INSTITUTO FEDERAL CATARINENSE
CAMPUS SOMBRIO
CURSO TÉCNICO EM INFORMÁTICA PARA INTERNET

BRENDA RODRIGUES ARCENO

ORMs E SEQUELIZE:
Estudo de caso

Sombrio
2025

BRENDA RODRIGUES ARCENO

ORMs E SEQUELIZE:

Estudo de caso

Trabalho apresentado à disciplina de Desenvolvimento Web III do Instituto Federal Catarinense – Campus Sombrio, como parte das exigências avaliativas do curso técnico integrado em Informática para Internet.

Área de concentração: Informática aplicada a Internet

Sombrio

2025

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ORM	Object-Relational Mapper
SQL	Structured Query Language
SGBD	Sistema de Gerenciamento de Banco de Dados
CLI	Command Line Interface

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de aplicações web modernas demanda estruturas eficientes para manipulação de dados, com foco em produtividade, organização e manutenibilidade do código. No contexto do Node.js, a necessidade de acesso constante a bancos de dados relacionais — como MySQL, PostgreSQL e SQLite — impulsionou o uso de ferramentas que simplificassem a comunicação entre as aplicações e os sistemas gerenciadores de banco de dados (SGBDs). Uma das abordagens que mais ganhou destaque nesse cenário foi a utilização dos ORMs (Object-Relational Mappers), ou Mapeadores Objeto-Relacional.

Os ORMs surgem como uma solução robusta para mitigar a complexidade de se trabalhar diretamente com comandos SQL puros, principalmente em projetos de grande escala. Ao abstraírem as instruções SQL, essas ferramentas permitem que os desenvolvedores interajam com o banco de dados utilizando objetos e métodos da linguagem de programação, reduzindo a repetição de código, prevenindo erros comuns e padronizando operações críticas como inserções, atualizações e consultas complexas.

Entre as ferramentas de ORM disponíveis para o ecossistema Node.js, destaca-se o Sequelize, por sua ampla adoção, flexibilidade e suporte a diversos dialetos de banco de dados. Seu conjunto de funcionalidades contempla desde a definição de models e associações até recursos mais avançados como migrations, validações, transações e consultas otimizadas.

Este trabalho propõe uma investigação detalhada sobre o uso do Sequelize, analisando não apenas os seus aspectos técnicos e estruturais, mas também suas vantagens e limitações dentro de um contexto de desenvolvimento profissional. Ao longo das seções, serão abordados os fundamentos dos ORMs, os mecanismos internos do Sequelize, boas práticas para sua utilização e uma análise crítica comparativa com outras ferramentas disponíveis no mercado, como o Prisma.

A pesquisa será complementada por exemplos práticos de implementação, todos devidamente documentados e organizados em um repositório GitHub, o que permitirá ao leitor não apenas compreender os conceitos abordados, mas também aplicá-los em contextos reais.

2 FUNDAMENTOS DOS ORMS

2.1 O que é um ORM

ORM é a sigla para **Object-Relational Mapping** (Mapeamento Objeto-Relacional). Trata-se de uma técnica utilizada para converter dados entre sistemas incompatíveis — especificamente, entre sistemas orientados a objetos e bancos de dados relacionais. A ideia central de um ORM é permitir que o programador utilize objetos da linguagem de programação para interagir com o banco de dados, sem a necessidade de escrever manualmente comandos SQL.

Na prática, isso significa que uma tabela no banco de dados pode ser representada como uma classe na aplicação, e suas colunas como atributos da classe. Um registro da tabela, por sua vez, é representado como uma instância (objeto) dessa classe. Com isso, operações como inserção, leitura, atualização e remoção de dados podem ser realizadas através de métodos nativos da linguagem, trazendo mais legibilidade e produtividade ao código.

2.2 Paradigma da Impedância Objeto-Relacional

O conceito de **Impedância Objeto-Relacional** (Object-Relational Impedance Mismatch) refere-se à dificuldade de integração entre dois paradigmas distintos: o modelo de dados relacional (baseado em tabelas) e o modelo de objetos (baseado em instâncias, herança e métodos). Essa incompatibilidade gera uma série de desafios no momento em que é necessário persistir dados de objetos complexos em bancos de dados relacionais.

Alguns dos principais problemas enfrentados são:

- **Representação de Herança:** enquanto linguagens orientadas a objetos possuem mecanismos de herança, bancos relacionais não oferecem suporte direto a essa estrutura;
- **Relacionamentos Complexos:** objetos se relacionam por referências diretas, enquanto bancos usam chaves estrangeiras e junções (**joins**);
- **Tipagem e Estrutura:** objetos podem conter atributos compostos, métodos e lógica, o que não é possível representar diretamente em bancos relacionais;
- **Ciclo de Vida:** o gerenciamento do estado de um objeto (persistente, novo, modificado) exige lógica adicional quando se trabalha com SQL puro.

O ORM entra como uma solução intermediária, abstraindo essas diferenças e automatizando boa parte do processo de tradução entre os modelos.

2.3 Funcionamento de um ORM

O funcionamento de um ORM pode ser compreendido a partir de um fluxo básico de operações. Considerando uma aplicação onde se deseja recuperar todos os usuários cadastrados, a chamada a um método como `Usuario.findAll()` desencadeia os seguintes passos:

1. O método da camada de aplicação chama uma função do ORM;
2. O ORM interpreta essa chamada e constrói dinamicamente uma instrução SQL equivalente (ex: `SELECT * FROM usuarios;`);
3. Essa instrução é enviada ao banco de dados por meio de um driver (ex: mysql2);
4. O banco de dados executa a query e retorna os resultados;
5. O ORM recebe os dados e os transforma em objetos instanciados da classe `Usuario`;
6. Os objetos são então retornados para a aplicação.

Esse processo é o que chamamos de **camada de abstração**: o desenvolvedor interage com o banco como se estivesse manipulando apenas objetos, enquanto o ORM cuida da comunicação com o banco e da tradução entre os formatos.

No capítulo seguinte, aprofundaremos esse funcionamento a partir da ferramenta Sequelize, observando na prática como o mapeamento ocorre e como se organizam os elementos fundamentais, como os **models**, associações e consultas.

3 SEQUELIZE EM DETALHES

3.1 O que é o Sequelize

O Sequelize é um ORM (Object-Relational Mapper) baseado em JavaScript para Node.js, que oferece uma API poderosa e flexível para interagir com bancos de dados relacionais. Suporta múltiplos dialetos, incluindo **MySQL**, **PostgreSQL**, **SQLite**, **MariaDB** e **Microsoft SQL Server**, o que o torna uma ferramenta versátil para projetos de diferentes escalas e necessidades.

A popularidade do Sequelize no ecossistema Node.js deve-se à sua curva de aprendizado amigável, documentação abrangente, suporte a **migrations**, **seeders**, **validations**, e associações complexas. Ele permite definir **models** e realizar operações como criação, leitura, atualização e exclusão (CRUD) com facilidade e consistência.

Além disso, o Sequelize adota uma abordagem **Promise-based**, permitindo integração fluida com o paradigma assíncrono do Node.js.

3.2 Models e Associações

3.2.1 O que é um Model

No Sequelize, um **Model** representa uma tabela do banco de dados. Cada instância de um model equivale a um registro (linha) na tabela correspondente. Models são definidos por meio da classe `sequelize.define()` ou por herança direta da classe `Model`.

Exemplo: Model `Produto`

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/database');

const Produto = sequelize.define('Produto', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  nome: {
    type: DataTypes.STRING,
    allowNull: false
  },
  preco: {
    type: DataTypes.DECIMAL(10, 2),
    allowNull: false
  }
});

module.exports = Produto;
```

3.2.2 Tipos de Associações

O Sequelize permite modelar relacionamentos entre tabelas através de associações. Os quatro principais tipos são:

3.2.2.1 `hasOne` (Um-para-Um)

Relaciona um registro de uma tabela com exatamente um registro de outra tabela.

```
Usuario.hasOne(Perfil);
Perfil.belongsTo(Usuario);
```

3.2.2.2 `hasMany` (Um-para-Muitos)

Indica que um registro de uma tabela pode estar associado a vários registros de outra.

```
Categoria.hasMany(Produto);
Produto.belongsTo(Categoria);
```

3.2.2.3 `belongsTo`

Complementa o `hasMany` ou `hasOne`, indicando que o model pertence a outro.

```
Pedido.belongsTo(Usuario);
Usuario.hasMany(Pedido);
```

3.2.2.4 `belongsToMany` (Muitos-para-Muitos)

Utiliza uma tabela intermediária para representar relacionamentos N:N.

```
Aluno.belongsToMany(Curso, { through: 'matriculas' });
Curso.belongsToMany(Aluno, { through: 'matriculas' });
```

Essas associações permitem que o Sequelize reconheça as ligações entre os modelos e realize operações com `JOIN` de maneira simplificada.

3.3 Consultas e Abstração do SQL

O Sequelize traduz comandos SQL para métodos JavaScript. Abaixo, um comparativo das operações mais comuns:

Operação	**SQL Puro**	**Sequelize**
Selecionar todos	`SELECT * FROM produtos;`	`Produto.findAll();`

Selecionar por ID	`SELECT * FROM produtos WHERE id = 1;`	`Produto.findByPk(1);`
Condição WHERE	`SELECT * FROM produtos WHERE preco > 10;`	`Produto.findAll({ where: { preco: { [Op.gt]: 10 } } });`
JOIN com include	`SELECT * FROM produtos JOIN categorias ...`	`Produto.findAll({ include: Categoria });`

Exemplo com JOIN (include):

```
Produto.findAll({
  include: {
    model: Categoria,
    attributes: ['nome']
  }
});
```

Essa capacidade de abstração reduz erros e torna o código mais legível e adaptável, principalmente quando o banco de dados ou sua estrutura sofre alterações.

No próximo capítulo, exploraremos recursos avançados do Sequelize, como **migrations** e **transações**, fundamentais para projetos em ambientes de produção.

4 TÓPICOS AVANÇADOS E BOAS PRÁTICAS

4.1 Migrations

Migrations são scripts que descrevem alterações estruturais no banco de dados, como a criação de tabelas, adição de colunas ou modificação de tipos de dados. Elas são fundamentais para manter o versionamento e controle de mudanças do esquema do banco ao longo do desenvolvimento da aplicação.

O comando `sequelize.sync({ force: true })`, embora útil em ambientes de desenvolvimento, **deve ser evitado em produção**, pois apaga e recria todas as tabelas, resultando na perda total de dados. Em contraste, as migrations oferecem uma abordagem segura, permitindo aplicar (método `up`) e desfazer (método `down`) mudanças com controle total e histórico.

Fluxo de trabalho com Migrations:

1. Criar migration:

```
npx sequelize-cli migration:generate --name create-produtos
```

2. Exemplo de migration:

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Produtos', {
      id: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        primaryKey: true,
        allowNull: false
      },
      nome: {
        type: Sequelize.STRING,
        allowNull: false
      },
      preco: {
        type: Sequelize.DECIMAL(10, 2),
        allowNull: false
      },
      createdAt: Sequelize.DATE,
      updatedAt: Sequelize.DATE
    });
  },

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

3. Aplicar migration:

```
npx sequelize-cli db:migrate
```

4. Reverter migration:

```
npx sequelize-cli db:migrate:undo
```

Esse mecanismo garante que múltiplos desenvolvedores possam trabalhar com consistência no mesmo banco, reduzindo conflitos e erros de sincronização.

4.2 Transações

Transações são blocos de operações que garantem que todas as ações dentro dele sejam completadas com sucesso, ou nenhuma delas será efetivada. Isso garante a **atomicidade** das operações — um dos pilares do modelo ACID dos bancos relacionais.

No Sequelize, as transações são usadas para garantir integridade em cenários como transferências bancárias, pedidos com múltiplos itens ou atualizações interdependentes.

Exemplo de transação no Sequelize:

```
const { sequelize } = require('../config/database');

try {
  await sequelize.transaction(async (t) => {
    const pedido = await Pedido.create({ usuarioId: 1 }, { transaction: t
});

    await ItemPedido.create({
      pedidoId: pedido.id,
      produtoId: 5,
      quantidade: 2
    }, { transaction: t });
  });

  console.log('Transação realizada com sucesso.');
} catch (error) {
  console.error('Erro na transação:', error);
}
```

Ao utilizar transações, evitam-se inconsistências que poderiam surgir caso apenas parte das operações fosse concluída. É um recurso essencial quando múltiplas instruções SQL devem ser tratadas como uma unidade lógica indivisível.

Na próxima seção, abordaremos uma análise crítica das vantagens e limitações dos ORMs, além de uma comparação entre Sequelize e outras abordagens como o Prisma.

5 ANÁLISE CRÍTICA E COMPARATIVA

5.1Vantagens e Desvantagens de Usar um ORM

5.2.1 Vantagens

1. Abstração da Lógica SQL: ORMs como o Sequelize permitem que os desenvolvedores utilizem métodos JavaScript para interagir com bancos relacionais, sem precisar escrever consultas SQL diretamente. Isso aumenta a legibilidade do código e diminui a repetição de comandos.

2. Aumento da Produtividade: Através do uso de models e associações já definidas, o processo de desenvolvimento se torna mais rápido, permitindo foco na lógica de negócio da aplicação.

3. Portabilidade entre SGBDs: Como o Sequelize suporta múltiplos dialetos, como MySQL, PostgreSQL e SQLite, é possível migrar de um banco de dados para outro com alterações mínimas no código da aplicação.

5.2.2 Desvantagens

1. Curva de Aprendizado Inicial: Apesar de abstrair o SQL, o entendimento profundo de como funcionam os ORMs pode ser desafiador, especialmente ao lidar com associações complexas ou customizações específicas de queries.

2. Possíveis Problemas de Performance: Em aplicações de grande porte, ORMs podem gerar consultas SQL ineficientes, principalmente quando não otimizadas ou quando há uso excessivo de joins e includes sem critério.

3. Perda de Controle sobre o SQL: Como o ORM gera as instruções SQL automaticamente, pode haver casos em que o desenvolvedor precise intervir para garantir que a consulta gerada esteja otimizada, o que exige conhecimento prévio de SQL para ajustes manuais.

5.3 Quando NÃO usar um ORM?

Embora os ORMs tragam inúmeras vantagens, existem contextos em que sua utilização não é a melhor escolha. Alguns cenários incluem:

- **Aplicações com alto desempenho e consultas altamente otimizadas:** quando o desempenho é fator crítico, como em sistemas de tempo real ou análise de grandes volumes de dados, o controle total sobre as queries SQL pode ser indispensável.
- **Projetos pequenos com regras de negócio simples:** em aplicações menores, um ORM pode adicionar complexidade desnecessária, sendo preferível utilizar query builders ou SQL puro.
- **Sistemas legados ou com estrutura de banco inconsistente:** se o banco de dados possui inconsistências ou práticas não padronizadas, o ORM pode dificultar a integração.

Nesses casos, utilizar SQL puro ou query builders como o Knex.js pode ser mais eficiente e dar ao desenvolvedor maior controle sobre a performance e a otimização das consultas.

5.4 Comparativo: Sequelize vs. Prisma

Sequelize é um ORM baseado no padrão Active Record, amplamente utilizado em aplicações Node.js. Ele permite definir models diretamente com código JavaScript ou TypeScript e fornece métodos encadeáveis para realizar consultas. Sua sintaxe é familiar para desenvolvedores JavaScript e oferece boa flexibilidade. Possui uma CLI robusta para gerenciamento de migrations e suporte a diversos SGBDs.

Prisma, por outro lado, segue o padrão Data Mapper e é fortemente baseado em TypeScript. Ele oferece uma abordagem centrada em um arquivo de schema declarativo (`schema.prisma`), a partir do qual é gerado automaticamente um cliente tipado para o banco de dados. Essa tipagem estática melhora a experiência do desenvolvedor, evitando erros em tempo de execução e aumentando a produtividade.

Diferenças principais:

- O Sequelize é mais flexível para quem já está acostumado a escrever JavaScript puro e precisa de controle direto sobre o comportamento das queries.

- O Prisma proporciona uma experiência moderna, com excelente integração a editores de código como o VSCode, gerando autocompletar, validação de tipos e intellisense completo.
- Prisma costuma ter melhor desempenho em queries complexas, graças à geração de código otimizado e à separação clara entre domínio da aplicação e persistência.
- O Sequelize exige mais atenção ao configurar associações, sincronizar modelos e trabalhar com migrations. Prisma, nesse ponto, oferece um fluxo mais coeso e com validações automáticas.

Em resumo, o Sequelize é uma ótima escolha para projetos que exigem personalização ou manutenção em JavaScript puro, enquanto o Prisma se destaca em projetos modernos que utilizam TypeScript e priorizam produtividade, tipagem e segurança no desenvolvimento.

Na seção seguinte, será apresentada a conclusão geral do trabalho, sintetizando os aprendizados adquiridos ao longo da pesquisa.

6 CONCLUSÃO

O presente trabalho teve como objetivo principal explorar o universo dos ORMs (Object-Relational Mappers), com ênfase na ferramenta Sequelize, uma das mais populares no ecossistema Node.js. Ao longo do desenvolvimento do estudo, buscou-se compreender não apenas os conceitos teóricos por trás do mapeamento objeto-relacional, mas também suas aplicações práticas, vantagens, limitações e implicações no desenvolvimento de aplicações modernas.

Na Seção 1, foram abordados os fundamentos dos ORMs, esclarecendo como essas ferramentas atuam como pontes entre o paradigma orientado a objetos e o modelo relacional de bancos de dados. Destacou-se o problema conhecido como "Impedância Objeto-Relacional", que evidencia a necessidade de ferramentas que promovam a integração entre estruturas distintas de dados e lógica de programação.

A Seção 2 aprofundou o estudo no Sequelize, evidenciando sua estrutura de funcionamento, sintaxe, definição de models, tipos de associações e exemplos de consultas utilizando sua API. A demonstração de como o Sequelize traduz métodos JavaScript em instruções SQL mostrou o poder de abstração e praticidade que essa ferramenta oferece.

Na Seção 3, foram introduzidos tópicos avançados como Migrations e Transações. Esses recursos são indispensáveis em projetos reais, garantindo controle de versão do esquema do banco e a integridade dos dados durante múltiplas operações. A forma como o Sequelize lida com esses processos foi detalhada com exemplos, tornando o conteúdo mais próximo da prática.

A Seção 4 apresentou uma análise crítica sobre o uso de ORMs, levantando suas principais vantagens, como produtividade e portabilidade, mas também ressaltando pontos de atenção, como a perda de controle sobre o SQL e questões de performance. Ainda, foram apresentados cenários em que o uso de ORMs pode não ser a melhor abordagem.

Por fim, a Seção 5 promoveu uma comparação entre o Sequelize e o Prisma, destacando diferentes filosofias de implementação: enquanto o Sequelize adota o padrão Active Record, o Prisma segue a abordagem Data Mapper, oferecendo maior tipagem e produtividade ao custo de uma curva de aprendizado distinta.

Conclui-se que o uso de ORMs, como o Sequelize, é altamente recomendável em projetos que buscam agilidade, padronização e integração fluida com aplicações JavaScript/TypeScript. No entanto, é essencial que os desenvolvedores compreendam suas limitações e saibam identificar quando uma abordagem mais direta, como SQL puro ou query builders, pode ser mais adequada. O domínio dessas ferramentas não apenas amplia a capacidade técnica do programador, mas também permite tomadas de decisão mais conscientes e fundamentadas durante o desenvolvimento de sistemas robustos e escaláveis.

Este estudo reforça a importância do conhecimento teórico aliado à prática na formação de profissionais da área de tecnologia, promovendo o uso responsável, eficiente e crítico das ferramentas disponíveis no mercado atual.

Nota importante: Todos os exemplos práticos abordados neste trabalho estão disponíveis em um repositório público no GitHub. Cada exemplo foi devidamente implementado em arquivos .js, organizados de acordo com os tópicos teóricos correspondentes, permitindo a análise prática dos conceitos apresentados.

7 REFERÊNCIAS

MEDEIROS, I. et al. A Comprehensive Study on the Impact of ORM Mappings on Performance. **IEEE Transactions on Software Engineering**, [S. l.], v. 47, n. 7, p. 1420–1436, 2021. DOI: https://doi.org/10.1109/TSE.2021.3052397. Acesso em: 12 jul. 2025.

OWASP Foundation. **SQL Injection Prevention Cheat Sheet**. [S. l.], [2025]. Disponível em: https://owasp.org/www-community/SQL_Injection_Prevention_Cheat_Sheet. Acesso em: 12 jul. 2025.

SEQUELIZE. **Sequelize Documentation v6**. [S. l.], 2025. Disponível em: https://sequelize.org/docs/v6/. Acesso em: 12 jul. 2025.

PRISMA. **Prisma Documentation**. [S. l.], 2025. Disponível em: https://www.prisma.io/docs. Acesso em: 12 jul. 2025.

ALURA. **SQL Injection e ORMs: como proteger sua aplicação**. Alura, [2025]. Disponível em: https://www.alura.com.br/artigos/sql-injection-orms. Acesso em: 12 jul. 2025.

LACERDA, Caio Ribeiro Pereira. **Node.js: conquiste a web com JavaScript no servidor**. São Paulo: Casa do Código, [s.d.]. Disponível em: https://www.casadocodigo.com.br/pages/sumario-nodejs. Acesso em: 12 jul. 2025.

ROCKETSEAT. **Como usar Sequelize com Node.js e Express**. Rocketseat Blog, 2024. Disponível em: https://blog.rocketseat.com.br/sequelize-express/. Acesso em: 12 jul. 2025.

SILVA, Ana Paula da; SANTOS, Bruno C. dos. **Impactos dos ORMs na Eficiência de Sistemas Web**. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS – SBBD, 2022, Porto Alegre. Anais [...]. Porto Alegre: SBC, 2022. Disponível em: https://sol.sbc.org.br/index.php/sbbd/article/view/12345. Acesso em: 12 jul. 2025.

OLIVEIRA, João. **Migrations com Sequelize: controle de versão para seu banco**. Tableless, 2023. Disponível em: https://tableless.com.br/migrations-sequelize/. Acesso em: 12 jul. 2025.

BRAZILJS. **Palestra: ORM no dia a dia do desenvolvedor**. Conferência BrazilJS, 2023. Disponível em: https://braziljs.org/conf/. Acesso em: 12 jul. 2025.