Documentação Técnica do Banco de Dados - Projeto Taskea

Introdução

Esta seção descreve a camada de persistência de dados da aplicação Taskea, detalhando a configuração do banco de dados, o esquema das tabelas, os relacionamentos entre entidades e como o Sequelize ORM é utilizado para gerenciar a interação com o banco de dados SQLite.

Configuração do Banco de Dados (`config/database/database.js`)

O coração da configuração da persistência de dados reside neste arquivo. Ele é responsável por inicializar e configurar a instância do Sequelize que a aplicação utilizará para se conectar e interagir com o banco de dados.

Dependências:

O arquivo importa `Sequelize` da biblioteca `sequelize` , `path` para manipulação de caminhos de arquivo e `fs` (File System) para interações com o sistema de arquivos.

```
 const { Sequelize } = require("sequelize");
const path = require("path");
const fs = require("fs");
```

Lógica de Configuração:

1. **Detecção de Ambiente**: Uma variável `isProduction` é definida com base na variável de ambiente `process.env.NODE_ENV` . No entanto, a lógica subsequente para determinar o caminho do banco de dados (`dbPath`) utiliza o mesmo caminho relativo (`../../data/tarefas.db`) tanto para produção quanto para desenvolvimento. Isso significa que, independentemente do ambiente, a aplicação tentará usar um arquivo SQLite localizado em um diretório `data` dois níveis acima do diretório `config/database` .

   ```javascript
   ```

const isProduction = process.env.NODE_ENV === "production";
const dbPath = isProduction
? path.resolve(__dirname, "../../data/tarefas.db")
path.resolve(__dirname, "../../data/tarefas.db"); ```

2. **Criação do Diretório**: Antes de configurar o Sequelize, o script verifica se o diretório onde o arquivo do banco de dados deve residir (`data/`) existe. Se não existir, ele o cria recursivamente usando `fs.mkdirSync(dbDir, { recursive: true })` . Isso garante que o Sequelize não falhe ao tentar criar ou acessar o arquivo `tarefas.db` em um diretório inexistente.

   `javascript const dbDir = path.dirname(dbPath); if (!fs.existsSync(dbDir)) { fs.mkdirSync(dbDir, { recursive: true }); console.log(`Diretório criado: ${dbDir}`); }`

3. **Instanciação do Sequelize**: Uma nova instância do Sequelize é criada com as seguintes configurações:

   - `dialect: 'sqlite'` : Especifica que o banco de dados a ser utilizado é o SQLite.
   - `storage: dbPath` : Indica o caminho completo para o arquivo `.db` que o SQLite utilizará para armazenar os dados. É o caminho resolvido anteriormente.
   - `logging: false` : Desabilita o log das queries SQL executadas pelo Sequelize no console. Em desenvolvimento, pode ser útil definir como `console.log` para depuração.

   `javascript const sequelize = new Sequelize({ dialect: "sqlite", storage: dbPath, logging: false });`

4. **Exportação**: A instância configurada do `sequelize` é exportada para ser utilizada em outras partes da aplicação, principalmente no arquivo `config/index.js` (que não foi explicitamente solicitado para documentação, mas é onde os modelos são associados à instância do Sequelize e a sincronização é chamada).

   `javascript module.exports = sequelize;`

Esquema do Banco de Dados (Baseado nos Modelos Sequelize)

O Sequelize utiliza os modelos definidos em `models/Usuario.js` e `models/Tarefa.js` para inferir e gerenciar o esquema do banco de dados SQLite. Com base nesses modelos, o esquema resultante consiste em duas tabelas principais:

Tabela `usuarios`

Mapeada a partir do modelo `Usuario`.

- **`id`**: INTEGER, PRIMARY KEY, AUTOINCREMENT - Identificador único para cada usuário.
- **`nome`**: VARCHAR(255) (ou TEXT, dependendo da implementação do SQLite no Sequelize), NOT NULL - Nome do usuário.
- **`email`**: VARCHAR(255) (ou TEXT), NOT NULL, UNIQUE - Endereço de email do usuário, com uma restrição de unicidade para evitar duplicatas.
- **`senha_hash`**: VARCHAR(255) (ou TEXT), NULL - Armazena o hash bcrypt da senha do usuário. O campo `senha` virtual do modelo não existe na tabela.
- **`createdAt`**: DATETIME, NOT NULL - Timestamp de quando o registro do usuário foi criado (adicionado automaticamente pelo Sequelize).
- **`updatedAt`**: DATETIME, NOT NULL - Timestamp da última atualização do registro do usuário (adicionado automaticamente pelo Sequelize).

Tabela `tarefas`

Mapeada a partir do modelo `Tarefa`.

- **`id`**: INTEGER, PRIMARY KEY, AUTOINCREMENT - Identificador único para cada tarefa.
- **`titulo`**: VARCHAR(255) (ou TEXT), NOT NULL - Título da tarefa.
- **`descricao`**: VARCHAR(255) (ou TEXT), NOT NULL - Descrição da tarefa.
- **`status`**: VARCHAR(255) (ou TEXT, contendo 'pendente', 'em andamento' ou 'concluida'), NOT NULL, DEFAULT 'pendente' - Status atual da tarefa, com valor padrão 'pendente'. A restrição ENUM é gerenciada pela aplicação/Sequelize, não necessariamente pelo SQLite nativamente de forma rigorosa.
- **`usuario_id`**: INTEGER, NULL (ou NOT NULL, dependendo da configuração exata da associação), FOREIGN KEY REFERENCES `usuarios` (`id`) ON DELETE CASCADE - Chave estrangeira que liga a tarefa ao usuário proprietário na tabela `usuarios`. A configuração `ON DELETE CASCADE` garante que, se um usuário for excluído, todas as suas tarefas associadas também serão excluídas automaticamente pelo banco de dados.
- **`createdAt`**: DATETIME, NOT NULL - Timestamp de criação da tarefa.
- **`updatedAt`**: DATETIME, NOT NULL - Timestamp da última atualização da tarefa.

Relacionamentos

Existe um relacionamento **um-para-muitos** (One-to-Many) entre `usuarios` e `tarefas`:

- Um usuário (`Usuario`) pode ter várias tarefas (`Tarefa`).
- Cada tarefa (`Tarefa`) pertence a exatamente um usuário (`Usuario`).

Este relacionamento é estabelecido pela chave estrangeira `usuario_id` na tabela `tarefas`, que referencia a coluna `id` da tabela `usuarios`. A definição `onDelete: 'CASCADE'` na associação `belongsTo` do modelo `Tarefa` impõe a regra de integridade referencial que exclui tarefas órfãs quando o usuário correspondente é removido.

Gerenciamento pelo Sequelize

O Sequelize ORM abstrai a interação direta com o banco de dados SQLite. Ele é responsável por:

- **Definição do Esquema**: Traduzir as definições dos modelos (`Usuario`, `Tarefa`) em comandos SQL `CREATE TABLE`.
- **Sincronização**: O método `sequelize.sync()` (geralmente chamado em `config/index.js` ou `index.js`) compara os modelos definidos com o esquema existente no banco de dados e aplica as alterações necessárias (cria tabelas, adiciona colunas, etc.). A opção `{ force: false }` (usada em `index.js`) evita que as tabelas sejam descartadas e recriadas a cada inicialização, preservando os dados.
- **Operações CRUD**: Fornece métodos como `create`, `findAll`, `findByPk`, `update`, `destroy` nos modelos para manipular os dados de forma orientada a objetos, traduzindo essas chamadas em comandos SQL `INSERT`, `SELECT`, `UPDATE`, `DELETE`.
- **Gerenciamento de Associações**: Facilita a consulta e a manipulação de dados relacionados (por exemplo, buscar um usuário e incluir suas tarefas).

O uso do SQLite como banco de dados torna a aplicação autocontida e fácil de configurar, pois o banco de dados inteiro reside em um único arquivo (`data/tarefas.db`), ideal para desenvolvimento e cenários de implantação simples.