11 DE AGOSTO DE 2023 / #NODE.JS

Como usar o MongoDB e o Mongoose com o Node.js – melhores práticas para devs de back-end

Daniel Rosa

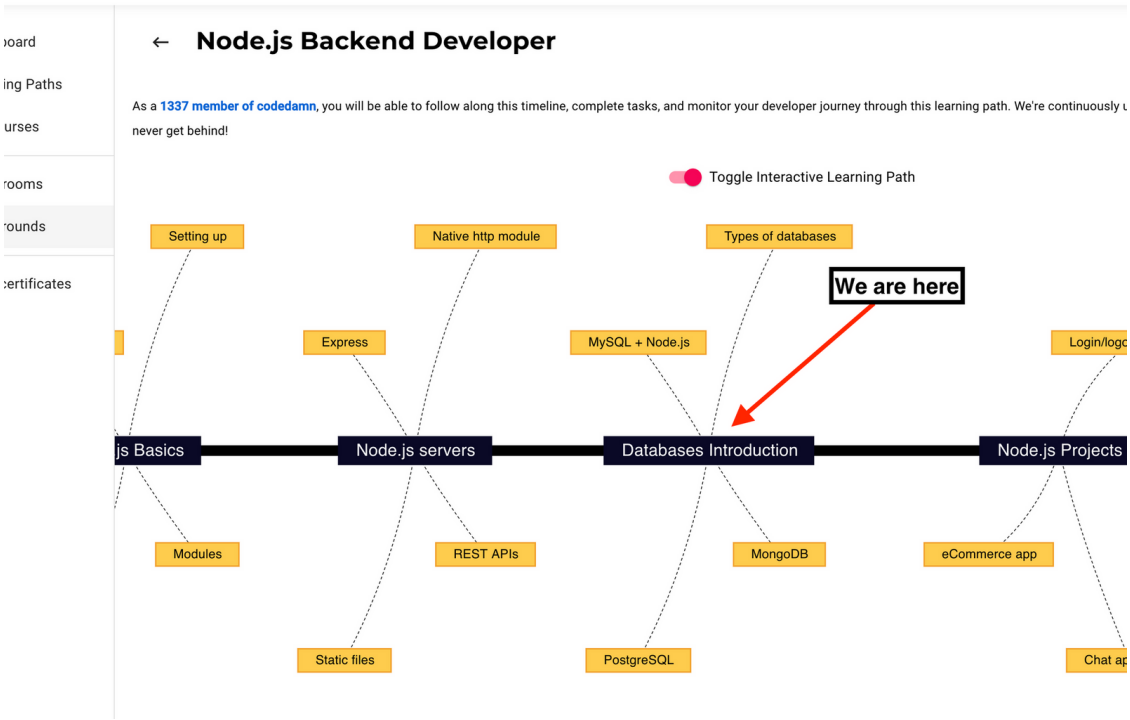
Artigo original:
https://www.freecodecamp.org/news/mongodb-mongoose-node-tutorial/

Neste artigo, analisaremos algumas das práticas recomendadas a serem seguidas ao configurar o MongoDB e o Mongoose com o Node.js.

Pré-requisitos para este artigo

Este artigo é um dos caminhos de aprendizado de *back-end* da codedamn (em inglês), onde começamos com os conceitos básicos de *back-end* e os abordamos em detalhes. Portanto, presumo que você já tenha alguma experiência com JavaScript (e Node.js).

Atualmente, estamos aqui:



Se você tem pouca experiência com Node.js/JavaScript ou *back-end* em geral, este é, provavelmente, um bom lugar para começar (em

Por que precisaremos do Mongoose?

Para entender por que precisamos do Mongoose, vamos entender como o MongoDB (e um banco de dados) funciona no nível da arquitetura.

- Você tem um servidor de banco de dados (servidor da comunidade do MongoDB, por exemplo)
- Você tem um script do Node.js em execução (como um processo)

O servidor do MongoDB escuta em um soquete TCP (normalmente) e o processo do Node.js pode se conectar a ele usando uma conexão TCP.

Além do TCP, o MongoDB também tem seu próprio protocolo para entender o que exatamente o *client* (nosso processo do Node.js) quer que o banco de dados faça.

Para essa comunicação, em vez de aprender as mensagens que temos que enviar na camada TCP, abstraímos isso com a ajuda de um software de "driver", chamado de *MongoDB driver*, neste caso. O *MongoDB driver* está disponível como um pacote do npm aqui.

Agora, lembre-se, o *MongoDB driver* é responsável por conectar e abstrair para você as solicitações/respostas de comunicação de baixo nível – é até aí que você consegue ir sendo um desenvolvedor.

área de superfície para gerar menos bugs e erros em seu código. Você precisa de algo mais.

Conheça o Mongoose. O Mongoose é uma abstração sobre o driver nativo do MongoDB (o pacote npm que mencionei acima).

A regra geral com abstrações (do jeito que eu entendo) é que, a cada abstração, você perde algum poder de operação de baixo nível. Isso, contudo, não significa necessariamente que seja ruim. Às vezes, isso aumenta a produtividade em mais de mil vezes, pois você nunca precisa realmente ter acesso total à API subjacente, de todo modo.

Uma boa maneira de pensar sobre isso é tecnicamente criar uma aplicação de bate-papo em tempo real, tanto em C, quanto em Python.

O exemplo do Python seria muito mais fácil e rápido para você, como desenvolvedor, implementar com maior produtividade.

O C *pode ser* mais eficiente, mas terá um custo enorme em produtividade/velocidade de desenvolvimento/bugs/travamentos. Além disso, na maioria das vezes, você não precisa ter o poder que o C oferece para implementar *websockets*.

Do mesmo modo, com o Mongoose, você pode limitar sua área de superfície de acesso à API de nível inferior, mas desbloquear muitos ganhos potenciais e uma boa DX (*Developer Experience*, ou, em português, experiência de desenvolvedor – em contraste com a experiência do usuário, ou UX).

Para começar, mostraremos rapidamente como conectar o banco de dados do MongoDB com o Mongoose:

```
mongoose.connect(DB_CONNECTION_STRING, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    useCreateIndex: true,
    useFindAndModify: false
})
```

Esse formato de conexão garante que você esteja usando o novo *parser* de URL do Mongoose e que não esteja usando práticas já preteridas (do inglês, *deprecated*). Você pode ler em profundidade sobre todas essas mensagens de preterimento aqui (em inglês), se quiser.

Como realizar operações com o Mongoose

Agora, discutiremos rapidamente as operações com o Mongoose e como devemos realizá-las.

O Mongoose traz opções para dois modos:

1. *Query* (em português, consulta) baseada no cursor
2. *Query* de busca completa (em inglês, *full fetching query*)

Query baseada no cursor

eficiente de se trabalhar com grandes quantidades de dados em um ambiente de memória limitada.

Imagine que você tenha que fazer o *parsing* documentos de 10 GB em um servidor em nuvem de 1 GB/1 núcleo. Você não pode buscar toda a coleção, pois não caberá no seu sistema. O cursor é uma boa (e, talvez, a única) opção aqui.

Query de busca completa

Esse é o tipo de consulta em que você obtém a resposta completa da consulta de uma só vez. Na maioria das vezes, é isso que você vai usar. Portanto, vamos nos concentrar principalmente nesse método aqui.

Como usar os modelos do Mongoose

Os modelos são o superpoder de Mongoose. Eles ajudam você a impor regras de "esquema" e fornecem uma integração perfeita do código do Node em chamadas de banco de dados.

O primeiro passo é definir um bom modelo:

```
import mongoose from 'mongoose'

const CompletedSchema = new mongoose.Schema(
        {
                type: { type: String, enum: ['course', 'classroo
                parentslug: { type: String, required: true },
                slug: { type: String, required: true },
                userid: { type: String, required: true }
```

```
const model = mongoose.model('Completed', CompletedSchema)
export default model
```

Este é um exemplo cortado diretamente da base de código da codedamn. Algumas coisas interessantes que você deve observar aqui:

1. Tente manter `required: true` em todos os campos que são obrigatórios. Isso pode poupar muitos problemas para você se você não usar um sistema de verificação de tipo estático como o TypeScript para ajudá-lo com nomes de propriedade corretos ao criar um objeto. Além disso, a validação gratuita também é muito legal.
2. Defina índices e campos exclusivos. A propriedade `unique` (exclusivo) também pode ser adicionada a um esquema. Os índices são um tema amplo, por isso não vou me aprofundar aqui. Em grande escala, porém, eles podem realmente ajudá-lo a acelerar muito suas consultas.
3. Defina um nome de coleção explicitamente. Embora o Mongoose possa dar, automaticamente, um nome à coleção com base no nome do modelo (aqui, por exemplo, `Completed`), isso é muita abstração na minha opinião. Você deve pelo menos saber sobre seus nomes de banco de dados e coleções em sua base de código.
4. Restrinja os valores, se puder, usando *enums*.

CRUD significa Create, Read, Update e Delete (criar, ler, atualizar e excluir, respectivamente, em português). Essas são as quatro opções fundamentais com as quais você pode executar qualquer tipo de manipulação de dados em um banco de dados. Vamos ver rapidamente alguns exemplos dessas operações.

A operação *Create*

Isso significa simplesmente criar um registro em um banco de dados. Vamos usar o modelo que definimos acima para criar um registro:

```
try {
    const res = await CompletedSchema.create(registro)
} catch(erro) {
    console.error(erro)
    // tratamento do erro
}
```

Mais uma vez, algumas coisas a apontar aqui:

1. Use *async-await* em vez de *callbacks* (é mais bonito – e não há muito benefício em termos de desempenho em se usar a outra opção)
2. Use blocos *try-catch* ao redor das *queries*, pois a *query pode* falhar por diversas razões (registro duplicado, valor incorreto e assim por diante)

A operação *Read*

Isso significa ler os valores existentes do banco de dados. É simples como parece, mas há algumas coisas que você deve saber com

```
const res = await CompletedSchema.find(info).lean()
```

1. Você percebeu a função `lean()`? Ela é muito útil para o desempenho. Por padrão, o Mongoose processa os documentos retornados do banco de dados e adiciona seus métodos *mágicos* a eles (por exemplo, `.save`)
2. Ao usar `.lean()`, o Mongoose retorna objetos em JSON simples em vez de documentos pesados em termos de memória e recursos. Isso torna as *queries* mais rápidas e faz com que pesem menos na CPU.
3. Porém, é possível omitir o `.lean()` se estiver, de fato, pensando em atualizar os dados (o que veremos a seguir)

A operação *Update*

Se você já tem um documento do Mongoose com você (sem utilizar o `.lean()`), pode simplesmente modificar a propriedade do objeto e salvá-lo usando `object.save()`:

```
const doc = await CompletedSchema.findOne(info)
doc.slug = 'outra-coisa'
await doc.save()
```

Lembre-se de que, aqui, duas chamadas ao banco de dados estão sendo feitas. A primeira é no `findOne` e a segunda é no `doc.save`.

Se puder, você deve sempre reduzir o número de solicitações que atingem o banco de dados (porque, se você estiver comparando

```
const res = await CompletedSchema.updateOne(<condição>, <consulta
```

Assim, teríamos apenas uma chamada ao banco de dados.

A operação *Delete*

Delete também é direto com relação ao Mongoose. Vejamos como podemos excluir um único documento:

```
const res = await CompletedSchema.deleteOne(<condição>)
```

Assim como `updateOne`, `deleteOne` também aceita o primeiro argumento como condição de correspondência para o documento.

Há também outro método chamado `deleteMany`, que deve ser usado apenas quando você sabe que deseja excluir vários documentos.

Em qualquer outro caso, use sempre `deleteOne` para evitar várias exclusões acidentais, especialmente quando você estiver tentando executar *queries* por conta própria.

Conclusão

Este artigo foi uma introdução simples ao mundo do Mongoose e do MongoDB para desenvolvedores do Node.js.

aprendizagem de back-end da freeCodeCamp (em inglês). Fique à vontade para entrar em contato com o autor pelo Twitter para fornecer seu *feedback*!

Daniel Rosa

Um profissional dos idiomas humanos apaixonado por linguagens de computador. | A world languages professional in love with computer languages.
