

Introdução ao TypeORM

Henrique Schmidt

@henriquels25

Código fonte utilizado na apresentação

<https://github.com/henriquels25/casas-api>

<https://github.com/henriquels25/fotos-api>

Roteiro

- O que é ORM e TypeORM?
- Entidades
- Repositórios
- Validação
- Relacionamentos
- Migrations
- Índices

**O que é ORM e
TypeORM?**

ORM

Object-relational mapping

Ferramenta para converter dados entre objetos e registros da base de dados.

Objeto vs Classe

Classe -> Template para criar objetos; um tipo.

Objeto -> Uma instância de uma classe.

Exemplo - Classe



Exemplo - Objeto



Exemplo - Classe

```
export class Casa {  
  nrQuartos: number;  
  nrJanelas: number  
  cor: string;  
  possuiGaragem: boolean;  
}
```

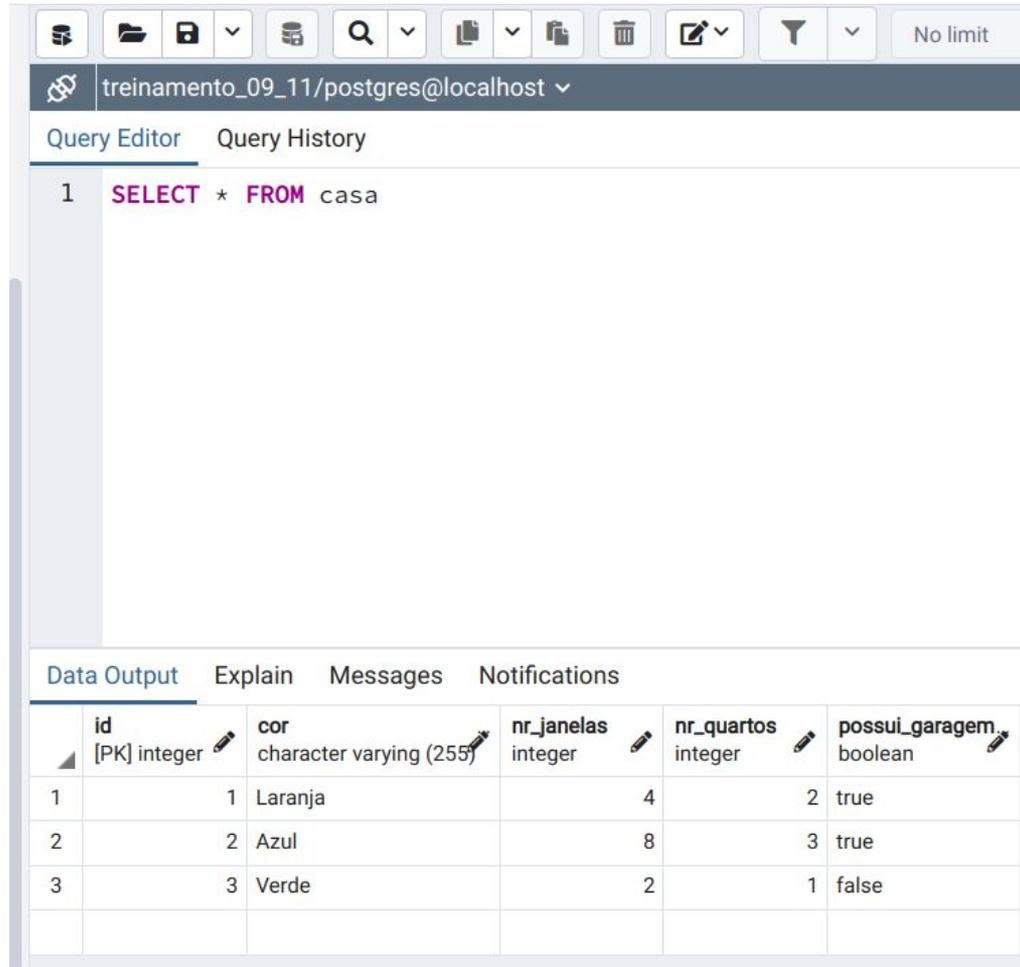
Exemplo - Objetos

```
export function createCasas(){  
  const casa1 = new Casa();  
  casa1.cor = 'Laranja';  
  casa1.nrJanelas = 4;  
  casa1.nrQuartos = 2;  
  casa1.possuiGaragem = true;  
  
  const casa2 = new Casa();  
  casa2.cor = 'Azul';  
  casa2.nrJanelas = 8;  
  casa2.nrQuartos = 3;  
  casa2.possuiGaragem = true;  
  
  const casa3 = new Casa();  
  casa3.cor = 'Verde';  
  casa3.nrJanelas = 2;  
  casa3.nrQuartos = 1;  
  casa3.possuiGaragem = false;  
}
```

Tabela

casa 			
	id	integer	 
	cor	varchar(255)	 
	nr_janelas	integer	 
	nr_quartos	integer	 
	possui_garagem	boolean	 
 Add field			

Registros



The screenshot shows a PostgreSQL query editor interface. At the top, there is a toolbar with icons for home, save, search, and other actions. Below the toolbar, the connection string is displayed as "treinamento_09_11/postgres@localhost". The main area is divided into two tabs: "Query Editor" and "Query History". The "Query Editor" tab is active, showing a single SQL query: "1 SELECT * FROM casa". Below the query editor, there are four tabs: "Data Output", "Explain", "Messages", and "Notifications". The "Data Output" tab is active, displaying a table with the results of the query. The table has five columns: "id", "cor", "nr_janelas", "nr_quartos", and "possui_garagem". The data is as follows:

	id [PK] integer	cor character varying (255)	nr_janelas integer	nr_quartos integer	possui_garagem boolean
1	1	Laranja	4	2	true
2	2	Azul	8	3	true
3	3	Verde	2	1	false

ORM - converter registros para objetos

	id [PK] integer	cor character varying (255)	nr_janelas integer	nr_quartos integer	possui_garagem boolean
1	1	Laranja	4	2	true
2	2	Azul	8	3	true
3	3	Verde	2	1	false

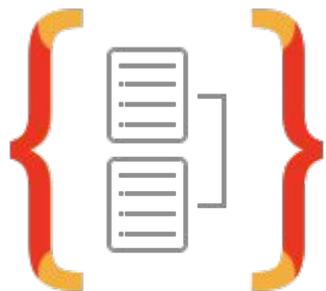


```
export function createCasas(){
  const casa1 = new Casa();
  casa1.cor = 'Laranja';
  casa1.nrJanelas = 4;
  casa1.nrQuartos = 2;
  casa1.possuiGaragem = true;

  const casa2 = new Casa();
  casa2.cor = 'Azul';
  casa2.nrJanelas = 8;
  casa2.nrQuartos = 3;
  casa2.possuiGaragem = true;

  const casa3 = new Casa();
  casa3.cor = 'Verde';
  casa3.nrJanelas = 2;
  casa3.nrQuartos = 1;
  casa3.possuiGaragem = false;
}
```

Opções de ORM



TYPEORM



TypeORM

- TypeORM é um ORM que pode ser usado com NodeJS e Javascript ou TypeScript.

Entidades

Definindo uma entidade

- Uma entidade é uma classe que faz o mapeamento entre uma tabela e uma classe.
- Deve-se colocar a anotação `@Entity()` em cima da classe

Definindo uma entidade

```
import { Entity } from "typeorm";  
  
@Entity()  
export class Casa {  
|
```

[entities/Casa.ts](#)

Definindo o nome da tabela

```
@Entity({name: 'casas'})  
export class Casa {  
|
```

[entities/Casa.ts](#)

Definindo a chave primária

- Normalmente, a chave primária é gerada automaticamente pelo banco.
- Para isso, temos a anotação `@PrimaryGeneratedColumn`

Definindo a chave primária

```
@PrimaryGeneratedColumn()  
id: number;
```

[entities/Casa.ts](#)

Definindo a chave primária

- Quando a chave primária não é gerada automaticamente, existe a anotação `@PrimaryColumn`

```
@PrimaryColumn()  
id: number;
```

Definindo colunas

- Cada coluna de uma tabela é mapeada para um atributo da classe relacionada.
- A anotação `@Column` é utilizada para isto.

Definindo colunas

```
@Column()  
cor: string;
```

[entities/Casa.ts](#)

Definindo colunas

- A anotação `@Column` possui o atributo `name` para o cenário em que o atributo tem um nome no BD

```
@Column({name: "nr_quartos"})  
nrQuartos: number;
```

Definindo colunas - tipos

- O primeiro parâmetro da anotação pode ser utilizado para definir o tipo.

```
@Column('int', {name: "nr_quartos"})  
nrQuartos: number;  
  
@Column('int', {name: "nr_janelas"})  
nrJanelas: number  
  
@Column('varchar')  
cor: string;
```

Definindo colunas - tipos

- Também pode ser definido desta forma:

```
@Column({name: "nr_quartos", type: 'int'})  
nrQuartos: number;  
  
@Column({name: "nr_janelas", type: 'int'})  
nrJanelas: number  
  
@Column({type: 'varchar'})  
cor: string;
```

Definindo colunas - limite de tamanho

- Para definir o tamanho limite de uma coluna varchar, o atributo length pode ser usado:

```
@Column('varchar', {length: 150})  
cor: string;
```

[entities/Casa.ts](#)

Definindo colunas - not null

- Para definir que uma coluna possa receber valores null, deve ser utilizado o atributo nullable;
- O default é false, ou seja, NOT NULL.

```
@Column({name: "data_construcao", nullable: true})  
dataConstrucao: Date
```

Definindo colunas - datas

- Por padrão, o tipo para coluna data é `TIMESTAMP` (data e hora).

```
@Column({name: "data_nascimento", nullable: true})
dataNascimento: Date;
```

[entities/Usuario.ts](#)

Definindo colunas - datas

- Por padrão, o tipo para coluna data é TIMESTAMP (data e hora).

```
@Column({name: "data_nascimento", nullable: true})
dataNascimento: Date;
```

```
{
  "id": 6,
  "nome": "Henrique4",
  "dataNascimento": "1997-02-05T00:00:00.000Z",
  "fotos": []
},
```

Definindo colunas - datas

- Podemos utilizar o primeiro parâmetro para definir uma coluna que guarda apenas a data:

```
@Column('date', {name: "data_nascimento", nullable: true})
dataNascimento: Date;
```

```
{
  "id": 5,
  "nome": "Henrique3",
  "dataNascimento": "1997-02-05",
  "fotos": []
},
```

Data de criação e atualização dos registros

- O TypeORM possui anotações que facilitam muito guardar a data que um registro foi criado ou atualizado.

Data de criação e atualização dos registros

```
@CreateDateColumn({name: 'data_criacao'})  
dataCriacao: Date;  
  
@UpdateDateColumn({name: 'data_atualizacao'})  
dataAtualizacao: Date;
```

Data de criação e atualização dos registros

	id [PK] integer	nome character varying	endereco_id integer	data_nascimento date	data_criacao timestamp without time zone	data_atualizacao timestamp without time zone
1	2	Gilberto	2	[null]	2021-11-29 00:17:49.353146	2021-11-29 00:17:49.353146
2	3	Henrique	3	[null]	2021-11-29 00:17:49.353146	2021-11-29 00:17:49.353146
3	4	Henrique	4	[null]	2021-11-29 00:17:49.353146	2021-11-29 00:17:49.353146
4	5	Henrique3	5	[null]	2021-11-29 00:17:49.353146	2021-11-29 00:17:49.353146
5	6	Henrique4	6	[null]	2021-11-29 00:17:49.353146	2021-11-29 00:17:49.353146

Data de criação e atualização dos registros

- É possível guardar a timezone do horário da criação ou atualização do registro.

```
@CreateDateColumn({name: 'data_criacao', type: 'timestampz'})  
dataCriacao: Date;
```

```
@UpdateDateColumn({name: 'data_atualizacao', type: 'timestampz'})  
dataAtualizacao: Date;
```

Data de criação e atualização dos registros

	id [PK] integer	nome character varying	endereco_id integer	data_nascimento date	data_criacao timestamp with time zone	data_atualizacao timestamp with time zone
1	2	Gilberto	2	[null]	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
2	3	Henrique	3	[null]	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
3	4	Henrique	4	[null]	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
4	5	Henrique3	5	[null]	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
5	6	Henrique4	6	[null]	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
6	7	Henrique4	7	1997-02-04	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00
7	8	Henrique8	8	1996-02-04	2021-11-29 00:23:12.10107+00	2021-11-29 00:23:12.10107+00

Repositórios

Repositório

- Para enviar e buscar entidades do banco de dados, um repository é utilizado.
- O método `getRepository` serve para obter um.

Repositório

```
import {getRepository} from "typeorm";  
  
export function createCasas(){  
    const repository = getRepository(Casa);
```

[CasaService.ts](#)

Salvar entidade

- O método save salva uma entidade no banco de dados.

```
const casa1 = new Casa();
casa1.cor = 'Laranja';
casa1.nrJanelas = 4;
casa1.nrQuartos = 2;
casa1.possuiGaragem = true;

await repository.save(casa1);
```

Buscar todos

- O método find busca todos registros do BD.

```
export async function buscaTodasCasas(){  
    const repository = getRepository(Casa);  
  
    return await repository.find();  
}
```

[CasaService.ts](#)

Buscar por id

- O método findOne busca um registro por id

```
export async function buscaCasaPorId(id: number) {  
    const repository = getRepository(Casa);  
  
    return await repository.findOne(id);  
}
```

Buscar por atributos

- O método find aceita um objeto com os filtros desejados

```
export async function buscaPorNumeroDeQuartos(nrQuartos: number){
  const repository = getRepository(Casa);

  return await repository.find({nrQuartos: nrQuartos});
}
```

Excluindo registros

- O método delete serve para excluir por id

```
export async function excluiCasa(id: number){  
  const repository = getRepository(Casa);  
  return await repository.delete(id);  
}
```

[CasaService.ts](#)

Excluindo registros

- O método delete também recebe um objeto de entidade para exclusão

```
export async function excluiCasa(id: number){
  const repository = getRepository(Casa);
  const casa = await repository.findOne(id);
  if (casa == undefined){
    throw 'Casa nao encontrada'
  }
  return await repository.delete(casa);
}
```

Atualizando

- Além de inserir, o método save pode ser utilizado para atualizar registros

Atualizando

```
export async function atualizaCor(id: number, cor: string) {
  const repository = getRepository(Casa);
  const casa = await repository.findOne(id);
  if (casa == undefined){
    throw 'Casa nao encontrada'
  }
  casa.cor = cor;
  return await repository.save(casa);
}
```

Validação

Validação

- Para validar a entidade antes de salvar no BD, usamos a biblioteca [class-validator](#).
- O objetivo é não deixarmos dados inconsistentes chegarem no banco.

Anotações

- A biblioteca usa anotações em cima dos atributos para configurar o que deve ser validado.

@NotEmpty

- Anotação que valida que o valor do campo não é null nem vazio.

```
import { NotEmpty } from 'class-validator'
```

```
@Column('varchar', {length: 150})  
@NotEmpty()  
cor: string;
```

@IsDate

- Anotação que valida que o valor é uma data

```
@Column({name: "data_construcao"})  
@IsDate()  
dataConstrucao: Date
```

[entities/Casa.ts](#)

@Min e @Max

- Anotação que valida o valor mínimo e máximo de um int.

```
@Column('int', {name: "nr_quartos"})  
@Min(0)  
@Max(20)  
nrQuartos: number;
```

@IsEmail

- Anotação que valida que um campo é um email válido.

```
@Column()  
@IsEmail()  
email: string;
```

[entities/Casa.ts](#)

@MaxLength

- Anotação que valida o tamanho de uma string.

```
@Column('varchar', {length: 150})  
@NotEmpty()  
@MaxLength(150)  
cor: string;
```

Como validar uma entidade

- Deve ser utilizado o método validate sempre antes de salvar algo no BD.

```
import {validate} from "class-validator";
```

```
const errors = await validate(casa1);  
if (errors.length > 0){  
    throw new Error('Casa inválida. Erros: ' + errors)  
}  
await repository.save(casa1);
```

Relacionamentos

Relacionamentos

- O TypeORM permite relacionamentos usando anotações:
 - 1 para 1: **@OneToOne**
 - n para 1: **@ManyToOne**
 - 1 para n: **@OneToMany**
 - n para n: **@ManyToMany**

1 para n - n para 1

- Exemplo:

usuarios			
	id	integer	
	nome	varchar(255)	
 Add field			



fotos			
	id	integer	
	url	varchar(255)	
	usuario_id	integer	
 Add field			

n para 1

```
@Entity()
export class Usuario {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  nome: string;
}
```

[entities/Usuario.ts](#)

```
@Entity()
export class Foto {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  url: string;

  @ManyToOne(() => Usuario)
  usuario: Usuario;
}
```

[entities/Foto.ts](#)

n para 1 - relações lazy

- Por padrão, relações são lazy e não são carregadas.

```
@Entity()
export class Foto {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  url: string;

  @ManyToOne(() => Usuario)
  usuario: Usuario;
}
```

```
[
  {
    "id": 1,
    "url": "https://foto1.jpg"
  },
]
```

n para 1 - eager

- Para configurar as relações para serem carregadas, há o atributo eager.
- Deve-se tomar cuidado para não afetar a **performance** do sistema.

n para 1 - eager

```
@Entity()
export class Foto {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  url: string;

  @ManyToOne(() => Usuario, {eager: true})
  usuario: Usuario;
}
```

[entities/Foto.ts](#)

```
[
  {
    "id": 1,
    "url": "https://foto1.jpg",
    "usuario": {
      "id": 1,
      "nome": "Henrique"
    }
  },
  {
```

n para 1 - definindo o nome da coluna

- É possível definir o nome da coluna a ser usada no relacionamento:

```
@ManyToOne(() => Usuario)
@JoinColumn({ name: 'usuario_id' })
usuario: Usuario;
```

[entities/Foto.ts](#)

- Por padrão, é igual ao nome da tabela relacionada + chave utilizada. Ex: usuarioid

n para 1 - Promise

- A opção para se utilizar relação lazy e obter os dados são as Promises.

```
@ManyToOne(() => Usuario)  
usuario: Promise<Usuario>;
```

```
const usuarioFoto = await f.usuario;  
console.log(usuarioFoto.id);
```

n para 1 - Promise

Note: if you came from other languages (Java, PHP, etc.) and are used to use lazy relations everywhere - be careful. Those languages aren't asynchronous and lazy loading is achieved different way, that's why you don't work with promises there. In JavaScript and Node.JS you have to use promises if you want to have lazy-loaded relations. This is non-standard technique and considered experimental in TypeORM.

1 para n

```
@Entity()
export class Usuario {

    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    nome: string;

    @OneToMany(() => Foto, foto => foto.usuario, {eager: true})
    fotos: Foto[];
}
```

1 para n

- É obrigatório o relacionamento n para 1 no outro lado do relacionamento.
- No exemplo, a lista de fotos só funciona porque a foto tem um relacionamento com o usuário.

1 para n

```
@OneToMany(() => Foto, foto => foto.usuario, {eager: true})  
fotos: Foto[];
```

[entities/Usuario.ts](#)

```
{  
  "id": 1,  
  "nome": "Henrique",  
  "fotos": [  
    {  
      "id": 1,  
      "url": "https://foto1.jpg"  
    },  
    {  
      "id": 2,  
      "url": "https://foto1.jpg"  
    }  
  ]  
},
```

1 para 1

endereco			
	id	integer	
	rua	varchar(255)	
	numero	varchar(255)	
 Add field			

usuarios			
	id	integer	
	nome	varchar(255)	
	endereco_id	integer	
 Add field			



1 para 1

No lado que é dono da relação, deve ficar a anotação `@JoinColumn`.

Classe Usuario:

```
@OneToOne(() => Endereco, endereco => endereco.usuario)
@JoinColumn()
endereco: Endereco;
```

1 para 1

Classe Usuario:

```
@OneToOne(() => Endereco, endereco => endereco.usuario)  
@JoinColumn()  
endereco: Endereco;
```

[entities/Usuario.ts](#)

Classe Endereco:

```
@OneToOne(() => Usuario, usuario => usuario.endereco)  
usuario: Usuario;
```

[entities/Endereco.ts](#)

1 para 1 - como salvar

```
app.post("/usuarios", async function(req: Request, res: Response) {  
  if (req.body.endereco != undefined){  
    const enderecoRepo = getRepository(Endereco);  
    await enderecoRepo.save(req.body.endereco);  
  }  
  const usuariosRepo = getRepository(Usuario);  
  await usuariosRepo.save(req.body);  
  return res.status(201).send();  
});
```

[app.ts](#)

1 para 1 - cascade

- Com cascade habilitado, a relação é salva automaticamente. Pode ser útil mas **perigoso**.

```
@OneToOne(() => Endereco, endereco => endereco.usuario,  
           {cascade: true})
```

[entities/Usuario.ts](#)

```
const usuariosRepo = getRepository(Usuario);  
await usuariosRepo.save(req.body);  
return res.status(201).send();
```

[app.ts](#)

Cascade

- Cascade funciona para todos tipos de relações.

Migrations

Synchronize

- O TypeORM possui a propriedade de conexão Synchronize.
- Quando está true, a aplicação cria e exclui tabelas e atributos conforme as entidades.
- **Não é recomendado o uso em produção.**

Migrations

- Como alternativa ao synchronize, as migrations podem ser utilizadas.

O que é uma Migration?

- Um arquivo com queries SQL para alterar o schema do banco de dados.
- Exemplo de query que altera o schema:

```
CREATE TABLE endereco (  
  "id" PRIMARY KEY SERIAL,  
  "rua" VARCHAR(255),  
  "numero" VARCHAR(255)  
)
```

Geração de migrations

- O TypeORM pode criar as migrations automaticamente a partir da conexão com um banco de dados:
- Comando:

```
npm run typeorm migration:generate -- -n <nome_migracao>
```

```
> fotos-api@1.0.0 typeorm
> node --require ts-node/register ./node_modules/typeorm/cli.js "migration:generate" "-n" "migracao_inicial"
```

```
Migration /home/henrique/IdeaProjects/personal/typescript/fotos-api/migration/1638137639346-migracao_inicial.ts has been generated successfully.
```

Geração de migrations

```
import {MigrationInterface, QueryRunner} from "typeorm";

export class migracaoInicial1638137639346 implements MigrationInterface {
  name = 'migracaoInicial1638137639346'

  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`CREATE TABLE "foto" ("id" SERIAL NOT NULL, "url" character varying
    await queryRunner.query(`CREATE TABLE "usuario" ("id" SERIAL NOT NULL, "nome" character var
    await queryRunner.query(`CREATE TABLE "endereco" ("id" SERIAL NOT NULL, "rua" character var
    await queryRunner.query(`ALTER TABLE "foto" ADD CONSTRAINT "FK_7960ecfde23620fb253883e6dad"
    await queryRunner.query(`ALTER TABLE "usuario" ADD CONSTRAINT "FK_6f962678dc18e5ec715e370e9
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`ALTER TABLE "usuario" DROP CONSTRAINT "FK_6f962678dc18e5ec715e370e9`);
  }
}
```

Exemplo - adicionar campo

```
@Column()  
idade: number;
```

npm run typeorm migration:generate -- -n adiciona_coluna_idade

```
export class adicionaColunaIdade1638140609051 implements MigrationInterface {  
  name = 'adicionaColunaIdade1638140609051'  
  
  public async up(queryRunner: QueryRunner): Promise<void> {  
    await queryRunner.query(`ALTER TABLE "usuario" ADD "idade" integer NOT NULL`);  
  }  
  
  public async down(queryRunner: QueryRunner): Promise<void> {  
    await queryRunner.query(`ALTER TABLE "usuario" DROP COLUMN "idade"`);  
  }  
}
```

[migrations/1638140609051-adiciona_coluna_idade.ts](#)

Execução de migração

- Para executar a migração no BD, o seguinte comando é utilizado:

```
npm run typeorm migration:run
```

Reversão de migração

- Caso a migração ocasione algum problema e seja necessário reverter, pode ser utilizado o seguinte comando:

```
npm run typeorm migration:revert
```

Índices

@Index

- Para adicionar um índice em uma coluna, basta adicionar a anotação @Index acima do atributo.

```
@Column()  
@Index()  
nome: string;
```

[entities/Usuario.ts](#)

Tópicos adicionais

- Configuração de conexão com DB
- QueryBuilder
- Transações

Dúvidas ou comentários?