NODE.JS

Tutorial CRUD em Node.js com driver nativo do MongoDB

Escrito por **Luiz Duarte**
em 18/03/2023

JUNTE-SE A MAIS DE 34 MIL DEVS

Entre para minha lista e receba conteúdos exclusivos e com prioridade

Seu email

Enviar

Node.js é uma plataforma que permite a você construir aplicações server-side em JavaScript, já falei disso extensivamente neste artigo.

MongoDB é um banco de dados NoSQL, orientado a documentos JSON(-like), o mesmo formato de dados usado no JavaScript, além de ser utilizado através de funções com sintaxe muito similar ao JS também, diferente dos bancos SQL tradicionais e também já falei disso extensivamente neste artigo.

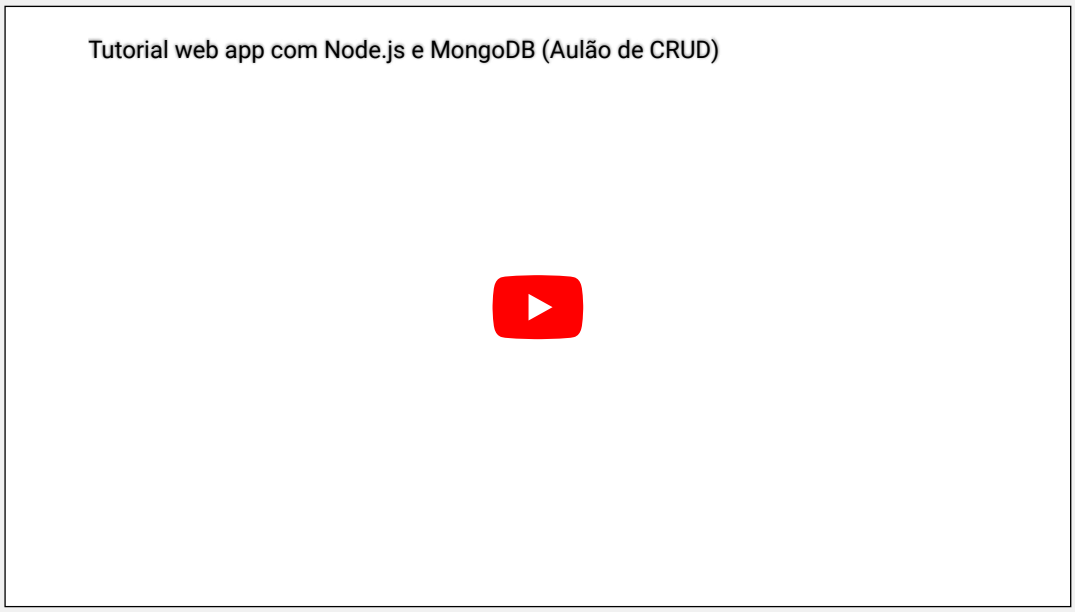
Vê algo em comum entre os dois? Pois é, ambos tem o JavaScript em comum e isso facilita bastante o uso destas duas tecnologias em

conjunto, motivo de ser uma dupla bem frequente para projetos de todos os tamanhos.

A ideia deste tutorial é justamente lhe dar um primeiro contato prático com ambos, fazendo o famoso CRUD (Create, Retrieve, Update e Delete).

Neste artigo você vai ver:

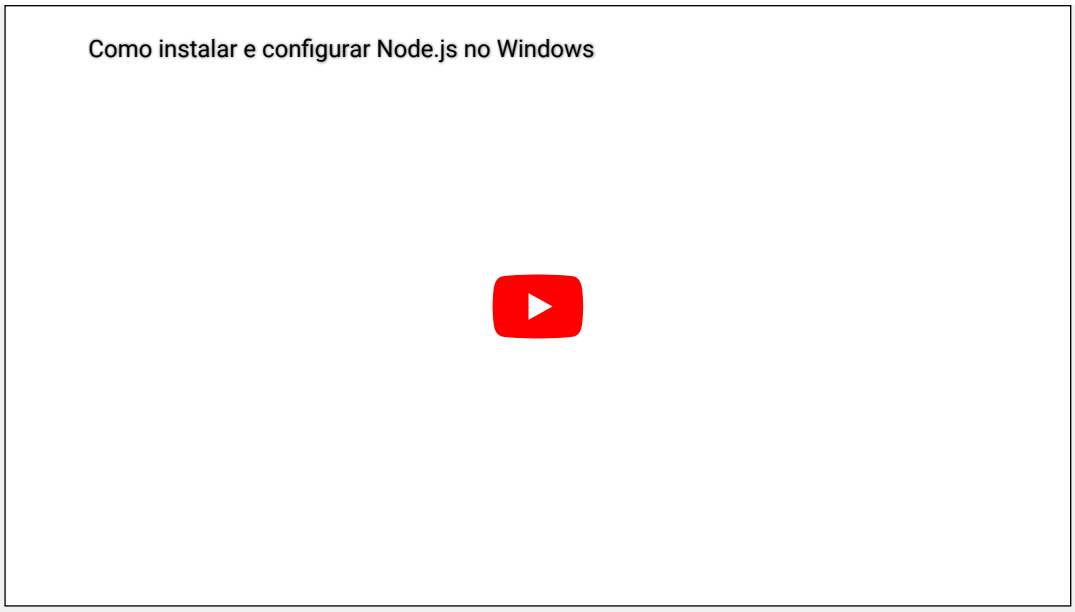
Se preferir, você pode aprender o mesmo conteúdo no vídeo abaixo.



Vamos lá!

#1 – Configurando o Node.js

Vamos começar instalando o Node.js: apenas clique no link do site oficial e depois no grande botão verde para baixar o executável certo para o seu sistema operacional (recomendo a versão LTS). Esse executável irá instalar o Node.js e o NPM, que é o gerenciador de pacotes do Node. Caso tenha dificuldade, o vídeo abaixo pode ajudar.



Uma vez com o NPM instalado, vamos instalar o módulo que será útil mais pra frente. Crie uma pasta para guardar os seus projetos Node no computador (recomendo C:\node) e dentro dela, via terminal de comando com permissão de administrador (sudo no Mac/Linux), rode o comando abaixo:

```
npm install -g express-generator
```

O Express é o web framework mais famoso da atualidade para Node.js. Com ele você consegue criar aplicações e APIs web muito rápida e facilmente. O que instalamos é o express-generator, um gerador de aplicação de exemplo, que podemos usar via linha de comando, da seguinte maneira:

```
express -e --git node-mongo-crud
```

O "-e" é para usar a view-engine (motor de renderização) EJS, ao invés do tradicional Jade/Pug. Já o "–git" deixa seu projeto preparado para versionamento com Git. Aperte Enter e o projeto será criado (talvez ele peça uma confirmação, apenas digite 'y' e confirme).

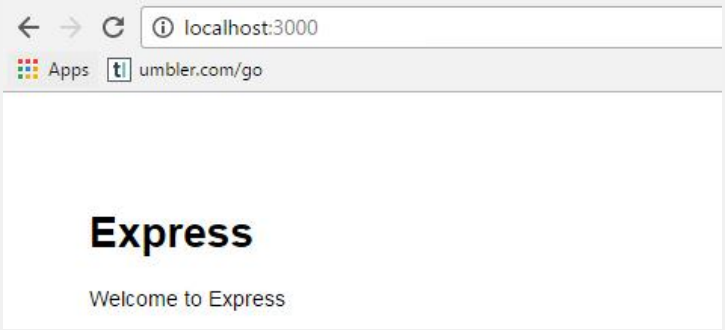
Depois entre na pasta e mande instalar as dependências com npm install:

```
cd node-mongo-crud
npm install
```

Ainda no terminal de linha de comando e, dentro da pasta do projeto, digite:

```
npm start
```

Isso vai fazer com que a aplicação default inicie sua execução em localhost:3000, que você pode acessar pelo seu navegador.



Happy Hello World!

#2 – Configurando o MongoDB

Vamos agora acessar o site oficial do MongoDB e baixar o Mongo. Clique no menu superior em Products > MongoDB Community Edition > MongoDB Community Server e busque a versão mais recente para o seu sistema operacional. Baixe o arquivo e, no caso do Windows, rode o executável que extrairá os arquivos na sua pasta de Arquivos de Programas, o que é está ok para a maioria dos casos. Coloquei esse passo a passo de instalação e configuração no vídeo abaixo, se preferir.

Como instalar e executar MongoDB (Server)

Dentro da pasta do seu projeto Node, que aqui chamei de workshoptdc, deve existir uma subpasta de nome data, crie ela agora. Nesta pasta vamos armazenar nossos dados do MongoDB.

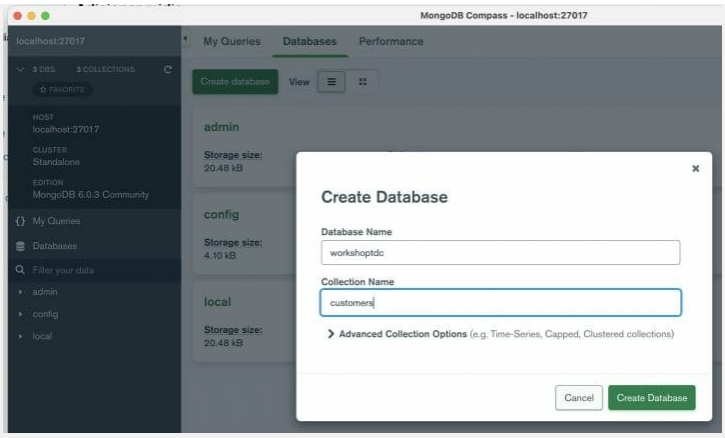
Pelo prompt de comando, entre na subpasta bin dentro da pasta de instalação do seu MongoDB e digite (no caso de Mac e Linux, coloque um ./ antes do mongod e ajuste o caminho da pasta data de acordo):

```
mongod --dbpath c:\node-mongo-crud\data
```

Isso irá iniciar o servidor do Mongo. Se não der nenhum erro no terminal, o MongoDB está pronto e está executando corretamente.

Agora o cliente que eu recomendo é o MongoDB Compass, que você também baixa pelo site oficial em Products > Tools > MongoDb Compass, sendo que o vídeo abaixo ilustra bem como funciona a ferramenta, caso queira um rápido overview.

Após a conexão funcionar do MongoDb Compass com o seu servidor de MongoDB, você deve criar uma nova database indo no menu lateral e escolhendo a opção Databaes > Create Database. Vou usar como nome da base workshoptdc e já vou aproveitar e informar o nome da coleção principal que é customers (clientes em inglês).

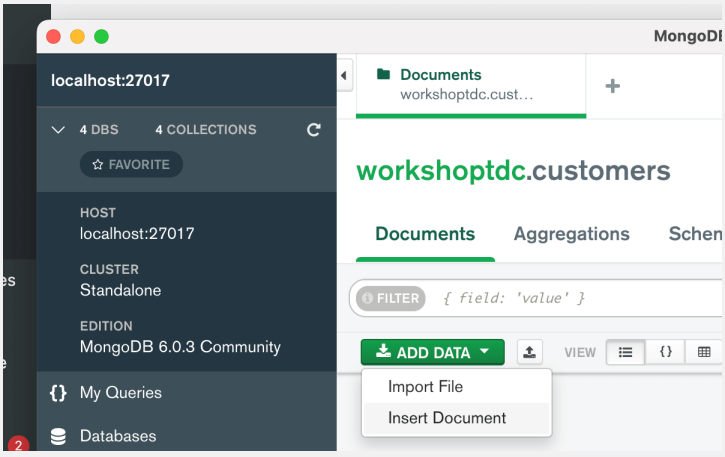


Uma de minhas coisas favoritas sobre MongoDB é que ele usa JSON como estrutura de dados, o que significa curva de aprendizagem zero para quem já conhece o padrão. Caso não seja o seu caso, terá que buscar algum tutorial de JSON na Internet antes de prosseguir.

Vamos adicionar um registro à nossa coleção (o equivalente do Mongo às tabelas do SQL). Para este tutorial teremos apenas uma base de customers (clientes), sendo o nosso formato de dados como abaixo:

```
{
"_id" : ObjectId("1234"),
"name" : "luiztools",
"age" : 32
}
```

O atributo _id pode ser omitido, neste caso o próprio Mongo gera um id pra você. No MongoDb Compass, entre na base workshoptdc e depois na coleção customers para ter acesso à opção “Add Data” e em seguida “Insert Document”.



Na tela que vai se abrir, adicione um JSON de cliente com o nome de Luiz e a idade de 34. Ou os dados que julgar melhores, sendo que o _id pode ser omitido pois vai ser gerado dinamicamente.

```
{
"name": "Luiz",
"age": 34
}
```

Após inserir, automaticamente a listagem da ferramenta se atualiza mostrando o registro recém inserido. Adicione alguns registros para ter uma base mínima para o desenvolvimento e agora sim, vamos interagir de verdade com o web server + MongoDB.

#3 – Conectando no MongoDB com Node

Agora sim vamos juntar as duas tecnologias-alvo deste post!

Precisamos adicionar uma dependência para que o MongoDB funcione com essa aplicação usando o driver nativo. Usaremos o NPM via linha de comando de novo:

```
npm install mongodb dotenv
```

Com isso, duas dependências novas serão baixadas para sua pasta node_modules e duas novas linhas de dependências serão adicionadas no package.json para dar suporte a MongoDB e a variáveis de ambiente (com DotEnv). As variáveis de ambiente precisam ser a primeira coisa a serem carregadas quando a aplicação subir. Então vá no arquivo bin/www e adicione essa linha bem no topo.

```
require("dotenv").config();
```

Essa instrução irá procurar um arquivo de variáveis de ambiente na raiz do seu projeto, que você deve criar com o nome .env e o seguinte conteúdo.

```
MONGO_HOST=mongodb://127.0.0.1:27017
MONGO_DATABASE=workshoptdc
```

A variável MONGO_HOST é o endereço do seu servidor de MongoDB (use IP, nunca localhost), enquanto que a variável MONGO_DATABASE é o nome da sua base de dados, então altere conforme julgar necessário.

Agora, para organizar nosso acesso à dados, vamos criar um novo arquivo chamado db.js na raiz da nossa aplicação Express (workshoptdc). Esse arquivo será o responsável pela conexão e

manipulação do nosso banco de dados, usando o driver nativo do MongoDB. Adicione estas linhas:

```
const { MongoClient, ObjectId } = require("mongodb");

let singleton;

async function connect() {
    if (singleton) return singleton;

    const client = new MongoClient(process.env.MONGO_HOST);
    await client.connect();

    singleton = client.db(process.env.MONGO_DATABASE);
    return singleton;
}
```

Estas linhas carregam um objeto cliente de conexão a partir do módulo 'mongodb' e depois fazem uma conexão em nosso banco de dados. Essa conexão é armazenada em uma variável global do módulo. A última linha ignore por enquanto, usaremos ela mais tarde.

A seguir, vamos modificar a nossa rota para que ela mostre dados vindos do banco de dados, usando esse db.js que acabamos de criar.

Para conseguirmos fazer uma listagem de clientes, o primeiro passo é ter uma função que retorne todos os clientes em nosso módulo db.js (arquivos JS que criamos são chamados de módulos se possuírem um module.exports no final), assim, adicione a seguinte função ao seu db.js (substituindo aquela linha do module.exports que tinha antes):

```
const COLLECTION = "customers";

async function findAll() {
    const db = await connect();
    return db.collection(COLLECTION).find().toArray();
}

module.exports = { findAll }
```

A consulta aqui é bem direta: usamos a função que criamos antes, connect, para obter a conexão e depois navegamos até a collection de customers e fazemos um find sem filtro algum. O resultado desse find é um cursor, então usamos o toArray para convertê-lo para um array e retorná-lo.

Repare na última instrução, ela diz que a nossa função findAll poderá ser usada em outros pontos da nossa aplicação que importem nosso módulo db:

```
module.exports = { findAll }
```

Agora vamos programar a lógica que vai usar esta função. Abra o arquivo ./routes/index.js no seu editor de texto (sugiro o Visual Studio Code). Dentro dele temos apenas a rota default, que é um get no path raiz. Vamos editar essa rota da seguinte maneira (atenção ao carregamento do módulo db.js no topo):

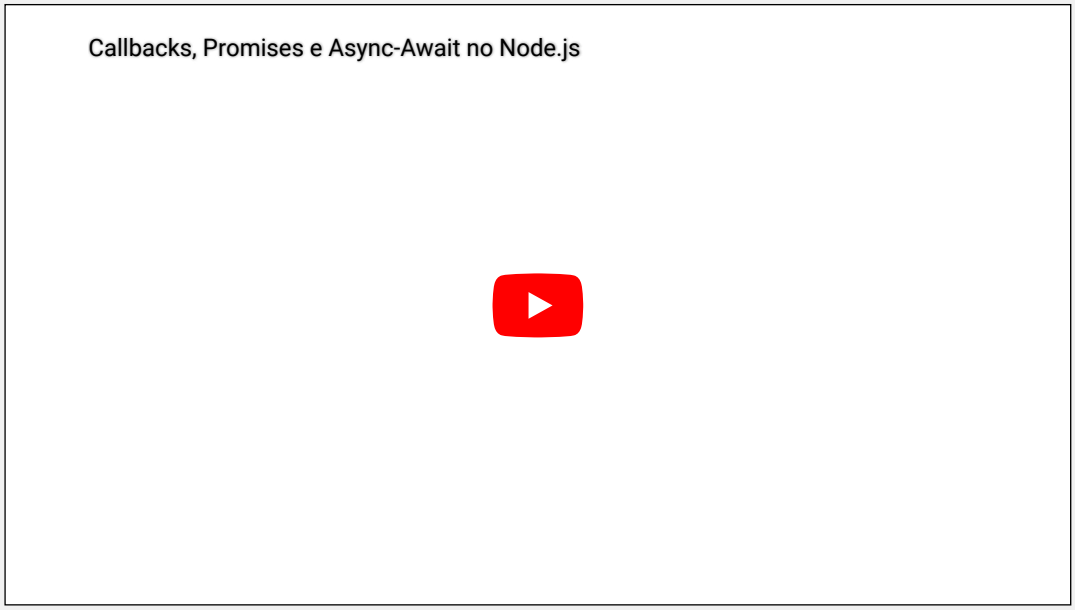
```
const db = require("../db");

/* GET home page. */
router.get('/', async (req, res, next) => {
  try {
```

```
    const docs = await db.findAll();
    res.render('index', { title: 'Lista de Clientes', docs });
  } catch (err) {
    next(err);
  }
})
```

router.get define a rota que trata essas requisições com o verbo GET. Quando recebemos um GET /, a função de callback dessa rota (aquela com req, res e next) é disparada e com isso usamos o findAll que acabamos de programar. Como esta é uma função assíncrona que vai no banco de dados, precisamos usar a palavra reservada await antes dela, ter um try/catch para tratar seus possíveis erros e o callback do router tem de ter a palavra async antes dos parâmetros da função.

Foge um pouco do escopo deste tutorial explicar programação assíncrona em Node.js, então recomendo o vídeo abaixo se quiser entender mais do assunto.



Agora vamos arrumar a nossa view para listar os clientes. Entre na pasta ./views/ e edite o arquivo index.ejs para que fique desse jeito:

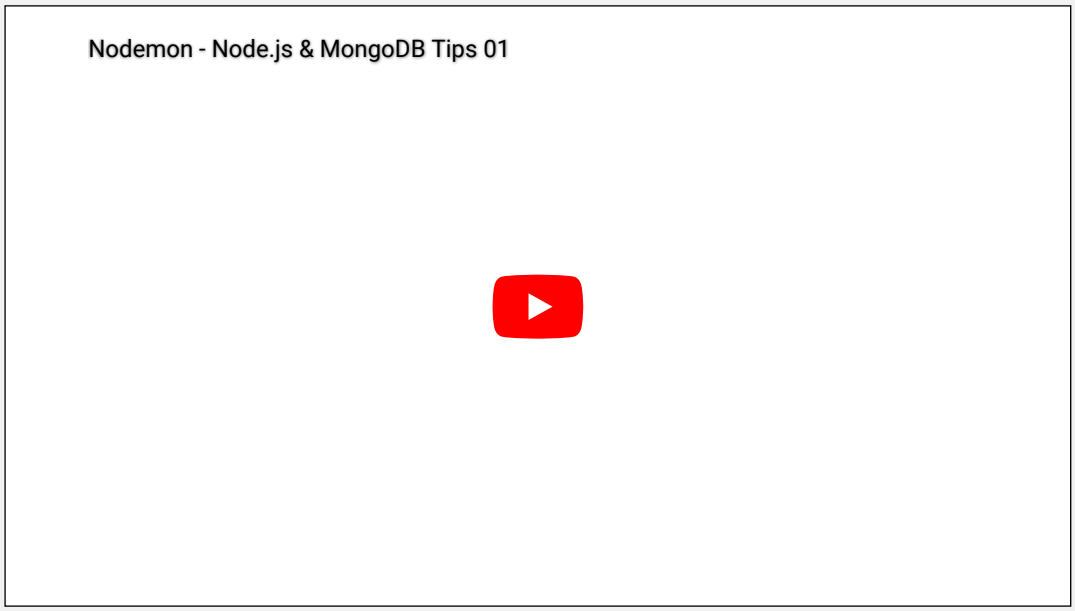
```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <ul>
      <% docs.forEach(function(customer){ %>
        <li>
            <%= customer.name %>
        </li>
      <% }) %>
    </ul>
  </body>
</html>
```

Aqui estamos dizendo que o objeto docs, que será retornado pela rota que criamos no passo anterior, será iterado com um forEach e seus objetos utilizados um-a-um para compor uma lista não-ordenada com seus nomes.

Isto é o bastante para a listagem funcionar. Para não ter que ficar derrubando e subindo novamente a sua aplicação toda hora, modifique o script de start no seu package.json para:

```
1  "scripts": {
2    "start": "npx nodemon ./bin/www"
3  },
```

Agora suba com npm start novamente e toda vez que fizer novas alterações, elas serão carregadas automaticamente por causa da extensão nodemon, que eu falo mais a respeito no vídeo abaixo.



Abra seu navegador, acesse http://localhost:3000/userlist e maravilhe-se com o resultado.

Se você viu a página acima é porque sua conexão com o banco de dados está funcionando!

Agora vamos seguir adiante com coisas mais incríveis em nosso projeto de CRUD!

Parte 4 – Cadastrando no banco

Listar dados é moleza e salvar dados no MongoDB não é algo particularmente difícil. Essencialmente precisamos definir uma rota para receber um POST, ao invés de um GET.

Primeiro vamos criar a nossa tela de cadastro de usuário com dois clássicos e horríveis campos de texto à moda da década de 90. Dentro da pasta views, crie um new.ejs com o seguinte HTML dentro:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <form action="/new" method="POST">
      <p>Nome:<input type="text" name="name"/></p>
      <p>Idade:<input type="number" name="age" /></p>
      <input type="submit" value="Salvar" />
    </form>
  </body>
</html>
```

Agora vamos voltar à pasta routes e abrir o nosso arquivo de rotas, o index.js onde vamos adicionar duas novas rotas. A primeira, é a rota GET para acessar a página new quando acessarmos /new no navegador:

```
router.get('/new', (req, res, next) => {
  res.render('new', { title: 'Novo Cadastro' });
});
```

Se você reiniciar seu servidor Node e acessar http://localhost:3000/newuser verá a página abaixo:

Se você preencher esse formulário agora e clicar em salvar, dará um erro 404. Isso porque ainda não criamos a rota que receberá o POST desse formulário!.

Então, vamos alterar nosso db.js para incluir uma nova função, desta vez para inserir clientes usando a conexão global e, novamente, incluindo esta função no module.exports:

```
async function insert(customer) {
    const db = await connect();
    return db.collection(COLLECTION).insertOne(customer);
}

module.exports = { findAll, insert }
```

Agora vamos criar uma rota para que, quando acessada via POST, nós chamaremos o objeto global db para salvar os dados no Mongo. A rota será a mesma /new, porém com o verbo POST. Então abra novamente o arquivo /routes/index.js e adicione o seguinte bloco de código logo após as outras rotas e antes do modules.export:

```
router.post('/new', async (req, res, next) => {
  const name = req.body.name;
  const age = parseInt(req.body.age);

  try {
    const result = await db.insert({ name, age });
    console.log(result);
    res.redirect('/');
  } catch (err) {
    next(err);
  }
})
```

Obviamente no mundo real você irá querer colocar validações (ensino aqui), mas aqui, apenas pego os dados que foram postados no body da requisição HTTP usando o objeto req (request/requisição). Crio um JSON com essas duas variáveis e envio para função insert que criamos agora a pouco.

Novamente, idas ao banco são assíncronas no Node.js, então isso exige o uso de async na função que a chamada estiver, o uso de await para recebermos o seu retorno e um try/catch para tratar erros. Se tudo der certo, a rota redireciona para a index novamente para que vejamos a lista atualizada. Apenas para finalizar, edite o views/index.ejs para incluir um link para a página /new:

```
<hr />
  <a href="/new">Cadastrar novo cliente</a>
</body>

</html>
```

Resultado:

#5 – Atualizando clientes

Para atualizar clientes (o U do CRUD) não é preciso muito esforço diferente do que estamos fazendo até agora. No entanto, são várias coisas menores que precisam ser feitas e é bom tomar cuidado para não deixar nada pra trás.

Primeiro, vamos editar nossa views/index.ejs para que quando clicarmos no nome do cliente, joguemos ele para uma tela de edição. Fazemos isso com uma âncora ao redor do nome, construída no EJS:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <ul>
      <% docs.forEach(function(customer){ %>
        <li>
          <a href="/edit/<%= customer._id %>">
            <%= customer.name %>
          </a>
        </li>
      <% }) %>
    </ul>
    <hr />
    <a href="/new">Cadastrar novo cliente</a>
  </body>
</html>
```

Note que este link aponta para uma rota /edit que ainda não possuímos, e que após a rota ele adiciona o _id do customer, que servirá para identificá-lo na página seguinte. Com esse _id em mãos, teremos de fazer uma consulta no banco para carregar seus dados no formulário permitindo um posterior update. Por ora, apenas mudou a aparência da tela de listagem:

Sendo assim, vamos começar criando uma nova função no db.js que retorna apenas um cliente, baseado em seu _id:

```
const ObjectId = require("mongodb").ObjectId;
async function findOne(id) {
    const db = await connect();
    return db.collection(COLLECTION).findOne({_id: new ObjectId(id)});
}

module.exports = { findAll, insert, findOne }
```

Como nosso filtro do find será o id, ele deve ser convertido para ObjectId, pois virá como string na URL e o Mongo não entende Strings como _ids. Não esqueça de incluir esta nova função no module.exports, que está crescendo.

Agora vamos criar a respectiva rota GET em nosso routes/index.js que carregará os dados do cliente para edição no mesmo formulário de cadastro:

```
router.get('/edit/:id', async (req, res, next) => {
  const id = req.params.id;

  try {
    const doc = await db.findOne(id);
    res.render('new', { title: 'Edição de Cliente', doc, action: '/edit/' + doc._id });
  } catch (err) {
    next(err);
  }
})
```

Esta rota está um pouco mais complexa que o normal. Aqui nós pedimos ao db que encontre o cliente cujo id veio como parâmetro da requisição (req.params.id). Após ele encontrar o dito cujo, mandamos renderizar a mesma view de cadastro, porém com um model inteiramente novo contendo apenas um documento (o cliente a ser editado) e a action do form da view 'new.ejs'.

Mas antes de editar a view new.ejs, vamos editar a rota GET em /new para incluir no model dela o cliente e a action, mantendo a uniformidade entre as respostas:

```
router.get('/new', (req, res, next) => {
  res.render('new', { title: 'Novo Cadastro', doc: {"name":"","age":""}, action: '/new' });
```

```
});
```

Agora sim, vamos na new.ejs e vamos editá-la para preencher os campos do formulário com o model recebido, bem como configurar o form com o mesmo model:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <form action="<%= action %>" method="POST">
      <p>Nome:<input type="text" name="name" value="<%= doc.name %>" /></p>
      <p>Idade:<input type="number" name="age" value="<%= doc.age %>" /></p>
      <input type="submit" value="Salvar" />
    </form>
  </body>
</html>
```

Agora, se você mandar rodar e clicar em um link de edição, deve ir para a tela de cadastro mas com os campos preenchidos com os dados daquele cliente em questão:

Agora estamos muito perto de terminar a edição. Primeiro precisamos criar um nova função no db.js para fazer update, como abaixo:

```
async function update(id, customer) {
    const db = await connect();
    return db.collection(COLLECTION).updateOne({ _id: new ObjectId(id) }, { $set: customer });
}

module.exports = { findAll, insert, findOne, update }
```

O processo não é muito diferente do insert, apenas temos de passar o filtro do update para saber qual documento será afetado (neste caso somente aquele que possui o id específico). Também já inclui o module.exports atualizado na última linha para que você não esqueça.

Para encerrar, apenas precisamos configurar uma rota para receber o POST em /edit com o id do cliente que está sendo editado, chamando a função que acabamos de criar:

```
router.post('/edit/:id', async (req, res) => {
  const id = req.params.id;
  const name = req.body.name;
  const age = parseInt(req.body.age);

  try {
    const result = await db.update(id, { name, age });
    console.log(result);
```

```
    res.redirect('/');
  } catch (err) {
    next(err);
  }
})
```

Aqui carreguei o id que veio como parâmetro na URL, e os dados de nome e idade no body da requisição (pois foram postados via formulário). Apenas passei esses dados nas posições corretas da função de update e passei um callback bem simples parecido com os anteriores, mas que redireciona o usuário para a index do projeto em caso de sucesso, voltando à listagem.

E com isso finalizamos o update!

#6 – Excluindo clientes

Agora em nossa última parte do tutorial, faremos o D do CRUD, D de Delete!

Vamos voltar à listagem e adicionar um link específico para exclusão, logo ao lado do nome de cada cliente, incluindo uma confirmação de exclusão nele via JavaScript, como abaixo:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <ul>
      <% docs.forEach(function(customer){ %>
        <li>
          <a href="/edit/<%= customer._id %>">
            <%= customer.name %>
          </a>
          <a href="/delete/<%= customer._id %>"
            onclick="return confirm('Tem certeza que deseja excluir?');">
            X
          </a>
        </li>
      <% }) %>
    </ul>
    <hr />
    <a href="/new">Cadastrar novo cliente</a>
  </body>
</html>
```

Note que não sou muito bom de front-end, então sinta-se à vontade para melhorar os layouts sugeridos. Aqui, o link de cada cliente aponta para uma rota /delete passando _id do mesmo. Essa

rota será acessada via GET, afinal é um link, e devemos configurar isso mais tarde.

Vamos no db.js adicionar nossa última função, de delete:

```
async function deleteOne(id) {
    const db = await connect();
    return db.collection(COLLECTION).deleteOne({ _id: new ObjectId(id) });
}

module.exports = { findAll, insert, findOne, update, deleteOne }
```

Essa função é bem simples de entender se você fez todas as operações anteriores. Na sequência, vamos criar a rota GET /delete no routes/index.js:

```
router.get('/delete/:id', async (req, res) => {
  const id = req.params.id;

  try {
    const result = await db.deleteOne(id);
    console.log(result);
    res.redirect('/');
  } catch (err) {
    next(err);
  }
})
```

Nessa rota, após excluirmos o cliente usando a função da variável global.db, redirecionamos o usuário de volta à tela de listagem para que a mesma se mostre atualizada.

E com isso finalizamos nosso CRUD!

A continuação deste tutorial você confere neste post.

Curtiu o post? Então clica no banner abaixo e dá uma conferida no meu livro sobre programação web com Node.js!

TAGS: mongodb nodejs

Artigos Relacionados

NODE.JS

Como testar services em NestJS com Jest

CRIPTO

Como criar um bot de arbitragem de criptomoedas na BityPreço com Node.js - Parte 2

NODE.JS

Como criar uma WebApi com NestJS, Prisma e banco SQL

CRIPTO

Integração com Smart Contracts com Node.js e Web3.js

Ver mais

Olá, tudo bem?

O que você achou deste conteúdo? Conte nos comentários.

Escreva seu comentário...

Seu nome? Seu email? Postar

Fernando , 30/07/2017 às 14:50

Sobre conexões abertas, nesse caso eu n preciso fechar ela em nenhum momento?

Responder

Luiz Fernando Jr , 30/07/2017 às 15:16

Neste tutorial usei apenas uma conexão compartilhada globalmente com todas requisições, que é uma tática bem comum inclusive por ORMs (aqui fiz com driver nativo mesmo). Neste caso não há necessidade de fechar pois é apenas uma aberta sempre.

Joao Felipe Barros Neto , 21/08/2017 às 01:27

Estou com esse erro, vc pode me ajudar?
{ [MongoError: wrong type for 'q' field, expected object, found q: ObjectId('599a21619b956d2f761a233f')]
name: 'MongoError',
message: 'wrong type for 'q' field, expected object, found q: ObjectId('599a21619b956d2f761a233f')',
ok: 0,
errmsg: 'wrong type for 'q' field, expected object, found q: ObjectId('599a21619b956d2f761a233f')',
code: 9 }

Responder

Luiz Fernando Jr , 21/08/2017 às 09:33

Você está mandando um ObjectId para um campo 'q' que deveria ser um objeto JSON. Revise o documento que está tentando enviar pro Mongo, o único campo que geralmente é ObjectId é o '_id', como mostrado neste tutorial.

Zé Das Coves , 07/09/2017 às 23:57

Luiz / João,
Eu tb passei por esse problema.
A função "update" do db.js está passando o id com a síntaxe errada: ……..updateOne(new ObjectId(id)……..
Porém, depois de verificar o método "delete" que está funcionando, eu tentei passar o id da mesma forma e funcionou.
A linha de comando da função "update" do db.js ficou assim: global.conn.collection("customers").updateOne({_id: new ObjectId(id)}, customer, callback)
Ao invés de passar
new ObjectId(id)
altere para
{_id: new ObjectId(id)}
Forte abraço e Luíz, você é o cara!!!

Luiz Fernando Jr , 08/09/2017 às 00:08

Valeu Zé das Coves, é exatamente isso. Vou dar uma revisada se fui eu que deixei passar essa pra corrigir. Obrigado novamente.

Zé Das Coves , 08/09/2017 às 00:14

Pra mim é uma honra Luíz!!!
Eu que te agradeço por se dedicar a nós, meros mortais rsrs!!!

Luiz Fernando Jr , 08/09/2017 às 11:02

Nossa, não é pra tanto, hehehe. É um prazer passar o pouco que sei adiante.

erick couto , 22/08/2017 às 23:29

Sou leigo em node, uma duvida, li o tutorial de criação de API tb, queria saber se os dados que foram cadastrado nesse tutorial aqui pode ser chamado de API tb e posso consumir esses dados?
Ou esse tuto n tem nada a ver com o tutorial de API ? Obrigado.

Responder

Luiz Fernando Jr , 22/08/2017 às 23:33

Esse tutorial é uma aplicação web CRUD, com interface gráfica para pessoas poderem usar. Além disso esse tutorial usa o driver nativo do MongoDB.
APIs não possuem interface gráfica e o outro tutorial usa Mongoose ao invés do driver nativo. Você cria uma API quando quer expor o seu banco de dados (ou parte dele) para outros sistemas usarem.
Resumindo: use esse tutorial para criar aplicações web. Use o outro tutorial para criar serviços web.

Luiz Fernando Jr , 22/08/2017 às 23:34

Embora você consiga adaptar uma aplicação para virar um serviço e vice-versa.

Manoel Braz Maciel , 28/09/2017 às 12:03

Boa tarde, Luiz … Sou muito grato pelo seu trabalho. Tem me ajudado muito. Estou fazendo o meu TCC sobre a stack MEAN. E os seus artigos tem me sanado muitas dúvidas. Muito obrigado. Felicidade!

Responder

Luiz Fernando Jr , 28/09/2017 às 14:30

Boa tarde Manoel,
M, E e N você encontra aqui no blog bastante material. Já o 'A' vou ficar te devendo, hehehehe
Fico feliz em estar ajudando.

Manoel Braz Maciel , 04/10/2017 às 22:03

Realmente, ainda não cheguei no A … Mas, já ajudou bastante. Muito obrigado. Felicidade!

Leandro Missel , 04/10/2017 às 17:13

Grande Luiz, valeu por ajudar a comunidade.
Cara, tá rolando um erro com a linha “var id = req.params.id;”
Sabe se algo mudou?
Valeu, abraço!
Cannot read property ‘id’ of undefined
TypeError: Cannot read property ‘id’ of undefined
at /var/html/www/testeMongoDB/wekshoptdc/routes/index.js:27:22
at Layer.handle [as handle_request]
...

Responder

Luiz Fernando Jr , 04/10/2017 às 20:37

No router.get a rota deve ser edit/:id. Pelo visto deve ter alguma diferença no seu código em relação ao meu. Já baixou o zip dos fontes para dar uma olhada?

Leandro Missel , 05/10/2017 às 08:49

Vou baixar meu velho, mas no router.get está certo...

Luiz Fernando Jr , 05/10/2017 às 22:23

Vamos lá: o erro está reclamando que o req.params está undefined, ou seja, não estão vindo parâmetros na sua requisição. Provavelmente o seu GET não está trazendo o id na URL (pode ser culpa do seu HTML que gera este GET, que pode estar errado). Experimenta colocar um console.log(req.url) na primeira linha dentro da sua rota para ver o que vai aparecer no console ou revisa a parte que está chamando essa rota.

Ricardo Reis , 30/01/2018 às 18:11

No #4, passo 2, onde editamos o arquivo “www” é importante dizer que o código ” global.db = require(‘../db’); ” deve ser inserido uma linha acima do código ” var app = require(‘../app’); “.
Eu havia inserido logo no início do arquivo, ou seja, na primeira linha, e então deu erro.

Responder

Ricardo Reis , 30/01/2018 às 18:30

No tutorial está escrito para acessar ” http://localhost:3000/userlist ” mas na verdade não precisa do “userlist”, basta acessar “http://localhost:3000/”

Responder

Ricardo Reis , 31/01/2018 às 17:09

Luiz,
No db.js, porque no tutorial vc usa: .then(conn => global.conn = conn.db("workshoptdc"))
Mas no arquivo fonte você usou: .then(conn => global.conn = conn)
E também no tutorial você usou:
global.conn.collection("customers").updateOne({_id:new ObjectId(id)}, customer, callback);
Mas no seu arquivo fonte está:
global.conn.collection("customers").updateOne(new ObjectId(id), customer, callback);
Ou eu estou louco e fazendo confusão?
abraço

Responder

Luiz Fernando Jr , 31/01/2018 às 19:20

Diferença de versão do driver do MongoDB. Nas versões atuais algumas coisas mudaram, logo, dê preferência ao que está no post, pois consigo atualizar ele com mais facilidade.
Se tu baixar os fontes e rodar um NPM INSTALL, vai instalar a versão antiga do driver e deve rodar tudo normalmente.
Agora se criar um projeto do zero, tem de usar os fontes do post.

Ricardo Reis , 31/01/2018 às 19:28

Certo Luiz, obrigado pela resposta.
Mas estou com um erro quando tento fazer o Update, poderia me ajudar?
ERRO:
https://uploads.disquscdn.com/images/85d754a1e180d23b62fe5d7d2bade6284a459dd52a013bbc50336a5c1a5d9fab.pn
INDEX.JS
https://uploads.disquscdn.com/images/d47b51918318bdd8a19c2a7758d570f917fcbb249f695a8b86dc67fe2ceb20cc.png
DB.JS
https://uploads.disquscdn.com/images/8edf495313f67898337a06ae12f7ab8299d6439bc6ef1e9cfce34f79a1f33105.png

Luiz Fernando Jr , 03/02/2018 às 20:00

O updateOne exige que você use operações atômicas ao invés de substituir o documento inteiro, algo nesse modelo {$set: { nome: novoNome, idade: novaIdade }} ao invés de passar o customer inteiro.

Ricardo Reis , 04/02/2018 às 15:39

Luiz, muito obrigado.
Eu usei o código abaixo e deu certo.
function update(id, customer, callback){
global.conn.collection("customers").updateOne({_id:new ObjectId(id)}, {$set:{nome:customer.nome, idade:customer.idade}}, callback);
}

Em desenvolvimento , 15/10/2018 às 15:00

[...] artigo de hoje é uma continuação de um outro onde ensino como fazer um sistema de cadastro bem simples em Node.js, usando o web framework ExpressJS e o banco de dados MongoDB. Ou seja, o famoso [...]

Responder

Carlos Diniz , 10/03/2023 às 12:17

Olá pessoal,
Ótimo material professor, muito obrigado por investindo na educação, estimo muito sucesso em seus projetos.
Notei que no findOne para editar os customers está faltando passar o id do obejeto.
tive o problema e solucionei passando o _id para o new, da seguinte forma:
async function findOne(id) {
const db = await connect();
return db.collection(COLLECTION).findOne({ _id: new ObjectId(id) });
}
Espero ter ajudado.

Responder

Luiz Duarte , 18/03/2023 às 19:04

Verdade, grato pelo apontamento, corrigi no tutorial e no repo.

Douglas , 02/04/2023 às 11:19

muito bom!

Responder

Luiz Duarte , 03/04/2023 às 18:16

Fico feliz que tenha gostado Douglas!

Guilherme , 11/04/2023 às 01:06

Oi, no meu em vez de atualizar um usuário está criando outro com as alterações que fiz, o que pode ser?

Responder

Luiz Duarte , 11/04/2023 às 23:35

Se está usando updateOne como eu, o parâmetro de filtro talvez esteja errado. Dá uma revisada nisso, no db.js.

Entre para minha lista e receba conteúdos exclusivos e com prioridade

Junte-se a mais de 34 mil devs

Seu email

Enviar

Categorias

- .NET (20)
- Agile (106)
- Android SDK (45)
- Carreira (73)
- Corona SDK (2)
- Cripto (89)
- Empreendedorismo (66)
- Livros (52)
- Mobile (71)
- Node.js (212)
- React Native (14)
- Web (185)

Últimas Novidades

- Tutorial ERC-1155 com Solidity e HardHat (JS)
- Como criar NFTs usando Solidity e HardHat (JS)
- Como criar NFTs usando Solidity e HardHat (JS) – Parte 2
- Tutorial ERC-1155 com Solidity e HardHat (JS) – Parte 2
- Como testar services em NestJS com Jest

Tags

agile android aws blo business c# camunda carreira corona criptomoedas digital ocean emp firebase heroku ios java jira mobile mongodb mssql mysql nestjs nextjs nodejs oauth performance phonegap postgresql prisma react redis regex resenha seguranca solidity sqlite tdd typescript web web3

2010-23, LuizTools. Todos os direitos reservados.