[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.
* [00:00] Vamos aproveitar que estamos fazendo algumas alterações no modelo e ver mais o que tem nos requisitos. Uma das funcionalidades que está sendo pedida aqui, a primeira sobre o soft delete já foi feito, a segunda já adicionamos um escopo de modelo para exibir as pessoas ativas por padrão, e em seguida o terceiro é sobre validação, foram percebidas algumas falhas de validação nos formulários por parte do front, o que resultou em dados de e-mail inválidos no banco e é desejável que essa validação não seja responsabilidade exclusiva do front, o que está muito certo, inclusive.
* [00:35] Essas validações, pelo Sequelize podemos fazer direto no modelo. Na verdade, existem algumas formas de se fazer validação em uma aplicação. Podemos tratar os dados recebidos no body, e aí dá para fazer esse tratamento direto usando JavaScript. Mas também dá para fazer com SQL, que é um comando que chama check.
* [01:00] Por exemplo, quando criamos uma tabela, damos um CREATE TABLE podemos usar um comando que é o check, e no caso queremos verificar matrícula, podemos verificar se os status das matrículas ou escrevem confirmado ou escrevem cancelado. Qualquer outra coisa que se tente incluir, qualquer outra string que não tenha nenhum desses dois valores ele não deixa entrar, dá um erro de validação.
* [01:33] Isso aqui é o MYSQL em ação, funcionando. E se conseguimos fazer validações no SQL, é bem capaz que o sequelize também cubra essa funcionalidade, e ele cobre. Chamamos isso de validação, e a parte de validações, validations e constraints do sequelize é onde encontramos as informações sobre como usar essa funcionalidade.
* [02:00] Na documentação do sequelize ele fornece para nós uma lista grande de validações possíveis que já deixa pronto para nós. Dá para validar se dá match numa expressão regular, se é float, se é decimal, se é upper case, se é data. Tem inclusive uma das validações possíveis se is credit card, que dá para validar o número de cartão de crédito, se é válido ou não, sem que precisemos usar uma biblioteca para isso ou escrever o algoritmo. E se precisar fazer alguma validação mais personalizada dá para fazer utilizando um if do JavaScript.
* [02:38] Os requisitos que recebemos pedem validação do campo e-mail, então podemos usar a opção de validação pronta que ele tem, que é is e-mail true. Essa validação é feita no modelo de acordo como está na documentação.
* [02:55] Vamos no modelo pessoas, vou fechar os requisitos e abrir o modelo de pessoas, e onde incluímos essa validação? A validação não é passada no objeto de opções, como fizemos com o paranoid e com os escopos. Passamos a validação direto no atributo.
* [03:18] O modelo pessoas tem nome, ativo, e-mail, role, se é estudante ou se é docente. No e-mail, onde atualmente ele só tem o valor de DataType.STRING é onde vamos fazer as alterações. Ao invés de DataType.STRING vamos abrir um objeto, e aí sim a primeira propriedade desse objeto vai ser type, que DataType.STRING precisamos continuar passando o tipo de dado que vai nesse campo.
* [03:50] Em seguida, vamos passar validate. Dentro de validate vamos abrir um outro objeto onde vamos incluir as validações que queremos que sejam feitas para o campo e-mail. Só queremos que seja feita uma, que chama is e-mail, de acordo com a documentação do sequelize. A validação que queremos fazer é se isEmail é true.
* [04:22] Se mantivermos dessa forma, só validate e o objeto isEmail true dentro ele funciona, mas podemos também passar algumas informações a mais. Vamos fazer dessa forma,isEmail ao invés de ter valor true vai ter valor de outro objeto, e dentro dele vamos passar args true, e msg, que é outra propriedade que podemos passar aqui, que é justamente uma mensagem que podemos retornar em caso de erro. Vou dizer na mensagem que dados do tipo e-mail inválidos.
* [05:10] Se passamos a validação dessa forma ela fica um pouco mais completa, conseguimos dar uma informação sobre o que está acontecendo. Não passou na validação, mas o que aconteceu? É importante às vezes passar uma mensagem, dar um aviso de por que não passou, o que está acontecendo quando uma coisa do sistema não vai como esperamos.
* [05:30] Podemos agora testar no Postman. Vou na rota localhost:3000/pessoas com post e vou criar uma nova pessoa. Toda pessoa tem nome, ativo, e-mail e role. Podemos inclusive copiar para não precisar escrever de novo. Nome vai ser João Carlos, o ativo pode ser true, role estudante, e no campo e-mail vou passar só uma letra, só jo, e ver se a validação está funcionando.
* [06:22] Vou dar send no meu post. Validation error, dado do tipo e-mail inválido, então meu validador está funcionando, ele bateu, não reconheceu o que passei, o string que passei como e-mail como um formato e-mail, mas se eu voltar na minha requisição e preencher para jo@jo.com, que é um formato de e-mail conhecido, e der um novo send, ele criou certo uma nova pessoa no banco, aceitou.
* [07:00] Está tudo funcionando, nossa validação está validando. No caso, os e-mails. Então não vamos mais ter esse problema de e-mails em formato errado. Lembrando que isso não é uma validação de e-mail que existe, aí é outro processo que normalmente é feito mandando e-mail de confirmação, etc, mas quanto ao formato da string já conseguimos cobrir essa validação. Agora podemos passar para os próximos requisitos.
* [00:00] Vamos aproveitar que estamos fazendo validações no modelo para adicionar mais algumas. No caso do nome, por exemplo, uma validação comum em nome é impedir a entrada de dados com menos de três caracteres. A pessoa tenta dizer que o nome dela é a e o sistema não deixa.
* [00:16] Dá para fazer isso usando um validador que já tem no Sequelize, que é o length, de tamanho da string, e aí você passa por parâmetro no valor uma array com tamanho mínimo e máximo da array de caracteres, mas também podemos criar uma função validadora um pouco mais específica, vamos ver como isso funciona.
* [00:42] Em nome, no atributo nome, onde está DataType.STRING como valor, vamos a princípio fazer o que fizemos no e-mail, abrir um objeto, o primeiro parâmetro vai ser type DataType.STRING, fizemos para validar o e-mail. E em seguida vou passar no validate também da mesma forma que fizemos no e-mail, vai receber como valor um objeto, e aqui podemos passar uma função validadora se quisermos fazer uma validação mais específica.
* [01:22] Vou mostrar para vocês como funcionaria isso. Damos um nome para essa função, vou chamar de funcaoValidadora, e é uma function, que vai receber o dado que queremos validar, abre e fecha chaves. Aqui dentro vou passar um if. Se o length, se o tamanho da minha string que estou tentando validar, que é a string do nome, for menor do que 3, eu vou lançar um erro, então posso dar um throw new Error. E passo nos parâmetros do erro o que quero voltar, o que quero jogar nesse novo erro, e vou passar uma mensagem mesmo, o campo nome deve ter mais de três caracteres.
* [02:40] Vamos testar com um nome fora do padrão. Venho no Postman, numa rota post, localhost:3000/pessoas com post, vou tentar mandar para o banco um novo registro com nome fora do padrão, com uma string de um caractere só. Dou um send no post. Erro de validação. Campo nome deve ter mais de três caracteres. Está funcionando.
* [03:11] Se eu voltar no meu corpo da requisição e refizer para Joana Carla, voltou a funcionar, criou um novo registro com nome, ativo, e-mail e role. A validação feita dessa forma via função validadora também está funcionando.
* [03:33] Os métodos de validação do Sequelize já cobrem grande parte dos casos, mas se você precisar e quiser usar uma coisa mais específica, é possível fazer isso com JavaScript puro, refinar, digamos assim, a validação, como fizemos usando um if e um operador de maior do que três, e a propriedade length de uma string. A parte de validação também agora já está feita e refinada inclusive.

A documentação do Sequelize utiliza o termo restrição (constraint). Constraints são como regras que regem as tabelas; são usados para limitar os tipos de dados que podem ser inseridos em uma tabela/coluna e garantir a integridade e confiabilidade dos dados que estão em um banco. Eles podem ser aplicados tanto em colunas individuais como de forma geral para toda a tabela. Eles podem ser definidos diretamente no modelo, como por exemplo, unique:true é um constraint que serve para garantir que o nome de User de quem for utilizar o sistema seja sempre único:

const Pessoa = sequelize.define('Pessoa', {

nomeUser: {

type: DataTypes.STRING,

unique: true

},

.

.

.

}COPIAR CÓDIGO

Para o exemplo acima funcionar, a tabela deve ser criada com Sequelize a partir desse modelo, ou já existir no banco com essa constraint. Se não for o caso, você pode utilizar o método [findOrCreate()](https://sequelize.org/master/manual/model-querying-finders.html" \l "-code-findorcreate--code-" \t "_blank), que verifica se o registro existe no banco antes de fazer qualquer outra ação. Existe uma diferença entre **validações**, que são feitas pelo Sequelize, em JavaScript, antes de qualquer query ser enviada para o banco, e constraints, que são regras definidas no SQL - tanto que podem ser definidos usando a sintaxe do SQL, como no caso da constraint NOT NULL abaixo:

CREATE TABLE Pessoa

(

ID int(6) NOT NULL,

NOME varchar(25) NOT NULL,

EMAIL varchar(20)

);COPIAR CÓDIGO

Diferente das validações, na verificação de constraints é executada uma query, e quem devolve o erro para o JavaScript é o SQL.

São constraints em SQL:

* NOT NULL - garante que não exista nenhum valor NULL na coluna;
* UNIQUE - Garante que não existam valores iguais em uma coluna;
* PRIMARY KEY - Identifica cada linha em uma tabela através de um valor único (junção de NOT NULL e UNIQUE);
* FOREIGN KEY - Identifica um valor único em outra tabela como chave;
* CHECK - Garante que todos os valores em uma coluna satisfazem uma condição específica;
* DEFAULT - Determina um valor padrão caso nenhum valor seja informado;
* INDEX - Para criar índices e facilitar o acesso a determinados conjuntos de dados.

Nesta aula, aprendemos a:

* Definir um escopo de modelo padrão (defaultScope)
* Definir outros escopos adicionais, conforme necessidade do projeto
* Utilizar um escopo adicional com o método .findAll()
* Validar dados de entrada utilizando validadores próprios do Sequelize
* Refinar e customizar validações de campos utilizando funções e JS puro
* [00:00] Vamos ver o próximo requisito da nossa lista. O último que lidamos foi a respeito de falhas de validação, que já está ok. O próximo que temos que tratar agora é que o cliente achou importante poder consultar todas as matrículas confirmadas referentes a um estudante de id X, de forma rápida.
* [00:18] Vamos pensar então em como fazemos isso. A primeira coisa que me ocorre é ir direto no controlador de pessoas e criar um método novo para isso. Podemos aproveitar o método apagaMatricula, que vai ser um pouco parecido. Vou copiar todo o método apagaMatricula, vou colar ele no final da classe pessoaController.
* [00:45] Esse método vai chamar pegaMatriculas, ao invés de apagaMatricula. Vai receber requisição e resposta. O que vamos receber aqui então nos parâmetros da requisição? Vai ser o id de estudante. Vai ser estudanteId, porque a partir do id de estudante é que vamos pedir lá na tabela de matrículas para ele filtrar todas as matrículas referentes a um estudante.
* [01:08] Dentro do nosso try vamos acessar database.Matriculas.findAll, e dentro do findAll vamos passar um where. Nesse where vai ter estudante\_id e o valor dele vai ser o valor que vamos trazer do parâmetro da requisição, que é estudanteId.
* [01:30] O que é esse estudante\_id que estamos declarando no where? Vamos voltar no diagrama de banco que usamos para criar nossas tabelas. Estudante\_id é uma coluna da tabela matrículas, é a coluna que liga a tabela matrículas à tabela pessoas através de uma chave estrangeira. Seria algo nessa linha, ele retornaria ao invés de uma mensagem um JSON com valor de matrículas, então aqui eu teria que criar uma const, que vou chamar de matrículas, para guardar os resultados desse findAll, e o JSON do retorno traria o conteúdo dessa variável matrículas.
* [02:18] Essa é uma forma de fazer. Mas tem outro jeito de fazer isso sem ficar passando where no findAll? Tem. A partir de um recurso que chamamos de escopo de associação. Se voltarmos no modelo pessoas, lembrando do curso anterior, nos modelos definimos algumas associações. Vamos ver como está essa parte de associações que é logo depois do código que escrevemos para definir os atributos do modelo e etc.
* [02:52] No modelo pessoas temos duas associações que já deixamos feitas, já trouxemos feitas do curso anterior. Tem pessoas.hasMany, que é uma associação de um para muitos de pessoas para turmas. E pessoas.hasMany, que é uma associação de pessoas com matrículas.
* [03:11] Por enquanto está dessa forma. A associação da tabela pessoas no modelo pessoas com turmas é a partir da coluna docente\_id, e de pessoas com matrículas é através da coluna estudante\_id, que foi a coluna que vimos no diagrama de banco.
* [03:30] Podemos a partir daí passar uma instrução nessa parte falando que já que esses dois modelos estão associados, conseguimos fazer eles se cruzarem? Por exemplo, uma função do tipo cruzar matrículas com pessoas, etc? Sim, conseguimos. Para fazer isso que servem justamente os escopos de associação.
* [03:52] Na documentação do Sequelize ele tem a parte de escopo de associação, é similar ao que fazemos anteriormente no escopo de modelo, só que aqui ao invés de definirmos o escopo dentro do próprio modelo, como fizemos anteriormente, se voltarmos um pouco no código, dentro do método define, que é o método onde definimos o modelo, passamos o nome dele e em seguida um objeto com os atributos, é dentro aqui desses parâmetros que passamos o escopo de modelo.
* [04:25] O escopo de associação definimos no trecho do código onde justamente declaramos todas as associações que são feitas com esse modelo. Vamos escrever para entender melhor o que está acontecendo. O que queremos associar é uma associação de pessoas com matrículas. Logo depois que escrevo foreignKey: estudante\_id vou adicionar mais duas propriedades nesse objeto.
* [04:50] A primeira propriedade vai ser scope, de escopo mesmo, o valor dela é um objeto. Dentro desse objeto vou passar justamente o escopo. Qual é o escopo dessa associação que quero fazer? É status confirmado.
* [05:07] O que é status confirmado, só lembrando? Se voltarmos no diagrama de banco, status é uma coluna da tabela matrículas e ele recebe confirmado ou cancelado. Como queremos nesse caso só filtrar os registros que tiverem status confirmado, passamos ele no escopo, e vamos passar outra propriedade, que é as, e ela vai ter um valor que vou passar como string que é o nome que quero dar para esse escopo também.
* [05:34] Vou dar para esse escopo o nome de aulasMatriculadas. Então, pessoas nesse trecho de código, pessoas.associate é uma instância do modelo pessoas que foi definido um pouco acima do código, a partir da linha 3.
* [06:00] O que acabamos de fazer é adicionar um escopo de associação passando duas informações. A primeira é o escopo que queremos trabalhar, que na verdade é a condição que vai ser passada no where quando o sequelize montar o array para nós, e em segundo o nome que queremos usar. E vamos usar esse nome para chamar esses tais métodos automáticos que fiquei me perguntando se o sequelize tem.
* [06:25] Vamos usar alguns desses métodos que ele cria para nós. Os nomes desses métodos são mixins. Também tem na documentação, vou deixar para vocês, como alguns casos de usos desses tais métodos que o Sequelize cria para nós a partir das associações. Vamos lá fazer esse escopo funcionar no controlador de pessoas.
* [00:00] Acabamos de ver que é possível definir escopos na associação entre modelos. O Sequelize gera alguns métodos automaticamente a partir desses escopos. Na documentação dele, ele mostra que no caso, por exemplo, de usuario.hasManyProjects, então usuário tem muitos projetos, ele cria automaticamente alguns métodos, por exemplo, getProjects, e também mais alguns que vou deixar na documentação para vocês.
* [00:30] Isso é ótimo porque ele reduz a quantidade de código que teríamos que escrever do zero para fazer algumas queries. Vamos usar um desses métodos, também chamamos eles de mixins, para consultar todas as matrículas confirmadas referentes a um estudante de id X.
* [00:44] No controlador de pessoas onde já tínhamos começado a escrever o método pegaMatriculas, vamos adicionar então um desses métodos prontos. Só que aqui, ao invés de consultar a tabela de matrículas vamos fazer um pouco diferente. Vou apagar a linha onde criamos essa const matrículas e vou criar uma nova const, vou chamar de pessoa, e await.
* [01:16] O que vamos consultar? Pessoas. Então database.Pessoas.findOne({where: {id: Number(estudante\_id)}}). Por enquanto é tudo que vimos no curso anterior. E aí após fazer isso precisamos pegar todas essas matrículas, precisamos fazer com que o sequelize percorra o banco e traga todas as matrículas confirmadas desse estudante.
* [02:15] Então vamos criar uma const matriculas = await pessoa, que é o retorno do findOne, que fomos no modelo pessoas. E olha o que vamos fazer. Vamos usar um método que chama getAulasMatriculadas, abro e fecho parênteses. O retorno desse await vai ser matrículas.
* [02:52] O que é esse getAulasMatriculadas? Vamos voltar no modelo pessoas, aulasMatriculadas é o nome que demos para o escopo. Esse é um comportamento que vemos em ORMs em geral, não é somente no sequelize. É a criação de métodos get, set, create, etc, a partir dos modelos e das associações.
* [03:12] Aqui estamos utilizando um desses métodos que o Sequelize cria automaticamente para nós. Ele pegou aulas matriculadas, que é o nome que definimos no escopo de associação, e criou automaticamente esse método, esse mixin, chamado getAulasMatriculadas, que como o próprio nome diz vai pegar todos os registros que estão de acordo com o escopo que passamos na associação que fizemos dentro do modelo pessoas, ou seja, tudo que tiver status confirmado.
* [03:40] Então vamos criar uma rota para testar se está tudo ok. Vamos abrir nosso arquivo de pessoasRoute, criar uma nova rota do tipo get. Vou organizar essas rotas todos os gets juntos, porque acho que fica mais organizado, e todos os puts, deletes, tudo junto, vou deixar mais organizado.
* [04:05] Vamos criar uma nova rota do tipo get, o primeiro parâmetro que vou passar é o endpoint, que vai ser pessoas/:estudanteId/matricula, e o segundo parâmetro é o método que vamos chamar nessa rota, que é PessoaController.pegaMatriculas. Vamos no Postman testar essa rota.
* [04:42] Vou copiar ela para não precisarmos escrever e vamos fazer um teste com o estudante de id 1, por exemplo. Vou em get no Postman, pessoas/1/matricula. Vamos ver se ele me retorna alguma coisa.
* [05:05] Ele retornou uma matrícula, só trouxe os registros do estudante 1 que tem status confirmado. É o id 1, o id de matrícula referente ao estudante id 1 que foi o estudante que buscamos.
* [05:16] Como sabemos que isso está certo? Podemos vir no terminal onde o node está exibindo as queries. Vou aumentar um pouco o texto. E olha a query que está sendo executada. No final da query dá para ver que ele procura todas as matrículas onde deletedAt é null, que já fizemos o soft delete, ele está buscando isso também, onde matriculas.status é igual a confirmado e matriculas estudante\_id é igual a 1.
* [05:50] Se voltarmos no terminal do MYSQL e pedirmos para ele fazer um select \* from matrículas, e pedir para ele exibir para nós, vamos ver que o estudante\_id tem uma outra matrícula. No seu banco você pode ter feito outras inserções, pode estar um pouco diferente, mas no meu banco tenho outro registro no id 5 que se refere também ao estudante de id 1, mas ela está cancelada. Então o Postman não trouxe para mim essa matrícula, está tudo certo.
* [06:38] Dá para usar também esses métodos automáticos sem passar nenhum escopo para eles? Dá sim. Os mixins são criados a partir das associações. Todas as associações que fazemos nos modelos, e temos associações nos nossos quatro modelos do projeto, eles criam automaticamente esses métodos e são super úteis para usar com o escopo, mas pode ser usado também sem eles.
* [07:02] Vamos tirar o escopo. Vou só comentar a lista onde passamos o escope e o objeto com status confirmado, só para testar e ver se está funcionando. Podemos voltar no Postman, fazer a mesma requisição, e ele traz as duas matrículas referentes a estudante id 1, tanto a confirmada quanto a cancelada.
* [07:25] O requisito de ter uma rota que traga todas as matrículas confirmadas a respeito de certo estudante, estudante x, já está feita. Vou tirar o comentário porque o escopo está valendo, e já podemos passar para o próximo requisito do nosso projeto.
* Podemos resumir mixins em: classes que contêm métodos que podem ser utilizados por outras classes, sem a necessidade de herança direta. Dito de outra forma, um mixin fornece métodos que implementam um certo comportamento em objetos, sem poder ser utilizado sozinho, mas sim para adicionar esse comportamento a outras classes.
* No Sequelize, temos uma diferença entre escopos de modelo, que são aplicados em chamadas estáticas ao modelo (como no exemplo que fizemos no vídeo, Pessoas.scope('todos').findAll()), e escopos de associação, que são uma regra, ou um conjunto de atributos que são automaticamente aplicados em instâncias do modelo, como em Pessoas.associate = function(models) {...}.
* Escopos de associação se comportam da mesma forma que os escopos de modelo, no sentido que ambos aplicam palavras-chave como WHERE em chamadas ao banco; mas os mixins são métodos que existem somente nas instâncias dos modelos: Pessoas.getPessoasMatriculadas, Niveis.getNiveisPorTurma, etc.
* A lista de métodos criados automaticamente com as instâncias de modelo são:
* addModel()
* addModels()
* countModels()
* createModel()
* getModels()
* hasModel()
* hasModels()
* removeModel()
* removeModels()
* setModels()
* Lembrando que “Model” e “Models”, aqui, refere-se ao nome do modelo! Lembre-se também que o Sequelize cria os nomes automaticamente mas não entende muito bem o singular e plural em português, mas você pode definir nomes personalizados para seus mixins, como fizemos no vídeo.

## Transcrição

[00:00] Passando então para o próximo requisito, as matrículas confirmadas referentes à estudante x já fizemos, então vou dar um ok. O próximo é que o cliente gostaria de poder consultar turmas abertas por intervalo de data, assim ele não recebe informações desnecessárias, como turmas antigas.

[00:15] Pensando bem, isso é bem útil, pensando que um escola em alguns anos pode ter uma quantidade grande de turmas, nem sempre quer receber informações de turmas que passaram faz tempo.

[00:25] Nós então precisamos passar um intervalo de datas para a API pegar esse intervalo de datas e fazer um filtro. Como fazemos isso? Passar um intervalo de datas. Uma forma muito usada para passar esse tipo de informação, digamos assim, mais supérflua, por exemplo, só um filtro que queremos usar, é através de parâmetros de query, as chamadas query strings.

[00:50] No próprio endpoint de turmas vamos dar um get em localhost:3000/turmas. Podemos passar aqui as datas para nossa API. Para fazer via query strings começamos abrindo as queries com um ponto de interrogação e vamos concatenando quantos parâmetros quisermos.

[01:10] Por exemplo, vou passar um que vou chamar de data\_inicial, e aí colocamos o nome que quisermos, igual, vou passar uma data, por exemplo, 2020-01-01, que é primeiro de janeiro de 2020. Vamos adicionando outras separadas por um &, vou passar meu outro parâmetro que vou chamar de data\_final, igual a 2020-03-01, que seria primeiro de março de 2020.

[01:40] Estamos passando via query strings duas datas. A data inicial, primeiro de janeiro de 2020, data final, primeiro de março de 2020. Essa é a forma que vamos fazer para passar essa informação que é só um filtro para nossa API.

[01:58] Como fazemos para a API pegar isso e usar em um where para filtrar e limitar os resultados de um get? Para casos como esse podemos usar operadores. Operadores são palavras reservadas do SQL mesmo, que usamos principalmente em where e de forma similar a outras linguagens. Temos operadores maior que, menor que, de comparação, módulo, para somar, dividir, são os operadores normais que usamos em outras linguagens, ele verifica a condição, usamos da mesma forma.

[02:30] A lista que estou mostrando na tela agora vou deixar o link para vocês, é a lista de operadores do MYSQL, e se existe no SQL existe no sequelize? Existe sim no sequelize uma lista de operadores prontos que ele traz para facilitar nossa vida. Na parte da documentação do sequelize de operadores podemos ver alguns deles.

[02:47] Ele tem alguns operadores básicos, se é, se não é, por exemplo, operadores de número, e tem maior que, menor que, menor que ou igual, maior que ou igual, etc. Operadores que funcionam com strings, com rejects, e etc.

[03:02] Se o caso aqui é pegar um intervalo entre duas datas, tem dois operadores de números que podemos usar. Maior ou igual a que, menor ou igual a que, e conseguimos estabelecer um intervalo entre. Podemos usar o between também.

[03:16] Vamos usar no nosso caso o maior que e o menor que para podemos usar dois e ver como funciona. A documentação também diz na parte de cima, no começo do código de exemplo que op precisa ser importado. O que vamos fazer agora é ir no controlador de turma, no início, nas primeiras linhas do controlador onde fazemos todas as importações, vamos importar via ‘const Sequelize = require(‘sequelize’). E aqui posso importar const Op = Sequelize.Op.

[04:10] Agora os operadores do sequelize ficam disponíveis para utilizarmos. Vamos então reescrever o método principal do meu controlador de turma, que é o pegaTodasAsTurmas, que podemos fazer ele funcionar, fazer o get em todas as turmas, com um filtro de datas ou sem, porque ele precisa funcionar passando datas ou não, ou pego realmente todas as turmas ou passo o intervalo de datas via parâmetro.

[04:36] Do jeito que está agora o controlador ele só faz uma consulta em database.turmas através do método findAll, então database.Turmas.findAll e findAll não está recebendo nenhum parâmetro. Então aqui, teoricamente, passaríamos um where, para ele ter parâmetros de busca.

[04:55] Vamos começar criando uma const que vou chamar de where e vou passar como valor um objeto vazio. Esse where vai para dentro do findAll como parâmetro, também dentro de um objeto. Agora findAll está recebendo um where que por enquanto é só um objeto vazio.

[05:20] Podemos vir no Postman. Vou apagar as datas que colocamos. Deixar só turmas. O get tem que continuar funcionando. Continua funcionando normal. Está pegando todas as turmas. Agora que já temos um where, temos que pegar os parâmetros da requisição.

[05:42] Logo na primeira linha depois que abrimos o método pegaTodasAsTurmas vou criar uma const, vou chamar de data\_inicial, data\_final, no objeto ele vai ser descontruído, igual a req.query, porque aí vamos pegar via query strings o todos os parâmetros que estão descritos com esses dois nomes que escolhemos, data\_inicial e data\_final, que são os nomes que passamos na rota.

[06:21] Agora temos que passar esses parâmetros para o where, se eles forem formados. O lance é dizer para o JavaScript que se tiver parâmetro de data, ele adiciona no where. Se não tiver, ele deixa quieto. Vou montar aqui logo abaixo do método só um modelo de como tem que ficar esse objeto where, porque vamos criar um where e dentro dele passar alguns parâmetros, isso tem que ser montado em forma de objeto.

[06:50] Vamos abrir chaves, o nosso where vai receber como valor outro objeto, e aqui dentro do where, inclusive se voltarmos na documentação do sequelize na parte de operadores, ele dá um exemplo de como funciona o where dentro do findAll. O where recebe um objeto e dentro desse objeto vou passando as colunas onde quero que esse where seja aplicado.

[07:18] A coluna onde queremos que o where seja aplicado é a coluna chamada data\_inicio. O que é data\_inicio? Lembrando no diagrama de banco que usamos, a tabela de turmas tem uma coluna que chama data\_inicio, que é onde estamos passando justamente a data de início de cada uma das turmas, isso fizemos no projeto passado.

[07:44] O where é um objeto e dentro dele a propriedade data\_inicio, é o nome da coluna que queremos aplicar o where, e o valor dela também é um objeto. Dentro desse objeto vamos passar os operadores que queremos usar. Vamos usar dois operadores, passamos os dois entre colchetes. O primeiro é Op.gte, "gte" vem de greater than or equal, maior que ou igual, e o valor dele vai ser uma data, uma data qualquer que vamos passar.

[08:20] O outro operador que vamos usar é Op.lte, lower than or equal, menor que ou igual. Ele vai receber uma data também. Essas duas datas que vamos passar são justamente as datas final e inicial que vamos receber por parâmetros.

[08:42] Vou comentar esse trechinho de código, porque ele é só um modelo para seguirmos e montar o objeto do jeito certo. E agora é passar isso para o código. Se ter data ou não ter data inicial é uma condição, podemos fazer alguns ifs dentro do nosso método pegaTodasAsTurmas para passar isso ou não para dentro do where.

[09:08] O primeiro if que vamos fazer é data\_inicial || data\_final e vou criar alguns ifs ternários aqui. Esse primeiro data\_inicial ou data\_final vai verificar se existe algum desses dois parâmetros. Se existir, vou no meu objeto where, que por enquanto o valor dele é um objeto vazio, e vou criar uma propriedade chamada data\_inicio, é o nome da coluna, lembrando bem. Vou passar como valor de data\_inicio igual a um outro objeto vazio.

[09:42] Se não tiver nada, vou passar null. E agora vamos levar esses dados que vamos receber para dentro desse objeto. Então, data\_inicial existe? Coloco a interrogação. É true? Se for true vamos passar para dentro do objeto where que criamos. Então, where.data\_inicio, o nome da coluna, e aqui vou usar a notação de colchete para passar Op.gte como uma propriedade dentro do objeto data\_inicio. O valor dessa propriedade vai ser o valor que vamos receber de data\_inicial. Se não tiver nada, null.

[10:30] Em seguida vamos fazer a mesma coisa para data\_final. Se existir, se recebermos um parâmetro com data\_final, é true? Interrogação. Vamos dentro do objeto where que criamos, dentro de data\_inicio, e vamos passar com a notação de colchete uma nova propriedade chamada Op.lte e o valor dessa propriedade vai ser o valor que vamos pegar do parâmetro data\_final. Se não tiver nada, null.

[11:11] Vou salvar e vamos testar das duas formas no Postman. Primeiro vou testar localhost:3000/turmas com um get normal. As turmas continuam vindo. E agora vou testar com a interrogação, passando data\_inicial igual 2020-01-01 e com & data\_final igual 2020-03-01. Vou dar um novo get. Ele continua funcionando.

[11:50] Acabou trazendo todas porque quando criei no meu banco criei todas com a mesma data de início, 2020, primeiro de fevereiro, então ele continua trazendo tudo, mas vou passar a data final como 2020-01-31. Vamos ver se ele vai trazer.

[12:10] Ele não traz nada, porque todas as minhas turmas começam em fevereiro. É capaz de que no banco de vocês quando vocês tiverem feito vários testes vocês tenham outras informações e vocês podem modificar. Vou fazer um último teste, voltar a data final para 2020-03-01, vou passar a data inicial para 2020-02-01, e ele continua trazendo todos os registros que tem no meu banco.

[12:41] Nosso filtro está funcionando, já podemos passar para o próximo requisito do projeto.

[00:00] Partindo então para a próxima solicitação do cliente, a última que fizemos foi fazer uma espécie de filtro de data utilizando ordenadores e datas que recebíamos por query, esse já está feito. O próximo é que o cliente quer poder consultar as matrículas por turma e saber quais delas estão lotadas para organizar melhor as matrículas.

[00:20] Supondo que passemos um número, por exemplo, a partir de quatro estudantes uma turma está lotada, precisamos voltar a isso do banco para o sistema. Vamos pensar primeiro no primeiro caso, o cliente quer poder consultar as matrículas por turma. Vamos ver isso primeiro.

[00:40] Vamos então no controlador de pessoas. No controlador de pessoas porque é neste controlador que estão sendo escritos os métodos de matrícula, porque o que temos que consultar, se voltarmos no diagrama de banco, é a tabela matrículas e ver quantas matrículas tenho em uma turma através da coluna turma\_id, que é a coluna que liga a tabela de turmas com a tabela de matrículas.

[01:05] O último método do nosso pessoaController é o pegaMatriculas, vou copiar ele e colar por último o método para reaproveitarmos um pouco do que já escrevemos. Vou chamar esse novo método de pegaMatriculasPorTurma. E o que vamos precisar receber dos parâmetros? Vamos precisar receber turmaId, porque, queremos checar quantas matrículas tenho por id de turma.

[01:35] Vou apagar por enquanto tudo que está dentro do try, porque vamos escrever novamente. E como fazemos isso? Eu poderia usar só um findAll, por exemplo, database.Matriculas.findAll, e passo o findAll, ele traz todos os registros, passo dentro do where o turma\_id, ele traz todos os registros que tem com aquela turma na tabela de matrículas, e posso deixar o front fazer essa contagem de registros.

[02:11] Eu posso, mas não é a melhor prática, uma vez que o SQL tem recursos para fazer esse tipo de operação, e ele faz de forma mais performática. Então, já que o SQL oferece recursos para fazermos contagem de registros, soma de registros, etc, vemos se o sequelize tem alguma coisa parecida.

[02:28] Ele tem. Na verdade, o que queremos é um método tipo find, um método encontrador, além dos que já usamos. Usamos o findAll, usamos o findOne, além disso tem o findByPk, que encontra por uma chave primária, e tem o findOrCreate, se você precisar de algum registro que seja único, por exemplo, um id, checa se tem um id ou username no banco, se não tiver ele cria.

[02:52] E temos um que parece promissor, que chama findAndCountAll, encontre e conte todos. Esse parece que vai ser bom para nós. Vamos aproveitar no nosso método. Vou voltar onde começamos a escrever o método pegaMatriculasPorTurma, vou criar uma const para salvar os resultados que vamos trazer do banco, vou chamar de todasAsMatriculas, ela vai ser await database.Matriculas.findAndCountAll, esse é o método que vamos usar.

[03:38] O método findAndCountAll vamos passar um where para ele como parâmetro, para podermos dizer para ele que queremos que ele veja tais turmas, etc. Dentro do where, o que temos que passar primeiro no findAndCountAll? A primeira coisa que vamos ter que passar é, vou abrir um objeto, a propriedade where, o valor dela é um outro objeto onde passamos o que queremos colocar no onde, no where.

[04:05] O primeiro vai ser turma\_id, porque turma\_id é nome da coluna lá no diagrama de banco, e o que vou passar no turma\_id? Vai ser number, para converter para número, turmaId, e esse turmaId é o turmaId que estamos recebendo dos parâmetros.

[04:26] Vou passar mais uma coisa nesse where. Vou passar status: ‘confirmado’. O que é esse status confirmado? Se voltarmos no diagrama de banco, status é uma outra coluna na tabela matrículas, e ele recebe uma string confirmado ou cancelado, porque o cliente quer que tragamos todas as matrículas que estão confirmadas em determinada turma, então não podemos nos esquecer dessa parte.

[04:55] A última coisa que vou colocar é um return, para podermos mandar esses dados para a frente. Então, return res.status(200).json(todasAsMatriculas). Agora esqueci de adicionar uma rota para esse teste, então vamos na pasta de rotas, na rota de pessoas, vai ser uma rota do tipo get, então router.get (‘/pessoas/matricula/:turmaId/confirmadas’, PessoaController.pegaMatriculasPorTurma).

[06:16] Vou copiar essa rota para testarmos ela no Postman. Vamos lá no Postman, get localhost:3000/pessoas/matricula/, preciso passar um id de turma. Vamos dar uma olhada no terminal do MYSQL para ver o que podemos colocar como teste. Vou pedir então select \* from Matriculas.

[06:50] Na turma 1, no meu banco, que o de vocês pode estar um pouco diferente, tenho duas matrículas confirmadas na turma de id 1, do estudante 1 e do estudante 2. Vamos no Postman dar um get e ver se ele vai retornar na turma 1, que é a turma que tem duas matrículas confirmadas.

[07:20] pessoas/matricula/1/confirmadas. Ele me trouxe as duas matrículas, a matrícula de id 1 e de id 2, ambas referentes à turma 1. Id de estudante 1, e id de estudante 2. Olha que interessante, olha o que o JSON está retornando para mim. Ele está retornando um objeto com duas propriedades. A primeira chama count, e ela tem um valor número de 2, e a segunda chama rows, de linhas, e o valor dela é uma array contendo os resultados que bateram com o where.

[07:52] Ele primeiro conta todos os registros, ele contou, trouxe para nós que são dois registros, e em seguida ele trouxe um array com esses registros. Podemos voltar no nosso método para fazer um teste e pedir para o JSON retornar todas as matrículas .count. Aí vamos no get de novo no Postman. Ele trouxe só o valor que tem dentro de count.

[08:16] Também podemos adicionar como parâmetro do findAndCountAll algumas outras opções. Vou adicionar duas para testar. Uma que é o limit. O limit é bem útil para fazer paginação. Podemos limitar a quantidade de registros que são trazidos por vez do banco.

[08:37] Não vou colocar 20, que é o mais padrão, porque só temos dois registros no banco. Vou colocar um registro só para testarmos. Vamos voltar ao Postman, vou dar um send de novo aqui. No count ele contou que tem dois registros, mas ele trouxe na array de resultados só um, só o primeiro.

[08:58] Podemos usar o limit quando tivermos vários registros e quisermos limitar a quantidade de registros que vêm de uma vez. Vou deixar 20 só para ter guardada essa opção. E uma outra opção que podemos trazer também é ordenar, order. Com order ordenamos os resultados. Passamos a forma como queremos ordenar dentro de duas arrays e vou passar aqui dentro, no primeiro índice, a coluna que quero ordenar, pela qual quero ordenar. Vou ordenar por estudante, estudante\_id é a coluna que quero usar para ordenar.

[09:33] O outro índice dessa array é se quero ordenar de forma ascendente ou descendente. Se colocar as letras ASC vou ordenar por ordem ascendente. No meu retorno não vai fazer diferença, porque estudante 1 e estudante 2 estão logo em seguida um do outro. Mas se eu trocar por DESC e pedir os resultados de novo no Postman, ele trocou, primeiro traz o estudante de id 2 e depois o de id 1.

[10:05] Já vimos como utilizamos alguns outros métodos do tipo finder, e como podemos passar algumas outras opções além do where para organizar melhor os resultados que mandamos para frente, que trazemos com a nossa API.

[00:00] Para finalizar esse requisito, vamos precisar de algo um pouco mais específico, que é contar quantos registros temos da mesma turma, numa matrícula. Se voltarmos no terminal do MYSQL, temos que contar quantas matrículas temos referentes ao mesmo id de turma, então no meu banco, por exemplo, tenho duas confirmadas na turma de id 1, comparar com o número máximo que se pode ter de turmas com matrículas confirmadas, que precisamos passar isso, e exibir só os registros que passem dessa comparação.

[00:33] Se uma turma lotada for uma turma com duas matrículas, tenho que exibir só o que passa. Primeiro, vamos criar um método, voltamos no nosso controlador de pessoas, que é onde estamos trabalhando com matrículas, com o modelo de matrículas. Vou copiar o último método, que é o pegaMatriculasPorTurma e vou colar de novo por último para podermos trabalhar em cima.

[00:56] Esse método que vamos criar vou passar o nome dele como pegaTurmasLotadas, vai receber requisição e resposta. Vou apagar o que tem no try, para podermos reescrever. O try catch pode deixar. O que vamos passar primeiro? Primeiro temos que passar justamente a quantidade de registros que fazem com que uma turma seja lotada.

[01:25] Então vou criar uma const chamada lotacaoTurma, vou passar um valor como 2, por exemplo. Uma turma com 2 matrículas é uma turma lotada. O método que vamos usar é o mesmo que usamos, é o *find and count all*, porque o sequelize vai ter que fazer isso mesmo, vai ter que ir no banco, encontrar registros e contar quantos tem.

[02:00] Vou criar uma const turmasLotadas, igual await database.Matriculas.findAndCountAll(), aqui dentro o que vamos passar? Vamos passar de novo um where. Vou jogar para a linha de baixo para ficar mais fácil de ver.

[02:40] O que vamos passar dentro desse where? status: ‘confirmado’, que só queremos que traga as matrículas que têm status confirmado. O retorno já deixo pronto também, res.status(200).json(turmasLotadas), vai devolver o que estiver dentro dessa variável.

[03:08] Se já criarmos uma rota para teste podemos ir vendo o que vamos receber. Vamos em pessoasRoute, mais uma rota do tipo get, então aí ser router.get(‘/pessoas/matricula/lotada’), porque dessa vez não precisamos passar nenhum parâmetro, porque não vou passar qual turma que quero consultar, quero que o banco, o sequelize consulte o banco e traga o que tem de informação sobre turmas lotadas.

[03:42] O segundo parâmetro é o método PessoaController.pegaTurmasLotadas. Já está pronto. Vamos ver o que retornamos aqui no Postman. Vou copiar a rota /pessoas/matricula/lotada, get, vamos ver o que aparece. Ele já trouxe todas as turmas confirmadas e contou quantas turmas têm, lembrando que por enquanto não estamos falando nada de lotação ainda, a única coisa que fizemos no nosso método foi criar uma const que ainda não está sendo usada que passa a lotação de turma.

[04:28] Quantas turmas tenho no banco com registro, com status confirmado? Tenho 4, ele contou para nós. Vamos precisar de mais alguns recursos do SQL, porque vamos precisar agrupar os registros que têm o mesmo valor, ou seja, que tem o valor de confirmado, e contar os registros.

[04:50] Se você já viu um pouco mais sobre queries com SQL vamos precisar de duas palavras-chave que são group e having, que são agrupar e ter. Se tem no SQL, mesma coisa de sempre, vamos na documentação do Sequelize, porque conseguimos bem provavelmente fazer direto com o Sequelize.

[05:11] Na documentação ele tem uma documentação sobre agrupamento, sobre group, então podemos usar o que tem na documentação, que é passar dentro de um objeto a propriedade group e por onde. Vamos no where, embaixo de status confirmado, depois de where.

[05:32] Da mesma forma que fizemos vídeo passado, depois de where vamos passar como segunda propriedade desse objeto, a primeira propriedade é where, o segundo vou passar attributes: [‘turma\_id’], e vou passar também uma outra propriedade, que é group: [‘turma\_id’].

[06:14] Primeiro, o que estamos fazendo? Passando em attributes só o atributo do modelo que queremos trabalhar, e em seguida falamos qual é o atributo que vamos usar para agrupar, ou seja, junte os resultados checando o valor desta coluna, da coluna turma\_id.

[06:35] Vamos ver o que então está acontecendo no Postman. Vou dar um send. Ele já está contando. Está quase lá. Ele está contando, está trazendo a turma\_id, count tenho duas matrículas na turma\_id já confirmadas. Na turma de id 2 tenho uma e na turma de id 3 tenho outra. E aqui ele está trazendo só os rools que tem turma\_id, porque são os atributos que queremos trabalhar. Pedimos para trabalhar só com o atributo turma\_id e é só isso que ele está retornando.

[07:10] Está quase lá. Ele trouxe a quantidade de registros. Só falta contar e comparar. Aqui vamos precisar fazer uma coisa que foi inédita até agora neste projeto, que é realmente escrever SQL, porque batemos numa coisa que o Sequelize não consegue fazer para nós sem SQL.

[07:30] Como fazemos para inserir queries de SQL, palavras-chave dentro do Sequelize, do node através do sequelize? Vamos usar um método que chama sequelize.literal.

[07:44] A primeira coisa que vamos ter que fazer é vir nas primeiras linhas do controlador de pessoas, porque vamos precisar importar sequelize para poder usar esse método, então const Sequelize = require(‘sequelize’). Agora podemos voltar no método e passar depois de group uma quarta propriedade, que é having, que tem, tendo.

[08:30] E aí, having: Sequelize.literal(‘count(turma\_id) >== ${lotacaoTurma}’). O que estou fazendo? Estou pedindo para ele contar quantos resultados de turma\_id são maiores, então quantas matrículas de uma certa turma, de um certo id de turma que tem uma quantidade maior do que o número que passei na const lotacaoTurma, que por enquanto eu passei dois.

[09:35] Vamos testar? Vou voltar no Postman, send, ele não trouxe nada, porque tem que ser maior ou igual, passei dois e ele tem dois registros no banco. Agora deve voltar. Na turma\_id de número 1 tenho duas turmas e ele está retornando isso para nós. Ou seja, para fazer esse where alinhado, entre várias aspas, passamos como SQL mesmo, e pedimos para incluir na conta apenas os registros de turma\_id, com quantidade de registros maior ou igual do que o inteiro que passamos na const lotacaoTurma.

[10:17] Vamos voltar no terminal onde o node está mostrando as queries para nós e vamos mostrar essa query, vamos ver qual foi a query que o Sequelize montou para nós. Ele está agrupando por turma\_id tendo um count, está montando exatamente com o que pedimos, ele agrupou por turma\_id e podemos ver que ele passou a palavra-chave count do jeito que fizemos para ser incluído na query.

[10:51] O ideal aqui então é devolver esse JSON somente com o count, que é o que queremos. Queremos passar a quantidade de ids que estão lotados. Podemos voltar, se quisermos, e passar em turmasLotadas.count no retorno, ele vai trazer para nós só uma array com a quantidade de turmas que está lotada. Como só temos uma turma, ele só traz um registro, mas vai colocando todos os registros que tiver.

[11:20] Agora já sabemos como fazer para incluir, fazer contagem de registros, e também como incluir literais do SQL dentro do sequelize para fazer coisas um pouco mais complexas.

Agora que estamos acrescentando alguma complexidade às queries que o Sequelize vai passar para o SQL, é interessante relembrar que existe uma ordem de execução para os operadores e cláusulas.

No caso de queries de SELECT, a ordem lógica é a seguinte:

FROM: pega as tabelas onde estão os dados

WHERE: filtra os dados

GROUP BY: agrega os dados

HAVING: filtra os dados agregados

SELECT: retorna os resultados

ORDER BY: ordena os resultados

LIMIT: limita a quantidade de resultados

Ou seja, cada query começa encontrando os dados, filtrando e ordenando. Essa ordem pode fazer com que certos resultados sejam ou não acessíveis em dado momento. Por exemplo, a cláusula WHERE é executada antes de GROUP BY, então não podemos depender de dados retornados pelo GROUP BY para então passar WHERE.

Porém, os DBMS (Database Management Systems) como MySQL, PostgreSQL, MSSQL, entre outras, utilizam database engines, ou algo como “motores de database” numa tradução mais literal, para executar as queries. Esses engines, na prática, reorganizam a ordem lógica acima para otimizar as queries e deixá-las mais rápidas e com melhor performance, **enquanto essa reorganização não modificar os resultados da query**. Os database engines também fazem algumas verificações para garantir que a query faça sentido como um todo antes de fazer essa reorganização e rodar qualquer coisa.

Assim, embora exista uma ordem lógica na execução de uma query SELECT, e seja uma boa prática nos basearmos nela, na prática não temos realmente como saber qual é a ordem que será efetivamente utilizada, pois isso vai depender de como cada engine vai calcular a forma mais otimizada de execução da query.

A Julia Evans tem um [artigo](https://jvns.ca/blog/2019/10/03/sql-queries-don-t-start-with-select/) (em inglês) que explica esta questão de forma bastante visual.