

TRYBE

Modulo III – Back-end

Bloco 31: Arquitetura: SOLID e ORM

1) Princípios SOLID

Boas práticas que recomendam como o código deve ser escrito e organizado para otimizar manutenção, legibilidade e testabilidade.

O que exatamente é SOLID?

*S*ingle responsibility principle (Princípio da responsabilidade única);

*O*pen/Closed principle (Princípio aberto/fechado);

*L*iskov substitution principle (Princípio de substituição de Liskov);

*I*nterface segregation principle (Princípio da segregação da interface);

*D*ependency inversion principle (Princípio da inversão da dependência).

Hoje veremos S, O, D. Respondendo à pergunta:

Como podemos reusar o código no futuro para outros contextos sem alterar o código que já existe?

Single responsibility principle

Princípio da responsabilidade única, contra a alerta de alta complexidade cognitiva do CodeClimate.

Dica: ao interpretar o requisito, determinar **uma função por verbo**. Procurar simplicidade. Perceber ao dar nome para a função se tem bem responsabilidade única.

Open/Closed principle

Deixar uma function **aberta a extensões** para poder mantê-la **fechada a modificações**.

Dica: criação de objeto, function genérica, sentir alerta quando passar de 5 parâmetros.

Dependency Inversion Principle

Inversão de dependência - quem usa decide qual dependência a função terá (ex: axios ou fetch).

Dicas:

- Dependência é a linha que não pode deletar sem quebrar a function.
- Passar objeto no param.
- Abrir connection não é responsabilidade de model nem controller, pode ser service ou index.
- [curry function](#) programming
- factory- fábrica de functions, ver [code aula](#).

```
const factory = (db) => ({
  create: create(db),
  validateUser: validateUser(email, password, role, username),
  validateRole: validateRole(role)
});
```

```
10 const db = databaseConnection;
11 const model = UserModel.factory(db);
12
13 app.use(bodyParser.urlencoded({ extended: true }));
14 app.use(bodyParser.json());
15
16 app.post('/users', userController.createUser(model));
```

2) ORM - Interface da aplicação com o banco de dados

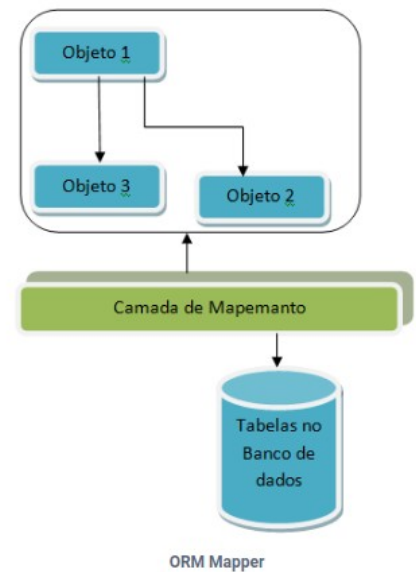
Mapeamento objeto-relacional, ou ORM, provê uma forma de, através de código **JavaScript**, **alterar e interagir com um banco de dados**.

ORM (Object Relational Mapper) é uma técnica/camada de mapeamento que permite fazer uma relação de estruturas de dados da nossa aplicação com os dados do banco de dados que as mesmas representam, abstraindo a diferença entre elas.

Bibliotecas de ORM ficam responsáveis por receber o objeto JavaScript e inserir os dados no BD.

No Node, biblioteca famosa **Sequelize**, com suporte para BDs PostgreSQL, MariaDB, MySQL, SQLite e Microsoft SQL Server.

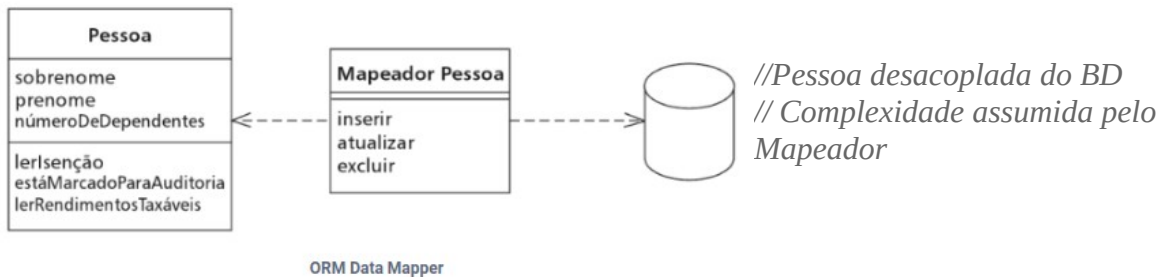
// ORM facilita tanto a vida que tem um para cada: c sharp, python, php, java...



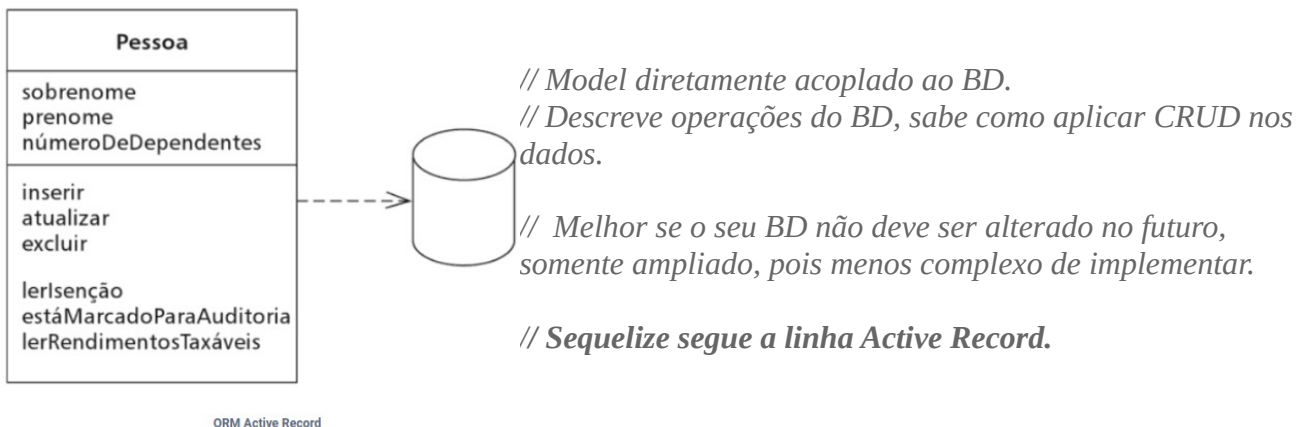
Mapeamentos

Dois padrões de mapeamento no mercado.

Data Mapper



Active Record



Sequelize

Iniciar

`\$ npm init`

`\$ npm install sequelize` (ou `npm i express nodemon sequelize mysql2` para instalar tudo)

Criar BD, por exemplo via `CREATE DATABASE IF NOT EXISTS orm_example;`

Configurar

CLI responsável por gerar e executar as operações, mysql2 para usar MySQL com sequelize.

`\$ npm install --save-dev sequelize-cli` (ou `npm i -D sequelize-cli`)

`\$ npm install mysql2`

Criar projeto vazio com pastas config, models, migrations, seeders.

`\$ npx sequelize-cli init`

No config/config.json

```
{
  "development": {
    "username": "root",
    "password": "mypassword", // melhor prática colocar dados sensíveis em variáveis de ambiente
    "database": "orm_example",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  }
}
```

Migrações

→ O que é uma migration e para que serve
Forma de **versionar** o schema do banco de dados. Cada arquivo é marcado com uma estampa **datetime**. uem clona projeto apenas roda migrations para receber o BD mais recente.

Dois movimentos: execução (Up), reversão (Down).

→ Primeira migration para criar uma tabela:

`\$ npx sequelize migration:generate --name create-users`

→ Entender code da migration

- **queryInterface** é usado pelo sequelize para modificar o BD;

- **objeto Sequelize** armazena os tipos de dados disponíveis no contexto do banco.

→ Manipular code do arquivo migration para deixar reversível

→ Fazer e desfazer migration

`\$ npx sequelize db:migrate`

`\$ npx sequelize db:migrate:undo`

migrations/[timestamp]-create-users.js

```
"use strict";

module.exports = {
  up: async (queryInterface, Sequelize) => {
    const UsersTable = queryInterface.createTable("Users", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      fullName: {
        allowNull: false,
        type: Sequelize.STRING,
      },
      email: {
        allowNull: false,
        unique: true,
        type: Sequelize.STRING,
      },
    });

    return UsersTable;
  },

  down: async (queryInterface) => queryInterface.dropTable("Users"),
};
```

Seeders

seeders/[timestamp]-users.js

Para **alimentar o BD com informações necessárias para o funcionamento mínimo** da aplicação.

→ Criar novo seed

`\$ npx sequelize seed:generate --name users`

→ Inserir dados no code do arquivo do seeder

- Parâmetro recebido pela função `queryInterface` para conversar com o BD;
- `bulkInsert` insere múltiplos dados na tabela, `bulkDelete` tira.

→ Executar e reverter seed, também usando `up` e `down`:

`\$ npx sequelize db:seed:all`

`\$ npx sequelize db:seed:undo:all`

```
"use strict";

module.exports = {
  up: async (queryInterface, Sequelize) =>
    queryInterface.bulkInsert(
      "Users",
      [
        {
          fullName: "Leonardo",
          email: "leo@test.com",
        },
        {
          fullName: "JEduardo",
          email: "edu@test.com",
        },
      ],
      {}
    ),
  down: async (queryInterface) => queryInterface.bulkDelete("Users", null, {}),
};
```

Modelo

models/User.js

Representa uma **tabela do BD**, uma instância sendo uma linha de tabela.

```
const User = (sequelize, DataTypes) => {
  const User = sequelize.define("User", {
    fullname: DataTypes.STRING,
    email: DataTypes.STRING});

  return User;
};

module.exports = User;
```

← Pode escrever na mão

Ou rodar

`\$ npx sequelize model:generate --name NomeDoModel --attributes nomeDoAtributo:string`

(que automaticamente também faz migration).

! → *sem Sequelize, model é para interagir com BD. Com, essa lógica se centraliza nos controllers.*

Operações

→ `controllers/example.js`

Realizar todos CRUDs sem precisar escrever **query SQL**

- **Sequelize abstrai** isso.

→ `models/index.js`

Criado automaticamente com `init sequelize`. Conecta com BD, coleta todos models, associa modelo com outro.

```
const express = require('express');
const { User } = require('../models');
const router = express.Router();

router.get('/', (req, res, next) => {
  User.findAll()
    .then(users => {
      res.status(200).json(users);
    })
    .catch(e => {
      console.log(e.message);
      res.status(500).json({ message: 'Algo deu errado' });
    });
});

// ...

module.exports = router;
```

Transações

Unidade de trabalho **indivisível** executada do BD de forma confiável e independente de outras transações – conceito de **atomicidade**. Ou tudo nessa unidade acontece, ou nada, conservando integridade. No BD relacional, conceito **ACID** (**atômica, consistente, isolada e durável**).

→ Unmanaged transactions

Indicar *manualmente* a circunstância em que uma transação deve ser finalizada ou revertida.

```
/_ Primeiro iniciamos a transação _/
const t = await sequelize.transaction();
try {
/_ Depois executamos as operações _/
const user = await User.create(
{
  firstName: "Bart",
  lastName: "Simpson",
},
{ transaction: t }
);
await user.addSibling(
{
  firstName: "Lisa",
  lastName: "Simpson",
},
{ transaction: t }
);
/_ Se chegou até essa linha, quer dizer que nenhum erro ocorreu.
Com isso, podemos finalizar a transação. _/
await t.commit();
} catch (error) {
/_ Se entrou nesse bloco é porque alguma operação falhou.
Portanto, reverteremos todas as operações anteriores _/
await t.rollback();
}
```

→ Managed transactions

Sequelize controla quando deve finalizar ou reverter uma transação.

```
try {
  const result = await sequelize.transaction(async (t) => {
    const user = await User.create(
      {
        firstName: "Abraham",
        lastName: "Lincoln",
      },
      { transaction: t }
    );
    await user.setShooter(
      {
        firstName: "John",
        lastName: "Boothe",
      },
      { transaction: t }
    );
    return user;
  });
/_ Se chegou até aqui é porque as operações foram concluídas com sucesso,
não sendo necessário finalizar a transação manualmente.
  `result` terá o resultado da transação, no caso um user criado _/
} catch (error) {
/_ Se entrou nesse bloco é porque alguma operação falhou.
Nesse caso, o sequelize irá reverter as operações anteriores, não sendo necessário fazer manualmente _/
}
```

Dicas diversas

dependências que precisamos no sequelize

→ →

—
`npx sequelize help` – para ver comandos disponíveis com essa interface

—
Como funciona o `index.js` criado pelo `init`:

pega todos modelos (que representam tabelas do BD e retorna `db` no final (`module.exports = db;`)).

—
Três jeitos equivalentes de criar database:

1. CREATE DATABASE etc no workbench

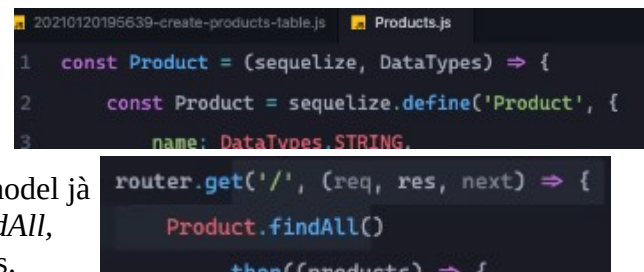
2. num arquivo do VsCode e rodar `mysql -u root -p` e `source database.sql` no terminal



```
20210120195639-create-products-table.js database.sql
1 CREATE DATABASE IF NOT EXISTS sequelize_example;
```

3. `npx sequelize db:create` depois de colocar o seu nome no config

—
Padrão: nome de fato no plural / nome ao representar no singular (ex. aqui com `Products-Product`)



```
20210120195639-create-products-table.js Products.js
1 const Product = (sequelize, DataTypes) => {
2   const Product = sequelize.define('Product', {
3     name: DataTypes.STRING,
4
5   });
6
7   router.get('/', (req, res, next) => {
8     Product.findAll()
9     .then((products) => {
```

—
No controller, ao fazer CRUD em diferentes endpoints, o model já é injetado ‘segretamente’ com as functions que precisa (`findAll`, `findByPk`, `create`, `update`, `destroy`...). Ver [doc](#) dessas queries.

—
Cuidado: migration imutável! O correto é criar, migrar, desfazer, remigrar.

—
Log: sequelize faz um log das queries que ele roda, automatico no terminal do servidor.

3) ORM – Associations

Sequelize possui ferramentas para criar, manipular e ler as tabelas e seus relacionamentos.

Relacionamentos 1:1

→ Mesmo procedimento de criação de app com sequelize do conteúdo anterior.

→ Diferença a partir da migração

references.model : Indica qual tabela nossa FK está referenciando, utilizando o nome do Model que representa aquela tabela no código.

references.key : Indica qual coluna da tabela estrangeira deve ser utilizada para nossa FK

onUpdate & onDelete : Configura o que deve acontecer ao atualizar ou excluir um usuário. Nesse caso, ‘CASCADE’, todos os produtos daquele usuário serão alterados ou excluídos.



```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    const AddressesTable = queryInterface.createTable('Addresses', {
      address_id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      city: { allowNull: false, type: Sequelize.STRING },
      street: { allowNull: false, type: Sequelize.STRING },
      number: { allowNull: false, type: Sequelize.INTEGER },
      employee_id: {
        type: Sequelize.INTEGER,
        allowNull: false,
        onUpdate: 'CASCADE',
        onDelete: 'CASCADE',
        references: { model: 'Employees', key: 'employee_id' },
      },
    });

    return AddressesTable;
  },
  down: async (queryInterface) => queryInterface.dropTable('Addresses'),
};
```


→ Escrever relações nos modelos

```
// models/Employees.js
const createEmployees = (sequelize, DataTypes) => {
  const Employees = sequelize.define('Employees', {
    employee_id: { type: DataTypes.INTEGER, primaryKey: true },
    first_name: DataTypes.STRING,
    last_name: DataTypes.STRING,
    age: DataTypes.INTEGER,
  },
  {
    timestamps: false,
  });

  Employees.associate = (models) => {
    Employees.hasOne(models.Addresses,
      { foreignKey: 'employee_id', as: 'addresses' });
  };

  return Employees;
};

module.exports = createEmployees;
```

Os métodos de criação de associações que o sequelize disponibiliza são:

- hasOne
- belongsTo
- hasMany
- belongsToMany

// No caso 1:1, hasOne e reciprocamente belongsTo .

→ criar e escrever seeders para integrar primeiros dados no BD, e `npx sequelize db:seed:all`

// Para depois mudar de seeders, mexer novamente no code deles e rodar:

npx sequelize db:migrate:undo:all

npx sequelize db:migrate

npx sequelize db:seed:all

→ criar servidor para testar associations

```
// index.js
const express = require('express');
const { Addresses, Employees } = require('./models');

const app = express();

app.get('/employees', (_req, res) => Employees
  .findAll({ include: { model: Addresses, as: 'addresses' } })
  .then((answer) => res.status(200).json(answer))
  .catch(() => res.status(500).json({ message: 'Algo deu errado' })));

const PORT = 3000;
app.listen(PORT, () => console.log(`Port: ${PORT}`));
```

// sem MSC no exemplo

← Campo **include** indica ao Sequelize as configurações da requisição:

- propriedade *model* se refere a qual tabela será utilizada
- propriedade *as* deve ser mesmo nome que no momento da criação da associação no respectivo model.

→ iniciar servidor `npx nodemon index.js` e fazer req neste endpoint no postman para verificar.

Relacionamentos 1:N

Mesmo processo, apenas substituir `hasOne` com **hasMany** no modelo.

```
// Employees.associate = (models) => {
  Employees.hasMany(models.Addresses,
    { foreignKey: 'employee_id', as: 'addresses' });
// }
```

Utilizando os relacionamentos

Dois métodos de utilização dos relacionamentos:

- * **Eager loading** , ou carregamento antecipado;
- * **Lazy loading** , ou carregamento tardio.

Eager Loading

Carrega todos os dados na mesma request. Bom para quando sabemos que iremos usar tudo.

Dica: no campo include do endpoint, pode usar propriedade attributes – exclude para não ver tudo.

```
include: [{  
  model: Addresses, as: 'addresses', attributes: { exclude: ['number'] },  
}],
```

Lazy Loading

Bom para quando sabemos que iremos usar parcialmente os dados.

Como: **condicionar a query no banco e assim termos dois usos para o endpoint.**

```
// app.get('/employees/:id', (req, res) => Employees  
  .findAll({ where: { employee_id: req.params.id } })  
  // .then((employee) => {  
  //   if (!employee.length) {  
  //     return res.status(404).send({ message: 'Funcionário não encontrado' });  
  //   }  
  //   if (req.query.includeAddresses === 'true') {  
  //     return Addresses.findAll({ where: { employee_id: req.params.id } })  
  //       .then((address) => res.status(200).json(  
  //         { employee: employee[0], address },  
  //       ));  
  //   }  
  //   return res.status(200).json(employee);  
  // })  
  // .catch(() => res.status(500).json({ message: 'Algo deu errado' })));
```

// Dois usos do endpoint:

// req GET para endpoint

`http://localhost:3000/employees/1?`

`includeAddresses=true`

// req GET para endpoint

`http://localhost:3000/employees/1`

Relacionamentos N:N

→ Criar novos modelos, inclusive um que vai ser a **tabela de associação** das duas outras, ou seja sem atributo próprio e tendo belongsToMany:

```
UserBooks.associate = (models) => {  
  models.Books.belongsToMany(models.Users, {  
    as: 'users',  
    through: UserBooks,  
    foreignKey: 'book_id', // se refere ao model em que chamamos belongsToMany  
    otherKey: 'user_id', // se refere ao model com o qual estamos criando a associação  
  });  
  // e a recíproca:  
  models.Users.belongsToMany(models.Books, {  
    as: 'books',  
    through: UserBooks,  
    foreignKey: 'user_id',  
    otherKey: 'book_id',  
  });  
};
```


→ todo fluxo igual de migrations e seeders
→ no endpoint do index.js, attributes[] para não deixar aparecer tudo
.findAll({
 where: { user_id: req.params.id },
 include: [{ model: Books, as: 'books', through: { attributes: [] } }],
})

4) Boas práticas na escrita de testes

Dicas para aumentar legibilidade e utilidade dos testes.

Com que um teste parece

Nomes e agrupamentos

- Em uma olhada o teste deve responder três perguntas: "O que está sendo testado?", "Em qual contexto esse teste está sendo rodado?" e "Qual o resultado esperado desse teste?"
- Estruturar o describe com uma legenda;
- Lembrar de nomear de jeito consistente com desenvolvimento e equipe de produto.

Estruturando testes: arrange, act, assert

3A: Arrange (code necessário para testar), **Act** (ação que queremos testar), **Assert** (esperado).

```
test("When user's age is bigger than 18, should be tagged as an adult", () => {  
  /* Arrange/Organização: inicia o usuário como um mock */  
  const user = { name: "Eduardo Pedroso", age: 21, email: "mail@mail.com" };  
  
  /* Act/Ação: chama uma função e guarda o retorno, aqui é onde o teste acontece de fato */  
  const isUserAnAdult = userUtils.isUserAnAdult(user);  
  
  /* Assert/Afirmação: a flag major realmente voltou true? É aqui que vamos ter a resposta para essa pergunta */  
  expect(isUserAnAdult).toEqual(true);  
});
```

O que testar

Teste de comportamento: pode refatorar e continua passando porque foca na experiência.

Stubs e spies

(Integrados no setup da primeira etapa Arrange). ([Documentação Jest](#) para lembrar).

Spy: objeto que grava as **interações** com outros objetos.

Verifica se determinada function foi chamada, com que params.

jest.fn() gera spy.

Stub: estabelece o retorno esperado de uma chamada **simulada**. = fake, double, dummy, mock.
Importante: devemos simular apenas dependências diretas dos nossos testes.

jest.mock()

```
/* É necessário importar o modelo que será mockado */  
const UserModel = require("../model");  
  
/* Mocka o método getById da classe UserModel */  
const getById = jest  
  .spyOn(UserModel.prototype, "getById")  
  .mockReturnValueOnce(users);
```

Dados nos testes

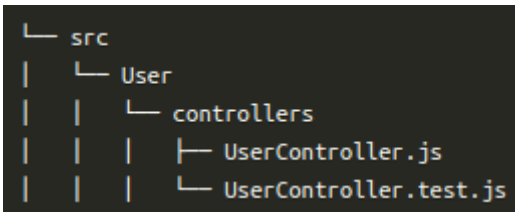
Dados usados precisam ter **nome**, ser **explícitos** e o mais **próximos da realidade** possível.

Dicas:

- Biblioteca simulando dados realistas: [FakerJs](#);
- Biblioteca de Property-based testing: [fast-check](#), simula todos cenários de produção;
- Não usar variáveis globais para dados, tudo o que testes precisam deve estar dentro deles.

Tests Colocation

Dependendo de como os arquivos são editados, eles devem estar em um **local específico**. Estruturar pastas de jeito a deixar fácil editar o teste junto com o arquivo principal, por ex:



Particularidades dos testes de back-end

Pirâmide de testes

Modelo que era norma onde testes do topo são mais lentos e abrangentes, e da base o oposto.

Testes de integração

Filosofia oposta ao teste de cada parte do código separadamente.

Vantagem: Testa cada comportamento separadamente, o que reproduz melhor o cenário real e cobra o app inteiro.

Desvantagem: Quando quebra, fica difícil identificar diretamente onde está o problema.

Ex: testar requisição endpoint como no postman.

Testar express

Sempre testar cada middleware em isolamento:

```
async function checkAuth(req, res, next) {
  if (!req.session.data) {
    return res.status(401);
  }
  next();
}

app.use(checkAuth);

describe("Middlewares", () => {
  describe("Check Auth", () => {
    it("Request received without session data, should return a 401", () => {
      /* Fazemos o setup. Como nossas funções não chamam outras funções, podemos usar um mock com jest.fn */
      const req = { session: { data: null } };
      const res = { status: jest.fn() };
      const next = jest.fn();
      /* Rodamos o teste */
      checkAuth(req, res, next);
      /* Verificamos se a chamada deu certo */
      expected(res.status).toHaveBeenCalled();
      /* E se a função next não foi chamada */
      expected(next).not.toHaveBeenCalled();
    });
  });
});
```

5) Projeto - API de Blogs

Individual – três dias.

Dicas diversas do projeto:

Para ver o que contem um token e o payload dele, usar <https://jwt.io/> .

[Doc](#) sobre operadores do sequelize.