Fundamentos de Banco de Dados

Aula 2
Msc. Maiara Coelho
Instituto de Computação
UFAM

Integrando o sequelize-typescript em uma API Express

- Use os comandos abaixo para instalar o Sequelize com suporte ao banco de dados MySQL e decorators para facilitar o uso do sequelize: Sequelize-typescript

```
$ npm init -y
$ npm install sequelize mysql2 sequelize-typescript
```

- Instalação de dependências e ferramentas de desenvolvimento.

```
$ npm install --save -dev typescript ts-node-dev
@types/express @types/node @types/node @types/validator
$ npm install reflect-metadata concurrently
```

Sequelize-typescript

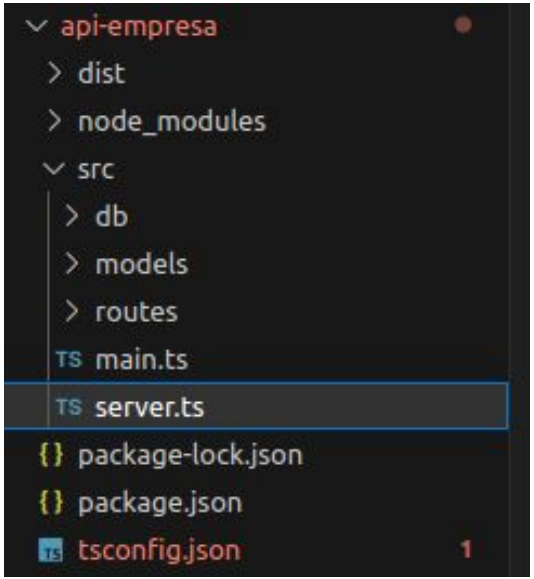
- Documentação
  - https://www.npmjs.com/package/sequelize-typescript

- Principais características
  - Definição de tabelas através de decoradores
  - Criação de tabelas o banco através dos models

Sequelize-typescript

- `@Table`
- `@Column - opções`
  - `type: DataType.FLOAT (https://sequelize.org/v5/manual/data-types.html)`
  - `primaryKey: true`
  - `unique=true`
  - `autoIncrement=true`
- `@PrimaryKey`
- `@AllowNull(allowNull?: boolean)`
- `@AutoIncrement`
- `@Unique(options? UniqueOptions)`
- `@Default(value: any)`

Estrutura da Aplicação



Configuração da API Empresa: tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2016",
   "lib": ["es6"],
   "module": "commonjs",
   "rootDir": "./src",
   "allowJs": true,
   "outDir": "./dist",
   "esModuleInterop": true,
   "forceConsistentCasingInFileNames": true,
   "strict": true,
   "noImplicitAny": true,
   "skipLibCheck": true,
   "resolveJsonModule": true,
   "experimentalDecorators": true,
   "emitDecoratorMetadata": true,
 }
}
```

Configuração da API Empresa: package.json

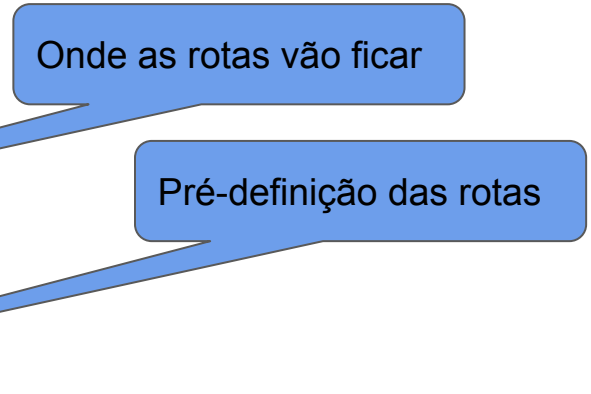
```
…
 "scripts": {
   "start": "concurrently \"npm run watch\" \"npm run dev\"",
   "dev": "ts-node-dev src/main.ts",
   "watch": "tsc -w",
   "test": "echo \"Error: no test specified\" && exit 1"
 },
…
```

Criando a API: main.ts

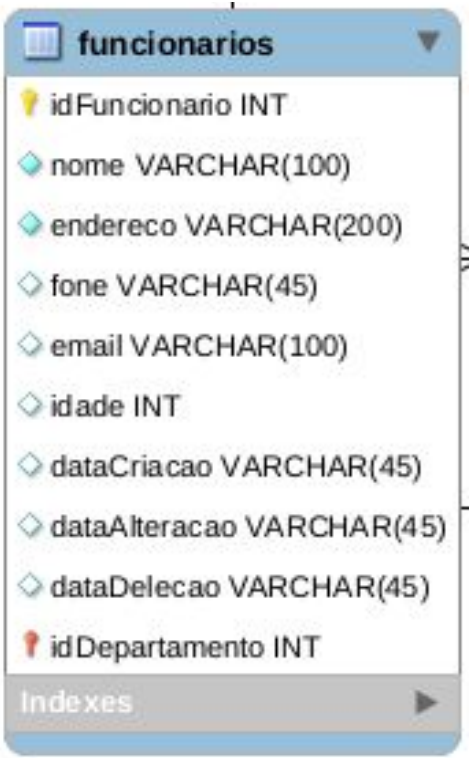
```
import { Api } from "./server"
const start = async (): Promise<void> => {
   try {
       new Api().server.listen(3000, () => {
       console.log("Server started on port 3000");
     });
   } catch (error) {
     console.error(error);
     process.exit(1);
   }
 };
 void start();
```

Criando a API: server.ts

```
import express from "express";
import { router } from "router/router";
export class Api{
 public server: express.Application;
 constructor(){
   this.server = express();
   this.middleware();
   this.router();
 }
  private middleware(){
   this.server.use(express.json());
 }
 private router(){
   this.server.use(router);
 }
}
```



Criando um Model




```
| Funcionarios |
```

show datatables;

Adicione em db/config.ts:

```
models:
  [
    Funcionarios
  ],
```

Primeiro Model: Funcionarios.ts

```
import { Table, Model, Column, DataType, IsUUID, PrimaryKey,
AllowNull, IsEmail, Unique } from "sequelize-typescript";
@Table({
 timestamps: true,
})
export class Funcionarios extends Model {
 @IsUUID('all')
 @PrimaryKey
 @Column({
   type: DataType.UUID,
   defaultValue: DataType.UUIDV1,
 })
 id!: string;
 @AllowNull(false)
 @Column({
   type: DataType.STRING,
 })
 name!: string;
 @AllowNull(false)
 @Column({
   type: DataType.STRING,
 })
 fone!: string;
 @AllowNull(false)
 @Unique
 @IsEmail
 @Column({
   type: DataType.STRING,
 })
 email!: string;
 @AllowNull(false)
 @Column({
   type: DataType.INTEGER,
 })
 idade!: number;
}
```

Configurando a conexão com o banco

```
import { Sequelize } from "sequelize-typescript";
import { Funcionarios } from "../models/Funcionarios";

const connection = new Sequelize({
 dialect: "mysql",
 host: HOST,
 username: USUARIO_MYSQL,
 password: SENHA,
 database: BANCO,
 logging: false,
 models: [Funcionarios],
});
export default connection;
```

Sempre que surgirem novos models, eles devem ser adicionados aqui

Adicionando a conexão com o banco

```
//main.ts

import connection from "./db/config";
import { Api } from "./server"
const start = async (): Promise<void> =>
{
   try {
       await connection.sync();
  …
 };
 void start();
```

Encontrando registros

- Para recuperar registros no banco de dados:

```
const records = await Something.findAll(criteria)
```

- Exemplos:
  - SELECT * FROM Funcionarios WHERE nome = 'Carlos':

```
Funcionarios.findAll({ where: { nome: 'Carlos' }});
```

  - SELECT nome,idade FROM user WHERE nome = 'Carlos':

```
Funcionarios.findAll({
 where: { nome: 'Carlos' },
 Attributes: [ 'nome', 'idade' ]
});
```

Encontrando registros

- O objeto Sequelize.Op pode ser usado para criar comparações mais complexas durante as consultas

```
const Op = Sequelize.Op
[Op.and]: {a: 5}           // AND (a = 5)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
[Op.gt]: 6,                // > 6
[Op.gte]: 6,               // >= 6
[Op.lt]: 10,               // < 10
[Op.lte]: 10,              // <= 10
[Op.ne]: 20,               // != 20
[Op.eq]: 3,                // = 3
[Op.is]: null              // IS NULL
[Op.not]: true,            // IS NOT TRUE
[Op.between]: [6, 10],     // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
[Op.in]: [1, 2],           // IN [1, 2]
[Op.notIn]: [1, 2],        // NOT IN [1, 2]
```

Encontrando registros

- O objeto Sequelize.Op pode ser usado para criar comparações mais complexas durante as consultas

```
const Op = Sequelize.Op
[Op.and]: {a: 5}           // AND (a = 5)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
[Op.gt]: 6                 // > 6
[Op.like]: '%hat',         // LIKE '%hat'
[Op.notLike]: '%hat'       // NOT LIKE '%hat'
[Op.startsWith]: 'hat'     // LIKE 'hat%'
[Op.endsWith]: 'hat'       // LIKE '%hat'
[Op.substring]: 'hat'      // LIKE '%hat%'
[Op.regexp]: '^[h|a|t]'    // REGEXP/~ '^[h|a|t]'
[Op.notRegexp]: '^[h|a|t]' // NOT REGEXP/!~ '^[h|a|t]'
[Op.between]: [6, 10],     // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
[Op.in]: [1, 2],           // IN [1, 2]
[Op.notIn]: [1, 2],        // NOT IN [1, 2]
```

Encontrando registros

- Exemplos de consultas com o objeto Sequelize.Op
- `SELECT * FROM Funcionarios WHERE idade < 30:`

```
Funcionarios.findAll({ where: { idade: { [Op.lt]: 30 }}});
```

- `SELECT * FROM Funcionarios WHERE idade >= 21:`

```
Funcionarios.findAll({ where: { idade: { [Op.gte]: 21 }}});
```

- `SELECT * FROM Funcionarios WHERE nome IN ('Carlos', 'Maria'):`

```
Funcionarios.findAll({
  where: { nome: { [Op.in]: ['Carlos', 'Maria'] }}
});
```

- `SELECT * FROM Funcionarios WHERE nome NOT IN ('Carlos', 'Maria'):`

```
Funcionarios.findAll({
  where: { nome: { [Op.notIn]: ['Carlos', 'Maria'] }}
});
```

Encontrando registros

- `SELECT * FROM Funcionarios WHERE nome like '%Carlos%':`

```
Funcionarios.findAll({
  where: { nome: { [Op.like]: '%Carlos%' }}
});
```

Nomes que possuem a string Carlos

- `SELECT * FROM Funcionarios WHERE nome like 'Carlos%':`

```
Funcionarios.findAll({
  where: { nome: { [Op.like]: 'Carlos%' }}
});
```

Nomes que começam com a string Carlos

- `SELECT * FROM user WHERE nome like '%Oliveira':`

```
User.findAll({
  where: { nome: { [Op.like]: '%Oliveira' }}
});
```

Nomes que terminam com a string Oliveira

Paginação

- As cláusulas offset e limit podem ser usadas em conjunto para construir um sistema de paginação.
- `SELECT * FROM Funcionarios WHERE idade > 20 LIMIT 10 OFFSET 0;`

```
Funcionarios.findAll({
   where: { idade: { [Op.gt]: 20 }},
   limit: 10, offset: 0
});
```

Funcionários 1-10 com idade maior que 20

- `SELECT * FROM Funcionarios WHERE idade > 20 LIMIT 10 OFFSET 10;`

```
Funcionarios.findAll({
   where: { idade: { [Op.gt]: 20 }},
   limit: 10, offset: 10
});
```

Funcionários 11-20 com idade maior que 20

Ordenando Registros

- Os resultados recuperados podem ser ordenados de forma crescente (ASC) ou descrescente (DESC) por qualquer atributo

- SELECT * FROM Funcionarios WHERE idade >= 18 ORDER BY nome ASC;

```
Funcionarios.findAll({
    where: { idade: { [Op.gte]: 18 }},
           order: [['nome', 'ASC']]
});
```

Funcionários >=18 ordenados pelo nome em ordem crescente

- SELECT * FROM Funcionarios WHERE idade >= 18 ORDER BY nome DESC;

```
Funcionarios.findAll({
    where: { idade: { [Op.gte]: 18 }},
           order: [['nome', 'DESC']]
});
```

Funcionários >=18 ordenados pelo nome em ordem decrescente

Selecionando apenas um registro

- O comando **findAll()** retorna um array com todos os registros que atenderem os critérios da cláusula **where**
  - Note que, se apenas um registro atender aos critérios da cláusula where, será retornado um array com uma única posição
- Uma alternativa é o uso de **findOne()**, que ao invés de recuperar um array, retorna apenas um objeto
  - `SELECT * FROM Funcionarios WHERE id = 1;`

```
const funcionario = await Funcionarios.findOne({
                                    where: {id:1}
                                });
```

Criando novos registros

- Criação de novos registros no banco de dados:

```
await Something.create(initialValues);
```

- Exemplo:
  - INSERT INTO Funcionarios (nome, endereco, fone, email, idade) VALUES ('Carlos', 'Rua, n 1, bairro', 3333-3333,'email@dominio.com', 26);

```
await Funcionarios.create({
    nome:'Carlos',
    endereco: 'Rua, n 1, bairro',
    fone: 3333-3333,
    email: 'email@dominio.com',
    idade: 26
});
```

Atualizando registros existentes

- Atualização de registros já existentes no banco de dados:

```
await Something.update({values},{criteria});
```

- Exemplo:
  - `UPDATE Funcionarios SET idade=30 WHERE nome = 'Carlos';`

```
await Funcionarios.update({
    idade: 30 },{
    where: {
        nome: 'Carlos'
    }
});
```

Apagando registros existentes

- Apagando registros no banco de dados:

```
await Something.destroy(criteria);
```

- Exemplo:
  - `DELETE FROM Funcionarios WHERE nome = 'Carlos';`

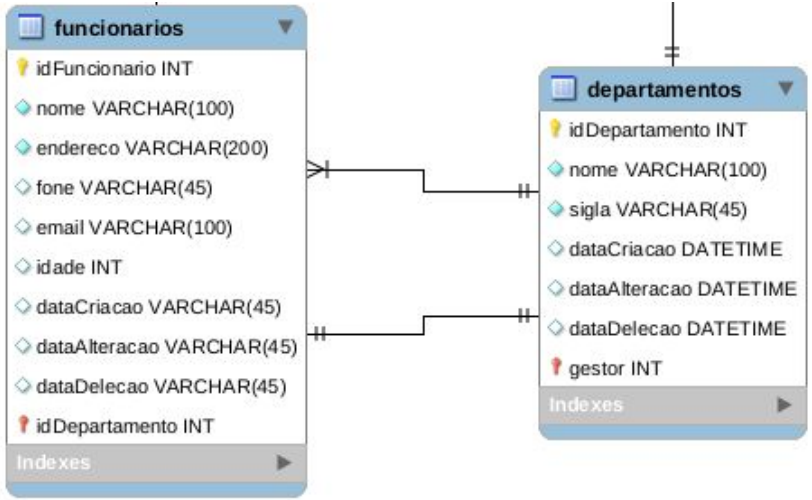
```
await Funcionarios.destroy({
  where: { nome: 'Carlos' }
});
```

  - `DELETE FROM Funcionarios WHERE id IN (3, 97);`

```
await Funcionarios.destroy({
  where: { [Op.in]: [3, 97] }
});
```

Adicionando Relações entre Tabelas

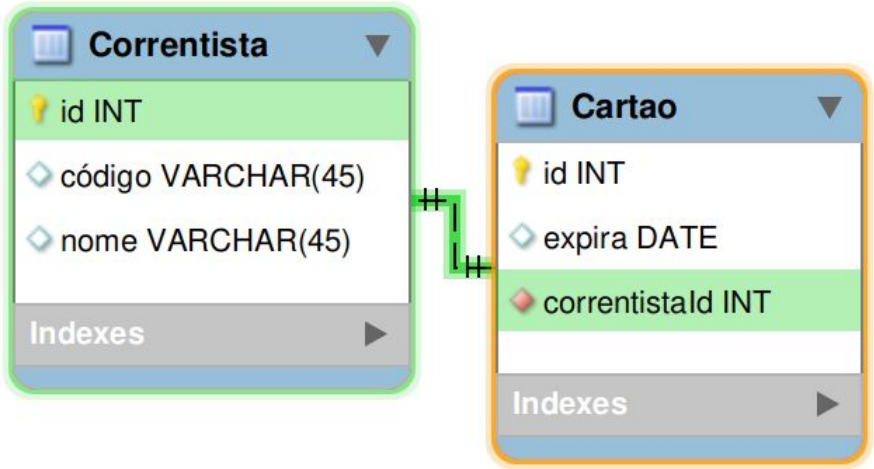
- Além dos atributos (do tipo string, number, etc), os modelos também podem possuir conexões com outros modelos.
- Essas conexões com outros modelos são chamados de associações.
- Por exemplo, no esquema de banco de dados de nossa aplicação, cada funcionário está sempre associado a um departamento. E um departamento tem vários funcionários, assim como também tem um um funcionário gestor.



Associações Um para Um

- Em um associação um para um, uma instância de um modelo está associada com apenas uma instância de outro modelo
- Para definirmos uma associação um para um no Sequelize, usamos as chaves estrangeiras e os decoradores belongsTo e hasOne.

```
@ForeignKey(() => Correntista)
 @AllowNull(false)
 @Column({
   type: DataType.UUID,
 })
 correntistaId: string;
 @BelongsTo(() => Correntista)
 correntista: Correntista;
```



Associações Um para Um

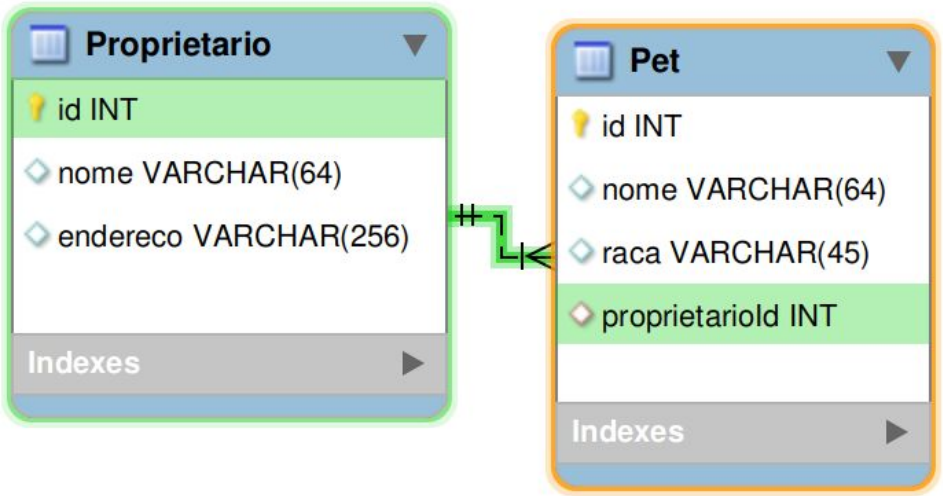
- Para carregar os dados do cartão de um correntista, podemos usar a função **findAll** (ou **findOne**, etc) com a opção **include**:

```
const correntista = await Correntista.findByPk(123, {
   include: models.Cartao,
});
```

- No exemplo acima, **correntista.Cartao** será um objeto contendo todos os dados do cartão do correntista **id = 123**

Associações Um para Muitos

- Em uma associação um para muitos, uma instância de um modelo pode estar associada à várias instâncias do outro modelo
- Por exemplo, no esquema abaixo, um proprietário pode ter vários pets (animais de estimação)



Associações Um para Muitos

- Para definirmos uma associação um para muitos no Sequelize, usamos os métodos belongsTo e hasMany

```
// Proprietario.ts
@HasMany(() => Pet)
 pet: Pet;
```

```
// Pet.ts
@ForeignKey(() => Pet)
 @AllowNull(false)
 @Column({
   type: DataType.UUID,
 })
 proprietarioId: string;
 @BelongsTo(() => Proprietario)
 proprietario: Proprietario;
```

Associações Um para Muitos

- Para carregar os pets de um proprietario, podemos usar a função **findAll** (ou **findOne, findByPK**, etc) com a opção **include**:

```
const proprietario = await Proprietario.findByPk(123,
{
   include: models.Pet,
});
```

- No exemplo acima, **proprietario.Pet** é um array de objetos contendo todos os pets do proprietário com **id = 123**

Associações Muitos para Muitos

- Em uma associação muitos para muitos, um registro pode ser associado com **muitos outros e vice-versa**
- Por exemplo, no esquema abaixo, um engenheiro pode ter muitos projetos e um projeto pode ter vários engenheiros

Associações Muitos para Muitos

- Para definirmos uma associação muitos para muitos, usamos os o método belongsToMany nos dois lados da associação

```
// Engenheiro.ts
@BelongsToMany(() => Projeto, ()
=> EngenheiroProjeto)
 projetos: Projeto[];
```

```
// Projeto.ts
@BelongsToMany(() => Engenheiro,
() => EngenheiroProjeto)
 engenheiros: Engenheiro[];
```

Associações Muitos para Muitos

- Podemos carregar os projetos de um **engenheiro** através da função **findAll** (ou **findOne**, **findByPk**, etc) com a opção **include**

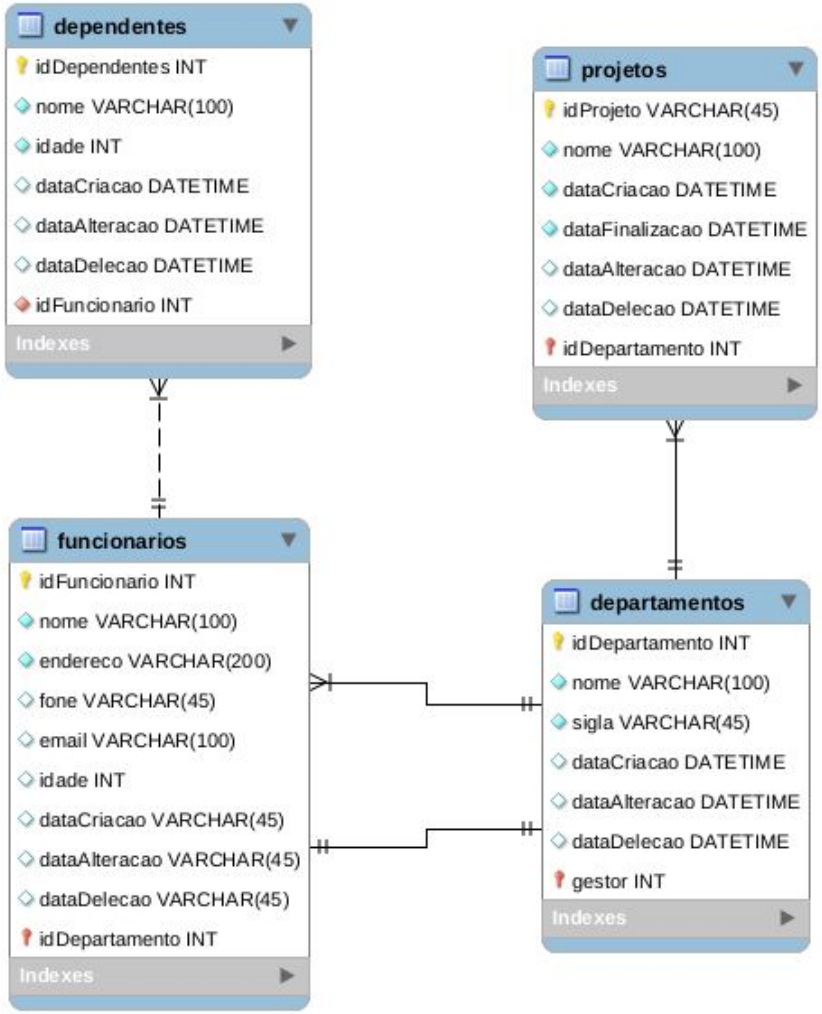
```
const engenheiro = await Engenheiro.findByPk(123, {
   include: models.Projeto
});
```

- No exemplo acima, **engenheiro.Projeto** é um array de objetos contendo todos os projetos do engenheiro com **id = 123**

Adicionando Relações entre Tabelas

Exercício 3
1 - Crie o restante das tabelas do modelo.
2 - Faça as associações necessárias.

Obs: Esse exercício é para ser entregue ainda hoje (13/06/2023) via colabweb, ele é parte da nota 2 e vale 2 pontos.



Exemplo: Departamentos.ts

```
import { Table, Model, Column,
DataType, IsUUID, PrimaryKey,
AllowNull, IsEmail, Unique,
ForeignKey, BelongsTo, HasOne, HasMany
} from "sequelize-typescript";
import { Funcionarios } from
"./Funcionarios";
import { Projetos } from "./Projetos";

@Table({
 timestamps: true,
})
export class Departamentos extends
Model {
```

```
@IsUUID('all')
@PrimaryKey
@Column({
  type: DataType.UUID,
  defaultValue: DataType.UUIDV1,
})
id!: string;

@AllowNull(false)
@Unique
@Column({
  type: DataType.STRING,
})
name!: string;
```

Exemplo: Departamentos.ts

```
@AllowNull(false)
@Column({
  type: DataType.STRING,
})
sigla!: string;

@ForeignKey(() => Funcionarios)
@AllowNull(true)
@Column({
  type: DataType.UUID,
})
gestorId!: string;
```

```
 @HasOne(() => Funcionarios,
'gestorId')
 gestor!: Funcionarios;

 @HasMany(() => Projetos)
 projetos!: Projetos[];

}
```

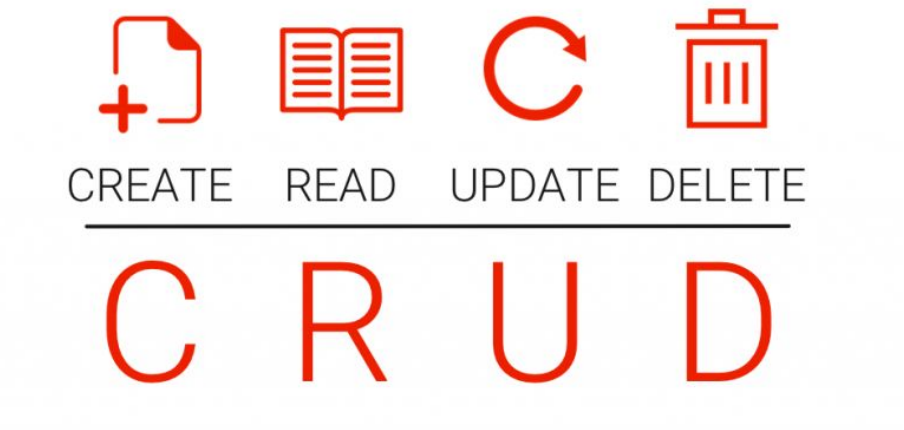
Verificação das Tabelas Criadas no MySQL

Adicione em db/config.ts:

```
models:
  [
    Funcionarios,
    Departamentos,
    Projetos,
    Dependentes
  ],
```

Desenvolvendo CRUDs em uma API

- Uma parte importante do desenvolvimento de uma api é a criação do CRUD a alguns modelos
- O CRUD de um modelo é um conjunto de rotas responsáveis por quatro operações sobre esse modelo:



Desenvolvendo CRUDs em uma API

- Para exemplificar a criação de novos CRUDs, vamos desenvolver um para o modelo Departamentos

- O primeiro passo é criar um arquivo .router.ts vazio para o CRUD de Departamento dentro do diretório /src/routes, chamado Departamentos.router.ts

Desenvolvendo CRUDs em uma API

- GET /departamentos: retorna todos os departamentos da empresa.

```
import { Router } from "express";
import { Request, Response } from "express";
import { Departamentos } from "../models/Departamentos";

const departamentosRouter: Router = Router();

departamentosRouter.get("/departamentos", async (req: Request, res: Response):
Promise<Response> => {
   const todosDepartamentos: Departamentos[] = await Departamentos.findAll();
   return res.status(200).json(todosDepartamentos);
});
```

Desenvolvendo CRUDs em uma API

- GET /departamentos/:id: retorna o departamento cuja chave primária é igual a informada por parâmetro.
- Para ler o valor de id dentro, podemos usar o atributo param de req (objeto da requisição do usuário):

```
departamentosRouter.get("/departamentos/:id", async (req: Request, res: Response):
Promise<Response> => {
   const { id } = req.params;
   const departamento: Departamentos | null = await Departamentos.findByPk(id);
   return res.status(200).json(departamento);
});
```

Desenvolvendo CRUDs em uma API

- Quando o usuário preenche e submete um formulário POST, os dados informados pelo usuário são enviados para o servidor
- Após a submissão, os dados são enviados através do corpo da requisição HTTP (request body)
- POST /departamentos: pega os dados que estão em **req.body** e cria uma instância para Departamento, a qual será inserida no banco via Sequelize:

```
departamentosRouter.post("/departamentos", async (req: Request, res: Response):
Promise<Response> => {
   const departamento: Departamentos = await Departamentos.create({ ...req.body });
   return res.status(201).json(departamento);
});
```

Desenvolvendo CRUDs em uma API

- PUT /departamentos/:id: atualiza o departamento cujo id corresponde ao informado.

```
departamentosRouter.put("/departamentos/:id", async (req: Request, res: Response):
Promise<Response> => {
   const { id } = req.params;
   await Departamentos.update({ ...req.body }, { where: { id } });
   const updatedDepartamento: Departamentos | null = await Departamentos.findByPk(id);
   return res.status(200).json(updatedDepartamento);
});
```

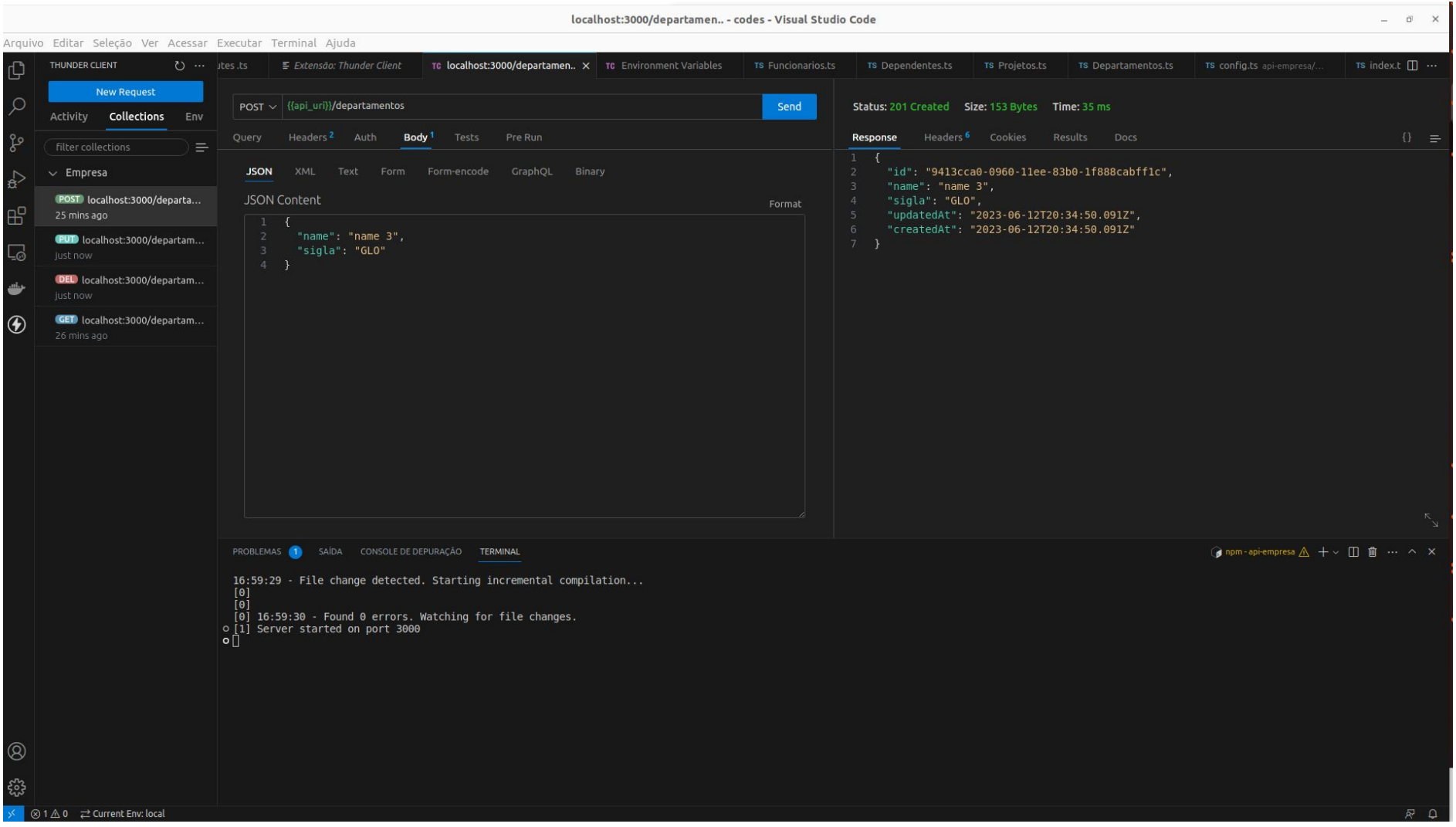
Desenvolvendo CRUDs em uma API

- DELETE /departamentos/:id: deleta o departamento cujo id corresponde ao informado.

```
departamentosRouter.delete("/departamentos/:id", async (req: Request, res: Response):
Promise<Response> => {
   const { id } = req.params;
   const deletedDepartamento: Departamentos | null = await Departamentos.findByPk(id);
   await Departamentos.destroy({ where: { id } });
   return res.status(200).json(deletedDepartamento);
 }
);
export { departamentosRouter };
```

Testando as Rotas

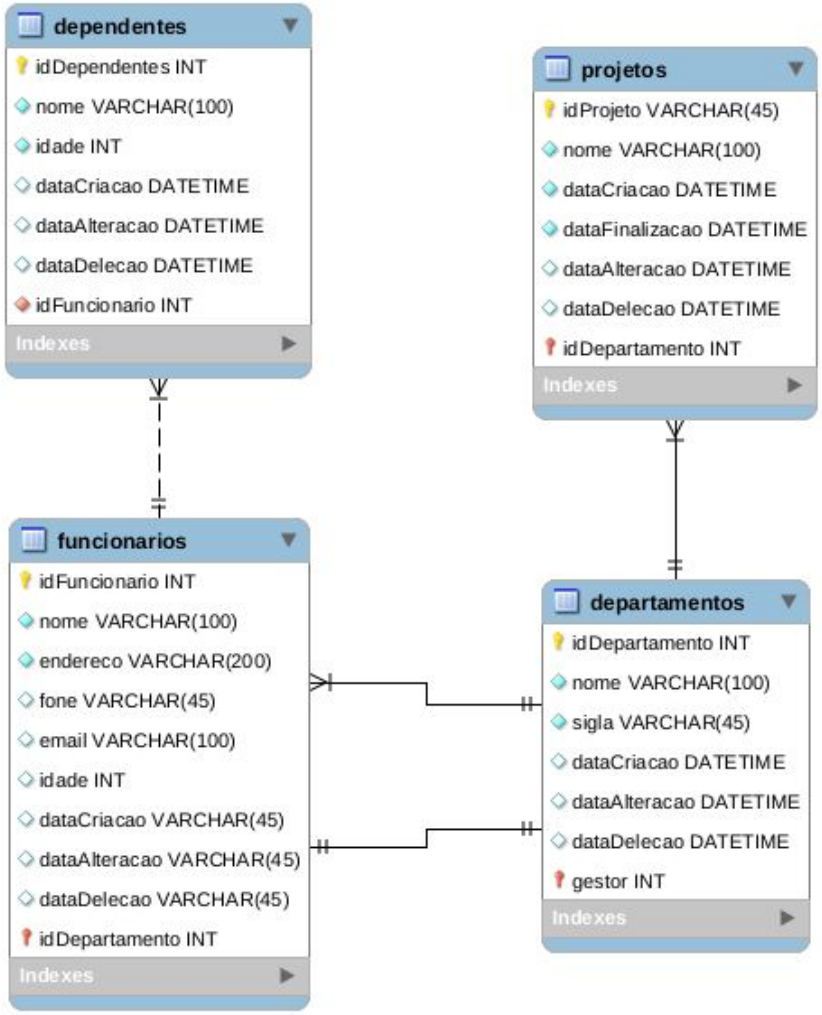
- Cliente para testar APIs
- Extensão no Visual Studio Code
- Documentação
  - https://github.com/rangav/thunder-client-support
- Tutorial de utilização
  - https://rangav.medium.com/thunder-client-cli-a-new-way-to-test-apis-inside-vscode-d91eb5c71d8e


localhost:3000/departamen.. - codes - Visual Studio Code
Arquivo Editar Seleção Ver Acessar Executar Terminal Ajuda
THUNDER CLIENT
New Request
Activity
Collections
Env
filter collections
Empresa
POST localhost:3000/departa...
25 mins ago
PUT localhost:3000/departam...
Just now
DEL localhost:3000/departam...
just now
GET localhost:3000/departam...
26 mins ago
Extensão: Thunder Client
localhost:3000/departamen..
Environment Variables
Funcionarios.ts
Dependentes.ts
Projetos.ts
Departamentos.ts
config.ts api-empresa/...
index.t
POST
{{api_url}}/departamentos
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
{
"name": "name 3",
"sigla": "GLO"
}
Status: 201 Created
Size: 153 Bytes
Time: 35 ms
Response
Headers 6
Cookies
Results
Docs
{
"id": "9413cca0-0960-11ee-83b0-1f888cabff1c",
"name": "name 3",
"sigla": "GLO",
"updatedAt": "2023-06-12T20:34:50.091Z",
"createdAt": "2023-06-12T20:34:50.091Z"
}
PROBLEMAS 1
SAÍDA
CONSOLE DE DEPURAÇÃO
TERMINAL
npm - api-empresa
16:59:29 - File change detected. Starting incremental compilation...
[0]
[0]
[0] 16:59:30 - Found 0 errors. Watching for file changes.
[1] Server started on port 3000
1 0
Current Env: local

Adicionando Relações entre Tabelas

Exercício 3
3 - Crie CRUDs para o restante dos models

Obs: Esse exercício é para ser entregue ainda hoje (13/06/2023) via colabweb, ele é parte da nota 2 e vale 2 pontos.



Exercício 4

1 - Crie uma api com models, suas associações e CRUDs básicos para cada uma das tabelas do modelo “Loja Virtual”, gerado na questão 4 do exercício 2.

Obs: Esse exercício é para ser entregue até o dia 15/06/2023 às 17, via colabweb, ele é parte da nota 2 e vale 8 pontos.