NODE.JS

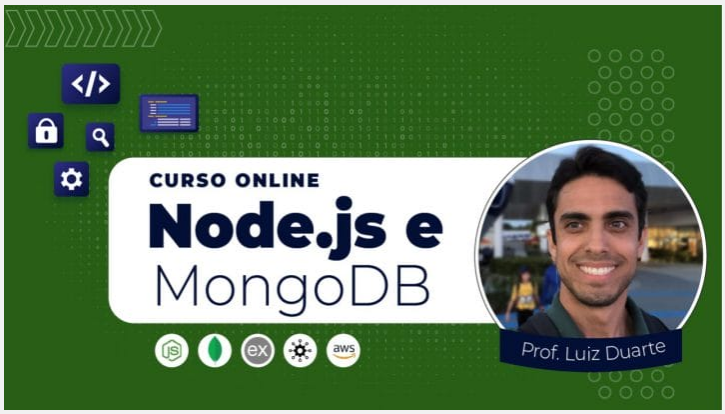
Como criar uma WebApi com Node.js, Express, TypeScript e MongoDB

Escrito por **Luiz Duarte**
em 14/03/2023

Recentemente escrevi um tutorial sobre como criar uma WebAPI com Node.js, Express e TypeScript, mas que não envolvia banco de dados. Fizemos tudo de forma extremamente profissional e já pensando em futuramente adicionar o suporte a banco de dados. Pois bem, essa hora chegou.

Neste tutorial vamos refatorar o projeto anterior, que você precisa ter feito para conseguir entender esse, visando adicionar o suporte ao banco de dados MongoDB. Também é interessante, embora não obrigatório, que você já conheça MongoDB também, sendo que o material recomendado é minha série de MongoDB para Iniciantes em NoSQL.

Dito isso, vamos ao tutorial!



#1 – Setup do Ambiente

O primeiro passo quando vamos trabalhar com um projeto envolvendo MongoDB é nos certificar que temos um servidor de MongoDB rodando à nossa disposição. Você pode ler o texto abaixo ou assistir a esse vídeo, com o procedimento de instalação.

Como instalar e executar MongoDB (Server)

Para fazer isso vamos acessar o site oficial do MongoDB e baixar o MongoDB. 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.

Dentro da pasta do seu projeto Node, que aqui chamei de webapi-mongodb, 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:\webapi-mongodb\data
```

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

Opcionalmente você pode fazer setup de um cliente também, o que facilita a gestão e os testes. 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.

Instalação e uso do MongoDB Compass (2023)

Nosso próximo passo é configurar algumas questões mais gerais do projeto. Vamos começar pelas dependências, precisamos adicionar o pacote do MongoDB e os seus types.

```
npm i mongodb
npm i -D @types/mongodb
```

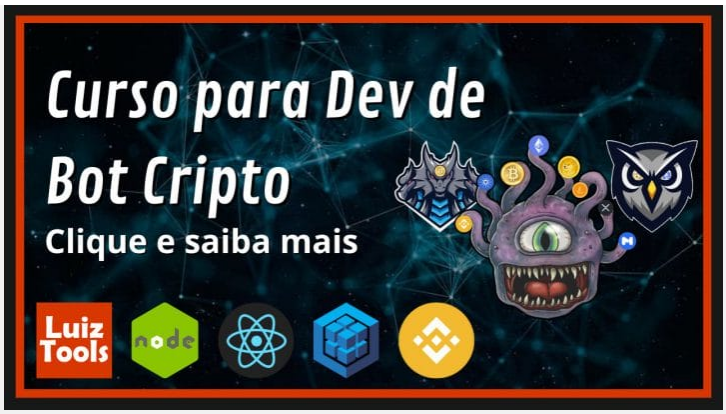
Agora, precisamos adicionar no .env mais duas variáveis de ambiente, ligadas ao seu servidor MongoDB.

```
PORT=3000
MONGO_HOST=mongodb://127.0.0.1:27017
MONGO_DATABASE=webapi
```

A saber:

- MONGO_HOST: o endereço do seu servidor de MongoDB (não use localhost, use o IP);
- MONGO_DATABASE: o nome da base de dados do seu projeto (chamei de webapi);

Com isso terminamos a etapa mais inicial de configuração.



#2 – Model e Controller

O MongoDb trabalha com documentos BSON, que é muito semelhante ao formato JSON, mas serializado de forma binária ao invés de texto. Dadas estas similaridades é muito simples de usar os documentos como se fossem objetos JS, bastando pequenos ajustes. Um dos ajustes que temos de fazer em nossa entidade Customer é que o id não será mais um número inteiro, mas sim um ObjectId, que é o tipo padrão para identificadores únicos no MongoDB, ficando como abaixo.

```
import { ObjectId } from "mongodb";

export default class Customer {
    _id?: ObjectId;
    name: string;
    cpf: string;

    constructor(name: string, cpf: string, id?: ObjectId) {
        this._id = id;
        this.name = name;
        this.cpf = cpf;
    }
}
```

Começo importando o ObjectId do pacote MongoDB já no topo, sendo que o uso na declaração da propriedade _id (antiga id) que

agora além de mudar de tipo virou opcional. Isso porque na criação na customers nós não temos esta informação.

Dada esta alteração, tem um lugar em nossa aplicação que esperava que os ids fossem numéricos e que agora precisaremos ajustar. Estou falando do customerController.ts que em diversas funções fazia parseInt no parâmetro id que originalmente vem em texto nas URLs. Agora podemos passar esse texto diretamente ao repository, que tratará depois de converter para ObjectId internamente mais à frente.

Abaixo coloquei somente as funções que sofreram alteração por causa dessa troca de tipo do ObjectId.

```
async function getCustomer(req: Request, res: Response, next: NextFunction) {
    const id = req.params.id;
    const customer = await customerRepository.getCustomer(id);
    if (customer)
        res.json(customer);
    else
        res.sendStatus(404);
}

async function patchCustomer(req: Request, res: Response, next: NextFunction) {
    const id = req.params.id;
    const customer = req.body as Customer;
    const result = await customerRepository.updateCustomer(id, customer);
    if (result)
        res.json(result);
    else
        res.sendStatus(404);
}

async function deleteCustomer(req: Request, res: Response, next: NextFunction) {
    const id = req.params.id;
    const success = await customerRepository.deleteCustomer(id);
    if (success)
        res.sendStatus(204);
    else
        res.sendStatus(404);
}
```

E por fim, vamos criar um novo módulo na nossa aplicação, chamado db.ts. Como podemos ter diversos repositories diferentes em uma webapi real, convém criarmos um módulo central que saiba realizar atividades comuns à todos repositories, como conexão ao banco, por exemplo. Assim, essa será nossa única funcionalidade por enquanto, nesse bloco completamente inédito de código, explicado adiante.

```
import { Db, MongoClient } from "mongodb";

let singleton: Db;

export default async (): Promise<Db> => {
    if (singleton) return singleton;

    const client = new MongoClient(`${process.env.MONGO_HOST}`);

    await client.connect();

    singleton = client.db(process.env.MONGO_DATABASE);

    return singleton;
```

Começamos importando o type e a classe necessários. Depois, declaramos uma variável singleton em alusão ao design pattern de mesmo nome, pois queremos ter apenas uma conexão para toda a aplicação. A ideia é que, quando chamarmos a função de conexão, iremos verificar se a variável singleton já foi inicializada. Se já foi, retornamos ela imediatamente. Se não foi, fazemos o processo de inicialização.

O processo de inicialização de uma conexão é bem simples e consiste de instanciar um MongoClient com a connection string do

servidor, depois chamar a função que faz a conexão e por último selecionarmos a base de dados específica que queremos do servidor. Isso tudo vai nos dar um objeto de conexão com o banco de dados pronto para ser usado.

E com isso, finalizamos a penúltima rodada de alterações, agora vem a mais pesada e final.

#3 – MongoDB Repository

E por último vem uma rodada intensa de alterações no customerRepository.ts. Primeiro vamos ajustar nossas importações e as funções de leitura, o que já nos traz alguns conceitos novos e importantes (repare que tirei as promises pois as funções do MongoDB já retornam promises por si só).

```
1  import connect from '../db';
2  import { ObjectId } from 'mongodb';
3
4  const COLLECTION = "customers";
5
6  async function getCustomer(id: ObjectId | string): Promise<Customer | null> {
7      if (!ObjectId.isValid(id)) throw new Error(`Invalid id.`);
```

```
8
9       const db = await connect();
10      const customer = await db.collection(COLLECTION)
11          .findOne({ _id: new ObjectId(id) });
12
13      if (!customer) return null;
14
15      return new Customer(customer.name, customer.cpf, customer._id);
16  }
17
18  async function getCustomers(): Promise<Customer[]> {
19      const db = await connect();
20      const customers = await db.collection(COLLECTION)
21          .find()
22          .toArray();
23
24      return customers.map(c => new Customer(c.name, c.cpf, c._id));
25  }
```

Sobre as novas importações, carregamos o módulo db como sendo uma única função connect, que serve para obter acesso a uma conexão com o banco de dados. Além disso carregamos a classe ObjectId que vamos usar bastante e definimos uma constante com o nome da coleção de documentos que vamos manipular neste repository.

A função getCustomer espera agora um id que pode estar no formato correto (ObjectId) ou como string, aí neste caso necessitando de conversão antes de ser usada como filtro na função findOne do MongoDB. Além disso, antes da conversão, eu verifico se o id é um ObjectId válido. Depois, usamos a função connect para pegar uma conexão com o banco e usamos ela para acessar a coleção de customers e enviar o comando findOne que usará o id como filtro e retornará um documento “raw” de customer. Este documento BSON não pode ser convertido diretamente para a classe Customer, então usamos do constructor da

mesma para fazer a cópia dos dados e deixar tudo no tipo correto e esperado pela aplicação.

A getCustomers faz praticamente a mesma coisa que sua versão "solo", apenas usando um find sem filtro, para retornar todos elementos, em a função toArray para garantir que todos os dados do banco sejam retornados de uma só vez em formato de array de documentos. Array este que logo na sequência é transformado em array de Customer com a ajuda de um map, de forma semelhante ao que fizemos antes.

Agora vamos às funções de escrita. Um ponto recorrente aqui segue sendo as conversões de ids string para ObjectId e o teste de validade dos mesmos, sempre que existir este parâmetro.

```
sync function addCustomer(customer: Customer): Promise<Customer> {
    if (!customer.name || !customer.cpf)
        throw new Error(`Invalid customer.`);

    const db = await connect();
    const result = await db.collection(COLLECTION)
        .insertOne(customer);

    customer._id = result.insertedId;
    return customer;
}

async function updateCustomer(id: string | ObjectId, newCustomer: Customer): Promise<Customer |
    if (!ObjectId.isValid(id)) throw new Error(`Invalid id.`);

    const db = await connect();
    await db.collection(COLLECTION)
        .updateOne({ _id: new ObjectId(id) }, { $set: newCustomer });

    return getCustomer(id);
}

async function deleteCustomer(id: string | ObjectId): Promise<boolean> {
    if (!ObjectId.isValid(id)) throw new Error(`Invalid id.`);

    const db = await connect();
    const result = await db.collection(COLLECTION)
```

```
28          .deleteOne({ _id: new ObjectId(id) });
29
30      return result.deletedCount > 0;
31  }
```

Aqui temos um fluxo muito semelhante dos anteriores em cada uma das funções, apenas mudando as funções que chamamos para cada uma das operações. Segue um resumo:

- findOne: retorna apenas um documento conforme filtro;
- find: retorna um array de documentos conforme filtro;
- insertOne: insere um documento com os dados passados;
- updateOne: altera um documento conforme filtro usando os dados passados;
- deleteOne: exclui um documento conforme filtro;

Vale uma atenção especial à função updateCustomer apenas, já que ela é a mais complexa. O primeiro parâmetro é um filtro, assim como faríamos em um findOne. O documento encontrado por aquele filtro será alterado conforme o operador de atualização definido no segundo parâmetro. O uso de $set indica que todos os campos passados deverão ser alterados e os demais, ignorados, o que mantém exatamente o mesmo comportamento que á tínhamos no passado.

Não esqueça de exportar as funções criadas ao final do módulo e com isso, finalizamos a implementação do CRUD com MongoDB em nossa WebAPI. Recomendo que teste via Postman cada um dos endpoints a fim de verificar se estão todos funcionando corretamente. E se quiser aprender a usar Prisma ORM ao invés do driver nativo do MongoDB, use este tutorial.

Até a próxima!

TAGS: mongodb nodejs typescript

Artigos Relacionados

CRIPTO

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

NODE.JS

Como compartilhar Prisma Schema/Client em Monorepo

NODE.JS

Autenticação JSON Web Token (JWT) em NestJS

NODE.JS

Como usar Variáveis de Ambiente com NestJS

Ver mais

Olá, tudo bem?

O que você achou deste conteúdo? Conte nos comentários.

Escreva seu comentário...

Seu nome? Seu email? Postar

Entre para minha lista e receba conteúdos exclusivos e com prioridade

Junte-se a mais de 34 mil devs

eu email Enviar

Categorias

.NET (20)

Agile (106)

Android SDK (45)

Últimas Novidades

Como criar bot/robô trader para Uniswap V3 em Node.js

Tags

agile android aws blo business c# camunda carreira