

# Node e CRUD

Usabilidade, desenvolvimento web, mobile e jogos

Prof. Me. Gustavo Torres Custódio  
gustavo.custodio@ulife.com.br

# Conteúdo

**CRUD**

**Exercícios**



Node e CRUD

CRUD

# CRUD

- CRUD é uma sigla que designa quatro operações:
  - Criação (Create);
  - Recuperação (Read);
  - Atualização (Update);
  - Remoção (Delete).

# CRUD

- Primeiro, precisamos criar um banco de dados para a aplicação.
  - Vamos criar um banco utilizando o sqlite3.
  - Ele permite criar um banco de dados dentro da pasta da própria instalação.
  - Não precisamos nos preocupar em criar e gerenciar permissões e usuários.

# CRUD

- Crie uma pasta chamada database.

- Instale o sqlite utilizando o npm:

- `npm i sqlite3`

- Execute o script abaixo:

```
var sqlite3 = require('sqlite3').verbose();  
// Cria um novo banco de dados  
var db = new sqlite3.Database('./database/Empresa.db');
```

- Observe que um novo arquivo foi criado na pasta database.

# SQL

- O que acabou de ser criado é o banco de dados que vai conter todas as informações adicionadas na aplicação.
- O banco de dados utiliza a linguagem SQL.
- As operações create, update, read e delete são realizadas por meio de instruções SQL.

# SQL

- O SQL é uma linguagem para manipular bancos de dados relacionais.
- Ela realiza operações no banco de dados por meio de queries.
  - Elas contém a operação que será realizada no banco de dados e quais tabelas e registros serão afetados.



# SQL - Tabelas

- Um banco de dados em SQL é dividido em tabelas.
  - Cada tabela possui um conjunto de linhas e colunas.
    - As linhas são itens individuais da tabela.
    - As colunas representam atributos dos itens.
  - Suponha uma tabela chamada Funcionario:

## SQL - Tabelas

ID	Nome	Idade	Endereço
1	Gustavo	30	Rua das Ruas 1
2	Guilherme	25	Rua das Ruas 3

- Cada linha representa um funcionário diferente.
- Cada coluna representa uma característica do funcionário: **nome**, **idade**, etc.

## SQL - Criando uma tabela

- Vamos criar uma tabela.

```
var sqlite3 = require('sqlite3').verbose();

var db = new sqlite3.Database('./database/Empresa.db');

// Cria a tabela funcionário mostrada anteriormente
db.run('CREATE TABLE IF NOT EXISTS Funcionario (id INT NOT NULL, nome
      VARCHAR (20) NOT NULL, idade INT NOT NULL, endereco VARCHAR (25),
      PRIMARY KEY (id) )');

console.log("Tabela Criada com sucesso");
```

- Vamos quebrar essa instrução SQL em partes...

# SQL - Operações

- O SQL possui quatro operações básicas:
  - Insert (Create);
  - Select (Read);
  - Update;
  - Delete.

# INSERT

- A operação Insert cria um ou mais registros novos no banco de dados.
  - Ela possui a sintaxe:
  - `INSERT INTO Tabela (atributo1, atributo2, ...) VALUES ('valor1', 'valor2', ...).`

# INSERT

- Utilizaremos o mesmo arquivo `formulario.js` da aula passada.
- Não esqueça de instalar o `express` para esse projeto.
  - `npm install express`

# INSERT

- Adicionamos o código para atender requisições POST.

```
// Rota POST para adicionar no banco
app.post('/insert', function(req,res){
  db.serialize(()=>{
    db.run(
      // Adicionar um novo funcionário no banco
      "INSERT INTO Funcionario (id, nome, idade, endereco) " +
        "VALUES(?, ?, ?, ?)",

      // Aqui são os valores dos ?, que correspondem ao formulário
      [req.body.id, req.body.nome, req.body.idade, req.body.endereco],
```

# INSERT

```
// Depois de rodar a instrução sql, essa função é executada
function(err) {
  // Se houver algum problema na inserção, mostre um erro
  if (err) {
    return console.log(err.message);
  }
  console.log("Novo funcionário adicionado com sucesso");
  // Se der tudo certo, ele devolve para o cliente essa mensagem
  res.send( "Novo funcionário com ID = " + req.body.id +
    " e nome = "+req.body.nome);
}
);
});
});
```



# INSERT

- Utilizamos o mesmo arquivo `formulario.html` da aula passada para enviar os dados para o *servidor*.
  - Não esqueça de mudar o `action` do formulário para `/insert`.
  - O arquivo `formulario.js` contém a rota para abrir o formulário na tela.

# SELECT

- Após inserir um elemento no banco de dados, precisamos consultá-lo para verificar se o mesmo foi inserido corretamente.
  - Para isso utilizamos a instrução SELECT.
  - Sintaxe:
    - `SELECT atributo1, atributo2, ... FROM Tabela WHERE condição.`
      - A condição do WHERE pode, por exemplo, indicar um funcionário com um ID específico.
      - `SELECT * ... FROM Tabela` seleciona todas as colunas.

# SELECT

- Adicionamos um POST para receber o número de usuário buscado.

```
app.post('/select', function(req, res){  
  db.serialize(()=>{  
    db.each('SELECT id, nome FROM Funcionario WHERE id = ?',  
      // encontre o funcionario com o id preenchido  
      [req.body.id],
```

# SELECT

```
// A função recebe um row que corresponde a uma linha da tabela
function(err, row){
  // row corresponde a linha encontrada
  // Erro caso o funcionário não seja encontrado
  if(err){
    res.send("Erro ao encontrar funcionário");
    return console.error(err.message);
  }
  // Envia para o cliente o nome e o id do funcionário caso ele
  // seja encontrado
  res.send(`<p>Id: ${row.id}</p> <p> Nome: ${row.nome}</p><hr>`);
  console.log("Funcionário encontrado");
});
});
});
```

## SELECT

- Vamos criar um formulário para buscar o usuário pelo seu id.
  - Podemos adicionar no mesmo arquivo formulario.html.

```
<form action="/select" method="POST">
  <fieldset id="dados">
    <legend>Busca</legend>
    <p>
      <label for="id">Identificação:</label>
      <input type="number" name="id" id="id">
    </p>
  </fieldset>
  <button type="submit">OK</button>
</form>
```

- Busque o número 1 nesse formulário.

Id: 1

Nome: Gustavo

---

## Busca

- Utilizar o `db.each` retorna apenas um resultado.
- Se quisermos retornar múltiplos resultados, utilizamos o `db.all`.
- Vamos adicionar uma requisição GET para uma página que lista todos os funcionários.

# Busca

```
// Lista todos os funcionários na tabela
app.get('/showall', function(req,res){
  db.serialize(()=>{
    //db.all() inclui todos os funcionarios
    db.all('SELECT id, nome FROM Funcionario',
```



# Busca

```
function(err, rows){
  if(err){
    res.send("Erro ao encontrar funcionario");
    return console.error(err.message);
  }
  var resultado = "";
  // Itera sobre todos os resultados retornados
  for (var row of rows) {
    // Coloca o nome e id dos funcionários em parágrafos.
    // Concatena nome e id dos funcionários na variável resultado.
    resultado += `<p>Id: ${row.id}</p> <p> Nome: ${row.nome}</p><hr>`;
  }

  // No final, a variável resultado contendo conteúdo
  // sobre todos os funcionários é retornada
  res.send(resultado);
  console.log("Funcionarios encontrados");
});
```

# Busca

Id: 1

Nome: Gustavo

---

Id: 2

Nome: Joao

---

Id: 3

Nome: Gustavo

---

## Resumo

- Criamos um servidor em node.
  - O servidor espera o acesso na respectiva porta alocada.
  - Conseguimos inserir e consultar informações de usuário.
  - As próximas operações a serem vistas são o UPDATE e o DELETE.



Node e CRUD

## Exercícios

## Exercício 1

- Crie um site em HTML que utilize a tabela Funcionário e que possui três páginas:
  - Home;
  - Busca;
  - Cadastro.
- A Home possui um menu que direciona o usuário para as outras páginas.
- A página de busca deve permitir buscar o usuário por nome ou id.
  - Em caso de busca por nome, use o SQL com a cláusula **LIKE**.

## Referências



Paul, S. (2020).

Read html form data using get and post method in node.js.

<https://medium.com/swlh/read-html-form-data-using-get-and-post-method-in-node-js-8d2c7880adbf>.

Obrigado

[gustavo.custodio@ulife.com.br](mailto:gustavo.custodio@ulife.com.br)