

Aula 07:

Conexão ao banco de dados com Mongoose

Common JS x ES6 Modules



A partir de agora para trabalharmos com **módulos** utilizaremos o modo de importação do **EcmaScript 6 (ES6)**. Ele difere do modo de importação de módulos do **CommonJS** que é nativo do Node. Veja a diferença:

Common JS :

```
const express = require("express") // Importando o Express
const app = express() // Iniciando o Express
```

ES6 Modules:

```
import express from 'express' // Importando o Express
const app = express() // Iniciando o Express
```

JS

ES6 Modules



Para que a mudança funcione, precisamos informar no arquivo **package.json** do projeto que estaremos utilizando os módulos do ES6, conforme a seguir:

```
{  
  "name": "loja",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "ejs": "^3.1.9",  
    "express": "^4.18.2",  
    "nodemon": "^3.0.1"  
  },  
  "type": "module"  
}
```

Incluir essa linha no arquivo package.json do projeto:

JS

Criando um banco no MongoDB

Antes de começarmos a criar os arquivos para a manipulação do banco de dados, o MongoDB já deve estar instalado em seu sistema e um banco de dados deve ser criado.

Link para download do MongoDB:

https://fastdl.mongodb.org/windows/mongodb-windows-x86_64-7.0.2-signed.msi

Link para download somente da GUI MongoDB Compass:

<https://www.mongodb.com/try/download/compass>

Após a instalação, basta abrir o **MongoDB Compass** e abrir um conexão:

New Connection

Connect to a MongoDB deployment

FAVORITE

URI ⓘ Edit Connection String ☒

mongodb://localhost:27017/

> Advanced Connection Options

Save Save & Connect Connect



JS

Criando um banco no MongoDB

Feito isso, crie um novo banco conforme a seguir:

The image shows the MongoDB Compass interface. On the left, the 'Databases' tab is selected, showing a list of databases: MongoTeste, admin, agendamento, config, and local. A red box highlights a '+' icon next to the 'Databases' tab, with a red arrow pointing to the 'Create Database' dialog box on the right.

The 'Create Database' dialog box has the following fields and options:

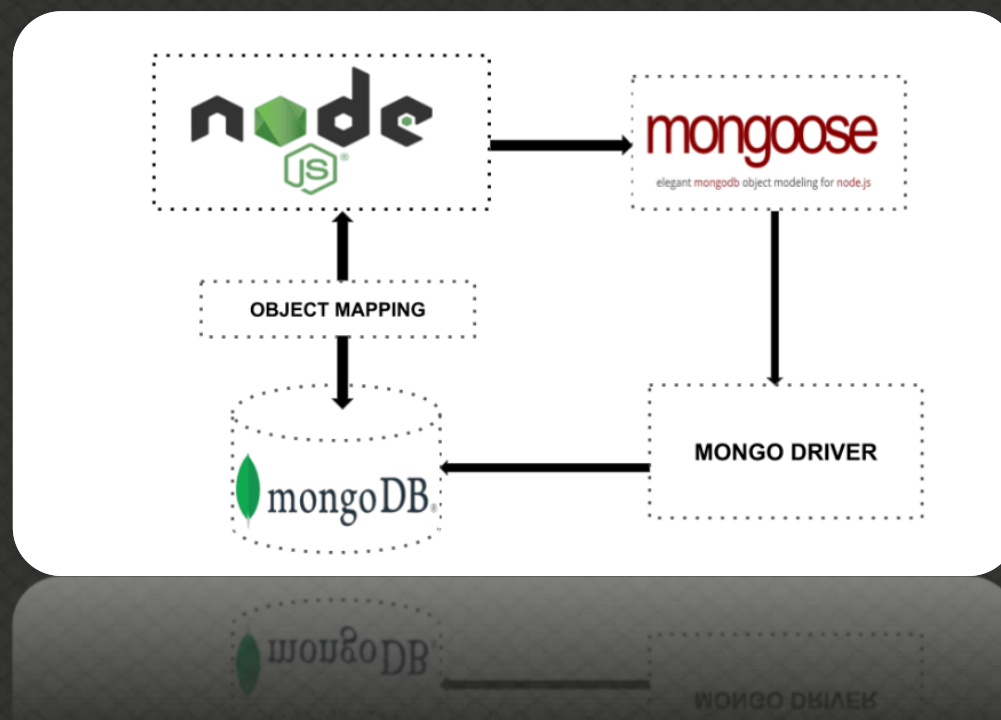
- Database Name:** A text input field containing 'loja'.
- Collection Name:** A text input field containing 'teste'.
- ☐ **Time-Series**
Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)
- Additional preferences** (e.g. Custom collation, Capped, Clustered collections)
- Buttons:** 'Cancel' and 'Create Database' (highlighted with a red box).



JS

Conhecendo a biblioteca Mongoose

Mongoose é uma biblioteca de Modelagem de Dados de Objeto (ou ODM, do inglês: Object Data Modeling) para **MongoDB** e Node.js. Ele gerencia o relacionamento entre dados, fornece a validação de esquemas e é usado como tradutor entre objetos no código e a representação desses objetos no MongoDB.



JS

Instalando e importando o Mongoose



Para instalar o Mongoose no seu projeto utilize o comando: **npm install mongoose**

```
PS C:\Users\Prof. Diego\OneDrive - Etec Centro Paula Souza\Desktop\SistemaLoja> npm install mongoose
added 25 packages, and audited 129 packages in 4s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\Prof. Diego\OneDrive - Etec Centro Paula Souza\Desktop\SistemaLoja> █
```

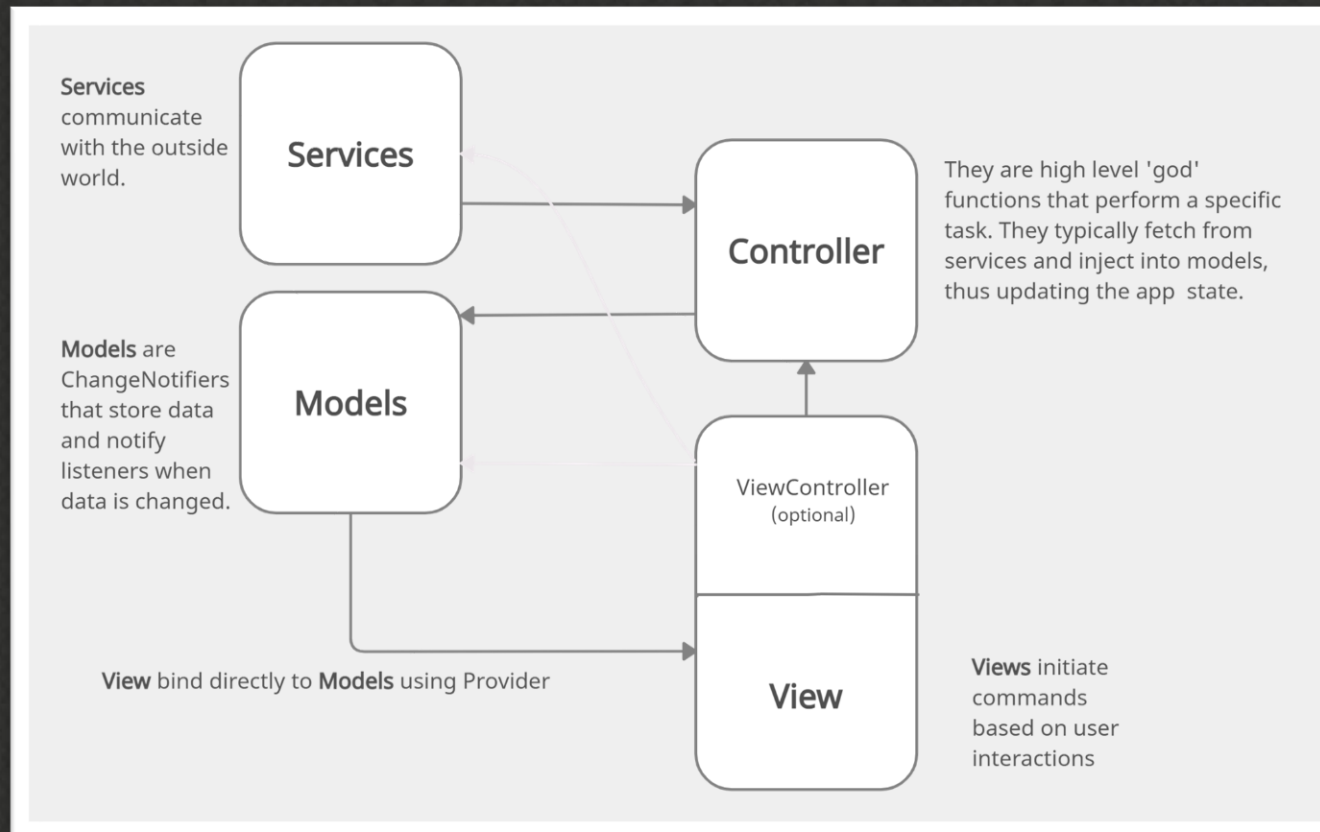
Com o Mongoose instalado no seu projeto, faremos sua importação agora no arquivo **index.js**, conforme a seguir:

```
import express from 'express'
const app = express()
import mongoose from 'mongoose'
```

JS

Criando um Model

Na arquitetura **MVC (Model-View-Controller)**, o Model é responsável pela leitura e escrita de dados, e também de suas validações. Um model pode ser uma representação abstrata de coisas do mundo real. Exemplo: Pessoa, Cachorro, Gato, etc.



JS

Criando um Model

Se tivermos fazendo um sistema de pedido, vamos ter o model **Cliente** que representa nosso cliente, dentro dele podemos ter as informações: Nome, CPF, Endereço, etc

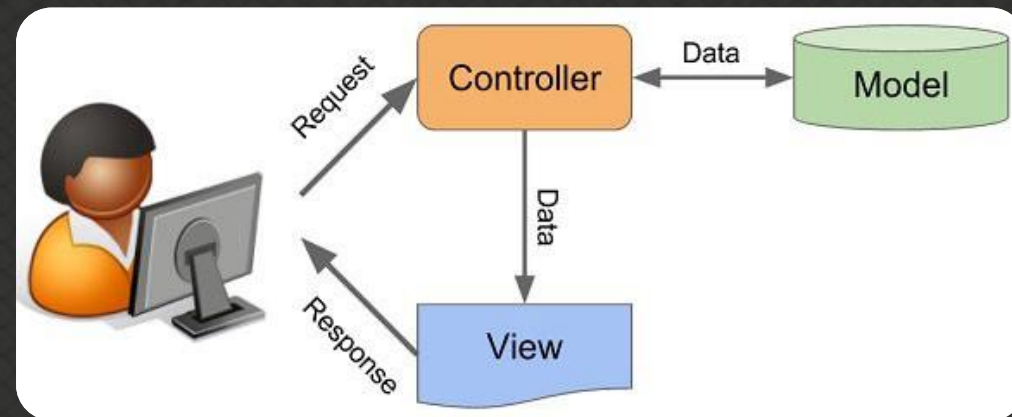
Em resumo:

Models: manipulação dos dados;

Views: interação do usuário;

Controllers: camada de controle;

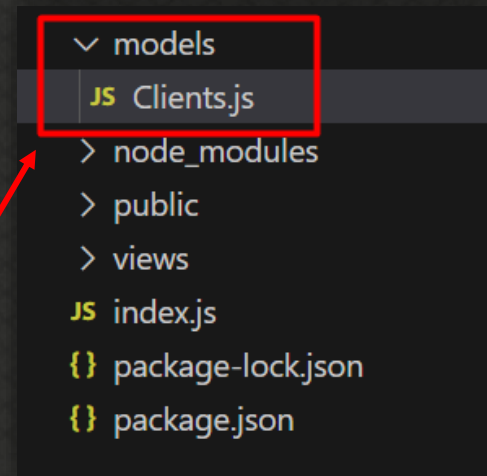
Services: responsáveis pela lógica de negócio, além de ser responsável por se comunicar com as camadas mais internas, como por exemplo, uma camada de Dados.



JS

Criando um Model

Para criarmos nosso primeiro Model, iremos criar uma pasta com o nome “**models**” dentro da pasta do projeto. E em seguida criar o arquivo do nosso model dentro desta pasta com o nome “**Clients.js**”.



No arquivo **Clients.js**:

1 - Fazemos a importação da biblioteca Mongoose;

```
import mongoose from 'mongoose'
```

2 - Criamos um novo **Schema*** que será armazenado na constante **client**;

```
const client = new mongoose.Schema({  
  name: String,  
  cpf: String,  
  adress: String,  
})
```

3 - Fazemos a exportação de **client**;

```
export default client
```

* Um Schema define a estrutura e o conteúdo dos seus dados. Schemas são a especificação do modelo de dados do seu aplicativo.



JS

Criando um Model

Os seguintes tipos de esquemas são permitidos:

- Array
- Boolean (ou booleano, em português)
- Buffer
- Date (ou formato de data, em português)
- Mixed (um tipo genérico/flexível de dados)
- Number (ou numérico, em português)
- ObjectId
- String



JS

Criando a conexão com o banco de dados

Com nosso primeiro model criado, para criarmos a conexão com o banco de dados do MongoDB, iremos incluir a seguinte linha no nosso arquivo **index.js**:

```
import express from 'express' // Importando o Express
const app = express() // Iniciando o Express
import mongoose from 'mongoose' // Importando o Mongoose
```

```
mongoose.connect("mongodb://127.0.0.1:27017/loja", {useNewUrlParser: true, useUnifiedTopology: true})
```

Método do mongoose para se conectar ao banco

URL do banco

Nome do banco

Usar novo modelo de URL do Mongo

Relacionado a funções de monitoramento do banco



JS

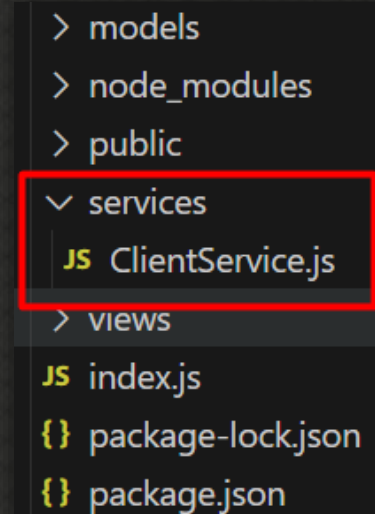
Criando o service de Cliente

Para criarmos nosso service de Cliente, criaremos uma pasta com o nome “**services**” dentro da pasta do projeto. E após isso, criaremos um arquivo com o nome “**ClientService.js**” →

O service de Cliente será responsável por conter os métodos para **cadastrar**, **consultar**, **alterar** e **excluir** um cliente no banco de dados.

Com o arquivo criado, precisamos importar nosso **Model** e a biblioteca **Mongoose**, conforme abaixo:

```
services > JS ClientService.js > ...  
import client from "../models/Clients.js"  
import mongoose from "mongoose"
```



```
> models  
> node_modules  
> public  
▼ services  
  JS ClientService.js  
> views  
JS index.js  
{ } package-lock.json  
{ } package.json
```



JS

Service de Cliente

Após isso, no nosso arquivo de service de cliente iremos iniciar o nosso model conforme ao lado:

Em seguida, criaremos uma **classe** que será responsável por conter os nossos métodos para manipulações no banco e faremos sua exportação:

```
import client from "../models/Clients.js"
import mongoose from "mongoose"

const Client = mongoose.model("Client", client)

class ClientService {
}

export default new ClientService()
```

```
import client from "../models/Clients.js"
import mongoose from "mongoose"
```

```
const Client = mongoose.model("Client", client)
```

Método do mongoose para iniciar um model

Nome da coleção no MongoDB (será alterado para o plural "clients" no banco)

Schema que será utilizado, no caso o schema cliente criado anteriormente no nosso arquivo de model.



JS

Service de Cliente

Agora já podemos importar a classe **ClientService** do nosso arquivo **ClientService.js** no nosso arquivo **index.js**, conforme abaixo:

```
> JS index.js > ...
import express from 'express' // Importando o Express
import mongoose from 'mongoose' // Importando o Mongoose
import ClientService from './services/ClientService.js' // Importando o service de
cliente
const app = express() // Iniciando o Express

// Iniciando conexão com o banco de dados do MongoDB
mongoose.connect("mongodb://127.0.0.1:27017/loja1", {useNewUrlParser: true,
useUnifiedTopology: true})
```

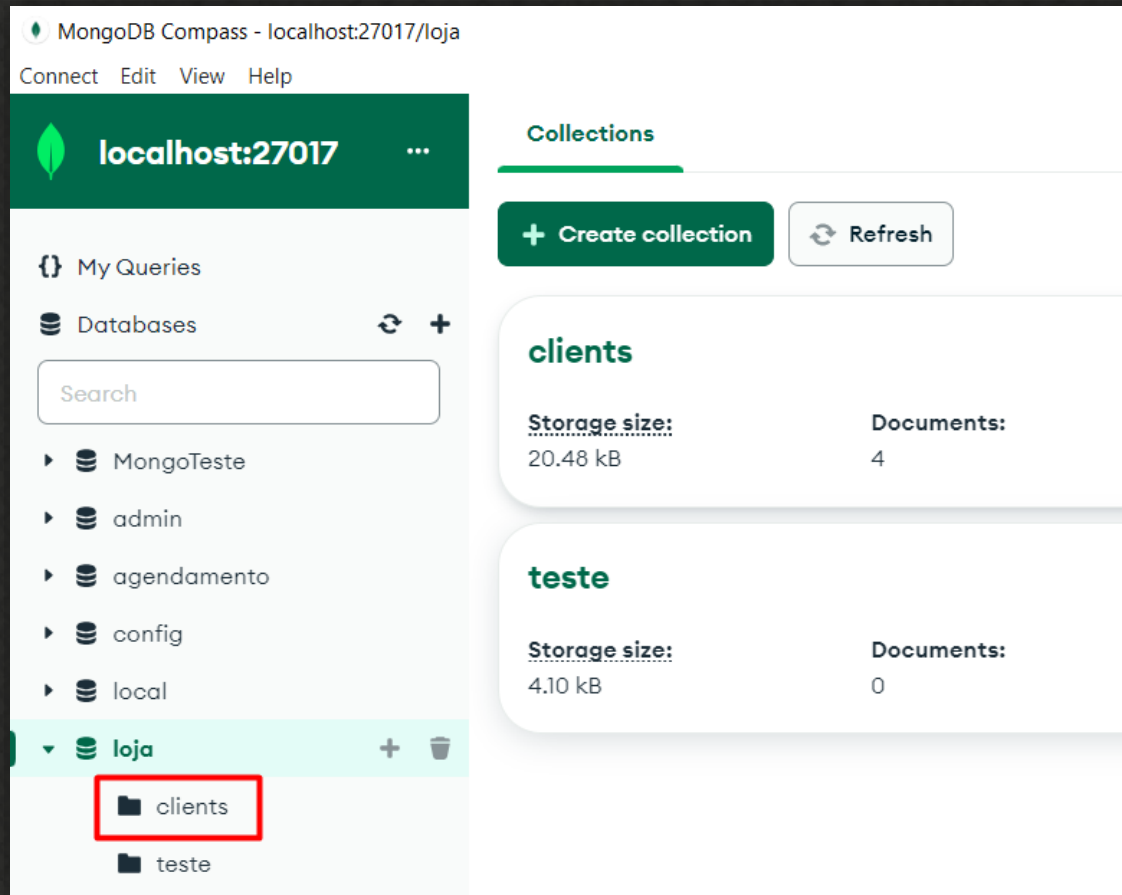
Realizado esses passos, já é o suficiente para que nossa coleção “**Client**” que criamos no **model**, seja criada no MongoDB. Porém é importante observar que as coleções sempre têm o seu nome alterado para o plural, ou seja, no MongoDB será criado a coleção “**clients**”.



JS

Criando a primeira coleção no banco

Podemos então reiniciar o servidor do Node e checar no **MongoDB Compass** se a coleção “**clients**” foi criada com sucesso. Com a coleção criada, já podemos começar a gravar dados no banco.



JS

Cadastrando clientes no banco

Voltando ao nosso arquivo **ClientService.js**, em nossa classe **ClientService**, criaremos uma função **assíncrona* com o nome “**Create**”, que será utilizado para salvar dados no banco:

```
class ClientService {  
  async Create(name, cpf, address) {  
    const newClient = new Client({  
      name,  
      cpf,  
      address  
    })  
    try {  
      await newClient.save()  
      return true  
    } catch (err) {  
      console.log(err)  
      return false  
    }  
  }  
}
```

** Uma função **assíncrona** permite que seu programa inicie uma tarefa potencialmente de longa duração e ainda seja capaz de responder a outros eventos enquanto essa tarefa é executada, em vez de ter que esperar até que essa tarefa seja concluída. Ou seja, essa função é executada de forma independente do restante do código.*

O método **.save()** do Mongoose é usado para salvar dados no banco. Antes dele utilizamos o operador **await**. Esse operador é utilizado para esperar por uma **Promise**. O **await** só pode ser usado dentro de uma função **async**. Uma Promise por sua vez representa a eventual conclusão (ou falha) de uma operação **assíncrona** e seu valor resultante.



JS

Formulário de Cadastro

Nossa view **clientes.ejs**, já está preparada e possui um formulário de cadastro com os campos **name**, **cpf** e **adress**, exatamente a estrutura que montamos em nosso **model**. Essa view pode ser acessada através da rota **/clientes**.

Além disso, estamos utilizando o método **POST** no formulário para fazer o envio dos dados, e seu **action** está definido como **"createClient"**.

Essa será a rota que os dados serão enviados para serem gravados no banco e ainda iremos cria-la.

```
views > clientes.ejs > ? > ?  
<%- include ('partials/header.ejs') %>  
<%- include ('partials/navbar.ejs') %>  
<div class="container">  
  <div class="card">  
    <div class="card-header">  
      <h1>Cadastrar cliente</h1>  
    <hr>  
    <form method="POST" action="createClient">  
      <label>Nome do cliente:</label>  
      <input type="text" name="name" id="name" placeholder="Insira o nome do cliente" class="form-control">  
  
      <label>CPF do cliente:</label>  
      <input type="text" name="cpf" id="cpf" placeholder="Insira o CPF do cliente" class="form-control cpf">  
  
      <label>Endereço do cliente:</label>  
      <input type="text" name="adress" id="adress" placeholder="Insira o endereço do cliente" class="form-control">  
      <br>  
      <button class="btn btn-success">Cadastrar</button>  
    </form>  
  </div>  
</div>  
<br>
```

Cadastrar cliente

Nome do cliente:

CPF do cliente:

Endereço do cliente:

Cadastrar

Capturando dados do formulário

Para fazermos a captura dos dados enviados através do formulário iremos utilizar a biblioteca **Body-Parser**.

A biblioteca Body-Parser será responsável por traduzir os dados enviados através de formulários em uma estrutura Javascript (inclusive json) que possamos utilizar em nosso back-end.

O primeiro passo é fazer sua instalação através do terminal em nosso projeto. Para isso, basta executar o comando **npm install body-parser**.

Com a biblioteca instalada iremos importa-la e configura-la em nosso arquivo **index.js**.

```
import bodyParser from 'body-parser' //Importando a biblioteca body-parser
```

```
app.use(bodyParser.urlencoded({extended: false}))
```

```
app.use(bodyParser.json())
```

Permite receber dados de formulários via json. (opcional)

BODY 
PARSER

```
npm install body-parser
```

Decodifica os dados recebidos via formulários em uma estrutura Javascript.



JS

Criando a rota de criação de clientes

Com o body-parser já configurado, iremos criar agora a rota **/CreateClient**, responsável por receber os dados do formulário e chamar o método para gravar esses dados no banco. Esse método é o **Create()** que criamos na classe **ClientService**, anteriormente.

Para isso, devemos então incluir as seguintes linhas em nosso arquivo **index.js**:

Utiliza o verbo post, ou seja, será uma rota do tipo post, para envio de dados.

Após o método **Create** gravar os dados no banco, redireciona o usuário novamente para a rota **/clientes**

```
// ROTA DE CRIAÇÃO DE CLIENTES
app.post("/createClient", async (req, res) => {
  var status = await ClientService.Create(
    req.body.name,
    req.body.cpf,
    req.body.adress
  )
  if(status) {
    res.redirect("/clientes")
  }else{
    res.send("Ocorreu uma falha!")
  }
})
```

Passa como parâmetro para o método **Create**, os dados **name**, **cpf** e **adress**, recebidos através do formulário (**req.body**)



JS

Cadastrando Clientes

Agora basta irmos até o nosso formulário, tentar realizar um cadastro e verificar se os dados estão sendo gravados no banco com sucesso. Se necessário, reinicie o servidor antes.

Cadastrar cliente

Nome do cliente:

José Pereira

CPF do cliente:



555.444.333.22

Endereço do cliente:

Rua Cinco, nº 90 - Jardim Primavera

Cadastrar

Documents Aggregations Schema Indexes Validation

Filter  Type a query: { field: 'value' } or [Generate query](#) 

[+ ADD DATA](#) [EXPORT DATA](#)

```
{
  "_id": ObjectId('653ab080d0976ba37bebbba31')
  "name": "José Pereira"
  "cpf": "555.444.333.22"
  "adress": "Rua Cinco, nº 90 - Jardim Primavera"
  "__v": 0
}
```



JS

Consultando e exibindo dados

Agora que já estamos cadastrando os dados dos clientes no banco, iremos exibir esses dados na view de clientes. O primeiro passo é criarmos uma nova função na classe **ClientService** que fará a consulta no banco de dados. Para isso basta adicionar as seguintes linhas:

```
class ClientService {  
  async Create(name, cpf, adress) {  
    const newClient = new Client({  
      name,  
      cpf,  
      adress  
    })  
    try {  
      await newClient.save()  
      return true  
    } catch (err) {  
      console.log(err)  
      return false  
    }  
  }  
  
  async GetAll() {  
    const clients = await Client.find()  
    return clients  
  }  
}
```

O método **.find()** do Mongoose é usado para buscar dados no banco. Antes dele utilizamos o operador **await**. Esse operador é utilizado para esperar por uma **Promise**. O **await** só pode ser usado dentro de uma função **async**. Uma Promise por sua vez representa a eventual conclusão (ou falha) de uma operação **assíncrona** e seu valor resultante.

O resultado da busca será passado a variável **clients**, que por sua vez, será o retorno da função.



JS

Consultando e exibindo dados

Com nosso método de consulta criado, devemos chama-lo em nossa rota **/clientes**, para que quando a página for renderizada, automaticamente receba as informações dos clientes e as exiba na página. Para isso, inclua as seguintes linhas no arquivo **index.js**:

```
// ROTA CLIENTES
app.get("/clientes", async (req,res) => {
  var clients = await ClientService.GetAll()
  res.render("clientes", {
    clients: clients
  })
})
```



JS

Exibindo dados do banco na página

Com os dados sendo coletados do banco e sendo repassado a nossa view de clientes, basta que exibamos esses dados agora em uma tabela, logo abaixo o formulário de cadastro.

Sendo assim, a cada novo cadastro, esses dados já serão buscados e exibidos em seguida na página.

```
<div class="container">
  <div class="card">
    <div class="card-header">
      <h1>Lista de clientes</h1>
    </div>
    <div class="card-body">
      <table>
        <thead>
          <tr>
            <th>Nome:</th>
            <th>CPF:</th>
            <th>Endereço:</th>
          </tr>
        </thead>
        <tbody>
          <% clients.forEach(client => { %>
            <tr>
              <td><%= client.name %></td>
              <td><%= client.cpf %></td>
              <td><%= client.adress %></td>
            </tr>
          <% }) %>
        </tbody>
      </table>
    </div>
  </div>
</div>
```



JS

Exibindo dados do banco na página



E assim será o resultado final de cadastro e busca de dados no banco de dados:

Cadastrar cliente

Nome do cliente:

Maria dos Santos

CPF do cliente:

111.222.333.444.55

Endereço do cliente:

Rua Seis, nº 44 - Centro

Cadastrar

Cadastrar cliente

Nome do cliente:

Insira o nome do cliente

CPF do cliente:

Insira o CPF do cliente

Endereço do cliente:

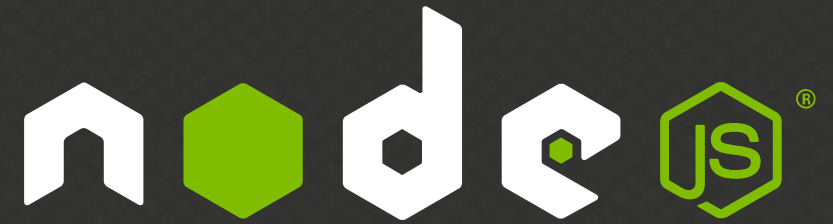
Insira o endereço do cliente

Cadastrar

Clientes

Nome:	CPF:	Endereço:
José Pereira	555.444.333.22	Rua Cinco, nº 90 - Jardim Primavera
Maria dos Santos	111.222.333.444.55	Rua Seis, nº 44 - Centro

JS



Aula 07:

Conexão ao banco de dados com Mongoose