EXERCÍCIO 1

a. Explique como é a resposta que temos quando usamos o `raw`.

RESPOSTA: O raw retorna um Array com objetos, mas a única parte realmente relevante é o objeto na posição [0] do Array.

---

b. Faça uma função que busque um ator pelo nome;

RESPOSTA:

try{

const getActorByName = async (name: string): Promise<any> => {

const result = await connection.raw(`SELECT * FROM Actor WHERE name = '${name}'`)

console.log(return[0][0];

}

getActorByName(“Juliana Paes”);

}catch (error){

console.error(error.message);

};

---

c. Faça uma função que receba um `gender` retorne a quantidade de itens na tabela Actor com esse `gender`. Para atrizes, `female` e para atores `male`.

RESPOSTA:

try{

const getActorsByGender = async (gender: string): Promise<any> => {

const result = await connection.raw(`SELECT COUNT(*) as count FROM Actor WHERE gender = "${gender}"`);

const count = result[0].count;

console.log(count);

};

getActorsByGender(“female”);

}catch (error){

console.error(error.message);

}

Exercício 2

a. Uma função que receba um salário e um id e realiza a atualização do salário do ator em questão

RESPOSTA:

```
try{
const modifyActor = async (id: string, salary: number): Promise<any> => {
await connection("Actor")
.update({
salary: salary,
})
.where("id", id);
};
modifyActor("003", 576000)
}catch (error){
console.error(error.message);
}
```

---

b. Uma função que receba um id e delete um ator da tabela

RESPOSTA: (Não funcionou)

```
try{
const deleteActor = async (id: string): Promise<any> => {
await connection("Actor")
.delete()
.where("id", id);
};
deleteActor("002")
}catch (error){
console.error(error.message);
}
```

---

c. Uma função que receba um `gender` e devolva a média dos salários de atrizes ou atores desse `gender`

RESPOSTA:

```
try{
const avgSalary = async (gender: string): Promise<any> => {
const result = await connection("Actor")
.avg("salary as average")
.where({ gender });
console.log(result[0].average);
};
avgSalary("female");
}catch (error){
```

console.error(error.message);

}

EXERCÍCIO 3

a. Por que o id está sendo lido assim: `req.params.id`?

RESPSOTA: Por que o ID vai ser requisitado como path param.

---

b. O que as últimas linhas do try (`res.status(200).send(actor)`) e do catch (`res.status(400).send({...}`) fazem? Teste o código se precisar.

RESPOSTA: `res.status(200).send(actor)` serve para enviar a resposta de requisição em caso de sucesso e com status 200.

`res.status(400).send({...}` serve para enviar um código de erro em caso de falha na requisição.

---

c. Crie um endpoint agora com as seguintes especificações:

- Deve ser um GET (`/actor`)
- Receber o gênero como um *query param* (`/actor?gender=`)
- Devolver a quantidade de atores/atrizes desse gênero

  RESPOSTA:

  app.get("/actor", async (req: Request, res: Response) => {

  try {

  const gender = req.body.gender

  const result = await connection.raw(`SELECT COUNT * FROM Actor WHERE gender = "${gender}"`);

  res.status(200).send({

  Actors: result[0] ,

  });

  } catch (error) {

  res.status(400).send({

  message: error.message,

  });

  }

  });

EXERCÍCIO 4

Crie um endpoint para cada uma das especificações abaixo:

a.

- Deve ser um POST (`/actor`)

- Receber o salário e o id pelo body

- Simplesmente atualizar o salário do ator com id em questão

RESPOSTA:

```
app.post("/actor", async (req: Request, res: Response) => {
try {
const id = req.body.id
const newSalary = req.body.salary
await connection.raw(`UPDATE Actor SET salary = ${newSalary} WHERE id = ${id}`);
res.status(200).send({
message: “Success”,
});
} catch (error) {
res.status(400).send({
message: error.message,
});
}
});
```

---

b.

- Deve ser um DELETE (`/actor/:id`)

- Receber id do ator como *path param*

- Simplesmente deletar o ator da tabela

RESPSOTA:

```
app.delete("/actor", async (req: Request, res: Response)=>{
try{
const id = req.body.id;
await connection.raw(`DELETE FROM Actor WHERE id = "${id}"`);
res.status(200).send({message: “Ator apagado com sucesso”});
}catch(err){
res.status(400).send({error: err.message});
}
})
```

EXERCÍCIO 5

Especificações do Endpoint:

- Deve ser um POST (`/movie`)
- Receber todas as informações pelo body
- Criar o filme na tabela

RESPOSTA:

```
app.post("/movie", async (req: Request, res: Response) => {
try {
const id = req.body.id
const title = req.body.tytle
const synopsis = req.body.synopsis
const release_Date = req.body.release_Date
const rating = req.body.rating
const playing_limit_date = req.body.playing_limit_date
await connection.raw( INSERT INTO Movie VALUES (
"${id}","${title}","${synopsis}","${release_Date}","${rating}","${playing_limit_date}"
) );
res.status(200).send({
message: “Filme cadastrado com sucesso”,
});
} catch (error) {
res.status(400).send({
message: error.message,
});
}
});
```

EXERCÍCIO 6

Especificações do Endpoint:

- Deve ser um GET (`/movie/all`)
- Não recebe nada
- Retorna todos os filmes. Ele deve retornar, no máximo, uma lista com 15 itens

RESPOSTA:

```
app.get("/movie/all", async (req: Request, res: Response) => {
```

```
try {
const result = await connection.raw( SELECT * FROM Movie LIMIT 15 );
res.status(200).send({
movies: result[0] ,
});
} catch (error) {
res.status(400).send({
message: error.message,
});
}
});
```

EXERCÍCIO 7

Especificações do Endpoint:

- Deve ser um GET (`/movie/search`)
- Deve receber o termo de busca como uma query string (`/movie/search?query=`)
- Faz a busca entre todos os filmes que tenham o termo de busca no nome ou na sinopse. Além disso, a lista deve vir ordenada pela data de lançamento

RESPOSTA:

```
app.get("/movie/search", async (req: Request, res: Response) => {
try {
const data = req.body.search_term
const result = await connection.raw( SELECT * FROM Movie WHERE synopsis = "%${data}%" OR title = "%${data}%" ORDER BY release_Date ASC );
res.status(200).send({
movies: result[0] ,
});
} catch (error) {
res.status(400).send({
message: error.message,
});
}
});
```