INSTITUTO FEDERAL CATARINENSE
CAMPUS SOMBRIO
CURSO TÉCNICO DE INFORMÁTICA PARA INTERNET

INTEGRADO AO ENSINO MÉDIO

BERNARDO DE MATTOS MOTA

DESVENDANDO O MUNDO DOS ORMS COM SEQUELIZE

SOMBRIO

2024

SUMÁRIO

Fundamentos dos ORMs

O que é um ORM?

ORM significa *Object-Relational Mapping* (Mapeamento Objeto-Relacional). É uma técnica que permite que a gente trabalhe com o banco de dados usando objetos, em vez de escrever comandos SQL diretamente. Com um ORM, como o Sequelize, conseguimos criar, ler, atualizar e deletar dados usando JavaScript, o que torna o código mais organizado e mais fácil de entender. O Paradigma da Impedância Objeto-Relacional é a diferença entre como os dados são representados no banco e como usamos eles no código, os ORMs fazem essa “tradução”.

Como um ORM funciona em alto nível?

Exemplo em fluxograma:

1. Usuario.findAll()

↓

2. ORM interpreta a requisição

↓

3. ORM gera o SQL equivalente (ex: SELECT * FROM Usuarios)

↓

4. SQL é enviado para o banco de dados

↓

5. Banco de dados executa a consulta e retorna os dados

↓

6. ORM recebe os dados e transforma em objetos JavaScript

↓

7. A aplicação recebe os dados prontos para usar

A camada de abstração é o que o ORM cria para esconder os detalhes do banco de dados. Com o ORM vemos objetos e métodos, mas por trás, ele ainda está rodando SQL normalmente.

Sequelize em Detalhes

O que é o Sequelize?

O Sequelize é um ORM para Node.js, é muito popular porque é fácil de usar, mesmo para quem está começando, funciona com diferentes tipos de banco de dados (MySQL, PostgreSQL, SQLite, MariaDB, Microsoft SQL Server…) tem uma documentação bem completa, uma comunidade ativa e ainda oferece recursos avançados como as associações.

Models e Associações: O Coração do Sequelize

No Sequelize, um **Model** representa uma tabela no banco de dados. Cada instância do model é como uma linha dessa tabela. Nele, definimos os nomes das colunas e os tipos de dados que elas vão armazenar.

models/Usuario.js
↓

```
const db = require('./db');

const Usuario = db.sequelize.define('Usuario', {
        cod: {
                type: db.Sequelize.INTEGER,
                primaryKey: true,
                autoIncrement: true,
                allowNull: false
           },
        nome: {
                type: db.Sequelize.STRING(40),
                allowNull: false
           },
        preco: {
                type: db.SequelizeDECIMAL(10, 2),
                allowNull: false
        }
```

As **associações** servem para definir o relacionamento entre os models (tabelas). O Sequelize oferece 4 tipos principais:

hasOne — Um-para-Um

hasMany — Um-para-Muitos

belongsTo — Pertence a

belongsToMany — Muitos-para-Muitos

models/Index.js

↓

Exemplo 1 (hasMany + belongsTo):

```
const db = require('./db');
const Usuario = require('./Usuario');
const Endereco = require('./Endereco');

	Usuario.belongsTo(Endereco, {
		foreignKey: 'enderecoID',
	});
	Endereco.hasMany(Usuario, {
		foreignKey: 'enderecoID',
	});
```

Exemplo 2 (hasOne + belongsTo):

```
const db = require('./db');
const Usuario = require('./Usuario');
const Perfil = require('./Perfil');

	Usuario.hasOne(Perfil, {
		foreignKey: 'usuarioID'
	 });
	Perfil.belongsTo(Usuario, {
		foreignKey: 'usuarioID'
	});
```

Exemplo 3 (belongsToMany):

```
const db = require('./db');
const Aluno = require('./Aluno');
const Disciplina = require('./Disciplina');

    Aluno.belongsToMany(Disciplina, {
        through: 'AlunoDisciplina', // tabela intermediária
        foreignKey: 'alunoID'
    });

    Disciplina.belongsToMany(Aluno, {
        through: 'AlunoDisciplina',
        foreignKey: 'disciplinaID'
    });
```

Quando usamos muitos-para-muitos, o Sequelize cria uma tabela extra para ligar os dois modelos. Essa tabela chama-se tabela de ligação ou tabela intermediária.O through é o nome dessa tabela.

Consultas (Queries) e a Abstração do SQL

SQL	Sequelize
SELECT * FROM Usuarios;	Usuario.findAll();
SELECT * FROM Usuarios WHERE id = 1;	Usuario.findByPk(1);
SELECT * FROM Usuarios WHERE enderecoID = 2;	Usuario.findAll({ where: { enderecoID: 2 } });
SELECT Usuarios.*, Enderecos.rua, Enderecos.cidade FROM Usuarios JOIN Enderecos ON Usuarios.enderecoID = Enderecos.cod;	Usuario.findAll({ include: Endereco});

Tópicos Avançados e Boas Práticas

O que são Migrations? Qual problema elas resolvem?

Migrations são arquivos que guardam passo a passo as mudanças no banco, como criação de tabelas, colunas, alterações e remoções. Elas resolvem o problema de manter o banco de dados atualizado de forma segura, sem apagar dados, e permitem que a equipe tenha controle das mudanças no banco com histórico.

Por que sequelize.sync({ force: true }) é perigoso em produção?

Esse comando é usado para sincronizar os models com o banco de dados. Quando você faz force: true, ele apaga todas as tabelas existentes e recria tudo do zero com base nos seus models. Esse comando é útil só durante o desenvolvimento, quando você ainda está testando o sistema e os dados não são importantes, depois ele se torna perigoso, já que refazer as tabelas significa apagar o dados também.

Fluxo de trabalho básico de uma migration

1. Criando a migration:
   npx sequelize-cli migration:generate --name criar-tabela-usuarios
   ↓
2. Modificando:
   ```
   module.exports = {
     up: async (queryInterface, Sequelize) => {
       await queryInterface.createTable('Usuarios', {
         id: {
           type: Sequelize.INTEGER,
           primaryKey: true,
           autoIncrement: true,
           allowNull: false
         },
         nome: {
           type: Sequelize.STRING,
           allowNull: false
   ```

```
    },
    email: {
      type: Sequelize.STRING,
      allowNull: false
    }
  });
 },

 down: async (queryInterface, Sequelize) => {
   await queryInterface.dropTable('Usuarios');
 }
};
```

↓

3. Executando a migration para a tabela ser criada:
   npx sequelize-cli db:migrate
   ↓
4. Revertendo (usando o down)
   npx sequelize-cli db:migrate:undo

O que é uma transação em um banco de dados e por que ela é importante para garantir a integridade dos dados?

Uma transação é um conjunto de operações no banco de dados que são tratadas como uma única ação. Ou seja, ou todas as operações dentro da transação são concluídas com sucesso ou nenhuma delas é aplicada. Isso garante a atomicidade, que evita situações em que só parte dos dados é atualizada, o que poderia deixar o banco inconsistente.

Como o Sequelize permite executar múltiplas operações dentro de uma única transação?

O Sequelize oferece uma API para criar e usar transações. Você inicia uma transação, executa várias operações dentro dela, e só confirma (commit) se todas derem certo. Se algum erro ocorrer, você pode desfazer tudo (rollback).

Exemplo conceitual:

```
const { Sequelize, sequelize, Usuario } = require('./models');

async function atualizarDados() {
const t = await sequelize.transaction();
    try {
        await Usuario.update({ nome: 'João' }, { where: { id: 1 }, transaction: t
    });
        await Usuario.create({ nome: 'Maria', email: 'maria@email.com' }, {
    transaction: t });
        await t.commit();
      } catch (error) {
         await t.rollback();
      }
    }
```

Análise Crítica e Comparativa

Vantagens e Desvantagens de Usar um ORM

Usar um ORM como o Sequelize traz vantagens na produtividade, porque permite que a gente trabalhe com objetos em vez de escrever comandos SQL diretamente, deixando o desenvolvimento mais rápido e com menos chances de erro. Outra vantagem é a portabilidade, já que o Sequelize suporta diferentes tipos de banco de dados, como MySQL, PostgreSQL e SQLite, o que facilita trocar o banco sem precisar refazer tudo. Além disso, o ORM facilita o gerenciamento de dados complexos, principalmente quando se trata de relacionamentos entre tabelas, porque ele oferece métodos prontos para lidar com essas associações sem precisar escrever joins manuais.

Por outro lado, uma das principais desvantagens comentadas sobre o uso de ORM é que em sistemas maiores o processo de busca de dados dele começa a ficar lento.

Quando NÃO usar um ORM?

Quando a aplicação precisa de um desempenho muito alto e faz consultas complexas ou muito específicas, onde o controle direto do SQL é importante para otimizar cada detalhe. Nesses casos, usar SQL puro ou um Query Builder pode ser melhor, porque eles dão mais liberdade para escrever consultas eficientes e específicas, sem a camada extra de abstração que o ORM traz.

Comparativo: Sequelize vs. Knex.js

Critério	Sequelize	Knex.js
Tipo de ferramenta	ORM	
Linguagem principal	JavaScript	JavaScript
Forma de definir o schema	Via código (models)	Via código (métodos encadeados)
Facilidade de uso	Mais fácil para quem prefere trabalhar com objetos e abstração de SQL	Exige mais conhecimento de SQL, mas oferece mais controle sobre as consultas