Knex.js

Labenu_

O que vamos ver hoje? 👀

- Como interagir com o MySQL usando Typescript
- Como proteger dados sensíveis no nosso código

Knex

Labenu_

Knex

- ***Knex*** é uma biblioteca de Javascript que permite fazer conexões com vários bancos SQL

- Para isso, precisa que definamos um *client* de SQL para funcionar, ou seja:
  - MySQL
  - PostgreSQL
  - MariaDB
  - etc

Knex

- Assim, ao iniciar um projeto com Typescript em NodeJS, precisamos instalar o client **mysql** juntamente com o **knex** e sua versão tipada

```
npm install knex @types/knex mysql

yarn add knex @types/knex mysql
```

Knex

- Ao fazer isso, podemos estabelecer uma conexão com o Mysql no nosso código, importando o knex e fazendo as seguintes configurações:

```
import knex from 'knex'

const connection = knex({  // Estabelece conexão com o
banco
   client: "mysql",
   connection: {
       host: "35.226.146.116",
       port: 3306,
       user: "aluno",
       password: "ahninanab",
       database: "turma-aluno",
       multipleStatements: true
   }
})

export default connection
/
```

Knex: raw 🥩

- Com essa configuração, podemos **acessar** o banco de dados de alguma formas.

- Uma delas é o método ***raw***, que permite que enviemos uma **query** para o banco usando a linguagem **SQL** diretamente

- Normalmente, usamos **template strings** para montar as queries do *raw*. Isso permite **quebrar linhas** e acessar **variáveis** do código.

- Para exemplificar, vamos escrever endpoints para popular e consultar nossa tabela de atores **através de código**

Knex: raw 🥩

- As funções farão uma comunicação **externa** com o banco
- Isso indica que elas devem ser **assíncronas**
- Toda função assíncrona devolve uma **Promise**
- Usamos o **await** para esperar terminar a comunicação com o banco

```
app.post("/actor", async (req, res): Promise<void> => {
  try {
    await connection.raw(`
      INSERT INTO Actor
        (id, name, salary, birth_date, gender)
      VALUES (
        ${Date.now().toString()},
        "${req.body.name}",
        ${req.body.salary},
        "${req.body.birthDate}",
        "${req.body.gender}"
      );
    `)

    res.status(201).send("Success!")
  } catch (error) {
    console.log(error.message);
    res.status(500).send("An unexpected error occurred")
  }
})
```

Knex: raw 🥩

- Quando usamos as template *strings*, devemos lembrar de usar a sintaxe **${}** para passar os valores das variáveis
- As **strings** devem sempre ser passadas com **aspas** (duplas ou simples)
- Os **números** devem ser passados dentro das template strings **sem aspas**

```
app.post("/actor", async (req, res): Promise<void> => {
  try {
     await connection.raw(`
        INSERT INTO Actor
           (id, name, salary, birth_date, gender)
        VALUES (
           ${Date.now().toString()},
           "${req.body.name}",
           ${req.body.salary},
           "${req.body.birthDate}",
           "${req.body.gender}"
        );
     `)

     res.status(201).send("Success!")
  } catch (error) {
     console.log(error.message);
     res.status(500).send("An unexpected error occurred")
  }
})
```

Knex: raw 🥩

- Os outros comandos que modificam a tabela funcionam da mesma forma: *UPDATE*, *DELETE*, *ALTER*. etc

- Para as queries de leitura, devemos nos atentar a como a resposta é devolvida

```
app.get('/actor', async(req, res) => {
  try {

    const result = await connection.raw(`
       SELECT * FROM Actor
    `)

    res.send(result)
  } catch (error) {
    console.log(error.message);
    res.status(500).send("An unexpected error occurred")
  }
})
```

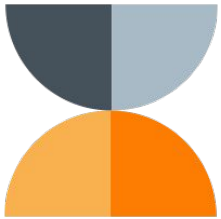
Knex: raw 🥩

- O resultado no console é algo parecido com isso:

```
[
   [
      {"name": "Tony Ramos"},
      {"name": "Camila Pitanga"},
      {"name": "Antônio Fagundes"}
      ...
   ],
   [
      {
         "catalog": "def",
         "db": "teachers-mateus-gesualdo",
         "table": "Actor",
         ...
      },
      ...
   ]
]
```

Knex: raw 🥩

- Ele devolve pra gente o **resultado da query** e **outras informações**
- Essa é a forma com o que o **MySQL** devolve as queries naturalmente
- O que precisamos fazer é simples: **pegar somente as informações que desejamos**
- Os dados que queremos estão na **primeira posição do array**
- Então, para acessá-los é só pegar a primeira posição da resposta da query



Pausa para relaxar 😴

10 min

- O knex é uma biblioteca que permite fazer conexões com **bancos SQL**
- Como queremos conectar com um MySQL, devemos baixar o ***client* específico** dele para node
- O jeito mais direto de fazer queries é usando o método **raw** que devolve diretamente a resposta do banco

Knex: Query Builder

Labenu_

Knex: Query Builder 👷‍♀️

- O ***Query Builder*** é uma funcionalidade do Knex que tende a **facilitar a criação das queries**

- Além da escrita, ele facilita na hora de tratar os dados

- Existem várias sintaxes relacionadas ao *Query Builder*, por isso é sempre bom confirmar na documentação se o que você está fazendo está certo

- Para exemplificar, vamos criar um endpoint para atualizar os dados de um ator

Knex: Query Builder 👷‍♀️

```
app.put('/actor/:id', async (req, res) => {
  try {

    await connection("Actor")
        .update({
          name: req.body.name,
          salary: req.body.salary,
          birth_date: req.body.birthDate,
          gender: req.body.gender
        })
        .where({ id: req.params.id })

    res.send("Success!")
  } catch (error) {
    console.log(error.message);
    res.status(500).send("An unexpected error occurred")
  }
})
```

```
// busca todos os atores
await connection("Actor")
```

```
// deleta ator por id
await connection("Actor")
   .delete()
   .where({ id: req.params.id })
```

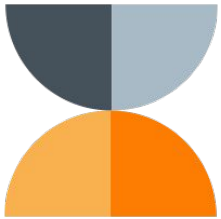
Exercício

1. Utilizando endpoint, faça inserção de um novo usuário na tabela

2. Faça a busca de todos os usuários: nome dos usuários e produtos comprados. Ordene pela data

Pausa para relaxar 😴

5 min



- O knex é um **query builder**, dispensando, em muitos casos, o uso de SQL no backend
- As consultas retornam apenas o que foi pedido

dotenv

Labenu_

dotenv 

- Existe uma lib que nos permite fazer isso de uma forma muito simples no nosso projeto: ***dotenv***

```
npm install dotenv

yarn add dotenv
```

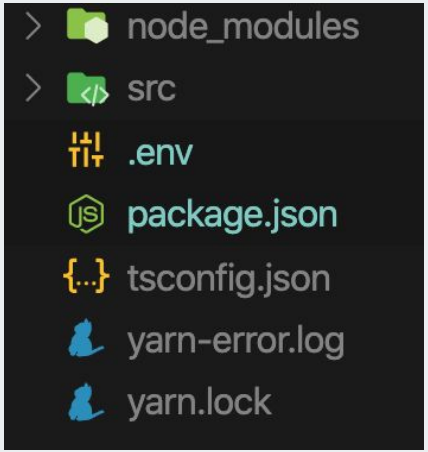
dotenv 

- Quando vamos usar algumas **constantes** que permitem **acessar** algum serviço importante para o nosso projeto, dizemos que elas são "**variáveis de ambiente**"

- Normalmente, elas são informações **bem sensíveis** que queremos proteger ao máximo

dotenv 

- Após a instalação, criaremos um arquivo chamado **.env**.
- Colocamos ele na altura do package.json



dotenv 

- Dentro desse arquivo, definiremos **uma chave** ("nome") para cada uma das constantes

- Os valores podem ser qualquer tipo, que será retornado sempre uma string;

```
DB_HOST = 35.226.146.116
DB_USER = aluno
DB_PASSWORD = ahninanab
DB_SCHEMA = turma-aluno
```

dotenv 

- Finalmente, importaremos a lib para o código e executaremos o comando **dotenv.config()**

- A partir de então, as variáveis de ambiente poderão ser acessadas por **process.env.NOME_DA_VARIAVEL**

dotenv 

```
import knex from 'knex'
import dotenv from 'dotenv'

dotenv.config()

const connection = knex({  // Estabelece conexão com o banco
  client: "mysql",
  connection: {
      host: process.env.DB_HOST,
      port: 3306,
      user: process.env.DB_USER,
      password: process.env.DB_PASSWORD,
      database: process.env.DB_SCHEMA,
      multipleStatements: true
  }
})

export default connection
```

Dúvidas? 🧐

Labenu_

Resumo

Labenu_

Resumo 📓

- O knex é uma biblioteca que permite fazer conexões com **bancos SQL**

- Como queremos conectar com um MySQL, devemos baixar o ***client* específico** dele para node

- O jeito mais direto de fazer queries é usando o método **raw** que devolve diretamente a resposta do banco

Resumo 📓

- Um jeito mais simples de fazer queries é usando as ***Queries Builders***
- Elas retornam a **resposta já pré-computada**
- As informações sensíveis do banco devem ficar salvas num arquivo **.env**
- Usamos a lib *dotenv* para **configurar** as variáveis que estão nesse arquivo

Obrigado!