

Minicurso

RESTful API Node.js +

Express + PostgreSQL

Guilherme Gomes

`guilherme.gomes@academico.ifpb.edu.br`

Danilo Marques

`danilo.marques@academico.ifpb.edu.br`

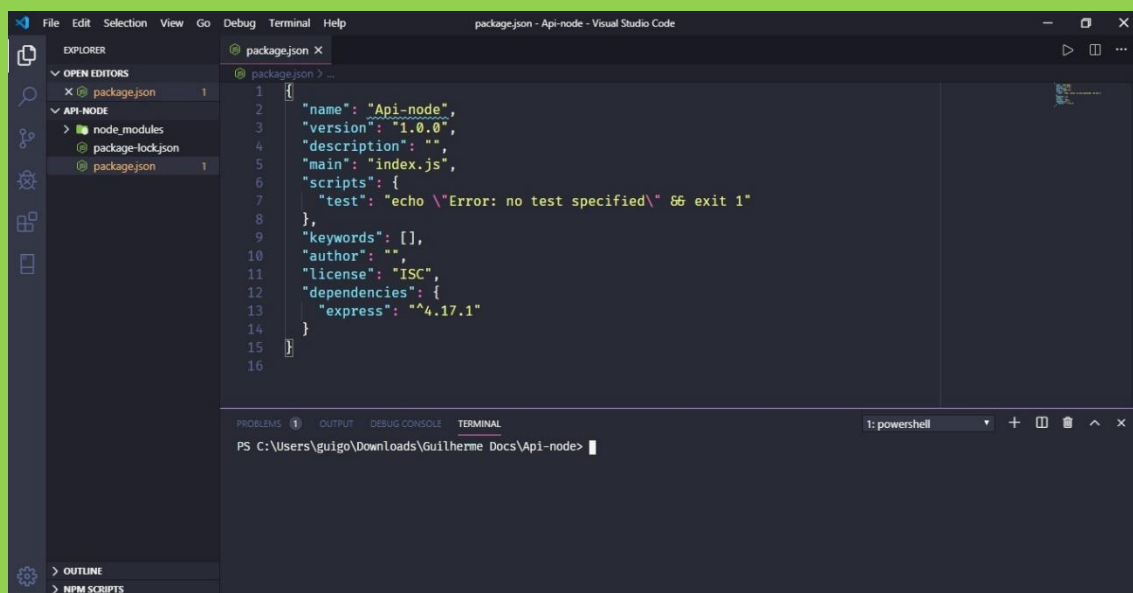
Seção 01 – Criação do banco de dados

1. Abra o psql (shell script do postgres) e tecla enter até pedir o password.
2. Digite a senha + Enter.
3. Digite \l + Enter. (Você verá a lista de todos os banco de dados disponíveis.)
4. Acesse: <https://github.com/guigomes94/minicurso-node-postgres/blob/master/musicfy.sql>
5. Copie e cole o script sql no psql para criar o banco e povoa-lo.
6. Por enquanto minimize e agora crie uma pasta com o nome que você quer usar no projeto, aqui usaremos o nome “API-NODE”. Em seguida abra o VS Code.

Seção 02 – Preparando o Ambiente de Desenvolvimento

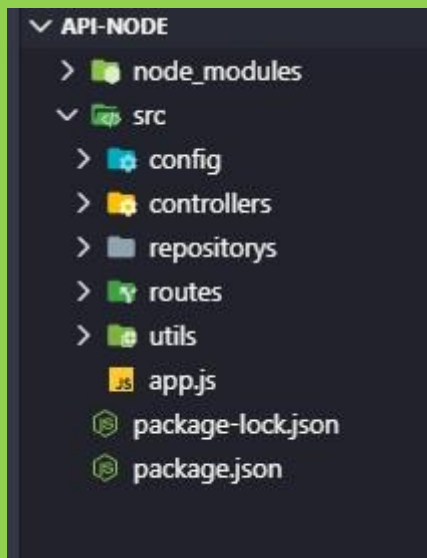
7. No VS Code vá em File -> Open Folder..., escolha a pasta que você criou.
8. Agora depois de abri vá em Terminal -> New Terminal.
9. No terminal digite node -v e depois npm -v, exibirá as versões que estão instaladas na sua máquina, isso nos comprova que as ferramentas estão funcionando corretamente, só assim poderemos prosseguir.
10. Digite agora npm init -y, gerará um arquivo chamado package.json, esse arquivo é muito importante, em hipótese alguma será excluído. Ele é responsável pelas configurações do sistema e onde fica armazenado as dependências instaladas pelo npm.
11. Agora novamente no terminal digite: npm install express.

Seu ambiente deve estar com essa aparência:



12. Vamos agora criar algumas pastas para deixar o código organizado.
13. Crie a pasta src onde ficará todo o nosso código.
14. Crie dentro da pasta src as subpastas: controllers, config, repositorys, utils e routes.
15. Crie o arquivo app.js dentro de src.

Semelhante ao exemplo a seguir:



Seção 03 – Criando o servidor Node.js + Express.js

16. Primeiro importar o express para o código: `const express = require("express");`.
17. Instância o express em uma variável: `const server = express();`.
18. Para sempre trabalharmos com json na resposta ative essa função da seguinte maneira: `server.use(express.json());`.
19. Para testar vamos criar uma requisição GET:
`server.get("/", (req, res) => {
 return res.json({ "info": "API NODE"});
});`.
20. Por fim, escolhemos uma porta: `server.listen(3333);`.

O código deve estar assim:

```
const express = require("express");
const server = express();

server.use(express.json());

server.get("/", (req, res) => {
    return res.send({ "info": "API NODE" });
});

server.listen(3333);
```

21. Abra o arquivo package.json e altere scripts, por "start": "node src/app.js".

```
"scripts": {  
  "start": "node src/app.js"  
},
```

22. Salve o arquivo e no terminal digite `npm start` e agora vá em qualquer navegador e coloque na url: `localhost:3333`.
23. Encerre o servidor, teclando `ctrl+c` no terminal.
24. Vamos incluir mais duas dependências: `npm install sucrase nodemon -D`.

O Node ainda não reconhece o javascript mais moderno, então vamos utilizar um compilador em nível de desenvolvimento para trabalharmos com o javascript mais atual, para isso instalamos o sucrase. Também instalamos o nodemon para automatizar o restart do servidor a cada alteração que fizermos. O `-D` no final do comando significa que são dependências de desenvolvimento, não serão utilizadas em produção.

25. Volte no arquivo `package.json` e altere novamente script, no lugar de `node` escreva `nodemon`.
26. Crie um arquivo chamado `nodemon.json` fora do `src` com o seguinte código:

```
nodemon.json > ...  
1 {  
2   "execMap": {  
3     "js": "node -r sucrase/register"  
4   }  
5 }
```

Desta maneira o nodemon executará o sucrase antes do node para compilar o código.

27. Altere o código em `app.js`, agora podemos utilizar o `'import'`, mais recente: `import express from "express";`.
28. No terminal dê o comando `'npm start'` para deixar nosso servidor no ar.

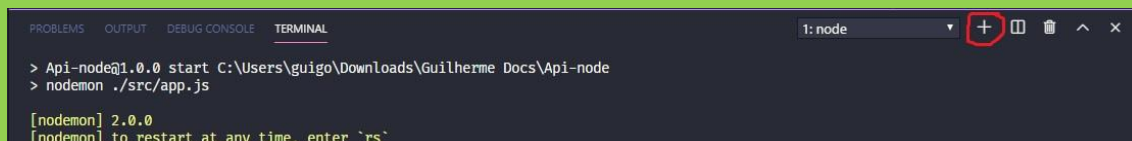
Código de `app.js` até o momento:

```
import express from 'express';  
const server = express();  
  
server.use(express.json());  
  
server.get("/", (req, res) => {  
  return res.send({ "info": "API NODE" });  
});
```

```
server.listen(3333);
```

Seção 04 – Conexão com o banco de dados

29. Abra um novo terminal, clique no botão '+', como no exemplo:

A screenshot of a Visual Studio Code interface. The 'TERMINAL' tab is active, showing a command prompt with the following text:

```
> Api-node@1.0.0 start C:\Users\guigo\Downloads\Guilherme Docs\Api-node  
> nodemon ./src/app.js  
  
[nodemon] 2.0.0  
[nodemon] to restart at any time, enter `rs`
```

 In the top right corner of the terminal window, there is a toolbar with several icons. A red circle highlights the '+' icon, which is used to open a new terminal instance.

30. Digite `npm install pg`, essa dependência nos permitirá trabalhar com o postgres no express.

31. Na pasta config crie o arquivo `conn.js`.

32. No arquivo `conn.js` importe apenas o Pool do pg.

33. No Pool precisaremos de alguns parâmetros conforme exemplo abaixo.

```
import { Pool } from 'pg';  
  
const pool = new Pool({  
  user: 'postgres',  
  host: 'localhost',  
  database: 'musicfy',  
  password: 'ifpb',  
  port: 5432,  
});  
  
module.exports = pool;
```

O Pool mantém a conexão ativa com o banco de dados e vamos utilizá-lo para realizar as nossas consultas SQL.

Seção 05 – Construindo rotas, controllers e repositorys

34. Crie um arquivo na pasta routes com o nome `routes.js`.

35. Vamos importar e instanciar a classe Router do express e fazer nossa primeira requisição GET ao banco. Para isso também devemos importar o pool no arquivo `routes.js`.

36. Temos quatro entidades no nosso banco, vamos começar com a entidade Artist.

```
import { Router } from 'express';  
import pool from '../config/conn';
```

```
const routes = new Router();

routes.get('/artist', (req, res) => {
  pool.query('SELECT * FROM artist', (err, result) => {
    if (err) {
      res.status(400).send(err.stack);
    }
    else {
      res.status(200).send(result.rows);
    }
  })
})

export default routes;
```

37. Agora volte no app.js e importe routes e habilite no servidor com a expressão `server.use(routes)`. O código deve estar assim:

```
import express from 'express';
import routes from '../src/routes/routes';

const server = express();

server.use(express.json());
server.use(routes);

server.get("/", (req, res) => {
  return res.json({ info: 'RESTful API Node.js + Express + Postgres' });
});

server.listen(3333);
```

38. Abra o navegador e veja se foi bem sucedido na url: `localhost:3333/artist`.

Realizamos nossa primeira consulta e vimos que está funcionando nossa comunicação com o banco de dados, vamos desenvolver as demais operações do CRUD e das entidades que faltam, porém, veja que o código em routes fica muito poluído dessa maneira, vamos dividir o nosso código em diferentes arquivos lhe dando funções bem específicas, isso torna o código mais legível e de fácil manutenção.

39. Primeiro no arquivo app.js remova a requisição GET e a coloque no arquivo routes.js.

app.js

```
import express from 'express';
import routes from '../src/routes/routes';

const server = express();

server.use(express.json());
server.use(routes);

server.listen(3333);
```

routes.js

```
import { Router } from 'express';
import pool from '../config/conn';

const routes = new Router();

routes.get('/', (req, res) => {
  return res.json({ "info": "API NODE" });
});

routes.get('/artist', (req, res) => {
  pool.query('SELECT * FROM artist', (err, result) => {
    if (err) {
      res.status(400).send(err.stack);
    }
    else {
      res.status(200).send(result.rows);
    }
  })
})

export default routes;
```

40. Na pasta repositorys crie o arquivo ArtistRepository.js. Nesta pasta ficará as nossas consultas e demais operações no banco de dados.

41. Na pasta controllers crie o arquivo ArtistController.js. Nesta pasta ficará apenas o tratamento do resultado das consultas obtidos do banco.
42. No arquivo ArtistRepository importe o pool e crie a classe ArtistRepository.

```
import pool from '../config/conn';

class ArtistRepository {

}

export default new ArtistRepository();
```

43. Nessa classe faremos nossas operações com a tabela Artist, primeiro método findAll(), para retornar todos os resultados.

```
async findAll() {
  try {
    const result = await pool.query('SELECT * FROM artist');
    return result.rows;
  } catch (err) {
    throw err;
  }
}
```

44. No arquivo ArtistController importe o ArtistRepository e crie a classe ArtistController.

```
import ArtistRepository from "../repositories/ArtistRepository";

class ArtistController {

}

export default new ArtistController();
```

45. No controller o método **index** será responsável por tratar o resultado de findAll().


```

async index(req, res) {
  try {
    const result = await ArtistRepository.findAll();
    res.status(200).send(result);
  } catch (err) {
    res.status(400).send({ "message": err.message });
  }
}

```

Utilizamos o código de status http corretamente e colocamos nossa requisição num async/await no lugar da promise, para melhor legibilidade do código e manter o padrão mais recente do javascript.

46.No arquivo routes.js precisamos importar nosso controller e chamar os métodos corretamente na respectiva rota.

```

import { Router } from 'express';
import ArtistController from '../controllers/ArtistController';

const routes = new Router();

// main
routes.get('/', (req, res) => {
  return res.json({ "info": "API NODE" });
});

// artist
routes.get('/artist', ArtistController.index);

export default routes;

```

47. Teste no navegador e veja se está recebendo o resultado desejado.

48. Faremos agora a consulta para apenas um artista e não todos, será um método GET, usaremos o id para busca.

49. Crie o método findOneById(id), ele receberá um id para realizar a consulta. Ele terá dois tratamentos de erro um para erro do banco de dados e outra para caso não exista o id procurado.

```

async findOneById(id) {
  try {
    const result = await pool.query(`SELECT * FROM artist WHERE idartist = ${id}`);
    if (result.rows.length > 0) {
      return result.rows;
    } else {
      throw new Error("ID NOT FOUND!");
    }
  } catch (err) {
    throw err;
  }
}

```

Podemos fazer a busca por nome também, segue a mesma lógica da busca por id, então deixarei como um desafio para você fazer o findOneByName(name).

50. Vamos no arquivo ArtistController criar o método **show**. Nesse método capturamos o id no parâmetro da requisição e chamamos o método findOneById(id) do repositório e concluímos com o tratamento do resultado como no findAll().

```

async show(req, res) {
  const { id } = req.params;

  try {
    const result = await ArtistRepository.findOneById(id);
    res.status(200).send(result);
  } catch (err) {
    if (err.message !== "ID NOT FOUND!") {
      res.status(400).send({ "message": err.message });
    } else {
      res.status(404).send({ "message": err.message });
    }
  }
}

```

51. Vamos no arquivo routes.js criar a rota: routes.get('/artist/:id', ArtistController.show);.
52. Faça os testes no navegador, por exemplo: localhost:3333/artist/5.
53. Faremos agora o método create(name) no ArtistRepository.

Para adicionar um artista na tabela precisamos apenas do nome, que receberemos do controller ao pegar essa informação no corpo da requisição, o id será gerado automaticamente pelo banco. Insert não tem uma resposta como resultado, vamos personalizar uma mensagem para saber que deu certo a operação.

54. Utilizaremos o método `findOneByName` antes de executar o `create`, para garantir que não façamos registros duplicados.

`findOneByName`:

```
async findOneByName(name) {
  try {
    const result = await pool.query(
      `SELECT * FROM artist WHERE name = '${name}'`
    );
    if (result.rows.length > 0) {
      return result.rows;
    } else {
      return [];
    }
  } catch (err) {
    throw err;
  }
}
```

`create(name)`:

```
async create(name) {
  const exist = await this.findOneByName(name);
  if (exist.length > 0) {
    throw new Error(Constants.DUPLICATE);
  } else {
    try {
      let result = await pool.query(`INSERT INTO artist
        VALUES(nextval('artist_idartist_seq'),'${name}')`);
      if (result) {
        result = Constants.CREATED;
      }
      return result;
    } catch (err) {
      throw err;
    }
  }
}
```

55. Vamos no arquivo `ArtistController.js` tratar a resposta, crie o método `store` e capture o `'name'` do corpo da requisição.

```

async store(req, res) {
  const { name } = req.body;

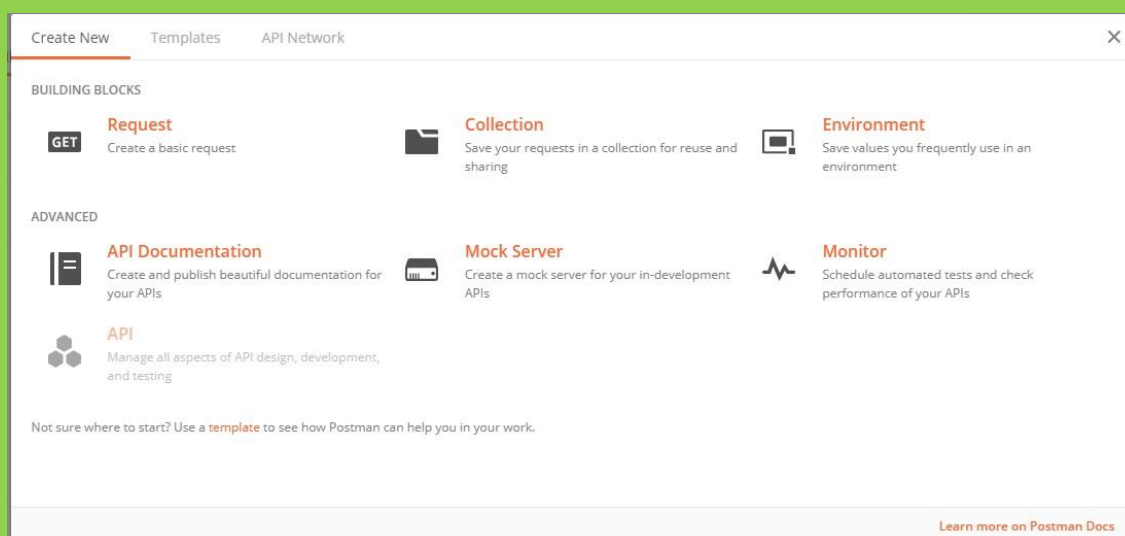
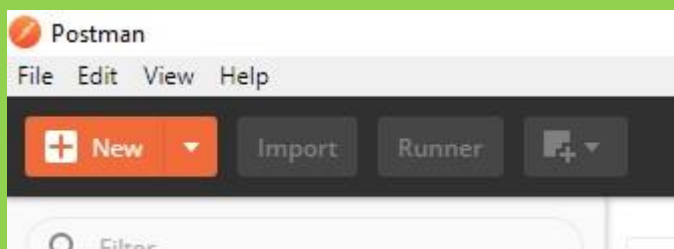
  try {
    const result = await ArtistRepository.create(name);
    res.status(201).send({ "message": result });
  } catch (err) {
    res.status(404).send({ "message": err.message });
  }
}
}

```

56. Configure a rota em routes.js: `routes.post('/artist', ArtistController.store);`.

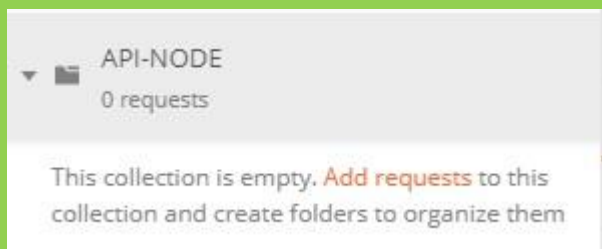
O navegador só nos permite trabalhar com o método GET, agora que vamos testar o POST, precisaremos de outro software, um testador de API, nesse curso utilizaremos o **Postman**.

57. Abra o Postman e clique em new -> collection.

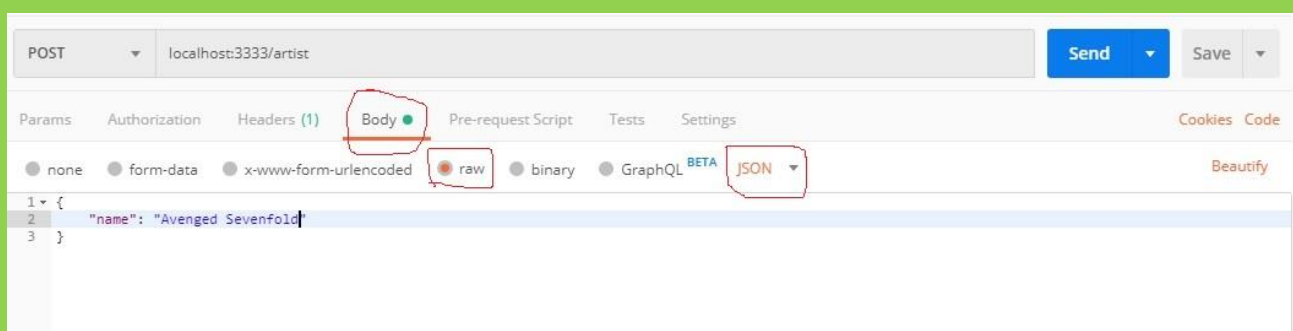


58. Escreva um nome para collection, usaremos API-NODE e clique em create.

59. Clique na pasta e clique em add request.



60. Chame o primeiro request de index, utilizando GET, na url coloque localhost:3333/artist. Ele se comporta igual o navegador e nos traz a resposta da requisição.
61. Crie um request chamado show, para testar use a url: localhost/3333/artist/5.
62. Para testar o store, crie uma request com o método POST, a url é a mesma: localhost:3333/artist.
63. Agora vamos no campo body, selecionar raw e no lugar de text -> json. Digite: {"name": "Avenged Sevenfold"}.
64. Clique em send para efetuar a operação.



65. Vamos em frente para o método UPDATE. Também vamos precisar acessar dados da requisição. No ArtistRepository crie o método findOneAndUpdate(id, name).
66. Vamos utilizar internamente o findOneById(), porque se não for encontrado o registro nem mesmo tentará o update.

findOneAndUpdate(id,name):

```
async findOneAndUpdate(id, name) {
  const exist = await this.findOneById(id);
  if (exist.length > 0) {
    try {
      let result = await pool.query(`UPDATE artist SET name = '
${name}'
WHERE idartist = ${id}`);
      if (result) {
        result = Constants.UPDATED;
      }
      return result;
    } catch (err) {
      throw err;
    }
  }
}
```

```

    } else {
      throw new Error(Constants.ID_NOT_FOUND);
    }
  }
}

```

67. Em ArtistController configure o método **update**.

```

async update(req, res) {
  const { id } = req.params;
  const { name } = req.body;

  try {
    const result = await ArtistRepository.findOneAndUpdate(id, name);
    res.status(200).send({ "message": result });
  } catch (err) {
    res.status(404).send({ "message": err.message });
  }
}

```

68. Configure routes: `routes.put('/artist/:id', ArtistController.update);`.

69. Crie o método PUT no Postman para os testes e utilize o body para atualizar os campos.

70. Por fim, faremos o método `findOneAndDelete(id)` no ArtistRepository e o método **destroy** no ArtistController, para eliminarmos dados. Faremos a mesma lógica que o update, para verificar se existe antes de fazer o delete.

`findOneAndDelete(id)`:

```

async findOneAndDelete(id) {
  const exist = await this.findOneById(id);
  if (exist.length > 0) {
    try {
      let result = await pool.query(`DELETE FROM artist WHERE id = ${id}`);
      if (result) {
        result = Constants.REMOVED;
      }
      return result;
    } catch (err) {
      throw err;
    }
  } else {

```

```
    throw new Error(Constants.ID_NOT_FOUND);
  }
}
```

destroy:

```
async destroy(req, res) {
  const { id } = req.params;

  try {
    const result = await ArtistRepository.findOneAndDelete(id);
    res.status(200).send({ message: result });
  } catch (err) {
    res.status(404).send({ message: err.message });
  }
}
```

71. Faça as configurações necessárias em routes.js e crie a requisição DELETE no Postman e realize os testes.

Código final do ArtistRepository.js:

```
import pool from "../config/conn";
import Constants from "../util/Constants";

class ArtistRepository {
  async findAll() {
    try {
      const result = await pool.query("SELECT * FROM artist");
      return result.rows;
    } catch (err) {
      throw err;
    }
  }

  async findOneById(id) {
    try {
      const result = await pool.query(
        `SELECT * FROM artist WHERE idartist = ${id}`
      );
      if (result.rows.length > 0) {
        return result.rows;
      } else {

```

```

        throw new Error(Constants.ID_NOT_FOUND);
    }
} catch (err) {
    throw err;
}
}

async findOneByName(name) {
    try {
        const result = await pool.query(
            `SELECT * FROM artist WHERE name = '${name}'`
        );
        if (result.rows.length > 0) {
            return result.rows;
        } else {
            return [];
        }
    } catch (err) {
        throw err;
    }
}

async create(name) {
    const exist = await this.findOneByName(name);
    if (exist.length > 0) {
        throw new Error(Constants.DUPLICATE);
    } else {
        try {
            let result = await pool.query(`INSERT INTO artist
            VALUES(nextval('artist_idartist_seq'),'${name}')`);
            if (result) {
                result = Constants.CREATED;
            }
            return result;
        } catch (err) {
            throw err;
        }
    }
}

async findOneAndUpdate(id, name) {
    const exist = await this.findOneById(id);

```



```
    if (exist.length > 0) {
      try {
        let result = await pool.query(`UPDATE artist SET name = '
${name}`
        WHERE idartist = ${id}`);
        if (result) {
          result = Constants.UPDATED;
        }
        return result;
      } catch (err) {
        throw err;
      }
    } else {
      throw new Error(Constants.ID_NOT_FOUND);
    }
  }

  async findOneAndDelete(id) {
    const exist = await this.findOneById(id);
    if (exist.length > 0) {
      try {
        let result = await pool.query(`DELETE FROM artist WHERE i
dartist = ${id}`);
        if (result) {
          result = Constants.REMOVED;
        }
        return result;
      } catch (err) {
        throw err;
      }
    } else {
      throw new Error(Constants.ID_NOT_FOUND);
    }
  }
}

export default new ArtistRepository();
```

Código final do ArtistController.js:

```
import ArtistRepository from "../repositorys/ArtistRepository";

class ArtistController {
  async index(req, res) {

    try {
      const result = await ArtistRepository.findAll();
      res.status(200).send(result);

    } catch (err) {
      res.status(400).send({ "message": err.message });
    }

  }

  async show(req, res) {
    const { id } = req.params;

    try {
      const result = await ArtistRepository.findOneById(id);
      res.status(200).send(result);

    } catch (err) {
      if (err.message !== "ID NOT FOUND!") {
        res.status(400).send({ "message": err.message });
      } else {
        res.status(404).send({ "message": err.message });
      }

    }

  }

  async store(req, res) {
    const { name } = req.body;

    try {
      const result = await ArtistRepository.create(name);
      res.status(201).send({ "message": result });

    } catch (err) {
```

```

        res.status(404).send({ "message": err.message });
    }
}

async update(req, res) {
    const { id } = req.params;
    const { name } = req.body;

    try {
        const result = await ArtistRepository.findOneAndUpdate(id,
name);
        res.status(200).send({ "message": result });

    } catch (err) {
        res.status(404).send({ "message": err.message });
    }
}

async destroy(req, res) {
    const { id } = req.params;

    try {
        const result = await ArtistRepository.findOneAndDelete(id);
        res.status(200).send({ message: result });

    } catch (err) {
        res.status(404).send({ "message": err.message });
    }
}
}

export default new ArtistController();

```

routes.js:

```

import { Router } from 'express';
import ArtistController from '../controllers/ArtistController';

const routes = new Router();

// main

```

```
routes.get('/', (req, res) => {
  return res.json({ "info": "API NODE" });
});

// artist
routes.get('/artist', ArtistController.index);
routes.get('/artist/:id', ArtistController.show);
routes.post('/artist/', ArtistController.store);
routes.put('/artist/:id', ArtistController.update);
routes.delete('/artist/:id', ArtistController.destroy);

export default routes;
```

Seção 06 – Utilizando constantes

Para desenvolver a interação com as demais entidades dos bancos vamos repetir alguns resultados, para isso vamos criar algumas constantes para evitar a repetição.

72. Na pasta util crie o arquivo Constants.js e reproduza o código abaixo:

```
export default class Constants {
  static ID_NOT_FOUND = "ID NOT FOUND!";
  static DUPLICATE = "You cant duplicate registries!";
  static CREATED = "Created with success!";
  static UPDATED = "Updated with success!";
  static REMOVED = "Removed with success!";
}
```

73. Importe esse arquivo em ArtistRepository.js e onde tem as frases acima substitua pelas constantes. Por exemplo, Constants.CREATED, no caso de "Created with success!"

VAMOS AGORA PARA O DESAFIO:

Temos mais três entidades para construir os controllers e repositorys: album, gender e music. Crie novas classes e repositórios para cada entidade, as rotas devem ser no mesmo arquivo só fazer o 'import' e as associações corretas.

Os três primeiros que concluírem o projeto e estiver funcionando corretamente ganharam um chocolate.

Podem pedir ajudar e tirar dúvidas!