

## Material de apoio

Site: [Geração Tech](#)  
Curso: Formação em Desenvolvedor Web - Online  
Livro: Material de apoio

Impresso por: JOÃO VITOR DE MELO FREITAS  
Data: sexta-feira, 2 ago. 2024, 08:57

# Índice

## 1. Banco de Dados

- 1.1. Descrição da Tabela
- 1.2. Criando a Tabela
- 1.3. Comando DELETE
- 1.4. Utilizando o UPDATE
- 1.5. Preparando o Ambiente
- 1.6. Instalando Bibliotecas Necessárias
- 1.7. Nomes de Tabelas
- 1.8. Estrutura do Banco de Dados
- 1.9. Estrutura do Projeto
- 1.10. Vídeo aula
- 1.11. Criando um Arquivo Central para Sincronização
- 1.12. Vídeo Aula
- 1.13. Criando o Model Tags
- 1.14. Vídeo Aula
- 1.15. Vídeo Aula
- 1.16. Implementando a Função de Criação

# 1. Banco de Dados


Estrutura de Banco de Dados e Filtros

## Passo a Passo da Atividade

### 1. Excluir o Banco de Dados Existente

Primeiro, excluímos o banco de dados existente para garantir que estamos começando do zero.

sql

 Copiar código

```
DROP DATABASE IF EXISTS blog;
```

### 2. Criar um Novo Banco de Dados

Criamos um novo banco de dados com o mesmo nome.

sql

 Copiar código

```
CREATE DATABASE blog;
```

### 3. Utilizar o Banco de Dados Criado

Selecionamos o banco de dados recém-criado para que todas as operações subsequentes sejam feitas nele.

sql

 Copiar código

```
USE blog;
```


### 4. Criar Tabelas

#### Tabela **users**

Criamos a tabela **users** com as seguintes colunas e tipos de dados:

- **user\_id**: inteiro auto-incremento
- **user\_name**: texto com até 45 caracteres
- **email**: texto com até 45 caracteres

sql

 Copiar código

```
CREATE TABLE users ( user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(45), email VARCHAR(45) );
```

#### Tabela **posts**

Criamos a tabela **posts** com as seguintes colunas e tipos de dados:

- **post\_id**: inteiro auto-incremento
- **post\_title**: texto com até 255 caracteres
- **post\_content**: texto
- **post\_date**: data
- **user\_id**: inteiro (chave estrangeira de **users**)

sql

 Copiar código

```
CREATE TABLE posts ( post_id INT AUTO_INCREMENT PRIMARY KEY, post_title VARCHAR(255), post_content TEXT, post_date DATE, user_id INT, FOREIGN KEY (user_id) REFERENCES users(user_id) );
```

dia 36 video 01



## 5. Inserir Dados nas Tabelas

### Inserir Dados na Tabela **users**

Inserimos alguns usuários na tabela **users**.

sql

 Copiar código

```
INSERT INTO users (user_name, email) VALUES ('Alice', 'alice@example.com'), ('Bob', 'bob@example.com'), ('Carol', 'carol@example.com');
```

### Inserir Dados na Tabela **posts**

Inserimos alguns posts na tabela **posts**.

sql

 Copiar código

```
INSERT INTO posts (post_title, post_content, post_date, user_id) VALUES ('Primeiro Post', 'Conteúdo do primeiro post.', '2024-01-01', 1), ('Segundo Post', 'Conteúdo do segundo post.', '2024-02-01', 2), ('Terceiro Post', 'Conteúdo do terceiro post.', '2024-03-01', 3);
```

## Revisão de Filtros e Consultas

Após a criação e inserção dos dados, podemos aplicar vários filtros para consultar as tabelas. Aqui estão alguns exemplos de consultas que podemos fazer.

### Selecionar Todos os Usuários


sql

 Copiar código

```
SELECT * FROM users;
```

### Selecionar Todos os Posts

sql

 Copiar código

```
SELECT * FROM posts;
```

## Selecionar Posts por um Usuário Específico

sql

 Copiar código

```
SELECT * FROM posts WHERE user_id = 1;
```

## Selecionar Posts Entre Duas Datas

sql

 Copiar código

```
SELECT * FROM posts WHERE post_date BETWEEN '2024-01-01' AND '2024-03-01';
```

## Selecionar Posts com um Determinado Título

sql

 Copiar código

```
SELECT * FROM posts WHERE post_title LIKE '%Post%';
```

## Conclusão

Este exercício revisou a criação de bancos de dados e tabelas, a inserção de dados e a aplicação de filtros para consultas. Comparando o seu código com o apresentado aqui, você pode ajustar e aprimorar suas habilidades em SQL.

## Próximos Passos

Continuaremos explorando novas funcionalidades e técnicas em SQL nas próximas aulas. Certifique-se de entender bem esses conceitos, pois eles são fundamentais para o desenvolvimento eficiente de bancos de dados.

Até a próxima aula!

## 1.1. Descrição da Tabela


Nas aulas anteriores, exploramos a criação de bancos de dados e tabelas, além de aplicarmos diversos filtros.

Agora, vamos nos aprofundar nas definições de colunas e constraints que ajudam a garantir a integridade dos dados no banco.

### Descrição da Tabela

Para entender melhor como nossas tabelas estão estruturadas, utilizamos o comando `DESCRIBE`. Vamos ver como isso funciona para a tabela `users`.

sql

 Copiar código

```
DESCRIBE users;
```

### Resultado

O comando `DESCRIBE` nos mostra as colunas, tipos de dados, se permitem valores nulos, se são chaves primárias e outras características. Por exemplo, nossa tabela `users` pode ter o seguinte resultado:


Field	Type	Null	Key	Default	Extra
user_id	int	NO	PRI	NULL	auto_increment
user_name	varchar(45)	YES		NULL	
email	varchar(45)	YES		NULL	

### Inserção de Dados e Constraints

#### Inserindo Dados

Vamos inserir alguns dados na tabela `users`.

sql

 Copiar código


```
INSERT INTO users (user_name, email) VALUES ('Test User', 'test@example.com');
```

### Campos Não Nulos

Vamos fazer um teste para inserir dados sem passar valores para colunas específicas.

#### Teste com Colunas Opcionais

sql


 Copiar código

```
INSERT INTO users (user_name) VALUES ('User Without Email');
```

#### Teste com Colunas Obrigatórias

Vamos criar uma tabela onde o campo `email` é obrigatório.

sql


 Copiar código

```
CREATE TABLE users_required ( user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(45), email VARCHAR(45) NOT NULL );
```

### Inserindo Sem Valores Obrigatórios

Tentar inserir sem o valor obrigatório:

sql

 Copiar código


```
INSERT INTO users_required (user_name) VALUES ('User Without Email');
```

Isso resultará em erro, pois a coluna `email` não pode ser nula.

## Definindo Valores Padrão

Podemos definir valores padrão para colunas ao criar a tabela. Vamos ver como fazer isso.

sql


 Copiar código

```
CREATE TABLE users_default ( user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(45), email VARCHAR(45) DEFAULT 'default@example.com' );
```

## Inserindo com Valores Padrão

Inserindo sem especificar a coluna `email`, o valor padrão será usado:

sql

 Copiar código

```
INSERT INTO users_default (user_name) VALUES ('User With Default Email');
```

dia 36 video 02



## Resumo

Com essas instruções, aprendemos a definir colunas obrigatórias (`NOT NULL`), a definir valores padrão (`DEFAULT`) e a garantir que os dados inseridos nas tabelas atendam aos critérios de integridade que estabelecemos.

Isso ajuda a manter a consistência e a confiabilidade dos dados no banco de dados.

## Próximos Passos

Vamos continuar explorando outras funcionalidades e práticas recomendadas para o uso de bancos de dados SQL.

Certifique-se de entender bem esses conceitos, pois são fundamentais para a manipulação e integridade dos dados.

Até a próxima aula!

## 1.2. Criando a Tabela

Olá alunos, vamos a mais uma aula!

Vamos abordar duas formas de trabalhar com tabelas: criando uma tabela do zero e configurando uma tabela já existente.

Primeiramente, vamos criar uma tabela de exemplo chamada `posts2`.

Depois, excluiríamos ela, mas por enquanto servirá para o nosso aprendizado.

### Criando a Tabela

Vamos criar a tabela `posts2` com as colunas `title` e `content`. Definiremos `title` como um campo obrigatório e `content` como opcional. Precisamos também de um identificador único para cada linha da tabela, que será a coluna `id`.

sql

 Copiar código

```
CREATE TABLE posts2 ( id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, title VARCHAR(255) NOT NULL, content TEXT );
```

### Inserindo Dados

Agora que criamos a tabela, vamos inserir alguns dados. O campo `id` será preenchido automaticamente pelo sistema.

sql

 Copiar código

```
INSERT INTO posts2 (title, content) VALUES ('JS', 'Conteúdo do post');
```

Após a inserção, faremos uma consulta para verificar os dados.


sql

 Copiar código

```
SELECT * FROM posts2;
```

O resultado mostrará a coluna `id` preenchida automaticamente:

lua

 Copiar código

```
+----+-----+-----+ | id | title | content | +----+-----+-----+ | 1 | JS | Conteúdo do post | +----+-----+-----+
```

### Atualizando a Tabela

Se já temos uma tabela existente e queremos adicionar uma nova coluna, podemos fazer isso sem precisar deletar a tabela.

sql

 Copiar código

```
ALTER TABLE posts ADD COLUMN id INT NOT NULL AUTO_INCREMENT PRIMARY KEY;
```

A coluna `id` será adicionada e configurada para ser uma chave primária e auto incrementada.

### Visualizando as Configurações

Podemos visualizar as configurações da tabela usando o comando `DESCRIBE`:

sql



[Copiar código](#)`DESCRIBE posts2;`

Isso nos mostrará detalhes das colunas, tipos de dados e restrições.

dia 36 video 03



## Considerações Finais

As chaves primárias são essenciais para identificar de forma única cada item na nossa tabela, facilitando buscas e relacionamentos com outras tabelas.

E por hoje é isso! Ficamos por aqui e até a próxima aula.

## 1.3. Comando DELETE

Bom pessoal, aqui no vídeo de hoje a gente vai aprender outra operação que podemos fazer no banco de dados.

Até agora, já aprendemos a criar tabelas usando o comando `CREATE TABLE`, a inserir informações com o comando `INSERT` e a visualizar dados com o comando `SELECT`.


Hoje, vamos focar em uma operação essencial: o comando `DELETE`, que usamos para excluir linhas específicas da nossa tabela.

Para ilustrar isso, eu dupliquei a tabela `posts` para criar outras três tabelas (`posts2`, `posts3` e `posts4`), assim podemos fazer vários exemplos sem precisar recriar dados toda hora. Vou usar a tabela `posts2` para os exemplos.

### Comando DELETE

Vamos começar com o comando mais básico:

sql


 Copiar código

```
DELETE FROM posts2;
```

Este comando vai deletar **todas** as linhas da tabela `posts2`. Executar esse comando sem um filtro é muito perigoso, pois você pode perder todos os dados da tabela.

Vamos ver como funciona:

sql


 Copiar código

```
DELETE FROM posts2;
```

Ao executar, recebo um erro indicando que o `Safe updates` está ativado. Isso é uma medida de segurança do MySQL que evita a exclusão de todas as linhas acidentalmente.

Para fins didáticos, vou desabilitar essa segurança:

sql

 Copiar código

```
SET SQL_SAFE_UPDATES = 0;
```

Agora, posso executar o comando `DELETE FROM posts2;` e deletar todas as linhas da tabela `posts2`.

Ao fazer uma nova consulta, veremos que a tabela está vazia.

### Deletando Linhas Específicas

Para evitar deletar todas as linhas acidentalmente, usamos um filtro com a cláusula `WHERE`.

Vamos supor que queremos deletar apenas a linha com `id = 4`:

sql

 Copiar código

```
DELETE FROM posts2 WHERE id = 4;
```

Executando esse comando, deletamos apenas a linha com `id = 4`. Se quisermos deletar outra linha, por exemplo, com `id = 6`:

sql


 Copiar código

```
DELETE FROM posts2 WHERE id = 6;
```

## Praticando

Vamos praticar. Na tabela `posts3`, que tem seis linhas, quero deletar a linha com `id = 4`. Primeiro, vamos ver como está a tabela:


sql

 Copiar código

```
SELECT * FROM posts3;
```

Agora, vamos deletar a linha com `id = 4`:

sql

 Copiar código

```
DELETE FROM posts3 WHERE id = 4;
```

Ao executar e verificar novamente com `SELECT * FROM posts3;`, veremos que a linha com `id = 4` foi removida.

Lembre-se sempre de usar a cláusula `WHERE` ao deletar linhas para evitar perder todos os dados da tabela.

A chave primária (`id`) é uma ótima forma de garantir que estamos deletando exatamente a linha desejada, já que é única para cada linha.

## Reativando o Modo Seguro

Por fim, vou reativar o modo de `Safe updates` para evitar futuras exclusões acidentais:

sql

 Copiar código

```
SET SQL_SAFE_UPDATES = 1;
```

E com isso, concluímos a nossa aula de hoje sobre o comando `DELETE`.

Na próxima aula, veremos como usar o comando `UPDATE` para atualizar informações em nossas tabelas.

Até a próxima aula, pessoal!

## 1.4. Utilizando o UPDATE

Bom pessoal, agora a gente vai ver o **UPDATE**.

Já vimos o **INSERT**, o **SELECT**, acabamos de ver como funciona o **DELETE** e agora vamos para o **UPDATE**, que usamos quando queremos alterar um valor que já está salvo no banco de dados.


### Utilizando o UPDATE

O comando **UPDATE** é usado para modificar os dados existentes em uma tabela. Vamos ver como ele funciona na prática. Suponha que temos uma tabela chamada **posts2** e queremos atualizar um valor em uma coluna específica.

### Estrutura do Comando UPDATE

O comando **UPDATE** tem a seguinte estrutura básica:


sql

 Copiar código

```
UPDATE nome_da_tabela SET nome_da_coluna = novo_valor WHERE condição;
```

Vamos praticar isso. Primeiramente, veja o conteúdo atual da tabela **posts2**:


sql

 Copiar código

```
SELECT * FROM posts2;
```

Suponhamos que temos a seguinte saída:


lua

 Copiar código

```
+---+-----+-----+ | id | title | content | +---+-----+-----+ | 1 | Post 1 | Conteúdo do post 1 |
| 2 | Post 2 | Conteúdo do post 2 |
| 3 | Post 3 | Conteúdo do post 3 |
| 4 | Post 4 | Conteúdo do post 4 |
| 5 | Post 5 | Conteúdo do post 5 |
| 6 | Post 6 | Conteúdo do post 6 | +---+-----+-----+
```

Agora, queremos atualizar o título do post com **id = 3** para "Atualizando Post 3". Para isso, usamos o comando **UPDATE** com a cláusula **WHERE** para especificar qual linha queremos modificar:

sql

 Copiar código

```
UPDATE posts2 SET title = 'Atualizando Post 3' WHERE id = 3;
```

### Erro de Safe Updates


Assim como no comando **DELETE**, o MySQL pode impedir a execução de um **UPDATE** sem uma cláusula **WHERE** para proteger contra atualizações acidentais de todas as linhas.

Se você tentar executar um **UPDATE** sem **WHERE**, você pode ver um erro relacionado ao modo de "Safe updates".

### Desativando o Modo de Safe Updates

Para fins didáticos, desativamos o modo de "Safe updates":

sql

 Copiar código

```
SET SQL_SAFE_UPDATES = 0;
```

Atualizando com Condição

Vamos novamente tentar atualizar o título do post com id = 3:

```
sql
Copiar código
```

```
UPDATE posts2 SET title = 'Atualizando Post 3' WHERE id = 3;
```

Executando o comando acima, você verá que apenas a linha com id = 3 foi atualizada. Agora, vamos verificar:

```
sql
Copiar código
```

```
SELECT * FROM posts2 WHERE id = 3;
```

A saída será:

```
lua
Copiar código
```

	id	title	content	
	3	Atualizando Post 3	Conteúdo do post 3	

Atualizando Várias Colunas

Podemos também atualizar múltiplas colunas ao mesmo tempo. Suponha que queremos atualizar tanto o title quanto o content do post com id = 6:

```
sql
Copiar código
```

```
UPDATE posts2 SET title = 'Novo Título 6', content = 'Novo Conteúdo do Post 6' WHERE id = 6;
```

Vamos verificar a tabela após essa atualização:

```
sql
Copiar código
```

```
SELECT * FROM posts2 WHERE id = 6;
```

A saída será:

```
lua
Copiar código
```

	id	title	content	
	6	Novo Título 6	Novo Conteúdo do Post 6	

Reativando o Modo de Safe Updates

Após fazer todas as alterações necessárias, é uma boa prática reativar o modo de "Safe updates":

```
sql
Copiar código
```

```
SET SQL_SAFE_UPDATES = 1;
```

dia 36 video 04



## Considerações Finais

E com isso, cobrimos as quatro operações principais de manipulação de dados: inserir (**INSERT**), selecionar (**SELECT**), atualizar (**UPDATE**) e deletar (**DELETE**).

Não esqueça de sempre usar a cláusula **WHERE** no **UPDATE** e **DELETE** para evitar modificar ou remover todas as linhas acidentalmente.

Até o próximo vídeo, pessoal!

## 1.5. Preparando o Ambiente

Bom pessoal, já aprendemos a criar tabelas, bancos de dados e a fazer manipulações de dados.

Agora, vamos configurar nosso projeto para que possamos utilizar um banco de dados real em vez de simular os dados.

No projeto que estávamos usando para construir a API, os dados eram armazenados em um array, e agora vamos configurar para usar um banco de dados de verdade.


### Preparando o Ambiente

Primeiro, precisamos instalar duas bibliotecas essenciais: `mysql2` e `sequelize`.

O `mysql2` fornecerá os recursos para conectar com o banco de dados MySQL, e o `sequelize` será o nosso ORM (Object-Relational Mapper), facilitando a manipulação dos dados.

1. Abra o terminal no diretório do seu projeto.
2. Execute o seguinte comando para instalar as bibliotecas:


sh

 Copiar código

```
npm install mysql2 sequelize
```

Após a instalação, verifique no `package.json` se as bibliotecas foram adicionadas:

json

 Copiar código

```
"dependencies": { "mysql2": "^2.0.0", "sequelize": "^6.0.0" }
```


### Configurando a Conexão com o Banco de Dados

Agora, vamos configurar a conexão com o banco de dados. Dentro da pasta da API, criaremos uma nova pasta chamada `config` para armazenar as configurações.

1. Crie a pasta `config`.
2. Dentro da pasta `config`, crie um arquivo chamado `database.js`.

No arquivo `database.js`, vamos configurar a conexão com o banco de dados usando `sequelize` e `mysql2`:

javascript

 Copiar código


```
const { Sequelize } = require('sequelize'); // Configuração da conexão com o banco de dados const sequelize = new Sequelize('blog', 'root', 'sua_senha', { host: 'localhost', dialect: 'mysql', port: 3306 }); module.exports = sequelize;
```

### Testando a Conexão

Para garantir que a conexão está funcionando corretamente, vamos criar um arquivo de teste simples.

Crie um arquivo chamado `testConnection.js` na raiz do seu projeto:


javascript

 Copiar código

```
const sequelize = require('./config/database'); async function testConnection() { try { await sequelize.authenticate(); console.log('Conexão estabelecida com sucesso.')} catch (error) { console.error('Erro ao conectar com o banco de dados:', error); } } testConnection();
```

Agora, execute o arquivo para testar a conexão:

sh

 Copiar código

```
node testConnection.js
```

Se a conexão estiver correta, você verá a mensagem "Conexão estabelecida com sucesso." no console.

Caso contrário, verifique as configurações e o estado do seu servidor MySQL.


## Estruturando o Projeto

Agora que a conexão está funcionando, vamos estruturar o projeto para utilizar o banco de dados em vez do array.

1. Crie uma pasta `models` dentro da pasta da API.
2. Dentro da pasta `models`, crie um arquivo chamado `Post.js`.

No arquivo `Post.js`, vamos definir o modelo para a tabela `posts`:

javascript

 Copiar código


```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); // Definindo o modelo Post
const Post = sequelize.define('Post', { title: { type: DataTypes.STRING, allowNull: false }, content: { type:
DataTypes.TEXT, allowNull: true } }); module.exports = Post;
```

## Sincronizando o Modelo

Para criar a tabela `posts` no banco de dados, precisamos sincronizar o modelo com o banco.

Crie um arquivo chamado `sync.js` na raiz do projeto:

javascript

 Copiar código

```
const sequelize = require('../config/database'); const Post = require('../models/Post'); async function syncDatabase() { try {
await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.'); } catch (error) { console.error('Erro ao
sincronizar o banco de dados:', error); } } syncDatabase();
```

Execute o arquivo para sincronizar o modelo com o banco de dados:

sh

 Copiar código

```
node sync.js
```

Se tudo estiver correto, a tabela `posts` será criada no banco de dados.



dia 36 video 05



## Próximos Passos

Com o banco de dados configurado, agora podemos começar a substituir os arrays por operações reais no banco de dados utilizando **sequelize** nos controladores da API.

E é isso por hoje! Configuramos nosso projeto para conectar e manipular dados em um banco de dados real. Até o próximo vídeo!

## 1.6. Instalando Bibliotecas Necessárias

Agora, vamos configurar nosso projeto para criar tabelas através do Node.js utilizando o Sequelize.

Vamos aproveitar os benefícios do ORM para facilitar nosso trabalho com o banco de dados.

### Instalando Bibliotecas Necessárias

Primeiro, precisamos garantir que temos as bibliotecas necessárias instaladas:

sh


 Copiar código

```
npm install mysql2 sequelize
```

### Configurando a Conexão

Vamos configurar a conexão com o banco de dados. Supondo que já temos um arquivo `config/database.js` configurado da seguinte forma:

javascript

 Copiar código

```
const { Sequelize } = require('sequelize'); const sequelize = new Sequelize('blog', 'root', 'sua_senha', { host: 'localhost', dialect: 'mysql', port: 3306 }); module.exports = sequelize;
```

### Criando Modelos com Sequelize

Agora, vamos criar nossas tabelas usando o Sequelize. Em vez de escrever comandos SQL diretamente, vamos definir modelos que representam nossas tabelas.

1. Crie uma pasta `models` dentro da pasta da API.
2. Dentro da pasta `models`, crie um arquivo chamado `Post.js`.

No arquivo `Post.js`, vamos definir o modelo para a tabela `posts`:

javascript

 Copiar código

```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); // Definindo o modelo Post const Post = sequelize.define('Post', { title: { type: DataTypes.STRING, allowNull: false }, content: { type: DataTypes.TEXT, allowNull: true } }); module.exports = Post;
```

### Sincronizando o Modelo com o Banco de Dados

Para criar a tabela `posts` no banco de dados, precisamos sincronizar o modelo com o banco. Crie um arquivo chamado `sync.js` na raiz do projeto:

javascript

 Copiar código

```
const sequelize = require('./config/database'); const Post = require('./models/Post'); async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.'); } catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

### Executando a Sincronização

Execute o arquivo `sync.js` para sincronizar o modelo com o banco de dados:

sh

 Copiar código

```
node sync.js
```

Se tudo estiver correto, a tabela `posts` será criada no banco de dados.

Verificando a Criação da Tabela

Vamos verificar se a tabela `posts` foi criada corretamente no banco de dados. Abra seu cliente MySQL (como MySQL Workbench) e execute:

```
sql
```

Copiar código

```
DESCRIBE posts;
```


Você deve ver algo semelhante a isso:

```
sql
```

Copiar código

```
+-----+-----+-----+-----+-----+-----+ | Field | Type | Null | Key | Default | Extra | +-----+
+-----+-----+-----+-----+-----+-----+ | id | int(11) | NO | PRI | NULL | auto_increment | | title |
varchar(255) | NO | | NULL | | | content | text | YES | | NULL | | | createdAt | datetime | NO | | NULL | | | updatedAt |
datetime | NO | | NULL | | +-----+-----+-----+-----+-----+-----+
```

dia 36 video 06



Conclusão

Criamos um modelo com Sequelize e sincronizamos com o banco de dados, criando a tabela `posts`.

A partir de agora, podemos usar esse modelo para interagir com a tabela de forma simplificada e estruturada, aproveitando todos os benefícios que o Sequelize nos oferece.

Até o próximo vídeo, pessoal!

## 1.7. Nomes de Tabelas

Bom pessoal, agora vamos prestar atenção em alguns detalhes importantes ao criar nossas tabelas com o Sequelize.


### Pluralização dos Nomes de Tabelas

Por padrão, o Sequelize pluraliza o nome do modelo ao criar a tabela. Por exemplo, se o nome do modelo é `Test`, ele criará uma tabela chamada `Tests`. Vamos ver isso na prática.

### Definindo o Nome da Tabela

Para garantir que o nome da tabela seja exatamente o que desejamos, podemos usar as opções extras ao definir o modelo. Vamos criar uma tabela chamada `Test` sem pluralizar o nome.

javascript


 Copiar código

```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); const Test =
sequelize.define('Test', { coluna_teste: { type: DataTypes.STRING(50), allowNull: false } }, { tableName: 'Test' // Define
explicitamente o nome da tabela }); module.exports = Test;
```

### Sincronizando o Modelo

Agora, vamos sincronizar o modelo com o banco de dados. No arquivo `sync.js`, vamos garantir que o nome da tabela seja `Test` e não `Tests`.


javascript

 Copiar código

```
const sequelize = require('../config/database'); const Test = require('../models/Test'); async function syncDatabase() { try {
await sequelize.sync({ force: true }); // Força a recriação da tabela console.log('Banco de dados sincronizado.')} catch
(error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

Execute o arquivo `sync.js`:

sh

 Copiar código

```
node sync.js
```

### Verificando a Criação da Tabela

Vamos verificar se a tabela `Test` foi criada corretamente no banco de dados. Abra seu cliente MySQL (como MySQL Workbench) e execute:


sql

 Copiar código

```
DESCRIBE Test;
```

Você deve ver algo semelhante a isso:

sql

 Copiar código

```
+-----+-----+-----+-----+-----+-----+ | Field | Type | Null | Key | Default | Extra | +-----+
+-----+-----+-----+-----+-----+ | id | int(11) | NO | PRI | NULL | auto_increment | |
coluna_teste| varchar(50) | NO | | NULL | | | createdAt | datetime | NO | | NULL | | | updatedAt | datetime | NO | | NULL |
| +-----+-----+-----+-----+-----+-----+-----+
```

## Alterando Tabelas Existentes

Se a tabela já existe e queremos adicionar uma nova coluna ou modificar uma existente sem perder os dados, usamos o método `sync` com a opção `alter`.


javascript

 Copiar código

```
const sequelize = require('./config/database'); const Test = require('./models/Test'); async function syncDatabase() { try {
  await sequelize.sync({ alter: true }); // Altera a tabela existente sem apagar os dados console.log('Banco de dados
  sincronizado. '); } catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

Adicionando uma nova coluna:


javascript

 Copiar código

```
const Test = sequelize.define('Test', { coluna_teste: { type: DataTypes.STRING(50), allowNull: false }, coluna_nova: { //
  Nova coluna adicionada type: DataTypes.INTEGER, allowNull: true } }, { tableName: 'Test' });
```

Execute novamente o `sync.js` para sincronizar as alterações:

sh

 Copiar código

```
node sync.js
```

## Sincronizando Tabelas Individualmente

Se tivermos múltiplos modelos e queremos sincronizar apenas um específico, podemos fazer isso de forma isolada.

javascript

 Copiar código

```
const Test = require('./models/Test'); async function syncIndividualModel() { try { await Test.sync({ alter: true }); //
  Sincroniza apenas o modelo Test console.log('Modelo Test sincronizado. '); } catch (error) { console.error('Erro ao
  sincronizar o modelo Test:', error); } } syncIndividualModel();
```

dia 36 video 07



## Conclusão

Vimos como criar tabelas usando o Sequelize, definir explicitamente o nome das tabelas, alterar tabelas existentes sem perder dados e sincronizar tabelas individualmente.

Na próxima aula, vamos organizar melhor as definições dos nossos modelos e explorar o uso de classes.

Até o próximo vídeo, pessoal!

## 1.8. Estrutura do Banco de Dados

### Organizando os Models com Sequelize

Vamos iniciar organizando nossos models em arquivos separados, cada um representando uma tabela no banco de dados.

A estrutura do nosso banco de dados será baseada na imagem e no arquivo SQL que você mencionou, com seis tabelas.

Neste momento, vamos focar na criação das tabelas `users` e `tags`, que não possuem dependências diretas de outras tabelas.

### Estrutura do Banco de Dados

A estrutura que precisamos criar é a seguinte:

- `users`:
  - `id` (chave primária)
  - `name` (VARCHAR(45))
  - `createdAt` (gerenciado pelo Sequelize)
  - `updatedAt` (gerenciado pelo Sequelize)
- `tags`:
  - `id` (chave primária)
  - `name` (VARCHAR(45))
  - `createdAt` (opcional, gerenciado pelo Sequelize)
  - `updatedAt` (opcional, gerenciado pelo Sequelize)

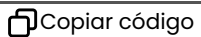
### Passo a Passo

1. Crie um diretório `models` dentro do seu projeto, se ainda não existir.
2. Dentro do diretório `models`, crie dois arquivos: `User.js` e `Tag.js`.

### Model User

No arquivo `User.js`, defina o modelo para a tabela `users`:

javascript



```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); // Definindo o modelo User
const User = sequelize.define('User', { name: { type: DataTypes.STRING(45), allowNull: false } }); module.exports = User;
```

### Model Tag

No arquivo `Tag.js`, defina o modelo para a tabela `tags`:

javascript

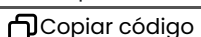


```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); // Definindo o modelo Tag
const Tag = sequelize.define('Tag', { name: { type: DataTypes.STRING(45), allowNull: true } }, { timestamps: false // Desabilitando timestamps para esta tabela }); module.exports = Tag;
```

### Sincronizando os Models

Vamos agora sincronizar esses models com o banco de dados. Crie um arquivo chamado `sync.js` na raiz do projeto:

javascript



```
const sequelize = require('../config/database'); const User = require('../models/User'); const Tag = require('../models/Tag');
async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.'); }
catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

Execute o arquivo `sync.js` para sincronizar os models com o banco de dados:

sh


 Copiar código

`node sync.js`

## Verificando a Criação das Tabelas

Abra seu cliente MySQL (como MySQL Workbench) e verifique se as tabelas foram criadas corretamente:

sql

 Copiar código

`DESCRIBE users; DESCRIBE tags;`

Você deve ver algo semelhante a isso para a tabela `users`:


sql

 Copiar código

```
+-----+-----+-----+-----+-----+-----+ | Field | Type | Null | Key | Default | Extra | +-----+
+-----+-----+-----+-----+-----+-----+ | id | int(11) | NO | PRI | NULL | auto_increment | | name |
varchar(45) | NO | | NULL | | | createdAt | datetime | NO | | NULL | | | updatedAt | datetime | NO | | NULL | | +-----+
+-----+-----+-----+-----+-----+-----+
```

E para a tabela `tags`:

sql

 Copiar código

```
+-----+-----+-----+-----+-----+-----+ | Field| Type | Null | Key | Default | Extra | +-----+-----+
+-----+-----+-----+-----+-----+-----+ | id | int(11) | NO | PRI | NULL | auto_increment | | name | varchar(45)
| YES | | NULL | | +-----+-----+-----+-----+-----+-----+
```

dia 36 video 08



## Próximos Passos

Nos próximos vídeos, continuaremos criando as outras tabelas e abordaremos os relacionamentos entre elas.



Até lá, revise os models criados e certifique-se de que tudo está funcionando corretamente.

Até o próximo vídeo!

## 1.9. Estrutura do Projeto

### Continuando a Criação dos Models

Vamos seguir adiante e organizar melhor os nossos models no Sequelize, dividindo a responsabilidade de cada um em arquivos diferentes.

No exemplo anterior, criamos os models `User` e `Tag`. Agora, vamos criar o `UserTypesModel`.

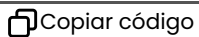
### Estrutura do Projeto

Dentro da pasta `models`, criaremos o arquivo `UserTypesModel.js` e definiremos a tabela `user_types` com as colunas `id` e `type`.

### Arquivo de Conexão

Primeiro, vamos ajustar o arquivo de conexão `config/database.js` para exportar apenas a conexão:

javascript



```
const { Sequelize } = require('sequelize'); const sequelize = new Sequelize('blog', 'root', 'sua_senha', { host: 'localhost', dialect: 'mysql', port: 3306 }); module.exports = sequelize;
```

### Definindo o Model `UserType`

Crie o arquivo `models/UserTypesModel.js` e defina o modelo para a tabela `user_types`:

javascript

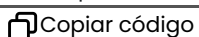


```
const { DataTypes } = require('sequelize'); const sequelize = require('../config/database'); // Definindo o modelo UserType const UserType = sequelize.define('UserType', { type: { type: DataTypes.STRING(45), allowNull: false } }, { tableName: 'user_types' // Define explicitamente o nome da tabela }); module.exports = UserType;
```

### Sincronizando os Models

Vamos agora sincronizar esses models com o banco de dados. Atualize o arquivo `sync.js` na raiz do projeto para incluir o novo model `UserType`:

javascript



```
const sequelize = require('./config/database'); const User = require('./models/User'); const Tag = require('./models/Tag'); const UserType = require('./models/UserTypesModel'); // Importando o novo modelo async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.'); } catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

### Executando a Sincronização

Execute o arquivo `sync.js` para sincronizar os models com o banco de dados:

sh



```
node sync.js
```

### Verificando a Criação da Tabela

Abra seu cliente MySQL (como MySQL Workbench) e verifique se a tabela `user_types` foi criada corretamente:


sql

 Copiar código

```
DESCRIBE user_types;
```

Você deve ver algo semelhante a isso:

sql

 Copiar código

	Field	Type	Null	Key	Default	Extra	
	id	int(11)	NO	PRI	NULL	auto_increment	type varchar(45)
	NO		NULL				

dia 36 video 09



Próximos Passos

No próximo vídeo, vamos criar mais models e configurar a sincronização das tabelas. Vamos também falar sobre relacionamentos entre tabelas e como configurá-los no Sequelize.

Até o próximo vídeo!

## 1.10. Vídeo aula

dia 36 video 10



## 1.11. Criando um Arquivo Central para Sincronização

Vamos criar um ponto central para gerenciar a criação e sincronização das nossas tabelas.

Isso garantirá que todos os models sejam corretamente carregados e sincronizados com o banco de dados.

### Criando um Arquivo Central para Sincronização

Dentro da pasta `api`, criaremos uma nova pasta chamada `db` e, dentro dela, um arquivo chamado `sync-force.js`.

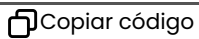
#### Estrutura do Arquivo `sync-force.js`

1. **Importar a Conexão e os Models:** Importamos a conexão do Sequelize e todos os models necessários.
2. **Sincronizar o Banco de Dados:** Chamamos a função `sync` do Sequelize para sincronizar as tabelas.

#### Passo a Passo:

1. **Criar a pasta e o arquivo:**
  - Crie a pasta `db` dentro da pasta `api`.
  - Dentro da pasta `db`, crie um arquivo chamado `sync-force.js`.
2. **Editar o Arquivo `sync-force.js`:**

javascript

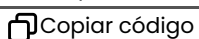


```
const sequelize = require('../config/database'); const User = require('../models/User'); const Tag = require('../models/Tag'); const UserType = require('../models/UserTypesModel'); // Importando o modelo UserType async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.')} catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

### Importando os Models para Sincronização

Para garantir que todos os models sejam carregados e reconhecidos pelo Sequelize, vamos importar os models dentro do `sync-force.js`.

javascript

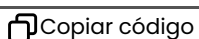


```
const sequelize = require('../config/database'); const User = require('../models/User'); const Tag = require('../models/Tag'); const UserType = require('../models/UserTypesModel'); async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.')} catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

### Executando a Sincronização

Execute o arquivo `sync-force.js` para sincronizar os models com o banco de dados:

sh



```
node api/db/sync-force.js
```

### Verificando a Criação das Tabelas

Abra seu cliente MySQL (como MySQL Workbench) e verifique se as tabelas foram criadas corretamente:

sql



```
DESCRIBE users; DESCRIBE tags; DESCRIBE user_types;
```

## Considerações Finais

Agora, sempre que precisarmos sincronizar os models com o banco de dados, podemos simplesmente executar o arquivo `sync-force.js`.

Isso garantirá que todas as tabelas sejam criadas e atualizadas conforme as definições dos nossos models.

## Próximos Passos

Vamos continuar criando e sincronizando os models restantes e, em seguida, integrar esses models na nossa API para permitir a consulta e manipulação dos dados.

Até o próximo vídeo!

## 1.12. Vídeo Aula

dia 36 video 11



## 1.13. Criando o Model Tags

### Criando o Model Tags

Agora que temos nosso model `UserTypesModel`, vamos criar o próximo model, que será ainda mais simples: `Tags`.

Vamos seguir o padrão que estamos utilizando e criar esse novo model.

### Criando o Arquivo `Tag.js`

1. Dentro da pasta `models`, crie um novo arquivo chamado `Tag.js`.

### Definindo o Model `Tag` usando Classe

Vamos definir o model `Tag` estendendo a classe `Model` do Sequelize. A documentação do Sequelize oferece uma maneira clara de fazer isso, e vamos seguir essa abordagem.

#### Passo a Passo:

1. **Importar as Dependências:** Importamos a conexão e as classes necessárias do Sequelize.
2. **Criar a Classe `Tag`:** Estendemos a classe `Model` do Sequelize e definimos as colunas.
3. **Exportar o Model:** Exportamos a classe `Tag` para que possa ser usada em outras partes do projeto.

Aqui está o código completo para o arquivo `Tag.js`:

javascript



```
const { DataTypes, Model } = require('sequelize'); const sequelize = require('../config/database'); // Importa a conexão com o banco de dados // Definindo a classe Tag que estende Model class Tag extends Model {} // Inicializando a classe Tag com suas colunas e configurações Tag.init({ name: { type: DataTypes.STRING(45), allowNull: true // Permite valores nulos } }, { sequelize, // Conexão passada para a classe modelName: 'Tag', // Nome do modelo tableName: 'tags', // Nome da tabela timestamps: false // Desabilita os timestamps `createdAt` e `updatedAt` }); module.exports = Tag;
```

### Sincronizando os Models

Vamos atualizar o arquivo `sync.js` para incluir o novo model `Tag` e sincronizar com o banco de dados.

javascript



```
const sequelize = require('../config/database'); const User = require('../models/User'); const Tag = require('../models/Tag'); const UserType = require('../models/UserTypesModel'); // Importando o modelo UserType async function syncDatabase() { try { await sequelize.sync({ force: true }); console.log('Banco de dados sincronizado.'); } catch (error) { console.error('Erro ao sincronizar o banco de dados:', error); } } syncDatabase();
```

### Executando a Sincronização

Execute o arquivo `sync.js` para sincronizar os models com o banco de dados:

sh



```
node sync.js
```

### Verificando a Criação da Tabela

Abra seu cliente MySQL (como MySQL Workbench) e verifique se a tabela `tags` foi criada corretamente:

sql






```
DESCRIBE tags;
```

Você deve ver algo semelhante a isso:

sql

 Copiar código

	Field	Type	Null	Key	Default	Extra	
	id	int(11)	NO	PRI	NULL	auto_increment	
	name	varchar(45)	YES		NULL		

Conclusão

Criamos o model `Tag` utilizando a abordagem de classes do Sequelize e sincronizamos com o banco de dados.

No próximo vídeo, continuaremos a criação dos próximos models e abordaremos a sincronização das tabelas.

Até o próximo vídeo!

## 1.14. Vídeo Aula

dia 36 video 12



## 1.15. Vídeo Aula

dia 36 video 13



## 1.16. Implementando a Função de Criação

### Criando a API para Cadastrar Informações no Banco de Dados

Agora, vamos implementar a funcionalidade de cadastro de informações no nosso banco de dados através da API.

Utilizaremos o Sequelize para interagir com o banco de dados, especificamente o método `create` para inserir novos registros.

### Implementando a Função de Criação

Vamos atualizar a função `create` em nosso controlador para usar o `Tag` model e criar novos registros no banco de dados.

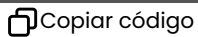
#### Passo a Passo:

1. **Importar o Model:** Importar o model `Tag` no controlador.
2. **Criar a Função de Cadastro:** Implementar a função para criar um novo registro usando o método `create` do Sequelize.
3. **Testar a API:** Utilizar o Insomnia ou Postman para testar a funcionalidade de criação.

#### Código da Função `create`:

No controlador, vamos implementar a função `create` para cadastrar uma nova tag:

javascript



```
const Tag = require('../models/Tag'); async function create(req, res) { try { const { name } = req.body; const newTag = await Tag.create({ name }); return res.status(201).json({ message: 'Tag criada com sucesso', tag: newTag }); } catch (error) { console.error('Erro ao criar a tag:', error); return res.status(500).json({ message: 'Erro ao criar a tag', error: error.message }); } } module.exports = { create };
```

### Testando a Função de Criação

Vamos utilizar o Insomnia ou Postman para enviar uma requisição `POST` e testar a criação da tag.

1. **Configurar a Requisição:**
  - **Método:** POST
  - **URL:** `http://localhost:3000/tags` (ou a rota que você configurou)
  - **Corpo da Requisição:** JSON com a estrutura da tag que queremos criar.

Exemplo de corpo da requisição:

json



```
{ "name": "CSS" }
```

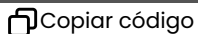
2. **Enviar a Requisição:**

- Envie a requisição e verifique a resposta. Você deve receber um status `201` e uma mensagem de sucesso.

### Verificando no Banco de Dados

Para garantir que a tag foi realmente criada, podemos fazer uma consulta no banco de dados:

sql



```
SELECT * FROM tags;
```

Você deve ver a nova tag inserida na tabela `tags`.

### Exemplo Completo de Teste no Insomnia

1. **Criar Nova Tag:**

- **Método:** POST

- **URL:** `http://localhost:3000/tags`
- **Corpo:**


json

 Copiar código

```
{ "name": "CSS" }
```

- **Resposta Esperada:**

json


 Copiar código

```
{ "message": "Tag criada com sucesso", "tag": { "id": 1, "name": "CSS", "createdAt": "2023-07-28T12:34:56.789Z",  
"updatedAt": "2023-07-28T12:34:56.789Z" } }
```

## 2. Consultar Tags Existentes:

- **Método:** GET
- **URL:** `http://localhost:3000/tags`
- **Resposta Esperada:**

json

 Copiar código

```
[ { "id": 1, "name": "CSS", "createdAt": "2023-07-28T12:34:56.789Z", "updatedAt": "2023-07-28T12:34:56.789Z" }, { "id":  
2, "name": "HTML", "createdAt": "2023-07-28T12:34:56.789Z", "updatedAt": "2023-07-28T12:34:56.789Z" } ]
```

## Conclusão

Implementamos a funcionalidade de criação de tags na API utilizando o Sequelize para interagir com o banco de dados.

Testamos a criação de novos registros utilizando o Insomnia e verificamos no banco de dados se as informações foram inseridas corretamente.

Até o próximo vídeo!