INSTITUTO FEDERAL CATARINENSE – CAMPUS SOMBRIO

TÉCNICO EM INFORMÁTICA PARA A INTERNET

ISADORA GOMES PINHEIRO

GIULIA GUIZZO BALADÃO SANTOS

TRABALHO DE DESENVOLVIMENTO WEB III

MAPEAMENTO OBJETO-RELACIONAL (ORM)

SOMBRIO
2025

1. INTRODUÇÃO

O presente trabalho tem por objetivo explorar o conceito de Mapeamento Objeto-Relacional (ORM) e sua utilização prática por meio do Sequelize, um ORM amplamente utilizado no ambiente Node.js. Serão abordados aspectos teóricos—como o paradigma da Impedância Objeto-Relacional—funcionamento, vantagens, limitações e comparação com outras ferramentas, garantindo profundidade e rigor acadêmico.

2. FUNDAMENTOS DO ORM

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

O Mapeamento Objeto-Relacional (Object-Relational Mapping, ou ORM) é uma técnica de desenvolvimento de software que estabelece uma ponte entre sistemas orientados a objetos e bancos de dados relacionais. Por meio dessa abordagem, os objetos definidos na aplicação (em linguagens como JavaScript, Java, Python, entre outras) são automaticamente mapeados para tabelas de um banco de dados relacional, e vice-versa.

Em outras palavras, um ORM permite que operações como criação, leitura, atualização e exclusão de registros sejam realizadas diretamente sobre objetos do código, sem que o desenvolvedor precise escrever instruções SQL explicitamente. Assim, torna-se possível interagir com o banco de dados de forma mais intuitiva e segura, promovendo maior produtividade e manutenção mais simples do código-fonte da aplicação.

2.2 O paradigma da impedância do Objeto-relacional

O principal problema que os ORMs se propõem a resolver é conhecido como Impedância Objeto-Relacional (Object-Relational Impedance Mismatch). Este termo designa as divergências conceituais e estruturais entre dois modelos distintos: o modelo orientado a objetos, utilizado nas linguagens de programação modernas, e o modelo relacional, utilizado por bancos de dados.
Alguns dos principais pontos de conflito são:

Modelos de dados distintos: objetos são estruturados com atributos, métodos e relacionamentos via ponteiros; enquanto o modelo relacional baseia-se em tabelas, colunas, linhas e chaves.

Herança vs. Normalização: o paradigma orientado a objetos favorece o uso de herança, enquanto bancos de dados utilizam a normalização, o que dificulta a representação direta das hierarquias de classes.

Identidade de objetos: em programação orientada a objetos, dois objetos podem ser considerados diferentes mesmo que possuam os mesmos valores em seus atributos; no banco de dados relacional, a identidade está ligada a chaves primárias e valores.

Diante dessas incompatibilidades, os ORMs surgem como uma solução intermediária, promovendo a tradução e adaptação entre os dois paradigmas, de forma transparente para o desenvolvedor.

2.3 Funcionamento de um ORM em alto nível

O funcionamento de um ORM pode ser compreendido em camadas, em que cada etapa da execução de um comando na aplicação é traduzida para instruções compreensíveis pelo banco de dados, e os resultados são convertidos novamente em objetos utilizáveis no código.

2.4 Fluxograma Simplificado

```
flowchart TD
    A[Aplicação: Usuario.findAll()] --> B[ORM: traduz requisição]
    B --> C[ORM: gera SQL SELECT * FROM Usuarios]
    C --> D[Banco de Dados: executa SQL]
    D --> E[Resultados em formato de tabela]
    E --> F[ORM: converte registros em objetos Usuario]
    F --> G[Aplicação recebe lista de objetos Usuario]
```

Figura 1 – Fluxo simplificado de uma consulta ORM.

2.5 A Camada de Abstração

A principal função do ORM está em sua camada de abstração, responsável por isolar o desenvolvedor das complexidades da linguagem SQL, do modelo relacional e da especificidade de cada sistema gerenciador de banco de dados (SGBD). Essa camada traduz chamadas de métodos em comandos SQL compatíveis com o SGBD, converte os resultados das consultas em objetos que podem ser manipulados diretamente, garante, sempre que possível, portabilidade entre diferentes bancos de dados (por exemplo, PostgreSQL e MySQL), através de "dialetos", implementa práticas de segurança como prevenção a injeção de SQL, automatiza validações, relacionamentos e persistência de dados com menor acoplamento entre código e banco.

Portanto, a abstração promovida por um ORM não apenas facilita o desenvolvimento, como também contribui para a manutenção, escalabilidade e consistência da aplicação ao longo do tempo.

3. O QUE É SEQUELIZE

O Sequelize é um ORM para Node.js, baseado em Promises/async-await, amplamente utilizado pela comunidade devido à sua robustez, documentação e suporte a múltiplos SGBDs. Ele oferece recursos como definição de modelos (models), associações,

migrations, transações e hooks. Ele suporta os dialetos PostgreSQL, MySQL, MariaDB, SQLite, SQL Server, também possui suporte parcial para Redshift e Snowflake (suporte combinado de documentação e comunidade).

3.1 Models e Associações (explicar o que é model)

Definição de Model `Produto`:

```
const { DataTypes } = require('sequelize');

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 }
});
```

Tipos de Associação:

hasOne (um-para-um)

```
    Usuario.hasOne(Endereco);
```

hasMany (um-para-muitos)

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

belongsTo (muitos-para-um)

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

belongsToMany (muitos-para-muitos)

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

Essas associações permitem que o Sequelize compreenda ligações entre modelos e gere automaticamente consultas SQL com joins incluídos.

3.2 Consultas e Abstração do SQL

Comparativo SQL vs Sequelize

Operação	SQL	Sequelize
Buscar todos	SELECT * FROM Produtos;	Produto.findAll( );
Buscar por ID	SELECT * FROM Produtos WHERE id	Produto.findByPk (1);
Buscar com condição	SELECT * FROM Produtos WHERE preco > 100;	Produto.findAll ({ where: { preco: {[Op.gt]: 100} } });
JOIN com tabela associada	SELECT p.*, u.* FROM Pedido p JOIN Usuario u …;	Pedido.findAll({ include: Usuario });

Esses exemplos demonstram a clareza e concisão da API do Sequelize em substituir blocos de SQL por métodos JavaScript bem definidos.

3.3 Migrations: Gerenciamento da Evolução do Banco de Dados

Perigo do `sequelize.sync({ force: true })`

O método `sequelize.sync({ force: true })` recria todas as tabelas no banco de dados a partir dos modelos definidos no código. Embora útil em ambientes de desenvolvimento e testes, seu uso em produção é extremamente arriscado, pois remove todas as tabelas existentes, eliminando os dados armazenados. Isso pode causar perda irreversível de informações críticas em ambientes reais.

Migrations são scripts versionados e reutilizáveis que descrevem mudanças no esquema do banco de dados (como criação de tabelas, adição de colunas ou remoção de constraints). Elas resolvem o problema de manter a estrutura do banco de dados sincronizada com os modelos da aplicação, sem comprometer os dados existentes.

Fluxo de uma Migration

Criar a migration:

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

Editar o arquivo gerado:

```
up: async (queryInterface, Sequelize) => {
return queryInterface.createTable('Produtos', {
id: { type: Sequelize.INTEGER, autoIncrement: true, primaryKey:
true },
nome: { type: Sequelize.STRING },
```

```
preco: { type: Sequelize.DECIMAL(10,2) }
});
},
down: async (queryInterface, Sequelize) => {
  return queryInterface.dropTable('Produtos');
}
```

Executar a migration:

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

Reverter (rollback):

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

3.4 Transações (Transactions)

Transações são blocos de operações que devem ser executadas integralmente ou não executadas de forma alguma. São essenciais para manter a consistência e integridade dos dados, sobretudo quando múltiplas tabelas ou operações estão envolvidas. Isso está diretamente relacionado ao princípio da atomicidade no modelo ACID de bancos de dados relacionais.

O Sequelize permite encapsular diversas operações em uma única transação:

```
const t = await sequelize.transaction();
try {
    const usuario = await Usuario.create({ nome: 'João' }, {
   transaction: t });
   await Pedido.create({ usuarioId: usuario.id }, { transaction: t
    });
  await t.commit();
} catch (err) {
  await t.rollback();
}
```

Neste exemplo, se qualquer uma das operações falhar, todas as outras são revertidas automaticamente, protegendo os dados de inconsistência.

4. ANÁLISE CRÍTICA E COMPARATIVA

4.1 Vantagens e Desvantagens de Usar um ORM

Como em várias ferramentas, há vantagens e desvantagens ao usar um ORM, nas vantagens estão incluídos aumento de produtividade, o uso de métodos orientados a objetos para manipular dados evita a escrita repetitiva de SQL, acelerando o desenvolvimento; Portabilidade, ORMs como o Sequelize suportam múltiplos bancos de dados (MySQL, PostgreSQL, etc.), permitindo migrações com pouco ou nenhum ajuste de código; E segurança, abstraem práticas seguras, como escaping de valores e prevenção contra injeção de SQL, por padrão.

Já as desvantagens são a curva de aprendizado, o desenvolvedor precisa compreender tanto os conceitos de banco de dados quanto os padrões do ORM (ex: migrations, associations, scopes); Performance, ORMs, por abstraírem o SQL, podem gerar consultas menos otimizadas que um SQL escrito à mão, especialmente em queries complexas; E abstração pode “vazar”, em cenários muito específicos, o desenvolvedor pode precisar escrever SQL nativo, contrariando o propósito da abstração e dificultando a manutenção.

4.2 Quando NÃO usar um ORM

Embora ORMs ofereçam muitos benefícios, existem situações onde seu uso pode ser desvantajoso, como em aplicações com alta exigência de performance, sistemas que processam grandes volumes de dados ou dependem de consultas extremamente otimizadas podem ser prejudicados por consultas genéricas geradas pelo ORM; Consultas muito complexas, relatórios com múltiplos joins, subqueries e funções agregadas complexas tendem a exigir SQL personalizado; E sistemas legados ou bancos já existentes, quando o esquema de dados é fixo e não pode ser alterado para se ajustar ao ORM, usar SQL direto ou query builders é mais adequado. Nesses casos, usar um Query Builder como o Knex.js ou mesmo escrever SQL puro é mais indicado.

4.3 Comparativo: Sequelize vs Prisma

Critério	Sequelize	Prisma
Tipo de ferramenta	ORM tradicional	ORM moderno com foco em DX
Linguagem principal	JavaScript/TypeScript	TypeScript (com suporte parcial a JS)
Definição de schema	Models via código JS/TS	Schema declarativo (schema.prisma)
Facilidade de uso	Exige configuração manual de modelos e associações	Interface mais declarativa, mais fácil para iniciantes
Migrations	Sequelize CLI	Prisma Migrate
Desempenho	Bom, com otimizações manuais	Muito bom, co queries geradas de forma eficiente

Prisma tem se destacado por sua abordagem declarativa e foco na experiência do desenvolvedor (DX), enquanto o Sequelize permanece uma opção sólida e madura, com forte adoção em sistemas já existentes e comunidades JavaScript puras.

5 CONSIDERAÇÕES FINAIS

O presente trabalho apresentou uma análise abrangente do conceito de Mapeamento Objeto-Relacional (ORM), com ênfase na ferramenta Sequelize, destacando sua relevância no ecossistema Node.js. Ao longo do estudo, foram discutidos aspectos teóricos fundamentais, como a impedância objeto-relacional, bem como os mecanismos de funcionamento de um ORM, suas camadas de abstração e o impacto que exercem sobre a produtividade e manutenção de aplicações modernas.

A análise prática do Sequelize demonstrou suas funcionalidades principais, como a definição de modelos e associações, uso de migrations, operações transacionais e abstração de consultas SQL. Também foram discutidas suas vantagens — como segurança, portabilidade e agilidade no desenvolvimento — e limitações, especialmente no que se refere à performance e complexidade de uso em cenários específicos.

Adicionalmente, o trabalho abordou comparativamente outras soluções, como o Prisma, permitindo uma reflexão crítica sobre quando e por que utilizar (ou não) um ORM em determinados projetos. Constatou-se que, embora o uso de ORMs traga inúmeros benefícios, sua adoção deve ser avaliada com cautela, conforme as necessidades técnicas, estruturais e operacionais de cada sistema.

Em síntese, compreende-se que o Sequelize representa uma ferramenta madura e robusta no desenvolvimento backend com JavaScript, sendo especialmente útil em projetos que prezam pela organização, abstração de banco de dados e escalabilidade. Contudo, como qualquer tecnologia, sua escolha deve estar alinhada ao contexto da aplicação, aos objetivos do projeto e à experiência da equipe envolvida.

REFERÊNCIAS BIBLIOGRÁFICAS

CADU. *ORM (Object Relational Mapper)*. DevMedia, 2011. Disponível em: https://www.devmedia.com.br/orm-object-relational-mapper/19056. Acesso em: 9 jul. 2025.

DEDIGAMA, Maneesha. *How To Use Sequelize with Node.js and MySQL*. Tradução técnica de Rachel Lee. DigitalOcean, 20 jul. 2022. Disponível em: https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql. Acesso em: 9 jul. 2025.

ERICKSON, Jeffrey. *MySQL: Entendendo o que é e como é usado*. Oracle, 29 ago. 2024. Disponível em: https://www.oracle.com/br/mysql/what-is-mysql/. Acesso em: 9 jul. 2025.

TIDB TEAM. *Understanding Prisma ORM*. PingCAP, 18 jul. 2024. Disponível em: https://www.pingcap.com/article/understanding-prisma-orm/. Acesso em: 9 jul. 2025.