Estruturas de Dados Utilizadas no Projeto

Neste projeto, desenvolvido com Node.js, Express.js e SQLite, utilizamos diversas estruturas de dados fundamentais para a manipulação, armazenamento e recuperação eficiente das informações dos usuários do sistema. A seguir, descrevemos detalhadamente cada uma delas, sua função no projeto e os motivos que justificam sua escolha.

- ## Banco de Dados Relacional (SQLite)

A principal estrutura de dados usada no projeto é o banco de dados relacional **SQLite**, que foi escolhido por ser leve, embutido e de fácil configuração para ambientes de desenvolvimento e testes.

Tabela users
```
db.run(
  `CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nome TEXT NOT NULL,
    telefone TEXT NOT NULL,
    email TEXT NOT NULL,
    cpf TEXT NOT NULL,
    sexo TEXT NOT NULL CHECK (sexo IN ('feminino', 'masculino')),
    senha TEXT NOT NULL,
    aceita_motorista_mulher BOOLEAN DEFAULT 0
)`);
```

Essa tabela representa uma estrutura tabular composta por linhas e colunas. Cada coluna representa um campo de dado do usuário (como nome, email, sexo), e cada linha representa uma instância (ou seja, um usuário registrado). Essa organização permite armazenar os dados de forma estruturada, garantindo integridade e consistência com validações como **not null**, **check** e **boolean**.

- ## Objetos JavaScript (req.body, req.params, user, etc.)

Durante o processamento das requisições HTTP, utilizamos objetos para representar e manipular os dados dos usuários.

```
const { nome, telefone, email, cpf, sexo, senha, aceita_motorista_mulher } = req.body;
```

Aqui, req.body é um objeto que representa o corpo da requisição recebida via POST. Ele contém pares chave-valor, que permitem acessar diretamente os dados enviados pelo usuário.

Essa estrutura é ideal pela sua flexibilidade e compatibilidade com os padrões de APIs RESTful.

Também usamos objetos para representar o resultado de consultas no banco:

db.get(`SELECT sexo, aceita_motorista_mulher FROM users WHERE id = ?`, [usuarioId], (err, user) => {

});

- **Arrays (Lista de usuários e motoristas)**

O SQLite retorna conjuntos de dados em arrays de objetos, especialmente ao buscar múltiplos registros com db.all():

db.all("SELECT * FROM users", [], (err, rows) => {

  res.json(rows);

});

Neste trecho, a variável rows é um array onde cada elemento é um objeto representando um usuário. Essa estrutura é fundamental para manipular listas de dados, como na exibição de todos os usuários cadastrados ou motoristas disponíveis.

- **Tipos Primitivos (Strings, Booleanos, Inteiros)**

Usamos tipos primitivos nas definições dos campos:

- TEXT (string): para nomes, e-mails, senhas etc.
- BOOLEAN (0 ou 1): para preferências como aceita_motorista_mulher.
- INTEGER: para o campo id, como chave primária.

Esses tipos são simples, eficientes e apropriados para o tipo de dados que o projeto manipula. Exemplo prático:

[nome, telefone, email, cpf, sexo, senha, aceita_motorista_mulher ? 1 : 0]

Aqui, convertendo aceita_motorista_mulher em 1 ou 0, garantimos a compatibilidade com o tipo **BOOLEAN** do SQLite, que na prática representa valores lógicos como true ou false.

- **Strings Dinâmicas (para consultas SQL)**

Algumas consultas SQL são construídas dinamicamente com base nas condições do usuário, o que utiliza strings concatenadas como estrutura:

```
let query = `SELECT * FROM motoristas`;

if (user.sexo === "feminino" && user.aceita_motorista_mulher) {

  query += ` WHERE sexo = 'feminino'`;

}
```

Essa forma de montagem dinâmica permite flexibilidade e personalização da lógica, adaptando as consultas com base nas preferências dos usuários.

Justificativa das Escolhas

As estruturas de dados escolhidas foram baseadas em critérios como:

- **Simplicidade e eficiência**: SQLite permite armazenar dados relacionais de maneira leve e eficaz.

- **Facilidade de acesso**: objetos JavaScript permitem acesso rápido aos dados com sintaxe simples.

- **Escalabilidade**: arrays são ideais para lidar com múltiplos registros e permitem iteração fácil.

- **Segurança e Validação**: o uso de tipos apropriados e constraints no SQLite assegura que os dados inseridos sejam consistentes.