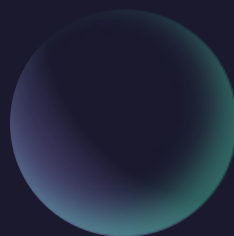


Gerenciador de Bibliotecas


Trabalho Final de Curso Back End

Dupla: Anderson e Luiz



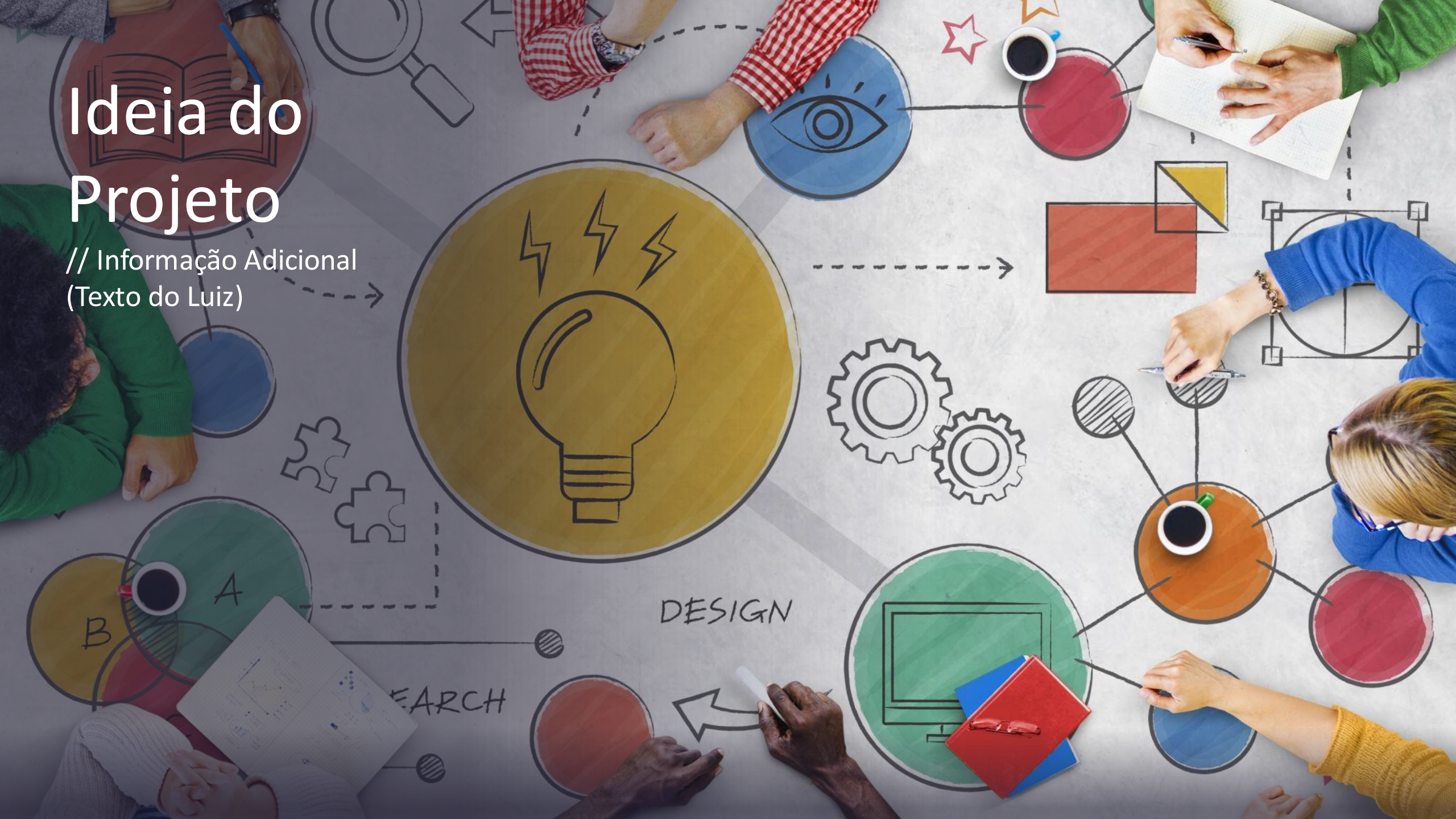


Introdução

- Ideia do Projeto
 - Problema e Solução
 - Diagrama de Entidades
 - Linhas de Código
 - Explicação
 - Requisições
 - Referências
 - Contracapa do Trabalho
- 

Ideia do Projeto

// Informação Adicional
(Texto do Luiz)



Descrição

Nosso trabalho final foi desenvolvido para melhorar o desempenho de uma biblioteca que deseja utilizar o Banco de Dados MySQL para armazenar os registros da entrada e saída de livros, através de uma API podemos definir rotas que realizem as operações CRUD (Create, Read, Update, Delete). Desta maneira o atendente da biblioteca pode facilmente monitorar os dados e atualiza-los sempre que necessário para que possa obter um resultado satisfatório em seu dia de trabalho.

CRUD é um acrônimo que representa as quatro operações básicas realizadas em bancos de dados ou sistemas de arquivos. Essas operações são essenciais para interagir com dados e garantir a consistência e integridade das informações. Vamos dar uma olhada em cada uma delas:

- **Create (Criar):** Essa operação é usada para adicionar novos registros ao banco de dados. Por exemplo, se você estiver desenvolvendo um aplicativo de gerenciamento de estoque, pode usar a operação Create para adicionar um novo item ao estoque.
- **Read (Ler):** A operação Read é usada para recuperar dados do banco de dados. Por exemplo, se você quiser exibir todos os itens em estoque no seu aplicativo, usaria a operação Read para buscar essas informações.
- **Update (Atualizar):** Com a operação Update, é possível modificar registros existentes no banco de dados. Por exemplo, se a quantidade de um item em estoque mudar, você usaria a operação Update para refletir essa alteração.
- **Delete (Excluir):** A operação Delete permite remover registros do banco de dados. Se um item não estiver mais disponível no estoque, você usaria a operação Delete para excluí-lo.

Essas operações são fundamentais para o desenvolvimento de aplicativos e sistemas que precisam armazenar, consultar e gerenciar dados.



Problema e Solução

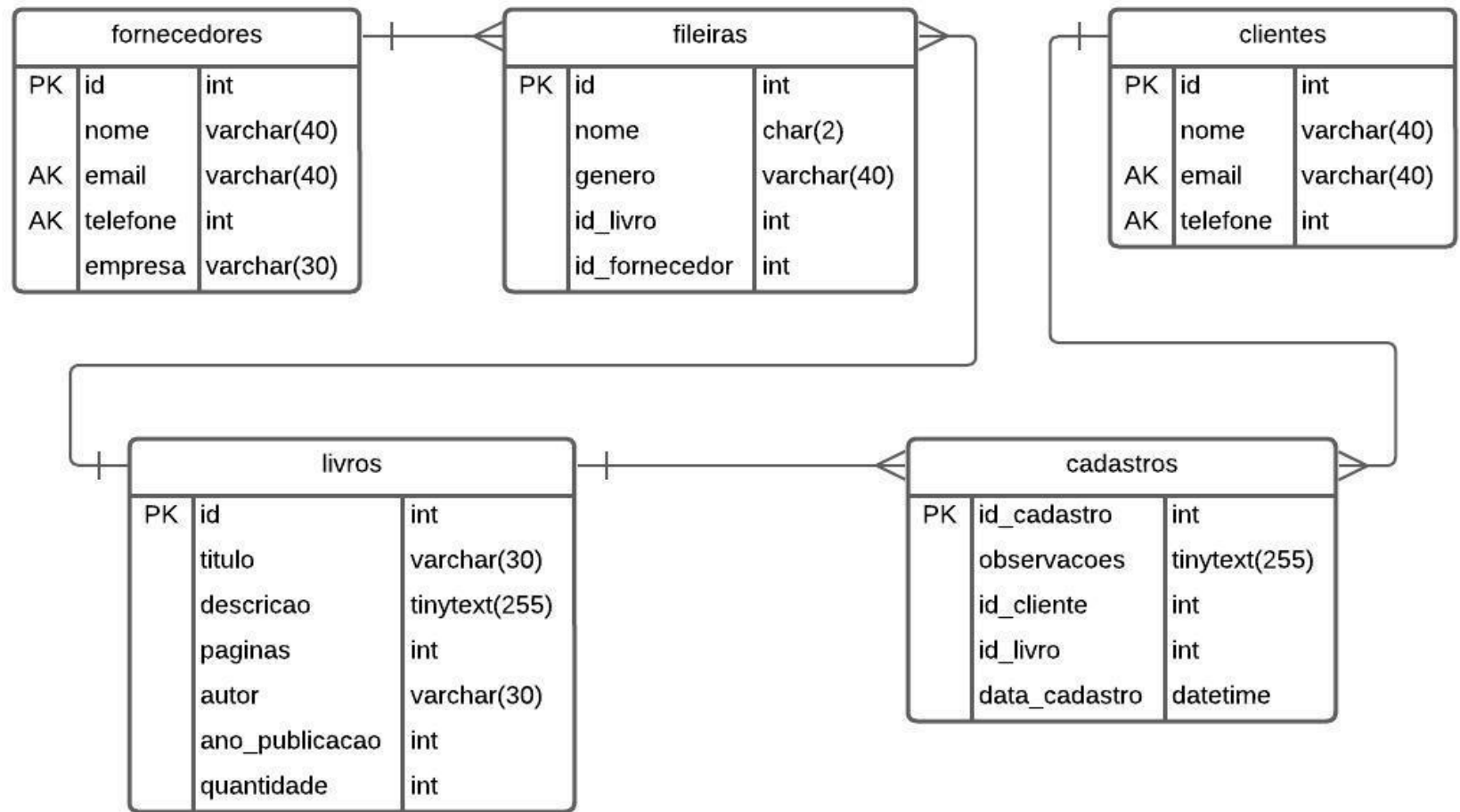
Devido ao aumento da demanda de livros e clientes a biblioteca foi obrigada a aumentar seu espaço físico e principalmente o digital, desta forma as planilhas já se tornaram obsoletas, causando assim um transtorno para os atendentes que acabam não conseguindo oferecer um serviço rápido e de qualidade para seus clientes, por esse motivo o dono da biblioteca nos procurou para solucionar o problema na gerência de dados.

Por este motivo, propomos a biblioteca a substituição da ferramenta Excel feita para inserir registros em planilhas pelo Banco de Dados MySQL que vem a ser usado para o mesmo fim, armazenar de forma organizada em linhas e colunas diferentes tipos de dados. Em adição, temos que comentar sobre como este recurso permitiria a vantagem de agilidade na inserção de registros e uma busca eficiência devido as relações entre as tabelas pelas chaves estrangeiras, resultando em uma consulta totalmente precisa.



Diagrama

Para maior entendimento de nosso trabalho, criamos um diagrama do Banco de Dados com as interações entre cada tabela.



Linhas de Código

Esse gerenciador de bibliotecas foi desenvolvido com a linguagem Typescript usando recursos de bibliotecas do NPM (gerenciador de pacotes para NodeJS)

```
... object to mirror_mod.mirror_object
operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True
```

```
selection at the end -add
mirror_ob.select= 1
modifier_ob.select=1
context.scene.objects.active
("Selected" + str(modifier_ob.name))
mirror_ob.select = 0
= bpy.context.selected_objects
data.objects[one.name].select
print("please select exactly
```

-- OPERATOR CLASSES -----

```
types.Operator):
    X mirror to the selected
    object.mirror_mirror_x"
    mirror X"
```

Primeiros Passos

Para começar foi necessário instalar as dependências que serão usadas durante o desenvolvimento, estas que são 'express', 'typescript', 'ts-node-dev', 'mysql2', '@types/express' e '@types/node'.

- Express: Um framework web para NodeJS que facilita a criação de aplicações web e APIs, fornecendo um conjunto robusto de funcionalidades para desenvolvimento de servidores web.
- Typescript: Um superconjunto de Javascript que adiciona tipagem estática opcional ao código, ajudando a detectar erros durante o desenvolvimento e melhorando a manutenção do código.
- TS-Node-Dev: Uma ferramenta que combina TS-Node e Nodemon, permitindo executar e reiniciar automaticamente aplicações NodeJS escritas em Typescript sempre que há mudanças no código.
- MySQL2: Um cliente MySQL para NodeJS que oferece suporte a promessas e async/await, além de ser mais rápido e eficiente em comparação com o módulo MySQL original.
- @types/express: Pacote de definições de tipos para o Express, permitindo que você use o Express com Typescript e tenha suporte a tipagem estática.
- @types/node: Pacote de definições de tipos para NodeJS, fornecendo tipagem estática para as APIs do NodeJS quando usado com Typescript.

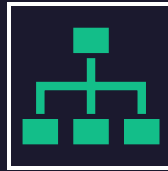
Organização no Visual Studio Code



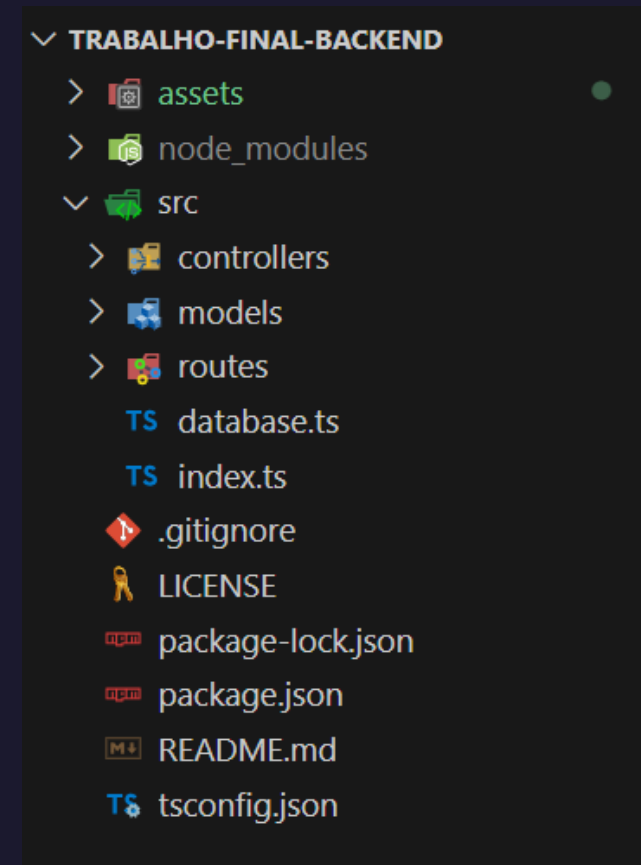
Depois de terem sido instaladas as bibliotecas prosseguimos com organização das pastas e arquivos.



As pastas 'node_modules' e 'src' (ou seja, source) estarão em conjunto dos arquivos 'gitignore', 'tsconfig.json', 'package.json' e 'package-lock.json' na pasta 'trabalho-final-backend'.



As pastas 'controllers', 'models' e 'routes' estarão em conjunto dos arquivos 'index.ts' e 'database.ts' dentro da pasta src (pasta citada anteriormente).



Pastas e Arquivos

Em uma aplicação TypeScript com Express, os conceitos de controllers, models e routes são fundamentais para estruturar o backend. Vou explicar brevemente cada um deles:

Controllers: Os controllers contêm a lógica da aplicação e lidam com o processamento das requisições do cliente. Eles são responsáveis por receber as requisições HTTP (como GET, POST, PUT, DELETE) e retornar as respostas adequadas. Um padrão comum de estruturação de uma aplicação Express é o Model-View-Controller (MVC), e os controllers fazem parte desse padrão. Em TypeScript, podemos usar classes para criar controllers e organizar a lógica de negócios.

Models: Os models representam a camada de dados da aplicação. Eles definem como os dados são armazenados e manipulados no banco de dados. Por exemplo, se você tem uma entidade “Usuário”, o model correspondente definiria os campos desse usuário e as operações relacionadas (como criar, atualizar, buscar etc.).

Routes: As rotas (ou routes) definem os endpoints da API. Elas mapeiam URLs específicas para os controllers apropriados. Por exemplo, uma rota “/usuarios” pode ser associada a um controller que lida com operações relacionadas a usuários. O uso do Express Router é comum para organizar as rotas em arquivos separados e manter o código limpo e modular.

Explicação

Controllers: Dentro da pasta de controllers, você normalmente encontra arquivos que definem as ações (métodos) relacionadas a cada rota. Cada rota (endpoint) da sua aplicação terá um arquivo de controller associado. Por exemplo, se você tem uma rota “/usuarios”, o arquivo de controller correspondente pode conter métodos como `listarUsuarios`, `criarUsuario`, `atualizarUsuario`, etc.

Models: Na pasta de models, você encontrará os arquivos que definem a estrutura dos dados da sua aplicação. Esses arquivos geralmente representam as tabelas do banco de dados ou outras entidades. Por exemplo, se você tem uma tabela “Usuários”, o arquivo de model correspondente definirá os campos dessa tabela e as relações com outras tabelas (se houver).

Routes: Dentro da pasta de rotas, você encontrará os arquivos que definem as rotas da sua aplicação. Cada rota é mapeada para um método no controller apropriado. Por exemplo, a rota “/usuarios” pode ser definida no arquivo de rotas como `router.get("/usuarios", UserController.listarUsuarios)`



```
1  import { Router } from "express";
2  import { CadastrosController } from "../controllers/cadastrosController";
3
4  const router = Router();
5  router.get('/cadastros', CadastrosController.getAll);
6  router.get('/cadastros/:id', CadastrosController.getById);
7  router.post('/cadastros', CadastrosController.create);
8  router.put('/cadastros/:id', CadastrosController.update);
9  router.delete('/cadastros/:id', CadastrosController.delete);
10
11 export default router;
```



```

1 import { Request, Response } from 'express';
2 import { Cadastro } from '../models/cadastro';
3
4 export class CadastrosController{
5   static async getAll(req: Request, res: Response){
6     try {
7       const cadastros = await Cadastro.getAll();
8       res.status(200).json(cadastros);
9     } catch(error) {
10      res.status(500).json({'message': error});
11    }
12  }
13   static async getById(req: Request, res: Response){
14     try {
15       const { id } = req.params;
16       const cadastro = await Cadastro.getById(parseInt(id, 10));
17       res.status(200).json(cadastro);
18     } catch(error) {
19       res.status(404).json({'message': error});
20     }
21   }
22   static async create(req: Request, res: Response){
23     try {
24       const { observacoes, idCliente, idLivro, dataCadastro } = req.body;
25       const resultado = await Cadastro.create(observacoes, idCliente, idLivro, dataCadastro);
26       res.status(201).json(resultado);
27     } catch(error) {
28       res.status(500).json({'message': error});
29     }
30   }
31   static async update(req: Request, res: Response){
32     try {
33       const { id } = req.params;
34       const { observacoes, idCliente, idLivro, dataCadastro } = req.body;
35       const resultado = await Cadastro.update(parseInt(id, 10), observacoes, idCliente, idLivro, dataCadastro);
36       res.status(200).json(resultado);
37     } catch(error) {
38       res.status(500).json({'message': error});
39     }
40   }
41   static async delete(req: Request, res: Response){
42     try {
43       const { id } = req.params;
44       const resultado = await Cadastro.delete(parseInt(id, 10));
45       res.status(200).json(resultado);
46     } catch(error) {
47       res.status(500).json({'message': error});
48     }
49   }
50 }

```

```

1 static async getAll(req: Request, res: Response){
2   try {
3     const cadastros = await Cadastro.getAll();
4     res.status(200).json(cadastros);
5   } catch(error) {
6     res.status(500).json({'message': error});
7   }
8 }
9 static async getById(req: Request, res: Response){
10  try {
11    const { id } = req.params;
12    const cadastro = await Cadastro.getById(parseInt(id, 10));
13    res.status(200).json(cadastro);
14  } catch(error) {
15    res.status(404).json({'message': error});
16  }
17 }

```

```
1 import { db } from "../database";
2
3 export class Cadastro{
4   private _idCadastro: number;
5   private _observacoes: string;
6   private _idCliente: number;
7   private _idLivro: number;
8   private _dataCadastro: string;
9
10  constructor(idCadastro: number, observacoes: string, idCliente: number, idLivro: number, dataCadastro: string){
11    this._idCadastro = idCadastro;
12    this._observacoes = observacoes;
13    this._idCliente = idCliente;
14    this._idLivro = idLivro;
15    this._dataCadastro = dataCadastro;
16  }
17
18  static async getAll(){
19    try {
20      const [rows] = await db.query('SELECT * FROM Cadastros');
21      return rows;
22    } catch {
23      throw new Error('Erro ao buscar cadastros no banco de dados.');
```

```
24    }
25  }
26  static async getById(idCadastro: number){
27    try {
28      const [rows] = await db.query('SELECT * FROM Cadastros WHERE id_cadastro = ?', [idCadastro]);
29      return rows;
30    } catch {
31      throw new Error('Erro ao buscar cadastro no banco de dados.');
```

```
32    }
33  }
34  static async create(observacoes: string, idCliente: number, idLivro: number, dataCadastro: string){
35    try {
36      const [rows] = await db.query('INSERT INTO Cadastros (observacoes, id_cliente, id_livro, data_cadastro) VALUES (?, ?, ?, ?)', [observacoes, idCliente, idLivro, dataCadastro]);
37      return rows;
38    } catch {
39      throw new Error('Erro ao inserir cadastro no banco de dados.');
```

```
40    }
41  }
42  static async update(idCadastro: number, observacoes: string, idCliente: number, idLivro: number, dataCadastro: string){
43    try {
44      const [rows] = await db.query('UPDATE Cadastros SET observacoes = ?, id_cliente = ?, id_livro = ?, data_cadastro = ? WHERE id_cadastro = ?', [observacoes, idCliente, idLivro, dataCadastro, idCadastro]);
45      return rows;
46    } catch {
47      throw new Error('Erro ao atualizar cadastro no banco de dados.');
```

```
48    }
49  }
50  static async delete(idCadastro: number){
51    try {
52      const [rows] = await db.query('DELETE FROM Cadastros WHERE id_cadastro = ?', [idCadastro]);
53      return rows;
54    } catch {
55      throw new Error('Erro ao deletar cadastro no banco de dados.');
```

```
56    }
57  }
58 }
```

Banco de Dados

Foram usados `CREATE DATABASE NOME_DO_BANCO;` Para criar o banco de dados.

`USE NOME_DO_BANCO;` Para acessar o devido banco de dados.

`CREATE TABLE NOME_DA_COLUNA;` Para criação das colunas, com identificações, chaves primarias e estrangeiras.

`INSERT INTO NOME_DA_COLUNA(`

`.....`

`.....`

`.....`

`);` Para inserir os dados que serão acessados pelo código de programação.


```
63 • INSERT INTO Fornecedores (id, nome, email, telefone, empresa) VALUES
64 (default, "William", "william@gmail.com", 974538434, "Universal Files"),
65 (default, "Felipe", "felipe@gmail.com", 990325442, "Ler é Viver"),
66 (default, "Roberto", "roberto@gmail.com", 974843944, "Revisitando"),
67 (default, "Carlos", "carlos@gmail.com", 946769921, "Ler é Viver"),
68 (default, "Marcos", "marcos@gmail.com", 954927938, "Universal Files");
69 • INSERT INTO Livros (id, titulo, descricao, paginas, autor, ano_publicacao, quantidade) VALUES
70 (default, "A Árvore do Jardim", "Um menino decide construir uma casa na árvore de seu próprio quintal", 200, "Lúcio", 2017, 20),
71 (default, "Que bicho te mordeu?", "Uma menina age rebeldia depois de ter seu pedido negado", 100, "Murilo", 2019, 30),
72 (default, "Robert: O Urso de Pelúcia", "Um brinquedo repentinamente ganha consciência e busca entender a sociedade humana", 200, "Muri
73 (default, "A Rosa Espinhosa", "Um detetive tenta desvendar um caso assassiinato misterioso", 300, "Paul", 2017, 100),
74 (default, "O Mundo na Real", "Uma jornada cheia de descobertas sobre a espécie humana", 200, "Webber", 2013, 50);
75 -- a => adulto ex AB
76 -- b => juvenil ex BD
77 -- c => infantil ex CA
78 -- d -> livre ex DO
79 • INSERT INTO Fileiras (id, nome, genero, id_fornecedor, id_livro) VALUES
```

Linhas de Código

Por exemplo, comentaremos um pouco sobre este bloco de código do arquivo 'cadastroController.ts'.

Nele podemos observar uma função estática e assíncrona com os parâmetros req e res.

```
1 static async update(req: Request, res: Response){  
2   try {  
3     const { id } = req.params;  
4     const { observacoes, idCliente, idLivro, dataCadastro } = req.body;  
5     const resultado = await Cadastro.update(parseInt(id, 10), observacoes, idCliente, idLivro, dataCadastro);  
6     res.status(200).json(resultado);  
7   } catch(error) {  
8     res.status(500).json({'message': error});  
9   }  
10 }
```

Uma função estática permite que a função seja chamada diretamente de dentro de uma classe (não precisando de uma instância) e a característica assíncrona impede que o sistema pare após as consultas, que requerem um determinado tempo para serem concluídas pelo banco de dados. Também temos um try/catch para verificar se há algum erro e notificar com o código HTTP e a mensagem correspondente. Por fim, com as variáveis constantes conseguimos armazenar os dados de requisição e usamos eles dentro dos parênteses para chamar a função update do cadastro (model).

No exemplo mostrado, exibimos em formato JSON o resultado do update e através do res o código HTTP de sucesso ('OK'), sendo que o parseInt dentro dos parênteses converte o resultado da busca em um inteiro.

APIs e Requisições

As APIs (Interfaces de Programação de Aplicações) estão presentes em todas as aplicações que usamos no dia a dia. Elas permitem a comunicação entre diferentes sistemas e aplicativos, facilitando tarefas como obter localizações, informações climáticas, realizar pagamentos, rastrear encomendas e muito mais. Vou apresentar alguns exemplos de requisições em APIs:

Requisições HTTP:

GET: Recupera dados de um servidor. Por exemplo: GET /users/bruno/claims Retorna uma lista de “claims” do usuário “bruno”.

POST: Envia dados para o servidor. Por exemplo: POST /users/bruno/claims Cria uma “claim” para o usuário “bruno”.

PUT: Atualiza dados no servidor. Por exemplo: PUT /users/bruno/claims/6 Atualiza a “claim” com ID 6 do usuário “bruno”.

DELETE: Está solicitando que o servidor remova um recurso específico. Por exemplo: DELETE /users/bruno



```
> ts-node-dev src/terminal.ts
```

```
[INFO] 12:52:28 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typescript ver. 5.5.4)
```

```
----- Menu Inicial -----
```

1. Cadastros
2. Fornecedores
3. Livros
4. Fileiras
5. Clientes
6. Encerrar

```
-----
```

```
Escolha uma das opcoes: 3
```

```
----- Menu Inicial -----
```

1. Listar todos
2. Listar por Id
3. Inserir
4. Modificar
5. Deletar
6. Encerrar

```
-----
```

```
Escolha uma das opcoes: 1
```

```
[{"id":1,"titulo":"A Árvore do Jardim","descricao":"Um menino decide construir uma casa na árvore de seu próprio quintal","paginas":200,"autor":"Lúcio","ano_publicacao":2017,"quantidade":20}, {"id":2,"titulo":"Que bicho te mordeu?","descricao":"Uma menina age rebeldia depois de ter seu pedido negado","paginas":100,"autor":"Murilo","ano_publicacao":2019,"quantidade":30}, {"id":3,"titulo":"Robert: O Urso de Pelúcia","descricao":"Um brinquedo repentinamente ganha consciência e busca entender a sociedade humana","paginas":200,"autor":"Murilo","ano_publicacao":2015,"quantidade":40}, {"id":4,"titulo":"A Rosa Espinhosa","descricao":"Um detetive tenta desvendar um caso assassiinato misterioso","paginas":300,"autor":"Paul","ano_publicacao":2017,"quantidade":100}, {"id":5,"titulo":"O Mundo na Real","descricao":"Uma jornada cheia de descobertas sobre a espécie humana","paginas":200,"autor":"Webber","ano_publicacao":2013,"quantidade":50}]
```

```
-----
```


Referências

Google

- W3Schools (Site)
- SQL Server Tutorial (Site)

Youtube

- Curso em Vídeo (Canal)
- Felipe Rocha (Canal)
- Dev Aprender (Canal)





Obrigado por sua atenção!

