

INSTITUTO FEDERAL DE MATO GROSSO DO SUL
CAMPUS NOVA ANDRADINA

NOTAS DE AULA

PROGRAMAÇÃO E TECNOLOGIAS
PARA APLICAÇÕES SERVIDOR 2

Prof. Me. Luiz F. Picolo

NOVA ANDRADINA - MS

Atualizado em 10 de maio de 2022

1 Criação e acesso a banco de dados relacional usando NodeJS

Hoje você acha cansativo, porém mais tarde receberá a recompensa por todo esse tempo que passou estudando.

Anônimo

Em nossa aula vamos aprender como podemos usar Node.js com PostgreSQL. Apesar de MongoDB e outras bases não relacionais serem uma das escolhas mais comuns com Node, muitos desenvolvedores, conhecem e usam PostgreSQL e não querer abrir mão de seu conhecimento a respeito. Também é importante deixar claro que partimos do pressuposto que você já saiba usar, minimamente, um banco de dados relacional como o Mysql por exemplo, e que também já saiba o básico de Node.js. Assim, vamos focar nossa aula em apenas ligar os pontos, ou seja, o Node.js com nosso banco dados PostgreSQL.

1.1 Usando o ElephantSQL

Para evitar o passo de instalação do banco em nossa maquina local, vamos usar um serviço online que poderá se comunicar com nossas aplicações desenvolvidas no Replit. Para tanto, vamos usar o plano *free* da <elephantsql.com> chamado de **Tiny Turtle**.

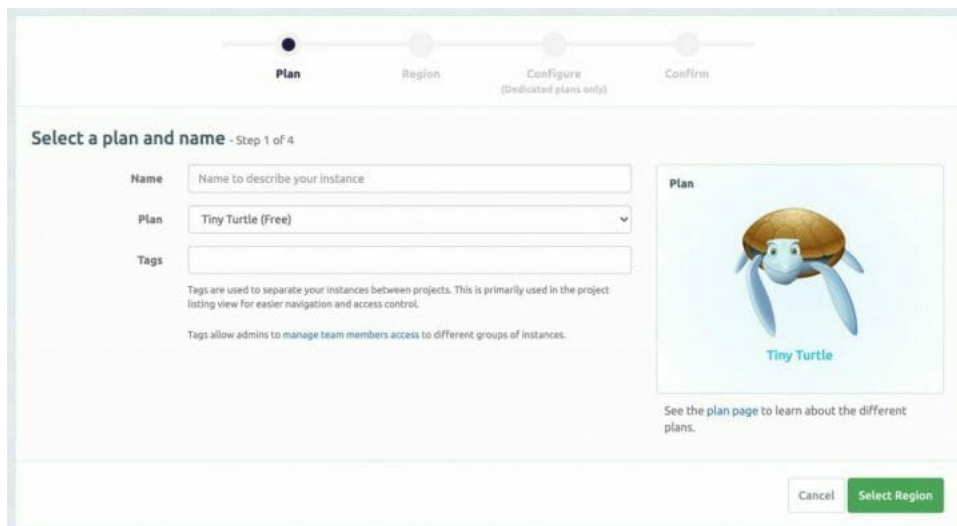
The image shows a web form for creating a new ElephantSQL instance. At the top, there is a progress bar with four steps: 'Plan' (selected), 'Region', 'Configure (Dedicated plans only)', and 'Confirm'. The main heading is 'Select a plan and name - Step 1 of 4'. Below this, there are three input fields: 'Name' with a placeholder 'Name to describe your instance', 'Plan' with a dropdown menu showing 'Tiny Turtle (Free)', and 'Tags' with a placeholder. Below the 'Tags' field, there is a small text block explaining that tags are used for separating instances between projects. To the right of the input fields, there is a box titled 'Plan' containing a cartoon illustration of a turtle and the text 'Tiny Turtle'. Below this box, there is a link to 'See the plan page to learn about the different plans.' At the bottom right of the form, there are two buttons: 'Cancel' and 'Select Region'.

Figura 1 – Criação da primeira instância

Após adicionar um nome, selecione a região (não é necessário alterar), revise os dados e clique em *Criar Instância*. Quando terminar, clique no nome da instância criada

você terá acesso às credenciais de acesso e todas as seções que utilizaremos para manipular nosso banco.

Para criar nossa primeira tabela, clique em *Browser* e será aberto o *SQL Browser*. Nele vamos digitar nossos comandos SQL.

1.1.1 Comando CREATE TABLE no PostgreSQL

O comando **SQL CREATE TABLE** é utilizado para definir uma nova tabela, inicialmente vazia (sem nenhum registro), no esquema de banco de dados atual. A tabela criada pertence ao usuário que executa o comando. Vejamos a sintaxe básica para a criação de uma tabela no postgres:

```
1 CREATE TABLE [IF NOT EXISTS] nome_tabela (  
2     nome_coluna tipo_dados [COLLATE colecao] constraint,  
3     nome_coluna tipo_dados constraint,  
4     nome_coluna tipo_dados constraint,  
5     ...,  
6     [FOREIGN KEY chave_estrangeira REFERENCES coluna]  
7     [ON DELETE acao ] [ ON UPDATE acao ]  
8 )
```

Vamos criar uma tabela **autores** que irá conter os campos **id nome**, **sobrenome** e **datanascimento**

```
1 CREATE TABLE autores (  
2     id SERIAL CONSTRAINT pk_id_autor PRIMARY KEY,  
3     nome varchar(30) NOT NULL,  
4     sobrenome varchar(40) NOT NULL,  
5     datanascimento date  
6 );
```

Agora, vamos criar a tabela de livros, incluindo os relacionamentos com as demais tabelas por meio do uso de chaves estrangeiras.

```
1 CREATE TABLE livros (  
2     id SERIAL CONSTRAINT pk_id_livro PRIMARY KEY,  
3     nome varchar(50) NOT NULL,
```

```
4      autor integer NOT NULL,  
5      editora integer NOT NULL,  
6      datapublicacao date,  
7      preco money,  
8      FOREIGN KEY (autor) REFERENCES autores (id) ON DELETE CASCADE  
9  );
```

Outra forma de estabelecer esse relacionamento é simplesmente indicar a referência de uma coluna em sua própria declaração, o que a torna uma chave primária. Neste exemplo poderíamos escrever simplesmente:

```
1  CREATE TABLE livros (  
2      id SERIAL CONSTRAINT pk_id_livro PRIMARY KEY,  
3      nome varchar(50) NOT NULL,  
4      autor integer REFERENCES autores(id) NOT NULL,  
5      editora integer NOT NULL,  
6      datapublicacao date,  
7      preco money  
8  );
```

1.1.2 Exercícios de fixação

Crie as seguintes tabelas com seus respectivos relacionamentos:

Empregado (id:Serial, matricula:integer, cpf:varchar, nome:varchar, endereço:varchar, cep:integer)

Projeto (id:Serial, nom:varchar, verba:money)

Alocação (id:Serial, projeto:foreignkey, empregado:foreignkey)

1. Escreva o código antes de adicionar ao banco
2. Valide o código com o professor
3. Adicione o código ao banco

1.2 Usando Node.js com o PostgreSQL

Para iniciarmos nossa jornada com Node.js e PostgreSQL, vamos criar nosso primeiro projeto usando o **express.js**. Para maiores detalhes, acesse o link para relem-

brar sobre o funcionamento do *Express Generator* <<https://expressjs.com/pt-br/starter/generator.html>>. Se, estiver usando o <<https://replit.com>> use o comando abaixo

```
1 npx express --view=ejs .
```

Contudo, para facilitar nosso trabalho, vamos realizar um *fork* do seguinte repositório <<https://github.com/luizpicolo/skeleton-nodejs-express-ejs.git>> e importá-lo no **replit**.

1.2.1 Conexão com o banco de dados

Assim, após todo o processo ter ocorrido sem erros, vamos configurar a conexão com o banco. Crie um arquivo **db.js** na raiz do seu repositório. Não podemos criando conexões infinitas no banco pois isso, além de ser lento, é inviável do ponto de vista de infraestrutura. Usaremos aqui um conceito chamado *connection pool*, onde um objeto irá gerenciar as nossas conexões, para abrir, fechar e reutilizar conforme possível. Vamos guardar este único pool em uma variável global, que testamos logo no início da execução para garantir que se já tivermos um *pool*, que vamos utilizar o mesmo.

```
1 let connect = function() {  
2   if (global.connection){  
3     return global.connection.connect();  
4   }  
5  
6   const { Pool } = require('pg');  
7   const pool = new Pool({  
8     connectionString: 'URL PARA O BANCO DE DADOS'  
9   });  
10  
11   global.connection = pool;  
12   return pool.connect();  
13 }  
14  
15 module.exports = { connect };
```

Para que o código acima funcione corretamente, devemos instalar a dependência **pg**, executando o comando abaixo.

```
1 npm i pg --save
```

1.2.2 Criando nosso primeiro modelo para acesso ao dados

Para que possamos testar a conexão com o banco em nossos modelos, vamos criar um modelo de exemplo chamado **Autor** e invocar o código de conexão da seguinte forma.

```
1 const db = require("../db");
2
3 class Autor {
4
5 }
6
7 module.exports = Autor;
```

Usando apenas a chamada acima, já possuímos um modelo que pode obter os dados necessários por meio de uma conexão com o banco de dados.

1.2.3 As quatro operações básicas em um banco de dados

Nas manipulações de registros realizadas diretamente em banco de dados ou em plataformas desenvolvidas no padrão *RESTful*, o conceito **CRUD** estabelece o modelo correto no manuseio desses dados.

CRUD representa as quatro principais operações realizadas em banco de dados, seja no modelo relacional (SQL) ou não-relacional (NoSQL), facilitando no processamento dos dados e na consistência e integridade das informações.

A sigla CRUD significa as iniciais das operações create (criação), read (leitura), update (atualização) e delete (exclusão). Essas quatro siglas tratam a respeito das operações executadas em bancos de dados relacional (SQL) e não-relacional (NoSQL). Essas operações pertencem ao agrupamento chamado de *Data Manipulation Language (DML)*, utilizado na linguagem Structured Query Language (SQL)¹.

1.2.3.1 Create

A operação de criação de um registro em uma tabela é realizada pelo comando INSERT. Exemplo:

¹ Para mais detalhes acesse: <<https://blog.betrybe.com/tecnologia/crud-operacoes-basicas/>>

```
1 class Autor {
2   static async insert(data){
3     const connect = await db.connect();
4     const sql = 'insert into autores(nome, sobrenome, datanascimento)
5       ↪ values ($1, $2, $3)';
6     const values = [data.nome, data.sobrenome, data.datanascimento];
7     return await connect.query(sql, values);
8   }
}
```

1.2.3.2 Read

A operação de consulta de um ou mais registros em uma tabela é realizada pelo comando SELECT. Exemplo:

```
1 static async select(){
2   const connect = await db.connect();
3   return await connect.query('select * from clientes');
4 }
```

1.2.3.3 Update

Comando utilizado para a atualização de um ou mais registros de uma tabela. Exemplo:

```
1 static async update(id, data){
2   const connect = await db.connect();
3   const sql = 'UPDATE clientes SET nome=$1, idade=$2, uf=$3 WHERE id=$4';
4   const values = [data.nome, data.idade, data.uf, id];
5   return await connect.query(sql, values);
6 }
```

1.2.3.4 Delete

Comando utilizado para a exclusão de registro (s) de uma tabela. Exemplo:

```
1 static async delete(id){
2   const connect = await db.connect();
3   const sql = 'DELETE FROM clientes where id=$1;';
4   return await connect.query(sql, [id]);
5 }
```

1.2.4 Invocando os métodos nas rotas

Para que possamos invocar os métodos de manipulação de dados do modelo, precisamos criar uma rota. Para tanto, crie uma nova rota utilizando o **express** e adicione a seguinte rota para que seja feita a seleção dos dados.

```
1 var express = require('express');
2 var router = express.Router();
3 // Invocando o modelo Autor
4 const Autor = require("../models/autor");
5
6 /* Listando os usuários e apresentando um Json */
7 router.get('/', async function(req, res, next) {
8   const data = await Autor.select();
9   res.json(data.rows);
10 });
11
12 module.exports = router;
```


2 Mapeamento objeto-relacional (ORM)

É melhor você tentar algo, vê-lo não funcionar e aprender com isso, do que não fazer nada.

Mark Zuckerberg

Criar um banco de dados, ou mais formalmente o Sistema Gerenciador de Banco de Dados (SGBD), é uma tarefa complexa presente em todos os projetos. Escolher qual será utilizar em meio a tantos como: **mysql**, **sqlserver**, **oracle**, **sqlite**, **postgre**, **mongodb** e **etc**, ou, qual se adapta melhor ao projeto é uma decisão que deve ser tomada com cautela.

Outro fato é a forma com que os dados são tratados em cada um. Por exemplo, para um campo **id** que deve ser auto-incrementado, no Postgres usando o seguinte código SQL:

```
1 CREATE TABLE Pessoas (  
2     id SERIAL NOT NULL,  
3     nome varchar  
4 );
```

Já no Mysql, o mesmo código seria escrito da seguinte forma:

```
1 CREATE TABLE Pessoas (  
2     id int NOT NULL AUTO_INCREMENT,  
3     nome varchar  
4 );
```

Logo, caso houvesse uma mudança, nosso código teria que ser alterado para satisfazer a nova base de dados com todas as suas diferenças. Assim, pensando nas possíveis mudanças que o projeto pode ter durante sua vida útil, foram desenvolvidas algumas técnicas para facilitar a vida dos desenvolvedores. Uma que iremos comentar é o Object Relational Mapping (ORM) ou Mapeamento Objeto Relacional. Para diminuir a complexidade, já que o ORM torna o banco de dados mais próximo da arquitetura de classe, removendo os comando SQL de vista, para que possamos focar em “Qual é o fluxo que minha aplicação deve seguir” e deixando de lado “Qual é a query que eu deveria usar aqui?”. Então, nesse aspecto, iremos abordar um pouco sobre o que são ORM, suas práticas e exemplos usando JavaScript.

2.1 ORM: o que é?

Como já citei anteriormente um ORM é a sigla em inglês para Mapeamento Objeto Relacional e consiste em manter o uso de orientação a objetos e um pouco do conceito de non-query, pois serão raros os momentos nos quais teremos que escrever uma linha de código SQL [Muramatsu 2020]. Um ORM é uma solução completa para resolver a incompatibilidade de **impedância** entre objetos de programa e tabelas de banco de dados, promovendo bancos de dados relacionais com recursos de orientação a objetos [Song e Gao 2012].

Outro fato muito importante e curioso sobre os ORM é que eles operam como um agente de banco de dados, sendo possível através de pouquíssimas mudanças, utilizar o mesmo código para mais de um banco de dados. Não importa se ele está em Mysql, SqlServer ou até mesmo Oracle. Ele consegue agir da mesma forma em alguns bancos de dados, você só precisa mudar o driver de conexão e está pronto para uso. Neste conceito é importante lembrar que cada uma de nossas tabelas são vistas como uma instância de uma classe, tendo suas características declaradas diretamente na sua classe “esquema”. Quando trabalhamos com esse tipo de esquema sempre teremos um arquivo de configuração, responsável por fornecer os dados para que o componente de ORM possa se comunicar com o banco e aplicação. Uma outra questão que pode causar dúvidas é como gerar o banco de dados através dessas classes que comentamos anteriormente. Bom veremos isso na prática mais a abaixo [Muramatsu 2020].

Além da manipulação, outra preocupação, quando se retrata o cenário de desenvolvimento de software com banco de dados, é a segurança. Ai utilizar um ORM (consolidado) adquirimos a possibilidade de direcionar os esforços do projeto para a inovação mantendo a confiabilidade, segurança, e tradição dos bancos de dados relacionais.

2.2 ORM: como funciona?

Para facilitar a aplicação desta técnica surgiram os frameworks ORM, como por exemplo, o Hibernate para Java, o ActiveRecord para Ruby, e o que usaremos para nossos projeto o Sequelize para NodeJS (JavaScript). Desta forma, esses frameworks se localizam em uma camada intermediária entre a lógica da sua aplicação e o seu SGDB [Magazine 2020].

O framework passa a receber as solicitações de interação com o SGDB através de objetos de sua aplicação e gera automaticamente todo o SQL necessário para a operação solicitada, nos poupando do trabalho de escrita e manutenção deste SQL e abstraindo o uso do SGDB, fazendo com que nos preocupemos apenas com nosso modelo de objetos. Além disso, o framework já trata as variações de tipos de dados existentes entre nossa

aplicação e nosso SGDB [Magazine 2020].

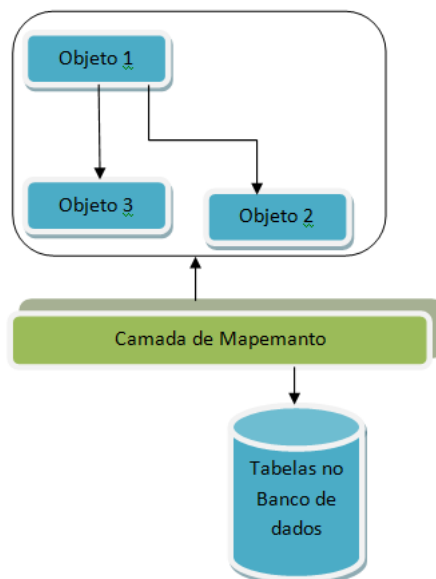


Figura 2 – Ilustração do mapeamento Fonte: [Magazine 2020]

O ORM funciona através do mapeamento das características da base de dados para os objetos de nossa aplicação. O primeiro conceito chave é traçar um paralelo entre Classe x Tabela e Propriedade x Coluna. O ORM nos permite informar em que tabela cada classe será persistida e em que coluna do SGDB cada propriedade ficará armazenada.

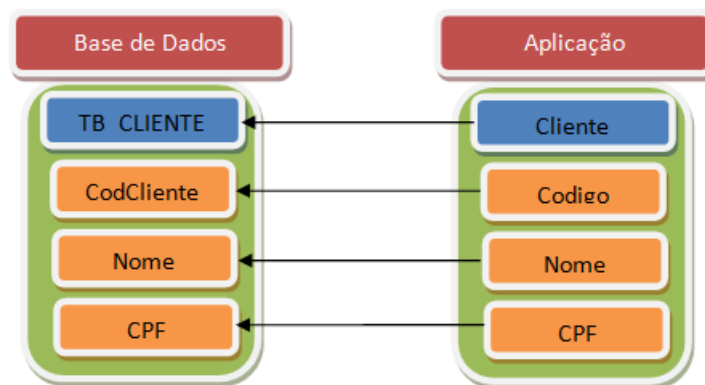


Figura 3 – Ilustração de uma classe para uma tabela Fonte: [Magazine 2020]

2.3 Configurando o frameworks ORM Sequelize

O Sequelize possui um utilitário de linha de comando chamado **Sequelize CLI** que auxilia em diversas atividades incluindo funcionalidades para nos ajudar com **migrations**, as quais veremos mais a frente.

De forma geral, vamos iniciar um novo projeto Node usando as instruções mais básicas. Primeiro, crie um diretório e, a partir dele, inicie um novo projeto.

```
1 mkdir novo-projeto
2 cd novo-projeto
3 npm init -y
```

Logo após, ainda no diretório, adicione as bibliotecas que usaremos

```
1 npm install sequelize pg --save
2 npm install --save-dev sequelize-cli
```

Agora, vamos inicializar nosso projeto com o **Sequelize-CLI**, rodando o comando abaixo.

```
1 npx sequelize-cli init
```

O resultado será o retorno e a criação de alguns diretórios listados abaixo.

```
1 Created "config/config.json"
2 Successfully created models folder at ...
3 Successfully created migrations folder at ...
4 Successfully created seeders folder at ...
```

O diretório **Models** nós já conhecemos. Contudo, os novos criados tem como função conectar ao banco e criar tabelas e adicionar dados. Vamos ver cada um:

- **config/config.json** é o local onde colocaremos os dados para conexão, como usuário, senha, host, etc. Nesse arquivo, um dos parâmetros é o **dialect**. Nele devemos colocar o SGBD desejado. Por padrão, ele é gerado como **mysql**, por isso, vamos mudar para **postgres**/
- **migrations** é o local no qual criaremos nosso código equivalente ao SQL para a criação das tabelas, porém, usando apenas JavaScript.
- **seeders** será onde criaremos o código que semeará os dados no banco.

2.4 Criando nosso primeiro modelo e migração

Vamos criar um cenário bem simples no qual criaremos uma nova tabela no banco de dados. Para criar uma **migration**, podemos usar o seguinte comando no console, que baixa e executa o **Sequelize CLI** com o comando de criação de migration.

```
1 npx sequelize-cli model:create --name usuario --attributes nome:string
```

O comando acima irá criar uma pasta **migrations** no seu projeto (se ela não existir) e dentro dela nosso primeiro arquivo de **migration**, idêntico ao abaixo.

```
1 'use strict';
2 module.exports = {
3   async up(queryInterface, Sequelize) {
4     await queryInterface.createTable('usuarios', {
5       id: {
6         allowNull: false,
7         autoIncrement: true,
8         primaryKey: true,
9         type: Sequelize.INTEGER
10      },
11      nome: {
12        type: Sequelize.STRING
13      },
14      createdAt: {
15        allowNull: false,
16        type: Sequelize.DATE
17      },
18      updatedAt: {
19        allowNull: false,
20        type: Sequelize.DATE
21      }
22    });
23  },
24  async down(queryInterface, Sequelize) {
25    await queryInterface.dropTable('usuarios');
26  }
27 };

```

Toda migração possui um **up** e um **down**, referente ao script de migration (criação) e o rollback (desfaz o que foi criado) da mesma. Isso permite que em caso de necessidade seja desfeita a migration, permitindo a gestão das alterações do banco de dados com muito mais detalhe.

Também será criado um modelo com o código abaixo

```
1  'use strict';
2  const { Model } = require('sequelize');
3  module.exports = (sequelize, DataTypes) => {
4    class usuario extends Model {
5      /**
6       * Helper method for defining associations.
7       * This method is not a part of Sequelize lifecycle.
8       * The `models/index` file will call this method automatically.
9       */
10     static associate(models) {
11       // define association here
12     }
13   }
14   usuario.init({
15     nome: DataTypes.STRING
16   }, {
17     sequelize,
18     modelName: 'usuario',
19   });
20   return usuario;
21 };
```

Assim, para enviar os dados ao banco, devemos executar dois comandos, o primeiro para criar o banco e outro para criar as tabelas do banco selecionado.

```
1  npx sequelize-cli db:create // Não deve ser usado para o ElephantSQL
2  npx sequelize-cli db:migrate
```

2.5 Adicionado o Frameworks Express ao Projeto

Para que possamos utilizar os recursos do **sequelize** em um ambiente web, vamos usar no frameworks **ExpressJS**. Para tanto, crie um arquivo, na raiz do projeto com o nome de **index.js** e adicione o seguinte código:

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function(req, res) {
5    res.send('Olá Mundo!');
6  });
7
8  app.listen(3000, function() {
9    console.log('App de Exemplo escutando na porta 3000!');
10 });
```

Vamos adicionar o ExpressJS ao nosso projeto executando o código abaixo no terminal.

```
1  npm i express --save
```

Outra biblioteca opcional é o **nodemon** que auxilia no desenvolvimento executando o *restart* da aplicação;

```
1  npm i nodemon --save-dev
```

Agora, edite o arquivo **package.json** e altere a seguinte linha:

```
1  // Terço código antigo
2  "scripts": {
3    "test": "echo \"Error: no test specified\" && exit 1"
4  },
```

```
1 // Trecho código novo
2 "scripts": {
3   "start": "nodemon index.js" // caso esteja usando o nodemon
4 },
5
6 "scripts": {
7   "start": "node index.js" // caso não esteja usando o nodemon
8 },
```

Por fim, executando o ‘npm start’ e abrindo o endereço ‘localhost:3000’ devemos visualizar a seguinte mensagem:

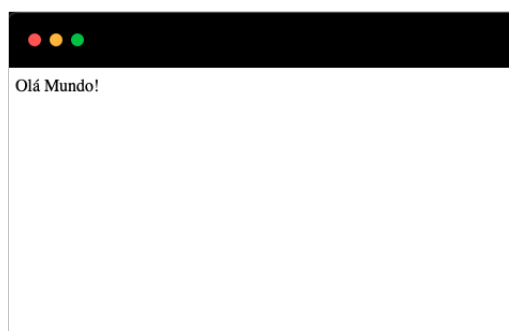


Figura 4 – Fonte: O autor

Agora, se tudo deu certo, estamos prontos para criar nossas interações com o banco de dados, também conhecido como CRUD.

2.6 CRUD com Sequelize

Os conceitos sobre CRUD (Create, Read, Update e Delete), já foram abordados no Capítulo 1.2.3. Portanto, não iremos retornar o conceitos, mas, apenas aplicá-lo.

2.6.1 Inserindo dados - Create

Agora vamos iniciar nosso CRUD com o C de Create. O Sequelize fornece alguns métodos para a inserção de dados, sendo a mais direto deles o método **create**. Esse método cria imediatamente o registro na tabela com o **objeto** passado por parâmetro. Veja o código:

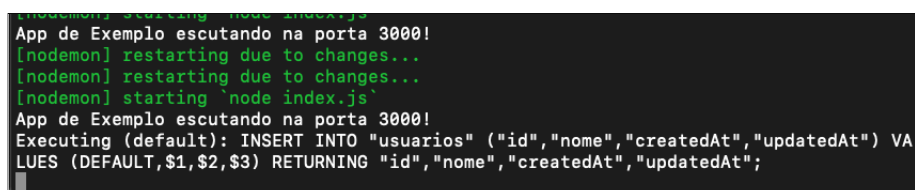

```
1  const usuario = await Usuario.create({
2    nome: 'Picoló'
3  })
```

Contudo, como estamos usando o ExpressJS, vamos adicionar o código criado em um rota com o verbo HTTP **Get** inicialmente.

```
1  var express = require('express');
2  var { usuario } = require('./models');
3
4  var app = express();
5
6  app.use(express.json());
7  app.use(express.urlencoded({ extended: true }));
8
9  app.post('/', async function(req, res) {
10    var resultado = await usuario.create()
11    res.json(resultado);
12  });
13
14  app.listen(3000, function() {
15    console.log('App de Exemplo escutando na porta 3000!');
16  });
```

Note as linhas 2. 6 e 7. Elas não existias no código anterior e foram adicionado, respectivamente, para importar o modelo **usuario** para nossas rotas e realizar parsing do conteúdo das requisições que ela receber.

Ao rodar a aplicação agora você vai notar que no console vão aparecer algumas informações relevantes desta vez, como, por exemplo, o SQL do INSERT (porque ela não existia) que foi gerado automaticamente para você.



```
[nodemon] starting 'node index.js'
App de Exemplo escutando na porta 3000!
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting 'node index.js'
App de Exemplo escutando na porta 3000!
Executing (default): INSERT INTO "usuarios" ("id","nome","createdAt","updatedAt") VA
LUES (DEFAULT,$1,$2,$3) RETURNING "id","nome","createdAt","updatedAt";
```

Figura 5 – Fonte: O autor

2.6.2 Retornando dados do banco - Read

Agora vamos fazer o R do CRUD, de Read/Retrieve. Ele é ainda mais simples do que o Create, basta usarmos a função **findAll** disponibilizada pelo nosso objeto **Usuario**, que teremos um array de produtos à nossa disposição.

```
1 app.get('/', async function(req, res) {
2   var resultado = await usuario.findAll();
3   res.json(resultado);
4 });
```

Caso seja necessário fazer uma busca no banco para retornar apenas um **usuário**, podemos utilizar o método **findByPk** ou **findOne**

```
1 app.get('/:id', async function(req, res) {
2   var resultado = await usuario.findByPk(req.params.id);
3   res.json(resultado);
4 });
5
6 app.get('/:id', async function(req, res) {
7   var resultado = await usuario.findOne({
8     where: {
9       id: req.params.id
10    }
11  });
12  res.json(resultado);
13 });
```

Aqui vale uma explicação sobre as três propriedades de requisição que são preenchidas de diferentes formas dependendo da origem:

req.query vem de parâmetros de consulta na URL, como link, em que **req.query.name** obtém o parâmetro passado pela seguinte url 'url/teste?nome=teste'

req.params vem de segmentos de caminho do URL que correspondem a um parâmetro na definição de rota, como `/song/:id`. Então, com uma rota usando essa designação e uma URL como `/song/48586`, então **req.params.id** retornará "48586".

Já a propriedade **req.body** vêm de uma postagem de formulário em que os dados do formulário (que são enviados no conteúdo do corpo) foram analisados nas propriedades da tag body.

2.7 Atualizando dados - Update

O próximo passo é atualizarmos um item da nossa tabela, o U do CRUD: Update! Para atualizarmos um item, primeiro precisamos retorná-lo do banco de dados usando alguma função de find do Sequelize. No passo anterior, usamos a `findByPk` para retornar o produto com ID 1. Vamos escrever o nosso código de update imediatamente abaixo.

```
1  app.put('/:id', async function(req, res) {
2    // Buscando dados
3    var resultado = await usuario.findByPk(req.params.id);
4
5    // Atualizada dados
6    resultado.nome = req.body.nome
7    var novo_resultado = await resultado.save();
8    res.json(novo_resultado);
9  });
```

2.8 Deletando dados - Delete

E por fim, vamos ao D do CRUD, de DELETE! Assim como para salvar e retornar dados existem diversas formas de fazer, para o Delete não é diferente. Você pode usar `Produto.destroy` e passar um `where` por parâmetro, ou então retornar um produto e usar a função `destroy` do próprio objeto retornado, você decide.

```
1  app.delete('/:id', async function(req, res) {
2    var resultado = await usuario.destroy({
3      where: { id: req.params.id }
4    });
5
6    res.json(resultado);
7  });
```

2.9 Padrão de rotas

Se vocês perceberam, as rotas seguem um padrão. Este pode ser usado em todos os projetos da forma descrita abaixo, sendo elas, as rotas, o básico de um projeto. Outras podem ser criadas especificadamente para atender determinada necessidade.

```
1  app.get('/', async function(req, res) {
2    // Listar todos os dados
3  });
4
5  app.get('/:id', async function(req, res) {
6    // Listar dado específico
7  });
8
9  app.put('/:id', async function(req, res) {
10   // Atualizar dado específico
11 });
12
13 app.post('/', async function(req, res) {
14   // Adicionar um novo dado
15 });
16
17 app.delete('/:id', async function(req, res) {
18   // Deletar dado específico
19 });
```

2.10 Relacionamentos com Sequelize

Ao trabalharmos com banco de dados relacionais, pela própria definição do nome, percebe-se que os relacionamentos se fazem presente. Dessa forma, o Sequelize, que é o ORM que estamos utilizando, tem mecanismos para abstrair os relacionamentos e “importá-los” para a programação orientada a objeto.

2.10.1 Tipos de relacionamento

Os tipos de relacionamento em um banco relacional, provavelmente, você já conhece. Sendo eles:

- 1 para 1

- 1 para N
- N para N

O Sequelize é compatível com as associações citadas, e os métodos de criação de relacionamentos, respectivamente, são:

- `hasOne` (tem um)
- `belongsTo` (pertence a)
- `hasMany` (tem muitos)
- `belongsToMany` (pertence a muitos)

2.11 Criando o primeiro relacionamento com a entidade **Usuario**

Utilizando a entidade **Usuario** criada anteriormente, vamos criar uma nova entidade chamada **Empresa** com apenas o atributo **nome**.

1

```
npx sequelize-cli model:create --name empresa --attributes nome:string
```

Agora, devemos relacionar **Empresa** com **Usuario**, no qual, as duas entidades devem possuir a seguinte relação:



Figura 6 – Fonte: O Autor

Para tanto, devemos adicionar um novo atributo em **Usuário** que represente a **chave estrangeira** que referencia a tabela **empresas** que será gerada no banco de dados pelo Sequelize. Porém, nosso banco com a tabela **usuarios** já foi criada e, não devemos alterar as migração já persistida no banco. Para isso então, devemos criar uma **migração** que adicione os atributos desejados, da seguinte forma:

1

```
npx sequelize-cli migration:create --name
↪ adicionar-empresa-id-para-usuario
```

E, diferente de quanto criamos o modelo, as migrações não adicionam o código para que seja adicionado o atributo. Logo, devemos realizar as mudanças de forma manual. Abra o arquivo criado e adicione o seguinte código:

```
1      'use strict';
2
3      module.exports = {
4          async up (queryInterface, Sequelize) {
5              await queryInterface.addColumn('usuarios', 'empresaId', {
6                  allowNull: false,
7                  type: Sequelize.INTEGER,
8                  references: { model: 'empresas', key: 'id' }
9              })
10         },
11
12         async down (queryInterface, Sequelize) {
13             await queryInterface.removeColumn('pacientes', 'empresaId')
14         }
15     };
```

Neste momento, você também deve criar todas as rotas necessárias para manipular os dados de uma **empresa**. Caso não lembre, recomendo que retorne a seção 2.6. Assim, usando o Sequelize, a nível de Banco de Dados, criamos o relacionamento. Agora, devemos criar o mesmo usando as referências do Sequelize.

2.12 Relacionamento 1 para N

Para relacionarmos **empresa** com **usuario**, devemos usar os métodos de relacionamento presentes no Sequelize listando anteriormente. No caso de um relacionamento **1 para N** dizemos que:

- Um usuário pertence a uma empresa
- Já uma empresa, possui muitos usuários

Adicionado ao código para cada classe do respectivo modelo, ficaria da seguinte forma:

```
1 // Entidade Empresa
2 static associate(models) {
3   this.hasMany(models.usuario, { as: 'usuarios' })
4 }
5
6 // Entidade Usuario
7 static associate(models) {
8   this.belongsTo(models.empresa, { as: 'empresa' })
9 }
```

2.12.1 Retornado dados por meio do relacionamento criado

Agora, vamos retornar os dados de cada entidade. Um **usuário** deve estar contido dentro de uma empresa. Assim, vamos criar uma rota para retornar a empresa que o usuário está relacionado.

```
1 app.get('/usuarios/:id/empresa', async function(req, res) {
2   var resultado = await usuario.findPk(req.params.id, {include:
  ↳ 'empresa'});
3   res.json(resultado.empresa);
4 });
```

E da mesma forma, vamos criar uma rota que retorne todos os usuários de uma empresa.

```
1 app.get('/empresas/:id/usuarios', async function(req, res) {
2   var resultado = await empresa.findPk(req.params.id, {include:
  ↳ ['usuarios']});
3   res.json(resultado.usuarios);
4 });
```

2.13 Exercício de fixação

Crie dois modelos (usando o sequelize-cli) seguindo o Diagrama Entidade Relacional (DER) utilizado no trabalho do primeiro bimestre. Crie todas as rotas necessárias.

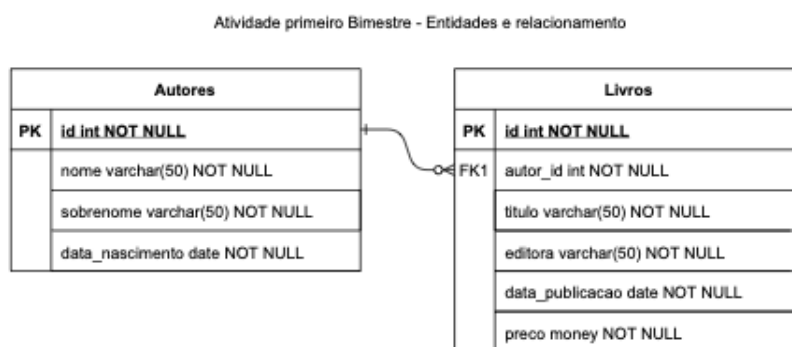


Figura 7 – Relacionamentos Autores e Livros Fonte: [?]

2.14 Relacionamento N para N

Referências

- ELMASRI, R. et al. *Sistemas de banco de dados*. Pearson Addison Wesley São Paulo, 2005. Nenhuma citação no texto.
- MAGAZINE, N. *ORM - Object Relational Mapping - Revista Easy .Net Magazine* 28. 2020. Disponível em: <<https://www.devmedia.com.br/orm-object-relational-mapping-revista-easy-net-magazine-28/27158>>. Acessado em: 18/04/2022. Citado 2 vezes nas páginas 10 e 11.
- MURAMATSU, A. *Introdução a ORM no Node.js com Sequelize + exemplo prático*. 2020. Disponível em: <<https://ezdevs.com.br/introducao-a-orm-no-node-js-com-sequelize/>>. Acessado em: 11/04/2022. Citado na página 10.
- SANCHES, A. R. *Disciplina: Fundamentos de Armazenamento e Manipulação de Dados*. 2005. Acessado em:. Disponível em: <<https://www.ime.usp.br/~andrers/aulas/bd2005-1/aula7.html>>. Nenhuma citação no texto.
- SONG, H.; GAO, L. Use orm middleware realize heterogeneous database connectivity. In: IEEE. *2012 Spring Congress on Engineering and Technology*. [S.l.], 2012. p. 1–4. Citado na página 10.
- TAKAI, O. K.; ITALIANO, I. C.; FERREIRA, J. E. *Introdução a banco de dados. Departamento de Ciências da Computação. Instituto de Matemática e Estatística. Universidade de São Paulo. São Paulo*, 2005. Nenhuma citação no texto.