }
10       resolve();
11     });
12     stmt.finalize();
13   });
14 }
```

Figura 10 – função **deleteById**.

Os arquivos de controllers, que por sua vez importam seus respectivos repositories, concentram-se principalmente nas operações essenciais de manipulação de dados, incluindo as funções "get" para recuperar dados, **getById** para obter um registro específico, **post** para enviar novos dados, **putById** para atualizar registros existentes e **deleteById** para excluir um registro com base no identificador. Essas funções desempenham um papel crucial na interação entre a aplicação e o sistema, possibilitando a comunicação eficiente e a gestão adequada dos dados.

A função **get** é uma função assíncrona projetada para lidar com solicitações **HTTP**, como uma requisição GET. Dentro da função, ela chama a função **findAll** do seu respectivo **repository** de forma assíncrona, utilizando o operador **await**. Essa abordagem permite que o código aguarde a conclusão da operação **findAll** antes de prosseguir para a próxima linha, garantindo que a resposta da solicitação só seja enviada após a obtenção de todos os produtos. Posteriormente, a função responde à requisição **HTTP** com os registros obtidos, convertendo-os para o formato JSON por meio da chamada **res.json(produtos)** (conforme exemplificado na figura 11). Essa estrutura assíncrona é útil para operações que envolvem a busca de dados, garantindo um fluxo de execução controlado e evitando bloqueios durante a espera por operações demoradas, como consultas a bancos de dados.

A função **getById** é uma função assíncrona destinada a lidar com solicitações **HTTP**, especificamente para obter informações sobre um produto com base em um identificador fornecido nos parâmetros da requisição (**req.params.id**). Dentro da função, é utilizado o método assíncrono **findById** do objeto **produtoRepository** para recuperar o produto correspondente ao ID fornecido (conforme exemplificado na figura 12). Em seguida, verifica-se se o produto foi encontrado. Se não for o caso, a função responde com um status **HTTP 404** (não encontrado) e envia um objeto JSON indicando que o produto não foi encontrado. Caso o produto seja encontrado, a função responde à requisição **HTTP** com os detalhes do produto no formato JSON.



```
1 async function get(req, res) {
2   const produtos = await produtoRepository.findAll();
3   res.json(produtos);
4 }
```

Figura 11 – função **get**.



```
1 async function getById(req, res) {
2   const produto = await produtoRepository.findById(req.params.id);
3   if (!produto) {
4     res.status(404).json({message: 'Produto não encontrado!'});
5     return;
6   }
7   res.json(produto);
8 }
```

Figura 12 – função **getById**.

A função **post** dentro do contexto é uma função assíncrona projetada para um novo recurso. No cenário específico (figura 13), a função utiliza o **produtoRepository** para inserir os dados contidos na requisição (**req.body**), que geralmente incluem informações sobre um novo recurso, como um produto. Se a operação de inserção for bem-sucedida, a função responde à requisição com um código de status **HTTP 201** (Created), indicando

que o recurso foi criado com sucesso, e retorna os dados do produto criado em formato JSON.

A função **putById** lida com requisições para atualizar um recurso específico no servidor. Utilizando o **produtoRepository** (conforme exemplificado na figura 14), ela busca o produto associado ao ID fornecido na requisição (`req.params.id`). Se o produto não é encontrado, a função responde com um código de status HTTP 404 (Not Found) e uma mensagem indicando a ausência do produto. Caso o produto seja encontrado, a função continua com a atualização dos dados, empregando o método **update** do **produtoRepository**.



```
1 async function post(req, res) {
2   try {
3     const produto = await produtoRepository.insert(req.body);
4     res.status(201).json(produto);
5   } catch (error) {
6     console.error(error);
7     res.status(500).json({ error: 'Erro do Servidor Interno.' });
8   }
9 }
```

Figura 13 – Função **post**.



```
1 async function putById(req, res) {
2   const produto = await produtoRepository.findById(req.params.id);
3   if (!produto) {
4     res.status(404).json({ message: 'Produto não encontrado!' });
5     return;
6   }
7   await produtoRepository.update(req.body);
8   res.status(204).json();
9 }
```

Figura 14 – Função **putById**.

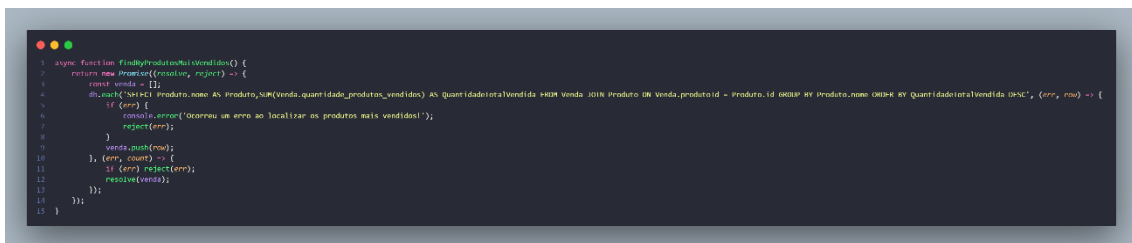
A função **deleteById** gerencia requisições para excluir um recurso no servidor. Ela utiliza o **produtoRepository** (conforme exemplificado na figura 15) para buscar o produto associado ao ID fornecido na requisição (`req.params.id`). Se o produto não é encontrado, a função responde com um código de status HTTP 404 (Not Found) e uma mensagem informando a ausência do produto. Em caso de localização do produto, a função continua com a exclusão do registro, utilizando o método **deleteById** do **produtoRepository**.



```
1 async function deleteById(req, res) {
2   const produto = await produtoRepository.findById(req.params.id);
3   if (!produto) {
4     res.status(404).json({ message: 'Produto não encontrado!' });
5     return;
6   }
7   await produtoRepository.deleteById(produto.id);
8   res.status(204).json();
9 }
```

Figura 15 – Função **deleteById**.

Para obter os relatórios, dentro do arquivo **relatorios-repository.js** temos as funções **findByProdutosMaisVendidos**, **findByProdutosPorCliente**, **findByConsumoCliente**, **findByEstoqueBaixo**. A função **findByProdutosMaisVendidos** é uma operação assíncrona em JavaScript que utiliza Promises para consultar um banco de dados, recuperando informações sobre os produtos mais vendidos. A consulta SQL agrupa os resultados pelo nome do produto, ordenando-os por quantidade total vendida em ordem decrescente. A função retorna uma Promise que resolve com um array de objetos representando cada produto e sua quantidade total vendida, ou rejeita em caso de erro, fornecendo uma mensagem de erro. Para acessar a função **findByProdutosMaisVendidos** dentro do arquivo **relatorios-controller.js** temos a função **getByProdutosMaisVendidos** que recupera informações sobre os produtos mais vendidos. Os resultados são então enviados como uma resposta JSON à requisição HTTP.



```
1 async function findByProdutosMaisVendidos() {
2   return new Promise((resolve, reject) => {
3     const venda = [];
4     db.query('SELECT produto.nome AS Produto, SUM(Venda.quantidade_produto_vendidos) AS QuantidadeTotalVendida FROM Venda JOIN Produto ON Venda.produtoId = Produto.id GROUP BY Produto.nome ORDER BY QuantidadeTotalVendida DESC', (err, row) => {
5       if (err) {
6         console.error('Ocorreu um erro ao localizar os produtos mais vendidos!');
7         reject(err);
8       }
9       venda.push(row);
10      1, (err, count) => {
11        if (err) reject(err);
12        resolve(venda);
13      });
14    });
15  });
16 }
```

Figura 16 – Função **findByProdutosMaisVendidos**.



```
1 async function getByProdutosMaisVendidos(req, res) {
2   const produtosMaisVendidos = await relatorioRepository.findByProdutosMaisVendidos();
3   res.json(produtosMaisVendidos);
4 }
```

Figura 17 – Função **getByProdutosMaisVendidos**.

A função **findByProdutosPorCliente** realiza uma consulta ao banco de dados para obter informações sobre os produtos comprados por cada cliente. Utilizando Promises, a função junta dados das tabelas de Venda, Cliente e Produto, recuperando o nome do cliente, nome do produto e quantidade comprada. Os resultados são armazenados em um array chamado venda. Em caso de sucesso, a Promise resolve com esse array, enquanto em caso de erro, é rejeitada com uma mensagem de erro. Para acessar essa função, dentro do arquivo

relatórios-controller.js temos a função **getByProdutosPorCliente** que recupera os produtos comprados por cada cliente. Os resultados são então enviados como uma resposta JSON à requisição HTTP.

```
1 async function findByProdutosPorCliente() {
2   return new Promise((resolve, reject) => {
3     const venda = [];
4     db.each("SELECT C.nome AS NomeCliente, P.nome AS NomeProduto, V.quantidade_produtos_vendidos AS QuantidadeComprada FROM Venda V JOIN Cliente C ON V.clienteId = C.id JOIN Produto P ON V.produtoId = P.id", (err, row) => {
5       if (err) {
6         console.error('Ocorreu um erro ao localizar os produtos por cliente!');
7         reject(err);
8       }
9       venda.push(row);
10    }, (err, count) => {
11      if (err) reject(err);
12      resolve(venda);
13    });
14  });
15 }
```

Figura 18 – Função **findByProdutosPorCliente**.

```
1 async function getByProdutosPorCliente(req, res) {
2   const produtosPorClientes = await relatorioRepository.findByProdutosPorCliente();
3   res.json(produtosPorClientes);
4 }
```

Figura 19 – Função **getByProdutosPorCliente**.

A função **findByConsumoCliente** realiza uma consulta ao banco de dados para calcular o consumo médio de cada cliente em um sistema de gerenciamento de vendas. Os resultados são agrupados pelo ID e nome do cliente, ordenados pela média de consumo em ordem decrescente. Os dados são armazenados em um array chamado venda, e a função retorna uma Promise que, em caso de sucesso, resolve com esse array contendo objetos representando cada cliente e sua média de consumo. Em caso de erro, a Promise é rejeitada com uma mensagem de erro. Para acessar essa função, dentro do arquivo **relatórios-controller.js** temos a função **getByConsumoCliente** que recupera o consumo médio de cada cliente. Os resultados são então enviados como uma resposta JSON à requisição HTTP.

```

1 async function findByConsumoCliente() {
2   return new Promise((resolve, reject) => {
3     const venda = [];
4     db.query('SELECT Cliente.id AS ClienteID, Cliente.nome AS ClienteNome, SUM(Venda.valor_total) AS ConsumoMedio FROM Cliente JOIN Venda ON Cliente.id = Venda.clienteId GROUP BY Cliente.id, Cliente.nome ORDER BY ConsumoMedio DESC', (err, row) => {
5       if (err) {
6         console.error('Ocorreu um erro ao localizar os produtos pelo consumo do cliente!');
7         reject(err);
8       }
9       venda.push(row);
10      }, (err, count) => {
11        if (err) reject(err);
12        resolve(venda);
13      });
14    });
15  }

```

Figura 20 – Função **findByConsumoCliente**.

```

1 async function getByConsumoCliente(req, res) {
2   const consumoCliente = await relatorioRepository.findByConsumoCliente();
3   res.json(consumoCliente);
4 }

```

Figura 21 – Função **getByConsumoCliente**.

A função **findByEstoqueBaixo** realiza uma consulta ao banco de dados para obter informações sobre produtos com estoque inferior a 50 unidades. Os resultados são armazenados em um array chamado venda. A função retorna uma Promise que, em caso de sucesso, resolve com esse array contendo objetos representando cada produto e sua quantidade em estoque. Em caso de erro, a Promise é rejeitada com uma mensagem de erro. Para acessar essa função, dentro do arquivo **relatórios-controller.js** temos a função **getByEstoqueBaixo** que recupera os produtos que estão com estoque baixo. Os resultados são então enviados como uma resposta JSON à requisição HTTP.

```

1 async function findByEstoqueBaixo() {
2   return new Promise((resolve, reject) => {
3     const venda = [];
4     db.query('SELECT Produto.nome AS NomeProduto, Estoque.quantidade_em_estoque AS QuantidadeEmEstoque FROM Produto JOIN Estoque ON Produto.id = Estoque.produtoId WHERE Estoque.quantidade_em_estoque < 50', (err, row) => {
5       if (err) {
6         console.error('Ocorreu um erro ao localizar os produtos com baixo estoque!');
7         reject(err);
8       }
9       venda.push(row);
10      }, (err, count) => {
11        if (err) reject(err);
12        resolve(venda);
13      });
14    });
15  }

```

Figura 22 – Função **findByEstoqueBaixo**.



```
1  async function getByEstoqueBaixo(req, res) {  
2      const baixoEstoque = await relatorioRepository.findByEstoqueBaixo();  
3      res.json(baixoEstoque);  
4  }
```

Figura 23 – Função **getByEstoqueBaixo**.

6 Postman

Documentação postman:

<https://documenter.getpostman.com/view/26587593/2s9YeBfZZ9>

Coleção postman:

https://mega.nz/file/jtxnCZRK#nXcmLViRKe_T-ybFsK7ZdsHBEDnADsP6pQraQfbvEcg

7 Considerações Finais

Nas considerações finais, é notável ressaltar a importância das funções assíncronas apresentadas, destacando seu papel fundamental no desenvolvimento de aplicações modernas. A utilização de Promises e a palavra-chave `await` demonstra um avanço significativo no tratamento de operações assíncronas em JavaScript, tornando o código mais eficiente e legível.

Ao examinar as funções específicas, como **`getByProdutosMaisVendidos`**, **`getByProdutosPorCliente`** e **`getByConsumoCliente`**, percebemos a versatilidade dessas operações para atender às demandas complexas de sistemas de gerenciamento de vendas. Essas funcionalidades proporcionam aos desenvolvedores meios eficazes para extrair e apresentar dados estatísticos de maneira precisa e oportuna.

A abordagem de consultas ao banco de dados, reflete a importância de compreender e otimizar as interações com a persistência de dados. Essa prática é crucial para garantir a eficiência operacional e a resposta ágil às necessidades dos usuários.

Em suma, as funções assíncronas discutidas neste contexto representam uma valiosa contribuição para o desenvolvimento de software robusto, eficiente e adaptável. A compreensão desses conceitos e sua implementação adequada são essenciais para construir aplicações modernas e responsivas, alinhadas com as expectativas crescentes dos usuários e as demandas dinâmicas do ambiente digital.

8 Referências:

Node.js: <https://nodejs.org/en>

Npm: <https://www.npmjs.com/>

Express: <https://www.npmjs.com/package/express>

Body-parser: <https://www.npmjs.com/package/body-parser>

Nodemon: <https://www.npmjs.com/package/nodemon>

Sqlite3: <https://www.npmjs.com/package/sqlite3>