[00:00] Olá. Boas-vindas ao segundo curso de Sequelize da formação de NodeJS da Alura. Esse curso continua imediatamente de onde paramos o Curso 1, mas o foco agora é avançar em algumas funcionalidades do Sequelize e também organizar melhor nossa aplicação. Vamos lendo o CRUD básico e vamos trabalhar com algumas ferramentas do Sequelize que vão ajudar a resolver alguns requisitos específicos de um projeto.

[00:22] O que vamos aprender? Mais ferramentas do Sequelize e como utilizá-las, vamos entender como o Sequelize traduz ferramentas do SQL, e como conseguimos usar isso para atender necessidades específicas de um projeto que vão além do CRUD básico.

[00:38] Como usamos o Sequelize para manter e garantir a integridade dos dados que estão sendo movimentados no banco. Isso é super importante, e como organizar melhor uma aplicação à medida que ela cresce em complexidade, indo além das camadas básicas do MVC.

[00:55] Se você ainda não conhece o Sequelize, é legal que você faça o primeiro curso para entender o que essa ferramenta faz, como ela funciona, como ela se conecta com SQL e etc, porque não vamos abordar isso durante o curso. Se você já teve contato com SQL e já sabe como ele funciona e quer avançar, você pode continuar o projeto de onde ele parou, vamos dar o link para você baixar e as instruções para instalação.

[01:16] No final deste curso você já vai ter visto uma boa quantidade de soluções que o Sequelize traz para o seu back-end, para o seu produto, para você gerar APIs rest, e também como implementar funcionalidades a mais em um projeto de acordo com a sua necessidade. Eu sou a Juliana Amoasei, vou estar com você durante este curso, então vamos começar.

**02 Preparando o ambiente**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/orm-nodejs-avancando-sequelize/task/79535/next)

Para esse curso, vamos continuar de onde paramos no [curso anterior](https://cursos.alura.com.br/course/orm-nodejs-api-sequelize-mysql). Se você já fez o curso anterior e quiser continuar direto com o mesmo projeto, pode pular essa parte; mas se precisar ou preferir começar com um projeto novo, baixe o [repositório](https://github.com/alura-cursos/1862-sequelize/tree/master) com a seguinte estrutura e arquivos iniciais:

.

├── api

│ ├── config

│ │ └── config.json

│ ├── controllers

│ │ ├── NivelController.js

│ │ ├── PessoaController.js

│ │ └── TurmaController.js

│ ├── index.js

│ ├── migrations

│ │ ├── 20200505131114-create-pessoas.js

│ │ ├── 20200526194618-create-niveis.js

│ │ ├── 20200526194804-create-turmas.js

│ │ └── 20200526194858-create-matriculas.js

│ ├── models

│ │ ├── index.js

│ │ ├── matriculas.js

│ │ ├── niveis.js

│ │ ├── pessoas.js

│ │ └── turmas.js

│ ├── routes

│ │ ├── index.js

│ │ ├── niveisRoute.js

│ │ ├── pessoasRoute.js

│ │ └── turmasRoute.js

│ └── seeders

│ ├── 20200505161755-demo-pessoa.js

│ ├── 20200601170039-demo-nivel.js

│ ├── 20200601170107-demo-turmas.js

│ └── 20200601170115-demo-matriculas.js

├── diagrama de banco - descricao da imagem.txt

├── Diagrama Relacional - escola de inglês.pdf

├── .gitignore

├── package.json

├── package-lock.json

├── requisitos.md

└── .sequelizercCOPIAR CÓDIGO

Agora siga os seguintes passos de instalação:

Navegue pelo terminal até o diretório do projeto e instale as dependências com o comando npm install.

Se você ainda não criou um banco de dados local para trabalhar nesse projeto, vai precisar fazer isso agora. No curso usamos o MySQL; caso precise pode seguir as instruções que usamos no [curso anterior](https://cursos.alura.com.br/course/orm-nodejs-api-sequelize-mysql) para criar um novo banco de dados chamado escola\_ingles e conectar-se nele. Durante os cursos vamos fazer as consultas direto no terminal do MySQL, mas se você quiser pode usar algum cliente, como o MySQL Workbench.

Uma vez criado o banco, confira os dados no arquivo api/config/config.json:

{

"development": {

"username": "alura", //utilize seu nome de user

"password": "admin123", //sua senha, se existir

"database": "escola\_ingles",

"host": "127.0.0.1",

"dialect": "mysql",

"operatorsAliases": false

},

//restante do código

}COPIAR CÓDIGO

Se for o caso, troque o nome e senha de usuário para os que você estiver usando em seu banco local. Se quiser usar outro banco que o Sequelize dê suporte, como SQLite, MSSQL Server, MariaDB ou PostgreSQL, modifique também essa informação conforme a documentação do Sequelize:

dialect: 'mysql' | 'mariadb' | 'postgres' | 'mssql'COPIAR CÓDIGO

E faça a instalação da dependência:

$ npm install --save pg pg-hstore # Postgres

$ npm install --save mysql2

$ npm install --save mariadb

$ npm install --save sqlite3

$ npm install --save tedious # Microsoft SQL ServerCOPIAR CÓDIGO

Rode os comandos de migração do Sequelize no terminal para criar as tabelas no banco: npx sequelize-cli db:migrate Você pode conferir se as tabelas foram criadas com sucesso através do comando show tables; no terminal do MySQL.

Você pode usar os arquivos de seed que estão na pasta seeders do projeto para popular as tabelas com dados de teste. Vamos usar bastante esses dados durante o curso, então é super recomendável que você faça este passo para que o seu banco tenha dados pra serem trabalhados no projeto. Você pode usar o comando no terminal npx sequelize-cli db:seed:all.

Rode na pasta raiz do projeto o comando de terminal npm start para subir o servidor local. O Express fará a conexão em localhost:3000/, caso queira modificar a porta você pode alterar no arquivo api/index.js, que é o ponto de entrada da aplicação.

Faça o teste no Postman das rotas, por exemplo GET localhost:3000/pessoas para trazer do banco os registros da tabela Pessoas, além das demais rotas que estão na pasta api/routes.

Neste projeto utilizamos um linter para cuidar do estilo do código. Caso não queira utilizar, basta deletar a dependência e o arquivo .eslintrc da raiz do projeto:

"devDependencies": {

"eslint": "^7.4.0", //delete esta linha

"nodemon": "^2.0.4"

}COPIAR CÓDIGO

Caso queira utilizar o Linter, faça a instalação e confira se o script do eslint está adicionado no package.json: "scripts": { "lint": "eslint api --fix", "start": "nodemon ./api/index.js "npm run lint"" },

Com o ambiente configurado, podemos seguir em frente!

[00:00] Para este curso já temos um projeto encaminhado. Vou deixar um link para vocês baixarem ele do zero, mas se você está emendando este curso no curso anterior de Sequelize você pode continuar do mesmo projeto. As únicas alterações que foram feitas aqui são esse arquivo requisitos.md, que é onde eu incluí os requisitos que vamos trabalhar durante este curso que estamos trabalhando agora, e também foi adicionado um linter para fazer a checagem de estilo do código.

[00:30] Se você nunca usou linter não tem problema, ele só está aqui para ver se a endentação está certa, se tem ponto e vírgula, essas coisas. Não vai fazer diferença no Sequelize, no banco de dados, nada.

[00:40] Vamos dar uma olhada no projeto atual. Por enquanto temos uma API que reflete esse diagrama de banco que recebemos anteriormente no Curso 1. A partir desse diagrama montamos modelo, controladores, rotas, podemos até dar uma olhada.

[00:55] Os modelos foram criados a partir do Sequelize, ele tem recursos e ferramentas para criar esses modelos automaticamente. Os controladores criamos, também as rotas para testar, tudo isso, está tudo funcionando, o CRUD básico que fizemos no curso 1, e o cliente pode utilizar o sistema.

[01:14] Se viermos no Postman, nas rotas que fizemos no curso anterior e que estão disponíveis neste projeto, estão todas funcionando na porta 3000. Está tudo ok. O que acontece? Depois de um tempo, o cliente percebeu que algumas funcionalidades não tinham e ele precisava. O que vamos fazer neste curso é implementar essas funcionalidades que coloquei no arquivo requisitos.md.

[01:40] Mas antes de começar realmente a trabalhar com isso vamos ter que lidar com uma coisa que aconteceu justamente entre o projeto anterior do curso 1 e esse aqui. O pessoal que desenvolve o Sequelize lançou uma atualização de versão. A versão que temos no nosso projeto é a 5.21, mas se viermos na documentação do Sequelize ele foi atualizado para a versão 6.

[02:06] Até aí tudo bem, vamos fazer o que fazemos nesses casos, damos uma olhada no registro de modificações, no *change log*, isso é normal, toda documentação de ferramenta tem para dizer “dessa versão atualizamos tal coisa”, e vemos se tem alguma mudança importante que impacta no nosso projeto, porque sempre pode acontecer de alguns métodos serem descontinuados, e se atualizamos nosso projeto fica quebrado.

[02:30] Se você está começando agora, está baixando o projeto do zero lá no link que vamos disponibilizar, não vai ter que se preocupar com isso, porque já vai vir ele todo atualizado.

[02:41] No caso aqui da documentação do Sequelize, do registro de mudanças, já me adiantei e vi que o método sequelize.import foi removido. Esse método está sendo usado no projeto que tem para começar somente em um arquivo, que é o arquivomodels/index.js. Esse arquivo é um arquivo que o Sequelize cria automaticamente para gerenciarmos os modelos.

[03:10] Ele criou isso para nós lá no início do projeto anterior. Vamos atualizar nosso projeto agora para a versão 6 para sempre trabalharmos com a versão mais recente, e já corrigimos esse problema.

[03:22] Se viermos no terminal e digitarmos no terminal ‘npm outdated’ ele vai mostrar para nós que o Sequelize não está atualizado, e nem o sequelize-cli, que é a ferramenta de linha de comando do Sequelize. O que vamos fazer? Vou no package JSON, vou abrir o arquivo, atualizo a dependência do Sequelize de 5.21.7 para 6.3.0, que é a última versão até o momento em que este curso está sendo gravado. E o sequelize-cli vou atualizar também para a versão 6.2.0, que também é a última versão no momento em que estamos gravando este vídeo.

[04:15] Se voltarmos no terminal agora e atualizarmos tudo com ‘npm update’, ele vai atualizar, vai demorar um pouco talvez, costuma ser rápido. Olha que interessante que aconteceu. Eu estava já com o projeto rodando, e o nodemon já percebeu, que ele vai vendo e escutando as alterações imediatamente, e percebeu que tem uma linha, justamente a linha que usa o import, que quebrou.

[04:55] Aqui é onde vamos ter que consertar. Como eu já tinha dado uma olhada na documentação, eu já tinha visto onde temos que arrumar essa linha com import que está com problema. Só vamos trocar essa linha, vamos trocar onde está o sequelize.import por um código um pouco diferente, que vai ser um require(\*path\*.join(dirname, file))(sequelize, Sequelize.DataTypes).

[05:50] Agora eu salvei o arquivo. Se voltarmos no terminator, pronto, ele já está rodando. Já consertou onde estava quebrado. Como atualizamos também o sequelize-cli para a versão 6, da próxima vez que você precisar criar qualquer coisa com o sequelize-cli, usar a linha de comando, essa linha de comando também vai estar atualizada, corrigida. Agora o servidor já voltou a rodar normalmente. Projeto revisado, atualizado. Agora podemos realmente começar a mexer no código.

# Para saber mais: Requisitos do projeto

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/orm-nodejs-avancando-sequelize/task/79538/next)

Recebemos uma lista de funcionalidades que o cliente deseja implementar, agora que o CRUD básico foi feito e o sistema está funcionando.

O cliente não gostaria que registros importantes do sistema, como as Pessoas, sejam apagados definitivamente do banco de dados.

Para deixar a interface mais limpa, o cliente gostaria que na lista de Pessoas, por padrão, fossem exibidos somente os usuários ativos.

Foram percebidas algumas falhas de validação dos formulários por parte do front-end, o que resultou em dados de email inválidos no banco. É desejável que essa validação não seja responsabilidade exclusiva do front.

É importante poder consultar todas as matrículas confirmadas referentes a estudante X de forma rápida.

O cliente gostaria de poder consultar as turmas abertas por intervalo de data, para não receber informações desnecessárias (como turmas antigas).

O cliente quer poder consultar as matrículas por turma e saber quais delas estão lotadas, para organizar melhor as matrículas.

O cliente gostaria que, uma vez que o cadastro de um estudante fosse desativado, todas as matrículas relativas a este estudante automaticamente passassem a constar como “canceladas”.

Durante o projeto vamos analisar esta lista e transformar esses requisitos em novas funcionalidades.

[00:00] Temos uma lista com alguns requisitos que o cliente passou para nós. Vamos dar uma olhada neles e começar pelo primeiro. O cliente não gostaria que registros importantes do sistema, como por exemplo pessoas, sejam apagados definitivamente do banco de dados.

[00:18] Ou seja, apagar um registro do banco de dados é uma operação permanente e o cliente quer que isso não aconteça. Como fazemos para que um registro não seja verdadeiramente excluído do banco, mas mantendo para o usuário essa funcionalidade? Ou seja, o usuário continua achando que está apagando, mas o registro fica lá.

[00:35] Vamos aqui no terminal abrir o terminal do MYSQL, com MYSQL, meu usuário, minha senha de banco local. Estou usando o terminal do MYSQL, mas se você quiser usar um cliente, por exemplo MYSQL work bench ou The beaver, fique à vontade, que só vamos usar para consultar mesmo, então não vai fazer diferença você usar um cliente ou um terminal, etc.

[01:05] Vamos dar uma olhada então. Vou entrar no banco de dados que estamos usando, o escola\_ingles, e vou dar um select para vermos como estão nossas tabelas. Vou dar um select all from pessoas para dar uma olhada. Nós populamos no curso anterior a tabela e ela tem essas colunas.

[01:30] Teríamos que dar um jeito de marcar os registros que queremos apagar com uma espécie de bandeira, para que o sistema pense que esses registros estão deletados, mas ele não é apagado do banco, sem que a linha seja apagada do banco, lembrando que apagar uma coisa do banco é uma ação permanente.

[01:50] Esse recurso existe em SQL, e o Sequelize traz para nós também. Ele é chamado de exclusão suave, ou *soft delete*. Vamos dar uma olhada na documentação do Sequelize.

[02:03] No Sequelize essa opção chama paranoid. Conseguimos fazer a exclusão suave via Sequelize com relativamente poucas linhas. Ele está dando na documentação. Quando utilizamos um modelo, os modelos do sequelize têm um objeto onde colocamos os atributos e outro objeto onde vamos colocar a opção paranoid true.

[02:30] Vou copiar a linha. Essa opção é definida nos modelos, então podemos vir no nosso código, nos modelos. Vou colocar não só em pessoas, vou colocar essa opção para todos os modelos, porque temos poucas tabelas, então não tem problema.

[02:50] Aqui está como na documentação, o primeiro objeto são os atributos da tabela, da linha 3 até a linha 5, e o sequelize deixou até para nós um objeto vazio para colocar as opções. É aqui que vou colocar o paranoid true. Coloquei a opção na tabela matrículas, e vou colocar nas outras também, no modelo níveis, no modelo pessoas, e no modelo turmas.

[03:25] Enquanto o valor de paranoid for true, garantimos que nenhum registro da tabela vai ser verdadeiramente deletado. O Sequelize vai ser avisado quando tentar excluir qualquer registro.

[03:40] E como o paranoid funciona? Vamos voltar na documentação. A documentação diz que essa opção, vou descer um pouco nas informações que tem na documentação, quando usamos paranoid true, ao invés de deletar, de usar a palavra-chave delete quando o sequelize monta a query de SQL, vai usar um update. Não é mais delete, vamos continuar usando o método sequelize destroy, porém o que vai ser feito na query? Vai ser feito um update e aí o que vai acontecer?

[04:12] A query vai adicionar um *timestamp* numa coluna que chama deletedAt. Deletado em. Não temos uma coluna, se voltarmos no terminal do MYSQL, deletedAt, que ele precisa para fazer o paranoid funcionar. Temos que criar essa coluna para conseguir implementar o *soft delete*, porque da próxima vez que usarmos o destroyvai ser feito um update, colocado um timestamp no registro nessa coluna e o sequelize vai puxar como registro ativo só o que não tiver nenhum timestamp nessa coluna. Vamos ver isso um pouco mais para frente.

[05:02] Mas primeiro precisamos adicionar essa coluna deletedAt em todas as nossas tabelas. Já usamos o sequelize para adicionar colunas através dos arquivos de migração. Vamos abrir as migrações. Usamos cada uma delas para conectar com o banco e criar as tabelas com o método create table, tem os atributos id, etc, 0, updatedAt.

[05:00] Conseguimos usar o sequelize para depois você criar uma tabela, cada uma delas, conseguimos adicionar colunas também usando o Sequelize? Vamos ver isso agora em seguida.

[00:00] Vamos precisar adicionar uma coluna em cada uma das nossas tabelas do banco, mas de que forma podemos fazer isso? Dava para fazer direto no terminal do MYSQL com um comando que é para adicionar coluna mesmo, add column. Vamos com alter table Pessoas, para alterar a coluna pessoas, e em seguida passamos o comando add column para adicionar coluna. Quero que o nome dessa coluna seja deletedAt, é uma coluna com dado do tipo datetime, e quero que entre depois de updatedAt.

[00:46] É um comando curto, dava para rodar ele aqui mesmo, mas se formos pensar, quando criamos as tabelas no banco não criamos por aqui. Criamos utilizando as migrações. Criamos arquivos para criar cada uma das nossas tabelas e rodamos no terminal um comando de migração.

[01:06] Um dos motivos para fazer alterações no banco dessa forma, utilizando migração, é que ela fica na documentação de tudo que foi criado e alterado no banco, não só aqui, mas também fica guardada no banco. Vou quebrar o que fiz para não correr o risco de rodar isso sem querer e criar uma coluna no banco.

[01:25] Se viermos no terminal do MYSQL e der um show tables, mostrar as tabelas, a tabela sequelizeMeta é a tabela que está monitorando todas as alterações do banco. Se fazemos qualquer alteração que possa vir a quebrar alguma parte da aplicação, o que pode acontecer, conseguimos rodar outro comando e desfazer a alteração que foi feita no banco através da migração.

[01:55] Se viermos no código e abrirmos o código do arquivo de migração que usamos para criar a tabela pessoas, ela tem up, que é o comando que é rodado quando mandamos criar uma migração. E se precisarmos desfazer o que foi feito, tem o down que roda um outro método.

[02:15] No up ele usa o método create table e vai criar uma tabela com os parâmetros que passarmos. No caso aqui estou passando o primeiro parâmetro o nome da tabela e em seguida um objeto com os atributos dela. E se precisar desfazer, o método que vai ser rodado é o drop table que vai dropar a tabela pessoas para nós.

[02:35] Se víssemos no banco e rodássemos esse comando, ele iria criar, mas não iria ficar o registro, não ia conseguir desfazer também se fosse necessário com a facilidade que desfazemos se usarmos migração.

[02:50] Como usamos as migrações para ao invés de criar uma tabela criarmos uma coluna nova, por exemplo? Vou usar como base nosso arquivo create pessoas, que criamos anteriormente, porque o código vai ser parecido. Onde vamos trocar os métodos, não vai ser mais create table, mas vamos criar primeiro um arquivo novo na pasta de migração, o nome desse arquivo vai ser a data de hoje, não precisa ser exatamente a data de hoje, só precisa ser uma data posterior às migrações anteriores, e em seguida vou dar um nome, que vai ser addcolumn-pessoas.js.

[03:35] Vou colar o código que usamos para criar a tabela pessoas. O código é parecido, só que ao invés de create table, vamos usar um outro método que chama add column. O add column vai usar parâmetros um pouco diferentes. O primeiro vai ser uma string com o nome da tabela, que continua sendo pessoas. O segundo vai ser outra string com o nome da coluna que queremos criar, queremos criar uma coluna chamada deletedAt, e o terceiro parâmetro vai ser um objeto onde vamos passar as propriedades dessa coluna.

[04:15] Vou apagar o restante do código que não precisamos mais. Vou arrumar minha endentação. Quais propriedades vamos precisar na coluna deletedAt? A primeira é o tipo de dado, que vai ser date, sequelize.date, que no MYSQL ele vai converter para datetime.

[04:37] Outro que vamos precisar é o allowNull, de permite ser nulo. Quando não queremos que um campo nunca seja nulo, por exemplo, em um id, ele não pode ser nunca nulo, deixamos como false, mas aqui vamos deixar como true. Sim, ele pode ser nulo e vamos ver daqui a pouco por que isso é importante.

[04:58] Para desfazer se for necessário a migração, o método que vamos usar vai ser o removeColumn. E os parâmetros que passamos são o nome da tabela, tabela pessoas, e o nome da coluna que vamos remover, que é a coluna deletedAt se precisarmos desfazer a migração.

[05:16] Agora podemos replicar esse código para criar colunas também nas nossas outras tabelas através dos modelos. Podemos copiar e colar esse arquivo addColumn e só renomear ele com o nome das outras tabelas. Vou fazer isso também em níveis, troco o nome do arquivo e aqui dentro troco de onde nos parâmetros está pessoas para níveis. O resto permanece como está, porque o nome da coluna vai ser o mesmo.

[05:55] Vamos fazer a mesma coisa com a tabela turmas, e aqui dentro do arquivo também temos que passar turmas, ao invés de pessoas. E por último matrículas, que é a última tabela do nosso banco. Então, matrículas, addColumn, e onde está pessoas, matrículas.

[06:26] Agora já podemos vir no terminal e rodar o comando de migração. Comando npx sequelize-cli db:migrate. Vamos ver se vai rodar. Rodou tudo. O sequelize só faz as migrações que ainda não foram feitas, então não precisamos nos preocupar em passar nenhuma opção, ele vê automaticamente quais foram feitas e quais não foram.

[06:56] O que vamos fazer agora é voltar no terminal do MYSQL e ver se realmente ele criou a coluna. Podemos fazer isso dando um describe pessoas, por exemplo, e aqui ele criou a coluna deletedAt com o tipo de dado datetime, e sim, pode ser nulo. É só testar agora.

[07:20] Vamos testar no Postman. Vou vir em pessoas. Dar um get() em todas as pessoas, e vamos de repente usar o delete e passar um delete no id de pessoas 1. Vamos ver se ele deleta. Deletou, deu a mensagem id 1 deletado, que é o que já tínhamos deixado pronto o CRUD básico no curso anterior, mas vamos voltar no terminal onde estou rodando o nodemon e vamos ver qual foi a query que o Sequelize executou.

[07:55] Continuamos usando métodos destroy para deletar, mas como estava exatamente da forma que estava na documentação, ele agora faz um update e dá um set na coluna deletedAt. Foi exatamente como está na documentação. Se voltarmos no terminal do MYSQL e der um select all from pessoas, vamos ver agora o primeiro registro, de id 1, na coluna deletedAt ele tem uma data ao invés de nulo.

[08:40] Como sabemos que funcionou ou não no Postman? Vamos no get, na rota de pessoas, id 1 não aparece mais, é como se ele tivesse sido deletado. E se voltarmos no terminal onde estamos rodando o nodemon para ver qual é a query que foi executada, o where(), o onde, só está agora trazendo no getAll registros onde pessoas deletedAt, a coluna deletedAt de pessoas, é nulo, por isso que tínhamos nas propriedades que manter o nulo como true. Por definição, tudo que está deletado está na coluna deletedAt como null, e o que estiver deletado o sequelize vai mandar uma query para colocar um timestamp.

[09:35] Agora o sequelize automaticamente toda vez que faz uma query onde antes dava um getAll ele dá um get() em tudo que for null por definição. Agora então já fizemos nosso soft delete, nada mais é deletado em definitivo do banco, mas nosso usuário continua achando que está tudo deletado, porque o getAll está trazendo somente o que tem null na coluna deletedAt para ir para todas as nossas tabelas.

[00:00] Se o *Soft Delete* deixa o usuário só fingir que deleta um registro, entre aspas, então isso quer dizer que podemos restaurar um registro caso precisemos? Dá sim. Isso fazemos no controlador. Vamos abrir o controlador de pessoas, pessoaController, porque aqui estão os métodos que estamos usando para pegar coisas no banco, alterar coisas no banco, etc.

[00:30]. Vamos criar no nosso controlador de pessoas um novo método, vou colocar embaixo de apagaPessoa, vou criar um novo método e vou chamar de restauraPessoa, ele vai ser estático, async, e vai chamar restauraPessoa. Vai receber requisição e resposta como parâmetros.

[01:00]. O que vamos precisar para restaurar uma pessoa? Do id. Então, const { id } = req.params. Já podemos fazer nosso try catch, catch error, ele só dá um retorno. Vou copiar a linha de retorno do erro que estamos usando por enquanto a mesma linha. No erro ele vai retornar o status de 500 e vai passar em um JSON error.message para sabermos o que está errado.

[01:40]. No try vamos dar um await para esperar acontecer tudo que tem que acontecer no banco, database.Pessoas, como fizemos nos outros métodos, e o método que vamos usar é o restore. É o método do sequelize. O que temos que passar no restore para ele restaurar? Como parâmetro um where, então onde, abrimos outro objeto, passamos id igual, converte para número, id.

[02:22]. Estou passando id como parâmetro para o sequelize poder montar a query sabendo qual o registro que ele tem que fazer o restore, que ele tem que restaurar, e depois de tudo feito podemos dar um return e res.status, passa 200 de ok, e depois concatena com um JSON e dentro desse JSON vou passar uma mensagem dizendo que deu tudo certo. Mensagem, id, passo o número do id no template string, é restaurado.

[03:05]. Usamos o método restore do Sequelize, vai reverter o registro da situação de deletado no deletedAt. Vamos voltar no nosso terminal do SQL. Vou dar um select all from pessoas, onde temos o registro 1, que está deletado, e podemos agora restaurar ele.

[03:30]. Antes de testar no Postman, temos que voltar no código e adicionar uma rota para conseguir fazer isso. Vamos no arquivo de rotas, pessoas.route, e adicionamos a rota, que vai ser uma rota do tipo post. Vamos colocar um .post, a rota vai ser pessoas/id/restaura, vai ser o primeiro parâmetro. Não é necessariamente um estudante, então não vou colocar estudanteId, só id, e restaura.

[04:08]. O segundo parâmetro é o método que vai ser chamado, que é pessoaController.restauraPessoa. Já trouxemos a tabela com select. Vamos agora no Postman, nessa rota,/pessoas/id/restaura. Qual o id que tínhamos deletado? Id 1, então pessoas/1/restaura. Vou dar um post nessa rota. Vamos testar e ver o que acontece.

[04:48]. Mensagem id 1 restaurado. Vamos voltar no terminal do SQL e vamos dar um select de novo from pessoas. Nosso Id 1 voltou a ser nulo, ou seja, ele foi restaurado. Se voltarmos no Postman e dermos um get em pessoas, um getAll, o id 1 voltou a fazer parte dos registros ativos. Ele não consta mais como deletado.

[05:15]. Podemos adicionar esse método e essa rota para os modelos de níveis, turmas e matriculas, e o primeiro requisito da lista do nosso cliente, que é fazer uma exclusão que não exclua definitivamente do sistema, está feita, está finalizada.

# Faça como eu fiz

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/orm-nodejs-avancando-sequelize/task/79552/next)

Durante o vídeo criamos um novo método para restaurar registros que foram apagados da tabela Pessoas usando \_soft delete\_. Uma vez que adicionamos a opção { paranoid: true } em todos os modelos, vamos incluir também este método nos outros controladores: Níveis, Turmas e também as matrículas que estão no mesmo controlador de Pessoas.

api/models/PessoasController.js

static async restauraMatricula(req, res) {

const { estudanteId, matriculaId } = req.params

try {

await database.Matriculas.restore({

where: {

id: Number(matriculaId),

estudante\_id: Number(estudanteId)

}

})

return res.status(200).json({ mensagem: `id ${id} restaurado`})

} catch (error) {

return res.status(500).json(error.message)

}

}

COPIAR CÓDIGO

api/models/NivelController.js

static async restauraNivel(req, res) {

const { id } = req.params

try {

await database.Niveis.restore( {where: { id: Number(id) } } )

return res.status(200).json({ mensagem: `id ${id} restaurado`})

} catch (error) {

return res.status(500).json(error.message)

}

}COPIAR CÓDIGO

api/models/TurmasController.js

static async restauraTurma(req, res) {

const { id } = req.params

try {

await database.Turmas.restore( {where: { id: Number(id) } } )

return res.status(200).json({ mensagem: `id ${id} restaurado`})

} catch (error) {

return res.status(500).json(error.message)

}

}COPIAR CÓDIGO

E não podemos esquecer de atualizar os arquivos de rotas para que a API tenha uma rota de restauração de registro para cada uma das tabelas:

pessoasRoute.js

.post('/pessoas/:id/restaura', PessoaController.restauraPessoa)

.post('/pessoas/:estudanteId/matricula/:matriculaId/restaura', PessoaController.restauraMatricula)COPIAR CÓDIGO

niveisRoute.js

.post('/niveis/:id/restaura', NivelController.restauraNivel)COPIAR CÓDIGO

turmasRoute.js

.post('/turmas/:id/restaura', TurmaController.restauraTurma)COPIAR CÓDIGO

Agora é possível restaurar todos os registros de todas as tabelas.

# O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/orm-nodejs-avancando-sequelize/task/79554/next)

Nesta aula, aprendemos a:

* Atualizar a versão do Sequelize
* Adicionar a opção "Paranoid" para fazer a exclusão suave
* Criar migrações para adicionar colunas às tabelas
* Criar coluna deletedAt para utilizar o recurso de exclusão suave
* Restaurar registros deletados via exclusão suave, utilizando o .restore()
* Caso queira começar a partir deste ponto, aqui você pode baixar o que foi feito até a aula anterior. Não esqueça de fazer as instalações listadas no tópico “Preparando o ambiente”!
* [repositório da aula 1](https://github.com/alura-cursos/1862-sequelize/tree/aula-1)
* [00:00] Vamos ver agora qual é o próximo requisito na nossa lista. Vou dar um ok no item anterior, porque já fizemos, já fizemos o primeiro requisito, que foi o soft delete. Agora o segundo. Para deixar a interface mais limpa, o cliente gostaria que na lista de pessoas por padrão fossem exibidos somente os usuários ativos, faz mais sentido para o sistema dele.
* [00:25] Temos que fazer com que a lista de pessoas seja exibida. Vamos vir no Postman. Já dei um get aqui. Vamos dar um get em pessoas. Ele quer que essa lista que vai ser exibida no front apareça só com os registros que tem ativo true. Por exemplo, o id 4 que o ativo é false não quer que apareça por padrão.
* [00:55] Ele quer que todo get que façamos através do método do Sequelize findAll seja feita, podemos dizer, a partir de um determinado escopo. Vamos ver o que o Sequelize tem a dizer para nós sobre escopos.
* [01:14] A documentação dá alguns exemplos, os escopos são usados para reutilizar código, ou seja, se tem um requisito como esse, eu quero que sempre as queries select sejam exibidas com determinado where, ou determinado include. Esse escopo serve para podermos reutilizar isso.
* [01:41] Ele dá alguns exemplos. O escopo por definição é declarado no modelo. Então, se vamos determinar um novo escopo padrão, que ele marca inclusive como escopo padrão, defaultScope, temos que fazer isso no nosso modelo de pessoas. Vou fechar a lista de requisitos, vamos nos modelos, modelo pessoas, porque é aqui nesse modelo que vamos definir onde vai entrar esse escopo padrão e o que vamos ter que declarar nele.
* [02:15] Se voltarmos na documentação, no primeiro objeto dentro do método que aqui estamos usando o define, então sequelize.define é um método e dentro vai o parâmetro. O primeiro parâmetro é o nome do modelo, segundo parâmetro é um objeto grande com todos os atributos desse modelo, nome, ativo, etc, e o terceiro é outro objeto onde estamos colocando opções.
* [02:44] Já temos uma opção, que é o paranoid true. Vamos adicionar outra opção depois de paranoid, colocar uma vírgula. Aqui vamos passar o escopo. Primeiro, paranoid true, depois vamos passar um escopo que vamos chamar de escopo padrão, ou seja, de defaultScope.
* [03:08] Essa é uma propriedade do sequelize. O escopo padrão. O que queremos que aconteça no get() padrão? Vamos passar um where. Quero que todos os meus selects só retornem registros que tenham o atributo ativo, que é uma das colunas do meu banco, true. Ou seja, no caso do nosso projeto, o id 4 da tabela de pessoas que é a Sandra, que o ativo dela é false, não tem que aparecer por padrão, e mais nenhum que seja false também.
* [03:52] Vou salvar. O que fizemos aqui foi passar esse escopo padrão. Agora por padrão o sequelize vai incluir esse where em todas as queries feitas no modelo pessoas. Já podemos vir no Postman e testar nosso get. Agora o id 1, que é a Sandra, não tem que aparecer na lista. Passa direto do 3 para o 5, ou seja, está funcionando. O único registro que temos ativo false não está aparecendo.
* [04:21] Se fizermos uma consulta na tabela pessoas usando o terminal do MYSQL, vou dar um use escola\_ingles e se der um select \* from pessoas, o registro continua no banco, o ativo é 0, 0 é falso, mas o Postman não está trazendo, então está tudo certo.
* [05:00] Essa parte teoricamente já está feita, mas pode ser que queiramos em algum momento sobrescrever essa regra. Como faço para sobrescrever um escopo padrão? É o que vamos ver em seguida.
* Falando de produto, escopos definem funcionalidades e funções.
* Quando falamos de programação, existem diversos tipos de escopo. Por exemplo, em JavaScript podemos definir o escopo de uma variável - escopo global, de uma função, etc. O escopo é como chamamos as regras que definem quão acessível ou “visível” uma informação (por exemplo, uma variável) está, dependendo da parte da aplicação. Um escopo pode ter identificadores, alguma instrução de código ou algoritmo.
* No caso do Sequelize, podemos determinar o escopo padrão (defaultScope) que justamente define quais restrições e definições serão utilizadas na query… por padrão. E além do escopo padrão podemos definir outros, enquanto fizer sentido para a aplicação, e dar a cada escopo um nome que será usado pelo JavaScript para identificá-lo.
* Com isso é possível reutilizar código, definindo escopos para queries mais utilizadas e refinando estas queries através de palavras-chave como where, include, etc.
* [00:00] A propriedade defaultScope, escope padrão, que incluímos agora, como o próprio nome diz, define um escopo padrão, ou seja, se não indicarmos nada o Sequelize sempre vai usar esse where que colocamos para montar a query, montar a query select dele. Mas dá para ter quantos escopos o projeto precisar. Então vamos adicionar mais um para sobrescrever essa regra padrão caso seja necessário trazer todos os registros, inclusive os que tem ativo false.
* [00:33] Temos uma propriedade aqui que é a defaultScope e vamos incluir em seguida uma outra propriedade que vai chamar scopes. Aqui dentro vamos abrir outro objeto e nesse objeto conseguimos passar quantos outros escopos forem necessários.
* [00:55] Vou criar outro escopo que vou chamar de todos e dentro dele abro outro objeto e passo outro where, só que nesse caso o where que vou passar vai ser um where vazio, mas aqui em seguida depois de todos podemos ter quantos precisarmos, com todas as restrições, os constraints que precisarmos passar.
* [01:22] Aqui só temos que prestar um pouco de atenção na sintaxe. Temos essa propriedade defaultScope que define o padrão, e em seguida temos outra onde colocamos os outros. O nome defaultScope é um nome dado pelo sequelize, esse não podemos mudar, mas onde escrevi todos foi um nome que eu dei e os outros escopos que não são o padrão vamos chamar pelo nome.
* [01:52] No todos só passei um where vazio. Vamos então fazer algumas modificações no controlador onde estamos chamando os métodos e nas rotas para poder usar os dois casos na nossa API. Vou fechar o modelo pessoas, abrir o controlador de pessoas, e aqui o primeiro método do nosso controlador de pessoas é um método que chama pegaTodasAsPessoas, que por enquanto está fazendo um findAll em tudo que tem no banco, no modelo pessoas.
* [02:36] Vou modificar o nome, porque agora ele não vai mais pegar todas as pessoas, ele vai pegar todas as pessoas que estão ativas. Eu vou, na verdade, copiar todo esse método pegaTodasAsPessoas, colar embaixo, e comentar por enquanto, porque vamos precisar adicionar outro método para usar o outro escopo que acabamos de criar, o escopo todos.
* [03:00] O método que chamava pegaTodasAsPessoas agora chama pegaPessoasAtivas, continua recebendo requisição resposta, e a const que é todasAsPessoas que foi criada onde vai receber o retorno vai chamar pessoasAtivas, continua indo na database, continua indo em pessoas. E o findAll não tem modificações, porque agora devido ao escopo padrão, o findAll vai trazer só as pessoas no banco que tem ativo true.
* [03:41] Na nossa rota de pessoas vamos ter que alterar também, então vamos no arquivo pessoasRoute. Agora pegaTodasAsPessoas virou pegaPessoasAtivas, e o get() é simples, /pessoas, sem nenhum outro parâmetro.
* [04:04] Podemos testar agora, ver se está funcionando. Vamos no Postman, dá um get() em 3000 pessoas, o id 4, que é nosso id que tem ativo falso, continua não aparecendo, por enquanto está tudo ok.
* [04:22] Agora vamos criar mais um método, vamos voltar em pessoaController para poder usar o outro escopo todos que criamos. Vou descomentar o método pegaTodasAsPessoas que eu tinha deixado comentado, porque agora sim vamos pegar todas as pessoas. O que antes era pegaTodasAsPessoas virou pegaPessoasAtivas, e agora o pegaTodasAsPessoas vai usar o escopo que criamos.
* [04:46] Onde incluímos, como o sequelize sabe qual escopo ele tem que usar? Na linha onde declaramos o método findAll, que é o método que vai no banco pegar todos os registros, antes dele passamos .scope. Vamos dar um await database.Pessoas.scope(‘todos’).findAll(). findAll aqui não precisa passar nenhum parâmetro porque queremos que ele pegue tudo do banco mesmo.
* [05:42] Voltamos em pessoasRoute e criamos uma nova rota get(), para poder pegar todas as pessoas. Vou criar um novo get aqui. Vou passar /pessoas/todos, para ficar o mesmo nome que colocamos no escopo. Em português temos gênero nas palavras, acaba ficando um pouco menos intuitivo do que no inglês.
* [06:30] Adicionei uma nova rota get( /pessoas/todos), e o método que vamos chamar nessa rota vai ser pegaTodasAsPessoas. Já podemos testar no Postman, dar um get() em localhost:3000/pessoas/todos, e vamos ver se vem todos os registros, inclusive o 4. Vieram todos, inclusive nosso registro que está ativo false, que é o id 4.
* [07:06] Quando queremos que o sequelize use um escopo específico passamos o nome desse escopo, o nome que damos como parâmetro do método .scope e já encadeamos com o método que estamos usando o findAll, mas poderia ser qualquer outro, como findOne, por exemplo.
* [07:25] Nosso escopo já está definido, já temos o requisito cumprido, inclusive adicionamos um método caso seja necessário passar por cima, digamos assim, desse escopo padrão, e já podemos passar para os nossos próximos requisitos de projeto.