Objetivos:

I. ORM;
II. TypeORM;
III. Data Source;
IV. Migrations;
V. Entidades e relacionamentos;
VI. Controllers.

Instruções para criar o projeto para reproduzir os exemplos:

a) Crie uma pasta de nome aula4 (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
b) Abra a pasta aula4 no VS Code e acesse o terminal do VS Code;
c) No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node, arquivo package.json;
d) No terminal, execute o comando `npm i express` para instalar o pacote express;
e) No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém as definições de tipos do pacote express. Quando usamos um pacote é preciso ter acesso às declarações de tipo do pacote para que o TS saiba quais tipos de dados esperar do framework;
f) No terminal, execute o comando `npm i dotenv` para instalar o pacote dotenv. As variáveis de ambientes são acessadas através do objeto `process.env`. Porém, as variáveis declaradas no arquivo .env não são carregadas pelo ambiente de execução do Node no objeto `process.env`. Usaremos o dotenv parar carregar as variáveis do arquivo .env no objeto `process.env`;
g) No terminal, execute o comando `npm i typeorm` para instalar a biblioteca que possui ferramentas para persistir dados no SGBD (https://www.npmjs.com/package/typeorm);
h) No terminal, execute o comando `npm i sqlite3` para instalar a biblioteca que possui ferramentas para acessar o BD do SQLite (https://www.npmjs.com/package/sqlite3);
i) No terminal, execute o comando `npm i pg` para instalar a biblioteca que possui ferramentas para acessar o SGBD PostgreSQL (https://www.npmjs.com/package/pg);
j) No terminal, execute o comando `npm i -D @types/pg` para instalar o pacote que contém as definições de tipos do pacote pg;
k) No terminal, execute o comando `npm i -D ts-node ts-node-dev typescript` para instalar os pacotes ts-node, ts-node-dev e typescript como dependências de desenvolvimento;
l) No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo tsconfig.json);
m) Crie o arquivo .gitignore na raiz do projeto e coloque a linha para ignorar a pasta node_modules;
n) Crie o arquivo .env na raiz do projeto e coloque as seguintes variáveis de ambiente:

```
PORT = 3001
```

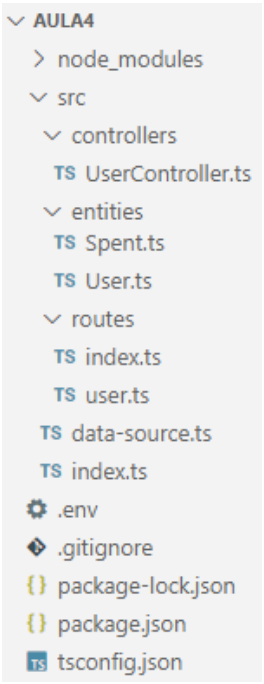
o) Crie a pasta src na raiz do projeto;

p) Crie as pastas entities, controllers e routes na pasta src;

q) Crie os arquivos index.ts e data-source.ts na pasta src e os demais arquivos nas pastas controllers, entities e routes.

No momento o projeto terá a estrutura mostrada ao lado.

Estrutura de pastas e arquivos do projeto:



i. ORM

ORM (Object-Relational Mapping - Mapeamento Objeto-Relacional) é uma técnica de programação que permite mapear objetos para tabelas de um BD relacional. Em outras palavras, ele fornece uma camada de abstração entre o código da aplicação e o BD, nos permitindo manipular dados no BD usando objetos e métodos em vez de escrever consultas SQL diretamente.

A principal finalidade de um ORM é facilitar a interação entre o código da aplicação, que normalmente é escrito em linguagens orientadas a objetos como Java, C# e TS, e um BD relacional, que armazena dados em tabelas. Ele faz isso fornecendo um conjunto de classes e métodos que nos permite realizar operações CRUD (Create, Read, Update, Delete) no BD usando objetos e suas propriedades em vez de escrever SQL manualmente.

Alguns benefícios do uso de um ORM incluem:

- Abstração do BD: os desenvolvedores podem trabalhar com objetos em vez de lidar diretamente com estruturas de banco de dados, tornando o desenvolvimento mais produtivo;
- Portabilidade de BD: um ORM pode ser configurado para funcionar com diferentes SGBDs como MySQL, PostgreSQL, Oracle, SQLite etc., o que facilita a migração do aplicativo entre diferentes plataformas de BD;
- Manutenção simplificada: o código da aplicação se torna mais limpo e fácil de manter, pois as consultas SQL estão encapsuladas nas classes do ORM;
- Segurança: ORMs geralmente oferecem mecanismos para evitar injeção de SQL, tornando as aplicações mais seguras;
- Desenvolvimento mais rápido: o uso de um ORM pode acelerar o desenvolvimento, já que a maior parte do código SQL é gerada automaticamente.

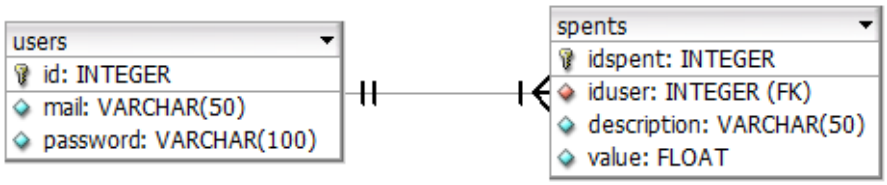
Alguns exemplos populares de ORMs incluem o Hibernate para Java, o Entity Framework para .NET, o Django ORM para Python, o Sequelize e Prisma para JavaScript/Node.js e o TypeORM para TypeScript.

ii. TypeORM

O TypeORM (https://typeorm.io) é uma biblioteca JS/TS que fornece funcionalidades de mapeamento objeto-relacional (ORM) para interagir com BD relacionais. Essa biblioteca permite que os desenvolvedores realizem operações de BD, como CRUD (criar, ler, atualizar e excluir registros), usando objetos e classes em seu código. Embora seja possível usar TypeORM com JS puro, sua integração com TS é mais direta, pois oferece um alto nível de tipagem estática.

O TypeORM faz o mapeamento de objetos do código para tabelas do BD e vice-versa através de entidades. As entidades se referem às classes que definimos para representar as estruturas de dados usadas para criar os objetos. Cada entidade geralmente está associada a uma tabela em um BD relacional.

Nos exemplos a seguir considere as tabelas users e spents (gastos) representadas no modelo:



No TypeORM as entidades são definidas como classes TS. A classe User, a seguir, será usada pelo TypeORM para configurar o esquema da tabela users no BD. Temos de usar os decorators para indicar ao TypeORM as classes que serão tabelas e as propriedades que serão colunas:

- O decorator `@Entity` é usado para indicar ao TypeORM que a classe User deverá ser mapeada para uma tabela no BD;
- O decorator `@PrimaryGeneratedColumn` é usado para indicar que a propriedade id define a coluna id na tabela e essa coluna será chave primária;
- O decorator `@Column` é usado para indicar que a propriedade será uma coluna da tabela.

Coloque o código a seguir no arquivo src/entities/User.ts:

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

@Entity({name:"users"})
export class User {
    // define a chave primária como autoincremento. A propriedade primaryKeyConstraintName
    // é opcional, usamos ela para setar o nome da restrição da chave primária no SGBD
    @PrimaryGeneratedColumn({primaryKeyConstraintName:"pk-user"})
    id: number;

    @Column({nullable: false, unique:true, length: 50})
    mail: string;

    @Column({nullable: false, length: 100})
    password: string;
}
```

Observação: faça as seguintes modificações no arquivo tsconfig.json para evitar os erros de análise estática no arquivo User.ts:

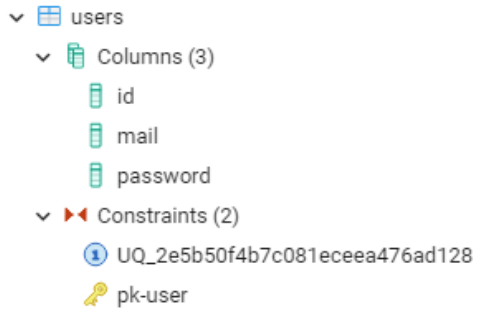
- Retirar os comentários das propriedades `"experimentalDecorators":true` e `"emitDecoratorMetadata": true`;
- Comentar a propriedade `//"strict": true`.

Os decorators são funções, por este motivo a sua chamada é seguida por parênteses: `@Entity()`, `@PrimaryGeneratedColumn()` e `@Column()`. Essas funções decorators podem receber parâmetros no formato de objeto JSON para configurar propriedades da tabela ou coluna. O objeto JSON na chamada da função `@Entity({name:"users"})` é usado para indicar que a tabela terá o nome de `users` no SGBD. O objeto JSON na chamada da função `@Column({nullable: false, unique:true, length: 50})` é usado para indicar que a coluna `mail` será varchar(50) not null e não poderá ter valores repetidos, pois possui o índice de valor único.

O TypeORM criará o seguinte comando SQL para o SGBD PostgreSQL usando a classe `User`. O tipo de dado da coluna é inferido pelo tipo de dado da propriedade (`id: number` e `mail: string`). A restrições (constraints) de valor único e chave primária foram definidas usando os decorators:

```
CREATE TABLE "users" (
   "id" SERIAL NOT NULL,
   "mail" character varying(50) NOT NULL,
   "password" character varying(100) NOT NULL,
   CONSTRAINT "UQ_2e5b50f4b7c081eceea476ad128" UNIQUE
   ("mail"),
   CONSTRAINT "pk-user" PRIMARY KEY ("id")
);
```

Tabela no SGBD PostgreSQL:



A mesma classe `User` poderá ser usada pelo TypeORM para criar o seguinte comando SQL para o SQLite:

```
CREATE TABLE "users" (
   "id" integer PRIMARY KEY AUTOINCREMENT NOT NULL,
   "mail" varchar(50) NOT NULL,
   "password" varchar(100) NOT NULL,
   CONSTRAINT "UQ_2e5b50f4b7c081eceea476ad128" UNIQUE ("mail")
);
```

> Decorators são criados pela chamada de uma função com a 1ª letra maiúscula, e precedidos por um "@"como, @Entity e @Column.
>
> Os decorators permitem adicionar metadados a classes, métodos, propriedades ou parâmetros de funções. Esses metadados são interpretados por algum framework/biblioteca para dar significado ao código. Como exemplo, o decorator @Entity só faz sentido para a biblioteca TypeORM.

iii. Data Source

O data source (fonte de dados) é a configuração que define como a aplicação se conectará ao SGBD. Essa configuração inclui dados essenciais para estabelecer a conexão com o BD, como o tipo de BD (PostgreSQL, MySQL, SQLite etc.), a URL de conexão, as credenciais de autenticação (usuário e senha) etc.

O objeto DataSource (https://typeorm.io/data-source-options), do TypeORM, mantém as configurações de conexão do BD e estabelece a conexão inicial com o BD ou pool de conexões. Para testar, coloque o código a seguir no arquivo data-source.ts. Veja que o DataSource recebe um JSON com os dados necessários para fazer a conexão com o BD no SGBD ou arquivo no caso do SQLite.

```
import { DataSource } from "typeorm";

//configuração para o SGBD PostgreSQL
const PgDataSource = new DataSource({
    database: 'bdaula', // se for SQLite, então use o arquivo bdaula.db
    type: "postgres", // se for SQLite, então use sqlite
    host: 'localhost', // não use esta propriedade se for sqlite
    port: 5432, // não use esta propriedade se for sqlite
    username: 'postgres', // não use esta propriedade se for sqlite
    password:'123', // não use esta propriedade se for sqlite
    // true indica que o schema do BD será criado a cada vez que a aplicação inicializar
    // deixe false ao usar migrations
    synchronize: false,
    logging: false, // true indica que as consultas e erros serão exibidas no terminal
    entities: ["src/entities/*.ts"], // entidades que serão convertidas em tabelas
    migrations: ["src/migrations/*.ts"] // local onde estarão os arquivos de migração
});

PgDataSource.initialize()
    .then(() => {
        console.log("Data Source inicializado!")
    })
    .catch((e) => {
        console.error("Erro na inicialização do Data Source:", e)
    });

export default PgDataSource;
```

Se quiser usar o SQLite, então substitua as propriedades JSON do objeto DataSource por

```
//configuração para SQLite
const SqliteDataSource = new DataSource({
    database: 'bdaula.db',
    type: "sqlite",
    synchronize: false,
    logging: false,
    entities: ["src/entities/*.ts"],
    migrations: ["src/migrations/*.ts"]
});
```

A conexão com o SGBD é estabelecida ao chamar o método initialize, do objeto DataSource. O método de inicialização do DataSource precisa ser chamado na inicialização do aplicativo e o método destroy deveria ser chamado após terminar o uso do BD. Porém, em aplicações servidoras, o backend precisa permanecer sempre em execução, ou seja, a conexão nunca precisará ser destruída.

Cada entidade precisa estar registrada na propriedade `entities` do DataSource, no nosso caso, endereçamos a pasta que colocaremos as classes que possuem as entidades.

Cada SGBD possui uma lista de propriedades, veja em https://typeorm.io/data-source-options.

iv. Migrations

Migrations (migrações) são uma maneira de gerenciar as alterações no esquema de BD de forma controlada e rastreável. As migrações são particularmente úteis quando estamos desenvolvendo uma aplicação que evolui ao longo do tempo e precisa atualizar a estrutura do BD de maneira consistente.

Para fazer uso das migrações no TypeORM adicione as seguintes propriedades no arquivo `package.json`:

```
"scripts": {
  "start": "ts-node ./src",
  "dev": "ts-node-dev ./src",
  "migration:generate": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:generate
./src/migrations/default",
  "migration:run": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:run",
  "migration:revert": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:revert"
},
```

O TypeORM fornece os seguintes comandos para gerir as migrações:

- `migration:generate`: cria um arquivo de migração. O nome desse arquivo é formado pelo timestamp (horário de criação do arquivo) + a palavra `default` que colocamos na linha de comando `./src/migrations/default`. A seguir tem-se o comando para rodar o script para gerar a migração:

```
PS D:\aula4> npm run migration:generate   ← Rodar o script para gerar a migração

> aula4@1.0.0 migration:generate
> typeorm-ts-node-commonjs -d ./src/data-source.ts migration:generate ./src/migrations/default

                                  Arquivo criado com os comandos SQL
Data Source inicializado!                 ↓
Migration D:\aula4\src\migrations/1693609461632-default.ts has been generated successfully.
```

O arquivo criado está na pasta `migrations` e possui uma classe com os métodos `up` e `down`. O método `up` possui os comandos SQL para aplicar a migração no SGBD e o método `down` possui o código para reverter a migração.

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

export class Default1693609461632 implements MigrationInterface {
    name = 'Default1693609461632'

    public async up(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`CREATE TABLE "users" (
                                    "id" SERIAL NOT NULL,
                                    "mail" character varying(50) NOT NULL,
                                    "password" character varying(100) NOT NULL,
                                    CONSTRAINT "UQ_2e5b50f4b7c081ee476ad128" UNIQUE ("mail"),
                                    CONSTRAINT "PK_a3ffb1c0c8416b9c907b7433" PRIMARY KEY ("id")
                                )`);
    }
```

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

- `migration:run`: submete os comandos SQL da migração no SGBD. Na prática serão submetidos os comandos SQL que estão no método `up` da classe que está no arquivo mais recente da pasta `migrations`. O caminho da pasta precisa ser especificado na propriedade `migrations` do DataSource no arquivo src/data-source.ts.

```
PS D:\aula4> npm run migration:run

> aula4@1.0.0 migration:run
> typeorm-ts-node-commonjs -d ./src/data-source.ts migration:run
```

O resultado foi a criação das tabelas `users` e `migrations` no SGBD PostgreSQL. A tabela `migrations` possui os dados da última migração:

```
select *
from migrations;
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	timestamp bigint	name character varying
1	1	1693609461632	Default1693609461632

- `migration:revert`: submete os comandos SQL para desfazer a última alteração no SGBD. Na prática serão submetidos os comandos SQL que estão no método `down` da classe que está no arquivo mais recente da pasta `migrations`:

```
select *
from migrations;
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	timestamp bigint	name character varying

Vantagens das migrações:

- Alterações no esquema: é comum a modelagem do BD sofrer alterações durante a evolução do desenvolvimento da aplicação. As migrações facilitam o processo de alteração das tabelas;
- Registros de migração: em vez de fazer essas alterações diretamente no BD, criamos migrações que são essencialmente scripts que descrevem as alterações. Cada migração registra as operações SQL necessárias para aplicar ou reverter a alteração;
- Controle de versão: as migrações são organizadas em uma sequência, onde novas migrações são adicionadas à medida que desenvolvemos a aplicação. Isso nos permite manter um histórico controlado das alterações ao longo do tempo;

- Aplicação de migrações: para submeter as alterações no BD, basta executarmos o comando TypeORM para aplicar as migrações pendentes. Isso fará com que o TypeORM execute as migrações na ordem apropriada, garantindo que o BD seja atualizado de acordo com as últimas alterações no esquema;
- Reversão de migrações: além de aplicar migrações, também podemos reverter migrações se for necessário desfazer uma alteração anterior no BD. Isso é especialmente útil em cenários de desenvolvimento e teste;
- Trabalho em equipe: migrações são úteis em ambientes de desenvolvimento colaborativo, pois permitem que os membros da equipe sincronizem facilmente as alterações de esquema em diferentes instâncias de BD.

v. Entidades e relacionamentos

Entidade é uma classe que mapeia para uma tabela do BD (https://typeorm.io/entities). O TypeORM faz uso extensivo de decorator) para configurar as entidades. A classe entidade possui o decorator `@Entity` e cada propriedade que recebe os decorators @Column ou @PrimaryGeneratedColumn será mapeada para uma coluna da tabela do BD.

Existem os seguintes tipos de relacionamentos entre as tabelas (https://typeorm.io/relations):

- Um-para-um: usando o decorator @OneToOne;
- Muitos-para-um: usando o decorator @ManyToOne;
- Um-para-muitos: usando o decorator @OneToMany;
- Muitos-para-muitos: usando o decorator @ManyToMany.

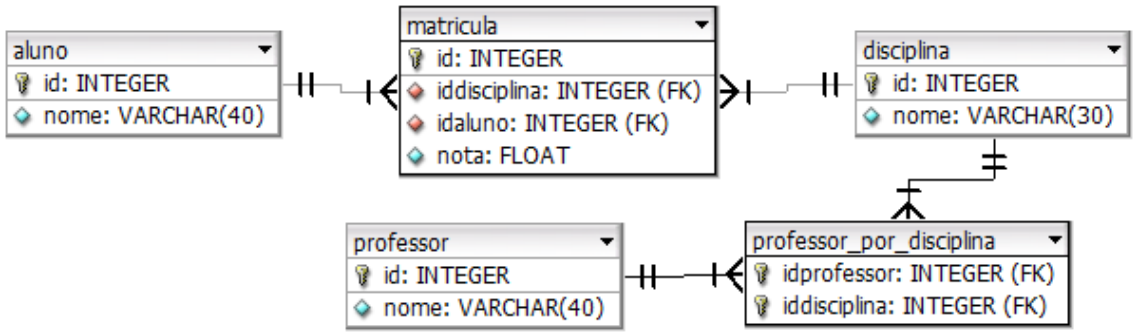
O relacionamento muitos-para-um entre as instâncias das tabelas usuários (`users`) e gastos (`spents`) determina que uma instância de usuário pode estar relacionada com muitas instâncias de gastos. Coloque o código a seguir no arquivo src/entities/Spent.ts. Ele possui o código da entidade `Spent` com o relacionamento de chave estrangeira na coluna `iduser`.

```
import { Entity, PrimaryGeneratedColumn, ManyToOne, JoinColumn, Column } from "typeorm";
import { User } from "./User";

@Entity({ name: "spents" })
export class Spent {
    @PrimaryGeneratedColumn()
    id: number;

    // cascade define que ao excluir o usuário os gastos serão excluídos
    @ManyToOne(() => User, { onDelete: 'CASCADE' })
    // JoinColum é usado para definir o lado da relação que contém a "join column" com a FK
    @JoinColumn({
        name: "iduser", // nome da coluna ser criada na tabela spents
        // pode ser qualquer nome usado para você identificar a FK
        foreignKeyConstraintName: "fk_user_id"
    })
    user: User; // a propriedade user recebe a referência para um objeto do tipo User

    @Column({ length: 50, nullable: false })
    description: string;

    @Column({ type: 'decimal', precision: 10, scale: 2, nullable: false })
    value: number;
```

```
}
```

O decorator `@JoinColum` é usado para definir o lado da relação que contém a coluna de junção com a chave estrangeira. Ele também é usado para definir o nome da coluna a ser criada (`name:"iduser"`) e o nome da restrição de chave estrangeira (`foreignKeyConstraintName:"fk_user_id"`) entre outras opções.
O decorator `@ManyToOne` recebe uma função que retorna a entidade referenciada (`() => User`) e outras opções, como o tipo de ação ao fazer um delete na tabela users (`{ onDelete: 'CASCADE' }`).

No exemplo a seguir, o relacionamento entre os tipos de entidades aluno e disciplina é muitos-para-muitos, uma vez que, um aluno pode se matricular em várias disciplinas e uma disciplina pode ser cursada por vários alunos. A tabela matrícula recebe o relacionamento muitos-para-muitos entre as entidades aluno e disciplina.



O código a seguir implementa o relacionamento muitos-para-muitos usando TypeORM.
Foi necessário criar a entidade matrícula para receber o relacionamento muitos-para-muitos pelo fato de o relacionamento possuir a propriedade nota, mas não é necessário criar entidade quando o relacionamento muitos-para-muitos não tiver propriedade, assim como no relacionamento entre professor e disciplina.

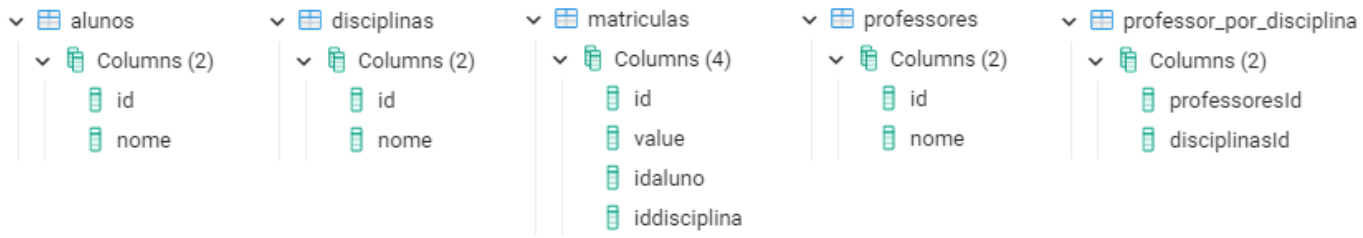
```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn, OneToMany } from
"typeorm";

@Entity({name:"alunos"})
export class Aluno {
    @PrimaryGeneratedColumn()
    id: number;

    @Column({nullable: false, length: 40})
    nome: string;

    @OneToMany(() => Matricula, matricula => matricula.aluno)
    matriculas: Matricula[]
}

@Entity({name:"disciplinas"})
export class Disciplina {
    @PrimaryGeneratedColumn()
    id: number;

    @Column({nullable: false, length: 30})
```

```
    nome: string;

    @OneToMany(() => Matricula, matricula => matricula.disciplina)
    matriculas: Matricula[]
}

@Entity({name:"matriculas"})
export class Matricula {
    @PrimaryGeneratedColumn()
    id: number;

    @ManyToOne(() => Aluno, (aluno) => aluno.matriculas)
    @JoinColumn({name:"idaluno"})
    aluno: Aluno;

    @ManyToOne(() => Disciplina, (disciplina) => disciplina.matriculas)
    @JoinColumn({name:"iddisciplina"})
    disciplina: Disciplina;

    @Column({ type: 'float'})
    value: number;
}

@Entity({name:"professores"})
export class Professor {
    @PrimaryGeneratedColumn()
    id: number;

    @Column({nullable: false, length: 40})
    nome: string;

    @ManyToMany(() => Disciplina)
    @JoinTable({name:"professor_por_disciplina"})
    disciplinas: Disciplina[]
}
```

Os relacionamentos podem ser unidirecionais e bidirecionais. Um relacionamento é considerado bidirecional se é possível navegar da entidade A para a entidade B e vice-versa. A implementação do relacionamento bidirecional se dá com os decorators @OneToMany de um lado e @ManyToOne do outro lado. No exemplo anterior o relacionamento é bidirecional, as entidades aluno e disciplina possuem o decorator `@OneToMany` e a entidade matrícula possui o decorator `@ManyToOne`. Lembre-se que somente um dos lados pode receber o decorator `@JoinColumn`.

O relacionamento muitos-para-muitos entre as entidades professor e disciplina foi criado usando os decorators `@ManyToMany` e `@JoinTable` em um dos lados do relacionamento.

O código anterior criará as seguintes tabelas no SGBD PostgreSQL:



vi. Controllers

Em uma arquitetura Model-View-Controller (MVC), os controladores são componentes responsáveis por gerenciar a lógica de negócios da aplicação. Eles atuam como intermediários entre as solicitações do cliente (através de rotas HTTP) e os modelos de dados representados pelas entidades do TypeORM.

O papel dos controladores em uma estrutura MVC com o TypeORM:

- Model (Modelo): os modelos representam as entidades de dados do aplicativo, geralmente mapeadas diretamente para tabelas no BD usando o TypeORM. Os modelos definem a estrutura dos dados e como eles são armazenados e recuperados no BD;
- View (Visualização): a visualização em um aplicativo MVC geralmente lida com a apresentação dos dados aos usuários. A visualização não se aplica no contexto de uma aplicação servidora, mesmo que a visualização por vezes possa ser interpretada como os dados são formatados para serem enviados como respostas a solicitações HTTP;
- Controller (Controlador): são responsáveis por receber as solicitações do cliente, interagir com os modelos (usando o TypeORM) para recuperar ou manipular dados e retornar uma resposta apropriada ao cliente. Eles implementam a lógica de negócios do aplicativo.

A função controladora é a função objetivo numa rota, ou seja, aquela função que processa a requisição. Geralmente, cada controlador é responsável por fazer o CRUD (Create, Read, Update, Delete) numa entidade. Coloque o código a seguir no arquivo src/controllers/UserController.ts.

```
import AppDataSource from "../data-source";
import { Request, Response } from 'express';
import { User } from '../entities/User';

class UserController {
  public async create(req: Request, res: Response): Promise<Response> {
    const { mail, password } = req.body;

    const obj = new User();
    obj.mail = mail;
    obj.password = password;
    const user = await AppDataSource.manager.save(obj);
    return res.json(user);
  }

  public async list(_: Request, res: Response): Promise<Response> {
    const users = await AppDataSource.manager.find(User);
    return res.json(users);
  }
```

```
  public async delete(req: Request, res: Response): Promise<Response> {
    const { id } = req.body; // id do registro a ser excluído
    const user = await AppDataSource.manager.findOneBy(User,{id});
    if( user ){
        await AppDataSource.manager.delete(User,user);
        return res.json(user);
    }
    return res.json({erro:`${id} inexistente`})
  }

  public async update(req: Request, res: Response): Promise<Response> {
    const { id, mail, password } = req.body;
    const user:User = await AppDataSource.manager.findOneBy(User,{id});
    if( user ){
        user.mail = mail;
        user.password = password;
        await AppDataSource.manager.save(user);
        return res.json(user);
    }
    return res.json({erro:`${id} inexistente`})
  }

}

export default new UserController();
```

O DataSource (definido no arquivo data-source.ts) provê os objetos EntityManager (https://typeorm.io/entity-manager-api) e Repository (https://typeorm.io/select-query-builder) para fazermos as operações de CRUD nas tabelas. Pela sua simplicidade, usamos o objeto EntityManager na classe UserController.

O objeto DataSource possui os seguintes métodos entre outros:

- save: insere ou atualiza um registro na tabela do BD. Ele foi usado nos métodos create e update da classe UserController;
- delete: remove um registro na tabela do BD. Ele foi usado no método delete da classe UserController;
- find: faz um select na tabela do BD para listar vários registros (https://typeorm.io/find-options). Ele foi usado no método list da classe UserController;
- findOneBy: faz um select na tabela do BD para retornar um registro. Ele foi usado nos métodos delete e update da classe UserController para obter o registro a ser excluído e atualizado, respectivamente.

Você pode acessar o gerenciador de entidades via DataSource. Exemplo de como usá-lo:

Para testar o controlador, coloque o código nos arquivos a seguir:

Arquivo src/routes/user.ts:

```
import { Router } from "express";
import UserController from "../controllers/UserController";
```

```
const routes = Router();

routes.post('/', UserController.create);
routes.get('/', UserController.list);
routes.delete('/', UserController.delete);
routes.put('/', UserController.update);

export default routes;
```

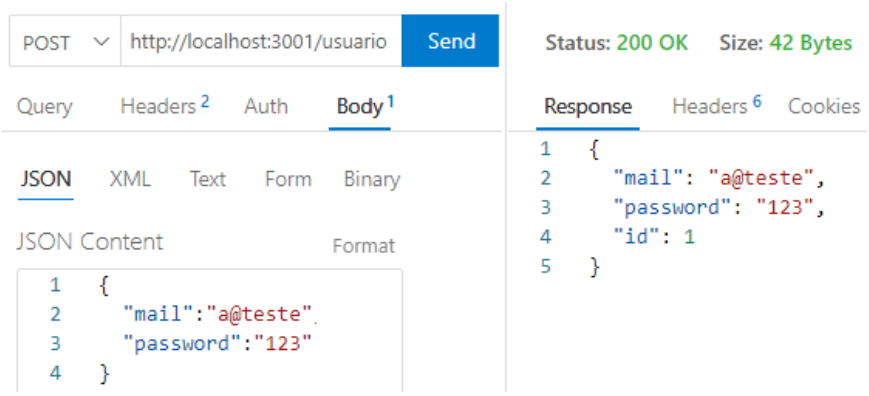
Arquivo src/routes/index.ts:

```
import { Router, Request, Response } from "express";
import user from './user';

const routes = Router();

routes.use("/usuario", user);

//aceita qualquer método HTTP ou URL
routes.use( (_:Request,res:Response) => res.json({error:"Requisição desconhecida"}) );

export default routes;
```

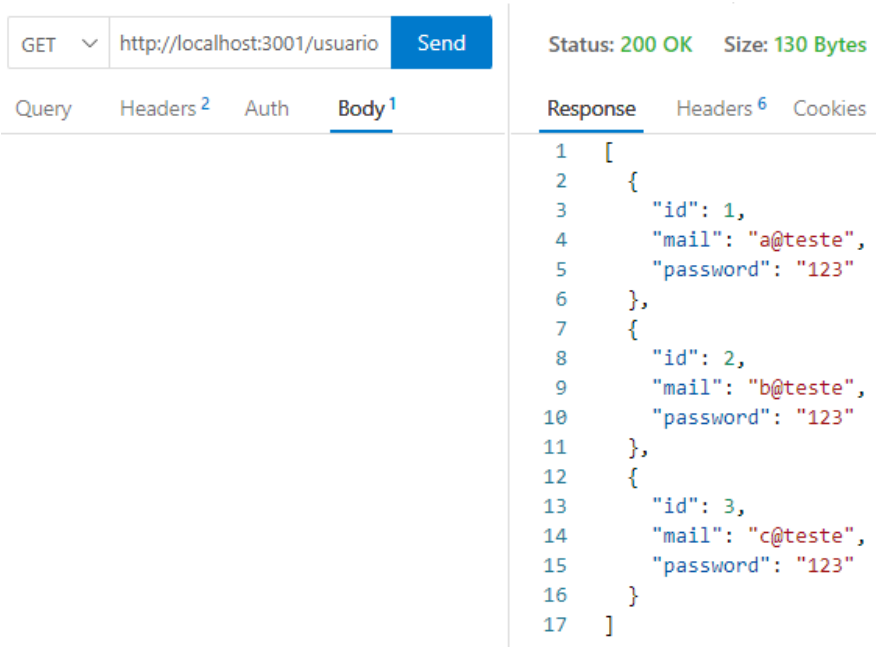
Arquivo src/index.ts:

```
import express from "express";
import routes from './routes';
import dotenv from "dotenv";
dotenv.config();

// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});

// define a rota para o pacote /routes
app.use(routes);
```

Teste da rota para inserir um registro na tabela users:



Teste da rota para listar os registros da tabela users:



Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

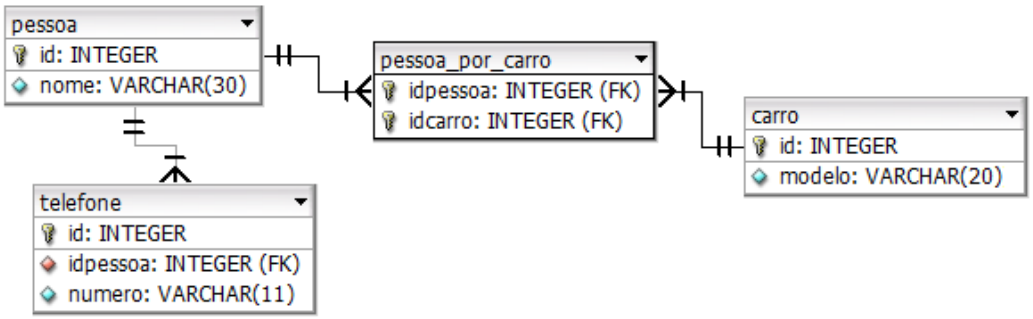
Exercício 1 - https://youtu.be/i06FxuR2PPs

Exercício 2 - https://youtu.be/FuqIT46VHFA

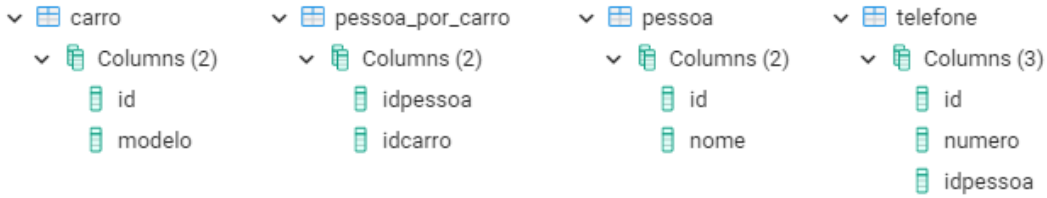
Exercício 3 - https://youtu.be/UYBlticPUt4

Exercício 4 - https://youtu.be/YUSp5l5E9lE

Exercício 1 – Codificar as entidades representadas no modelo a seguir usando TypeORM.



O resultado deverá ser as seguintes tabelas no SGBD PostgreSQL:

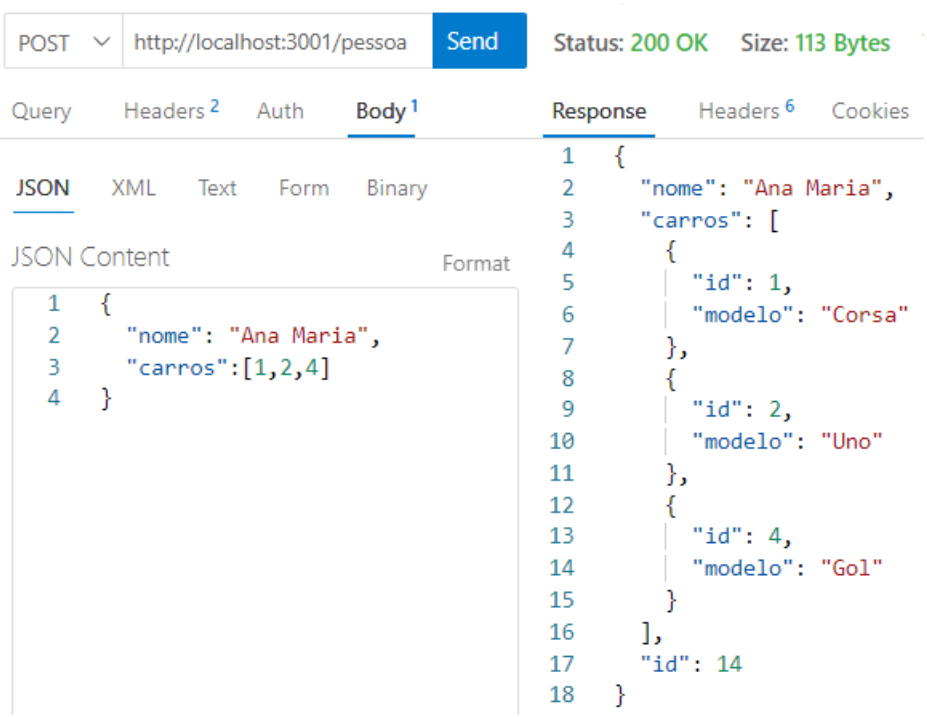


Dicas:

- Crie as classes Carro, Pessoa e Telefone como entidades;
- Como a tabela pessoa_por_carro não possui propriedades, além das chaves estrangeiras, então podemos criar o relacionamento muitos-para-muitos nas entidades carro ou pessoa (somente em uma delas) adicionando os decorators `@ManyToMany` e `@JoinTable`;
- Para renomear as colunas idpessoa e idcarro, na pessoa_por_carro, use as propriedades `joinColumn` e `inverseJoinColumn` no decorator `@JoinTable`. Essas propriedades são do tipo `JoinColumnOptions` e possui a propriedade `name` para renomear a coluna (use a navegação do VS Code para navegar pelas propriedades e ver o help);
- Como o relacionamento entre pessoa e telefone é um-para-muitos (1 pessoa possui muitos telefones), então adicione uma propriedade na entidade telefone com os decorators `@ManyToOne` e `@JoinColumn`.

Exercício 2 – Codificar o controlador e as rotas para fazer o CRUD na entidade carro. Ao lado tem-se o exemplo da requisição para listar os registros na tabela carro.

Exercício 3 – Codificar o controlador e as rotas para fazer o CRUD na entidade pessoa. Ao fazer as operações na tabela pessoa deverá ser persistido os registros na tabela pessoa_por_carro. A seguir tem-se o exemplo da requisição para inserir um registro na tabela pessoa, o array de carros corresponde aos ids das instâncias na tabela de carro.



Dicas:

- Antes de fazer o save na entidade pessoa tem-se de obter os objetos na entidade carro que possuem os ids fornecidos. No exemplo anterior temos de obter os carros que possuem os ids [1,2,4]. O código a seguir obtém os registros que possuem os ids que estão no array carros:

```
const cars = await AppDataSource.manager.findBy(Carro,{
  id: In(carros), // use o array de IDs como parte dos critérios
```

```
});
```

- Ao listar os registros da tabela pessoa deverão ser listados os carros da pessoa. Use a propriedade relations na busca (https://typeorm.io/many-to-many-relations#loading-many-to-many-relations):

```
const pessoas = await AppDataSource.manager.find(Pessoa,{
      relations:{
        carros:true  // carros é uma propriedade da entidade pessoa
      }
    });
```

Exercício 4 – Codificar o controlador e as rotas para fazer o CRUD na entidade telefone.