Banco de Dados

DigitalHouse >
Coding School

Desenho e definição de tabelas

DigitalHouse >
Coding School

O que é uma base de dados?

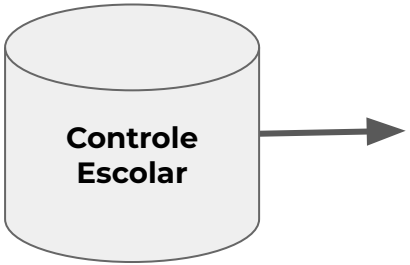
- Trata-se de um conjunto de dados pertencentes ao mesmo contexto organizados para um fim específico.
- É uma representação de aspectos da realidade.

Tem como função:

- **Armazenar** (adicionar, modificar, apagar) dados.
- **Acessar os** dados.
- **Manipular** dados e combiná-los.
- **Analisar** dados.

DigitalHouse> Coding School

Exemplo



Estudantes

ID	Nome	Sobrenome
	Rocco	Verdi
	Marian a	Lesco

Atribuições

Estudante_id	Curso_id
	MAT11
	DIB11

Cursos

Código	Título	Matéria
MAT11	Matemática	Ciência
DIB11	Desenho	Arte

Como é que eles funcionam?

Um *sistema de gestão de bases de dados* (*DataBase Management System*) é um sistema que permite a **criação**, **gestão** e **administração** de bases de dados, bem como a seleção e tratamento das estruturas necessárias ao armazenamento e pesquisa de informação da forma mais eficiente possível.

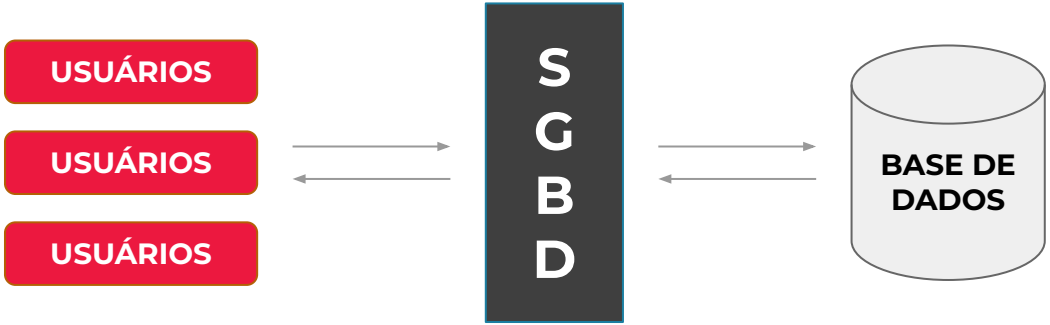
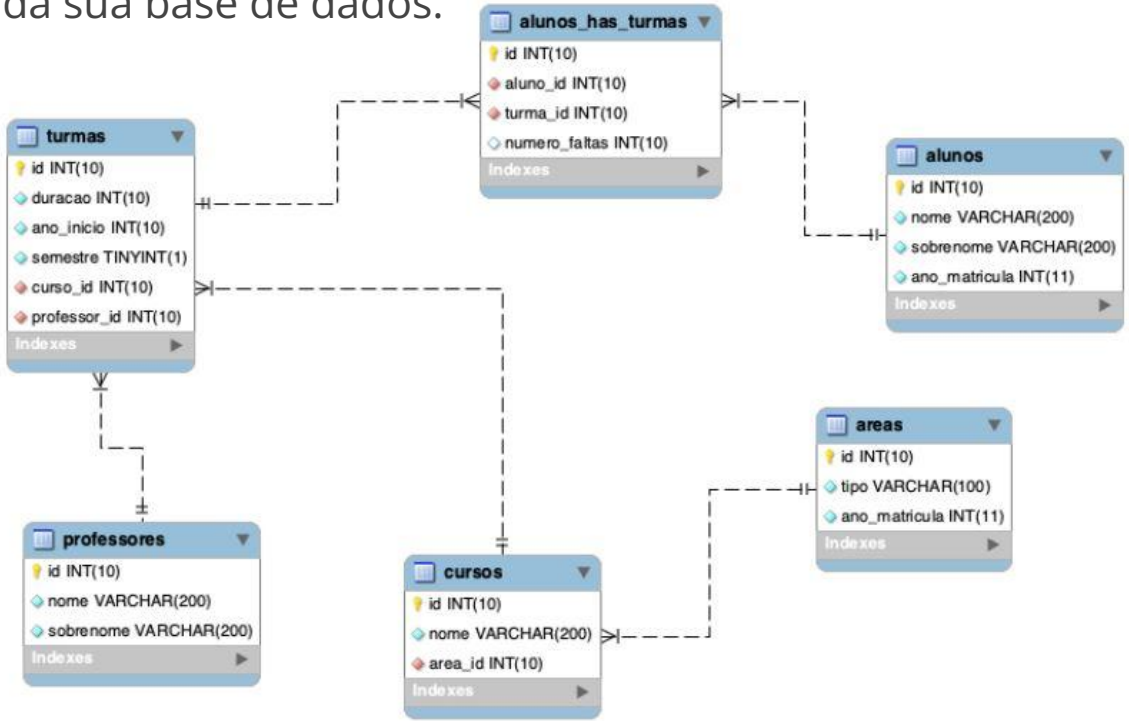


Diagrama relacional

Os diagramas relacionais permitem compreender de forma rápida e fácil **todas as** tabelas da sua base de dados.



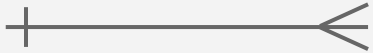
Chave primária

Para identificar a chave primária em uma entidade, podemos escrever o atributo em negrito seguido das iniciais **PK** entre parênteses.

FILMES
id (PK) título classificação data_de_lançamento país

Cardinalidade

É a forma como as entidades se relacionam umas com as outras. É na cardinalidade que são definidos os graus de relação entre duas entidades ou tabelas.

Cardinalidade	Se lê	Representação
1:1	**Um** a **um**	
1:M	**Um** para **muitos**	
M:1	**Muitos** para **um**	
M:M	**Muitos** para **muitos**	

Numérico sem decimais

TINYINT
-128 a 128, 0
a 255

SMALLINT
-32768 a 32767, 0
a 65535

MEDIUMINT
-8388608 a 8388607, 0
a 16777215

INT
-2147483648 a 2147483648, 0
a 4294967295

BIGINT
-9223372036854775808 a
9223372036854775807
0 a 18446744073709551615

DigitalHouse> Coding School

Dados do tipo de data

Ao armazenar datas, tenha em mente que o MySQL não verifica rigorosamente se uma data é válida ou não.

DATE
Armazena apenas a data no formato **AAAAA-MM-DD**.

TIME
Armazena apenas a hora no formato **HH:MM:SS**.

DATETIME
Corresponde a uma representação de data e hora completa, ou seja, por exemplo: 3 de Janeiro de 1967 às 8:00 da manhã, que é armazenada da seguinte forma: **1967-01-03 08:00:00:00**.

DigitalHouse> Coding School

Dados de tipo de texto

CHAR(num)

O número entre parênteses indicará o **número exato** de caracteres.

Exemplo*:* char(100) → 100 caracteres **sempre**

VARCHAR(num)

O número entre parênteses indicará o **número máximo** de caracteres.

Exemplo*:* varchar(100) → 100 caracteres **no máximo**

TEXT

Específica um tipo de dado de texto sem limite de caracteres. Típicamente usado para um post de blog, por exemplo.

Constraints (Restrições)

UNIQUE KEYS

Uma unique key é uma constraint que só permite valores únicos para um (ou vários) campos.

NULL / NOT NULL

"NULL" significa que o valor para esse campo não existe ou não é conhecido. "NULL" **não** é **nem** vazio ' ' (*em branco*) nem zero (0).

DEFAULT

Usado para definir um **valor padrão** para uma coluna. Este valor será anexado a cada novo registro, desde que nenhum outro valor seja especificado para sobregravá-lo.

Constraints (Restrições)

AUTO_INCREMENT

Gera um número único e incrementa-o automaticamente com cada novo registro na tabela. Muitas vezes usado para o campo **ID.**

DigitalHouse> Coding School

exemplo de TABELA DE CRIAÇÃO

SQL

```sql
CREATE TABLE movies (
    id  INT UNSIGNED PRIMARY KEY AUTO INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) UNSIGNED NOT NULL,
    awards INT UNSIGNED DEFAULT 0,
    release_date DATE NOT NULL,
    length INT UNSIGNED NOT NULL
);
```


DigitalHouse> Coding School

Comando DROP TABLE

DROP TABLE eliminará a tabela especificada na instrução.

```sql
DROP TABLE IF EXIST movies;
```

Exemplos TABELA ALTER

SQL
```sql
ALTER TABLE movies
ADD rating DECIMAL(3,1) UNSIGNED NOT NULL;
```

Adiciona a coluna de **classificação**, esclarecendo o tipo de dado e a restrição.

SQL
```sql
ALTER TABLE movies
MODIFY rating DECIMAL(4,1) UNSIGNED NOT NULL;
```

Modifica o decimal da coluna de **classificação**. Mesmo que o resto das configurações da tabela não seja modificado, é necessário escrevê-las na declaração.

SQL
```sql
ALTER TABLE movies
DROP rating;
```

Elimina a coluna de **classificação**.

DigitalHouse> Coding School

Foreign Key

Uma **FOREIGN KEY** é uma restrição que permite que uma coluna de uma tabela armazene como valor um determinado dado que se refere a outra tabela, e que serve para relacionar essas duas tabelas uma com a outra.

Geralmente, a **FOREIGN KEY** está relacionada ou ligada à **FOREIGN KEY** da outra tabela com a qual queremos criar a relação.

usuários		
id (**PK**)	e-mail	nome_do_utilizador

imagens		
id (**PK**)	url	user_id (**FK**)

DigitalHouse> Coding School

Full Stack Node.js

Diagrama de Relacionamento

DigitalHouse>

O que vimos no playground

- Introdução a Base de Dados
- Tabelas
- Relacionamentos
- Tipos de Dados
- Constraints
- Create, Drop, Alter
- Chave Estrangeira

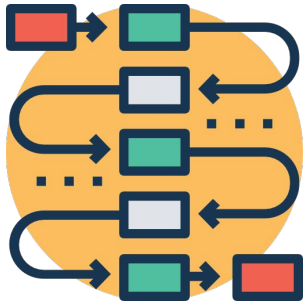
POR QUE **PLANEJAR** UMA BASE DE DADOS?

O que vamos ver hoje

- Modelagem de Dados
- Precisamos pensar na estrutura e tipos de dados

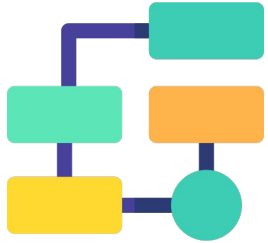
Planejar para...

- Manutenção custa mais que um bom planejamento
- Criar uma base escalável (crescimento do produto)
- Prever cenários diversos



A nível de banco de dados

- Entender entidades do sistema
- Representar o ambiente observado
- Documentar



E ONDE PODEMOS **MODELAR** ISSO?

Workbench

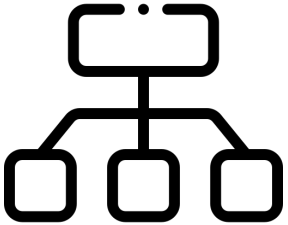
- Download: https://dev.mysql.com/downloads/workbench/
- Clique no menu em **File > new Model > Add Diagram**

Alternativa:

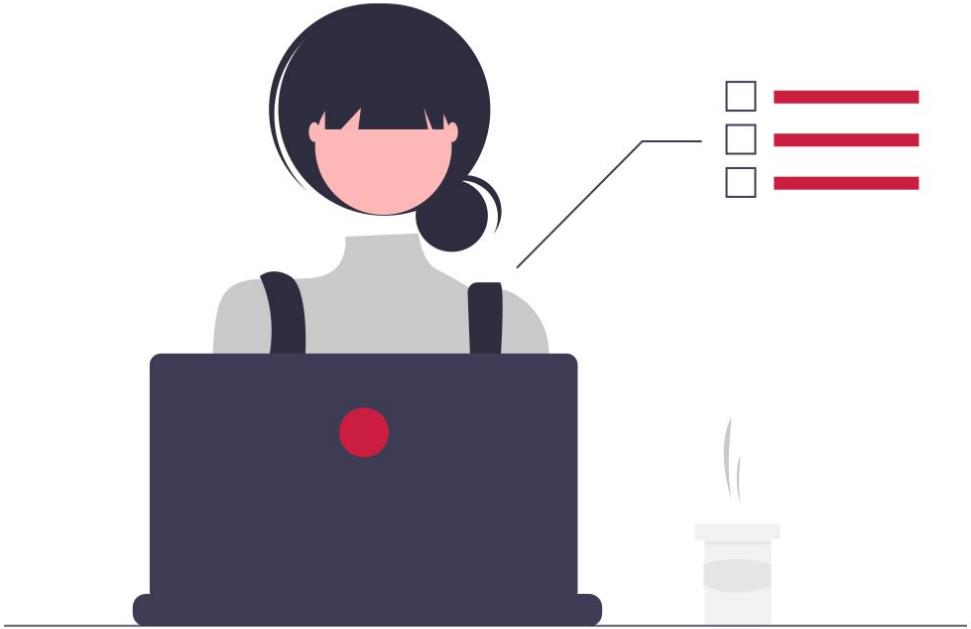
- **Lucidchart** - conta gratuita https://www.lucidchart.com/pages/pt

- **Miro** - conta gratuita https://miro.com/login/

- **Qualquer** outra ferramenta que permita criar visualização de tabela

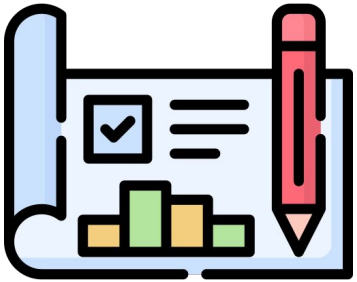


Hora de praticar!



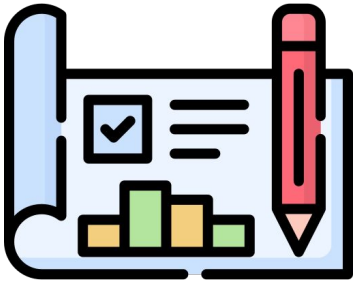
Atividade I - Planejando a base de dados de uma escola

- Um cliente nos contrata para o planejamento de uma base de dados para sua empresa, uma escola, a ***DHSchool*** (qualquer coincidência é mera semelhança)
- Pedimos então para que ele nos explique um pouco sobre o dia a dia da empresa, quais informações são importantes e devem ser armazenadas
- Ele nos respondeu dizendo que a escola possui áreas de conhecimento, e com base nelas, os cursos são planejados e criados...



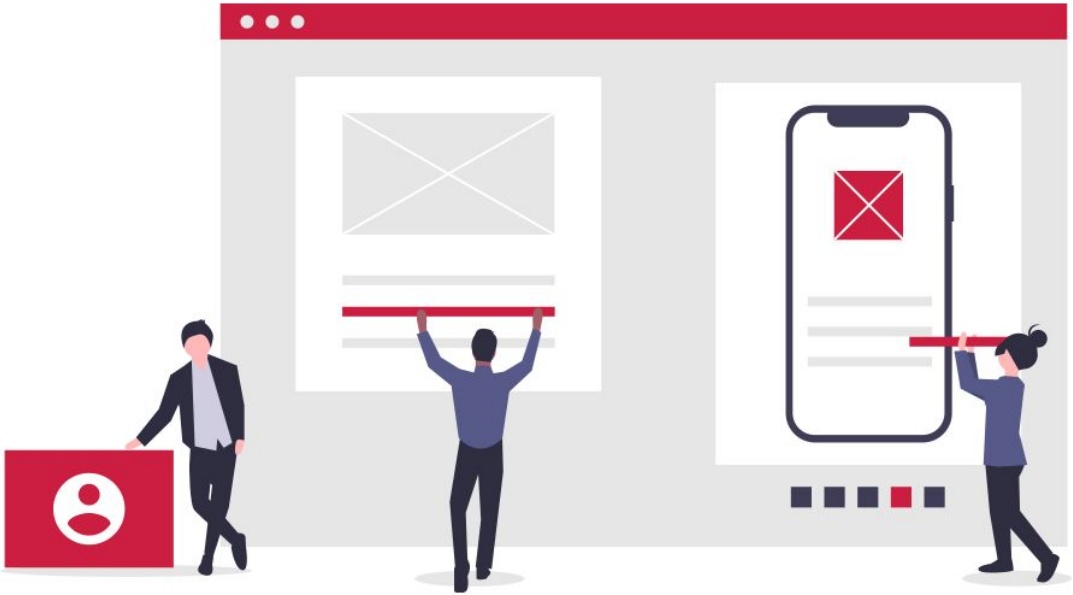
Atividade I - Planejando a base de dados de uma escola

- A partir dos cursos são criadas semestralmente diferentes turmas.
- Os alunos podem se matricular em quantas turmas desejarem.
- Ao fim da reunião, o cliente nos informou que é de grande importância armazenarmos o nome completo dos professores (estes podem ter várias turmas também).



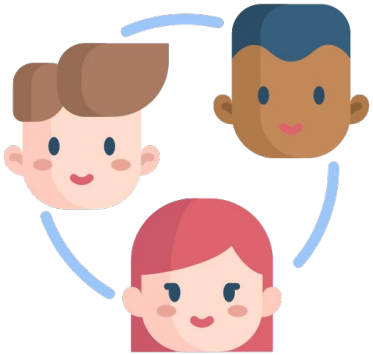
Com tais informações, o que é possível fazer do mapeamento até aqui?

Hora de praticar!



Atividade II - Fazendo o PI dar Certo

- Nada melhor do que planejar um banco de dados sendo você a pessoa que possui a ideia!
- Chegou a hora dos grupos do Projeto Integrador se reunirem e planejarem juntos a modelagem da base de dados do sistema em desenvolvimento



Guia de instalação Linux

DigitalHouse >
Coding School

1 | MySQL

“ **MySQL** é um **administrador** de **banco de dados relacionais** que usaremos durante o curso. Embora existam outros, este é o mais utilizado. ”

Como instalar

No terminal, execute:

```
>_ sudo apt update
```

```
>_ sudo apt install mysql-server
```

2 MySQL Workbench

“ **MySQL Workbench** é uma ferramenta multiplataforma que nos permitirá trabalhar com **bancos de dados** em forma **gráfica** . ”

Como instalar

No terminal, execute:

```
>_ sudo dpkg -i mysql-apt-config_0.5.3-1_all.deb
```

```
>_ sudo apt-get update
```

```
>_ sudo apt-get install mysql-workbench-community
```

Guia de instalação Mac

DigitalHouse >
Coding School

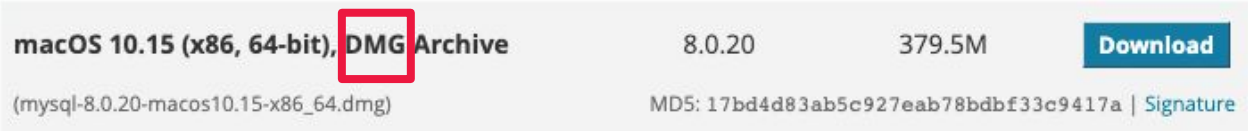
1 | MySQL

“ **MySQL** é um **administrador** de **banco de dados relacionais** que usaremos durante o curso. Embora existam outros, este é o mais utilizado. ”

Como instalar

1. Visitar o site: https://dev.mysql.com/downloads/mysql.

1. Selecionar a versão DMG.



1. Na página para onde nos direciona, na parte inferior, aparece a opção de baixar sem se cadastrar.

Como instalar

1. Extrair o arquivo baixado.

1. Abrir o arquivo extraído e seguir as instruções de instalação.

Guia de instalação Windows

DigitalHouse >
Coding School

1 | XAMPP

“ **XAMPP** é um ambiente de desenvolvimento fácil de instalar. Inclui o banco de dados **MySQL** que usaremos no curso. ”

Como instalar

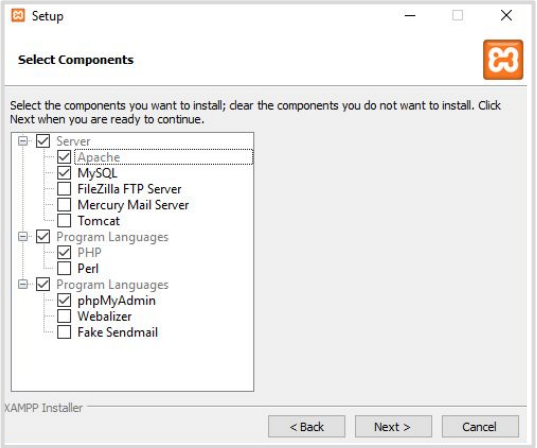
1. Visitar o site https://www.apachefriends.org/pt_br/index.html.

1. Clicar no link para baixar.

1. Na tela seguinte, selecionar “Seguinte”.

Como instalar

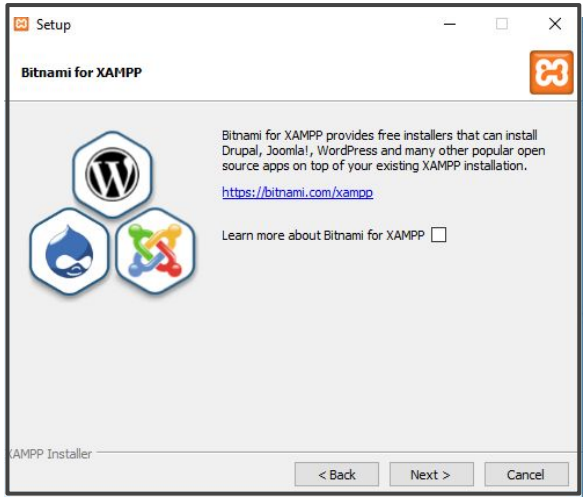
Na próxima tela, recomendamos **desmarcar** a seleção das opções dos programas que não utilizaremos no curso. Deixe apenas Apache, MySQL e PHP marcados. *



** Apesar de não usarmos PHP e Apache no curso, não é possível desmarcá-los no processo de instalação.*

Como instalar

Nesta tela, **desmarque** a opção “Mais informações sobre Bitnami para XAMPP”.

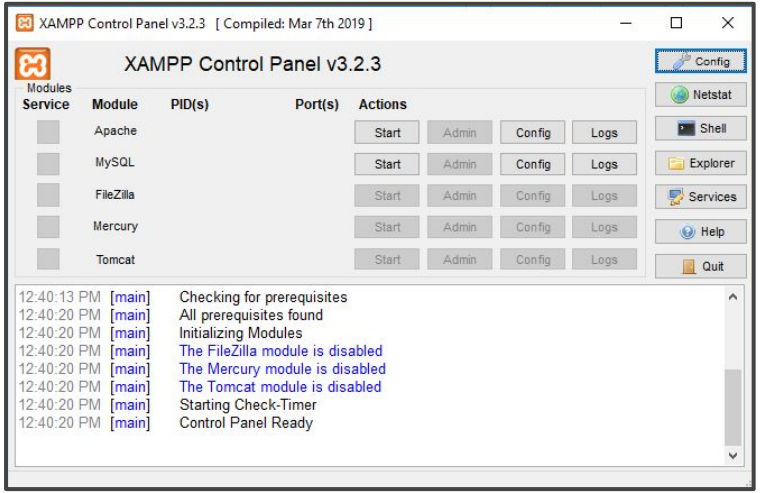


Como instalar

Abra o XAMPP no painel de controle.

Como instalar

Devemos ver a seguinte tela.



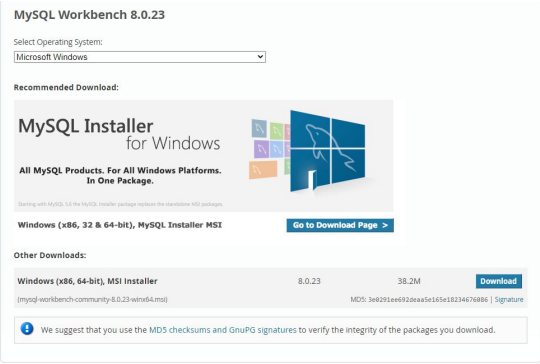
2 | MySQL Workbench

“ **MySQL Workbench** é uma ferramenta multiplataforma que nos permitirá trabalhar com **bancos de dados** de forma **gráfica.** ”

Como instalar

1. Visitar o site https://dev.mysql.com/downloads/workbench/.

1. Selecione o sistema operacional do seu PC (instalador do Windows 64).

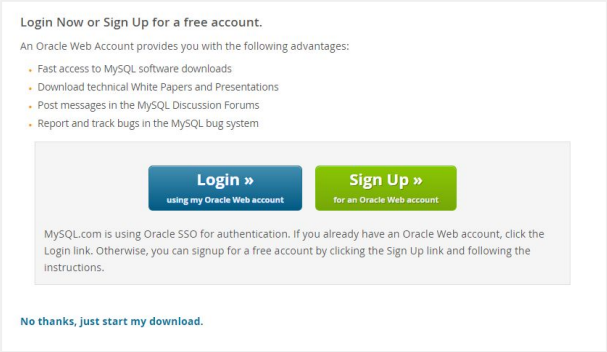


1. aperte o botão Download para baixar o executável.

Como instalar

4. O arquivo não baixará automaticamente. Ao invés disto, você será redirecionado para outra página. Nessa página, clique no link no final dela:

No thanks, just start my download.



5. Execute o arquivo baixado e siga as instruções.

Introdução a banco de dados

DigitalHouse >
Coding School

Índice

1. Bancos de Dados
2. Sistemas de gestão de bancos de dados
1. MySQL

1 Bancos de Dados

O que é um banco de dados?

- É um conjunto de dados pertencentes a um mesmo contexto organizado para uma finalidade específica.
- É uma representação de aspectos da realidade.

Nos servem para:

- **Armazenar** (adicionar, modificar, eliminar) dados.
- **Acessar** os dados.
- **Manipular** dados e combiná-los.
- **Analisar** dados.

Tabelas

- Dentro de um banco de dados, registros específicos são organizados em **tabelas** com **linhas** e **colunas**.

	Coluna 1	Coluna 2
Linha 1		
Linha 2		

Linhas: contêm os dados/registros especificamente.
Colunas: representam os atributos/campos de cada dado/registro .

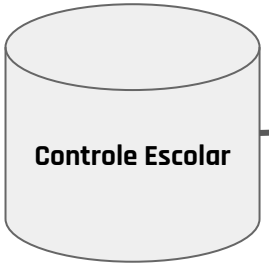
Tabelas

- Cada linha é identificada por uma ou mais colunas, que torna o registro único. Isso é chamado de **chave primária**, também conhecido como **primary key** (PK). Em geral, é uma coluna chamada **id**, mas pode ser mais de um dado da tabela.
- Ou seja, se nós tivermos linhas com valores repetidos, é a chave primária que vai tornar os registros diferentes uns dos outros.

	id	nome
Linha 1	1	Maria de Souza
Linha 2	2	Maria de Souza

Fique tranquilo: Vamos explorar isso melhor nas próximas aulas

Exemplo



Alunos

ID	Nome	Sobrenome
1	Rocco	Verdi
2	Mariana	Lesco

Matrículas

Id_aluno	Id_curso
1	MAT11
2	DIB11

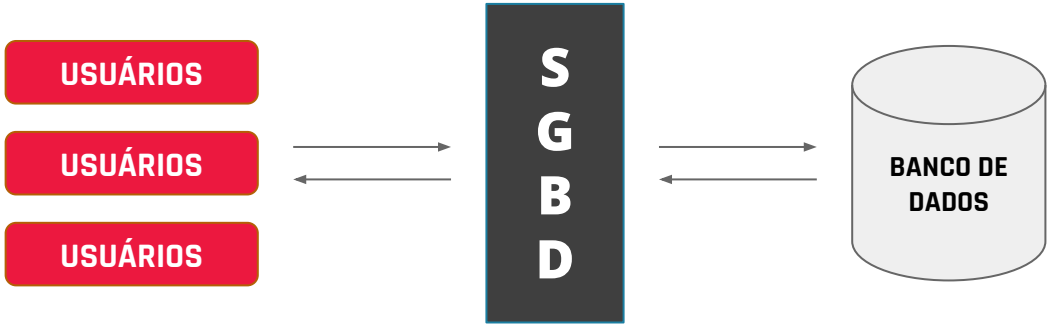
Cursos

Código	Título	Área
MAT11	Mate	Ciência
DIB11	Dibujo	Arte

2 Sistemas de gestão de bancos de dados

Como funcionam?

Um gerenciador de banco de dados (DataBase Management System) é um sistema que permite a **criação**, **gerenciamento** e **administração** de bancos de dados, bem como a escolha e gerenciamento das estruturas necessárias para o armazenamento e busca de informação da forma mais eficiente possível.



Alguns exemplos

PostgreSQL

MySQL

Amazon Redshift

Google BigQuery

MariaDB

Microsoft SQL Server

amazon web services

ORACLE

Google Cloud Platform

EXASOL

Amazon Aurora

3 | MySQL

Características Principais

Multiplataforma

Funciona em sistemas Unix, Windows, Mac, entre outros.

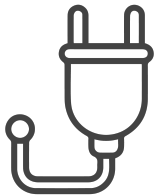
Padrão Sql92

Tipos de dados, codificação, Joins.

Open Source

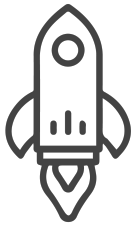
Adaptável a diversas necessidades.

Características Principais



Relacional

Normalização integridade referencial.



Velocidade

Rapidez em transições.

Multiconexões

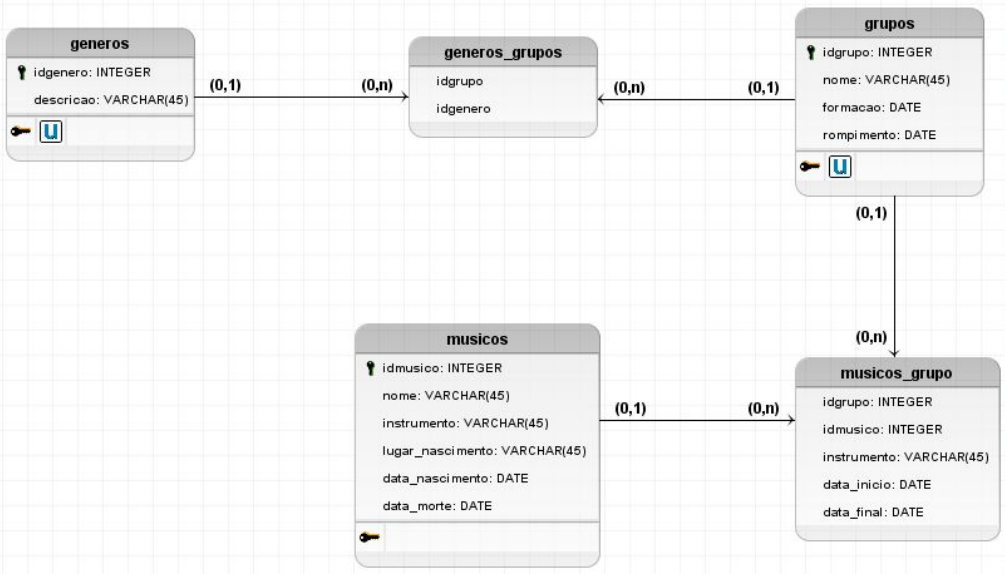
Segurança multi-threads. Múltiplos usuários.

Tabelas

DigitalHouse >
Coding School

Diagrama de relacionamento

Esses diagramas nos permitem entender rápida e facilmente **todas** as tabelas em nosso banco de dados .



1 Estrutura

Entidade

Dentro do nosso sistema, teremos **entidades**, que serão a representação de um objeto ou coisa na vida real. Para integrá-los em nosso diagrama, vamos representá-los usando um retângulo. Para o nome dessas tabelas, usaremos substantivos no **plural**.

Atributos

São as **características** que definirão cada entidade. Por exemplo, a entidade **Filmes** pode ter os seguintes atributos:

FILMES
título rating data de estreia país

Chave primária

Uma **Chave Primária** ou **Primary Key** é um campo (ou uma combinação de campos) que identifica **exclusivamente** cada linha de uma tabela. Em outras palavras, não pode haver duas linhas em uma tabela que tenha a mesma **PK**.

Não é obrigatório ter uma, mas é altamente recomendável que cada linha de uma tabela tenha alguma exclusividade.

id	titulo	rating	data_estreia	pais
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill Vol. 1	9.5	2003-11-27	Estados Unidos

Chave primária

Para identificar a chave primária em uma entidade, podemos escrever o atributo em negrito seguido pelas iniciais **PK** entre parênteses.

FILMES
id (PK) título rating data_estreia pais

Relacionamentos

DigitalHouse >
Coding School

Índice

1. Definição
2. Tipos de relacionamentos

1 | Definição

O que são?

Relacionamentos indicam como duas tabelas devem ser relacionadas. Em um banco de dados, existem 3 tipos de relacionamento:

- **Um** a **um.**
- **Um** para **muitos.**
- **Muitos** para **muitos.**

Como podemos saber como uma entidade está relacionada a outra?
Trazendo um exemplo concreto que nos ajuda a definir como essas duas entidades interagem entre si.

Cardinalidade

É a forma que as entidades se relacionam.

Cardinalidade	Se lê	Representação
1:1	**Um** para **um**	
1:N	**Um** para **muitos**	
N:1	**Muitos** para **um**	
N:N ou N:M	**Muitos** para **muitos**	

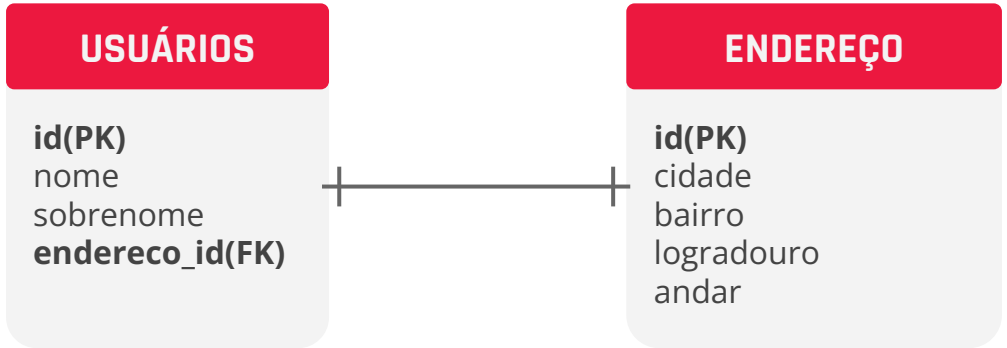
Chave estrangeira (FK)

A chave estrangeira (**Foreign Key** ou **FK**) é, de modo geral, a chave primária (**Primary Key**) de outra tabela.
Assim, quando dizemos que um registro tem uma FK, significa que aquela informação está **associada** com a informação primária de outra tabela.

Importante: A informação que será usada como FK precisa existir **antes** de criar o registro na tabela secundária.

Chave estrangeira (FK)

Veja esse exemplo. Nele, estamos indicando que um usuário tem um **endereço_id.** Antes de criar o usuário, o endereço precisa estar cadastrado.

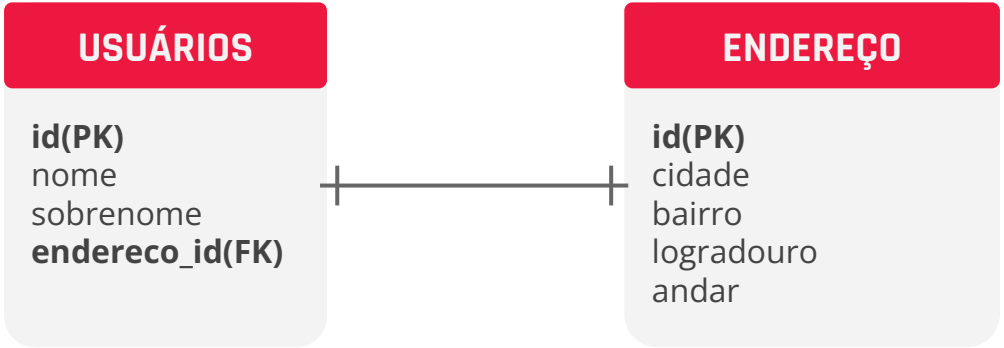


2 Tipos de Relacionamentos

Um para um (1:1)

Um usuário **possui** apenas um endereço. Um endereço **pertence** a apenas um **usuário**.

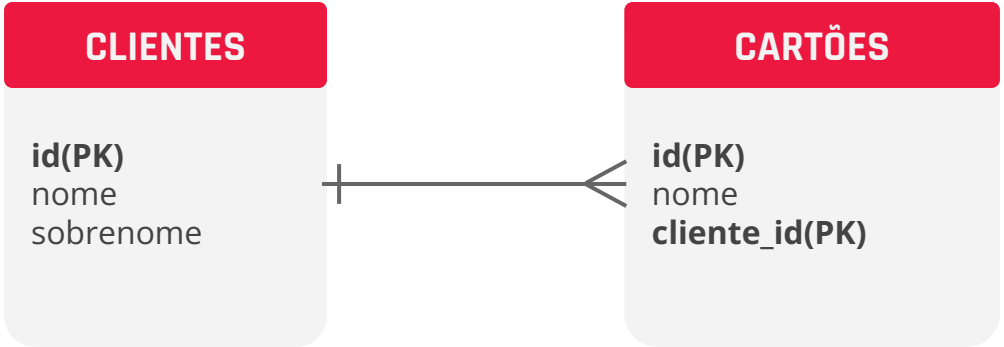
Para estabelecer a relação, colocamos a **chave primária** do endereço na tabela de usuários, indicando que **este** endereço está associado a **esse** usuário.



Um para muitos (1:N)

Um **cliente** pode ter vários **cartões**. Um cartão pertence a **apenas um** cliente.

Para estabelecer a relação, colocamos a **chave primária** do cliente na tabela de cartões, indicando que esses cartões estão associados a um determinado usuário.



Muitos para muitos (N:M)

Um cliente pode comprar **muitos produtos**. **Um produto** pode ser comprado por **varios clientes**. Nas relações N: M, no banco de dados, a própria relação **torna-se uma tabela**. Essa tabela intermediária - também conhecida como tabela dinâmica - possui, pelo menos, 3 informações: uma chave primária **(PK)** e duas chaves estrangeiras **(FK)**, cada uma referindo-se a cada tabela no relacionamento.

Muitos para muitos (N:M)

Neste exemplo, **cliente_produto** seria nossa tabela intermediária. Cada linha nesta tabela representa um cruzamento entre cliente e produto. Pode ser, neste caso, uma compra:

- A linha 1 indica que o **cliente 1** (Luan) comprou o **produto 1** (bola).
- A linha 2 indica que **Luan** também comprou o **produto 2** (laptop).
- A linha 3 indica que um **laptop** também foi comprado pelo **cliente 3** (Marta).

CLIENTES

id	nome	sobrenome
1	Luan	Pereira
2	Clara	Sanches
3	Marta	Rios

CLIENTE_PRODUTO

id	produto_id	cliente_id
1	1	1
2	2	1
3	2	3

PRODUTOS

id	nome
1	Bola
2	Laptop

Tipos de dados

DigitalHouse >
Coding School

Índice

1. Definição
2. Dados de tipo numérico
3. Dados de tipo data
4. Dados de tipo texto

1 Definição

Tipos de dados

Os dados ou atributos de cada registro em uma tabela devem ser de um tipo específico de dados.

Quando projetamos um banco de dados, temos que pensar que tipo de dado é útil para nosso modelo.

Cada tipo de dado tem um determinado tamanho e quanto mais precisão colocarmos nesse tipo de dado, mais **rápido** e com **melhor desempenho** o MySQL funcionará.

012345
Tipos numéricos

2020-05-09
Tipos de data

Ad lorem ipsum
Tipos de texto

2 Dados de tipo numérico

Numéricos sem decimais

TINYINT

-128 a 128, 0 a 255

SMALLINT

-32768 a 32767, 0 a 65535

MEDIUMINT

-8388608 a 8388607, 0 a 16777215

INT

-2147483648 to 214748364, 0 a 4294967295

BIGINT

-9223372036854775808 a 9223372036854775807

0 a 18446744073709551615

Booleanos

O MySQL salva booleanos por trás dos panos, como zero ou como um. Por motivos de desempenho, não é recomendado usar este tipo de dados no MySQL.

No caso de desejarmos salvar valores "verdadeiros" e "falsos", podemos usar o tipo de dados **tinyint** e usar 0 para representar **falso** e 1 para representar **verdadeiro**.

3 Dados de tipo data

Datos de tipo data

Ao armazenar datas, lembre-se de que o MySQL não verifica estritamente se uma data é válida ou não.

DATE

Armazena apenas a data no formato **YYYY-MM-DD**.

TIME

Armazena apenas a hora no formato **HH:MM:SS**.

DATETIME

Corresponde a uma representação completa da data e hora, ou seja, algo como 3 de janeiro de 1967 às 8h00, que é armazenado da seguinte forma: **1967-01-03 08:00:00**.

4 Dados de tipo texto

Dados de tipo texto

CHAR(num)

O número entre parênteses indicará o **número exato** de caracteres.

Exemplo: char (100) → 100 caracteres **sempre**.

VARCHAR(num)

O número entre parênteses indicará o **número máximo** de caracteres.

Exemplo: varchar (100) → 100 caracteres **no máximo**.

TEXT

Determina um dado do tipo texto sem limite de caracteres. Geralmente é usado para uma postagem de blog, por exemplo.

Constraints

DigitalHouse >
Coding School

Constraints

UNIQUE KEYS

Uma chave exclusiva é uma restrição que permite apenas valores exclusivos para um (ou vários) campos.

NULL / NOT NULL

"NULL" significa que o valor desse campo não existe ou não é conhecido. "NULL" não é vazio, “(em branco) ou zero (0)”.

DEFAULT

Usado para definir um **valor** padrão para uma coluna. Este valor será adicionado a cada novo registro, desde que nenhum outro valor seja especificado para substituí-lo.

AUTO_INCREMENT

Ele gera um número único e o incrementa automaticamente a cada novo registro na tabela. Geralmente é usado para o campo **ID**.

Create, Drop, Alter

DigitalHouse >
Coding School

Índice

1. Create table
2. Drop table
3. Alter table

1 Create table

Comando CREATE TABLE

Com **CREATE TABLE**, podemos criar uma tabela do zero, junto com suas colunas, seus tipos e suas restrições.

SQL

```sql
CREATE TABLE nome_da_tabela (
    nome_da_coluna_1 TIPO_DE_DADO CONSTRAINT,
    nome_da_coluna_2 TIPO_DE_DADO CONSTRAINT
)
```

SQL

```sql
CREATE TABLE post (
    id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200) NOT NULL
)
```

Exemplo CREATE TABLE

SQL

```sql
CREATE TABLE movies (
    id  INT UNSIGNED PRIMARY KEY AUTO INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) UNSIGNED NOT NULL,
    awards INT UNSIGNED DEFAULT 0,
    release_date DATE NOT NULL,
    length INT UNSIGNED NOT NULL
);
```

FOREIGN KEY

Quando criamos uma coluna que contém uma chave estrangeira, será necessário usar a instrução **FOREIGN KEY** para esclarecer a qual tabela e a qual coluna os dados se referem.

É importante observar que a tabela "**clientes**" deve existir antes de executar esta instrução para criar a tabela "pedidos".

SQL

```sql
CREATE TABLE orders (
    order_id INT NOT NULL,
    order_number INT NOT NULL,
    client_id INT,
    PRIMARY KEY (order_id),
    FOREIGN KEY (client_id) REFERENCES clients(id)
);
```

2 | Drop table

Comando DROP TABLE

DROP TABLE irá deletar a tabela que especificamos na declaração.

```sql
DROP TABLE IF EXIST movies;
```

3 | Alter table

Comando ALTER TABLE

ALTER TABLE permite que você altere uma tabela existente e funcionará com três comandos:

- **ADD**: para adicionar uma coluna.
- **MODIFY**: para modificar uma coluna.
- **DROP**: para apagar uma coluna.

Exemplos ALTER TABLE

SQL
```sql
ALTER TABLE movies
ADD rating DECIMAL(3,1) UNSIGNED NOT NULL;
```

Adicione a coluna **rating**, esclarecendo o tipo de dados e a restrição.

SQL
```sql
ALTER TABLE movies
MODIFY rating DECIMAL(4,1) UNSIGNED NOT NULL;
```

Modifica o decimal da coluna de **rating**. Embora o resto das configurações do campo não sejam modificadas, elas devem ser escritas na instrução.

SQL
```sql
ALTER TABLE movies
DROP rating;
```

Apaga a coluna **rating**.

Foreign Keys

DigitalHouse >
Coding School

Foreign Key

A **FOREIGN KEY** é uma restrição que permite a uma coluna de uma tabela guardar como valor um dado particular que se refere a outra tabela, e que serve para relacionar estas duas tabelas entre si.

Geralmente, a **FOREIGN KEY** está relacionada ou ligada à **PRIMARY KEY** da outra tabela com a qual queremos criar o relacionamento.

usuários		
id (**PK**)	email	nome_usuario

imagens		
id (**PK**)	url	id_usuario (**FK**)

Foreign Key

Como vimos na tabela de usuários, temos uma "coluna id" que é a **primary key** dessa tabela e, por sua vez, as colunas "email" e "nome_de_usuário".

Por outro lado, na tabela de imagens também temos uma **primary key** que representa o id da imagem. Também temos a coluna url da imagem e a última coluna: "user_id" que está marcada como **foreign key**. Isso significa que cada registro presente nesta tabela salvará, nesta coluna, o ID do usuário a que corresponde a referida imagem.

Exemplo Foreign Key

usuários		
id (**PK**)	email	nome_usuario
1	rick@email.com	rick_sanches
2	morty@email.com	morty_smith
3	homero@email.com	homero_simpson

A imagem com **id: 1** pertence ao usuário com **id 2: morty_smith**

A imagem com **id: 2** pertence ao usuário com **id 3: homero_simpson**

imagens		
id (**PK**)	url	id_usuario (**FK**)
1	1607023695616.jpg	**2**
2	1290440058422878.png	**3**
3	19284284935008.jpg	2
4	1084741033580100.jpg	2
5	1147414967273946.png	3

Full Stack Node.js

Manipulação de dados e consulta

DigitalHouse>

O que vimos no playground

- Insert, Update, Delete
- Select
- Where e Order By
- Between e Like
- Limit e Offset
- Alias
- Funções de Alteração

O que vamos ver hoje

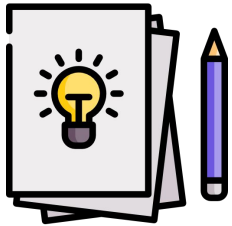
- Como criar uma base de dados
- Inserção de dados
- Alterando dados cadastrados
- Queries básicas

QUAL A DIFERENÇA ENTRE OS COMANDOS **CREATE** E **INSERT**?

COMO TIRAR DO PAPEL E CRIAR UM **BANCO DE DADOS**?

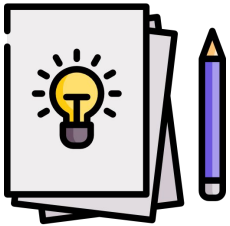
Como criar uma base de dados

```sql
CREATE DATABASE nome_base_de_dados
```



Como criar uma tabela

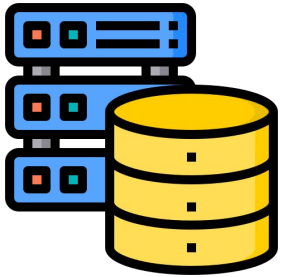
```sql
CREATE TABLE nome_tabela (
    coluna especifcações campo,
    coluna especifcações campo
)
```



Vamos praticar!

Atividade I - Populando uma Base de Dados

- Crie a base de dados **escola**
- Crie ao menos 4 das 7 tabelas contidas na modelagem



Instruções detalhadas no documento da **Atividade I**

Inserindo dados em uma tabela

SQL

```
INSERT INTO nome_tabela (coluna, coluna, coluna)
VALUES (valor, valor, valor)
```

SQL

```
INSERT INTO usuarios (email, senha, username)
VALUES ('usuario@email.com', '123@senha',
'user')
```

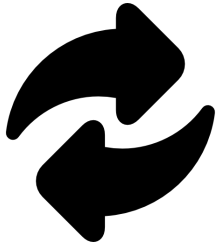
Alterando dados em uma tabela

SQL

```
UPDATE  nome_tabela  SET  coluna = valor
WHERE  condição
```

SQL

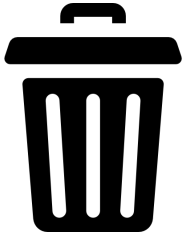
```
UPDATE usuarios SET username = ‘digitalHouser’
WHERE id = 1
```



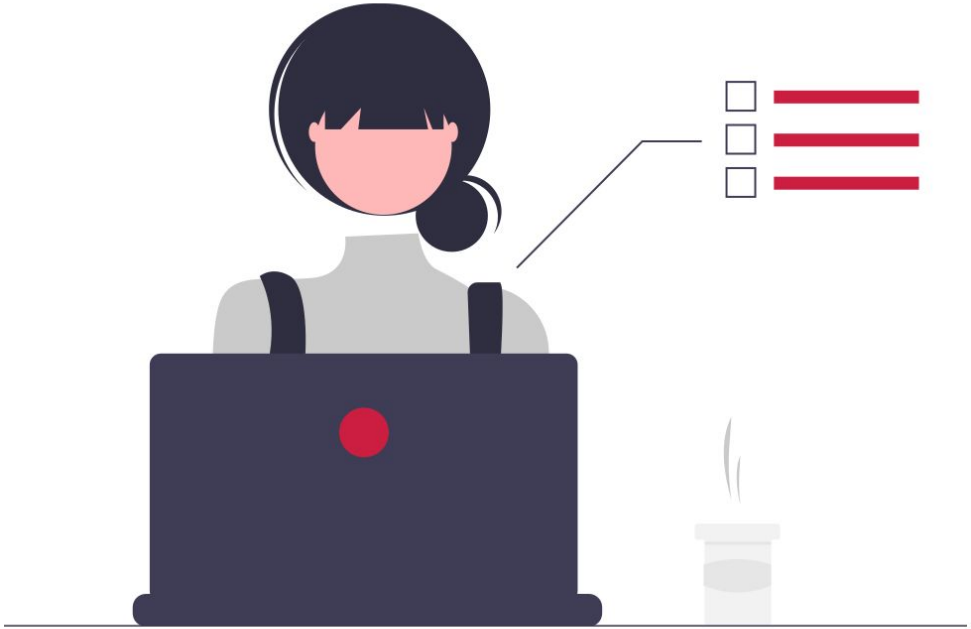
Deletando dados em uma tabela

```sql
DELETE FROM nome_tabela
WHERE condição
```

```sql
DELETE FROM usuarios WHERE id = 1
```

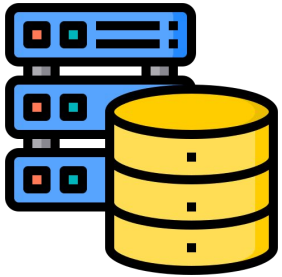


Hora de praticar!



Atividade II - Populando uma Base de Dados

- Utilizando a base de dados **X**
- Executar comandos no documento da **Atividade II**



Selecionando dados

```sql
SELECT coluna(s) FROM nome_tabela
```

Selecionando dados

Podemos otimizar esta busca com alguns complementos ao select, como:

Para filtrar registros retornados:	**WHERE**
Ordenar os registros retornados:	**ORDER BY**
Complementar a busca filtrada:	**BETWEEN | LIKE**
Limitar número de registros a retornar:	**LIMIT**

Selecionando dados

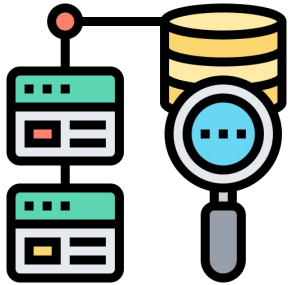
Podemos otimizar esta busca com alguns complementos ao select, como:

Indica a partir de qual linha de registro retornar:	**OFFSET**
Alterar nome de tabela ou coluna no retorno de uma busca:	**ALIAS**

Hora de praticar!

Atividade III - Buscando informações

- Montar as queries necessárias para cumprir com o enunciado no documento **Atividade III**
- Preferencialmente deixar todas as queries em um único arquivo sql (será mais fácil de organizar para estudo)



Insert, Update e Delete

DigitalHouse>

Insert

Existem duas maneiras de adicionar dados em uma tabela:

- Inserindo dados em **todas** as **colunas**;
- Inserindo dados nas **colunas** que **especificamos**.

Todas as colunas

Se estivermos inserindo dados em todas as colunas, não é necessário esclarecer os nomes de cada uma. No entanto, a ordem em que inserimos os valores deve estar na mesma ordem em que as colunas são atribuídas na tabela.

SQL
```sql
INSERT INTO table_name (coluna_1, coluna_2, coluna_3, ...)
VALUES (valor_1, valor_2, valor_3, ...);
```

SQL
```sql
INSERT INTO usuarios (id, nome, sobrenome)
VALUES (DEFAULT, 'Alice', 'Oliveira');
```

SQL
```sql
INSERT INTO usuarios
VALUES (DEFAULT, 'Alice', 'Oliveira');
```

Colunas específicas

Para inserir dados em uma coluna específica, declaramos o nome da tabela e depois escrevemos o nome das colunas entre parênteses.

SQL
```sql
INSERT INTO usuarios (nome)
VALUES ('Julia');
```

SQL
```sql
INSERT INTO filmes (duracao, titulo)
VALUES (112, 'Kill Bill');
```

Update

UPDATE modificará os registros existentes em uma tabela. É importante lembrar de usar o **WHERE** na instrução para adicionar a condição de quais linhas queremos alterar. Se não escrevêssemos o WHERE, estaríamos alterando **toda a tabela** e não somente um **registro específico**.

SQL
```
UPDATE nome_tabela
SET coluna_1 = valor_1, coluna_2 = valor_2, ...
WHERE condicao;
```

SQL
```
UPDATE usuarios
SET nome = 'Jane', sobrenome = 'Doe'
WHERE id = 1;
```

Delete

DELETE excluirá os registros existentes em uma tabela. Assim como no **UPDATE**, é importante não esquecer o **WHERE** quando escrevermos a instrução, esclarecendo a condição de quais linhas queremos excluir.

SQL
```
DELETE FROM nome_tabela WHERE condicao;
```

SQL
```
DELETE FROM usuarios WHERE usuario_id = 4;
```

Select

DigitalHouse>

Como usá-lo

Cada consulta ao banco de dados começará com a palavra **SELECT**. Sua funcionalidade é realizar consultas em **uma** ou **mais** colunas de uma tabela.

Para especificar em qual tabela queremos realizar esta consulta, usamos a palavra **FROM** seguida do nome da tabela.

SQL

```sql
SELECT nome_coluna, nome_coluna, ...
FROM nome_tabela;
```

Exemplo

id	titulo	rating	data_estreia	pais
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

Para saber os títulos e classificações dos filmes armazenados na tabela de **filmes**, devemos executar a seguinte consulta:

SQL

```
SELECT titulo, rating
FROM filmes;
```

Where e
Order By

DigitalHouse>

Where

A funcionalidade **WHERE** é condicionar e filtrar as consultas **SELECT** feitas em um banco de dados.

```sql
SELECT nome_coluna_1, nome_coluna_2, ...
FROM nome_tabela
WHERE condicao;
```

Tendo uma tabela de **usuários**, poderíamos consultar o nome e a idade, filtrando com um **WHERE** apenas usuários com **mais de 17 anos** de idade da seguinte maneira:

```sql
SELECT nome, idade
FROM usuarios
WHERE idade > 17;
```

Operadores

=	Igual a
>	Maior que
>=	Maior ou igual que
<	Menor que
<=	Menor ou igual que
<>	Diferente de
!=	Diferente de

Operadores

IS NULL → É nulo

BETWEEN → Entre dois valores

IN → Lista de valores

LIKE → Contém...

Queries de exemplo

SQL

```sql
SELECT nome, idade
FROM usuarios
WHERE idade > 17;
```

SQL

```sql
SELECT *
FROM movies
WHERE title LIKE 'Avatar';
```

Queries de exemplo

SQL

```sql
SELECT *
FROM movies
      WHERE awards >= 3
      AND awards < 8;
```

SQL

```sql
SELECT *
FROM movies
      WHERE awards = 2
      OR awards = 6;
```

Queries de exemplo

SQL

```sql
DELETE FROM usuarios
WHERE id = 2;
```

Se nesta consulta removermos o WHERE...
APAGAREMOS TODA A TABELA!

Order By

ORDER BY é usado para classificar os resultados de uma consulta **com base no valor** da coluna especificada. Por padrão, ele é classificado em ordem crescente (ASC) com base nos valores da coluna. Ele também pode ser classificado em ordem decrescente (DESC), esclarecendo-o na consulta.

SQL

```
SELECT nome_coluna1, nome_coluna2
FROM tabela
WHERE condicao
ORDER BY nome_coluna1;
```

Query de exemplo

Tendo uma tabela de **usuários**, poderíamos consultar os nomes, filtrar com um **WHERE** apenas usuários **maiores de 21 anos** e ordená-los em ordem decrescente, tomando como referência a coluna do nome.

SQL

```sql
SELECT nome, idade
FROM usuarios
WHERE idade > 21;
ORDER BY nome DESC;
```

Between e
Like

DigitalHouse>

Between

Quando precisamos obter valores **dentro de um intervalo**, usamos o operador BETWEEN:

- BETWEEN **inclui** os **extremos**.
- BETWEEN funciona com **números**, **textos** e **datas**.
- Se usa como um **filtro** de WHERE.

Por exemplo, coloquialmente:

- Dados os números: 4, 7, 5, 2, 9, 1

Se aplicássemos um BETWEEN entre 2 e 7, retornaria 4, 7, 5, 2 (exclui 9 e 1 por estarem fora do intervalo, e inclui 2 e 7 por serem os extremos).

Query de exemplo

Com a seguinte consulta, estaríamos selecionando nome e idade da tabela alunos, apenas quando as idades estiverem **entre** 6 e 12.

SQL

```sql
SELECT nome, idade
FROM alunos
WHERE idade BETWEEN 6 AND 12;
```

CURINGA %

É um substituto que representa **zero**, **um** ou **vários** caracteres.

Like

Quando fazemos um filtro com WHERE, podemos estabelecer um padrão de busca que nos permite **especificar** o que estamos procurando nos registros. Conseguimos isso através dos caracteres curinga (wildcards).

Por exemplo, poderíamos buscar:

- ➔ Os nomes que tenham a letra 'a' como segundo caractere;
- ➔ Endereços postais que incluem a palavra 'Paulista';
- ➔ Os clientes que começam com 'os' e terminam com 's'.

CURINGA _

É um substituto para um só caractere.

Queries de exemplo

SQL

```sql
SELECT nome
FROM usuarios
WHERE idade LIKE '_a%';
```

Devolve os nomes dos usuários que tenham a letra “a” como segundo caractere.

SQL

```sql
SELECT nome
FROM usuarios
WHERE endereco LIKE '%Henriqueta%';
```

Devolve os endereços dos usuários que incluem a rua “Henriqueta”.

Queries de exemplo

```sql
SELECT nome
FROM clientes
WHERE nome LIKE 'Os%s';
```

Devolve os nomes dos clientes que começam com “Os” e terminam com “s”.

Limit e Offset

DigitalHouse>

Limit

Sua função é **limitar o número de linhas** (registros/resultados) retornadas nas consultas SELECT.

Também define o **número máximo** de registros a serem excluídos com DELETE.

SQL

```
SELECT nome_coluna1, nome_coluna2
FROM nome_tabela
LIMIT quantidade_de_registros;
```

Query de exemplo

A partir de uma **tabela filmes**, podemos elaborar um **top10** com os filmes que tenham mais de 4 prêmios, usando um LIMIT na seguinte consulta:

SQL

```sql
SELECT *
FROM filmes
WHERE premios > 4
LIMIT 10;
```

Offset

Num cenário em que consultamos todos os filmes do banco de dados, ele retornaria muitos registros. Usando um **LIMIT**, podemos estabelecer um limite de 20 registros.

Como faríamos para exibir 20 filmes, mas pulando os 10 primeiros da tabela?

OFFSET permite especificar a partir de qual linha iniciará a exibição dos dados solicitados.

{código}

```
SELECT id, nome, sobrenome
FROM alunos
LIMIT 20
OFFSET 20;
```

{código}

```
SELECT id, nome, sobrenome
FROM alunos
LIMIT 20
OFFSET 20;
```

Selecionamos as colunas id, nome e sobrenome...

{código}

```
SELECT id, nome, sobrenome
FROM alunos
LIMIT 20
OFFSET 20;
```

...da tabela alunos.

{código}

```
SELECT id, nome, sobrenome
FROM alunos
LIMIT 20
OFFSET 20;
```

Limitamos os registros da tabela resultante a **20** registros.

{código}

```
SELECT id, nome, sobrenome
FROM alunos
LIMIT 20
OFFSET 20;
```

Movemos os resultados 20 posições para mostrar a partir da posição **21**.

Alias

DigitalHouse>

Alias

ALIAS são usados para dar um **nome temporário** e mais amigável a tabelas, colunas e funções. Os alias são definidos durante uma consulta e persistem **apenas** durante essa consulta.

Para definir um alias, usamos as iniciais AS precedendo a coluna que queremos atribuir esse Alias.

SQL

```
SELECT nome_coluna1 AS alias_nome_coluna1
FROM nome_tabela;
```


DigitalHouse>

Alias para uma coluna

```
SELECT razao_social_cliente AS nome
FROM cliente
WHERE nome LIKE 'a%';
```

Alias para uma coluna

```
SELECT razao_social_cliente AS nome
FROM cliente
WHERE nome LIKE 'a%';
```

Selecionamos a **coluna** *razao_social_cliente* e atribuímos o **ALIAS** *nome*.

DigitalHouse>

Alias para uma coluna

```
SELECT razao_social_cliente AS nome
FROM cliente
WHERE nome LIKE 'a%';
```

No **FROM**, escolhemos a tabela *cliente*.
Com o **WHERE**, filtramos os registros da coluna apelidada *nome* que começam com a letra “a”.

DigitalHouse>

Alias para uma tabela

```
SELECT nome, sobrenome, idade
FROM alunos_comissao_inicial AS alunos;
```

Alias para uma tabela

```
SELECT nome, sobrenome, idade
FROM alunos_comissao_inicial AS alunos;
```

Selecionamos as **colunas** *nome*, *sobrenome* e *idade*.

Alias para uma tabela

```
SELECT nome, sobrenome, idade
FROM alunos_comissao_inicial AS alunos;
```

Fizemos a consulta sobre a tabela:

alunos_comissao_inicial

e atribuímos o **ALIAS** *alunos*.

DigitalHouse>

Alias para uma tabela

Para atribuir um alias com espaço, é necessário escrevê-lo entre aspas simples:

SQL

```
SELECT nome, sobrenome, idade
FROM alunos_comissao_inicial AS ’os alunos’;
```

“Desse modo, podemos criar um alias das **colunas** e **tabelas** que estamos fazendo, e tornar a manipulação de dados mais **legível**. Lembrando sempre que os alias **não modificam** os nomes originais no banco de dados.”

Funções MySQL

DigitalHouse>

Concat

Usamos **CONCAT** para **concatenar** duas ou mais expressões:

SQL
```
SELECT CONCAT('Olá ', 'a ', 'todos.');
```

> 'Olá a todos.'

SQL
```
SELECT CONCAT('A resposta é: ', 24, '.');
```

> 'A resposta é 24.'

SQL
```
SELECT CONCAT('Nome: ', first_name, ' ', last_name)
FROM actors;
```

> 'Nome: Emilia Clarke'

Coalesce

Usamos **COALESCE** para obter a **primeira expressão** que **não seja NULL**:

```sql
SELECT COALESCE(NULL, 1, 20, 'Digital House');
```

```
> 1
```

```sql
SELECT COALESCE(NULL, NULL, 'Digital House');
```

```
> 'Digital House'
```

Coalesce

Os três clientes da seguinte tabela possuem um ou mais dados nulos:

CLIENTES				
id	**nome**	**celular**	**casa**	**trabalho**
1	Luana	124	345	980
2	Lucas		187	243
3	Maria			428

Coalesce

Usando **COALESCE** podemos obter o **primeiro dado não nulo** de cada registro, declarando as colunas a ter em conta.

SQL

```sql
SELECT id, nome, COALESCE(celular, casa, trabalho) AS telefone
FROM clientes;
```

CLIENTES		
id	**nome**	**telefone**
1	Luana	124
2	Lucas	187
3	Maria	428

Datediff

Usamos **DATEDIFF** para calcular a **diferença** entre duas datas, esclarecendo se queremos o resultado em horas, minutos, etc...

```sql
SELECT DATEDIFF(hour, '2017/08/25 07:00', '2017/08/25 12:45');
```

```
> 5
```

Devolve 5 porque é a quantidade de horas de diferença entre 7:00 e 12:45.

Essa unidade dá um resultado aproximado.

```sql
SELECT DATEDIFF(minute, '2017/08/25 07:00', '2017/08/25 12:45');
```

```
> 345
```

Devolve 345 porque é a quantidade de minutos de diferença entre 7:00 e 12:45 (300min + 45min).

Extract

Usamos **EXTRACT** para **extrair** partes de uma data:

```sql
SELECT EXTRACT(SECOND FROM '2014-02-13 08:44:21');
```

```
> 21
```

```sql
SELECT EXTRACT(MINUTE FROM '2014-02-13 08:44:21');
```

```
> 44
```

```sql
SELECT EXTRACT(HOUR FROM '2014-02-13 08:44:21');
```

```
> 8
```

```sql
SELECT EXTRACT(DAY FROM '2014-02-13 08:44:21');
```

```
> 13
```

Extract

```sql
SELECT EXTRACT(WEEK FROM '2014-02-13 08:44:21');
```

> 6

```sql
SELECT EXTRACT(MONTH FROM '2014-02-13 08:44:21');
```

> 2

```sql
SELECT EXTRACT(QUARTER FROM '2014-02-13 08:44:21');
```

> 1

```sql
SELECT EXTRACT(YEAR FROM '2014-02-13 08:44:21');
```

> 2014

Replace

Usamos **REPLACE** para substituir uma sequência de caracteres por outra em uma string.

SQL
```sql
SELECT REPLACE('abc abc', 'a', 'B');
```
```
> Bbc Bbc
```

SQL
```sql
SELECT REPLACE('abc abc', 'A', 'B');
```
```
> abc abc
```

SQL
```sql
SELECT REPLACE('123 123', '2', '5');
```
```
> 153 153
```

Date format

Usamos **DATE_FORMAT** para aplicar um estilo de formatação em datas, especificando como nós queremos que ela seja exibida.

SQL
```sql
SELECT DATE_FORMAT('2017-06-15', '%Y');
```
```
> '2017'
```

SQL
```sql
SELECT DATE_FORMAT('2017-06-15', '%W %M %e %Y');
```
```
> 'Thursday June 15 2017'
```

Case

Usamos CASE para avaliar condições e devolvemos a primeira que for cumprida. Neste exemplo, a tabela resultante terá 4 colunas: *id, titulo, avaliacao, avaliacao_categorias*. Esta última mostrará 'Ruim', 'Regular', etc, **de acordo** com a **classificação** do filme.

SQL

```
SELECT id, titulo, avaliacao
  CASE
    WHEN avaliacao < 4 THEN 'Ruim'
    WHEN avaliacao < 6 THEN 'Regular'
    WHEN avaliacao < 8 THEN 'Bom'
    WHEN avaliacao < 9.5 THEN 'Muito bom'
    ELSE 'Excelente'
  END AS avaliacao_categorias
FROM filmes
ORDER BY avaliacao
```

Full Stack Node.js

Funções de agregação e consultas avançadas

DigitalHouse>

O que vimos no Playground

- Joins
- Distincts
- Group By
- Funções de Agregação
- Having

O que vamos ver hoje

- Otimizar consultas por meio de agregação de dados
- Otimizar consultas por meio do cruzamento de tabelas
- Importância do uso de Primary e Foreign Key

Com o andamento das aulas, quais outros pontos chamaram atenção sobre o uso de primary e foreign key?

Importante saber:

Há 3 tipos de JOIN, cada um trará diferentes registros da base de dados.

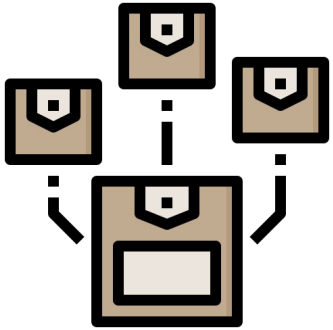
Quais são eles?

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN

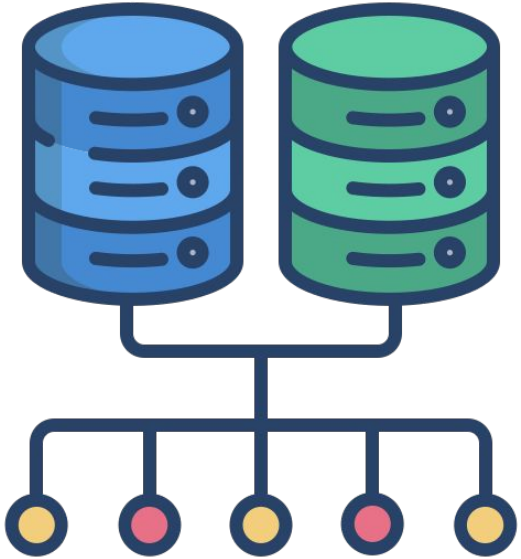
Importante saber:

Qual a característica de cada um?

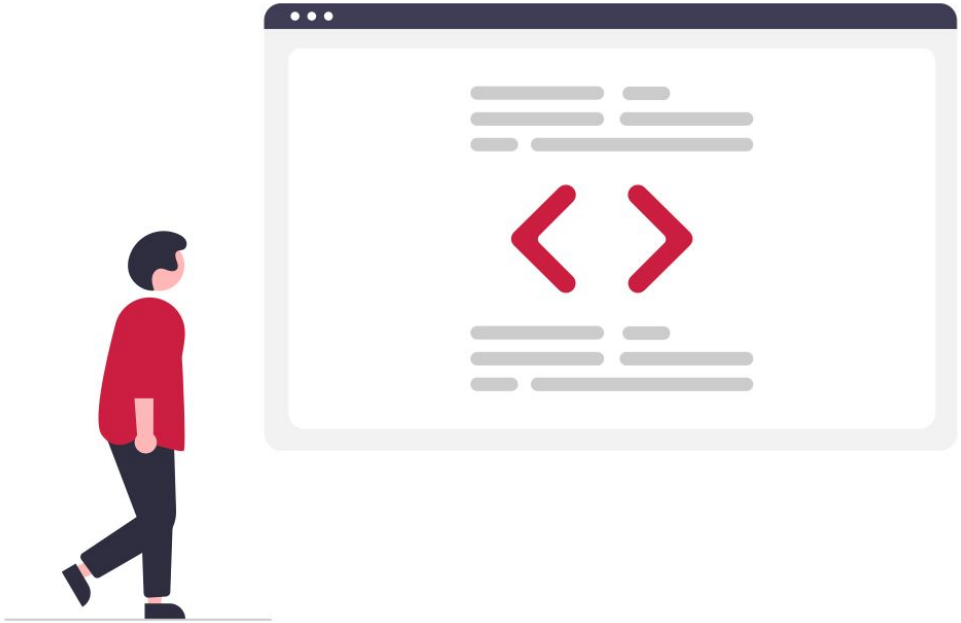
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN



Um caso de uso de JOIN



Agora vamos botar a mão na massa!



Atividade I - Atravessando tabelas

- Continuação prática escola_dh
- Execute o arquivo ***atualizacoes_escola_db.sql***
- Faça a busca na base de dados seguindo a orientação do documento Atividade I

AS BUSCAS COM GROUP BY E FUNÇÕES AGREGADORAS

Importante validarmos:

- Group by tem seu funcionamento dependente do uso em conjunto de funções agregadoras?
- O que pode ser utilizado nos casos sem funções agregadoras?
- Existe alguma limitação para o uso do group by (utilizar join, where, alias) ?

Um caso de uso de DISTRICT

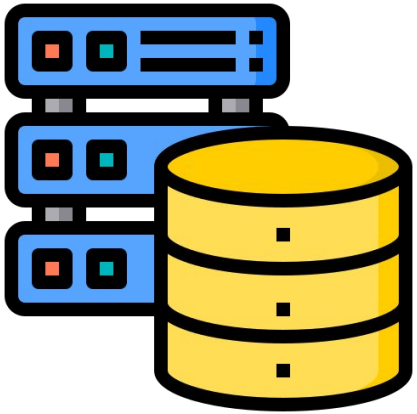
Vamos te desafiar um pouco mais!

Atividade II - Buscas mais refinadas

- Faça a busca na base de dados seguindo a orientação do documento Atividade II

Para ir além

- **Documentação na W3Schools Join's:**
  https://www.w3schools.com/sql/sql_join.asp



Full Stack Node.js

Funções de agregação e consultas avançadas

DigitalHouse>

O que vimos no Playground

- Joins
- Distincts
- Group By
- Funções de Agregação
- Having

O que vamos ver hoje

- Otimizar consultas por meio de agregação de dados
- Otimizar consultas por meio do cruzamento de tabelas
- Importância do uso de Primary e Foreign Key

Com o andamento das aulas, quais outros pontos chamaram atenção sobre o uso de primary e foreign key?

Importante saber:

Há 3 tipos de JOIN, cada um trará diferentes registros da base de dados.

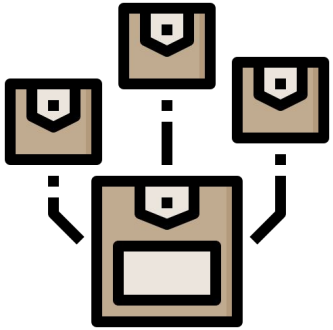
Quais são eles?

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN

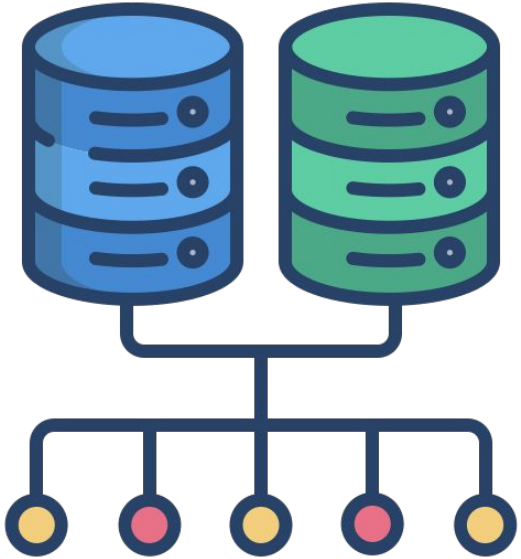
Importante saber:

Qual a característica de cada um?

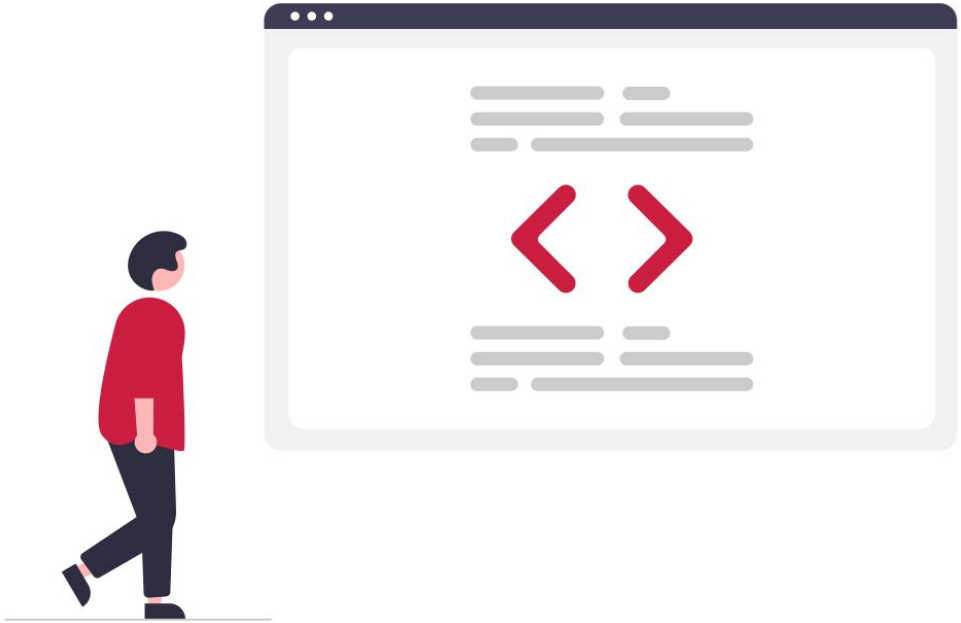
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN



Um caso de uso de JOIN



Agora vamos botar a mão na massa!



Atividade I - Atravessando tabelas

- Continuação prática escola_dh
- Execute o arquivo ***atualizacoes_escola_db.sql***
- Faça a busca na base de dados seguindo a orientação do documento Atividade I

AS BUSCAS COM GROUP BY E FUNÇÕES AGREGADORAS

Importante validarmos:

- Group by tem seu funcionamento dependente do uso em conjunto de funções agregadoras?
- O que pode ser utilizado nos casos sem funções agregadoras?
- Existe alguma limitação para o uso do group by (utilizar join, where, alias) ?

Um caso de uso de DISTRICT

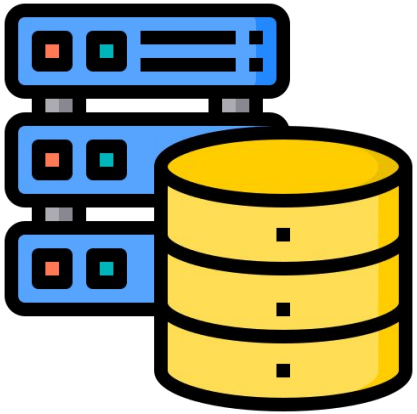
Vamos te desafiar um pouco mais!

Atividade II - Buscas mais refinadas

- Faça a busca na base de dados seguindo a orientação do documento Atividade II

Para ir além

- **Documentação na W3Schools Join's:**
  https://www.w3schools.com/sql/sql_join.asp



JOINS

DigitalHouse>

Por que usar joins?

Além de fazer consultas dentro de uma tabela, às vezes é necessário fazer consultas a **tabelas diferentes,** e unir esses resultados com **JOINS.**

Os **JOINS**, além de outras coisas:

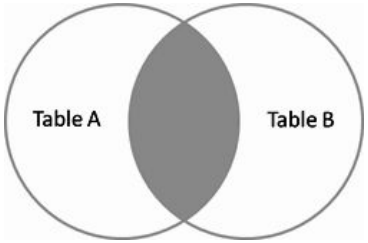
- São mais flexíveis,
- Sua sintaxe é muito mais utilizada,
- Apresentam uma melhor performance.

Inner join

O **INNER JOIN** fará um **cruzamento** entre duas tabelas. Se cruzarmos as tabelas de **clientes** e **vendas** e houver algum cliente **sem vendas**, o INNER JOIN **não traria** esse cliente como resultado.

INNER JOIN

CLIENTES		
id	**nome**	**sobrenome**
1	João	Silva
2	Clara	Sanches
3	Marta	Garcia

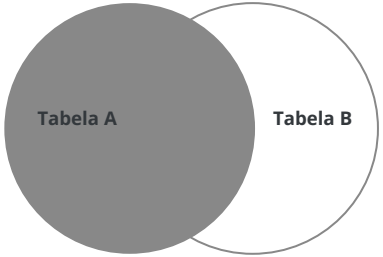


VENDAS		
id	**cliente_id**	**data**
1	2	12/03/2019
2	2	22/08/2019
3	1	04/09/2019

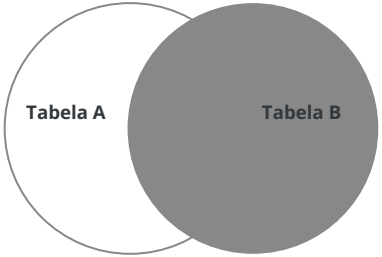
Left join - Right join

Estes tipos de JOINS **não excluem** resultados de alguma das duas tabelas. Se houvesse clientes **sem vendas**, poderíamos incluí-los no resultado usando o **LEFT** ou **RIGHT** JOIN.

LEFT JOIN



RIGHT JOIN



Criando um inner join

Antes *escrevíamos*:

```sql
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes, vendas
```

Agora *escreveremos*:

```sql
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes
INNER JOIN vendas
```

“Embora já tenhamos dado o primeiro passo, que é **cruzar** as duas tabelas, ainda precisamos esclarecer **onde** está esse cruzamento.

Ou seja, qual **chave primária (PK)** irá cruzar com qual **chave estrangeira (FK)**.”

Criando um inner join *(cont.)*

A sintaxe do join **não utiliza** o **WHERE**, mas sim **precisa** da palavra **ON**. É ali onde indicaremos o **filtro** a ser considerado para realizar o cruzamento.

Ou seja, o que antes escrevíamos no **WHERE,** agora escreveremos com **ON**.

SQL

```sql
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes
INNER JOIN vendas
ON clientes.id = vendas.cliente_id
```

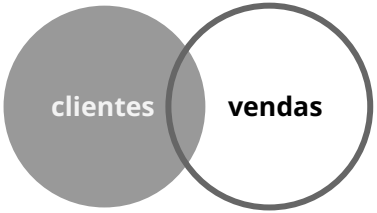
“E se quisermos **incluir** no resultado aqueles **clientes** que **NÃO** tenham **vendas** associadas?”

Criando um left join

Para incluir aqueles clientes sem vendas, basta mudar **INNER** JOIN para **LEFT** JOIN. O **LEFT JOIN** incluirá **todos** os registros da primeira tabela da consulta *(a tabela **esquerda**)*, incluindo quando não exista coincidência com a tabela da direita.

SQL

```sql
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes
LEFT JOIN vendas
ON clientes.id = vendas.cliente_id
```



DigitalHouse>

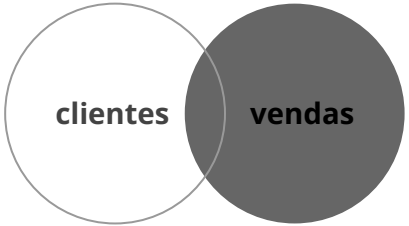
“E para **incluir** no resultado aquelas **vendas** que **NÃO** têm **clientes** associados?”

Criando um right join

Para incluir aquelas vendas sem clientes, basta mudar **LEFT** JOIN por **RIGHT** JOIN. O **RIGHT JOIN** incluirá **todos** os registros da tabela **direita**. *Se observamos a query, a tabela vendas aparece depois da tabela de clientes... à direita!*

SQL

```
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes
RIGHT JOIN vendas
ON clientes.id = vendas.cliente_id
```



Cruzando muitas tabelas

No exemplo a seguir, é possível ver como fazer cruzamentos de muitas tabelas em uma mesma consulta usando **joins**:

SQL

```sql
SELECT clientes.id AS id, clientes.nome, vendas.data
FROM clientes
INNER JOIN vendas
ON clientes.id = vendas.cliente_id
INNER JOIN produtos
ON produtos.id = vendas.produto_id
```

DISTINCT

DigitalHouse>

Como funciona

A cláusula **DISTINCT** retorna **valores únicos**. Em uma tabela, uma coluna pode conter valores duplicados e às vezes apenas é necessária uma lista com os diferentes valores, ou seja, aqueles que se repetem não aparecem.

SQL

```sql
SELECT DISTINCT coluna_1, coluna_2
FROM nome_tabla;
```

Distinct exemplo

A partir de uma tabela de **usuários**, se executarmos a consulta:

```sql
SELECT pais FROM usuarios;
```

Obteríamos:

'Perú, Perú, Argentina, França, Argentina'

Existem cenários em que precisaremos obter apenas os valores distintos que aparecem em uma coluna. Adicionando o **DISTINCT** na consulta:

```sql
SELECT DISTINCT pais FROM usuarios;
```

Obteríamos:

'Perú, Argentina, França'

DigitalHouse>

{código}

```
SELECT DISTINCT actors.first_name, actors.last_name
FROM actors
INNER JOIN actor_movie ON actors.id =
actor_movie.actor_id
INNER JOIN movies ON movies.id = actor_movie.movie_id
WHERE movies.title LIKE '%Harry Potter%';
```

Neste exemplo, vemos uma **query** que solicita os **atores** que atuaram em qualquer **filme de Harry Potter**.
Se não escrevêssemos ao DISTINCT os atores que participaram de mais de um filme, eles **apareceriam repetidos** no resultado.

GROUP BY

DigitalHouse>

Group By sintaxe

GROUP BY é usado para **agrupar os registros** da tabela resultantes de uma consulta por uma ou mais colunas.

SQL

```sql
SELECT coluna_1,
FROM nome_tabela
WHERE condicao
GROUP BY coluna_1;
```

Group By exemplo

```sql
SELECT marca,
FROM carros
GROUP BY marca;
```

Group By exemplo

id	marca	modelo
1	Renault	Clio
2	Renault	Megane
3	Seat	Ibiza
4	Seat	Leon
5	Opel	Corsa
6	Renault	Clio

marca
Renault
Seat
Opel

DigitalHouse>

Group By sintaxe

Como **GROUP BY** agrupa as informações, perdemos o detalhe de cada uma das linhas. Em outras palavras, não estamos mais interessados no valor de cada linha, mas em um resultado consolidado entre todas as linhas.

Consulta:

```sql
SELECT id, marca
FROM autos
GROUP BY marca;
```

Isso nos daria um erro. Se agruparmos os dados por **marca**, não podemos mais solicitar o campo **id**.

Group By sintaxe

- Portanto, ao usar **GROUP BY**, nos campos que aparecem como resultado no **SELECT**, podemos apenas indicar:
- Dados agrupados
- Funções de agregação

Vejamos alguns exemplos...

DigitalHouse>

Group By sintaxe

SQL

```sql
SELECT marca, MAX(preco)
FROM carros
GROUP BY marca;
```

SQL

```sql
SELECT genero.nome, AVG(duracao)
FROM filmes
INNER JOIN generos ON generos.id = genero_id
GROUP BY genero.nome;
```

Funções de agregação

DigitalHouse>

“As funções de agregação **realizam cálculos** sobre um conjunto de dados e retornam um **único resultado.** Exceto **COUNT**, as funções de agregação ignoram valores **NULL**.”

COUNT

Retornará a quantidade de **linhas/registros** que atendem aos critérios.

SQL
```sql
SELECT COUNT(*) FROM filmes;
```

Devolverá a quantidade de registros da tabela "filmes".

SQL
```sql
SELECT COUNT(id) AS total FROM filmes WHERE genero_id = 3;
```

Devolverá a quantidade de filmes da tabela "filmes" com o genero_id 3 em uma coluna chamada "total".

AVG, SUM

AVG *(average)* retornará a média de uma coluna com valores numéricos.

SUM retornará a soma de uma coluna com valores numéricos.

SQL
```sql
SELECT AVG(rating) FROM filmes;
```

Retornará a média do rating dos filmes da tabela "filmes".

SQL
```sql
SELECT SUM(length) FROM filmes;
```

Retornará a soma das durações dos filmes da tabela "filmes".

MIN, MAX

MIN retornará o valor mínimo de uma coluna com valores numéricos.

MAX retornará o valor máximo de uma coluna.

```sql
SELECT MIN(rating) FROM filmes;
```

Retornará o rating do filme que foi menos ranqueado.

```sql
SELECT MAX(rating) FROM filmes;
```

Retornará o rating do filme que foi melhor ranqueado.

HAVING

DigitalHouse>

HAVING

Ele cumpre a mesma função de **WHERE**, a diferença é que **HAVING** pode ser usado em conjunto com **funções de agregação** para filtrar **dados agregados**.

SQL

```sql
SELECT coluna
FROM tabela
WHERE condicao
GROUP BY coluna
HAVING condicao
ORDER BY coluna;
```

HAVING

Esta consulta retornará o número de clientes por país (agrupados por país). Apenas os países com **pelo menos** 3 clientes serão incluídos no resultado.

SQL

```sql
SELECT COUNT(cliente_id), pais
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId) > 3;
```

Estrutura de uma query

```
SELECT [Lista de seleção]
FROM   [Origem dos dados]
WHERE  [Condição de seleção]
HAVING  [Condição de seleção]
ORDER BY  [Colunas de ordenação]
;
```


DigitalHouse>

Sequelize

Introdução, modelos e consultas básicas

DigitalHouse>

Índice

1. Promises
2. Inicialização e configuração
3. Modelos
4. findAll(), findOne, findByPk
5. Where e operadores
6. Order, limit y offset

Objetivos

- Revisar os temas da classe através de perguntas.
- Analisar de forma prática e teórica o uso do modelo no modelo de projeto MVC.
- Configurar um projeto com um banco de dados usando Sequelize.
- Realizar exercícios para praticar a recuperação de dados usando o Sequelize.

“

E se lhes dissermos que eles podem fazer tudo o que aprenderam no MySQL, a partir do JavaScript?

”

Pacote Sequelize

Sequelize será o pacote que nos permitirá interagir com o banco de dados a partir do JavaScript.

Agora, vamos fazer uma breve revisão...

1 Promises

Promises

As **promises** são funções que permitem que o código **assíncrono** seja executado de modo eficiente.

.then()

```
getUsers()
        .then(function(data){
        return filterData(data);
    })
```

.catch()

```
getUsers()
        .then(function(data){
        console.log(data);
    })
        .catch(function(error){
        console.log(error);
    })
```

Promise.all()

Um método que permite a execução do código uma vez cumpridas duas ou mais promises. Promise.all() toma como parâmetro uma série de promises.

```
{}
let promiseMovies = getMovies();
let promiseGeneros = getGeneros();

Promise.all([promiseMovies, promiseGeneros])
```

2 Instalação

Sequelize
Instalação e configuração

1.

```
>_ npm instalar sequelize-cli -g
   npm instalar sequelize
   npm instalar mysql2
```

2. Configuração **.sequelizerc**

3.

```
>_ sequelizar o init
```

4. Configuração **config.js**

3 | Modelos

Sintaxe

Um modelo é a representação de nossa tabela no código.

- Devem ser escritos com a extensão **.js**, dentro da pasta **/database/models.**
- O nome do arquivo é escrito em **UpperCamelCase** e **singular.**

.define()

Cols

Definição do tipo de dado e caracteres visíveis por linha na tabela do banco de dados.

```
const Movies = sequelize.define(alias, cols, config);
return Movies;
```

Alias

Nome de identificação do modelo.

Config

Configurações adicionais.

4 | findAll(), findOne, findByPk()

Selecione

findAll()

Para pesquisar todos os dados registrados na tabela, devemos usar o findAll()

findOne()

findOne() nos permite buscar resultados que correspondem aos atributos especificados no objeto literal recebido pelo método.

findByPk()

O método findByPk() busca um registro com a chave primária do mesmo valor que o parâmetro passado.

5 Where e operadores

Where e operadores

Where

Para acrescentar uma condição à consulta, basta passar o atributo **where** ao método findAll():

Like

Para acrescentar condições à busca **Where,** usamos operadores embutidos do sequelize. Um deles é o **Like**.

6 Order, limit e offset

Order, limit e offset

Na sequência, para ordenar o resultado você simplesmente usa o atributo **order**, que recebe uma array.

```
db.User.findAll({ {
   order: [
      ['nome', 'ASC'],
   ],
});
```

Order, limit e offset

Para limitar o número de resultados, basta adicionar o atributo **limit** ao objeto e passá-lo para **findAll()**:

```
db.User.findAll({ {
   limit: 10
   })
      .then((results) => {
   console.log(results)
})
```

Order, limit e offset

Para omitir múltiplos elementos de resultado, simplesmente adicione o atributo **offset** ao objeto e passe-o para **findAll()**:

```
{}
db.User.findAll({ {
   offset: 10
   })
      .then((results) => {
   console.log(results)
})
```

Async/await

DigitalHouse >
Coding School

Índice

1. then() vs async/await
2. Observando os detalhes

then() vs async/await

As promises vieram de início para evitar o chamado **callback hell**. Entretanto, logo se percebeu que, apesar de ajudar bastante um código com alta quantidade de **then()** encadeado, se tornava difícil de ler e sua manutenção não era das melhores.

Para resolver essa situação, foi criado no **ES8** (ou ES2017) as funções **async**. No próximo slide, vamos comparar as duas sintaxes.

then() vs async/await

Aqui temos uma comparação da nossa promise sendo resolvida com **async function** e da forma **tradicional**.

```
const resolvePromise = async() => {
    try{
        const resultado = await minhaPromise
        console.log(resultado)
    }
    catch(err) {
        console.log(err)
    }
}
resolvePromise()
```

```
minhaPromise
    .then(res => console.log(res))
    .catch(err => console.log(err))
```

then() vs async/await

Apesar de parecer **mais verboso**, o formato com **async function** é mais fácil de se manter, uma vez que consegue resolver promises **sem** necessidade de **concatenações**, tornando o código mais limpo.

Observando os detalhes

Agora vamos dissecar a async function para entender melhor a lógica!

```
const resolvePromise = async() => {
    try{
        const resultado = await minhaPromise
        console.log(resultado)
    }
    catch(err) {
        console.log(err)
    }
}
resolvePromise()
```

- O código deve estar dentro de uma função marcada como **async**.
- O bloco de fluxo normal deve estar envolto em um **try**.
- A chamada à promise deve estar marcada como **await**.
- Todo bloco try precisa de um bloco **catch**.
- Como o código está em uma função, você deve chamar a mesma para que o código seja executado.

Criando Promises

DigitalHouse >
Coding School

Índice

1. Estados da Promise
2. Anatomia de uma Promise
3. Consumindo suas Promises

Estados das Promises

As promises podem ter 3 diferentes estados: Pending (pendente), Fulfilled (cumprida), Rejected (rejeitada).

Pending: É o estado de nascimento de uma promise, significa dizer que ela ainda não obteve retorno sobre a operação pretendida, portanto não sabe-se dizer se foi bem ou mal sucedida.

Estados das Promises

Fulfilled: Significa que a promessa foi cumprida, ou seja, a operação prometida ocorreu com sucesso e retornou o esperado.

Rejected: Quer dizer que ocorreu algum erro no decorrer da promessa e a operação relacionada a ela falhou.

Anatomia da Promise

Uma promisse recebe **uma função** que recebe 2 parâmetros: **resolve** e **reject**.

```
{}   const minhaPromise = new Promise((resolve, reject) => {})
```

Anatomia da Promise

O **resolve** deve ser chamado para finalizar a promise de forma bem sucedida, enquanto o **reject** deve servir para quando os pré-requisitos da promise não forem cumpridos.

```
{}
const minhaPromise = new Promise((resolve, reject) => {
    if(1) {
        resolve("A promise foi bem sucedida!")
    }
    reject("Ocorreu um erro!")
})
```

Anatomia da Promise

Também podemos passar **argumentos** para as promises, basta fazer como está no exemplo:

```
{}
const promiseComArgs = (argumento) => new Promise((resolve, reject) => {
    if(argumento > 1) resolve("O argumento é válido")
    reject("argumento inválido")
})
```

Consumindo suas Promises .then()

A função assíncrona retornará um resultado ou não. Enquanto isso, o código **continua em execução**.

```
obterUsuarios()
    .then(function(data){
        console.log(data);
    });

console.log("Continua a executar")
```

Função assíncrona.

Código que pode continuar a ser executado enquanto a promise está em execução.

Promises .then()

A função assíncrona retornará um resultado ou não. Enquanto isso, o código **continua em execução**.

```
{}
obterUsuarios()
    .then(function(data){
        console.log(data);
    });
console.log("Continua a executar!")
```

Execute o console.log() SOMENTE SE **obterUsuarios()** retornar um resultado. Este é recebido por **.then()** dentro de seu retorno de chamada, neste caso no parâmetro **data**.

Promises - Requisições Aninhadas

Às vezes, os **.then()** podem ter promises internas. Para resolver isso, precisamos utilizar outro **.then()** que entre em execução uma vez que se resolva o anterior.

```
obterUsuarios()
    .then(function(data){
        return filtrarDados(data);
    })
    .then(function(filteredData){
        console.log(filteredData);
    })
```

ATENÇÃO!

É importante lembrar que o **.then()** precisa retornar os dados processados para que possam ser usados por outro **.then()**.

Promises .catch()

Caso **NÃO** obtenha resultado, é gerado um erro. Para isso, usamos **.catch()**, que encapsula quaisquer erros que possam ser gerados por meio de promessas. Dentro desse método, decidimos o que fazer com o erro. Ele é recebido como um parâmetro no callback **.catch ()**. No exemplo a seguir, mostraremos o erro no console:

```
{}
obterUsuarios()
    .then(function(data){
        console.log(data);
    })
    .catch(function(error){
        console.log(error);
    })
```

ATENÇÃO!

É importante lembrar que toda promise precisa de um **.catch()** para tratamento de exceções!

Promises - Promise.all()

Promises em conjunto

Às vezes, precisamos que duas ou mais promessas sejam resolvidas para realizar uma determinada ação. Para isso, usamos Promise.all(). Isso conterá uma série de promessas de que, uma vez resolvidas, um .then() será executado com os resultados da mesma.

Promises - Promise.all()

O que devemos fazer primeiro é salvar em variáveis as promessas que precisamos obter.

```
let promiseFilmes = obterFilmes();

let promiseGeneros = obterGeneros();
```

Promise de filmes

Promise de gêneros

Promises - Promise.all()

A próxima etapa é usar o método **Promise.all()** que conterá um array com as promises que salvamos anteriormente.

```
{}    Promise.all([promiseFilmes, promiseGeneros])
```

Adicionando promises no array.

Promises - Promise.all()

A callback .then () recebe uma matriz com os resultados das promessas cumpridas.

```
Promise.all([promiseFilmes, promiseGeneros])
```

```
.then(function([resultadoFilmes,resultadoGeneros]){
  console.log(resultadoFilmes, resultadoGeneros);
})
```

O **.then()** somente executará se ambas as promises se cumprirem.

Promises - Documentação

Para saber mais sobre uso de promises e Promise.all (), você pode acessar a documentação oficial do Mozilla, clicando nos seguintes links:

- Promise.all()
- Uso de promises

Promises

DigitalHouse>

Índice

1. Requisições assíncronas
2. .then()
3. Cadeia de Promises
4. Promise.all()

“

As **promises** são funções que permitem que o código **assíncrono** seja executado eficientemente.

”

1 Requisição assíncrona

Requisição assíncrona

Uma solicitação assíncrona é um conjunto de instruções executadas por um mecanismo específico, como **callback**, **promise** ou um **evento**. Isto possibilita que a resposta seja processada em outro momento.

Como se pode inferir, seu comportamento é **não-bloqueante,** pois o pedido é executado em paralelo com o resto do código.

2 | .then()

.then()

A função assíncrona retornará um resultado, ou não. Enquanto isso, o código **continua a ser executado**.

```
getUsers()
    .then(function(data){
      console.log(data);

console.log("Ainda funcionando!")
```

Função assíncrona.

Código que poderia continuar a ser executado enquanto a promessa está sendo cumprida.

DigitalHouse>

.then()

Quando a promise é cumprida e um resultado é obtido, o primeiro **.then() é executado** e age somente como uma consequência da função assíncrona.

```
{}
getUsers()
      .then(function(data){
        console.log(data);
      });
console. log("Ainda funcionando!")
```

Execute o console.log() SOMENTE SE **getUsers()** retorna um resultado. Isto é recebido por **.then()** dentro de seu callback, neste caso, no parâmetro de **dados**.

3 | Cadeia de Promises

Cadeia de Promises

Às vezes, `()` têm muitas promises dentro delas. Para resolver isto, precisamos usar outro `.then()` que entre em execução uma vez que o anterior seja resolvido.

```
{}
getUsers()
        .then(function(data){
         return filterData(data);
    })
        .then(function(dataFiltered){
        console.log(dataFiltered);
    })
```

Cuidado!

É importante lembrar que o **.then()** precisa devolver os dados processados para que possam ser utilizados por outro **.then()**.

.catch()

No caso de **NÃO** se obter um resultado, é gerado um erro. Para isso, utilizamos `.catch()`, que captura quaisquer erros que possam ser gerados através das promises. Dentro deste método, nós decidimos o que fazer com o erro. O erro é recebido como um parâmetro dentro da callback de `.catch()`.

No exemplo a seguir, mostraremos o erro no console:

```
{}
getUsers()
      .then(function(data){
       console.log(data);
      })
      .catch(function(error){
       console.log(error);
     })
```

4 | Promise.all()

Promise.all( )

Às vezes, precisamos de duas ou mais promises para realizar uma determinada ação. Para isso, usamos o `Promise.all()`. Isto conterá uma série de promises que, uma vez resolvidas, executarão um `.then()` com os resultados dessas promises.

A primeira coisa que precisamos fazer é armazenar em variáveis as promises que precisamos obter.

```
{}
let promiseMovies = getMovies();

let promiseGenres = getGenres();
```

Promessa de filmes

Promessa de gênero

Promise.all()

O próximo passo é usar o método `Promise.all()` que conterá uma série das promises que salvamos anteriormente.

```
Promessa.all([promiseMovies, promiseGenres])
```

promises a serem resolvidas

{}

Promise.all()

O callback de `.then()` recebe um conjunto de resultados das promises cumpridas.

```
Promessa.all([promiseMovies, promiseGenres])
```

```
.then(function([resultFilms,resultGenres]){
  console.log(resultFilms, resultGenres);
})
```

O **.then()** só será executado se ambas as promises forem cumpridas.

DigitalHouse>

Documentação

Para saber mais sobre promises e Promise.all(), você pode acessar a documentação oficial do Mozilla clicando nos links a seguir:

Uso de promises

Promise.all()

Sequelize e sua configuração

DigitalHouse>

Índice

1. Instalação
2. Rotas e diretórios
3. Configuração
4. Objeto DB

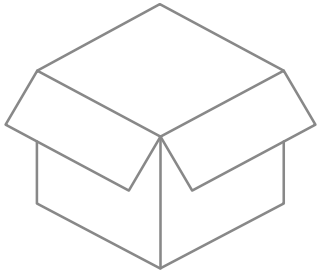
“

Sequelize é um ORM que nos ajuda a conectar e interagir com bancos de dados como Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server e muito mais.

”

Preparando o projeto

Antes de instalar **o Sequelize**, devemos considerar que, como se trata de um pacote utilizado pelo **Node.js,** teremos que utilizar **npm**.



1 Instalação

Instalação

Dentro da pasta do projeto Node.js, execute os seguintes comandos:

```
>_ npm install sequelize-cli -g

   npm install sequelize

   npm install mysql2
```

2 | Rotas e diretórios

Rotas e diretórios

Uma vez instalados os pacotes de que precisamos, é preciso estabelecer as rotas/caminhos e diretórios. Para fazer isso, precisamos criar um arquivo chamado `.sequelizerc` na raiz do projeto, e dentro dele, escrever o seguinte:

```
{}
const path = require('path')

module.exports = {
 config: path.resolve('./database/config', 'config.js'),
 'models-path': path.resolve('./database/models'),
 'seeders-path': path.resolve('./database/seeders'),
 'migrations-path': path.resolve('./database/migrations'),
}
```

Inicie o Sequelize no projeto

Para que o Sequelize possa criar todas as pastas e arquivos necessários para começar a trabalhar, precisamos executar o seguinte comando:

```
>_ Sequelize o init
```

3 | Configuração

Configuração

Finalmente, precisamos configurar a conexão com o banco de dados. Nas pastas criadas pela Sequelize, encontraremos o arquivo **config.js no** caminho **/database/config/config.js.** Dentro dele, encontramos um JSON com credenciais padrão que devemos substituir por nossas próprias credenciais.

```
{}
"development": {
    "username": "root",
    "password": "Monito123!",
    "database": "movies_db",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  }
```

ATENÇÃO!

Precisamos acrescentar um detalhe para não nos depararmos com um problema ao exigir o Sequelize. Para fazer isso, no arquivo config.js que editamos anteriormente, precisamos atribuir todo o JSON que modificamos ao **module.exports**.

JS *config.js*

database > config > JS config.js > [ ] <unknown>

```
1  module.exports = {
2    "development": {
3      "username": "root",
4      "password": "Monito123!",
5      "database": "movies_db",
6      "host": "127.0.0.1",
7      "dialect": "mysql",
8      "operatorsAliases": false
9    },
```

4 | Objeto DB

Objeto DB

Dizemos que no final do arquivo **index.js**, localizado em **/database/modelos/index.js**, encontramos a exportação do objeto **DB**.

Esta é a que chamaremos sempre que quisermos usar a Sequelize para consultar nosso banco de dados.

Documentação

Para saber mais, você pode acessar a documentação oficial de Sequelize clicando no link a seguir:

https://sequelize.org/

Sequelize Modelos

DigitalHouse>

Índice

1. Criação de um Modelo
2. Método .define()
3. Tipos de dados
4. TimeStamps
5. Configurações Adicionais

“Ao fazer uma consulta ao banco de dados, um **modelo** retornará as informações em um formato útil e confortável e então operará com elas.”

Sequelize - Models

Em padrões de projeto MVC (Model - View - Controller), os **modelos** contêm apenas os dados puros de aplicação; não há lógica que descreva como os dados podem ser apresentados a um usuário. Isso pode acessar a camada de armazenamento de dados. Idealmente, o modelo deve ser independente do sistema de armazenamento.

Sequelize - Models

Quer dizer...

Um **modelo** é a representação de nossa tabela em código. Com isso, obtemos recursos que nos permitem fazer consultas e interações com o banco de dados de forma simplificada, utilizando, neste caso, o sequelize.

Models - Criando um model

Sempre criamos um modelo para cada tabela em nosso banco de dados. O caminho onde o armazenamos é **/database/models.**

Os modelos são arquivos JS, portanto, devem ser criados com essa extensão.

Os nomes dos modelos devem ser escritos em UpperCamelCase e no singular.

Models - Criando um model

filmes.js

Filme.js

Models - Criando um modelo

Um modelo é naturalmente uma função que devemos definir e exportar com **module.export**. Essa função recebe dois parâmetros. Primeiramente, ele recebe o objeto **sequelize** para poder acessar seu método **define()**, e em segundo lugar, precisamos trazer o objeto **DataTypes**, que nos dará a possibilidade de dizer às nossas colunas que tipo de dados elas permitirão.

Lembre-se de que, ao lidar com parâmetros, não é necessário que sejam chamados assim, mas normalmente fazemos isso para entender por que os usamos.

Models - Método .define()

O método **define()** nos permite definir atribuições entre um modelo e uma tabela, que recebe **3 parâmetros**. O primeiro é um alias que identifica o model, o segundo é um objeto com a configuração das colunas do banco de dados e o terceiro é outro objeto com configurações adicionais (parâmetro opcional). O que **define()** retorna será armazenado em uma variável com o nome do model, para posteriormente ser retornado pela função que criamos.

```
{}
const Filme = sequelize.define(alias, cols, config);
return Filme;
```

Models - Alias

Como mencionamos no slide anterior, o primeiro é um alias que o Sequelize usa para **identificar o modelo**. Não é algo determinante. Normalmente, atribuímos o mesmo nome do modelo como **String**.

```
{}
const Filme = sequelize.define("Filme", cols, config);
return Filme;
```

Models - Tipos de dados no sequelize

Dentro de nosso segundo parâmetro, que chamamos de **cols**, existe um objeto que nos permite definir quais tipos de dados as colunas devem receber no banco de dados.

```
{}
cols = {
    id: {
        type: DataTypes.INTEGER
    },
    name: {
        type: DataTypes.STRING
    },
    admin: {
        type: DataTypes.BOOLEAN
    }
}
```

DataTypes - Exemplos mais utilizados

```
DataType.STRING                 // VARCHAR(255)
DataType.STRING(400)                // VARCHAR(400)

DataType.INTEGER                    // INTEGER
DataType.BIGINT                 // BIGINT
DataType.FLOAT                  // FLOAT
DataType.DOUBLE                 // DOUBLE
DataType.DECIMAL                    // DECIMAL

DataType.DATE                   // DATE
```

Você encontrará mais tipos de dados na documentação oficial do Sequelize. Para acessá-la, clique no seguinte Link.

Model - Timestamps

```
module.exports = (sequelize, DataTypes) => {
  const Usuario = sequelize.define("Usuario", {
    email: {
      type: DataTypes.STRING
      },
    createdAt: {
      type: DataTypes.DATE
      },
    modifiedAt: {
      type: DataTypes.DATE
      },
  });

  return Usuario;
}
```

Campos que salvam a data de criação e última edição.

Os **timestamps** não são obrigatórios, mas a maioria das tabelas geralmente os contém e fazem parte do padrão. Eles devem ser nomeados da mesma forma que no exemplo.

Models - Configurações adicionais

Dentro de nosso terceiro parâmetro de **define()**, podemos configurar coisas adicionais. Por exemplo, se o nome de nossa tabela está em inglês e o de nosso modelo em português, devemos deixar claro para o modelo que este é o caso por meio de um objeto literal, como no exemplo do próximo slide.

```
module.exports = (sequelize, DataTypes) => {

    const Filme = sequelize.define("Filme",
        {
  // Configurações das colunas.
        },
        {
{}          tableName: 'movies',
  //Se o nome da tabela coincide com o model
            timestamps: false,
  //Se tenho timestamps
        });

    return Filme;
  }
```

Sequelize - Documentação

Para obter mais informações sobre DataTypes e o que pode ser configurado na mesma coluna no DB, siga o seguinte Link.

findAll, findOne, findByPk

DigitalHouse>

Índice

1. findAll()
2. findOne()
3. findByPk()

“Sequelize usa a função **FIND** para encontrar informações em um banco de dados. Junto com **FIND**, temos algumas variações como **findAll(), findOne(), findByPk().**”

1 findAll()

Select

findAll()

Para encontrar todos os dados registrados na tabela, devemos usar **findAll()**.

```
{}
const db = require('../database/models');

db.Usuario.findAll()
.then((resultados) => {
        console.log(resultados);
});
```

Select

findAll()

Para encontrar todos os dados registrados na tabela, devemos usar findAll().

```
const db = require('../database/models');

db.Usuario.findAll()
.then((resultados) => {
        console.log(resultados);
});
```

Inclui a conexão com o banco de dados.

DigitalHouse>

Select

findAll()

Para encontrar todos os dados registrados na tabela, devemos usar **findAll()**.

```
const db = require('../database/models');

db.Usuario.findAll()
.then((resultados) => {
        console.log(resultados);
});
```

A função **findAll()** retorna uma **promise** portanto, a usamos para pegar o resultado da pesquisa.

Select

findAll()

Para encontrar todos os dados registrados na tabela, devemos usar **findAll()**.

```
const db = require('../database/models');

db.Usuario.findAll()
.then((resultados) => {
        console.log(resultados);
});
```

O resultado será atribuído no **parâmetro** desta função, aqui o chamamos de **resultados**, mas pode ser **qualquer outro nome**.

2 findOne()

Select

findOne()

findOne() nos permite buscar resultados que correspondem aos atributos indicados no objeto literal que o método recebe.

```
{}
db.Usuario.findOne({
    where: {
        nome: 'Tony'
      }
    }).then((resultado) =>{
        console.log(resultado);
    });
```

3 findByPk()

Select

findByPk()

O método **findByPk()** procura por um registro com a chave primária do mesmo valor do parâmetro passado:

```
{}
db.Auto.findByPk(42)
.then((resultado) => {
        console.log(resultado);
});
// SELECT * FROM autos WHERE id = 42;
```

Select - Documentação

Para mais informações, visite a documentação oficial, clicando neste Link.

Where e Like

DigitalHouse>

Índice

1. Where
2. Operadores
3. Like

“Frequentemente, queremos pesquisar algo no banco de dados, mas não queremos todos os registros, apenas aqueles que atendem a uma condição. Para filtrar dados, usamos um objeto literal com o atributo **WHERE** e um método de pesquisa.”

Sequelize

Where

Para adicionar uma condição à consulta, basta passar o atributo **where** para o método **findAll()**:

```
{}
const db = require('../database/models');
db.Auto.findAll({
     where: {
        marca: 'Fiat'
}
}).then(resultados=>{
       console.log(resultados);
   })
```

Sequelize

Where

Para adicionar uma condição à consulta, basta passar o atributo **where** para o método **findAll()**:

```
const db = require('../database/models');
db.Auto.findAll({
     where: {
        marca: 'Fiat'
 }
}).then(resultados=>{
       console.log(resultados);
   })
```

Dentro do where passamos o atributo de acordo com a coluna da tabela e o valor a pesquisar.

DigitalHouse>

Sequelize

Like

Para adicionar condições à pesquisa **Where**, usamos os operadores que sequelize traz. Um deles é **Like**.

```
{}
const db = require('../database/models');
const Op = Sequelize.Op;
Post.findAll({
   where: {
      sobrenome: {[Op.Like]:'%s%'}
   }
}).then(resultados=>{
       console.log(resultados);
   })
```

Sequelize

Like

Para adicionar condições à pesquisa **Where**, usamos os operadores que sequelize traz. Um deles é **Like**.

```
{}
const db = require('../database/models');
const Op = Sequelize.Op;
Post.findAll({
   where: {
      sobrenome: {[Op.Like]:'%s%'}
   }
}).then(resultados=>{
      console.log(resultados);
   })
```

Requisitamos os operadores do Sequelize

Sequelize

Like

Para adicionar condições à pesquisa **Where**, usamos os operadores que sequelize traz. Um deles é **Like**.

```
{}
const db = require('../database/models');
const Op = Sequelize.Op;
Post.findAll({
   where: {
       sobrenome: {[Op.Like]:'%s%'}
   }
}).then(resultados=>{
       console.log(resultados);
   })
```

Neste caso, ele é filtrado pela coluna de sobrenome. Usamos o operador **like**, da mesma forma que fizemos no SQL.

Sequelize

Like

Para adicionar condições à pesquisa **Where**, usamos os operadores que sequelize traz. Um deles é **Like**.

```
{}
const db = require('../database/models');
const Op = Sequelize.Op;
Post.findAll({
   where: {
      sobrenome: {[Op.Like]:'%s%'}
   }

 }).then(resultados=>{
        console.log(resultados);
    })
```

Neste caso, o resultado da consulta seriam todos os sobrenomes que contivessem a letra 's'.

Where

Documentação

Para obter mais informações, visite a documentação oficial, clicando neste Link.

Order, Limit e Offset

DigitalHouse>

Índice

1. Order
2. Limit
3. Offset

“**ORDER** é uma forma de ordenar o resultado da consulta ao banco de dados por meio de uma coluna escolhida.

Posso classificar os elementos por id, data de criação, nome, etc.”

Ordenação

Order

Em sequelize, para ordenar o resultado, basta usar o atributo **order**, que recebe um array.

```
db.Usuario.findAll({
   order: [
   ['nome', 'ASC'],
   ],
});
```


DigitalHouse>

Ordenação

Order

Em sequelize, para ordenar o resultado, basta usar o atributo **order**, que recebe um array.

```
db.Usuario.findAll({
  order: [
    ['nome', 'ASC'],
    ],
});
```

O primeiro valor na matriz é a coluna que você deseja classificar, o segundo valor é a ordem de classificação: crescente ou decrescente.

“ **LIMIT** serve para limitar o número de resultados a serem obtidos. ”

Ordenação

Limite

Para limitar o número de resultados, basta adicionar o atributo **limit** ao objeto e passá-lo para **findAll ()**:

```
db.Usuario.findAll({
   limit: 10
   })
   .then((resultados) => {
   console.log(resultados)
})
```

“**OFFSET** é usado para omitir resultados, amplamente usado para paginar os resultados.”

Ordenação

Offset

Para pular vários elementos de resultado, basta adicionar o atributo **offset** ao objeto e passá-lo para **findAll()**:

```
db.Usuario.findAll({
   offset: 10
   })
   .then((resultados) => {
   console.log(resultados)
})
```

Ordenação

Order + Limit + Offset

Todos os métodos podem ser usados juntos, simplesmente passando-os como parâmetros para **findAll()**:

```
{}
db.Usuario.findAll({
   order: [['nome', 'ASC']],
   offset: 5,
   limit: 10
})
```

Ordenação

Order + Limit + Offset

Para obter mais informações, visite a documentação oficial, clicando no Link.

Sequelize

Métodos e Relacionamentos

DigitalHouse>

Índice

1. Create
2. Update
3. Destroy
4. Associate
5. Relações 1:N
6. Relações N:N

Objetivos

- Revisar os temas da aula através de perguntas de gatilho.
- Exercitar os processos de registros no banco de dados usando Sequelize.
- Escrever e ler informações a partir de dados relacionados.

1 | Create

.create()

Ele recebe um **objeto literal** onde definimos quais campos vamos modificar e com quais valores.

```
{}
const db = require('.../database/models');

db.User.create({}...);
```

Precisamos chamar o **model** primeiro.

2 | Update

.update()

Recebe dois parâmetros. Ambos são objetos literais. No primeiro, devemos indicar **campos** e **valores** a serem atribuídos. No segundo, deve ter pelo menos um **where**.

```
{}
const db = require('.../database/models');

db.User.update({}...,{where: {} });
```

Não esqueça a **WHERE.**

3 | Destroy

.destroy()

Recebe um parâmetro que deve ter pelo menos um **where**.

```
const db = require('.../database/models');

db.User.destroy({where: {} {});
```

Não esqueça a **WHERE.**

4 | Associate

.associate()

É atribuída uma função anônima que recebe um único parâmetro. Este será um **objeto que contém todos os models**, podendo assim ter acesso a cada um deles.

```
{}  Filme.associate = function(models){...}
```

Primeiro, defina-o com o método .define().

5 | RELAÇÕES 1:N

.belongsTo()

Usamos isso para dizer que **um registro** pode estar **associado** a **um ou mais registros** em outra tabela.

```
{}
Filme.belongsTo(models.Genero,
{ as: "generos", foreignKey: "genre_id"});
```

Primeiro, defina-o com o método .define().

.hasMany()

Usamos isso para dizer que **um ou mais registros** podem ser **associados** a um único **registro** em outra tabela.

```
{}
Genero.hasMany(models.Filme,
{ as: "filmes", foreignKey: "genre_id"});
```

Sempre que criamos um relacionamento a partir de um modelo, devemos gerar o mesmo a partir do outro a quem ele está relacionado.

6 | RELAÇÕES N:N

.belongsToMany()

Usamos isso para dizer que **muitos registros** em nossa tabela podem estar relacionados a **vários registros** em outra tabela.

```
{}
Filme.belongsToMany(models.Atores, {
    as: "atores",
    through: "actor_movie",
    foreignKey: "movie_id",
    otherKey: "actor_id",
    timestamps: false
});
```

Sequelize CREATE

DigitalHouse>

“Create é um método que nos permite **adicionar** novos registros às nossas tabelas de banco de dados.”

.create()

Método

É um método que pertence aos modelos de nosso banco de dados. Portanto, para acessá-lo, precisamos primeiro chamar o modelo.

```
const db = require('../database/models');
db.Usuario.create({
 });
```

Como precisamos chamar o model, precisamos exigir a conexão com o banco de dados

DigitalHouse>

.create()

Definindo o método

Recebe um **objeto literal** onde definimos quais campos vamos modificar e com quais valores.

```
const db = require('../database/models');
db.Usuario.create({
    name: "Manoel",
    username: "manolito",
    password: "manolo123"
});
```

O objeto literal que recebe .create() deve conter como **KEY o mesmo nome do campo para escrever no banco de dados** e como valor o conteúdo que queremos armazenar nele.

Documentação

Para saber mais, você pode acessar a documentação oficial de Sequelize clicando no link.

DigitalHouse>

Sequelize UPDATE

DigitalHouse>

Índice

1. Método
2. Conceitos Importantes

“Update é um método que nos permite **editar** registros em nossas tabelas de banco de dados.”

1 | Método

.update()

É um método que pertence aos modelos de nosso banco de dados. Portanto, para acessá-lo, precisamos primeiro chamar o model.

```
{}
const db = require('../database/models');

db.Usuario.update();
```

2 | Conceitos Importantes

Conceitos Importantes

.update () leva dois parâmetros. **Ambos** são objetos literais.

No primeiro, devemos indicar qual campo da tabela **modificar** e qual valor atribuir.

O segundo objeto deve ter, pelo menos, um **where** que indique exclusivamente a qual registro aplicar as alterações. Caso contrário, todos os campos do BD serão modificados.

Não se esqueça do **WHERE.**

Exemplo

```
const db = require('../database/models');

db.Usuario.update(
{
    username: 'ManuelF'
},
{
   where: {id: 10}
});
```

Sequelize DESTROY

DigitalHouse>

Índice

1. Método
2. Conceitos Importantes

“Quando quiser **excluir** um registro da sua base de dados, basta utilizar o método **destroy()**.”

1 Método

.destroy()

É um método que pertence aos modelos de nosso banco de dados. Portanto, para acessá-lo, precisamos primeiro chamar o modelo.

```
{}
const db = require('../database/models');

db.Usuario.destroy({
   where: {id: 10}
});
```

2 Conceitos Importantes

Conceitos Importantes

.destroy () recebe um único parâmetro. Este será um **objeto** com um **where** que terá a condição que precisamos especificar para eliminar o registro que desejamos. Não fazer isso removerá todos os registros daquela tabela.

Não se esqueça do **WHERE.**

Sequelize: Relações e CRUD

DigitalHouse>

Índice

1. Relações 1:N
2. Relações N:M
3. CRUD

Objetivos

- Praticar o processo de criação, atualização e eliminação de registros.
- Reforçar a lógica e a utilidade da camada do modelo.
- Compreender como ler e escrever informações quando esses dados envolvem mais de uma tabela de banco de dados.
- Exercício sobre escrita e relações no banco de dados.

1 Relações 1:N

Razões 1:N

1 gênero pode se aplicar a **muitos** filmes

Filmes

Gêneros

1 filme, tem **1** gênero

Razões 1:N

Genre **hasMany**(modelos.Filmes)

1 gênero pode se aplicar a **muitos** filmes

1 filme, tem **1** gênero

Filme.**belongsTo**(modelos.gêneros)

2 Relações N:M

Relacionamentos N:M

1 ator pode ter atuado em **muitos** filmes

Em **1** filme, **muitos** atores podem interpretar

Relacionamentos N:M

Ator.**belongsToMany**(modelos.Filmes)

1 ator pode ter atuado em **muitos** filmes

Em **1** filme, **muitos** atores podem interpretar

Filme.**belongsToMany**(modelos.Atores)

3 | CRUD

CRUD

Create

Read

Update

Delete

CRUD

Create → `db.Filme.create()`

Read → `db.Filme.findAll()`

Update → `db.Filme.update()`

Delete → `db.Filme.destroy()`

Sequelize
Relações 1:N

DigitalHouse>

Índice

.associate
.belongsTo()
.hasMany()

“Os relacionamentos no Sequelize existem para otimizar a recuperação de dados em uma consulta de banco de dados.”

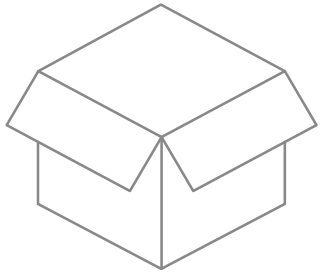
1 | .associate

Sequelize

.associate

Os relacionamentos são aplicados em nosso modelo de tabela. Depois de defini-la com o método **.define()**, devemos chamar a variável criada e usar a propriedade **.associate** para definir nossos relacionamentos.

.associate deve armazenar uma **função anônima** que leva um **único parâmetro**. Este será um objeto que conterá todos os seus **modelos**, podendo assim acessar cada um deles.



Sequelize

.associate

```
const Filme = sequelize.define(alias, cols, config);

Filme.associate = function(modelos){
    // Relacionamento
}
```

Definimos uma função na propriedade **.associate** da nossa **variável que representa o modelo** (Filme).

Sequelize

.belongsTo() | .hasMany()

.belongsTo() e **.hasMany()** são métodos do Sequelize que nos permitem indicar aos nossos modelos sua relação com os outros. Eles são declarados dentro da função que atribuímos a **.associate**, e devem ser aplicados à nossa variável definida com **.define()**.

```
const Filme = sequelize.define(alias, cols, config);


Filme.associate = function(modelos){
    // Filme.belognsTo()
    // Filme.hasMany()
}
```

Criamos as relações dentro da função atribuída a **.associate.**

2 | .belongsTo()

Sequelize

.belongsTo()

.belongsTo() é usado para dizer que um registro pode ser associado a uma ou mais tabelas. Nesse caso, a relação seria 1:N.

Para definir o relacionamento, **.belongsTo()** recebe dois parâmetros, onde o primeiro é o modelo com o qual você deseja relacioná-lo (chamando-o através do parâmetro que contém nossos modelos) e o segundo é um objeto onde devemos detalhar o relacionamento.

Sequelize

.belongsTo()

```
Filme.associate = function(modelos){
   Filme.belongsTo(modelos.Generos, {

     as: "generos",

    foreignKey: "genre_id"
    });
}
```

Atribuímos um alias com o qual chamaremos a relação.

DigitalHouse>

Sequelize

.belongsTo()

```
Filme.associate = function(modelos){
   Filme.belongsTo(modelos.Generos, {

      as: "generos",

     foreignKey: "genre_id"

    });
}
```

Especificamos a foreign key onde as duas tabelas se relacionam

DigitalHouse>

3 .hasMany()

Sequelize

.hasMany()

.hasMany() é usado para dizer que um ou mais registros podem ser associados a apenas um de outra tabela. Nesse caso, a relação seria N:1.

Para definir a relação, **.hasMany()** recebe dois parâmetros, onde o primeiro é o modelo com o qual você deseja relacioná-lo (chamando-o através do parâmetro que contém nossos modelos) e o segundo é um objeto onde devemos detalhar o relacionamento.

Sequelize

.hasMany()

```
{}
Genero.associate = function(modelos){
   Genero.hasMany(modelos.Filme, {

     as: "filmes",

    foreignKey: "genre_id"
    });
}
```

Atribuímos um alias com o qual chamaremos a relação.

Sequelize

.hasMany()

```
{}
Genero.associate = function(modelos){
   Genero.hasMany(modelos.Filme, {

     as: "filmes",

     foreignKey: "genre_id"
     });
}
```

Especificamos a foreign key onde as duas tabelas se relacionam

ATENÇÃO!

Sempre que criamos um relacionamento a partir de um modelo, devemos gerar o mesmo a partir do outro com quem se relaciona. Se não o fizermos, Sequelize não reconhecerá a associação.

Sequelize

Documentação

Para saber mais, você pode acessar a documentação oficial do Sequelize, clicando no seguinte Link.

Sequelize
Relações N:M

DigitalHouse>

Índice

.belongsToMany()
Requerimentos
Implementação

“Os relacionamentos no Sequelize existem para otimizar a recuperação de dados em uma consulta de banco de dados.”

1 .belongsToMany()

Sequelize

.belongsToMany()

Há momentos em que uma tabela tem um relacionamento muitos para muitos. Ou seja, muitos registros em nossa tabela podem estar relacionados a vários em outra. Para esclarecer essa relação com o Sequelize, devemos usar o método .belongsToMany().

Para definir o relacionamento, .belongsToMany () recebe dois parâmetros, onde o primeiro é o modelo com o qual você deseja relacioná-lo (chamando-o através do parâmetro que contém nossos modelos) e o segundo é um objeto onde devemos detalhar o relacionamento.

Sequelize

.belongsToMany()

```
Filme.associate = function(modelos){
   Filme.belongsToMany(modelos.Atores, {
     as: "atores",
     through: "actor_movie",
     foreignKey: "movie_id",
     otherKey: "actor_id",
     timestamps: false
     });
}
```

Atribuímos um alias com o qual chamaremos o relacionamento.

DigitalHouse>

Sequelize

.belongsToMany()

```
Filme.associate = function(modelos){
   Filme.belongsToMany(modelos.Atores, {
     as: "atores",

     through: "actor_movie",
     foreignKey: "movie_id",
     otherKey: "actor_id",
     timestamps: false
     });
}
```

Nome da tabela intermediária que criará o relacionamento.

DigitalHouse>

Sequelize

.belongsToMany()

```
Filme.associate = function(modelos){
   Filme.belongsToMany(modelos.Atores, {
     as: "atores",
     through: "actor_movie",

{}
      foreignKey: "movie_id",
     otherKey: "actor_id",
     timestamps: false
     });
}
```

Referência à tabela de modelo que estamos associando.

Sequelize

.belongsToMany()

```
Filme.associate = function(modelos){
   Filme.belongsToMany(modelos.Atores, {
     as: "atores",
     through: "actor_movie",
     foreignKey: "movie_id",
     otherKey: "actor_id",
     timestamps: false
     });
}
```

Referência à outra tabela que queremos associar.

Sequelize

.belongsToMany()

```
{}
Filme.associate = function(modelos){
   Filme.belongsToMany(modelos.Atores, {
     as: "atores",
     through: "actor_movie",
     foreignKey: "movie_id",
     otherKey: "actor_id",

      timestamps: false

     });
}
```

Inferimos se a tabela pivô possui timestamp.

2 Requerimentos e Implementação

Sequelize

Requerimentos

Verifique os requisitos a seguir para garantir que os relacionamentos sejam bem criados ao usar o Sequelize.

Sequelize

Implementação

- As consultas ao BD geralmente recebem um objeto como parâmetro.
- Dentro desse objeto, devemos adicionar um atributo include que recebe um array de objetos.
- Cada objeto nessa matriz representa um relacionamento. Devemos esclarecer qual queremos usar.
- Para definir o relacionamento, criamos um atributo de **associação** que contém o mesmo alias que usamos no modelo para chamar o relacionamento.

Sequelize

Implementação

```
const db = require('../database/models');
db.Filme.findAll({
 include: [
 {association: "filmes"},
 {association: "atores"}
 ]
}).then(resultados=>{
        console.log(resultados);
   })
```

Incluímos as associações (relacionamentos) que criamos nos modelos.

DigitalHouse>

Sequelize

Na view

Para usar essas informações na view, devemos chamar as associações por meio dos atributos do modelo. Eles têm o mesmo nome do alias que estabelecemos quando criamos o relacionamento.

EJS

```
// Mostra o gênero a que pertence o filme
<%= filme.genero %>
   // Mostra os atores do filme
   <% for (let i = 0; i < filme.atores.length; i++) { %>
       <%= filme.atores[i] %>
   <% } %>
```

ATENÇÃO!

Sempre que criamos um relacionamento a partir de um modelo, devemos gerar o mesmo a partir do outro com quem se relaciona. Se não o fizermos, Sequelize não reconhecerá a associação.

Sequelize

Documentação

Para saber mais, você pode acessar a documentação oficial do Sequelize, clicando no seguinte Link.

Sequelize
Raw Queries

DigitalHouse >
Coding School

Índice

Raw Queries
Substituições
Parâmetros de associação

“Frequentemente, você vai se deparar com situações em que é simplesmente mais fácil lidar com queries literais.”

1 Raw Queries

Raw Queries

.query

Por padrão, a função retornará **dois argumentos** - um array de **resultados** e um objeto contendo **metadados** (como a quantidade de linhas afetadas, etc). Observe que, como esta é uma consulta bruta, os metadados são **específicos do dialeto**. Alguns dialetos retornam os metadados "dentro" do objeto de resultados (como propriedades em um array). No entanto, dois argumentos sempre serão retornados, mas para MSSQL e MySQL serão duas referências ao mesmo objeto.

Raw Queries

.query

```
{}
const [resultado, metadata] = await sequelize.query("UPDATE
filmes SET nome = 'Vingadores Ultimato' WHERE id = 148")
```

O **resultado** será um array vazio e **metadata** conterá o número de linhas afetadas.

Raw Queries

.query

```
const { QueryTypes } = require('sequelize');
const users = await sequelize.query("SELECT * FROM 'filmes'",
{ type: QueryTypes.SELECT });
```

Não precisamos **desestruturar** o resultado aqui, pois eles são **retornados diretamente**.

2 | Substituições

Raw Queries

Substituição

As **substituições**, também chamadas de **replacements**, em uma consulta podem ser feitas de **duas maneiras** diferentes: usando parâmetros **nomeados** (começando com `:`) ou **não nomeados**, representados por um `?`. As substituições são passadas no objeto de opções.

Raw Queries

Substituição

```
const { QueryTypes } = require('sequelize');

await sequelize.query(
  'SELECT * FROM atores WHERE nacionalidade = ?',
  {
    replacements: ['brasileira'],
    type: QueryTypes.SELECT
  }
);
```

Se for passado um array as **?** serão substituídas na ordem que aparecem no array.

Raw Queries

Substituição

```
const { QueryTypes } = require('sequelize');

await sequelize.query(
    'SELECT * FROM filmes WHERE lingua = :lingua',
    {
      replacements: { lingua: 'português' },
      type: QueryTypes.SELECT
    }
  );
```

Se for passado um objeto os atributos **:key** serão substituídos a partir do objeto de origem.

3 Parâmetros de associação

Raw Queries

Parâmetros de associação

Os **parâmetros de associação**, também conhecidas como **bind parameters,** são como substituições. Exceto que **substituições** são inseridas na consulta pelo Sequelize **antes** que a consulta seja enviada ao banco de dados, enquanto os bind parameters são enviados ao banco de dados **fora do texto** da consulta SQL.

Raw Queries

Parâmetros de associação

```
const { QueryTypes } = require('sequelize');

await sequelize.query(
    'SELECT * FROM atores WHERE nome = $1',
    {
      bind: ['Ana'],
      type: QueryTypes.SELECT
    }
  );
```

Se um **array** for passado o parâmetro **$1** está associado ao **primeiro elemento** do array (bind[0])

Raw Queries

Parâmetros de associação

```
const { QueryTypes } = require('sequelize');

await sequelize.query(
    'SELECT *, "text with literal $$1 and literal $$status"
as t FROM projects WHERE status = $status',
    {
      bind: { status: 'active' },
      type: QueryTypes.SELECT
    }
  );
```

Se for passado um objeto os atributos **$key** serão substituídos a partir do objeto de origem. Cada $key deve **começar** com caractere **não-numérico**.

Sequelize

Documentação

Para saber mais, você pode acessar a documentação oficial do Sequelize, clicando no seguinte Link.

DigitalHouse>
Coding School