INSTITUTO FEDERAL CATARINENSE – CAMPUS SOMBRIO
TÉCNICO EM INFORMÁTICA PARA A INTERNET

ISADORA SOFIA MARCELINO PEREIRA

TRABALHO DE DESENVOLVIMENTO WEB III

SOMBRIO

2025

Sumário

1. Introdução

O desenvolvimento de aplicações que dependem de bancos de dados relacionais é uma tarefa frequente na área de tecnologia. Tradicionalmente, a interação com esses bancos exige a escrita de consultas SQL, que pode ser repetitiva e propensa a erros. Para tornar esse processo mais natural e produtivo, surgiram ferramentas conhecidas como ORMs (Mapeamento Objeto-Relacional), que fazem a ponte entre objetos da aplicação e tabelas do banco.

Este trabalho explora o conceito de ORM, com foco no Sequelize, um ORM popular no ambiente Node.js. O objetivo é compreender seus fundamentos, funcionamento, principais características e melhores práticas, além de analisar criticamente seu uso comparado a outras ferramentas.

2. Fundamentos dos ORMs

2.1 O que é um ORM (Mapeamento Objeto-Relacional)

Um ORM é uma biblioteca ou framework que permite que desenvolvedores manipulem bancos de dados relacionais usando a linguagem orientada a objetos da aplicação, sem precisar escrever SQL diretamente. Isso facilita o desenvolvimento, pois transforma dados armazenados em tabelas em objetos que podem ser usados diretamente no código.

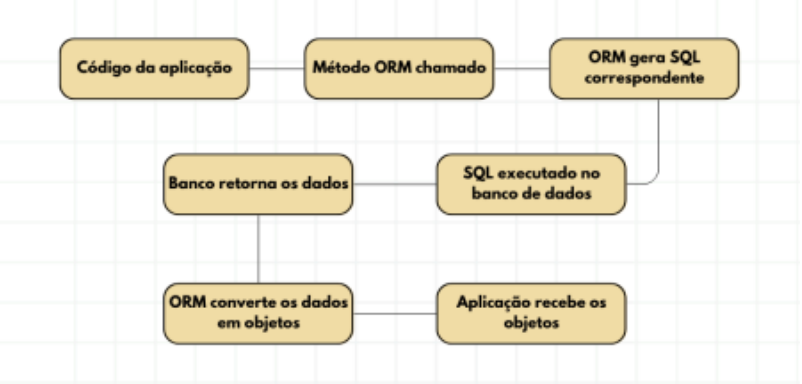
2.1.1. Paradigma da Impedância Objeto-Relacional

O paradigma da impedância objeto-relacional é a dificuldade natural de casar dois modelos distintos: o orientado a objetos (com herança, encapsulamento e relacionamentos por referências) e o relacional (baseado em tabelas, colunas e linhas). Por exemplo, enquanto um objeto pode conter listas de outros objetos, um banco relacional representa isso com tabelas relacionadas e chaves estrangeiras.

Essa diferença gera complexidade ao tentar mapear operações e estruturas entre o código e o banco, o que os ORMs procuram resolver abstraindo essa comunicação.

2.2 Como um ORM funciona em alto nível

O funcionamento de um ORM pode ser visualizado no fluxograma a seguir:



Dessa forma, o ORM atua como uma camada intermediária entre a aplicação e o banco de dados, simplificando o acesso aos dados, gerenciando conexões, gerando comandos SQL automaticamente e convertendo os resultados em objetos manipuláveis no código.

3. Sequelize em Detalhes

3.1 O que é o Sequelize

O Sequelize é um ORM (Object-Relational Mapping) para aplicações Node.js, que permite a integração com bancos de dados relacionais amplamente utilizados, como MySQL, PostgreSQL, SQLite, MariaDB e Microsoft SQL Server.

Sua principal proposta é facilitar operações de criação, leitura, atualização e exclusão de dados (CRUD) por meio de uma API baseada em JavaScript, abstraindo a necessidade de escrever comandos SQL diretamente.

Entre seus principais diferenciais, destacam-se:

- Popularidade e uma comunidade ativa de desenvolvedores.
- Suporte a múltiplos "dialetos" de bancos de dados.
- Sintaxe simplificada para definição de tabelas e estruturas, por meio de *models*.
- Capacidade de definir e manipular associações complexas entre entidades relacionais.

3.2 Models e Associações: O Coração do Sequelize

Um *Model* no Sequelize representa uma tabela no banco de dados, sendo que cada instância do model corresponde a uma linha (registro) dessa tabela. A estrutura do model define os campos (colunas) e suas propriedades, como tipo de dado, chaves primárias, obrigatoriedade, entre outros.

```
// Importa as classes Sequelize e DataTypes da biblioteca sequelize
const { Sequelize, DataTypes } = require('sequelize');

// Instancia um novo objeto Sequelize, conectando a um banco de dados SQLite em memória
// Essa configuração é utilizada para fins de teste, pois o banco de dados é temporário e não persiste dados
const sequelize = new Sequelize('sqlite::memory:');

// Define o model 'Produto', que corresponde à tabela 'Produtos' no banco de dados relacional
// Cada propriedade do objeto passado à função define uma coluna da tabela, incluindo seu tipo e restrições
const Produto = sequelize.define('Produto', {
  // Campo 'id' do tipo inteiro, configurado como chave primária e com auto incremento
  // Isso garante que cada registro tenha um identificador único gerado automaticamente
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  // Campo 'nome' do tipo string, definido como obrigatório (não aceita valores nulos)
  nome: {
    type: DataTypes.STRING,
    allowNull: false
  },
  // Campo 'preco' do tipo decimal com precisão de até 10 dígitos e 2 casas decimais, também obrigatório
  preco: {
    type: DataTypes.DECIMAL(10, 2),
    allowNull: false
  }
});

// Exporta o model 'Produto' para que possa ser utilizado em outras partes da aplicação
module.exports = Produto;
```

3.2.1. Tipos de Associações no Sequelize

- hasOne (Um-para-Um)

O método *hasOne* estabelece uma associação do tipo um-para-um entre dois models, indicando que uma instância de um model possui exatamente uma instância relacionada em outro model.

Exemplo: *Usuario.hasOne(Endereco);*

- hasMany (Um-para-Muitos)

O método *hasMany* estabelece uma associação do tipo um-para-muitos entre dois models, indicando que uma instância de um model pode estar associada a múltiplas instâncias de outro model.
Exemplo: *Usuario.hasMany(Post);*

- belongsTo (Pertence a)

Define que um registro pertence a outro, representando geralmente o lado “muitos” em uma relação um-para-muitos.

Exemplo: *Post.belongsTo(Usuario);*

- belongsToMany (Muitos-para-Muitos)

Usado quando múltiplos registros de uma tabela estão relacionados a múltiplos registros de outra, sendo necessária uma tabela intermediária para representar essa relação.

Exemplo: *Produto.belongsToMany(Pedido, { through: 'PedidoProduto' });*

Exemplo: *Pedido.belongsToMany(Produto, { through: 'PedidoProduto' });*

3.3 Operações comuns em SQL e Sequelize

	SQL	Sequelize
Buscar todos os registros	SELECT * FROM Produtos;	Produto.findAll();
Buscar registro por ID	SELECT * FROM Produtos WHERE id = 1;	Produto.findByPk(1);
Buscar com condição	SELECT * FROM Produtos WHERE preco > 100;	Produto.findAll({ where: { preco: { [Op.gt]: 100 } } });
Realizar JOIN com tabela relacionada	SELECT Usuarios.nome, Enderecos.rua FROM Usuarios JOIN Enderecos ON Usuarios.id = Enderecos.usuarioId;	Usuario.findAll({ include: Endereco });

4. Tópicos Avançados e Boas Práticas

4.1 Migrations: Gerenciamento da Evolução do Banco de Dados

O método sequelize.sync({ force: true }) recria as tabelas do banco de dados, apagando todos os dados existentes, o que o torna perigoso para ambientes de produção. Para gerenciar as alterações no esquema do banco de forma controlada e incremental, utilizamos as **migrations**, que versionam e aplicam modificações de maneira organizada.

O fluxo básico de trabalho com migrations consiste em:

- Criar a migration, definindo a alteração a ser realizada no banco de dados;
- Aplicar a migration (up), executando as mudanças;
- Reverter a migration (down), desfazendo as alterações caso necessário.

4.2 Transações (Transactions)

Transações garantem que um conjunto de operações no banco de dados seja executado de forma atômica, ou seja, todas as operações são concluídas com sucesso ou nenhuma delas é aplicada, preservando a integridade dos dados.

Exemplo de uso de transação no Sequelize:

```
// Importa a instância do Sequelize e o model Produto do arquivo de models
const { sequelize, Produto } = require('./models');

async function exemploTransacao() {
  // Inicia uma nova transação
  const t = await sequelize.transaction();

  try {
    // Cria um novo produto "Caneta" dentro da transação
    await Produto.create({ nome: 'Caneta', preco: 2.5 }, { transaction: t });
    // Cria um novo produto "Caderno" dentro da transação
    await Produto.create({ nome: 'Caderno', preco: 15.0 }, { transaction: t });

    // Se todas as operações forem bem-sucedidas, confirma a transação
    await t.commit();
  } catch (error) {
    // Caso ocorra algum erro, desfaz todas as operações da transação
    await t.rollback();
    // Registra o erro ocorrido durante a transação
    console.error('Erro na transação:', error);
  }
}
```

5. Análise Crítica e Comparativa

5.1 Vantagens e Desvantagens de Usar um ORM

Vantagens:

- **Produtividade:** Evita a necessidade de escrever SQL manualmente, acelerando o desenvolvimento.
- **Portabilidade:** Facilita a troca entre diferentes bancos de dados, uma vez que o ORM abstrai os dialetos específicos.
- **Manutenção:** Promove um código mais limpo e organizado por meio do uso de models e associações.

Desvantagens:

- **Curva de aprendizado:** Compreender as abstrações do ORM pode ser complexo no início.
- **Performance:** Em alguns casos, as consultas geradas podem ser menos otimizadas do que consultas SQL escritas manualmente.
- **Abstração pode "vazar":** Algumas funcionalidades específicas do banco podem não ser suportadas pelo ORM, exigindo a escrita de SQL puro em determinados cenários.

5.2 Quando NÃO usar um ORM

Cenários em que o uso de ORM pode não ser recomendado:

- Aplicações que realizam consultas complexas e que demandam controle total sobre o SQL gerado.
- Contextos onde a performance é um requisito crítico e otimizações específicas no banco de dados são necessárias.
- Equipes que possuem preferência ou expertise para trabalhar diretamente com SQL ou com Query Builders, valorizando maior controle sobre as consultas.

5.3 Comparativo: Sequelize vs. Knex.js

Sequelize Knex.js

Tipo de ferramenta ORM Query Builder	JavaScript
Linguagem JavaScript **Definição de**	Não gerencia schema
schema Via código (models)	Mais flexível, requer SQL
Facilidade de uso Mais alto nível, abstrato	

6. Conclusão

O Sequelize é uma ferramenta robusta e eficiente para o desenvolvimento de aplicações Node.js que utilizam bancos de dados relacionais. Ele oferece uma camada de abstração que simplifica o manuseio dos dados e contribui para a organização e manutenção do código. Contudo, é essencial compreender suas limitações e avaliar cuidadosamente o contexto de aplicação, a fim de evitar problemas relacionados à performance ou à complexidade desnecessária.

7. Referências Bibliográficas

Alura. Node.js: Definição, Características, Vantagens e Usos. Disponível em: https://www.alura.com.br/artigos/

Alura. Transações no SQL: Mantendo os Dados Íntegros. Disponível em: https://www.alura.com.br/artigos/

Boson Treinamentos. O que é uma Transação em Banco de Dados?. Disponível em: https://www.bosontreinamentos.com.br/bancos-de-dados/

Casa do Desenvolvedor. Mapeamento Objeto-Relacional. Disponível em: https://blog.casadodesenvolvedor.com.br/

Databricks. Definição de Transações ACID. Disponível em: https://www.databricks.com/br/glossary/

Dev.to. Sequelize or TypeORM: Pros and Cons. Disponível em: https://dev.to/

DevMedia. ORM - Object-Relational Mapper. Disponível em: https://www.devmedia.com.br/

FreeCodeCamp (EN). What is an ORM: The Meaning of Object-Relational Mapping Database Tools. Disponível em: https://www.freecodecamp.org/news/

FreeCodeCamp (PT). O que é um ORM: O Significado das Ferramentas de Mapeamento Relacional de Objetos de Banco de Dados. Disponível em: https://www.freecodecamp.org/portuguese/news/

GeekHunter. Mapeamento Objeto-Relacional. Disponível em: https://blog.geekhunter.com.br/

iMasters. Tutorial de Migrations com Node.js e Sequelize. Disponível em: https://imasters.com.br/banco-de-dados/

LuizTools. Tutorial de Migrations com Node.js e Sequelize. Disponível em: https://www.luiztools.com.br/post/

Medium. Object-Relational Impedance Mismatch. Disponível em: https://medium.com/

Medium. PG Driver vs Knex.js vs Sequelize vs TypeORM. Disponível em: https://medium.com/

Medium. Sequelize vs TypeORM: Pros, Cons, and Use Cases. Disponível em: https://medium.com/

Medium. Sequelize vs TypeORM vs Prisma: Choosing the Right ORM for Your Node.js Project. Disponível em: https://medium.com/

Micilini. O que são ORMs e Quais as Vantagens de Utilizar em Seus Projetos. Disponível em: https://micilini.com/blog/

Prisma. Comparações entre ORM: Prisma vs Sequelize. Disponível em: https://www.prisma.io/docs/orm/

Sequelize. Documentação Oficial do Sequelize. Disponível em: https://sequelize.org/

Sequelize Docs. API de Dialects do Sequelize. Disponível em: https://sequelize.org/api/v7/

Sequelize Docs. Migrations com Sequelize. Disponível em: https://sequelize.org/docs/v6/

StackShare. Comparação entre Knex.js e Sequelize. Disponível em: https://stackshare.io/

TreinaWeb. O que é ORM. Disponível em: https://www.treinaweb.com.br/blog/

TreinaWeb. Usando Sequelize ORM com Node e Express. Disponível em: https://www.treinaweb.com.br/blog/