



Revisão da Semana - Banco de dados

💡 Esse material tem como **objetivo** te dar uma orientação para realizar uma breve revisão sobre como utilizar as novas features de back aprendidas no último bloco

Materiais Complementares

- ▶ PT
- ▶ EN

Materiais de Revisão ☺

▼ Banco de dados e introdução a SQL

✔ Banco de dados e introdução a SQL

Relembando

Até agora, vimos o armazenamento de dados em três diferentes formatos: no LocalStorage do navegador, na própria memória do processo e em APIs já prontas que contam com seus bancos de dados.

Porém, nesses tipos de armazenamento adicionar/ler/editar/deletar/encontrar/salvar dados é trabalhoso, além de ser pouco performático porque só podemos ler os dados todos de uma vez.

Vamos abordar no curso bancos relacionais, por terem uma linguagem que funciona em diferentes softwares, além de ter uma comunidade e suporte maiores.

Estrutura de um banco de dados

Tabelas - Bancos **relacionais** são organizados em **tabelas** que se relacionam por meio de chaves (primárias e estrangeiras)

- ▶ Exemplo

Chave Primária (primary key) - Identificador único para cada linha de uma tabela, evitando ambiguidades (exemplo: caso duas usuárias se chamem "Paula", podemos diferenciá-las por suas PK's)

Colunas - Além do nome, cada coluna possui um tipo de valor e, opcionalmente, outras restrições. Na tabela acima, por exemplo, a coluna *idade* poderia ser obrigatória ou possuir um valor padrão.

Acesso ao banco de dados

Para fins didáticos, oferecemos acesso no nosso banco de dados MySQL. Utilizem o acesso que passamos para vocês pelo slack.

Workbench

O Workbench é o client side que possibilita acessarmos o banco de dados que está rodando em algum server side, ou seja, um servidor remoto.

- ▼ 🌐 client side x server side

Client Side - São linguagens que apenas o seu **NAVEGADOR** vai entender. Quem vai processar essa linguagem não é o servidor, mas o seu browser (Chrome, IE, Firefox, etc...). Ou seja, aplicações que rodam no computador do usuário sem necessidade de processamento de seu servidor (ou host) para efetuar determinada tarefa

Server Side - As linguagens server-side são linguagens que o SERVIDOR (lado do servidor) entende, ou seja, aplicações que rodam no servidor. Isso quer dizer que você vai escrever um código onde o servidor vai processá-lo e então vai mandar para o seu navegador a resposta.

▼ 🔒 configurando o workbench

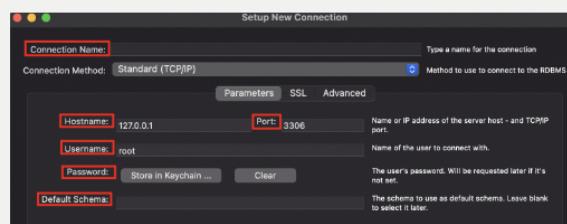
1. Primeiro passo verifique se, no menu lateral esquerdo, você está na aba correta como mostra o exemplo abaixo. Logo, clicando no botão + ao lado de MySQL Connections, a janela de adicionar conexão será aberta.

► Exemplo

2. Agora, preencha os valores que enviamos para vocês no slack nos seguintes campos:

- Connection Name, você escolhe um nome de sua preferência;
- Hostname, preencha com o valor `DB_HOST`;
- Username preecha com o valor `DB_USER`;
- Password, clique no botão Store in Keychain, vai abrir uma janela, preencha com `DB_PASSWORD` e salve;
- Default Schema, preencha com `DB_SCHEMA`;
- Clique em `OK`

▼ Exemplo



3. Pronto! Se os dados estão corretos, vai aparecer o acesso ao banco de dados na sua aba MySQL Connections.

► Exemplo

▼ Aprofundamento SQL

✓ Aprofundamento SQL

SQL é uma linguagem não procedural, com uma sintaxe própria, bem definida e documentada.

▼ O que é linguagem não procedural?

Linguagens procedurais detalham os procedimentos (JS, Python), enquanto linguagens não procedurais somente descrevem o que deve ser feito.

Sintaxe

A sintaxe é constituída de **keywords**, **types** e **values**.

Keywords

Todas operações **começam com uma keyword** descrevendo o que ela faz. Por exemplo: SELECT, INSERT, UPDATE, ALTER...

E devem terminar com ponto e vírgula ;

Types

- `INT / BIGINT / TINYINT` - números inteiros;
- `DOUBLE / FLOAT` - números não inteiros;
- `VARCHAR(n)` - strings de no máximo n caracteres;
- `DATE` - representa uma data (YYYY-MM-DD);
- `DATETIME / TIMESTAMP` - representa uma data com tempo (YYYY-MM-DD)

Constraints

São restrições, que servem para definirmos um comportamento ou uma regra para um determinado campo. As mais comuns são:

- **UNIQUE** - Os valores do campo não podem ser repetidos;
- **PRIMARY_KEY** - É a chave primária da tabela, um valor único, não nulo e que serve para identificar um registro na tabela (Ex.: CPF);
- **FOREIGN_KEY** - É uma chave estrangeira, que serve para identificar um registro de outra tabela (Ex: uma tabela de **posts** pode precisar de uma chave estrangeira para relacionar cada post ao seu autor, que seria um registro em uma tabela de **usuários**);
- **DEFAULT** - Indica um valor padrão para o campo, caso ele não seja informado;
- **NOT NULL** - Define que o valor do campo não pode ser nulo/vazio;
- **AUTO_INCREMENT** - Indica que caso não seja passado um valor, o campo receberá o valor seguinte ao último valor definido.

Exemplo de sintaxe

```
CREATE TABLE users(
    id VARCHAR(255) PRIMARY KEY,
    email VARCHAR(255) UNIQUE,
    password INT NOT NULL,
);
```

Principais comandos

Criando tabelas

 Comando: `CREATE TABLE nome_da_tabela (nome1 tipo, nome2 tipo)`

Usado para criar uma tabela. Recebe o nome da tabela e os nomes e tipos de cada coluna. Deve indicar restrições das colunas(contraints) como chave primária, obrigatoriedade, colunas únicas, entre outras.

► Exemplo

Deletando tabelas

 Comando: `DROP TABLE nome_da_tabela`

Usado para deletar tabelas

► Exemplo

Inserindo valores em uma tabela

 Comando: `INSERT INTO nome_tabela (coluna1, coluna2, coluna3, ...)`

`VALUES (valor1, valor2, valor3, ...);`

Insert é usado para adicionar um item a uma tabela. Recebe o nome da tabela, os nomes das colunas que serão inseridas e os valores das mesmas.

► Exemplo

Deletando valores de uma tabela

 Comando: `DELETE FROM nome_tabela`

`WHERE condição;`

Usado para apagar uma ou mais linhas de uma tabela.

Quando usado sem a cláusula WHERE apagará todas as linhas da tabela. Recebe uma tabela e, opcionalmente, condições em seguida apaga as linhas que atendam às condições.

► Exemplo

Comando: TRUNCATE nome_tabela;

Similar ao comando DELETE mas utilizado para esvaziar tabelas inteiras.

É mais rápido por ser irreversível, o comando DELETE opera de forma mais lenta pois possibilita a reversão da deleção.

▼ Exemplo

```
TRUNCATE TABLE professores_labenu;
```

Consultando a estrutura de uma tabela

Comando: DESCRIBE nome_tabela;

Usado para conferir a **estrutura** de uma tabela. Com esse comando não é possível consultar os valores de cada item da tabela, a sim, os nomes das colunas e tipos de valores.

▼ Exemplo

```
DESCRIBE professores_labenu;
```

Consultando valores de uma tabela

Comando: SELECT coluna1, coluna2, ...

```
FROM nome_tabela
```

```
WHERE condicao;
```

Select é usado para pegar dados no banco e deve receber quais colunas devem ser buscadas e de qual tabela. Pode receber condições com a keyword WHERE. Se não forem passadas, todos os itens são retornados.

▼ Exemplo

```
SELECT nome
FROM professores_labenu
WHERE id = 1;
```

▼ Exemplo utilizando operadores condicionais

```
SELECT id, nome
FROM professores_labenu
WHERE nome LIKE "%arv%" OR
(idade > 23 AND idade < 27);
```

▼ O que são operadores condicionais?

Operadores Condicionais

- `=` - Igualdade
- `</>/<=/>` - Maior/menor (igual)
- `<>` - Diferente
- `AND` - Usado para adicionar mais de uma condição, devendo ser todas válidas
- `OR` - Usado para adicionar mais uma condição, sendo qualquer uma delas válida
- `IN` - como um `=`, mas que recebe múltiplos valores
- `NOT` - Nega condição
- `LIKE` - Usado para comparar strings. % é curinga

Condições podem ser acumuladas, sendo a prioridade determinada por parênteses.

Comando: `SELECT * FROM nome_tabela`

Normalmente, na programação o * significa tudo, ou até mesmo, universal. Por isso, utilizamos esse operador quando desejarmos pegar todos os valores de uma tabela em uma vez só.

Exemplo

```
SELECT * FROM professores_labenu;
```

Knex

Knex

Após entendermos sobre a criação de tabelas em um banco de dados, precisamos ligar, de alguma forma, o código do nosso VSCode com o Workbench, certo?

Faremos esse processo utilizando uma lib javascript chamada Knex juntamente com o *client* que estamos utilizando.

O que é client?

Instalando o Knex

Comando: `npm install knex`

```
npm install @types/knex -D
```

Lembre-se que para desenvolver em TS é necessário instalar os types de cada pacote!

Instalando o client

Comando: `npm install mysql`

Escolhemos trabalhar aqui no curso com o client MySQL. Mas você pode ficar livre para pesquisar mais sobre outros clients.

Estabelecendo uma conexão com o MySQL

Para criar essa conexão do código com o banco, precisaremos criar um arquivo de configuração em que colocaremos os dados que passamos pra vocês no slack. [aqui](#)

```
import knex from 'knex'

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

export default connection
```

Inserindo dados no banco

Há duas formas com as quais podemos inserir dados no banco. RAW e Query Builder.

RAW

permite que enviamos uma query para o banco usando a linguagem SQL diretamente,

dentro de uma template string.

Exemplo:

```
app.post("/actor", async (req, res) => {
  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")
  }
})
```

Ele devolve pra gente o resultado da query e outras informações dentro de um array.

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, e os dados que queremos estão na primeira posição do array então, para acessá-los basta acessar a primeira posição da resposta.

Query Builders

O Query Builder é uma funcionalidade do Knex que tende a facilitar a criação das queries, ao invés de ser a linguagem crua SQL, utilizamos funções! Além da escrita, facilita na hora de tratar os dados.

Exemplo

```
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")
  }
})
```

Exemplo II

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

Exemplo III

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

Dotenv

No código temos informações sensíveis que de forma alguma devem se tornar públicas. Por esse motivo, existem as chamadas variáveis ambientais, ou seja, existem apenas no ambiente de desenvolvimento e nenhum outro lugar. Apenas no seu computador.

Pra facilitar nossa vida, temos uma lib que faz todo o trabalho pesado pra nos! Basta instalar, criar o arquivo de config .dotenv e colocar suas variáveis ambientes.

Configurando o dotenv no projeto

1. Instalar a biblioteca com o seguinte comando

```
Comando: npm install dotenv
```

2. Criar um arquivo chamado `.env` na raiz do projeto. (na altura do package.json)
3. Dentro desse arquivo, definiremos uma chave ("nome") para cada uma das constantes. Exemplo DB_HOST, DB_USER. (geralmente colocado em caixa alta pra identificarmos como variáveis ambientes.)

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

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

4. Importar no arquivo de conexão com o banco a chamada do dotenv como no exemplo abaixo:

```
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
```

A partir de então, as variáveis de ambiente poderão ser acessadas por

```
process.env.NOME_DA_VARIAVEL
```

```
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
```

▼ Relações em SQL

Relações SQL

Imagine a estrutura de um banco de dados de uma rede social. Além de contas de usuários, essas contas podem seguir outras contas e serem seguidos. Todos esses dados são guardados em tabelas relacionais, ou seja, que se relacionam umas com as outras.

Uma tabela que guarda os dados da conta de um usuário é relacionada com outra tabela que guarda apenas as contas seguidas por esse usuário e vice-versa.

Relações 1:1

Um para um

Uma relação 1:1 acontece sempre que existe uma paridade, se relacionando com outra tabela de forma com que o registro da primeira tabela só aparece relacionado uma única vez na segunda tabela.

Exemplo: Em uma rede social podemos criar uma tabela `usuário` que guarda informações como: `nome` e `nome de exibição` e outra tabela com o nome `perfil`, que relaciona dados de perfil como: `e-mail` de login e `número de telefone`, por que na rede social esses dados não precisam ser consultados o tempo todo. Todo usuário tem um perfil associado e cada perfil só diz respeito a um usuário.



Guardamos esses valores em tabelas diferentes por questões de performance. Se alguma das colunas é lida ou alterada com muitíssima mais frequência do que as demais, talvez valha a pena guardar em tabelas diferentes.

Relações 1:N

Um para muitos

Caso em que **um** elemento de uma tabela se relaciona com **vários** elementos de outra tabela.

Seguindo o exemplo das redes sociais, temos a tabela `usuario` que contém uma relação com vários posts da tabela `posts`.

Exemplo: Um usuário cria diversos posts nas redes sociais, mas cada post só possui um usuário criador. Nessas relações, os elementos de uma das tabelas se relacionam com vários da outra tabela.

Dessa forma, a tabela do lado (N) do relacionamento recebe a chave estrangeira da tabela do lado (1) do relacionamento.

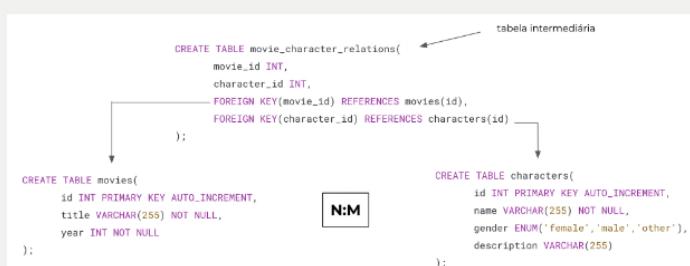


Relações N:M

muitos para muitos

São casos onde existe uma relação de muitos para muitos, de forma que uma tabela se relaciona com diversos elementos e estes elementos podem se relacionar com diversas tabelas. Nesse caso é necessário, então, criar uma **tabela intermediária**, também conhecida como **tabela junção**.

Exemplo: Em uma foto podem estar marcados diversos (N) usuários e um usuário pode estar em muitas (M) fotos. Dessa forma, se criar uma tabela junção `fotos_usuario`.



Criando relações em MySQL

Para relacionar duas tabelas, utilizamos uma keyword chamada `FOREIGN KEY`, que indica que a propriedade em questão é uma chave estrangeira que liga uma tabela a outra.

`FOREIGN KEY` deve sempre se referenciar a uma `PRIMARY KEY` da outra tabela.

Inserção de relações

Para criar um elemento que possui uma chave estrangeira, nós temos que passar para ele a chave de um elemento que já exista na outra tabela.

Por exemplo, antes de criar uma conta,

1. Precisamos ter um usuário criado;
2. Pegamos o id deste usuário;
3. Colocamos na query de criação da conta.

Deletando relações

Para deletar um elemento que foi usado na criação de um elemento de outra tabela, precisamos deletar todas as referências dele naquela tabela.

Logo, antes de deletar um post, precisamos deletar todas as referências que têm dele em outras tabelas. Outro exemplo é ao excluir, por exemplo, uma foto, precisamos excluir todas as referências desta foto nos álbuns dos usuários.

Buscando relações

- INNER JOIN

Retorna elementos relacionados nas duas tabelas. Utilizamos a cláusula `ON` para delimitar uma condição para que os dados sejam retornados.

► 🔍 Dica

- LEFT JOIN

Retorna todos elementos da tabela esquerda, e apenas os elementos associados da tabela direita.

► 🔍 Dica

- RIGHT JOIN

Retorna todos elementos da tabela direita, e apenas os elementos associados da tabela esquerda.

► 🔍 Dica

- JOIN

Retorna todos elementos que estão associados tanto na tabela direita ou esquerda.

Podemos utilizar o `JOIN` para retornar os registros que se relacionam através da tabela junção.

► 🔍 Dica