07

# Faça como eu fiz: CRUD de Turmas

Agora que já sabemos como funciona o `SQLDataSource`, podemos adiantar outros métodos para o CRUD de Turmas para focarmos em assuntos específicos do GraphQL nas próximas aulas. Será similar ao CRUD que fizemos para o tipo `User`, porém agora todos os métodos se baseiam em *queries* de SQL, que vamos escrever em JavaScript utilizando os métodos da biblioteca Knex.

> Você pode ir direto para os arquivos turma/schema/turma.graphql (https://github.com/alura-cursos/1982-graphql/blob/aula-1/api/turma/schema/turma.graphql), turma/resolvers/turmaResolvers.js (https://github.com/alura-cursos/1982-graphql/blob/aula-1/api/turma/resolvers/turmaResolvers.js) e

turma/datasource/turma.js (https://github.com/alura-cursos/1982-graphql/blob/aula-1/api/turma/datasource/turma.js) já atualizados com o CRUD completo. Ou pode seguir o passo-a-passo abaixo, com explicações sobre cada parte que modificamos, **para internalizar o fluxo: definir no schema > implementar no resolver > criar o método no datasource para conectar à base de dados > testar no playground.**

Para saber mais sobre o que é SQL, como funcionam as queries e a persistência de dados utilizando bancos relacionais, você pode conferir o curso de modelagem de banco de dados relacional (https://cursos.alura.com.br/course/modelagem-banco-relacional-sql) ou alguns dos cursos introdutórios de SQL com os gerenciadores mais utilizados: Postgres (https://cursos.alura.com.br/course/introducao-postgresql-primeiros-passos) e MySQL

(https://cursos.alura.com.br/course/mysql-manipule-dados-com-sql), entre outros cursos e formações neste tema.

## Buscar um registro de turma

Esta query já foi criada anteriormente, então só temos que implementá-la.

Atualizando o resolver de turmas:

```
turma: (_, { id }, { dataSources }) =>
```

COPIAR CÓDIGO

O parâmetro `{ id }` é o que a documentação do GraphQL chama de `args`, os dados que recebemos da query. Estes dados sempre são recebidos como um objeto, do qual queremos apenas a propriedade `id`.

Este método recebe como parâmetro um número de id que é passado pela query do GraphQL. O retorno destes métodos do Knex é sempre uma array, mesmo que tenha somente um índice.

```
async getTurma(id) {
  const turma = await this.db
    .select('*')
    .from('turmas')
    .where({ id: Number(id)})
  return turma[0]
}
```

COPIAR CÓDIGO

Convertendo para a linguagem de query do SQL:

```
SELECT * FROM turmas WHERE id = ?
```

COPIAR CÓDIGO

Onde `?` representa um valor recebido por parâmetro.

Você pode testar no playground com uma query similar a esta:

```
query {
  turma (id: 1) {
    descricao
  }
}
```

COPIAR CÓDIGO

## Criando mutations para alterações de dados

É bom lembrar que, para fazermos qualquer tipo de alteração no banco (qualquer operação que não seja uma consulta) o tipo-raiz utilizado é `Mutation` e não `Query`. Então, antes de qualquer coisa, vamos adicionar *no schema de Turmas* um novo `type Mutation`:

```
type Mutation {
  #aqui vão as mutations para incluir,
```

```
}
```

COPIAR CÓDIGO

Para deixar os testes mais ágeis, podemos definir também um tipo `Input` de Turmas:

```
input TurmaInput {
  descricao: String
  horario: String
  vagas: Int
  inicio: DateTime
  docente_id: Int
}
```

COPIAR CÓDIGO

No input acima, temos o campo `inicio` com um valor de `DateTime`. Como fizemos em `userResolvers.js`, precisamos resolver o custom scalar `DateTime` *também em* `turmaResolvers.js` para que se converta em um formato válido de data:

```
//importar módulo no início do arquivo
const { GraphQLScalarType } = require('

const turmaResolvers = {

 DateTime: new GraphQLScalarType({
   name: 'DateTime',
   description: 'string de data e hora
   serialize: (value) => new Date(value
   parseValue: (value) => new Date(valu
   parseLiteral: (ast) => new Date(ast.
 }),

 //restante do código (Query e Mutation
}
```

COPIAR CÓDIGO

Agora podemos criar as mutations.

## Adicionar um registro de turma

Começamos por adicionar esta mutation ao schema de turmas:

```
type Mutation {
  incluiTurma(turma: TurmaInput): Turm

}
```

COPIAR CÓDIGO

Atualizando em `turmaResolvers.js`:

```
Mutation: {
  incluiTurma: (_, {turma}, { dataSour
},
```

COPIAR CÓDIGO

Este método recebe como parâmetro um objeto contendo os dados da nova turma. Após inserir o novo registro no banco, utilizamos o método `returning` para retornar o `id` desse registro recém-criado; com o `id` podemos trazer de volta os dados da nova turma e passar para o GraphQL.

```
async incluiTurma(novaTurma) {
  const novaTurmaId = await this.db
```

```
      .insert(novaTurma)
      .returning('id')
      .into('turmas')

    const turmaInserida = await this.get
    return ({ ...turmaInserida })
  }
```

COPIAR CÓDIGO

Convertendo para a linguagem de query do SQL:

```
INSERT INTO turmas (descricao, docente_

SELECT * FROM turmas WHERE id = ?
```

COPIAR CÓDIGO

Faça o teste no playground com a query:

```
mutation {
  incluiTurma(turma: {
    descricao: "avançado"
    horario: "noturno"
```

```
    vagas: 5
    inicio: "2020-12-01"
    docente_id: 5
  }) {
    descricao
  }
}
```

COPIAR CÓDIGO

## Alterar um registro de turma

Começando pelo schema, criando o campo `atualizaTurma`:

```
type Mutation {
  incluiTurma(turma: TurmaInput): Turm
  atualizaTurma(id: ID!, turma: TurmaI

}
```

COPIAR CÓDIGO

Atualizando no resolver de turmas:

```
atualizaTurma: (_, novosDados, { dataSo
```

COPIAR CÓDIGO

Veja que, na mutation acima, passamos para a frente o objeto `novosDados` completo, com as duas propriedades internas ( `id` e `turma` ).

Este método recebe como parâmetro um objeto com dois dados diferentes: o `id` do registro que será alterado e outro objeto contendo os novos dados:

```
async atualizaTurma(novosDados) {
  await this.db
    .update({ ...novosDados.turma })
    .where({ id: Number(novosDados.id)
    .into('turmas')

  const turmaAtualizada = await this.g
  return ({
    ...turmaAtualizada
```

```
    })
  }
```

COPIAR CÓDIGO

Um exemplo do formato do objeto recebido através do parâmetro `novosDados`, que fará uma alteração no registro de `id 4` (você pode fazer um teste em seu projeto adicionando `console.log(novosDados)` na primeira linha do método):

```
{
  id: '4',
  turma: [Object: null prototype] {
    descricao: 'super avançado',
    horario: 'noturno',
    vagas: 5,
    inicio: 2020-12-01T00:00:00.000Z,
    docente_id: 5
  }
}
```

COPIAR CÓDIGO

No formato de query do SQL, a operação fica da seguinte forma:

```
UPDATE turmas SET descricao = ?, horari
SELECT * FROM turmas WHERE id = ?
```

COPIAR CÓDIGO

Você pode testar no playground com a query:

```
mutation {
  atualizaTurma(
    id: 4,
    turma: {
        descricao: "super avançado"
        horario: "noturno"
        vagas: 5
        inicio: "2020-12-01"
        docente_id: 5
      }) {
    descricao
  }
}
```

COPIAR CÓDIGO

Supondo aqui que o seu banco também tenha um registro de `id` 4! Você pode adaptar a query aos registros que tenha feito na tabela `turmas` do seu projeto.

## Deletar um registro de turma

Vamos adicionar este último campo ao tipo `Mutation`:

```
type Mutation {
 incluiTurma(turma: TurmaInput): Turma!
 atualizaTurma(id: ID!, turma: TurmaInp
 deletaTurma(id: ID!): RespostaPadrao!
}
```

COPIAR CÓDIGO

Conforme visto no curso anterior, podemos passar uma mensagem como retorno da query. Vamos adicionar uma mensagem padrão básica ao schema de Turmas:

```
interface Resposta {
  mensagem: String!
}

type RespostaPadrao implements Respost
  mensagem: String!
}
```

COPIAR CÓDIGO

Atualizando no resolver de Turmas:

```
deletaTurma: (_, { id }, { dataSources
```

COPIAR CÓDIGO

Em `datasource/turma.js`, antes de criarmos o método `deletaTurma`, precisamos implementar a interface `Resposta`. Há algumas formas de fazermos isso; para o momento, vamos seguir de forma similar à implementação que fizemos no curso anterior. No arquivo `datasources/turmas.js`:

```
  constructor(dbConfig) {
    super(dbConfig)
    this.Resposta = {
      mensagem: ""
    }
  }
```

COPIAR CÓDIGO

Agora podemos criar o último método. Para deletar um registro, precisamos somente do `id`.

```
  async deletaTurma(id) {
    await this.db('turmas')
      .where({ id: id })
      .del()

    this.Resposta.mensagem = "registro d
    return this.Resposta
  }
```

COPIAR CÓDIGO

Os métodos acima equivalem à query SQL:

```
DELETE FROM turmas WHERE id = ?
```

COPIAR CÓDIGO

E você pode testar no playground com a query:

```
mutation {
  deletaTurma(id: 4){
    mensagem
  }
}
```

COPIAR CÓDIGO

Agora que temos o CRUD completo para `Turmas`, podemos seguir em frente.