

Minicurso

RESTful API Node.js +

Express + PostgreSQL

Guilherme Gomes

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

Danilo Marques

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

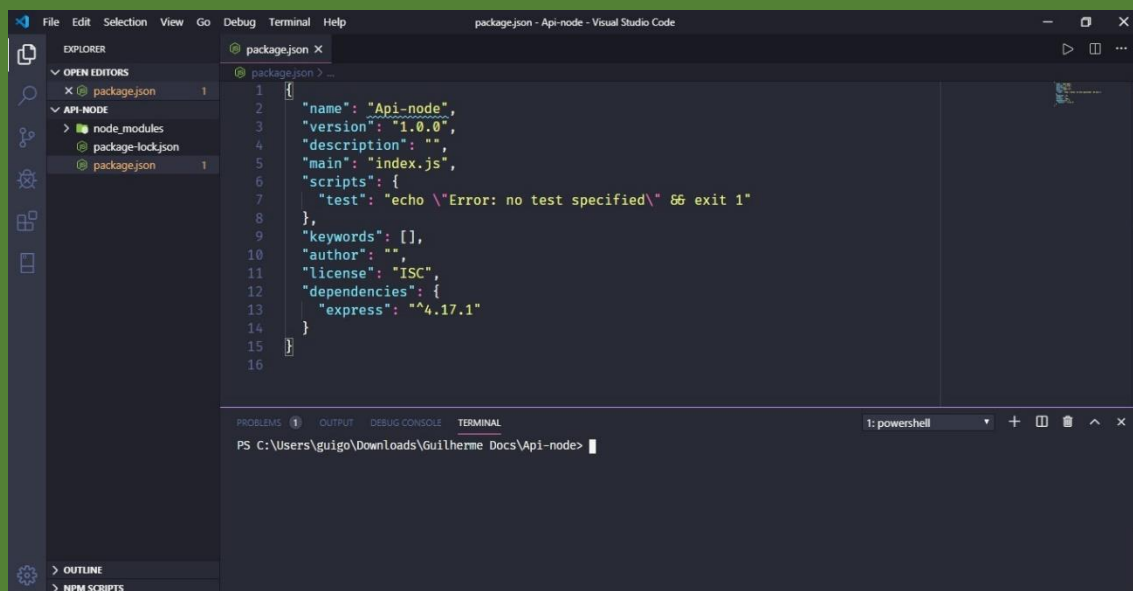
Seção 01 – Criação do banco de dados

1. Abra o psql (shell script do postgres) e tecle enter até pedir o password.
2. Digite a senha + Enter.
3. Digite \l (contra-barras + L) + Enter. Será exibido a lista dos bancos já criados no postgres no momento.
4. Acesse o arquivo musicfy.sql na pasta Material Complementar.
5. Copie e cole o script no psql para criar o banco e adicionar alguns dados.
6. Por enquanto minimize e agora crie uma pasta com o nome que você quer usar no projeto, aqui usaremos o nome “APINODE”. 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. Ainda no VS Code vá na aba Terminal -> New Terminal. No rodapé ficará disponível um terminal que utilizaremos recorrentemente.
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. Novamente no terminal digite: npm install express.

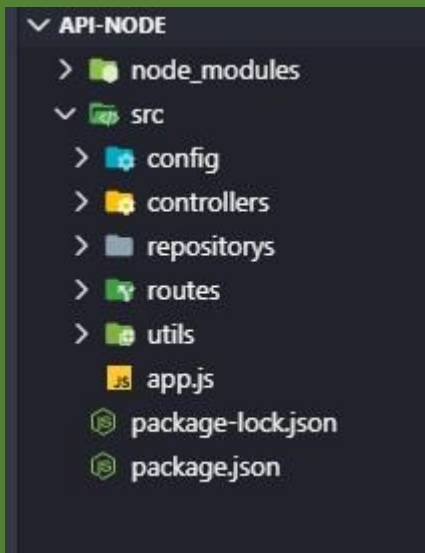
Seu ambiente deve estar com essa aparência:



O express baixou essa pasta “node_modules”, nela já está pronto várias funções que utilizaremos no projeto e sempre que adicionarmos mais alguma dependência, é nela que ficará armazenado.

12. Vamos agora criar algumas pastas para deixar o código organizado.
13. Crie a pasta src onde ficará todo os nosso códigos.
14. Crie dentro da pastar 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. Abra o arquivo app.js para começarmos.
17. Primeiro importar o express para o código: `const express = require("express");`.
18. Instâncie o express em uma variável: `const server = express();`.
19. Para sempre trabalharmos com json na resposta ative essa função da seguinte maneira: `server.use(express.json());`.
20. Para testar vamos criar uma requisição GET:
`server.get("/", (req, res) => {
 return res.json({ "info": "API NODE"});
});`.
21. 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);
```

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

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

23. Salve o arquivo e no terminal digite npm start e agora vá em qualquer navegador e coloque na url: localhost:3333.
24. Encerre o servidor, tecla ctrl+c no terminal.
25. 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 alterações que fizermos. O -D no final do comando significa que são dependências de desenvolvimento, não serão utilizadas em produção.

26. Volte no arquivo package.json e altere novamente script, substitua node por nodemon.
27. 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.

28. Altere o código em app.js, agora podemos utilizar o 'import', mais recente: import express from "express";.
29. 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();

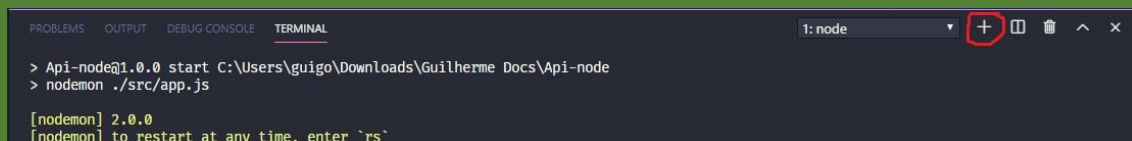
server.use(express.json());

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

server.listen(3333);
```

Seção 04 – Conexão com o banco de dados

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

A screenshot of a VS Code terminal window. The terminal title bar shows '1: node' and a red circle highlights the '+' button for opening a new terminal. The terminal content shows the command 'nodemon ./src/app.js' being executed, with output from 'nodemon 2.0.0' indicating it will restart at any time by entering 'rs'.

- 31. Digite `npm install pg`, essa dependência nos permitirá trabalhar com o postgres no express.
- 32. Na pasta config crie o arquivo `conn.js`.
- 33. No arquivo `conn.js` importe apenas o Pool do pg.
- 34. 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 utiliza-lo para realizar as nossas consultas SQL.

Seção 05 – 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.

35. Na pasta utils 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!";  
}
```

Mais na frente, entenderemos melhor o porquê de criar essas constantes.

Seção 06 – Construindo rotas, controllers e repositorys

36. Crie um arquivo na pasta routes com o nome routes.js.

37. 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.

38. 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;

```

39. 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 './routes/routes';

const server = express();

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

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

server.listen(3333);

```

40. 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.

41. 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 './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;
```

42. Na pasta repositorys crie o arquivo ArtistRepository.js. Nesta pasta ficará as nossas consultas e demais operações no banco de dados.
43. Na pasta controllers crie o arquivo ArtistController.js. Nesta pasta ficará apenas o tratamento do resultado das consultas obtidos do banco.
44. No arquivo ArtistRepository importe o pool e crie a classe ArtistRepository.


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

class ArtistRepository {

}

export default new ArtistRepository();
```

45. Nessa classe faremos nossas operações com a tabela Artist, dentro da class crie o primeiro método findAll() para retornar todos os resultados da tabale artist.

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

46. No arquivo ArtistController import o ArtistRepository e crie a classe ArtistController.

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

class ArtistController {

}

export default new ArtistController();
```

47. Detrno da classe ArtistController crie o método **index**, ele 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 });
  }
}
```

```
}  
}
```

Veja que 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.

48. 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;
```

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

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

51. Na classe `ArtistRepository`, 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(Constants.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)`, porém mais na frente esse código também estará disponível no material.

52. No `ArtistController` crie método **show**. Nesse método capturamos o id na 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 });
    }
  }
}
```

53. Vamos no arquivo `routes.js` criar a rota: `routes.get('/artist/:id', ArtistController.show);`.
54. Faça os teste no navegador, por exemplo: `localhost:3333/artist/5`.
55. Faremos agora o método `create(name)` no `ArtistRepository`.
56. Importe as constantes em `ArtistRepository`:

```
import Constants from "../util/Constants";
```

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 no banco de dados, não tem uma resposta como resultado, vamos personalizar uma mensagem para saber que deu certo a operação.

57. 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;
    }
  }
}

```

58. 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;

```

```

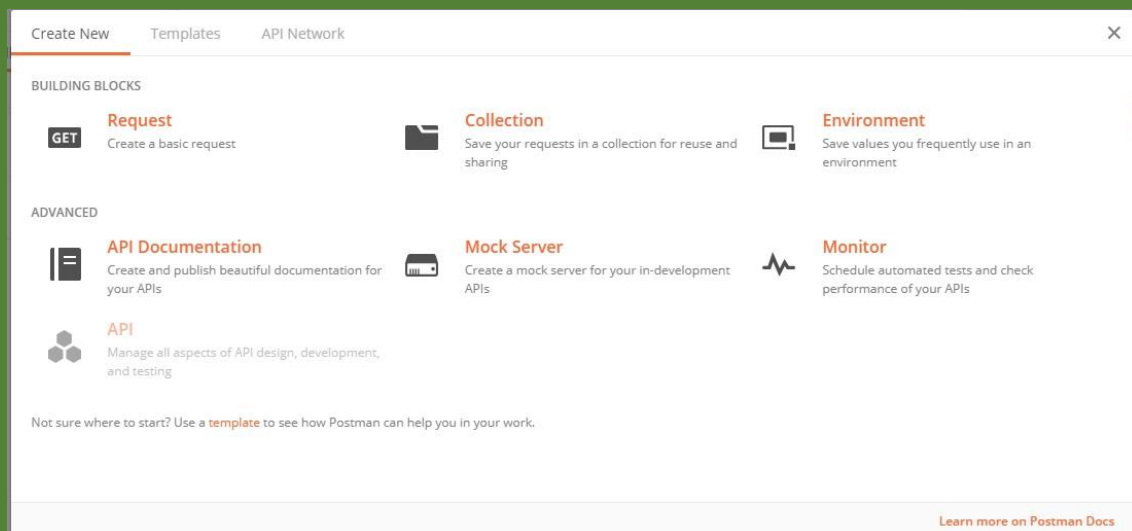
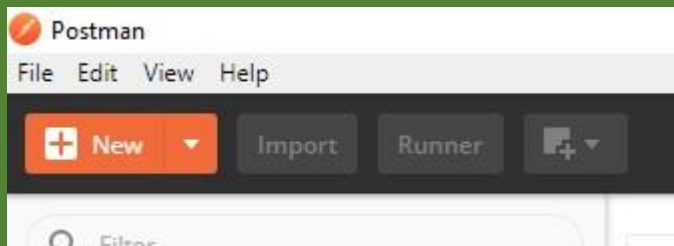
try {
  const result = await ArtistRepository.create(name);
  res.status(201).send({ message: result });
} catch (err) {
  res.status(404).send({ message: err.message });
}
}

```

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

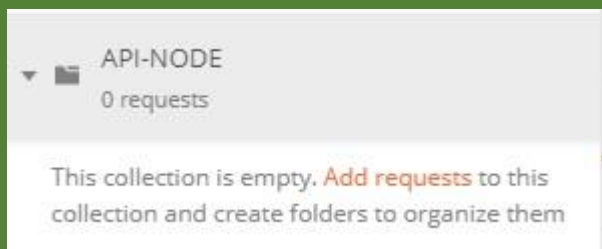
Não é possível trabalhar com POST no navegador, precisaremos de outro software, um testador de API, nesse curso utilizaremos o **Postman**.

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

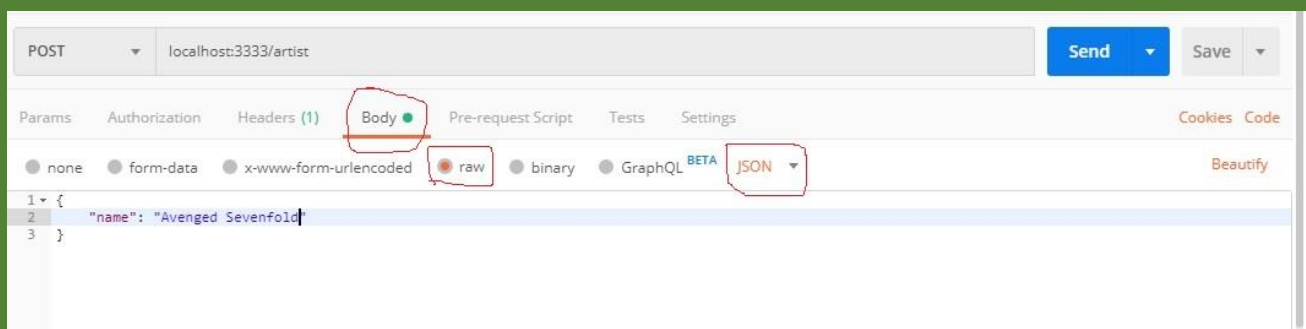


61. Escreva um nome para collection, usaremos API-NODE e clique em create.

62. Clique na pasta e clique em add request.



63. 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.
64. Crie um request chamado show, para testar use a url: localhost:3333/artist/5.
65. Para testar o store, crie uma request com o método POST, a url é a mesma do index: localhost:3333/artist.
66. Agora vamos no campo body, selecionar raw e no lugar de text -> json. Digite: {"name": "Avenge Sevenfold"}.
67. Clique em send para efetuar a operação, veja se deu tudo certo.



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

findOneAndUpdate(id,name):

```
async findOneAndUpdate(id, name) {
  const exist = await this.findById(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);
    }
  }
}

```

70. 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 });
  }
}

```

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

72. Crie o método PUT no Postman para os testes e utilize o body para atualizar os campos. Coloque o id na url: `localhost:3333/artist/5`, no body: `{ "name": "qualquercoisa" }` e teste.

73. 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 i
dartist = ${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 });
    }
}

```

74. 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;
```

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.

Basta copiar e colar o que foi feito em ArtistRepository e ArtistiController, o que vai mudar são apenas as queries SQL as quais deixarei na próxima e o projeto completo você poderá consular no github.

<https://github.com/guigomes94/minicurso-node-postgres>

Queries para Gender:

```
findAll(): "SELECT * FROM Gender"

findOneById(id): `SELECT * FROM Gender WHERE idGender = ${id}`

findOneByName(name): `SELECT * FROM gender WHERE name = '${name}'`

create(name): `INSERT INTO Gender
VALUES(nextval('Gender_idGender_seq'), '${name}')`

findOneAndUpdate(id, name): `UPDATE gender SET name = '${name}'
WHERE idgender = ${id}`

findOneAndDelete(id): `DELETE FROM gender WHERE idgender = ${id}`
```

Queries para Album:

```
findAll():`SELECT al.idalbum "id", a.name "artist",
g.name "gender", al.name , al.year

FROM artist a, gender g, album al
WHERE a.idartist = al.idartist AND g.idgender = al.idgender`

findOneById(id): `SELECT al.idalbum "id", a.name "artist",
g.name "gender", al.name , al.year

FROM artist a, gender g, album al
WHERE a.idartist = al.idartist AND g.idgender = al.idgender
AND idalbum = ${id}`

findOneByName(name):
`SELECT al.idalbum "id", a.name "artist", g.name "gender", al.name, al.year

FROM artist a, gender g, album al
WHERE a.idartist = al.idartist AND g.idgender = al.idgender AND a
l.name = '${name}'`

create(idart, idgd, name, year): `
`INSERT INTO Album VALUES(nextval('Album_idAlbum_seq'),
${idart}, ${idgd}, '${name}', ${year})`

findOneAndUpdate(id, idart, idgd, name, year):
`UPDATE Album SET idartist = ${idart}, idgender = ${idgd},
name = '${name}', year = ${year} WHERE idAlbum = ${id}`
```

```
findOneAndDelete(id): `DELETE FROM Album WHERE idAlbum = ${id}`
```

Queries para Music:

```
findAll():`SELECT m.idmusic, al.name "album", m.track, m.name, m.time  
FROM music m, album al WHERE m.idalbum = al.idalbum;`
```

```
findOneById(id): `SELECT m.idmusic, al.name "album", m.track,  
m.name, m.time
```

```
FROM music m, album al
```

```
WHERE m.idalbum = al.idalbum AND idMusic = ${id}`
```

```
findOneByName(name): `SELECT m.idmusic, al.name "album", m.track,  
m.name, m.time
```

```
FROM music m, album al
```

```
WHERE m.idalbum = al.idalbum AND m.name = '${name}'`
```

```
create(idalb, track, name, time):
```

```
`INSERT INTO Music VALUES(nextval('Music_idMusic_seq')  
,${idalb}, ${track},'${name}','${time}')
```

```
findOneAndUpdate(id, idalb, track, name, time):
```

```
`UPDATE Music SET idalbum = ${idalb}, track = ${track},  
name = '${name}', time = '${time}' WHERE idMusic = ${id}`
```

```
findOneAndDelete(id): `DELETE FROM music WHERE idmusic = ${id}`
```