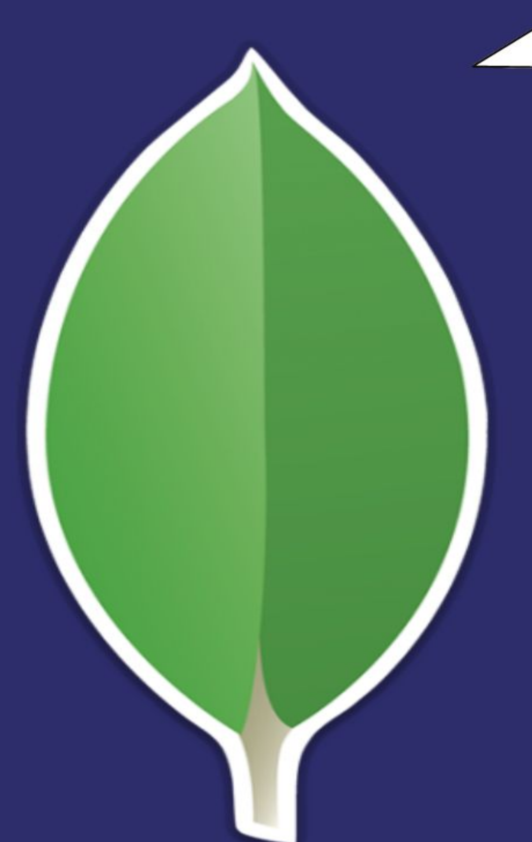
LUIZ DUARTE

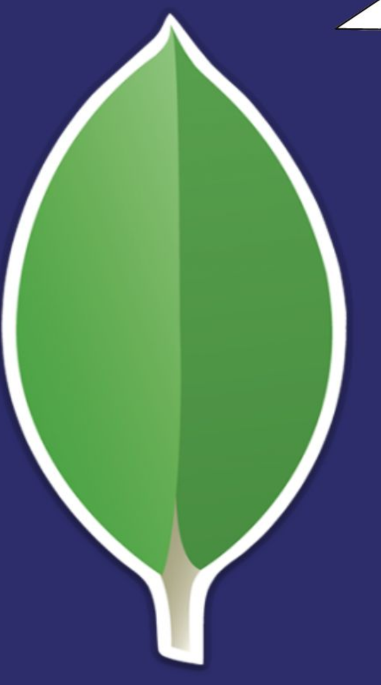
2023



MongoDB para Iniciantes

POR LUIZTOOLS

LUIZ DUARTE



MongoDB para Iniciantes

POR LUIZTOOLS

OceanofPDF.com

Luiz Fernando Duarte Júnior

MongoDB para iniciantes

Um guia prático

4 ª edição

Gravataí/RS
Edição do Autor
2023

OceanofPDF.com

Copyright © Luiz Fernando Duarte Júnior, 2023

MongoDB para Iniciantes

Um Guia Prático

Luiz Duarte

Duarte, Luiz,
MongoDB para Iniciantes - Um Guia Prático - 3 ª edição

Luiz Duarte - Gravataí/RS:2023

ISBN 978-65-900538-2-4

1. Computação, Internet e Mídia Digital. 2. Computação, Informática e Mídias Digitais.

Reservados todos os direitos.

OceanofPDF.com

MongoDB para Iniciantes
- Um Guia Prático

Sobre o autor

Antes de começar

1 Introdução ao NoSQL e MongoDB
- Introdução ao NoSQL
- Introdução ao MongoDB
- Principais Diferenças
- Quando devo usar MongoDB?
- Quando não devo usar MongoDB?
- Finalizando

2 Primeiros Passos
- Studio 3T

3 CRUD: Create, Read, Update & Delete
- Insert
  - insertOne e insertMany
- Find
  - findOne
  - Filter Operators
  - Personalizando retorno de consultas
- Update
  - Update Operators
  - Update Options
- Delete
- Outras Funções CRUD

4 Modelagem de Dados
- Princípios importantes
- Modelagem de Blog: Relacional
- Modelagem de Blog: Não-Relacional
- CRUD com Subdocumentos
- CRUD com campos multivalorados

- Padrões para Modelagem em MongoDB
  - Construindo com padrões
  - Approximation
  - Attribute
  - Bucket
  - Computed
  - Document Versioning
  - Extended Reference
  - Outlier
  - Pre-allocation
  - Polymorphic
  - Schema Versioning
  - Subset
  - Tree

5 Criando aplicações com MongoDB
- Node.js: Web API com MongoDB
  - Programando a API
- ASP.NET Core: Mecanismo de busca com MongoDB
  - Preparando o banco
  - Voltando ao Projeto
- PHP: Aplicação de Cadastro

6 Gerenciamento Básico
- Alterando metadados
- Analisando Comandos
- Autenticação
- Backup com MongoDump
- Restauração com MongoRestore
- Importando dados com MongoImport
- Replicando Dados
  - Criando um Replica Set
  - Usando um Replica Set
  - Segurança em Replica Sets
- MongoDB na AWS

        - Criando o cluster
        - Configurando o cluster
- 7 Boas práticas
    - Boas práticas de Hardware
        - Tenha memória RAM suficiente
        - Use discos adequados
        - Use múltiplos cores
    - Boas práticas de aplicação
        - Atualize apenas os campos alterados
        - Evite negações em queries
        - Analise o plano de execução
        - Configure a garantia de escrita apropriada
        - Use os drivers mais recentes
    - Boas práticas de Modelagem e Índices
        - Armazene todos os dados de uma entidade em um único documento
        - Evite documentos muito grandes
        - Evite nomes de campos desnecessariamente longos
        - Mantenha a manutenção dos índices em dia
        - Outras dicas gerais
- Seguindo em frente
    - Curtiu o Livro?

OceanofPDF.com

Sobre o autor

Luiz Fernando Duarte Júnior é Bacharel em Ciência da Computação pela Universidade Luterana do Brasil (ULBRA, 2010) e Especialista em Desenvolvimento de Aplicações para Dispositivos Móveis pela Universidade do Vale do Rio dos Sinos (UNISINOS, 2013).

Carrega ainda um diploma de Reparador de Equipamentos Eletrônicos (SENAI, 2005) e quase duas dezenas de certificações em métodos ágeis, liderança e coaching (Scrum.org, IBC e outros) obtidas em mais de uma década estudando e praticando estes assuntos, sendo algumas delas com reconhecimento internacional.

Atuando na área de TI desde 2006, na maior parte do tempo como desenvolvedor web, é apaixonado por desenvolvimento de software desde que teve o primeiro contato com a linguagem Assembly no curso de eletrônica. Falando de banco de dados, durante muitos anos trabalhou apenas com bancos relacionais, especialmente o Microsoft SQL Server, mas também com MySQL, SQLite e Microsoft Access. Porém em 2015 isso mudou, quando conheceu o MongoDB.

Foi amor à primeira vista e a paixão continua a crescer!

Trabalhando com MongoDB desenvolveu diversos projetos para empresas de todos os tamanhos, desde grandes empresas como Softplan até startups como Route, Busca Acelerada e Só Famosos, além de ministrar palestras e cursos de MongoDB para alunos do curso superior de várias universidades e eventos de tecnologia.

Um grande entusiasta deste banco de dados, espera que com esse livro possa ajudar ainda mais pessoas a aprenderem a desenvolver softwares com MongoDB e aumentar a competitividade das empresas brasileiras e a empregabilidade dos profissionais de TI.

Além de viciado em desenvolvimento, atua como consultor em sua própria empresa de software e treinamentos, como professor de pós-graduação em universidades (EAD) e é autor do blog https://www.luiztools.com.br , onde escreve semanalmente sobre métodos ágeis e desenvolvimento de software.

Entre em contato, o autor está sempre disposto a ouvir e ajudar seus leitores.

Me siga nas redes sociais: https://about.me/luiztools

Conheça meus outros livros: https://www.luiztools.com.br/meus-livros

Conheça meus cursos online: https://www.luiztools.com.br/meus-cursos

OceanofPDF.com

Antes de começar

Without requirements and design,
programming is the art of adding bugs to an empty text file.
- Louis Srygley

Antes de começarmos, é bom você ler esta seção para evitar surpresas e até para saber se este livro é para você.

Primeiramente, este livro vai lhe ensinar a usar o banco de dados não-relacional MongoDB, mas não vai lhe ensinar os conceitos essenciais de banco de dados, ele exige que você já saiba isso, ao menos em um nível básico (a primeira cadeira/módulo de banco de dados do seu curso de computação já deve ser o suficiente).

Segundo, este livro exige que você já tenha conhecimento técnico prévio sobre computadores, que ao menos saiba mexer em um e que preferencialmente possua um.

Parto do pressuposto que você é ou já foi um estudante de Técnico em informática, Ciência da Computação, Sistemas de Informação, Análise e Desenvolvimento de Sistemas ou algum curso semelhante. Usarei diversos termos técnicos ao longo do livro que são comumente aprendidos nestes cursos e que não tenho o intuito de explicar aqui.

O foco deste livro é ensinar diversos aspectos da utilização de banco de dados MongoDB para quem nunca usou o referido banco ou está apenas começando nessa tecnologia, com foco na manipulação dos dados armazenados no mesmo.

Ao término deste livro você estará apto a construir bancos de dados MongoDB usando qualquer sistema operacional que seja suportado por ele, bem como aprenderá como realizar consultas, inserções, alterações de dados, exclusões, modelagem não-relacional e gerenciamento básico, como backup e restauração de bases. Como um adendo, aprenderá rapidamente

(sem aprofundamentos) como utilizar as linguagens C#, Node.js e PHP para criar sistemas com este banco.

Além disso, terá uma noção do mercado existente para esta tecnologia.

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

1 Introdução ao NoSQL e MongoDB

Acho que foi em 2007, eu estava fazendo as cadeiras de Banco de Dados I e Banco de Dados II na faculdade de Ciência da Computação. Eu via como modelar um banco de dados relacional, como criar consultas e executar comandos SQL, além de álgebra relacional e um pouco de administração de banco de dados Oracle.

Isso tudo me permitiu passar a construir sistemas de verdade, com persistência de dados. A base em Oracle me permitiu aprender o simplíssimo MS Access rapidamente e, mais tarde, migrar facilmente para o concorrente, SQL Server. Posteriormente cheguei ainda a trabalhar com MySQL, SQL Server Compact, Firebird (apenas estudos) e SQLite (para apps Android).

Todos relacionais. Todos usando uma sintaxe SQL quase idêntica. Isso foi o bastante para mim durante alguns anos. Mas essa época já passou faz tempo. Hoje em dia, cada vez mais os projetos dos quais participo têm exigido de mim conhecimentos cada vez mais poliglotas de persistência de dados, ou seja, diferentes linguagens e mecanismos para lidar com os dados das aplicações, dentre eles, o MongoDB.

Introdução ao NoSQL

NoSQL (às vezes interpretado como Not Only SQL - Não Somente SQL) é um termo genérico para uma classe definida de banco de dados não-relacionais que rompe uma longa história de banco de dados relacionais com propriedades ACID. Os bancos de dados que estão sob esses rótulos não podem exigir esquemas de tabela fixa e, geralmente, não suportam instruções e operações de junção SQL (JOIN).

O termo NoSQL foi primeiramente utilizado em 1998 como o nome de um banco de dados relacional de código aberto que não possuía uma interface SQL. Seu autor, Carlo Strozzi, alega que o movimento NoSQL "é completamente distinto do modelo relacional e portanto deveria ser mais

apropriadamente chamado "NoREL" ou algo que produzisse o mesmo efeito".

Este termo foi re-introduzido no início de 2009 por um funcionário do Rackspace, Eric Evans, quando Johan Oskarsson da Last.fm queria organizar um evento para discutir bancos de dados open source distribuídos. O nome — uma tentativa de descrever o surgimento de um número crescente de banco de dados não relacionais, que não tinham a preocupação de fornecer garantias ACID — faz referência ao esquema de atribuição de nomes dos bancos de dados relacionais mais populares do mercado: MySQL, MS SQL, PostgreSQL etc.

O MongoDB é um banco NoSQL/Não-Relacional, por não possuir entidades e relacionamentos, como nos bancos tradicionais e não fazer uso da linguagem SQL para consulta/manipulação de dados.

Existem diversos tipos de bancos de dados não-relacionais, categorizados de acordo com a organização dos dados, ao contrário dos relacionais cujos dados são sempre armazenados em tabelas com linhas e colunas, sendo que existem relacionamentos entre as tabelas:

- **Orientado a Documentos:** os mais populares, armazenam os dados em coleções de documentos;
- **Orientado a Grafos:** armazenam os dados em grafos, com vértices e arestas;
- **Colunares:** armazenam os dados agrupados pelas colunas em comum;
- **Chave-Valor:** armazenam os dados no formato chave e valor, como em um array associativo, mas que também podem ter regras de conjuntos, filas, pilhas, etc;
- **Time-Series:** armazenam os dados baseados em eventos temporais;
- **Multi-modelo:** suportam o armazenamento dos dados em mais de um modelo (incluindo às vezes o modelo relacional) ao mesmo tempo;

Cada modelo de armazenamento atende uma necessidade específica da indústria de software e nenhum se propõe a substituir de fato os bancos SQL, exceto nos cenários específicos para o qual foram projetados, sendo os bancos SQL uma solução genérica que atende de maneira razoável todos os casos.

Para cada modelo existem expoentes no mercado de tecnologia atual, tais como:

- **Orientado a Documentos:** MongoDB, Couchbase, CouchDB, RavenDB, DynamoDB;
- **Orientado a Grafos:** Neo4j;
- **Colunares:** Cassandra, HBase;
- **Chave-Valor:** Redis;
- **Time-Series:** InfluxDB;
- **Multi-Modelo:** ArangoDB, Cosmos DB;

Introdução ao MongoDB

MongoDB é um banco de dados de código aberto, gratuito, de alta performance, sem esquemas e orientado a documentos lançado em fevereiro de 2009 pela empresa 10gen. Foi escrito na linguagem de programação C++ (o que o torna portável para diferentes sistemas operacionais) e seu desenvolvimento durou quase 2 anos, tendo iniciado em 2007.

Por ser orientado a documentos JSON (armazenados em modo binário, apelidado de JSON), muitas aplicações podem modelar informações de modo muito mais natural, pois os dados podem ser aninhados em hierarquias complexas e continuar a ser indexáveis e fáceis de buscar, igual ao que já é feito em JavaScript.

Existem dezenas de bancos NoSQL no mercado, não porque cada um inventa o seu, como nos fabricantes tradicionais de banco SQL (não existem diferenças tão gritantes assim entre um MariaDB e um MySQL atuais que justifique a existência dos dois, por exemplo). É apenas uma questão ideológica, para dizer o mínimo, e MongoDB é um deles.

Existem dezenas de bancos NOSQL porque existem dezenas de problemas de persistência de dados que o SQL tradicional não resolve. Bancos não-relacionais document-based (que armazenam seus dados em documentos) são os mais comuns e mais proeminentes de todos, sendo o seu maior expoente o banco MongoDB como o gráfico abaixo da pesquisa mais recente de bancos de dados utilizados pela audiência do StackOverflow em 2021 mostra.

Fonte: https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

Dentre todos os bancos não relacionais o MongoDB é o mais utilizado com 1/4 de todos os respondentes alegarem utilizar ele em seus projetos, o que é mais do que até mesmo o Oracle, um banco muito mais tradicional (que nem aparece ali na imagem).

Basicamente neste tipo de banco (document-based ou document-oriented) temos coleções de documentos, nas quais cada documento é autossuficiente,

contém todos os dados que possa precisar, ao invés do conceito de não repetição + chaves estrangeiras do modelo relacional.

A ideia é que você não tenha de fazer JOINs pois eles prejudicam muito a performance em suas queries (são um mal necessário no modelo relacional, infelizmente). Você modela a sua base de forma que a cada query você vai uma vez no banco e com apenas uma chave primária pega tudo que precisa.

Obviamente isto tem um custo: armazenamento em disco. Não é raro bancos MongoDB consumirem muitas vezes mais disco do que suas contrapartes relacionais.

Principais Diferenças

Dentre as diferenças de usar um banco relacional versus um não-relacional baseado em documentos como o MongoDB, podemos citar:

Bancos relacionais	**MongoDB**
Armazenam os dados em tabelas, com linhas divididas em colunas;	Armazenam os dados em coleções de documentos, com campos e subdocumentos;
Curva de aprendizagem maior;	Curva de aprendizagem menor;
Possuem relacionamentos entre as diferentes tabelas e registros;	Não possuem relacionamentos entre as coleções e/ou documentos;
Foco em garantir o ACID;	Foco em garantir escalabilidade, alta-disponibilidade e performance;
Uso de transações para garantir commits e rollbacks;	Não há uso de transações e o ACID é garantido apenas a nível de documento; Transações devem ser garantidas a nível de aplicação;

Uso da linguagem de consulta SQL para manipular o banco;	Uso de comandos JavaScript para manipular o banco;
Cada coluna de uma linha pode armazenar apenas um dado;	Cada campo de um documento pode armazenar múltiplos valores ou até mesmo outro documento;
Possuem chaves-estrangeiras e JOINs;	Não possuem "FKs" nem "JOINs";
Schema pré-definido e rígido;	Schema variável conforme uso da aplicação;
Foco em não-repetição de dados;	Foco em acesso rápido de dados;
Menor consumo de disco;	Maior consumo de disco (geralmente);
Menor consumo de memória;	Maior consumo de memória (geralmente);

Entre muitas outras diferenças que ficarão mais claras ao longo deste livro.

Quando devo usar MongoDB?

MongoDB foi criada com Big Data em mente. Ele suporta tanto escalonamento horizontal quanto vertical usando replica sets (instâncias espelhadas) e sharding (dados distribuídos), tornando-o uma opção muito interessante para grandes volumes de dados, especialmente os desestruturados.

Dados desestruturados são um problema para a imensa maioria dos bancos de dados relacionais, mas não tanto para o MongoDB. Quando o seu schema é variável, é livre, usar MongoDB vem muito bem a calhar. Os documentos BSON (JSON binário) do Mongo são schemaless e aceitam quase qualquer coisa que você quiser armazenar, sendo um mecanismo de

persistência perfeito para uso com tecnologias que trabalham com JSON nativamente, como JavaScript (e consequentemente Node.js).

```
{
  name: "sue",                            ← field: value
  age: 26,                                ← field: value
  status: "A",                            ← field: value
  groups: [ "news", "sports" ]            ← field: value
}
```

Os use cases listados abaixo e definidos pelo time de engenharia do MongoDB são os cenários em que ele melhor desempenha a sua função, o que corrobora com a minha máxima de alguns anos que MongoDB não substitui os bancos relacionais em todos cenários, mas sim em alguns cenários específicos, que, segundo o time do MongoDB, são:

- **Catalog** : como catálogo de produtos (em ecommerces) e serviços;
- **Content Management:** como em sistemas dos tipos ECM (gerenciadores de conteúdo corporativo) e CMS (blogs, portais, etc);
- **Internet of Things:** como em sistemas com sensores real-time, como na indústria 4.0, em wearable computing (smartwatches e smart glasses, por exemplo);
- **Mobile:** como em aplicações móveis de baixa latência e alta escala, como os grandes players que temos em nossos smartphones;
- **Personalization:** como em sistemas com schema de dados personalizável para entregar conteúdo relevante aos usuários, muito usados no marketing digital;
- **Real-Time Analytics:** como em sistemas de estatísticas real-time (analytics);
- **Single View:** como em sistemas de dashboards, gerenciais e outros que agregam informações de diferentes fontes em uma base só;

Embora o MongoDB consiga ser usado de maneira tão generalista quanto um banco relacional, o recomendado é focar seu uso nos cenários acima

descritos, seja como o banco de dados principal ou satélite, fornecendo seus melhores recursos para parte dos dados armazenados na solução.

Mais tarde, ainda neste livro, falarei de como modelar os dados da sua aplicação levando em conta diferentes cenários e linkando com os use cases acima descritos.

Quando não devo usar MongoDB?

Nem tudo são flores e o MongoDB não é uma "bala de prata", ele não resolve todos os tipos de problemas de persistência existentes.

Você não deve utilizar MongoDB quando relacionamentos entre diversas entidades são importantes para o seu sistema. Se for ter de usar muitas "chaves estrangeiras" e "JOINs", você está usando do jeito errado, ou, ao menos, não do jeito mais indicado.

Além disso, diversas entidades de pagamento (como bandeiras de cartão de crédito) não homologam sistemas cujos dados financeiros dos clientes não estejam em bancos de dados relacionais tradicionais. Obviamente isso não impede completamente o uso de MongoDB em sistemas financeiros, mas o restringe apenas a certas partes (como dados públicos).

Além disso, o MongoDB possui alguns concorrentes que possuem variações interessantes que de repente se encaixam melhor como solução ao seu problema. Outro mecanismo bem interessante desta categoria de banco de dados é o RethinkDB, que foca em consultas-push real-time de dados do banco, ao invés de polling como geralmente se faz para atualizar a tela.

Para os amantes de .NET tem o RavenDB, que permite usa a sintaxe do LINQ e das expressões Lambda do C# direto na caixa, com curva de aprendizagem mínima.
Mais uma adição para seu conhecimento: Elasticsearch. Um mecanismo de pesquisa orientado a documentos poderosíssimo quando o assunto é pesquisa textual, foco da ferramenta. Ele é uma implementação do Apache

Lucene, assim como Solr e Sphinx, mas muito superior à esses dois e bem mais popular atualmente também.

Finalizando

Algumas características do MongoDB têm levado a uma adoção sem precedentes deste banco de dados no mercado mundial de desenvolvimento de software. Algumas delas são:

MongoDB armazena os dados em formato JSON/BSON.
Por ser orientado a documentos JSON (armazenados em modo binário, apelidado de JSON), muitas aplicações podem modelar informações de modo muito mais natural, pois os dados podem ser aninhados em hierarquias complexas e continuar a ser indexáveis e fáceis de buscar, igual ao que já é feito em JavaScript.

MongoDB foi criada com Big Data em mente.
Ele suporta tanto escalonamento horizontal quanto vertical usando replica sets (instâncias espelhadas) e sharding (dados distribuídos), tornando-o uma opção muito interessante para grandes volumes de dados, especialmente os desestruturados.

MongoDB é muito leve e é multiplataforma.
Isso permite que você consiga rodar seus projetos em servidores com o SO que quiser, diminuindo bastante seu TCO (principalmente se estava usando Oracle antes e/ou pagava licenças de Windows). A economia com servidores de alguns projetos meus chegou a 80% com a mudança de SQL Server para MongoDB.

Aceitação da tecnologia no ambiente corporativo.
Hoje MongoDB é adotado por muitas empresas, incluindo grandes nomes como Gov.UK, ebay, McAfee, Adobe, Intuit, Citigroup, ADP, HSBC, Forbes, Verizon e Telefonica Digital.

Ferramentas.

Ferramentas modernas de gerenciamento e modelagem específicas para bancos MongoDB como Compass, Robo3T (antigo Robomongo), Studio3T (antigo MongoChef), MongoView e muito mais.

Schemaless.
Dados desestruturados são um problema para a imensa maioria dos bancos de dados relacionais, mas não tanto para o MongoDB. Quando o seu schema é variável, é livre, usar MongoDB vem muito bem a calhar. Os documentos BSON (JSON binário) do Mongo são schemaless e aceitam quase qualquer coisa que você quiser armazenar, sendo um mecanismo de persistência perfeito para uso com tecnologias que trabalham com JSON nativamente, como JavaScript (e consequentemente Node.js).

Muitas, mas muitas empresas e desenvolvedores ao redor do mundo usam MongoDB atualmente.

Eu realmente acredito que você deveria usar também.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Palestra sobre Persistência Poliglota
Palestra que ministrei durante a TechParty, na FACCAT, em Taquara/RS.
https://www.youtube.com/watch?v=Ewvk6t5pkRM

O que é MongoDB?
Palestra que ministrei durante o Telecomptec, na UniLaSalle, em Canoas/RS.
. https://www.youtube.com/watch?v=Ln6XJZj3wYI

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

2 Primeiros Passos

Talk is cheap. Show me the code.
- Linus Torvalds

Existem soluções em nuvem, como o MongoDB Atlas, para você utilizar esse incrível banco de dados sem se preocupar com infraestrutura. No entanto, é muito importante um conhecimento básico de administração local de MongoDB para entender melhor como tudo funciona. Não ficarei aqui em nenhum aspecto de segurança, de alta disponibilidade, de escala ou sequer de administração avançada de MongoDB. Alguns destes itens exploraremos mais tarde. Outros, deixo para você ver junto à documentação oficial no site oficial, onde inclusive você pode se aprofundar mais e tirar as certificações.

Caso ainda não tenha feito isso, acesse o site oficial do MongoDB e baixe gratuitamente a versão mais recente do Community Server (free) para o seu sistema operacional.

https://www.mongodb.com/try/download/community .

Baixe o arquivo compactado e, no caso do Windows, rode o executável que extrairá os arquivos na sua pasta de Arquivos de Programas (não há uma instalação de verdade, apenas extração de arquivos), seguido de uma pasta server/versão, o que é está ok para a maioria dos casos, mas que eu prefiro colocar em C:\Mongo ou dentro de Applications no caso do Mac.

Dentro dessa pasta do Mongo podem existir outras pastas, mas a que nos interessa é a pasta bin. Nessa pasta estão alguns utilitários de linha de comando que no caso do Windows, todos terminam com .exe). Apenas um nos interessa o **mongod** , ele inicializa o servidor de banco de dados.

Para subir um servidor de MongoDB na sua máquina local é muito fácil: execute o utilitário mongod via linha de comando como abaixo, onde

dbpath é o caminho onde seus dados serão salvos (certifique-se que esta pasta exista, para evitar erros).

Código 2.1: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
C : \mongo\bin > mongod -- dbpath C : \mongo\data
```

Atenção: *Note que estou considerando neste caso que você extraiu a pasta do MongoDB dentro de C:\mongo e que para que esta linha de comando funcione você deve navegar (CD) até a pasta bin do Mongo. Conhecimentos mínimos de interface de linha de comando (Windows ou Unix) serão necessários para que você consiga entender tudo.*

Isso irá iniciar o servidor do Mongo e irão correr uma série de comandos pela tela até parar e se tudo deu certo, sem nenhuma mensagem de erro. O servidor está executando corretamente e você já pode utilizá-lo, sem segurança alguma e na porta padrão 27017.

Nota: *se já existir dados de um banco MongoDB na pasta data, o mesmo banco que está salvo lá ficará ativo novamente, o que é muito útil para os nossos estudos.*

Agora vamos falar do terminal cliente, para que possamos nos conectar no servidor e enviar comandos para ele. A partir da versão 6 do MongoDB ele deve ser baixado separadamente e se chama Mongo Shell, disponível na seção Tools da área de downloads do site, link abaixo.

https://www.mongodb.com/products/shell

Baixe, extraia o conteúdo do zip e recomendo copiá-lo (todos arquivos) para dentro da pasta bin da instalação do MongoDB, apenas para fins de organização. O arquivo que nos interessa aqui é o mongosh e vou considerar que ele foi deixado ao lado do arquivo mongod, a partir dos próximos comandos, ok?

Agora abra outro prompt de comando (o anterior ficará executando o servidor) e novamente dentro da pasta bin do Mongo, digite:

Código 2.2: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
c : \mongo\bin > mongosh
```

Este utilitário é o cliente de MongoDB (que se conecta no servidor para fazer consultas). Após a conexão funcionar, se você olhar no prompt onde o servidor do Mongo está rodando, verá que uma conexão foi estabelecida e um sinal de ">" no console 'mongo' indicará que você já pode digitar os seus comandos e queries para enviar à essa conexão.

Ao contrário dos bancos relacionais, no MongoDB você não precisa construir a estrutura do seu banco previamente antes de sair utilizando ele. Tudo é criado conforme você for usando, o que não impede, é claro, que você planeje um pouco o que pretende fazer com o Mongo.

O comando abaixo no terminal cliente mostra os bancos existentes nesse servidor:

Código 2.3: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
show databases
```

Se é sua primeira execução ele deve listar as bases admin e local. Não usaremos nenhuma delas. Agora digite o seguinte comando para "usar" o banco de dados "workshop" (um banco que você sabe que não existe ainda):

Código 2.4: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
use workshop
```

O terminal vai lhe avisar que o contexto da variável "db" mudou para o banco workshop, que nem mesmo existe ainda (mas não se preocupe com isso!). Essa variável "db" representa agora o banco workshop e podemos verificar quais coleções existem atualmente neste banco usando o comando abaixo:

Código 2.5: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
show collections
```

Isso também não deve listar nada, mas não se importe com isso também, é porque ainda não salvamos nada em nosso banco. Assim como fazemos com objetos JavaScript que queremos chamar funções, usaremos a variável 'db' para listar os documentos de uma coleção de customers (clientes) da seguinte forma:

Código 2.6: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.customers.find()
```

find é a função para fazer consultas no MongoDB e, quando usada sem parâmetros (os parâmetros vão entre os parênteses), retorna todos os documentos da coleção. Obviamente não listará nada pois não inserimos nenhum documento ainda, o que vamos fazer agora com a função **insertOne** :

Código 2.7: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . insertOne ( { nome : " Luiz " , idade : 34 } )
```

A função **insertOne** espera um documento JSON por parâmetro com as informações que queremos inserir, sendo que além dessas informações o

MongoDB vai inserir um campo _id automático como chave primária desta coleção.

Nota: *o MongoDB usa JSON como padrão para documentos. Se ainda não conhece esse padrão, que é bem simples, sugiro acessar json.org e dar uma estudada rápida.*

Como sabemos se funcionou?

Além da resposta ao comando insert (insertedId indica o id gerado para este documento), você pode executar o find novamente para ver que agora sim temos customers no nosso banco de dados. Além disso se executar o "show collections" e o "show databases" verá que agora sim possuímos uma coleção customers e uma base workshop nesse servidor.

Tudo foi criado a partir do primeiro **insertOne** e isso mostra que está tudo funcionando bem no seu servidor MongoDB!

MongoDB Database Tools

Antigamente junto do Community Server vinham uma série de utilitários de linha de comando para fazer a gestão das bases de dados. Em algum momento na versão 4 o time do MongoDB achou melhor separar isso e assim nasceu o MongoDB Database Tools, um pacote separado apenas com utilitários de linha de comando para administração.

Você vai precisar destes utilitários para as últimas lições deste livro.

https://www.mongodb.com/try/download/database-tools?tck=docs_databasetools

Baixe gratuitamente para seu sistema operacional no link acima.

MongoDB Compass

Para gerenciar o MongoDB existem uma série de ferramentas de linha de comando que citei antes, a MongoDB Database Tools, mas se quiser uma

ferramenta visual, recomendo a MongoDB Compass, que é feita pelos criadores do MongoDB e é gratuita.

https://www.mongodb.com/products/compass

Recomendo a instalação desta ferramenta pois ela não apenas facilita fazer a gestão dos dados e bases do seu servidor MongoDB como também permite acesso a uma série de ferramentas como importação, exportação, backup, restauração, etc que serão úteis mais tarde.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Como rodar servidor MongoDB?
Vídeo onde ensino passo a passo como subir um servidor MongoDB localmente para desenvolvimento.
https://www.youtube.com/watch?v=2y4mY2PySKk

Vídeo de MongoDB Compass
Vídeo curto sobre uso da ferramenta Compass, que gravei no meu canal.
https://www.youtube.com/watch?v=bL5uH4V3O5o

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

3 CRUD: Create, Read, Update & Delete

Truth can only be found in one place: the code.
— Robert C. Martin

Independente de estar utilizando um banco relacional ou não-relacional, o aprendizado de uma nova tecnologia de persistência de dados sempre envolve iniciar pelo aprendizado do CRUD, acrônimo para Create (criar), Read (ler), Update (atualizar) e Delete (excluir), as quatro operações elementares com dados.

Enquanto que nos bancos relacionais existe a linguagem SQL e os conjuntos de comandos DDL e DML (Data Definition Language e Data Manipulation Language, respectivamente), no MongoDB nós temos um conjunto de funções com sintaxe similar à linguagem JavaScript, para realizar todas operações.

No capítulo anterior, além de colocar um servidor MongoDB pra rodar, executamos alguns comandos muito simples e úteis:

- **show databases** : para listar as bases de dados existentes no servidor;
- **use <nome_da_database>** : para se 'conectar' a uma base específica;
- **show collections** : para listar as coleções de documentos de uma base;
- **db.<nome_da_colecao>.find()** : para retornar todos documentos de uma coleção;
- **db.<nome_da_colecao>.insert(<objeto_json>)** : para inserir um ou mais documentos em uma coleção;
- **db.<nome_da_colecao>.insertOne(<objeto_json>)** : para inserir um único documento em uma coleção e retornar o seu id;

Note que:

- todos os comandos devem ser executados em uma janela do terminal de linha de comando usando o utilitário 'mongo', existente na pasta bin da sua instalação de MongoDB;

- para que o utilitário 'mongo' funcione, já deve existir um servidor MongoDB rodando, em outra janela de terminal (através do utilitário 'mongod');
- comandos de manipulação do servidor são escritos diretamente no terminal de linha de comando, logo após o ponteiro '>' (como em 'show databases');
- você deve usar o comando 'use <nome_da_database>' para poder manipular uma database específica, fazendo com que a variável 'db' esteja apontando para ela;
- você deve usar a sintaxe db.<nome_da_collection>.<comando> para usar um comando em uma collection específica da database

As funções do MongoDB cobrem praticamente todos os comandos SQL existentes e até mesmo fornecem alguns comportamentos inéditos quando se trata de banco de dados. A tabela abaixo traça um comparativo superficial que pode ser muito útil para quem já conhece SQL a mais tempo e gostaria de encontrar alguns equivalentes no Mongo:

SQL	**MongoDB**
SELECT	find e findOne
INSERT	insertOne e insertMany
DELETE	remove
UPDATE	update, replaceOne e updateOne
TOP, LIMIT (dependendo do fornecedor)	limit
ORDER BY	sort
CREATE INDEX	createIndex
LIKE	find usando expressões regulares
CREATE TABLE	não precisa, a coleção é criada ao inserir o primeiro documento

CREATE DATABASE	não precisa, a database é criada quando a primeira coleção é criada
ALTER TABLE	existem diversas funções para alterar a estrutura dos seus documentos
COUNT	count
GROUP BY	aggregate
DISTINCT	distinct

Visto isso, vamos dar sequência ao nosso estudo dos comandos elementares do MongoDB, o famoso CRUD!

Insert

No capítulo anterior vimos rapidamente como inserir um documento em uma coleção. Vamos relembrar como é?

Considerando uma coleção customers (clientes), o uso do comando insertOne (precedido por db.<nome_da_collection>), como um objeto JSON sendo passado por parâmetro,faz com que o mesmo e torne um documento da referida coleção:

Código 3.1: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . insertOne ( { nome : " Luiz " , idade : 34 } )
```

A execução deste comando retorna o id do documento inserido (**insertedId**). Este campo '_id' único e auto incremental é um ObjectId, um objeto alfanumérico longo e complexo, e será sua chave primária. Caso você deseje ter controle da chave primária, deverá passar o campo _id (com o valor que quiser) juntamente com seu objeto.

Além do insert de um documento passado por parâmetro, MongoDB permite realizar inserções em lote passando um array de documentos, como

abaixo (o uso de colchetes indica um array de objetos, enquanto cada par de chaves indica um objeto):

Código 3.2: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
custArray = [ { nome : " Fernando " , idade : 32 } , { nome : " Teste " , uf : " RS " } ]
db . customers . insertMany ( custArray )
```

Atenção: para o nome dos campos dos seus documentos e até mesmo para o nome das coleções do seu banco, use o padrão de nomes JavaScript (camel-case, sem acentos, sem espaços dentro do nome, etc).

Nota: no exemplo acima a variável custArray passa a existir durante toda a sessão do terminal a partir da linha seguinte.

Nesse exemplo passei um array com vários documentos para nossa função insertMany inserir na coleção customers e isso nos trás algumas coisas interessantes para serem debatidas. Primeiro, sim, você pode passar um array de documentos por parâmetro para o insert. Segundo, você notou que o segundo documento não possui "idade"? E que ele possui um campo "uf"?

Isso é perfeitamente aceitável no MongoDB, você pode ter documentos na mesma coleção que não compartilham o mesmo schema. Obviamente para que as collections façam sentido, o ideal é que haja um mínimo de informações em comum entre todos documentos que façam parte da mesma coleção, embora essa regra rígida não exista de fato.

Você verá o efeito disso mais pra frente.

Find

Também no capítulo passado usamos um simples comando find, sem parâmetro algum, para retornar todos documento de uma collection, certo?

Que tal fazermos isso novamente? Para se certificar que todos documentos foram realmente inseridos na coleção customers, use o seguinte comando:

Código 3.3: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find (). pretty ()
```

É o mesmo comando find() que usamos anteriormente, mas com a função pretty() no final para identar o resultado da função no terminal, ficando mais fácil de ler. Use e você vai notar a diferença, principalmente em consultas com vários resultados.

Mas voltando à questão do "uf", que existe somente em um dos documentos, como fica isso nas consultas? E se eu quiser filtrar por "uf"?

Não tem problema!

Essa é uma boa deixa para eu mostrar como filtrar uma consulta baseado em um campo do documento:

Código 3.4: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { uf : " RS " } )
```

Note que a função find pode receber um documento por parâmetro representando o filtro a ser aplicado sobre a coleção para retornar documentos específicos. Nesse caso, disse ao find que retornasse todos os documentos que possuam o campo uf definido como "RS". O resultado no seu terminal deve ser somente o customer de nome "Teste".

Atenção: MongoDB é case-sensitive ao contrário dos bancos relacionais, então cuidado com maiúsculas e minúsculas!

Experimente digitar outros valores ao invés de "RS" e verá que eles não retornam nada, afinal, não basta ter o campo uf, ele deve ser exatamente igual a "RS".

Esse objeto JSON usado como filtro permite que você passe mais de um campo para filtrar, funcionando conforme a lógica AND, ou seja, todos os filtros devem ser atendidos para que a consulta retorne resultados:

O que retornará a consulta abaixo?

Código 3.5: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { uf : " RS " , nome : " Luiz " } )
```

Em tese, essa consulta retornaria todos os customers cujo uf seja RS e o nome seja Luiz, porém na prática, ela não retornará nada, visto que não existe nenhum documento com estas características na coleção customers.

findOne

Além do find comum, o MongoDB possui uma variante que é o findOne. A única diferença prática em relação ao find normal é que o findOne retorna somente a primeira ocorrência de documento que atenda aos filtros. Para consultas que você só vai usar um documento do retorno, o ideal é usar o findOne, visto que ele tende a ser mais rápido pois ao encontrar um documento que atenda aos filtros, ele pára de buscar no restante da coleção.

Filter Operators

Além de campos com valores literais, o parâmetro do find permite usar uma infinidade de operadores. Cada operador possui um funcionamento próprio e todos eles são compostos de uma palavra e iniciam com o símbolo $.

Um deles, bastante utilizado em buscas com campos de texto é o operador $regex que permite, por exemplo, trazer todos documentos que possuam a letra 'a' no nome:

Código 3.6: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { nome : { $regex : / a / i }} )
```

Ou através do atalho abaixo, que faz exatamente a mesma coisa:

```
db . customers . find ( { nome : / a / i } )
```

Se você já mexeu com expressões regulares (regex) em JavaScript antes, sabe exatamente como usar o poder desse recurso junto a um banco de dados, sendo um equivalente muito mais poderoso do que o LIKE dos bancos relacionais.

Nota *: se você não está acostumado com expressões regulares, entre as barras '/' temos a expressão em si, e depois da segunda barra temos modificadores. O modificador 'i' indica case-insensitiveness, ou seja, ignora maiúsculas e minúsculas.*

A maioria dos operadores é usada exatamente desta forma, no lugar do valor absoluto com seu próprio par de chaves, como se fosse um sub-objeto.

Mas e se eu quiser trazer todos os customers maiores de idade? Usa-se o operador **$gte** :

Código 3.7: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { idade : { $gte : 18 }} )
```

O operador $gte (Greater Than or Equal) retorna todos os documentos que possuam o campo idade e que o valor do mesmo seja igual ou superior à 18.

Podemos facilmente combinar filtros absolutos e operadores usando vírgulas dentro do documento passado por parâmetro, assim como citado anteriormente:

Código 3.8: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { nome : " Luiz " , idade : { $gte : 18 }} )
```

O que a expressão acima irá retornar?

Se você disse customers cujo nome sejam Luiz e que sejam maiores de idade, você acertou!

E a expressão abaixo?

Código 3.9: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { nome : { $regex : / a / i } , idade : { $gte : 18 }} )
```

Customers cujo nome contenham a letra 'a' e que sejam maiores de idade, é claro!

Alguns operadores que você pode usar junto ao filtro do find são:

- **$eq** : exatamente igual (=)
- **$ne** : diferente (<> ou !=)
- **$gt** : maior do que (>)
- **$lt** : menor do que (<)
- **$lte** : menor ou igual a (<=)
- **$in** : o valor está contido em um array de possibilidades, como em um OU.
- **$all** : MongoDB permite campos com arrays.

entre outros que veremos ao longo do livro (especialmente $in e $all, que são vistos no capítulo de modelagem)!

Atenção *: evite usar o operador $eq ou qualquer outro operador baseado em negação. Operadores de negação exigem que seja feito um scan completo em todos documentos da coleção elevando o tempo de consulta sempre a um caso ruim (O(n)).*

Assim como dito anteriormente, você também pode usar findOne ao invés de find para retornar apenas o primeiro documento usando operadores de filtro também.

Para finalizar nosso estudo básico de operadores, ainda temos dois que valem a pena ser mencionados.

Primeiro, o **$or** . O operador $or permite que você crie mais de um filtro para seu find, podendo ser completamente diferente um do outro e serão retornados os documentos que atendam qualquer um deles ou até mesmo os dois:

Código 3.10: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { $or : [ { idade : 32 } , { nome : " Teste " } ] } )
```

Neste exemplo prático, serão retornados todos clientes que tenham idade igual a 29 anos ou o nome igual a Teste. Cada filtro do array de filtros do $or (que aqui são dois, mais poderiam ser muito mais) pode ser construído usando tudo que você aprendeu até o momento, incluindo filter-operators.

Existe uma variação deste operador que é o **$nor** , utilizado quando temos um array de filtros que os documentos que vão ser retornados NÃO devem atender (ao contrário do $or).

E por fim, o operador **$not** serve para negar o resultado de uma outra expressão de filtro. Ou seja, algo que seria verdadeiro na consulta, se

tornará falso. Não é algo muito recomendado de se utilizar por questões práticas de lógica (negações costumam gerar confusão nas discussões) e por questões de performance também, pois negações sempre exigem full scan na coleção.

Personalizando retorno de consultas

Existem diversas funções e opções super úteis de serem adicionadas em finds visando personalizar o resultado dos mesmos.
Por exemplo, temos no MongoDB duas funções para fazer paginação de resultados: **skip** e **limit** :

Código 3.11: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find (). skip ( 1 ). limit ( 10 )
```

No exemplo acima retornaremos dez customers (limit) ignorando o primeiro existente na coleção (skip). Você pode usar elas em conjunto como fiz, ou individualmente, para apenas escapar ou apenas limitar a quantidade de resultados.

E para ordenar?

Usamos a função sort no final de todas as outras, com um documento indicando quais campos e se a ordenação por aquele campo é crescente (1) ou decrescente (-1), como abaixo, onde retorno todos os customers ordenados da menor idade para a maior:

Código 3.12: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find (). sort ( { idade : 1 } )
```

Nota: *assim como nos bancos relacionais, os métodos de consulta retornam em ordem de chave primária por padrão, o que neste caso é o _id.*

Para contar quantos elementos há em uma coleção, podemos usar a função count logo após o nome da coleção:

Código 3.13: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . count ()
```

Agora, se quer saber quantos documentos serão retornados por uma consulta, aplique a função count após o find:

Código 3.14: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { nome : " Luiz " } ). count ()
```

Neste caso eu quero saber quantos clientes possuem o nome igual a Luiz.

E por fim, temos as projeções. Todas as consultas que fizemos até agora consideravam que você queria retornar todos os dados dos documentos que atendiam ao filtro. No entanto, sabemos que nem sempre esse é o caso e principalmente no MongoDB, vão haver muitos cenários em que não vamos querer trafegar todos os dados de diversos documentos pela rede. Resolvemos isso definindo uma projection junto ao find para dizer quais dados queremos retornar.

Código 3.15: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . find ( { nome : " Luiz " } , { idade : 1 } )
```

Aqui eu estou pedindo todos os documentos dos clientes cujo nome seja igual a Luiz, mas ao invés de trazer todos os dados de cada cliente, o MongoDB vai me retornar o _id (que sempre vem) e a idade. E mais nada.

Caso você queira todos os campos exceto alguns, pode citar o nome do campo a ser "dispensado" passando -1 ao invés de 1 como eu fiz.

Ok, vimos como usar o find de maneiras bem interessantes e úteis, mas e os demais comandos de manipulação do banco?

Update

Além do insertOne e insertMany que vimos antes, também podemos atualizar documentos já existentes, por exemplo usando o comando replaceOne e derivados. O jeito mais simples de atualizar um documento é chamando a função replaceOne na coleção. Esta função possui dois parâmetros obrigatórios e um opcional:

- filtro para saber qual(is) documento(s) será(ão) atualizado(s);
- novo documento que substituirá o antigo;
- opções de atualização;

Como em:

Código 3.16: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . replaceOne ( { nome : " Luiz " } , { nome : " Luiz " , idade : 32 , uf : " RS " } )
```

Neste comando estou dizendo para substituir (atualizar completamente) somente o primeiro documento cujo nome seja (literalmente) Luiz pelo objeto presente no segundo parâmetro.

Como resultado você deve ter um **modifiedCount** igual a 1, mostrando quantos documentos foram atualizados.

E aqui vão alguns cuidados que você deve ter…

Primeiro, esta função de update substitui completamente o documento filtrado com o JSON passado como segundo argumento. Ou seja, ela exige que você passe o documento completo a ser atualizado no segundo parâmetro, pois ele substituirá o original!

Então vamos a algumas dicas que eu chamo de "update inteligente".

Primeira regra do update inteligente: se você quer atualizar um documento apenas, comece usando a função **updateOne** ao invés de **replaceOne** . O updateOne vai te obrigar a usar operadores de atualização ao invés de um documento inteiro para a atualização, o que é muito mais seguro.

Segunda regra do update inteligente: sempre que possível, use a chave primária (_id) como filtro da atualização, pois ela é sempre única dentro da coleção.

Terceira regra do update inteligente: evite usar updateMany sempre que possível.

Mas que operadores de atualização são esses?

Update Operators

Assim como o **find** possui operadores de filtro, o updateOne, que é a função mais recomendada de update, possui operadores de atualização. Se eu quero, por exemplo, mudar apenas o nome de um customer, eu não preciso enviar todo o documento do respectivo customer com o nome alterado, mas sim apenas a expressão de alteração do nome, como abaixo (já usando o _id como filtro, que é mais seguro):

Código 3.17: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . updateOne ( { _id : ObjectId ( "
59ab46e433959e2724be2cbd " ) } , { $set : { idade : 28 }} )
```

Nota: *para saber o _id correto do seu update, faça um find primeiro e não tente copiar o meu exemplo de _id pois não vai repetir.*

Esta função vai alterar (operador $set) a idade para o valor 28 do documento cujo _id seja "59ab46e433959e2724be2cbd" (note que usei uma função ObjectId para converter, pois esse valor não é uma string).

Nota: *você pode usar null se quiser "limpar" um campo.*

O operador $set recebe um documento contendo todos os campos que devem ser alterados e seus respectivos novos valores. Qualquer campo do documento original que não seja indicado no set continuará com os valores originais.

Atenção: *se o campo a ser alterado não existir no documento, ele será criado.*

Não importa o valor que ela tenha antes, o operador $set vai sobrescrevê-lo. Agora, se o valor anterior importa, como quando queremos incrementar o valor de um campo, não se usa o operador $set, mas sim outros operadores.

A lista dos operadores de update mais comuns estão abaixo:

- **$unset** : remove o respectivo campo do documento;
- **$inc** : incrementa o valor original do campo com o valor especificado;
- **$mul** : multiplica o valor original do campo com o valor especificado;
- **$rename** : muda o nome do campo para o nome especificado;

Atenção: *os operadores $rename, $unset e potencialmente o operador $set, podem ser usados para alterar a estrutura original de um documento (renomeando, removendo ou adicionando campos, respectivamente). No entanto, como não há um schema compartilhando entre todos os documentos de uma coleção, eles alteram apenas o(s) documento(s) englobados no filtro de update utilizado.*

Update Options

Existe um terceiro parâmetro opcional no update e updateOne que são as opções de update. Dentre elas, existe uma muito interessante do MongoDB: **upsert** , como abaixo:

Código 3.18: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . updateOne ( { nome : " LuizTools " } , {$set: { nome : " LuizTools " , uf : " RS " } } , { upsert : true } )
```

Um upsert é um update como qualquer outro, ou seja, vai atualizar o documento que atender ao filtro passado como primeiro parâmetro, porém, se não existir nenhum documento com o respectivo filtro, ele será inserido, como se fosse um insert.

upsert = "update ou insert"

Eu já falei como amo esse banco de dados? :D

Delete

Pra encerrar o nosso conjunto de comandos mais elementares do MongoDB falta o delete.
Existe uma função deleteOne e uma deleteMany, o que a essa altura do campeonato você já deve imaginar a diferença. Também existe uma função remove, mas não aconselho utilizá-la.

Além disso, assim como o find e o update, o primeiro parâmetro dos deletes é o filtro que vai definir quais documentos serão deletados e todos os filtros normais do find são aplicáveis.

Sendo assim, de maneira bem direta:

Código 3.19: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . deleteMany ( { nome : " Luiz " } )
```

Vai excluir todos os clientes cujo nome seja igual a "Luiz", enquanto que na expressão abaixo, no máximo um customer será deletado.

Código 3.20: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db . customers . deleteOne ( { _id : ObjectId ( " abc123def456ghi789 " ) } )
```

Note que para que o deleteOne seja mais preciso, recomenda-se que o filtro utilizado seja um índice único, como o _id, por exemplo.

Simples, não?!

Outras Funções CRUD

Existem outras funções de CRUD além das já citadas nas seções anteriores. A maioria delas é apenas syntax-sugar, ou seja, apenas uma forma diferente de fazer a mesma coisa que outras funções já fazem, geralmente englobando um ou mais comportamentos com uma lógica em cima do documento passado por parâmetro.

De qualquer forma, é interessante conhecê-las, mesmo que de maneira superficial como você verá a seguir:

- **save** : age como se fosse um insertOne, mas se o documento passado por parâmetro possuir um campo _id, age como se fosse um updateOne sobre o documento original com aquele _id.
- **findAndModify** : busca os documentos com o filtro especificado e faz um update deles, retornando a versão original deles ao término da operação. Opcionalmente você pode passar um parâmetro para retornar a versão nova dos documentos;
- **findOneAndDelete** : busca um documento com o filtro especificado e faz um delete nele, retornando a versão original ao término da

operação;

- **findOneAndUpdate** : assim como o findAndModify, mas sobre apenas um documento;

Obviamente existem coisas muito mais avançadas do que esse rápido capítulo de MongoDB, então siga em frente!

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

CRUD MongoDB no Terminal
Neste vídeo eu mostro como usar o terminal de linha de comando para fazer as operações de CRUD.
https://www.youtube.com/watch?v=JBPXpwLj7qc

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

4 Modelagem de Dados

Hofstadter's Law: It always takes longer than you expect,
even when you take into account Hofstadter's Law.
— Douglas Hofstadter

MongoDB possui uma curva de aprendizado incrivelmente simples para dar os primeiros passos e absurdamente complexa para criar arquiteturas de dados realmente escaláveis ao mesmo tempo que atendem às necessidades de negócio das aplicações.

Talvez porque durante anos fomos instruídos a modelar os nossos bancos de dados usando as Formas Normais. Talvez porque durante anos fomos instruídos a modelar as nossas tabelas pensando em não-repetição de dados. E obviamente nada disso é exatamente ruim, apenas não é o modelo que deve ser utilizado com o MongoDB, e por isso é tão difícil para algumas pessoas entenderem como devem modelar seus documentos.

Me basearei aqui em experiências próprias (trabalho com esta tecnologia desde 2015) e mais ao fim do capítulo em padrões consolidados de mercado. Ao invés de ficar falando sobre como você deve modelar ou lhe passando um monte de padrões abstratos, usarei uma didática baseada em use cases famosos e muitas vezes trarei comparações com bancos relacionais, para que consiga entender vantagens e desvantagens.

Para começar, me focarei bastante em um use case famoso, e mais pra frente abrirei o leque de opções.

Preparado para finalmente entender como modelar seus bancos MongoDB?

Princípios importantes

Primeiramente, tenha a mente aberta. Entenda que tudo que você aprendeu nas disciplinas de banco de dados da faculdade estão certas, mas em outro contexto, não nesse. Que as Formas Normais não significam nada aqui.

Eu tive a sorte de ter conhecido o MongoDB em um projeto já maduro que rodava há três anos e tinha muitos dados. Com isso, já aprendi do "jeito certo", pois quando você cria um banco em um projeto pequeno, qualquer modelagem funciona e é difícil de imaginar como algumas decisões realmente podem lhe afetar no futuro.

O primeiro ponto a entender, e que muitas se recusam veemente é que você deve evitar relacionamentos entre documentos diferentes. Apesar dos avanços neste sentido nas últimas versões do MongoDB, este ainda é um banco não-relacional e, portanto, não faz sentido algum modelá-lo pensando em relacionamentos.

O segundo ponto é que documentos não são equivalentes a linhas de banco de dados. Essa é uma comparação muito simplória e que tende a levar ao erro. Documentos são entidades auto-suficientes, com todas as informações que lhes competem. Uma analogia que me ajuda a pensar nos documentos do jeito certo são as INDEXED VIEWS dos bancos relacionais. O plano é que, na maioria das consultas, com um filtro e sem "JOIN" algum, você consiga trazer todos os dados que necessita para montar uma tela de sua aplicação.

O terceiro e último ponto é manter simples. Não é porque o MongoDB permite que você aninhe até 100 níveis de subdocumentos dentro de um documento que você deve fazê-lo. Não é porque o MongoDB permite até 16MB por documento que você deve ter documentos com este tamanho. Não é porque você pode ter até 64 índices por coleção que você deve ter tudo isso. Essas e outras limitações estão lá na documentação oficial.

MongoDB não é magia, encare ele com a mesma seriedade que encara os bancos relacionais e estude bastante. A curva de aprendizagem inicial é realmente mais simples do que SQL, mas a longo prazo, é tão difícil quanto. Como já mencionei no tópico sobre persistência poliglota, os bancos não-relacionais não são melhores que os relacionais. Eles não eliminam os problemas dos relacionais, ao invés disso eles possuem os seus próprios problemas.

Para que você entenda tudo isso na prática, preparei um case simples de banco de dados que é bem comum, enquanto que futuramente espero adicionar novos cases, desta vez mais complexos. Obviamente podem haver variações tanto na implementação relacional citada aqui, quanto a não-relacional que eu sugeri. Se apegue mais às ideias do que aos detalhes e me envie suas considerações pelo Facebook e Twitter (luiztools) para discutirmos o que você não concorda e/ou não entendeu.

Modelagem de Blog: Relacional

Bom, provavelmente você já acesso um blog alguma vez na sua vida, então nada que eu disser aqui será uma novidade para você. Um blog possui basicamente artigos com id, título, data de publicação, conteúdo, categorias, tags, autor, status (rascunho, agendado, publicado, etc) e URL. As categorias possuem id, nome e descrição. Os autores possuem id, nome, foto, permissões, usuário, senha e bio. Além disso, temos os comentários. Ah os comentários, possuem toda uma complexidade própria: autor, data, texto, etc.

Em um banco relacional, como faríamos? (note que uso tabelas no singular e sem prefixos, não se apegue a isso)

Tabela 1: Artigo , com ID (PK), Titulo (NVARCHAR), DataPublicacao (DATETIME), Conteúdo (NVARCHAR), IDAutor (FK), Status (INT - considerando um enumerador) e URL (NVARCHAR). Esta é a tabela principal e como era de esperar, o IDAutor vai referenciar outra tabela. Mas cadê tags e categorias?

Tabela 2: Autor , com ID (PK), Nome (NVARCHAR), Bio (NVARCHAR), Foto (NVARCHAR - porque guardarei apenas o caminho da foto), Usuario (NVARCHAR, unique) e Senha (NVARCHAR)

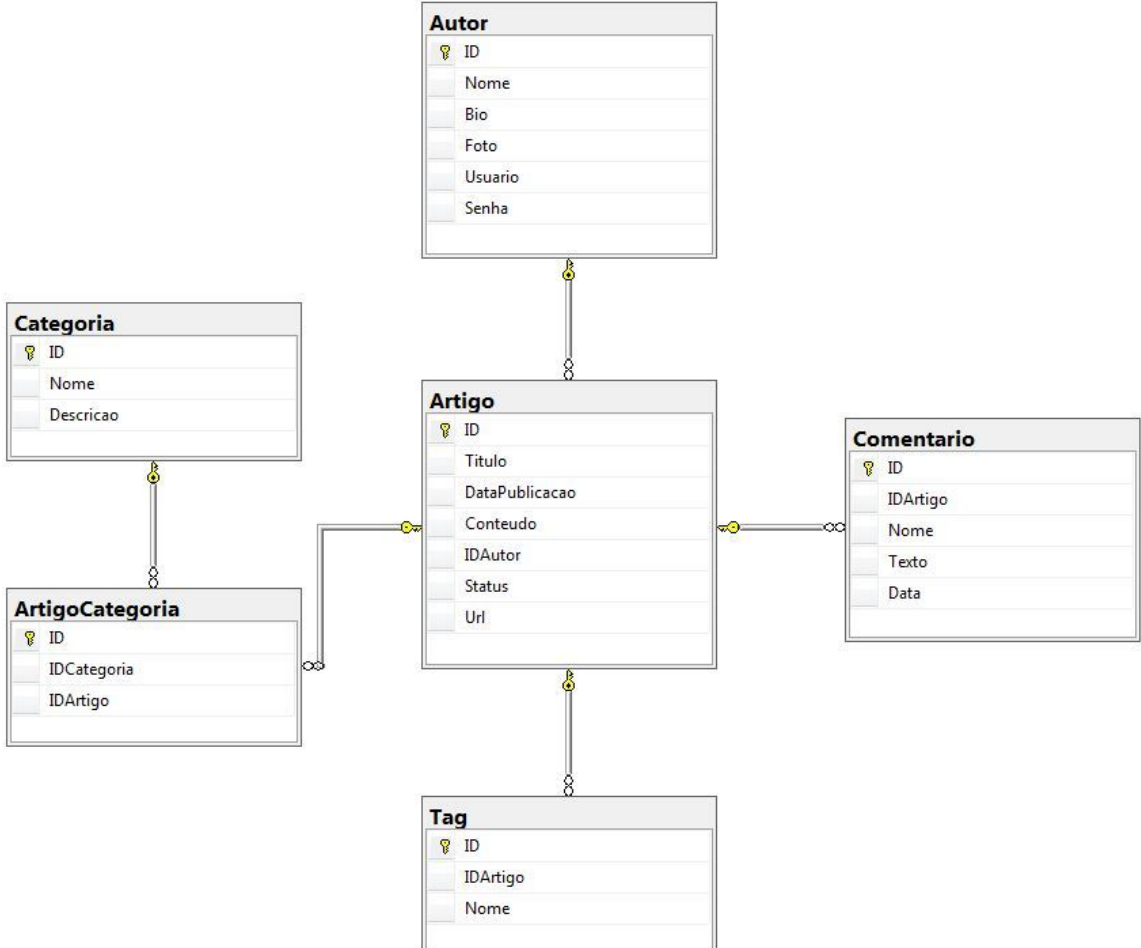
Tabela 3: Categoria , com ID (PK), Nome (NVARCHAR) e Descricao (NVARCHAR). A descrição é útil nas páginas de posts de uma mesma categoria.

Tabela 4: ArtigoCategoria , com ID (PK), IDArtigo (FK), IDCategoria (FK). Um artigo pode ter várias categorias e cada categoria pode estar em vários artigos, o clássico "N para N", um câncer dos bancos relacionais na minha opinião.

Tabela 5: Tag, com ID (PK), IDArtigo (FK), Nome (NVARCHAR). Aqui podemos fazer um "1 para N", onde cada artigo pode ter várias tags ou um "N para N", se quiser a não-repetição dos nomes de tags.

Tabela 6: Comentario , com ID (PK), IDArtigo (FK), Nome (NVARCHAR - autor do comentário), Texto (NVARCHAR - o comentário em si), Data (DATETIME).

Pode ser mais complexo e poderoso que isso, mas vamos manter assim, visto que o objetivo aqui não é modelar o melhor banco de dados para blogs do mundo, mas sim apenas para mostrar como uma modelagem dessas pode ser feita em um banco relacional e em um não-relacional no mesmo capítulo.



Considerando o banco descrito acima, como faríamos para montar uma tela de uma postagem completa no site do blog?

```
SELECT Artigo .*, Autor . Nome , Autor . ID FROM Artigo
INNER JOIN Autor ON Autor . ID = Artigo . IDAutor
WHERE Artigo . URL = 'XXX'
```

Essa consulta traz as informações do Artigo e de seu Autor (só o que importa do autor para a página do post). Mas e as tags e categorias? Precisamos de mais duas consultas para isso:

```
SELECT Categoria .* FROM Categoria
INNER JOIN ArtigoCategoria ON ArtigoCategoria . IDCategoria =
Categoria . ID
INNER JOIN Artigo ON Artigo . ID = ArtigoCategoria . IDArtigo
WHERE Artigo . URL = 'xxx' ;

SELECT Tag .* FROM Tag
INNER JOIN Artigo ON Artigo . ID = Tag . IDArtigo
WHERE Artigo . URL = 'xxx' ;
```

Puxa, esqueci dos comentários ainda, certo? Vamos trazê-los também!

```
SELECT Comentario .* FROM Comentario
INNER JOIN Artigo ON Artigo . ID = Comentario . IDArtigo
WHERE Artigo . URL = 'xxx'
```

Com isso conseguimos finalmente montar uma página bem simples, com apenas uma postagem de um blog. Precisamos de 4 consultas com 5 JOINs diferentes.

Complicado, não?!

E no MongoDB, como ficaria isso?

Modelagem de Blog: Não-Relacional

Em bancos de dados não-relacionais, o primeiro ponto a considerar é que você deve evitar relacionamentos, afinal, se for pra usar um monte de FK, use SQL e seja feliz!

A maioria das relações são substituídas por composições de sub-documentos dentro de documentos, mas vamos começar por partes. Primeiro, vamos criar um documento JSON que represente o mesmo artigo

do exemplo anterior com SQL (dados de exemplo apenas), como pertencendo a uma coleção Artigo:

Código 4.1: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
  titulo : " Tutorial MongoDB " ,
  dataPublicacao : new Date ( " 2016-05-18T16:00:00Z " ),
  conteudo : " Be-a-bá do MongoDB " ,
  idAutor : ObjectId ( " 456def " ),
  status : 1 ,
  url : " http://www.luiztools.com.br/post/1 "
}
```

Primeira diferença: aqui os IDs não são numéricos auto-incrementais, mas sim ObjectIds auto-incrementais. Segunda diferença: não há schema rígido, então eu posso ter artigos com mais informações do que apenas estas, ou com menos. Mas e aquele idAutor ali?

Se eu colocar daquele jeito ali, com idAutor, quando eu for montar a tela do artigo eu sempre teria de ir na coleção de Autores para pegar os dados do autor, certo? Mas se eu sei quais informações eu preciso exibir (apenas nome e id, assim como na consulta SQL), o certo a se fazer aqui é reproduzir estas informações como um subdocumento de artigo, como abaixo:

Código 4.2: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
  titulo : " Tutorial MongoDB " ,
  dataPublicacao : new Date ( " 2016-05-18T16:00:00Z " ),
  conteudo : " Be-a-bá do MongoDB " ,
  autor : {
```

```
    _id : ObjectId ( " 456def " ),
    nome : " Luiz "
  } ,
  status : 1 ,
  url : " http://www.luiztools.com.br/post/1 "
}
```

Esse é o jeito "certo" de fazer com MongoDB! Note que o subdocumento autor possui apenas os dados necessários para montar uma tela de artigo. Mas se precisarmos depois da informação completa do autor, podemos pegá-la a partir de seu id.

Como MongoDB não garante integridade referencial por não possuir FK, temos de tomar muito cuidado quando você for excluir um autor, para desvinculá-lo em todos os artigos dele, e quando você for atualizar o nome dele, para atualizar em todos artigos dele.

Já a coleção Autor, fica com documentos bem parecidos com a tabela original em SQL:

Código 4.3: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 456def " ),
  nome : " Luiz Duarte " ,
  bio : " Empreendedor, professor, etc " ,
  foto : " 1.jpg " ,
  usuario : " luiz " ,
  senha : " ds6dsv8ds5v76sd5v67d5v6 "
}
```

Note que aqui eu não vou embutir todos os artigos deste autor, pois seria inviável. O foco do blog são os artigos, não os autores, logo, os artigos vão conter o seu autor dentro de si, e não o contrário!

Um adendo: MongoDB trabalha muito bem com arquivos binários embutidos em documentos, então se quisesse usar um binário de imagem no campo foto, funcionaria perfeitamente bem, melhor que no SQL tradicional (BLOBs, arghhh!).

Mas e as categorias e tags? Como ficam?

Primeiro que aqui não precisamos de tabelas-meio quando temos relacionamento N-N. Todos os relacionamentos podem ser representados na modalidade 1-N usando campos multivalorados, como mostrado no exemplo abaixo, de uma v3 do documento de artigo:

Código 4.4: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
  titulo : " Tutorial MongoDB " ,
  dataPublicacao : new Date ( " 2016-05-18T16:00:00Z " ),
  conteudo : " Be-a-bá do MongoDB " ,
  autor : {
    _id : ObjectId ( " 456def " ),
    nome : " Luiz "
  } ,
  status : 1 ,
  url : " http://www.luiztools.com.br/post/1 " ,
  categorias : [ {
    _id : ObjectId ( " 789ghi " ),
    nome : " cat1 "
  } ],
  tags : [ " tag1 " , " tag2 " ]
}
```

Tags são apenas strings tanto na modelagem relacional original quanto nesta modelagem orientada a documentos. Sendo assim, um campo do tipo array

de String resolve este cenário sem maiores problemas.

Já as categorias são um pouco mais complexas, pois podem ter informações extras como uma descrição. No entanto, como para exibir o artigo na tela do sistema nós só precisamos do nome e do id das categorias, podemos ter um array de subdocumentos de categoria dentro do documento de artigo.

Em paralelo deveremos ter uma coleção de categorias, para armazenar os demais dados, mantendo o mesmo ObjectId:

Código 4.5: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 789ghi " ),
  nome : " cat1 " ,
  descricao : " Categoria bacana "
}
```

Novamente, caso no futuro você venha a excluir categorias, terá de percorrer toda a coleção de artigos visando remover as ocorrências das mesmas. No caso de edição de nome da categoria original, também. Com as tags não temos exatamente este problema, uma vez que elas são mais dinâmicas.

Para encerrar, temos os comentários. Assim como fizemos com as categorias, vamos embutir os documentos dos comentários dentro do documento do artigo ao qual eles fazem parte, afinal, não faz sentido algum eles existirem alheios ao artigo do qual foram originados.

Código 4.6: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
  titulo : " Tutorial MongoDB " ,
  dataPublicacao : new Date ( " 2017-10-18T16:00:00Z " ),
```

```
  conteudo : " Be-a-bá do MongoDB " ,
  autor : {
    _id : ObjectId ( " 456def " ),
    nome : " Luiz "
  } ,
  status : 1 ,
  url : " http://www.luiztools.com.br/post/1 " ,
  categorias : [ {
    _id : ObjectId ( " 789ghi " ),
    nome : " cat1 "
  } ],
  tags : [ " tag1 " , " tag2 " ],
  comentarios : [ {
    _id : ObjectId ( " 012jkl " ),
    nome : " Hater " ,
    texto : " Não gosto do seu blog " ,
    data : new Date ( " 2017-10-18T18:00:00Z " )
  } ]
}
```

Note que aqui optei por ter um _id no comentário, mesmo ele não possuindo uma coleção em separado. Isso para que seja possível mais tarde moderar comentários.

E esse é um exemplo de documento completo, modelado a partir do problema de um site de artigos, blog, wiki, etc.

Ah, mas e como fazemos a consulta necessária para montar a tela de um artigo? Lembra que em SQL eram inúmeras consultas cheias de INNER JOINs? No MongoDB, precisamos de apenas uma consulta, bem simples, para montar a mesma tela:

Código 4.7: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . findOne ( { url : ' http://www.luiztools.com.br/post/1 ' } ) ;
```

Essa consulta traz os mesmos dados que todas aquelas que fiz em SQL juntas. Isso porque a modelagem dos documentos é feita sem relacionamentos externos, o documento é auto-suficiente em termos de dados, como se fosse uma VIEW do SQL, mas muito mais poderoso.

Claro, existem aqueles pontos de atenção que mencionei, sobre updates e deletes, uma vez que não há uma garantia nativa de consistência entre coleções. Mas lhe garanto, a performance de busca é incomparável, uma vez que a complexidade é baixíssima. Você inclusive pode criar um índice no campo url da coleção Artigo, para tornar a busca ainda mais veloz, com o comando abaixo:

Código 4.8: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . createIndex ( { url : 1 } ) ;
```

Demais, não?!

Você pode estar preocupado agora com a redundância de dados, afinal diversas strings se repetirão entre os diversos anúncios, como o nome do autor, tags, nomes de categoria, etc. Não se preocupe com isso, o MongoDB trabalha muito bem com volumes muito grandes de dados e desde que você tenha bastante espaço em disco, não terá problemas tão cedo. Obviamente não recomendo que duplique conteúdos extensos, mas palavras como essas que ficaram redundantes na minha modelagem não são um problema.

No entanto, lidar com subdocumentos e campos multivalorados adiciona um pouco mais de complexidade à manipulação do MongoDB, e tratarei disso logo adiante.

CRUD com Subdocumentos

No terceiro capítulo deste livro nós vimos como buscar, inserir, atualizar e excluir documentos em coleções MongoDB. No entanto, sempre com

documentos planos, sem níveis, a forma mais básica de armazenar dados.

Mas e quando temos um subdocumento dentro de outro documento, assim como o autor dentro de um artigo de blog?

Código 4.9: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
  titulo : " Artigo 1 " ,
  autor : {
    _id : ObjectId ( " 456def " ),
    nome : " Luiz "
  }
  tags : [ " NodeJs " , " MongoDB " ]
}
```

Vamos começar pelo C do CRUD, que no caso do MongoDB é representado pelo método insert. O comando abaixo insere um novo artigo incluindo o subdocumento 'autor':

Código 4.10: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . insertOne ( { titulo : " Artigo 1 " , autor : { _id : ObjectId ( " 456def " ), nome : " Luiz " } , tags : null } )
```

Note que eu não passo o _id do artigo pois ele é autoincremental e controlado pelo próprio MongoDB. Já no caso do autor, ele pertence à outra coleção, a de autores, e o _id que está junto dele deve ser o mesmo do autor original na sua referida coleção. Imagina-se que em uma aplicação que salve um artigo, que a informação do autor do artigo esteja em sessão ou em um componente de tela para ser passada ao MongoDB corretamente.

Falando do R do CRUD, o find no MongoDB, podemos facilmente usar campos de subdocumentos como filtros em nossas consultas, como segue:

Código 4.11: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . find ( { " autor.nome " : " Luiz " } )
```

Essa consulta retorna todos os artigos cujo nome do autor seja literalmente Luiz. Qualquer filtro existente pode ser usado aqui, sobre qualquer campo do subdocumento autor, mas atenção à forma que referenciei o campo, usando o nome do subdocumento, seguido de um '.', e depois o nome do campo. Repare também que neste caso o uso de aspas ao redor da expressão é obrigatório.

Seguindo com o U do CRUD, vale tudo o que vimos até agora. Para substituir documentos que possuam subdocumentos usando o comando updateOne, você tem de passar eles também, para não se perderem na atualização, como abaixo:

Código 4.12: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . updateOne ( { _id : ObjectId ( " 456def " ) } ,{$set: { titulo :
" Artigo 1 " , autor : { nome : " Luiz " } , tags : null } } )
```

Se for usar um campo de um subdocumento no filtro do update, valem as mesmas regras do filtro do find que expliquei anteriormente. O mesmo vale caso queira aplicar algum update operator em um campo de um subdocumento. Quer um exemplo prático?

Na modelagem de blog que fizemos no tópico anterior, replicamos o nome e _id do autor em todos os posts escritos por ele. Mas o que acontece caso o nome do autor seja alterado no documento original dele, que fica na coleção de autores?

Teremos de replicar esta alteração em todos os artigos que tenham sido escritos por aquele autor, como abaixo. Neste exemplo, suponha que o autor Luiz teve o nome alterado para Luiz Fernando, então temos de atualizar todos os posts escritos por ele. Como é somente esta pequena informação que mudou, usaremos o update-operator **$set** , para não ter de sobrescrever os documentos inteiros.

Código 4.13: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . updateOne ( { " autor.nome " : " Luiz " } , { $set : { " autor.nome " : " Luiz Fernando " }} )
```

Para um update mais preciso, eu poderia substituir o filtro autor.nome por autor._id, considerando que nomes de autores podem ser repetir em um mesmo blog.

Com os update-operators **$set** , **$unset** e **$rename** é possível manipular os campos dos subdocumentos também, da mesma forma que faria com o documento principal, apenas usando a notação "subdocumento.campo".

Finalizando o CRUD com subdocumentos, o D de delete é realizado usando as mesmas regras de filtro do find, caso queira excluir todos os documentos que possuam um valor específico em um campo de um subdocumento. Sem mistério algum.

CRUD com campos multivalorados

Outra preocupação é com a manipulação de elementos em campos multi-valorados, algo inexistente em bancos relacionais que sempre assumem relações 1-N ou N-N nestes casos. Salvo gambiarras que já vi por aí de salvar strings CSV ou JSON em coluna de registro e outras loucuras que sempre acabam gerando dor de cabeça pois não são pesquisáveis.

Começando pelo C do CRUD, o insert do MongoDB funciona de maneira muito óbvia para campos multivalorados: apenas passe null para nenhum

valor ou o array (entre colchetes) de valores iniciais daquele elemento (sim, o MongoDB permite que depois você adicione ou remova elementos).

Se você procurar no exemplo anterior de insert, verá que passei null no campo multivalorado de tags do artigo. A outra opção, passando valores iniciais, segue abaixo:

Código 4.14: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . insertOne ( { titulo : " Artigo 1 " , autor : { nome : " Luiz " }
, tags : [ " NodeJs " , " MongoDB " ] } )
```

Neste caso o campo multivalorado tags é um array de strings. Caso deseje, você pode inserir um documento que possua campos multivalorados de documentos também, como no caso das categorias que modelamos no tópico anterior:

Código 4.15: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . insertOne ( { titulo : " Artigo 1 " , autor : { nome : " Luiz " }
, tags : [ " NodeJs " , " MongoDB " ], categorias : [ { _id : ObjectId ( "
abc123 " ), nome : " Desenvolvimento " } ] } )
```

Mas é quando entramos no R do CRUD com campos multivalorados em MongoDB que começamos a explorar um novo universo de possibilidades e novos filter-operators da função find.

Considere que temos três artigos salvos na nossa base MongoDB (omitirei os campos que não nos importam no momento):

Código 4.16: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  _id : ObjectId ( " 123abc " ),
```

```
  titulo : " Artigo 1 " ,
  tags : [ " NodeJs " , " MongoDB " ],
  categorias : [ { _id : 1 , nome : " Desenvolvimento " } , { _id : 2 , nome : " Banco de Dados " } ]
} ,
{
  _id : ObjectId ( " 456def " ),
  titulo : " Artigo 2 " ,
  tags : [ " NodeJs " ],
  categorias : [ { _id : 1 , nome : " Desenvolvimento " } ]
} ,
{
  _id : ObjectId ( " 789ghi " ),
  titulo : " Artigo 3 " ,
  tags : [ " MongoDB " ],
  categorias : [ { _id : 2 , nome : " Banco de Dados " } ]
}
```

Para fazer buscas usando campos multivalorados como filtro é muito simples: você deve usar os operadores de filtro $all e $in. Exemplo de consulta por todos os artigos que possuam todas (ALL) as seguintes tags NodeJs e MongoDB :

Código 4.17: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . find ( { tags : { $all : [ " NodeJs " , " MongoDB " ] }} ) ;
```

O operador $all vai garantir que só sejam retornados artigos que possuam no mínimo as duas tags informadas (somente o Artigo 1), equivalente à consulta SQL abaixo:

```
SELECT Artigo .* FROM Artigo
INNER JOIN Tag ON Tag . IDArtigo = Artigo . ID
```

```
WHERE Tag . Nome = 'NodeJs'
INTERSECT
SELECT Artigo .* FROM Artigo
INNER JOIN Tag ON Tag . IDArtigo = Artigo . ID
WHERE Tag . Nome = 'MongoDB'
```

Agora um exemplo de consulta por todos os artigos que possuam uma (IN) das seguintes tags NodeJs ou MongoDB:

Código 4.18: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . Artigo . find ( { tags : { $in : [ " NodeJs " , " MongoDB " ] }} ) ;
```

Neste caso todos os três artigos serão retornados, assim como no equivalente SQL abaixo (coloquei reticências por preguiça de preencher todas as colunas):

```
SELECT Artigo . ID , Artigo . Titulo ... FROM Artigo
INNER JOIN Tag ON Tag . IDArtigo = Artigo . ID
WHERE Tag . Nome IN ( "NodeJs" , "MongoDB" )
GROUP BY Artigo . ID , Artigo . Titulo ...
```

Mas agora se quisermos apenas os artigos que possuam ao menos uma categoria (o campo de categorias é multivalorado, lembra?) com o nome Desenvolvimento?

Quando aplicamos um filtro sobre um campo que é multivalorado o MongoDB entende que podemos passar um novo filtro a seguir, que será aplicado aos campos do subdocumento. Por exemplo:

Código 4.19: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . artigos . find ( { categorias : { nome : " Desenvolvimento " }} )
```

Vai retornar todos os documentos que tenham ao menos uma categoria cujo nome seja 'Desenvolvimento'. Também poderia usar filter-operators se fosse necessário como **$gte** , **$regex** , etc.

Avançando para o U do CRUD, no update de campos multivalorados também temos diversas opções a serem exploradas pois é no update que adicionamos e removemos elementos do nosso array, quando quisermos fazê-lo. Para tais tarefas deve-se usar os operadores **$pull** (remove o elemento do array) e **$push** (insere o elemento no array), sempre dentro de um comando de update, como segundo parâmetro (update-operator). Ex:

Código 4.20: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
//remove a string tag1 do array tags
> db . artigos . updateOne ( { _id : 1 } , { $pull : { tags : " tag1 " }} )

//adiciona a string tag1 no array tags
> db . artigos . updateOne ( { _id : 1 } , { $push : { tags : " tag1 " }} )
```

Pode parecer um pouco estranho no início, mas na verdade é muito mais simples, pois é a mesma ideia que já fazemos há anos com coleções em linguagens de programação, onde geralmente temos métodos como add e remove ou assemelhados.

Finalizando, o D do CRUD, em campos multivalorados funciona da mesma forma que o find, considerando que no **deleteOne** e **deleteMany** do MongoDB também passamos um filtro por parâmetro.

Padrões para Modelagem em MongoDB

Uma pergunta muito frequente que recebo é como modelar corretamente bases MongoDB. A resposta mais honesta é que depende.

Sua aplicação faz mais leitura que escritas? Que dados precisam estar agrupados quando é feito uma leitura? Quais são as preocupações com performance que devemos ter na sua aplicação? O quão grande serão os documentos? O quão grande será a base como um todo?

Todas essas perguntas, e muitas outras, definem como devemos projetar nossos schemas em MongoDB. Sim, MongoDB é schemaless, mas ainda assim, a maior parte dos problemas de performance (e frustrações dos desenvolvedores) que usam MongoDB advém de schemas ruins. Então, sim, você deve pensar sobre seu schema e os padrões que vou mostrar a seguir são justamente pra te ajudar com isso.

Construindo com padrões

Assim como na engenharia de software nós temos os Padrões de Projeto (Design Patterns no original), o time do MongoDB resolveu compilar o conhecimento acumulado da última década de uso deste banco não relacional em alguns padrões de modelagem comuns e reutilizáveis, o que eles chamam de Building With Patterns (construindo com padrões).

Nesta seção, vou trazer os padrões descritos pelo time do MongoDB, mas não apenas traduzindo-os, mas tentando trazer de uma maneira mais didática cada um dos padrões. Começando pela matriz abaixo que eles desenvolveram e que serve como uma linha guia, mas que facilita bastante para consultas rápidas. No eixo horizontal, os padrões de modelagem encontrados pelo time do MongoDB, e no vertical, os casos de uso mais comuns destes padrões (falei deles lá no capítulo 1, lembra?).

Use Case Categories

Patterns	Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Approximation	✅		✅	✅		✅	
Attribute	✅	✅					✅
Bucket			✅			✅	
Computed	✅		✅	✅	✅	✅	✅
Document Versioning	✅	✅			✅		✅
Extended Reference	✅			✅		✅	
Outlier			✅	✅	✅		
Preallocated			✅			✅	
Polymorphic	✅	✅		✅			✅
Schema Versioning	✅	✅	✅	✅	✅	✅	✅
Subset	✅	✅		✅	✅		
Tree and Graph	✅	✅					

Claro que a palavra final sobre o uso ou não de um padrão deve partir única e exclusivamente da sua análise, mas dar uma olhada na tabela acima pode lhe ajudar a guiar os seus estudos quando estiver modelando o schema da sua base para o seu use case em específico. Por exemplo, se eu estivesse criando uma aplicação de marketing digital agora (Personalization) eu daria uma olhada mais aprofundada nos padrões Computed, Document Versioning, Outlier, Schema Versioning e Subset.

Os padrões que nós veremos são:

- **Approximation** : para maior performance em escrita de estatísticas mas com menos precisão, adicionando uma camada de controle na

aplicação;

- **Attribute** : para maior performance em buscas de campos similares que estão sofrendo com múltiplos índices, tornando-os um atributo array;
- **Bucket** : para reduzir o tamanho da base que cresce rapidamente com registros em real time, agregando-os em documentos;
- **Computed** : para aumentar a performance de consulta de agregações de dados que podem ser pré-computados;
- **Document Versioning:** para manter histórico de todas versões de um documento, sem prejudicar a performance de consulta;
- **Extended Reference:** para diminuir os JOINs entre coleções mantendo cópias de dados essenciais e que não mudam com frequência nos documentos principais;
- **Outlier** : para tratar exceções em documentos sem ferir o seu schema que atende a maioria da coleção e sem agredir a performance;
- **Pre-allocation:** para ganho de tempo de escrita (em versões anteriores a MongoDB 3.2), pré-alocando o espaço para documentos com tamanho de campos previsível;
- **Polymorphic** : para agrupar documentos similares, mas não idênticos, na mesma coleção;
- **Schema Versioning:** para permitir que diferentes versões de schema consigam coexistir na mesma coleção de documentos;
- **Subset:** para ganho de performance quando temos arrays internos grandes demais nos documentos e nem sempre usamos todos os itens do array;
- **Tree:** para armazenamento de dados hierárquicos de maneira performática;

E agora vamos aos padrões!

Approximation

Este padrão, Aproximação, é útil quando temos que manter registro de estatísticas em grande volume na aplicação, estamos sofrendo com a quantidade de escritas e não precisamos que estas estatísticas sejam exatas, podem ser aproximadas. Ex: população de países, pageviews de um grande portal, número de resultados de pesquisa de um grande buscador, etc.

Para fazer isto, a aplicação deve armazenar as estatísticas recentemente recebidas em um cache por exemplo e somente escrever no banco de dados quando atingir um limiar aceitável (por ex, 100). Usando este limiar hipotético de 100:1, conseguimos reduzir drasticamente (em 99 vezes, neste caso) as escritas no MongoDB, liberando sua performance para operações mais importantes.

A desvantagem deste padrão é que as estatísticas não serão exatas e eventualmente pode haver pequena perda de dados caso ocorra falha no cache. Por isso é importante que seja usado somente em casos que possamos ter estatísticas aproximadas.

Attribute

Este padrão, Atributo, é útil quando temos diversos campos em um documento que são usados em filtros de consulta e estamos sofrendo com a quantidade de índices necessários para que as consultas performem com velocidade. Ex: as diferentes datas de lançamento de um filme internacional, as diferentes unidades de medida de um produto internacional, as diferentes traduções de um texto, etc.

```
{
   title: "Star Wars",
   director: "George Lucas",
   ...
   release_US: ISODate("1977-05-20T01:00:00+01:00"),
   release_France: ISODate("1977-10-19T01:00:00+01:00"),
   release_Italy: ISODate("1977-10-20T01:00:00+01:00"),
   release_UK: ISODate("1977-12-27T01:00:00+01:00"),
   ...
}
```

Se um conjunto destes campos possuem características em comum, como o mesmo tipo, podemos agrupá-los em um atributo do tipo array, criando apenas um índice nele. Por exemplo, em uma base de filmes de cinema, ao invés de termos atributos de lançamento para cada país, criamos um

atributo "releases" como um array de subdocumentos estilo chave-valor, onde a chave seria o país e o valor seria a data de lançamento naquele país.

```
{
   title: "Star Wars",
   director: "George Lucas",
   …
   releases: [
      {
      location: "USA",
      date: ISODate("1977-05-20T01:00:00+01:00")
      },
      {
      location: "France",
      date: ISODate("1977-10-19T01:00:00+01:00")
      },
      {
      location: "Italy",
      date: ISODate("1977-10-20T01:00:00+01:00")
      },
      {
      location: "UK",
      date: ISODate("1977-12-27T01:00:00+01:00")
      },
      …
   ],
   …
}
```

A contra-indicação deste padrão é se você não aplicá-lo no conjunto de atributos correto, que façam sentido serem agrupados em um atributo só.

Bucket

Este padrão, Balde, é útil quando temos um alto fluxo de dados de dados em tempo real e estamos sofrendo com a escrita elevada e o o crescimento

rápido da base, o que acaba afetando as consultas. Ex: sensores que enviam dados em real time, monitoramento de recursos de servidores, etc.

```
{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:00:00.000Z"),
  temperature: 40
}

{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:01:00.000Z"),
  temperature: 40
}

{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:02:00.000Z"),
  temperature: 41
}
```

Ao invés de escrever cada registro em um documento separado, criamos "baldes" de registros baseados em timestamp. Ou seja, se você recebe um registro por segundo, ao invés de termos 60 documentos por minuto, podemos ter um documento por minuto compreendendo o timestamp dos últimos 60 segundos.

Essa abordagem não reduz tanto a escrita quanto o padrão Approximation, pois estaremos escrevendo na mesma quantidade (mas não no mesmo volume de dados), mas mantém a fidelidade das estatísticas no detalhe.

```
{
   sensor_id: 12345,
   start_date: ISODate("2019-01-31T10:00:00.000Z"),
   end_date: ISODate("2019-01-31T10:59:59.000Z"),
   measurements: [
```

```
    {
    timestamp: ISODate("2019-01-31T10:00:00.000Z"),
    temperature: 40
    },
    {
    timestamp: ISODate("2019-01-31T10:01:00.000Z"),
    temperature: 40
    },
    …
    {
    timestamp: ISODate("2019-01-31T10:42:00.000Z"),
    temperature: 42
    }
  ],
  transaction_count: 42,
  sum_temperature: 2413
}
```

A desvantagem é apenas a lógica de escrita que fica um pouco mais complexa, pois ao invés de simplesmente sair salvando novos documentos, você tem de guardar a estatística no "balde" certo.

Computed

Este padrão, Computado, é útil quando temos a necessidade e ter dados agregados em bases com grande volume, fazendo com que a performance de agregações não seja boa. Ex: soma de vendas por produto, visitantes de um grande portal por país, ocorrência de palavras em volumes de texto ou no exemplo abaixo, ingressos de cinema vendidos e receita por filme.

```
{
  "ts": DateTime(xxx),
  "theather": "Overland Park Cinema",
  "location": "Boise, ID",
  "movie_title": "Jack Ryan: Shadow Recruit",
  "num_viewers": 760,
```

```
  "revenue": 7600
}

{
  "ts": DateTime(xxx),
  "theather": "City Cinema",
  "location": "New York, NY",
  "movie_title": "Jack Ryan: Shadow Recruit",
  "num_viewers": 1496,
  "revenue": 22440
}

{
  "ts": DateTime(xxx),
  "theather": "Alger Cinema",
  "location": "Lakeville, OR",
  "movie_title": "Jack Ryan: Shadow Recruit",
  "num_viewers": 344,
  "revenue": 3440
}
```

Para resolver este problema, deixamos os dados pré-computados para consulta posterior, com duas alternativas diferentes.

Alternativa 1, se a escrita não é frequente, podemos ter campos computados no documento principal que, a cada atualização, são recalculados. Por exemplo, um campo num_viewers em um documento de filme de cinema que é incrementado a cada nova venda de ingresso daquele filme.

Alternativa 2, se a escrita é frequente, a aplicação pode controlar um intervalo de tempo em que ela vai realizar as agregações e deixar salvo, provavelmente em uma base específica para armazenar estas computações que serão analisadas mais tarde.

```
{
  "ts": DateTime(xxx),
  "movie_title": "Jack Ryan: Shadow Recruit",
```

```
  "num_viewers": 2600,
  "revenue": 33480
}
```

Este é um padrão que adiciona relativa complexidade na aplicação, então não deve ser usado exceto quando realmente necessário.

Document Versioning

Este padrão, Versionamento de Documento, é útil quando você precisa manter versões anteriores de um mesmo documento, para fins de histórico, mas o mais comum de ser acessado é o atual. Ex: contratos e leis que são alteradas ao longo do tempo mas que não podem ser simplesmente sobrescritas/atualizadas.

Para resolver este problema, cria-se uma coleção para fins de histórico do documento em questão e coloca-se um campo para versionamento do mesmo (um id, uma data, etc). Na coleção principal, fica sempre a versão mais recente do documento, para consulta rápida, e na coleção secundária, todas as versões, sendo esta uma coleção muito maior e consequentemente mais lenta para consulta.

```
//único documento na coleção principal para este cliente
{
  _id: ObjectId<ObjectId>,
  name: 'Bilbo Baggins',
  revision: 2,
  items_insured: ['Elven Sword', 'One Ring'],
  ...
}

//documentos existentes na coleção de arquivo
{
  _id: ObjectId<ObjectId>,
  name: 'Bilbo Baggins',
  revision: 1,
  items_insured: ['Elven Sword'],
```

```
  ...
}

{
  _id: ObjectId<ObjectId>,
  name: 'Bilbo Baggins',
  revision: 2,
  items_insured: ['Elven Sword', 'One Ring'],
  ...
}
```

Como desvantagens deste padrão temos o dobro de escritas (sempre uma vez na coleção corrente e outra na coleção arquivo) e também deve ajustar sua aplicação para sempre pegar o documento da coleção certa.

Extended Reference

Este padrão, Referência Extendida, é útil quando você teria de fazer diversos JOINs no modelo relacional para ter os dados que precisa que sejam exibidos juntos no sistema com grande frequência, juntando dados de diversas tabelas. Ex: catálogo de produtos, ordem de compra em um ecommerce, prontuário médico de paciente, etc.

Para resolver este problema, mantém-se a divisão em várias coleções, porém agregam-se os dados mais necessários de maneira duplicada no documento principal. Por exemplo, uma ordem de pedido, ao invés de ter apenas o id do cliente que fez a compra, pode ter também seu nome e seu telefone principal, bem como o nome, quantidade e preço de cada produto adquirido, que são os dados que serão exibidos no sistema na tela de resumo do pedido.

Customer Collection

```
{
    _id: 123,
    name: "Katrina Pope",
    street: "123 Main St",
    city: "Somewhere",
    country: "Someplace",
    date_of_birth: ISODate("1992-11-03"),
    social_handles: [
        twitter: "@somethingamazing123"
    ]
    ...
}
```

Order Collection

```
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  date: ISODate("2019-02-18"),
  customer_id: 123,
  shipping_address: {
    name: "Katrina Pope",
    street: "123 Main St",
    city: "Somewhere",
    country: "Someplace"
  },
  order: [
    {
    product: "widget",
    qty: 5,
    cost: {
        value: NumberDecimal("11.99"),
        currency: "USD" }
    }
  ],
...
}
```

Na minha opinião, esse é um dos padrões que mais usei nos projetos que apliquei MongoDB como persistência e até conhecer esse nome eu chamava de Light Sub-documents, pois já entendia que usar versões menores (mais "light") de documentos maiores funcionava muito bem nesse modelo de banco.

A desvantagem mais óbvia deste padrão é a duplicação dos dados entre os diferentes documentos, então o ideal é não fazer uma referência extendida de campos que possam mudar com frequência.

Outlier

Este padrão, Caso Isolado, é útil quando sua coleção está modelada de um jeito bacana, mas alguns documentos dela fogem à regras gerais que você criou, o que pode pôr abaixo toda a modelagem da maioria, por causa de um problema de poucos documentos. Ex: um usuário de rede social que tem os seus amigos em um array, mas alguém muito popular, tem tantos amigos, que não cabe em um array do MongoDB. Um documento de produto que possui fotos junto dos dados do produto, mas que alguns possuem tantas fotos que não cabem no mesmo documento.

Para resolver este problema, mantém-se na coleção os documentos com a modelagem que atende à maioria. No documento que foge à regra de modelagem, cria-se um campo que sinaliza que este documento possui dados adicionais, em outra coleção, como abaixo, de um documento de

livro que contém tantos clientes que compraram ele, que precisa alocar os clientes adicionais em outra coleção (has_extras sinaliza isso).

```
{
   "_id": ObjectID("507f191e810c19729de860ea"),
   "title": "Harry Potter, the Next Chapter",
   "author": "J.K. Rowling",
   …,
  "customers_purchased": ["user00", "user01", "user02", …, "user999"],
  "has_extras": "true"
}
```

A aplicação então, quando vai carregar os dados deste documento, ao encontrar a flag de que possui campos adicionais, os procura na outra coleção, usando o _id do documento original como filtro.

É verdade que o schema em MongoDB é dinâmico e que você pode pensar que não faz sentido este padrão se simplesmente podem haver campos diferentes entre os documentos. No entanto, existem limites do próprio MongoDB, de itens em um array, de KB em um documento e por aí vai.

Assim, ao invés de particionar toda sua coleção por causa de poucos documentos, você mantém os dados juntos e somente nos documentos que é necessário o particionamento, você o faz.

A desvantagem desta padrão é que a aplicação tem de manter este controle, de carregar dados adicionais quando encontrar a flag, adicionando complexidade adicional. Além disso, esses Casos Isolados terão uma performance inferior do que o normal da aplicação.

Pre-allocation

Este padrão, Pré-alocação, é útil quando você sabe a estrutura exata que seu documento vai ter e a sua aplicação apenas vai preenchê-lo com dados. Ex: documento de sala de cinema que já poderia ter os subdocumentos dos assentos pré-alocados.

Este padrão era mais útil nas versões anteriores à 3.2 do MongoDB, quando alocar espaço de maneira adiantada dava ganhos de performance no antigo

sistema de arquivos MMAPv1. Com o advento do Wired Tiger e consequente descontinuação do MMAPv1, ele não é mais tão útil quanto antigamente.

A vantagem desta solução é a simplicidade da solução apenas, já que a performance não é mais significativa na alocação de espaço e como desvantagem temos o uso mais intenso de disco e memória por causa dos documentos maiores desde o início.

Sinceramente? Não vejo mais utilidade neste pattern, mas pode ser miopia da minha parte.

Polymorphic

Este padrão, Polimórfico, é útil quando você tem uma variedade de documentos muito similares entre si (mas não idênticos) e precisam ser mantidos na mesma coleção pois isso facilita as consultas a essas informações e tem mais performance do que fazer JOINs. Ex: coleção de atletas de diferentes esportes, base de veículos de diferentes tipos, classificados online de diversos produtos, etc.

Vamos pegar este exemplo de coleção de atletas de diferentes esportes. Em um banco relacional, teríamos uma tabela-mãe contendo os campos em comum entre todos atletas e depois as tabelas-filhas, especializadas para cada esporte, com uma chave estrangeira para a tabela-mãe.

No entanto, em MongoDB, usando o padrão polimórfico, teríamos documentos com uma base semelhante e características exclusivas, ao mesmo tempo, como abaixo.

```
{
    "sport": "ten_pin_bowling",
    "athlete_name": "Earl Anthony",
    "career_earnings": {value: NumberDecimal("1441061"), currency: "USD"},
    "300_games": 25,
    "career_titles": 43,
    "other_sports": "baseball"
}

{
    "sport": "tennis",
    "athlete_name": "Martina Navratilova",
    "career_earnings": {value: NumberDecimal("216226089"), currency: "USD"},
    "event": {
        "type": "singles",
        "career_tournaments": 390,
        "career_titles": 167
    }
}
```

Common fields

Assim, quando você desejar fazer consultas de todos atletas, consegue de maneira simples e performática. Se desejar fazer consultas de atletas de um esporte específico, considerando filtros específicos desse esporte, também consegue, de maneira simples e performática também.

Uma coisa interessante é que esse padrão aqui também funciona bem em sub-documentos, como abaixo, onde em alguns eventos são solo e outros em duplas, no tênis.

```
{
    "sport": "tennis",
    "athlete_name": "Martina Navratilova",
    "career_earnings": {value: NumberDecimal("216226089"), currency: "USD"},
    "career_tournaments": 390,
    "career_titles": 167,
    "event": [ {
        "type": "singles",
        "career_tournaments": 390,
        "career_titles": 167
    },
    {
        "type": "doubles",
        "career_tournaments": 233,
        "career_titles": 177,
        "partner": ["Tomanova", "Fernandez", "Morozova", "Evert", ...]
    },
...
}
```

Polymorphic Sub-Documents

Na sua aplicação, obviamente, será necessário ter essa resiliência para os dados que virão do banco, até mesmo usando o polimorfismo presente em algumas linguagens de programação como Java e C#. Talvez possamos considerar isso como a "desvantagem" desse padrão, pois exige lógica adicional na aplicação.

Este é, de longe, o padrão mais usado em MongoDB, o que eu chamava simplesmente de "VIEW" antes de conhecer o nome certo, pois me lembrava as VIEWS dos bancos relacionais que justamente faziam esse papel de agregar em uma tabela só os dados de diferentes tabelas. Curiosamente, o caso de uso mais comum deste padrão é em aplicações Single View.

Schema Versioning

Este padrão, Versionamento de Schema, é útil quando o seu schema mudou ao longo da existência da sua base de dados e documentos com diferentes versões de schema precisam coexistir na mesma coleção. Em uma base relacional, mudar o schema é um processo relativamente delicado pois

envolve uma análise profunda do schema atual vs novo, migração de dados e downtime da aplicação.

No MongoDB, é um processo igualmente importante, mas bem menos dolorido e sem downtime. Basicamente, se temos o documento abaixo, com uma pessoa e seus contatos telefônicos...

```
{
  "_id": "<ObjectId>",
  "name": "Anakin Skywalker",
  "home": "503-555-0000",
  "work": "503-555-0010"
}
```

...e mais tarde decidimos adicionar um campo para celular, é bem simples, certo? Basta adicionar mais um campo…

```
{
  "_id": "<ObjectId>",
  "name": "Darth Vader",
  "home": "503-555-0100",
  "work": "503-555-0110",
  "mobile": "503-555-0120"
}
```

Mas e se mais tarde entendemos que um telefone residencial (home) não é mais algo onipresente e ao mesmo tempo surgiram outras formas de comunicação ainda mais importantes, como as redes sociais.

Prevendo novas alterações, podemos repensar esta estrutura para um array de contatos, usando o padrão Attribute (que falei no _primeiro post da série), muito mais flexível e igualmente poderoso. Neste caso, temos um grande ponto de ruptura e aqui vale a pena versionar o schema, colocando um campo para isso:

```
{
  "_id": "<ObjectId>",
```

```
    "schema_version": "2",
    "name": "Anakin Skywalker (Retired)",
    "contact_method": [
        { "work": "503-555-0210" },
        { "mobile": "503-555-0220" },
        { "twitter": "@anakinskywalker" },
        { "skype": "AlwaysWithYou" }
    ]
}
```

Ambas versões de schema podem coexistir tranquilamente e caberá adaptar na sua aplicação para verificar esse campo para conseguir utilizar os documentos corretamente. Essa seria a primeira desvantagem deste padrão.

A segunda desvantagem é que com o passar do tempo, você pode ter de criar um script de migração de schema, em lote no MongoDB, se começar a ter tantas versões de schema que adicione muita complexidade na sua aplicação. Eu mesmo não recomendaria conviver com mais do que 2 schemas e ainda assim, recomendo que a sua própria aplicação, conforme os cadastros forem sendo atualizados, eles já serem salvos no schema mais recente.

Subset

Este padrão, Subconjunto, é útil quando a quantidade de memória RAM não está sendo o suficiente para o working set (os dados que são mais acessados, o Mongo mantém na memória) pois existem documentos grandes e, ao mesmo tempo, um subconjunto grande desses dados não são usados com frequência pela aplicação.

Como exemplo podemos citar documentos que possuem arrays de subdocumentos no seu interior. Muitas vezes não há a necessidade de se retornar todos estes elementos em uma consulta. Pense em um produto de um ecommerce com mil avaliações. Você vai listar todas elas na tela? Quando você faz uma consulta por um produto, necessariamente precisa-se retornar todas elas junto do documento?

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "Super Widget",
    description: "This is the most useful item in your toolbox."
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: "Kristina",
            review_text: "This is indeed an amazing widget.",
            published_date: ISODate("2019-02-18")
          },
          {
            review_id: 785,
            review_author: "Trina",
            review_text: "Very nice product, slow shipping.",
            published_date: ISODate("2019-02-17")
          },
   ...
          {
            review_id: 1,
            review_author: "Hans",
            review_text: "Meh, it's okay.",
            published_date: ISODate("2017-12-06")
          }
    ]
}
```

Muito provavelmente que somente as 10 avaliações mais recentes façam sentido de serem exibidas logo de cara. Assim, ao invés de armazenar TODAS as reviews do produto junto do mesmo, podemos armazenar somente um subconjunto, como as 10 mais recentes, e as demais ficam em uma coleção específica de avaliações, como abaixo.

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: “Super Widget”,
    description: “This is the most useful item in your toolbox.”
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
         {
           review_id: 786,
           review_author: “Kristina”,
           stars: 5
           review_text: “This is indeed an amazing widget.”,
           published_date: ISODate(”2019-02-18”)
         },
...
         {
           review_id: 776,
           review_author: “Pablo”,
           stars: 5
           review_text: “Wow! Amazing.”,
           published_date: ISODate(”2019-02-16”)
         }
    ]
}
```

Product Collection

```
{
      review_id: 786,
      product_id: ObjectId(”507f1f77bcf86cd799439011”),
      review_author: “Kristina”,
      review_text: “This is indeed an amazing widget.”,
      published_date: ISODate(”2019-02-18”)
}

{
      review_id: 785,
      product_id: ObjectId(”507f1f77bcf86cd799439011”),
      review_author: “Trina”,
      review_text: “Very nice product, slow shipping.”,
      published_date: ISODate(”2019-02-17”)
}

{
      review_id: 1,
      product_id: ObjectId(”507f1f77bcf86cd799439011”),
      review_author: “Hans”,
      review_text: “Meh, it’s okay.”,
      published_date: ISODate(”2017-12-06”)
}
```

Review Collection

Este é apenas um exemplo, o mindset é de que no caso de documentos grandes que estejam onerando o working set, somente os dados mais acessados deveriam estar no documento principal e os demais em subdocumentos.

A desvantagem aqui é a gestão deste subconjunto. Afinal, neste exemplo, se você quer ter sempre as 10 reviews mais recentes do produto, toda vez que uma nova review for postada, você terá de adicioná-la no subconjunto e na coleção de reviews, além de remover a mais antiga do subconjunto, certo? E esta é uma regra simples, então tome cuidado com a gestão dos seus subconjuntos!

Tree

Este padrão, Árvore, é útil quando você precisa armazenar dados hierárquicos no seu documento. Ex: gestores e subordinados de um funcionário, categorias e subcategorias de um produto, etc.

Enquanto que em um banco relacional isso geralmente é resolvido com diversos registros com chaves primárias e estrangeiras pra todo lado, em MongoDB a resolução é muito mais direta, embora envolva duplicação de dados.

Pegando o exemplo de categorias e subcategorias de um produto, o padrão Árvore define que o seu documento de produto deve ter um campo array para as categorias-pai (ancestor_categories, em ordem), além de um campo para a categoria imediatamente superior (parent_category), como abaixo.

```
{
 _id: <ObjectId>,
 name: "Samsung 860 EVO 1 TB",
 part_no: "MZ-76E1T0B",
 price: {
   value: NumberDecimal("169.99"),
   currency: "USD"
 },
 parent_category: "Solid State Drives",
 ancestor_categories: [
   "Solid State Drives",
```

```
    "Hard Drives",
    "Storage",
    "Computers",
    "Electronics"
  ]
}
```

Note que aqui usei apenas os nomes das categorias, mas que poderíamos ter subdocumentos com chave-valor, caso seja necessário. Note também, que ter um campo parent_category diretamente na raiz do documento permite usar os recursos de atravessamento em grafo do MongoDB.

No caso do use case de funcionários, bastaria ter um outro campo array para os subordinados dele, ao invés de apenas os seus gestores.

A vantagem deste padrão é a velocidade de carregar a árvore hierárquica deste documento e como desvantagem temos a duplicação de dados e a complexidade de atualização dos mesmos. Logo, é importante que ele seja aplicado em dados que mudem com muita frequência, embora a hierarquia em si possa mudar bastante, basicamente o mesmo risco do padrão Extended Reference, já citado anteriormente.

--

E com isso encerro este capítulo sobre modelagem de dados em MongoDB. Espero que seja de utilidade prática para as suas aplicações e que tenha conseguido lhe mostrar um pouco mais da versatilidade e poder da orientação à documentos deste fantástico banco de dados.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Modelagem de Dados
Neste vídeo eu falo da principal e mais básica forma de modelar dados no MongoDB.
https://www.youtube.com/watch?v=bCcWoNIVMlQ

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

5 Criando aplicações com MongoDB

Not all roots are buried down in the ground some are at the top of the tree.
— Jinvirle

O foco deste livro não é programação. Mas se você é um programador, passar os olhos por esses rápidos tutoriais de algumas linguagens famosas de programação lhe dará uma ideia de como criar suas aplicações usando MongoDB como persistência de dados.

Nenhum dos tópicos foca na linguagem de programação em si, parto do pressuposto que você lerá o tópico de uma linguagem que conheça bem ou ao menos saiba como criar a estrutura básica do projeto.

Node.js: Web API com MongoDB

Vamos começar criando um novo projeto Express em uma pasta webapi-node para guardarmos nossos fontes dentro. Dentro dessa pasta, crie um arquivo app.js vazio, onde codificaremos o coração da nossa web API mais tarde.

Agora acesse o terminal e navegue até essa pasta, usando o comando abaixo para criar o nosso arquivo package.json:

Código 5.1: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
npm init
```

O NPM init é um recurso interativo para criação do package.json. Ele vai lhe fazer diversas perguntas que você pode apenas apertar Enter para ficar a resposta default (indicada entre parênteses) ou escrever suas respostas personalizadas.

Uma das perguntas é o entry-point, ou arquivo de "start" da sua aplicação (seja ela uma web API ou não). Se você criou o app.js conforme

mencionou, o npm init irá lhe sugerir este arquivo como sendo o entry-point.

O próximo passo é instalar algumas dependências que vamos precisar, usando o npm install:

Código 5.2: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
npm i express mongodb
```

Aqui já informei o nome de mais de um pacote de uma vez só, o que é perfeitamente possível com o NPM.

Com essas dependências instaladas, vamos começar a programar!

Atenção: usaremos o mesmo banco de dados MongoDB que criamos há alguns capítulos atrás (banco workshop, com coleção customers,lá no capítulo 2). Então não se esqueça de garantir que ele esteja executando corretamente com o mongod.

Usaremos apenas o arquivo app.js neste exemplo de web API, então não ficarei repetindo isso o tempo todo. Comece adicionando as seguintes linhas bem no topo dele, explicarei na sequência.

Código 5.3: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
const { MongoClient } = require ( " mongodb " ) ;
async function connect () {
  if ( global . db ) return global . db ;
  const client = new MongoClient ( " mongodb://127.0.0.1:27017/ " ) ;
  await client . connect () ;

  global . db = await conn . db ( " workshop " ) ;
  return global . db ;
}
```

Na primeira linha, carregamos o módulo mongodb usando o comando require e de tudo que este módulo exporta vamos usar apenas o objeto MongoClient, que armazenamos em uma variável de mesmo nome. Usaremos esta variável na função connect que precisará ser criada.

Antes de qualquer coisa, eu verifico se existe uma variável global chamada db. Se existir, isso quer dizer que já temos uma conexão estabelecida e retornamos ela. Fim.

Existem duas palavras reservadas aí que podem lhe causar alguma confusão e elas são async e await. Async diz que nossa função é assíncrona e é um pré-requisito para conseguirmos usar await na sequência. O Await bloqueia a execução daquela linha até que termine o processamento. A conexão com o banco de dados pode demorar um pouco dependendo de onde ele esteja hospedado e sabemos que o Node.js não permite bloqueio da thread principal por padrão, por isso usamos o await para dizer a ele esperar pelo banco, pois não podemos avançar sem essa conexão.

Essa conexão retorna um objeto através do qual nós podemos selecionar o banco de dados que queremos, e vamos armazenar ele em uma variável global, para que não tenhamos de ficar abrindo múltiplas conexões sem necessidade no banco de dados.

```
global . db = await conn . db ( " workshop " ) ;
return global . db ;
```

Esse processo será necessário em vários pontos da nossa aplicação, por isso que resolvi criar uma função encapsulando tudo, para aumentar o reuso de código e facilitar possíveis manutenções futuras.

Vamos seguir nosso app.js definindo algumas variáveis locais através do carregamento de alguns módulos:

Código 5.4: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
const express = require ( ' express ' ) ;
const app = express () ;
const port = 3000 ; //porta padrão
```

O módulo express é carregado para criar o objeto da nossa aplicação (app). Antes de escrever o código que inicia o servidor web temos de configurar algumas coisas no app.js.

Primeiro, vamos configurar o app para usar serialização JSON para os dados recebidos nas requisições HTTP:

Código 5.5: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
app . use ( express . json ()) ;
```

Agora, vamos configurar como irá funcionar o roteamento das requisições:

Código 5.6: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
//definindo as rotas
const router = express . Router ();
router . get ( ' / ' , ( req , res ) => res . json ( { message : ' Funcionando! ' }
));
app . use ( ' / ' , router );
```

Aqui criei uma rota GET default que apenas retorna um JSON avisando que a API está funcionando. Web APIs não possuem views, elas retornam no corpo das requisições JSON ou XML, ao invés de HTML. Sendo assim, note como na função de callback do router.get (que eu usei uma arrow function) eu usei res.json ao invés de res.render, pois o retorno será um objeto JSON.

Preste atenção ao código do router.get, é aqui que vamos definir as nossas outras rotas mais tarde.

E por fim, vamos escrever o código que inicializa o nosso servidor no final do app.js:

Código 5.7: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
//inicia o servidor
app . listen ( port )
console . log ( ' API funcionando! ' )
```

Com isso, já podemos iniciar a nossa web API e ver se ela está funcionando. Curioso como em Node.js conseguimos criar coisas incríveis como um webserver HTTP com tão poucas linhas de código, não é mesmo?

Código 5.8: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
node app
```

Aqui usei o comando node ao invés de iniciar o projeto com o npm start, indicando qual o arquivo que deve ser executado para iniciar a aplicação/servidor.

O resultado no navegador, acessando localhost:3000 deve ser:

Como não temos views, todos os testes de nossa API terão como resultado objetos JSON, então vá se acostumando.

Programando a API

Agora que temos o projeto configurado e com a estrutura básica pronta, podemos programar e testar nossa API facilmente.

Para cada operação desejamos oferecer através da nossa API devemos criar uma rota em nosso app.js, a começar por outro router.get, logo abaixo do anterior.

Assim, vamos começar com uma operação muito simples: listar todos os clientes do banco de dados. Para isso, modifique o conteúdo do router.get para que chame a função connect e use essa conexão para ir no banco de dados buscar os clientes, como abaixo:

Código 5.9: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
router . get ( ' /clientes' , async ( req , res , next ) => {
  try {
    const db = await connect () ;
    res . json ( await db . collection ( " customers " ). find (). toArray ()) ;
  }
  catch ( ex ) {
    console . log ( ex ) ;
    res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
  }
} )
```

Aqui nós chamamos o connect com um await, para aguardar o término da sua execução. Na sequência usamos a função collection para selecionar a coleção de customers, seguido da função find para nos trazer tudo e por úlltimo a toArray para converter o resultado para um array de clientes.

Como essa operação também deve demorar alguns milisegundos (dependendo do volume de clientes cadastrados), usamos o await aqui

também para segurar o Node.js.

Note que em caso de erro (falha de conexão com o banco, por exemplo) nossa API vai retornar um código 400 com um JSON informando a causa do erro.

Nota: você não precisa definir o status das respostas quando elas forem um 200 OK, pois esse é o valor padrão de status.

O resultado da consulta vai ser devolvido em formato JSON usando res.json. Salve o arquivo e reinicie o servidor Node.js. Ainda se lembra de como fazer isso? Abra o prompt de comando, derrube o processo atual (se houver) com Ctrl+C e depois npm start novamente.

Agora abra seu navegador, acesse http://localhost:3000/clientes e maravilhe-se com o resultado. Como o acesso do navegador é sempre GET, você deve ver um array JSON com todos os clientes do seu banco de dados nele:

localhost:3000/clientes

[{"_id":"59ab419d33959e2724be2cbb","nome":"Luiz","idade":29,"uf":"RS"},{"_id":"59ab46e433959e2724be2cbc","nome":"Fernando","idade":29},
{"_id":"59ab46e433959e2724be2cbd","nome":"Teste","uf":"RS","idade":28}]

Mas e se quisermos oferecer um recurso que permita ver apenas um cliente específico?

Podemos modificar a nossa rota GET para que ela receba por parâmetro um id opcionalmente. Neste caso, ela deverá retornar somente um cliente ao invés de todos.

Antes de mexer na rota, precisaremos de mais um objeto, então volte ao início do arquivo customer.js para editar a nossa chamada ao módulo mongodb.

Código 5.10: disponível em http://www.luiztools.com.br/livro-mongo-fontes

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

Aqui precisei carregar o objeto ObjectId do módulo mongodb para poder converter o id, que virá como string na requisição, para o tipo correto que o MongoDB entende como sendo a chave primária das suas coleções. Você verá isso na prática no próximo trecho de código.

Com este código pronto, agora podemos fazer alguns ajustes na router.get:

Código 5.11: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
router . get ( ' /clientes/:id? ' , async ( req , res , next )=> {
  try {
    const db = await connect () ;
    if ( req . params . id )
      res . json ( await db . collection ( " customers " ). findOne ( { _id : new ObjectId ( req . params . id ) } )) ;
    else
      res . json ( await db . collection ( " customers " ). find (). toArray ()) ;
  }
  catch ( ex ) {
    console . log ( ex ) ;
    res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
  }
 } )
```

Existem poucas diferenças em relação à outra rota, como o parâmetro :id no caminho da URL e a chamada à função findOne quando veio o id. Note que usei uma interrogação logo após o :id, isso quer dizer que ele é opcional. Para acessar essa informação que veio na URL, basta usar req.params.<nome_do_parametro>.

Para testar, pegue um dos _ids dos clientes listados no teste anterior e experimente acessar uma URL localhost:3000/clientes/id trocando 'id' pelo respectivo id que deseja pesquisar. O retorno no navegador deve ser um único objeto JSON.

Caso o cliente não exista, deve aparecer null (o cliente não existir não é um erro). Se quiser você inclusive pode tratar isso facilmente com um if.

Poderíamos ficar aqui criando vários endpoints de retorno de clientes diferentes, usando os filtros em nossos finds, mas acho que você já pegou a ideia.

Listar dados é moleza e salvar dados no MongoDB não é algo particularmente difícil.

Para isso, vamos criar uma nova rota no app.js com router.post, uma vez que POST é o verbo HTTP usado para cadastrar coisas:

Código 5.12: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
router . post ( ' /clientes ' , async ( req , res , next ) => {
  try {
    const customer = req . body ;
    const db = await connect () ;
    res . json ( await db . collection ( " customers " ). insertOne ( customer ))
;
  }
  catch ( ex ) {
    console . log ( ex ) ;
    res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
  }
} )
```

Note que este código já começa diferente dos demais, com router.post ao invés de router.get, indicando que está rota tratará POST no endpoint /clientes. Na sequência, pegamos o body da requisição onde está o JSON do cliente a ser salvo na base de dados. Obviamente no mundo real você irá querer colocar validações, tratamento de erros e tudo mais.

Note que aqui estou usando JSON como padrão para requisições e respostas. Caso seja enviado dados em outro formato, causará erros de

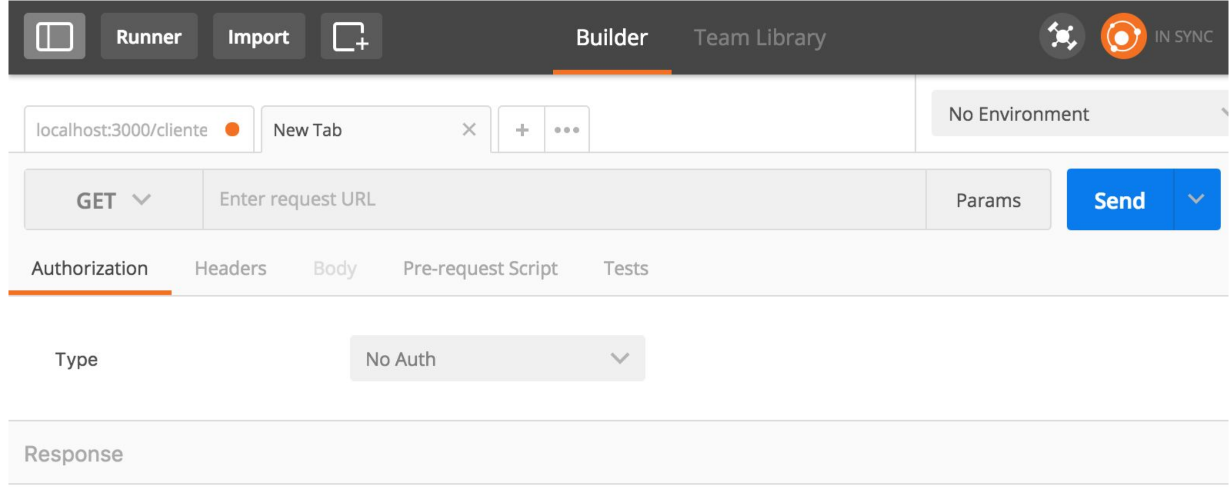
comunicação com a API.

Mas e agora, como vamos testar um POST? Pela barra de endereço do navegador não dá, pois ela só faz GET.

Temos diversas alternativas no mercado, sendo que vou falar de uma aqui: usando o POSTMAN. Caso não tenha baixado e instalado na sua máquina ainda, faça-o usando o link abaixo:

https://www.getpostman.com/

Basicamente o POSTMAN é uma ferramenta que lhe ajuda a testar APIs visualmente. Você verá a tela abaixo ao abrir o POSTMAN:

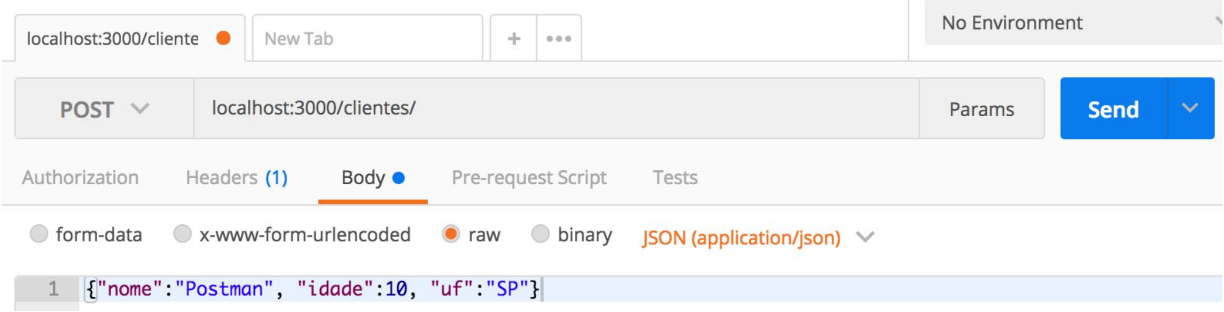


No primeiro select à esquerda você encontra os verbos HTTP, selecione POST nele.

Na barra de endereço, digite o endpoint no qual vamos fazer o POST, em nosso caso http://localhost:3000/clientes .

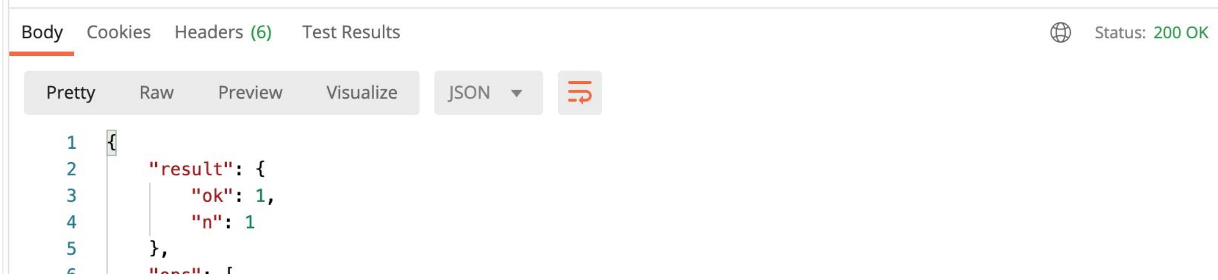
Logo abaixo temos algumas abas. Vá na aba Headers (cabeçalhos) e adicione uma linha com a seguinte configuração: key = Content-Type e value = application/json. Isso diz ao POSTMAN que essa requisição está enviando dados no formato JSON em seu corpo (body).

E por fim, vá na aba Body, marque a opção 'raw' e escreva na área de texto abaixo um objeto JSON de cliente (não esqueça de colocar aspas entre os nomes das propriedades aqui), como os inúmeros que já escrevemos antes.



Agora sim podemos testar!

Considerando que você já esteja com seu servidor atualizado e rodando, clique no enorme botão Send do POSTMAN e sua requisição será enviada. Quando isso acontece, o POSTMAN exibe em uma área logo abaixo da requisição, a resposta (response) da requisição, como abaixo:



Note que é mostrado o corpo da resposta (nesse caso o JSON de sucesso) e o status da mesma (nesse caso um 200 OK).

Mas será que funcionou mesmo?

Basta digitarmos localhost:3000/clientes em nosso navegador (ou construir uma requisição GET no POSTMAN) e veremos que nosso novo customer está lá!

Mas e se eu quiser atualizar um cliente?

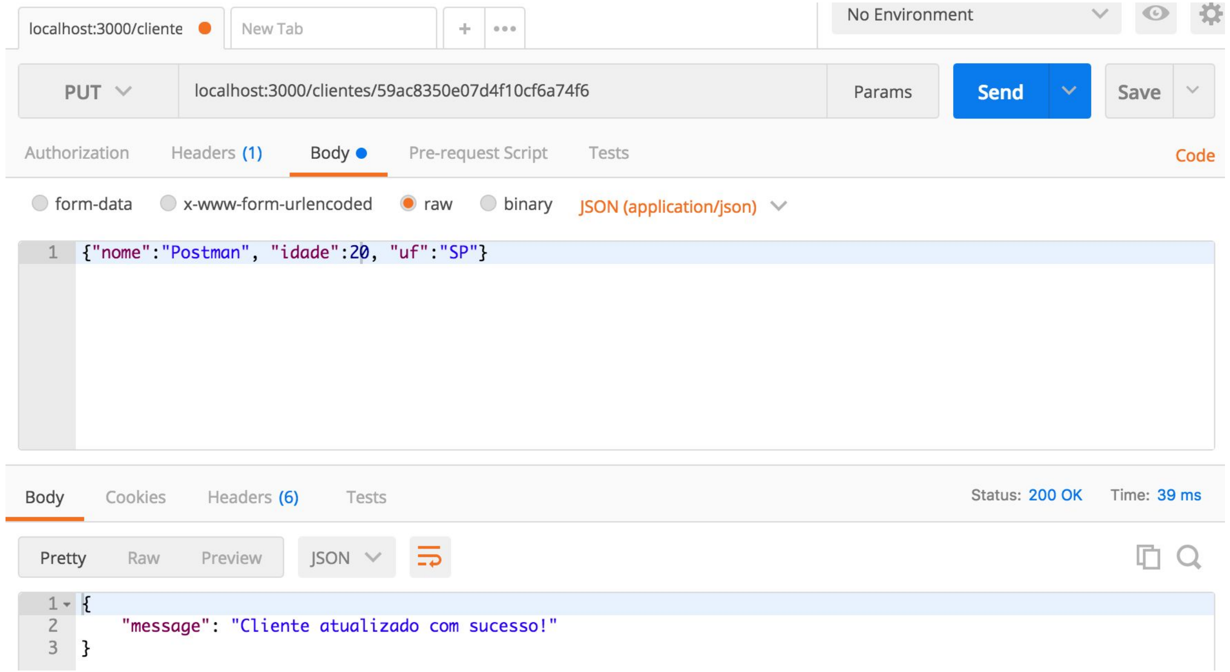
Se for uma atualização de um cliente inteiro, neste caso deve ser usado um PUT. Para fazer isso, adivinha, basta criar uma nova rota no app.js:

Código 5.13: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
router . put ( ' /clientes/:id ' , async ( req , res , next ) => {
  try {
    const customer = req . body ;
    const db = await connect () ;
    res . json ( await db . collection ( " customers " ). replaceOne ( { _id :
new ObjectId ( req . params . id ) } , customer )) ;
  }
  catch ( ex ) {
    console . log ( ex ) ;
    res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
  }
} )
```

Essa nova rota é praticamente idêntica à do router.post, com pequenas variações. É muito importante lembrar que o replaceOne substitui o customer com o id passado por parâmetro pelo objeto JSON passado pelo body da requisição. Então cuidado!

Para testar, configure uma requisição PUT no POSTMAN como abaixo, não esquecendo que o id da URL você deve pegar de algum cliente da sua base, pois eles não serão iguais. Além disso, a aba Headers tem o Content-Type definido como application/json e o verbo é PUT:



Incluí no mesmo print acima a resposta à minha requisição.

Update: check!

Para fazer um update parcial é um pouquinho mais complicado, mas não muito, pois você terá de usar o operador $set do MongoDB e usar o verbo PATCH.

Note que como segundo parâmetro, ao invés de passar o objeto diretamente para função update eu usei o operador $set e passarei o objeto pra ele. Isso fará com que o objeto updates possa conter somente os campos que eu desejo alterar nesse cliente, deixando os demais intactos. Muito mais prático e seguro do ponto de vista da integridade dos dados.

Depois faça a rota no app.js:

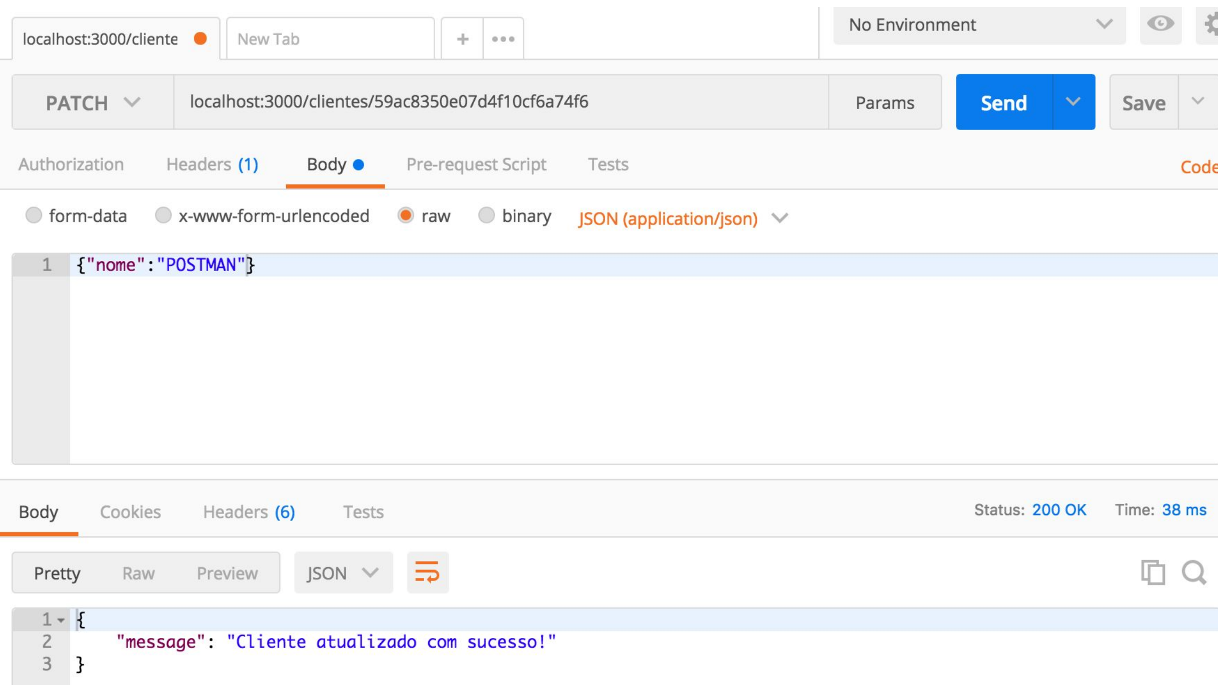
Código 5.14: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
router . patch ( ' /clientes/:id ' , async ( req , res , next ) => {
  try {
    const customer = req . body ;
```

```
    const db = await connect () ;
    const id = { _id : new ObjectId ( req . params . id ) };
    res . json ( await db . collection ( " customers " ). updateOne ( id , { $set
: customer } )) ;
  }
  catch ( ex ) {
    console . log ( ex ) ;
    res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
  }
} )
```

Para testar no POSTMAN é muito simples, troque o verbo para PATCH (considerando que você ainda tem o exemplo de PUT aberto) e no body da requisição, coloque um objeto JSON contendo apenas os dados que você deseja alterar do respectivo cliente (o que possui o id informado na URL).



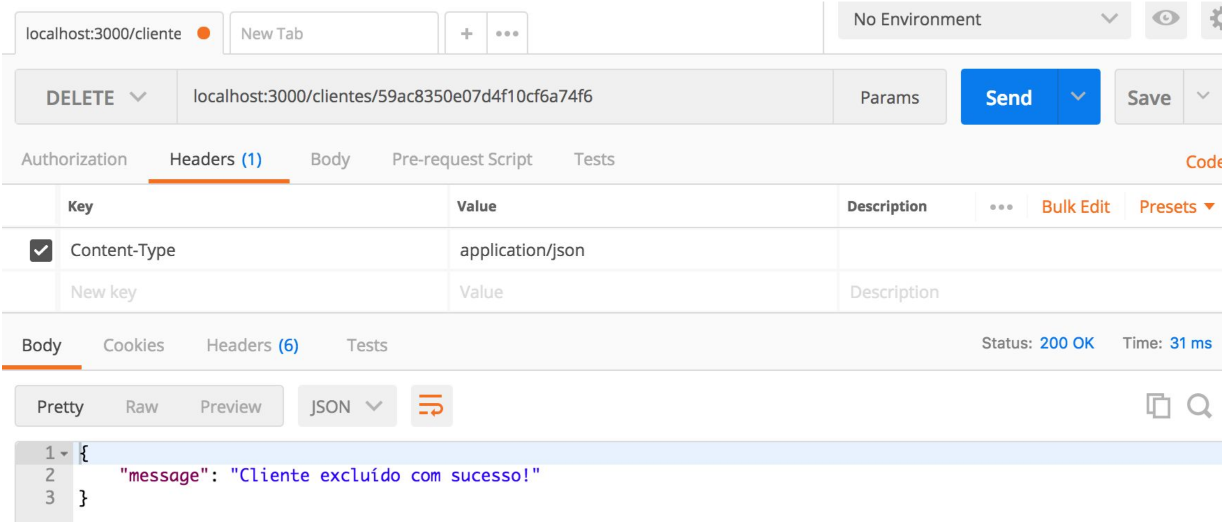
No exemplo acima, apenas troquei o nome do customer de Postman para POSTMAN.

Para encerrar a construção da nossa API REST, falta nosso delete que é muito simples e rápido de fazer:

Código 5.15: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
router . delete ( ' /clientes/:id ' , async ( req , res , next ) => {
   try {
     const db = await connect () ;
     res . json ( await db . collection ( " customers " ). deleteOne ( { _id :
new ObjectId ( req . params . id ) } )) ;
   }
   catch ( ex ) {
     console . log ( ex ) ;
     res . status ( 400 ). json ( { erro : `${ ex }` } ) ;
   }
} )
```

Para testar vamos recorrer novamente ao POSTMAN que apenas precisa ter o verbo definido como DELETE e a URL informando o id do cliente a ser excluído, nada além disso.



E com isso encerramos a criação de nossa web API REST usando Node.js, Express e MongoDB.

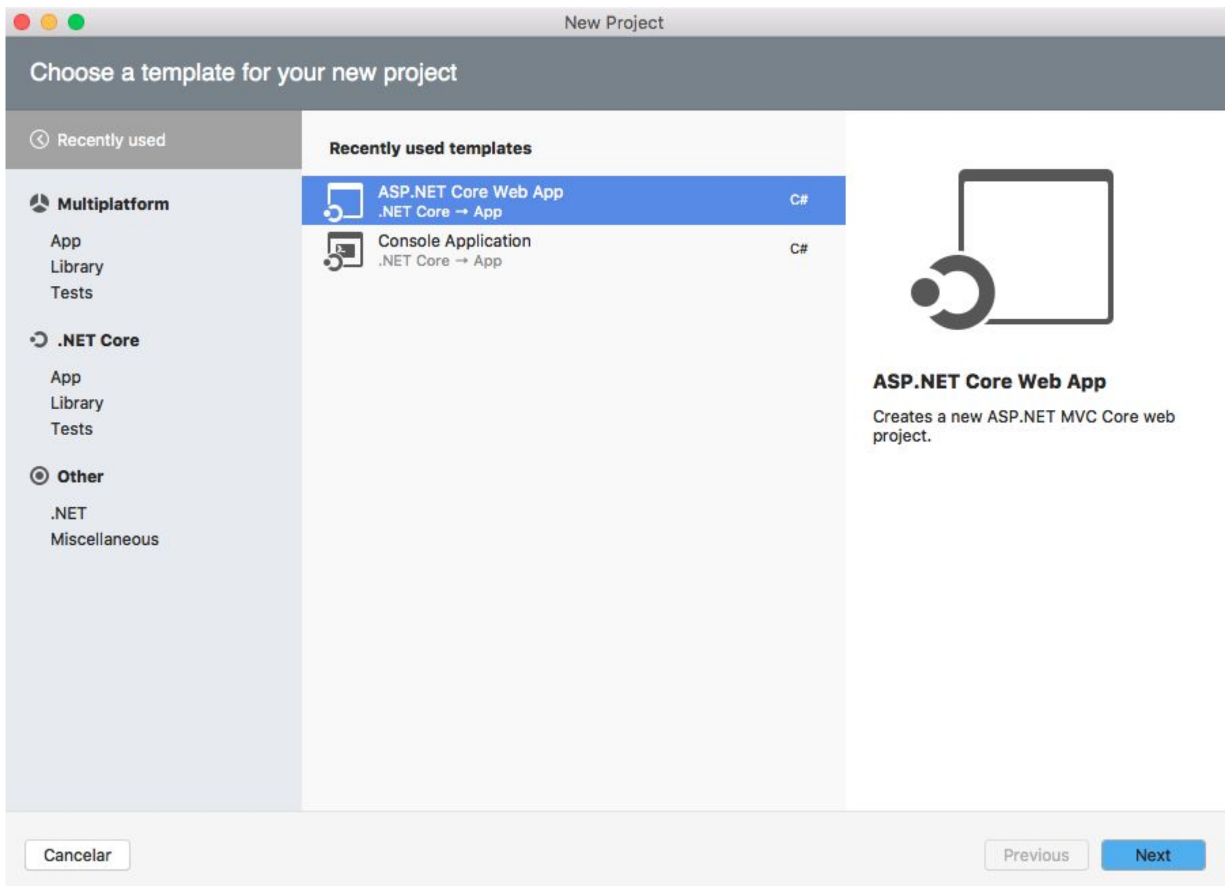
Querendo saber mais sobre desenvolvimento web com Node.js, recomendo o meu livro Programação Web com Node.js , à venda na Amazon.

ASP.NET Core: Mecanismo de busca com MongoDB

Baixe e instale o .NET Core na sua máquina.

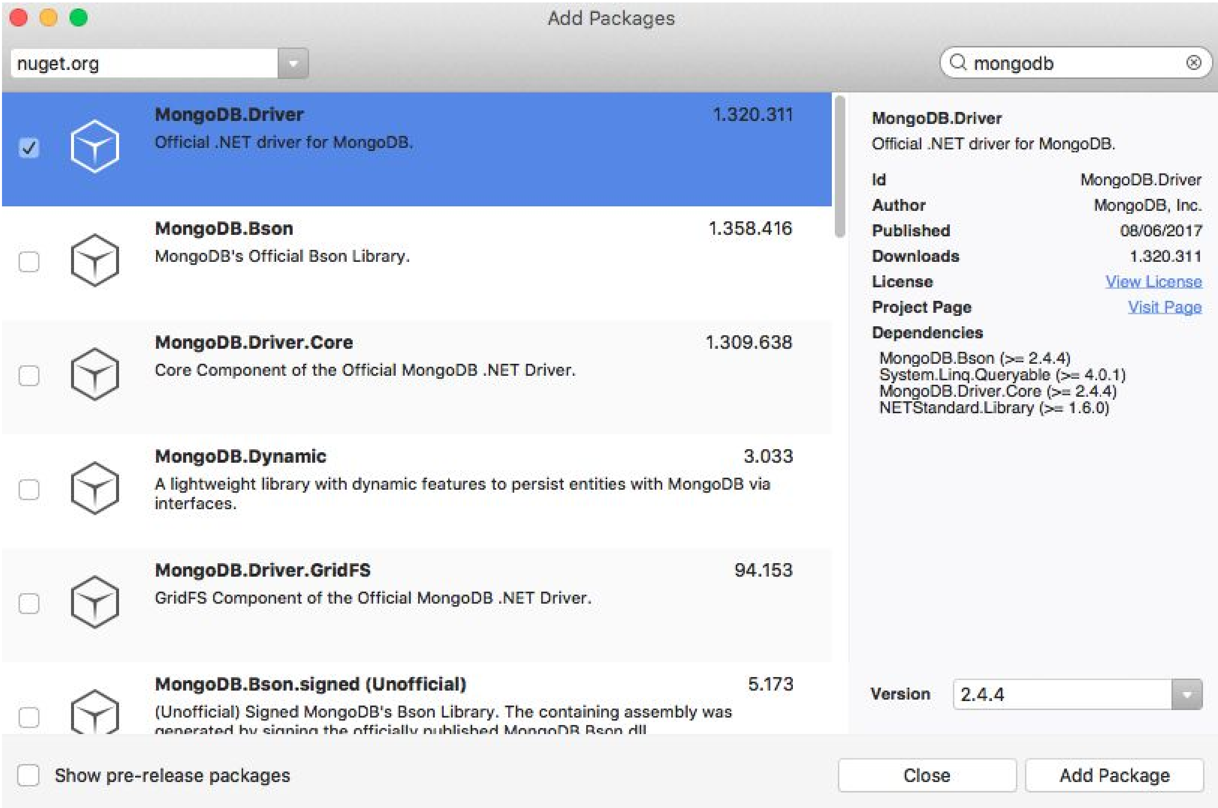
Agora, baixe e instale o Visual Studio Community na sua máquina, é gratuito e está disponível para Mac e Windows. Se estiver no Linux, você pode usar linha de comando ou o Visual Studio Code, mas o procedimento será um pouco diferente do mostrado aqui.

Com o Visual Studio aberto, crie um novo projeto ASP.NET Core Web App.



Assim que a estrutura do projeto é criada, o VS começa a restaurar os pacotes padrões do projeto e isso pode demorar alguns instantes.

Agora clique com botão direito na pasta Dependencies/NuGet do projeto e dê um "Add Packages" para abrir o gerenciador de dependências. Busque nele por "mongodb" e instale a extensão oficial:



Se você mandar executar o projeto irá ver o site MVC de exemplo do ASP.NET Core. Os demais arquivos vamos mexer na sequência.

Preparando o banco

Quando o assunto é mecanismo de busca, você pode ter uma base SQL com os dados consolidados do seu negócio e usar o MongoDB apenas como índice e/ou cache de busca. Ou então você pode usar apenas o MongoDB como fonte de dados. Fica à seu critério.

Caso escolha usar SQL e MongoDB, você terá de ter algum mecanismo para mandar os dados que deseja que sejam indexados pelo seu buscador

para o Mongo. Este livro não cobre migração de dados (mongoimport é o cara aqui, falaremos dele no capítulo 6), então você deve fazer por sua conta e risco usando os meios que conhecer.

Caso escolha apenas usar o Mongo, você apenas terá de alterar as suas coleções pesquisáveis para incluir um campo com o índice invertido que vamos criar na sequência, com nosso buscador de exemplo.

Em ambos os casos, a sua informação "pesquisável" deve ser armazenada de uma maneira prática de ser pesquisada, o que neste exemplo simples chamaremos de tags. Cada palavra dentro das informações pesquisáveis do seu sistema deve ser transformada em uma tag, que geralmente é um texto todo em maiúsculo (ou minúsculo) e sem acentos ou caracteres especiais.

Por exemplo, se quero tornar pesquisável os nomes dos meus clientes, que no meu SQL estão como "Luiz Júnior", eu devo normalizá-lo para as tags "LUIZ" e "JUNIOR", separadas. Assim, quando pesquisarem por luiz, por junior, or luiz junior e por junior luiz, este cliente será encontrado.

Assim, cada registro na sua coleção do MongoDB terá um atributo contendo as suas tags, ou informações pesquisáveis, o que facilmente fazemos com um atributo do tipo array no Mongo. Como abaixo:

Código 5.25: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{
  " _id " : ObjectId ( " 123-abc-456-def " ),
  " Nome " : " Luiz Fernando Duarte Júnior " ,
  " Tags " : [ " LUIZ " , " FERNANDO " , " DUARTE " , " JUNIOR " ],
  ...
}
```

Para podermos fazer a busca depois usaremos uma query com um **$in** ou um **$all** , que são os operadores do Mongo para pesquisar arrays de palavras (seus termos de busca) dentro de arrays de palavras (as tags).

Então, caso esteja migrando dados de um SQL para o Mongo, certifique-se de quebrar e normalizar as informações que deseja pesquisar dentro de um campo tags, como o acima, que será o nosso índice de pesquisa.

Para fins de exemplo, usaremos a massa de dados abaixo (apenas 2 registros) para pré-popular nosso banco com clientes (customers) que já possuem tags normalizadas como mencionado acima. Note que as tags de cada customer são um misto de seus nomes e profissões, o que você pode facilmente fazer com seus dados também.

Código 5.26: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
custArray = [ { " Nome " : " Luiz Júnior " , " Profissao " : " Professor " , " Tags " : [ " LUIZ " , " JUNIOR " , " PROFESSOR " ] } , { " Nome " : " Luiz Duarte " , " Profissao " : " Blogueiro " , " tags " : [ " LUIZ " , " DUARTE " , " BLOGUEIRO " ] } ]
db . Customer . insert ( custArray ) ;
```

O comando acima deve ser executado no console cliente do Mongo, logo após o "use searchengine".

Obviamente existem técnicas de modelagem de banco para mecanismos de busca muito mais elaboradas que essa. Aqui estamos tratando todas as informações textualmente sem classificação do que é cada uma, sem se importar com a ordem ou peso delas, etc. Mas a partir daqui você pode fazer as suas próprias pesquisas para melhorar nosso algoritmo.

Mais pra frente, quando fizermos as nossas pesquisas, vamos fazê-las sempre buscando no campo tags, ao invés de ir nos atributos do documento. Até porque nosso buscador terá apenas um campo de busca, assim como o Google, como veremos adiante.

Mas e a performance disso?

Para resolver este problema vamos criar um índice nesse campo no MongoDB. Mas não é qualquer índice, mas sim um índice multi-valorado

pois o campo tags é um array de elementos.

O Mongo organiza campos multi-valorados em índices invertidos, que são exatamente um dos melhores tipos de índices básicos que podemos querer em um mecanismo de busca simples como o nosso. Eu já mencionei sobre índices invertidos em um famoso artigo do meu blog (LuizTools) chamado Como criar um mecanismo de busca.

Código 5.27: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . Customer . createIndex ( { " Tags " : 1 } ) ;
```

O comando acima deve ser executado no console cliente do Mongo, logo após o "use searchengine". Todos os customers inseridos a partir de então respeitarão essa regra do índice no campo tags.

Para verificar se funcionou o nosso índice, teste no console cliente do Mongo consultas como essa que traz todos os clientes que possuam a tag LUIZ (isso funciona para lógica OR também, pois recebe um array de possibilidades):

Código 5.28: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . Customer . find ( { " Tags " : { $in : [ " LUIZ " ] }} ). pretty ()
```

Ou esse que traz todos com as tags LUIZ e JUNIOR (aqui temos lógica AND):

Código 5.29: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db . Customer . find ( { " Tags " : { $all : [ " LUIZ " , " JUNIOR " ] }} ).
pretty ()
```

Voltando ao Projeto

Adicione uma pasta Models no seu projeto se ela ainda não existir.

Dentro dela, adicione uma classe Customer.cs com o seguinte conteúdo dentro:

Código 5.30: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
using System ;
using System . Collections . Generic ;
using MongoDB . Bson ;
using MongoDB . Bson . Serialization . Attributes ;

public class Customer
{
   [ BsonId ]
   public ObjectId Id { get ; set ; }

   public string Nome { get ; set ; }

   public string Profissao { get ; set ; }

   public List < string > Tags { get ; set ; }
}
```

Essa classe é um espelho de um documento Customer que será armazenado na coleção homônima do MongoDB. Existem diversos attributes que podemos colocar sobre as propriedades desta classe para ajudar a mapear o MongoDB corretamente, sendo que aqui usei apenas o [BsonId] que diz que aquela propriedade é o "_id" do documento, campo obrigatório de existir. Outros attributes possíveis seriam o BsonElement para dizer o nome daquela propriedade na coleção (aqui estamos usando o mesmo nome em ambos), BsonRequired para dizer que uma propriedade é obrigatória e muito mais.

Para fazer a conexão e manipulação do banco, vamos criar uma classe que vai funcionar de maneira semelhante a um DAO ou Repository Pattern.

Adicione a classe DataAccess.cs na pasta Models do seu projeto, inicialmente com o seguinte conteúdo:

Código 5.31: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
public class DataAccess
{
    MongoClient _client ;
    IMongoDatabase _db ;

    public DataAccess ()
    {
        _client = new MongoClient ( " mongodb://127.0.0.1:27017 " ) ;
        _db = _client . GetDatabase ( " searchengine " ) ;
    }

    public long CountCustomers () {
        return _db . GetCollection < Customer >( typeof ( Customer ). Name
). Count ( new FilterDefinitionBuilder < Customer >(). Empty ) ;
    }
}
```

Aqui temos um construtor que faz a conexão com o servidor do Mongo. Nesta conexão, você deverá informar os seus dados de conexão, que caso seja servidor local, deve funcionar do jeito que coloquei no exemplo. Caso seja um banco remoto, você deverá ter uma connection string parecida com essa:

mongodb://usuario:senha@servidor:porta

Já o outro método da DataAccess.cs, CountCustomers() é um método que vai na coleção Customer do MongoDB e contabiliza quantos documentos estão salvos lá (passei um filtro vazio por parâmetro), retornando este número. Usaremos este método mais tarde, para testar se nossa conexão está funcionando.

Pronto, a configuração mínima da model já está ok!

Agora vamos criar a view que vamos utilizar como pesquisa e listagem de resultados (chamada de SERP pelos especialistas: Search Engine Results Page). Vamos fazer as duas em uma, por pura preguiça deste que vos escreve. ;)

Dentro da pasta Views/Shared do seu projeto, abra _Layout.cshtml e adicione um novo item no menu superior apontando para uma página /Search, deixando-o assim (mudou apenas o segundo link):

Código 5.32: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
< ul class = "nav navbar-nav" >
  < li >< a asp-area = "" asp-controller = "Home" asp- action = "Index" >
Home </ a ></ li >
  < li >< a asp-area = "" asp-controller = "Home" asp- action = "Search" >
Search </ a ></ li >
  < li >< a asp-area = "" asp-controller = "Home" asp- action = "About" >
About </ a ></ li >
  < li >< a asp-area = "" asp-controller = "Home" asp- action = "Contact" >
Contact </ a ></ li >
</ ul >
```

Quando este link for clicado (e você pode testar isso mandando executar o projeto), ele enviará o usuário para o endereço /Home/Search, indicando que na sua pasta Controllers deve existir um HomeController.cs com um método Search dentro. Não temos ainda este método, então vamos criá-lo no referido arquivo:

Código 5.33: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
// HomeController.cs
public IActionResult Search ()
{
  ViewData [ " Message " ] = " Search page. " ;
```

```
  ViewData [ " Count " ] = new DataAccess (). CountCustomers () ;
  return View () ;
}
```

Esse método Search é chamado de Action no ASP.NET MVC e ele será disparado automaticamente quando a URL /Home/Search for acessada no navegador (Home=Controller, Search=Action). Nele estamos instanciando o DataAccess, contando quantos customers existem no banco e salvando essa informação na ViewData, que poderá ser acessada mais tarde na view Search.cshtml, que vamos criar agora dentro da pasta Views:

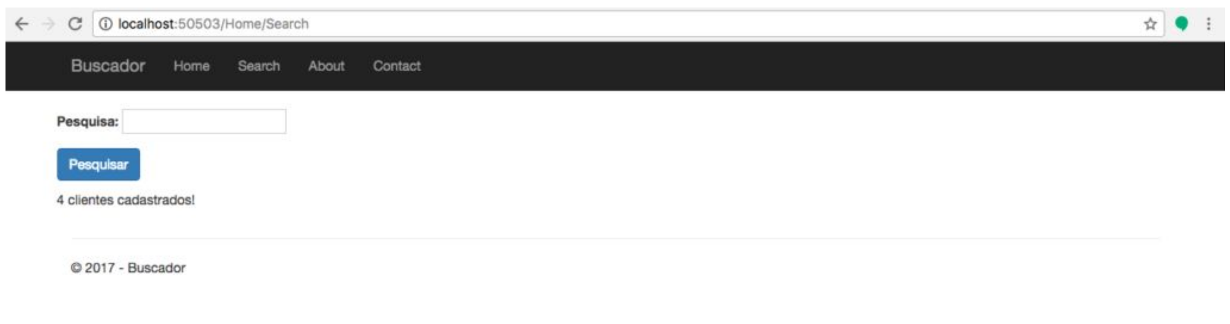
Código 5.34: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
@ {
   ViewData [ " Title " ] = " Search Page " ;
}

< div class = "row" style = " margin-top : 20 px " >
  < form method = "GET" action = "/Home/Search" >
    < p >< label > Pesquisa: < input type = "text" name = "q" /></ label >
</ p >
    < p >< input type = "submit" value = "Pesquisar" class = "btn btn-
primary" /></ p >
    < p > @Html . Raw(ViewData["Count"]) clientes cadastrados! </ p >
  </ form >
</ div >
```

Aqui eu criei um formulário de pesquisa bem tosco, apenas para mostrar o conceito funcionando para você. Temos um formulário HTML que faz um GET em /Home/Search submetendo uma variável 'q' na querystring (com o conteúdo da pesquisa) quando o botão Pesquisar for clicado.

Logo abaixo do botão eu incluí um código Razor que imprime a quantidade de documentos que tem na base, apenas para testarmos se o ASP.NET Core

está conseguindo de fato conectar e consultar o MongoDB. Mande executar e se tudo der certo, você deve ver algo semelhante à imagem abaixo:



Se você pesquisar alguma coisa, como a palavra 'autor', verá que não vai funcionar, mas vai aparecer na URL um '?q=autor'. Na sequência devemos programar o funcionamento da busca para usar essa query string aí.

Abra novamente o seu Views/Search.cshtml e inclua o seguinte código Razor na primeira linha do arquivo, antes de tudo:

Código 5.35: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
@model IEnumerable < Buscador . Models . Customer >
```

Esse código diz que o modelo de dados desta página é a classe Customer, que criamos lá atrás, lembra?

Agora ainda nesta mesma Search.cshtml, adicione o seguinte código Razor no final dela, depois de tudo:

Código 5.36: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
< hr />
@if(Model != null)
{
  < ul >
  @foreach(var item in Model)
  {
    < li > @Html . DisplayFor(modelItem = > item . Nome) </ li >
  }
  </ ul >
}
```

Esse código verifica se nosso Model tem algum resultado (!= null) o que significa que foi realizada uma pesquisa com sucesso. Se esse for o caso, ele faz um foreach entre todos os itens do Model imprimindo o nome de cada item em uma lista de elementos HTML.

Obviamente isso ainda não funciona pois nossa Action Search em HomeController.cs ainda não faz pesquisas, apenas conta os customers para fins de teste. Mas antes de voltar a mexer no HomeController.cs, vamos adicionar um novo método no DataAccess.cs para fazer a pesquisa no banco:

Código 5.37: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
public IEnumerable < Customer > GetCustomers ( string query )
{
  var tags = query . ToUpper (). Split ( new string [] { " " } ,
StringSplitOptions . RemoveEmptyEntries ). ToList () ;
  var filter = Builders < Customer >. Filter . All ( c => c . Tags , tags ) ;
  return _db . GetCollection < Customer >( typeof ( Customer ). Name ).
Find ( filter ). ToList () ;
}
```

Nesse novo método eu espero uma String que passa por uma normalização bem simples que consiste em usar apenas caixa-alta e quebrar a pesquisa por tags separadas por espaço em branco. Com essa lista de tags eu criei um Filter.All no campo Tags existente no Customer, ou seja, eu vou filtrar no banco todos os clientes que possuem TODAS (ALL) as tags informadas nesse filtro, mas não precisa ser na ordem, desde que tenha todas.

Com esse filtro pronto, é só fazer um Find na coleção Customer passando o mesmo e teremos como retorno uma lista de Customers que atendem o filtro.

Agora, para continuar, devemos voltar ao HomeController.cs e substituir o antigo método Search por esse novo, que espera um 'q' da query string com a possível busca. Digo possível porque caso seja o primeiro acesso à página /home/search, o 'q' estará vazio.

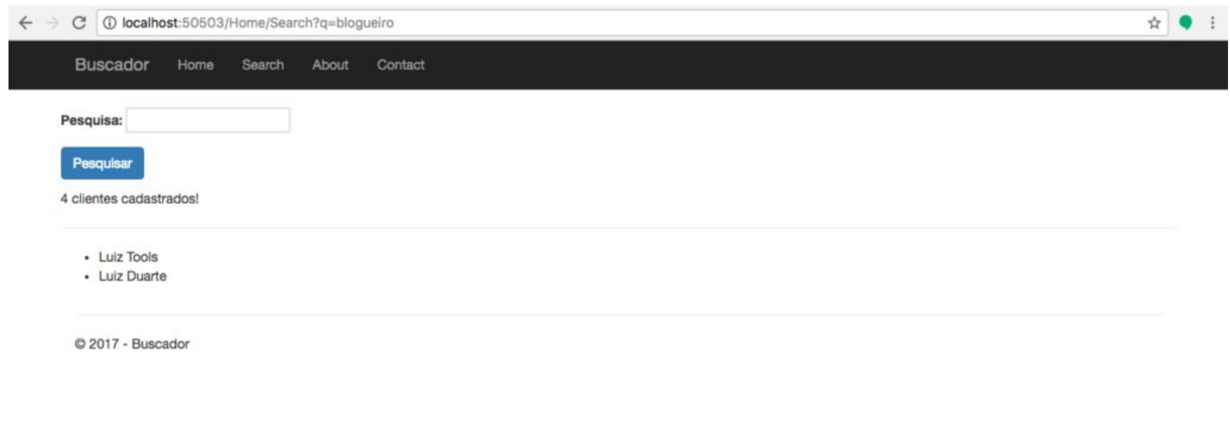
Código 5.38: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
public IActionResult Search ( String q )
{
  ViewData [ " Message " ] = " Search page. " ;
  var da = new DataAccess () ;
  ViewData [ " Count " ] = da . CountCustomers () ;

  if (! String . IsNullOrEmpty ( q ))
  {
    return View ( da . GetCustomers ( q )) ;
  }

  return View () ;
}
```

Nesta nova versão (que substitui a anterior), se vier a variável 'q' na query string (o mapeamento é automático) ele pesquisa os clientes que combinam com a pesquisa e retorna eles na View como sendo o Model dela (lembra do

@model que adicionamos anteriormente na view Search.cshtml?). Caso contrário, se não vier o 'q' na URL, apenas exibe a view normalmente.

Agora se você mandar executar e fazer uma pesquisa por palavras que existam nas tags dos customers, você verá eles listados como abaixo:



PHP: Aplicação de Cadastro

Para criar uma aplicação PHP você tem duas alternativas: configurar um servidor local ou usar um provedor de hospedagem, como a Umbler .

Caso você esteja utilizando uma empresa de hospedagem, descubra se ela possui suporte ao driver ‘mongodb’ (não use o driver ‘mongo’, ele é legado). Se essa informação não consta no seu painel de administração da hospedagem, rode um phpinfo que você descobre rapidinho, como no teste abaixo que fiz na minha hospedagem na Umbler:

mongodb

MongoDB support	enabled
MongoDB extension version	1.2.6
MongoDB extension stability	stable
libbson bundled version	1.5.5
libmongoc bundled version	1.5.5
libmongoc SSL	enabled
libmongoc SSL library	OpenSSL
libmongoc crypto	enabled
libmongoc crypto library	libcrypto
libmongoc crypto system profile	disabled
libmongoc SASL	enabled

Directive	Local Value	Master Value
mongodb.debug	no value	no value

Caso esteja rodando no seu PC local, você vai ter que instalar esse driver manualmente. O código abaixo exemplifica esse processo em sistemas Unix:

Primeiro copie e instale o driver na sua máquina:

Código 5.39: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
git clone https://github.com/mongodb/mongo-php-driver.git
cd mongo-php-driver
git submodule sync && git submodule update –init
phpize
./configure
make
sudo make install
```

Agora adicione a extensão ao seu php.ini:

Código 5.40: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
extension=mongodb.so
```

Para saber se está tudo funcionando, crie um arquivo mongoteste.php e coloque o seguinte código dentro:

Código 5.41: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$mongo = new MongoDB\Driver\Manager( 'sua string de
conexão’);
if( $mongo )
  echo “Funcionou!";
else
  echo “Não funcionou!";
```

Troque ‘sua string de conexão’ pela sua string de conexão de verdade e mande executar no seu servidor PHP. Caso esteja rodando local e sem autenticação, use:

mongodb://127.0.0.1:27017

Neste exemplo apenas estamos tentando criar um objeto de conexão com o banco de dados. Se a conexão for estabelecida com sucesso, será impresso no navegador a frase ‘Funcionou!’, caso contrário será impresso ‘Não funcionou!’ ou mesmo um erro PHP, dependendo do motivo de não ter funcionado.

O erro mais comum é o servidor estar inacessível (não está rodando) ou a connection string possuir algum erro de digitação.

Agora que sabemos que nosso ambiente está pronto e funcionando, vamos começar nosso projeto de exemplo, que será um cadastro de clientes. Crie um novo banco MongoDB chamado 'banco' e adicione os seguintes documentos nele:

Código 5.42: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
clienteArray = [{“nome”:”Luiz Júnior”, “profissao”:”Professor”},
{“nome”:”Luiz Duarte”, “profissao”:”Blogueiro”}]
db.clientes.insert(clienteArray);
```

Agora vamos criar o nosso script PHP que vai consultar os clientes no banco e listar eles em uma página web. Para isso, vamos começar criando um arquivo PHP com o nome de listagem.php e o seguinte script PHP dentro, que vou explicar na sequência:

Código 5.43: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
<?php
try {
    $mongo = new MongoDB\Driver\Manager( ‘string de conexao’ );
    $query = new MongoDB\Driver\Query([], [‘sort’ => [ ‘nome’ => 1],
‘limit’ => 5]);
    $rows = $mongo->executeQuery(“banco.clientes”, $query);
    foreach ($rows as $row) {
        echo “$row->nome : $row->profissao\n”;
    }
} catch (MongoDB\Driver\Exception\Exception $e) {
   $filename = basename(__FILE__);
   echo “Erro no arquivo $filename.\n”;
   echo “Exception:”, $e->getMessage(), “\n”;
   echo “Arquivo:”, $e->getFile(), “\n”;
   echo “Linha:”, $e->getLine(), “\n”;
}
?>
```

A primeira linha de código é para criar a conexão. Já havíamos visto isso antes, lembra? Apenas coloque a sua string de conexão para que funcione corretamente.

Código 5.44: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$mongo = new MongoDB\Driver\Manager( 'string de conexao' );
```

A segunda linha de código constrói a query que será realizada no MongoDB. O primeiro parâmetro ('[]') são os filtros, nenhum neste caso. Já o segundo parâmetro é um array associativo onde definimos a ordenação ('sort') e a quantidade máxima de documentos a serem retornados ('limit').

Código 5.45: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$query = new MongoDB\Driver\Query([], ['sort' => [ 'nome' => 1], 'limit' => 5]);
```

É na terceira linha que a consulta é de fato executada na conexão anteriormente criada. Note que devemos passar por parâmetro o path da coleção, no formato 'banco.colecao' e o objeto de consulta ($query). Isso vai nos retornar um array de linhas ($rows).

Código 5.46: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$rows = $mongo->executeQuery("banco.clientes", $query);
```

A lógica seguinte é para imprimir cada uma das linhas no formato que quisermos.

Código 5.47: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
foreach ($rows as $row) {
   echo "$row->nome : $row->profissao\n";
}
```

O tratamento de erros dispensa explicação (embora não seja recomendado em produção, para evitar expor informações do seu banco), vamos mandar ver esse arquivo listagem.php em nosso servidor e ver o que acontece:

Agora que nossa tela de listagem já está funcionando, vamos criar uma página cadastro.php, inicialmente com o formulário de cadastro dos clientes em HTML:

Código 5.48: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
<html>
<head></head>
<body>
   <form method=”POST” action=”cadastro.php”>
   <p>
                  <label for=”txtNome”>Nome: <input type=”text”
        id=”txtNome” name=”txtNome” />
           </p>
   <p>
                  <label for=”txtProfissao”>Profissão: <input type=”text”
        id=”txtProfissao” name=”txtProfissao” /></label>
           </p>
   <input type=”submit” value=”Salvar” />
   </form>
</body>
</html>
```

Se acessarmos no navegador, veremos essa tela abaixo:

Não vou entrar em detalhes do HTML mas atentar apenas ao fato de que nossa form action aponta para essa mesma página cadastro.php. Logo, vamos arregaçar as mangas e programar nosso código PHP nessa mesma página, logo antes da tag <html>, para receber o POST e salvar os dados preenchidos no nosso querido MongoDB:

Código 5.49: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
<?php
if ($_SERVER[‘REQUEST_METHOD’] === ‘POST’) {
try {
   $mongo = new MongoDB\Driver\Manager( ‘string de conexao’ );
   $bulk = new MongoDB\Driver\BulkWrite;

   $doc = [‘_id’ => new MongoDB\BSON\ObjectID, ‘nome’ =>
$_POST[“txtNome”], ‘profissao’ => $_POST[“txtProfissao”]];

   $bulk->insert($doc);$mongo->executeBulkWrite(‘banco.clientes’,
$bulk);
   header(‘Location: listagem.php’);    die();
} catch (MongoDB\Driver\Exception\Exception $e) {
    $filename = basename(__FILE__);
   echo “Erro no arquivo $filename.\n”;
   echo “Exception:”, $e->getMessage(), “\n”;
   echo “Arquivo:”, $e->getFile(), “\n”;
   echo “Linha:”, $e->getLine(), “\n”;
}
}
```

```
?>
```

Não se preocupe, vou explicar tudo direitinho!

Primeiro, vamos verificar se é um POST, pois caso contrário, não há necessidade de executar script algum.

Código5.50: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
if ($_SERVER[‘REQUEST_METHOD’] === ‘POST’) {
```

Segundo, vamos fazer conexão com o banco de dados. A essa altura do campeonato, é bem possível que você já tenha pensado em colocar esse código em algum arquivo PHP externo, não é mesmo?!

Terceiro, criei um objeto BulkWrite. Esse objeto é o mesmo para qualquer operação de escrita (inserts, deletes, updates, etc).

Código 5.51: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$bulk = new MongoDB\Driver\BulkWrite;
```

Na sequência, criei o documento ($doc, chame de objeto JSON/BSON se sentir mais confortável) que será inserido no banco, usando um array associativo, as variáveis POST que vieram pela submissão do formulário e um “_id”, que é gerado usando um objeto específico do MongoDB.

Código 5.52: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$doc = [‘_id’ => new MongoDB\BSON\ObjectID, ‘nome’ =>
$_POST[“txtNome”], ‘profissao’ => $_POST[“txtProfissao”]];
```

E por fim, adicionamos esse documento ao $bulk (o método insert é para informar documentos a serem salvos, existem ainda os métodos delete e update, para fazer exatamente o que o nome sugere) e executamos ele na coleção especificada.

Código 5.53: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$bulk->insert($doc);
$mongo->executeBulkWrite('banco.clientes', $bulk);
```

Ok, eu não citei o código que redireciona de volta para a página de listagem, mas acho que não é necessário, não é mesmo?!

Para testar o cadastro você pode voltar na página de listagem e adicionar um link HTML para a página de cadastro. Se estiver com preguiça (eu já disse como sou preguiçoso?) apenas acesse ela no seu navegador, preencha os campos e veja a mágica acontecer!

Para finalizar, vou me focar aqui nos comandos PHP de edição (update) e de deleção (delete) usando o driver mongodb, sem entrar em filosofias de como é a melhor abordagem para fazer edição e exclusão em páginas web. Para tanto, vamos relembrar um trecho de código que usamos na inserção:

Código 5.54: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$mongo = new MongoDB\Driver\Manager( 'string de conexao' );
$bulk = new MongoDB\Driver\BulkWrite;

$doc = ['_id' => new MongoDB\BSON\ObjectID, 'nome' =>
$_POST["txtNome"], 'profissao' => $_POST["txtProfissao"]];
$bulk->insert($doc);

$mongo->executeBulkWrite('banco.clientes', $bulk);
```

Agora vamos considerar que temos o seguinte documento em nosso banco de dados e que queremos atualizar ele:

Código 5.55: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
{“_id”:”abc-123-def-456”, “nome”: “Luiz Júnior”,
“profissao”:”Professor”}
```

Note que o _id dele (o identificador único de cada registro) é “abc-123-def-456”.

Agora vamos modificar o código que mostrei anteriormente para que o documento ($doc) seja uma versão atualizada do documento que mostrei logo acima (atenção ao ID, que deve ser idêntico):

Código 5.56: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$mongo = new MongoDB\Driver\Manager( ‘string de conexao’ );
$bulk = new MongoDB\Driver\BulkWrite;
$filter = [‘_id’ => “abc-123-def-456”];

$doc = [‘_id’ => “abc-123-def-456”, ‘nome’ => “Luiz Fernando”,
‘profissao’ => “Professor”];
$bulk->update($filter, $doc, [‘multi’ => false, ‘upsert’ => false]);

$mongo->executeBulkWrite(‘banco.clientes’, $bulk);
```

Atenção às partes em negrito, que são as alterações que fiz para representar a atualização de um cliente, ao invés de inserção. Basicamente foram estas alterações:

- o $filter é o filtro a ser utilizado para encontrar o(s) documento(s) que vai(vão) ser afetado(s). Neste caso o _id dá conta do recado.
- o $doc é uma cópia do documento original (incluindo _id) mas com o nome alterado (é a atualização que quero fazer). Nesse exemplo eu

vou sobrescrever o documento inteiro, embora pudesse aplicar os operadores do Mongo ($set, $inc, etc) pra alterar somente os campos que eu desejasse.

- o método usado para adicionar esse comando no $bulk é update ao invés de insert (pois esse documento será atualizado no banco, e não inserido) e ele possui um terceiro parâmetro opcional que são as updateOptions:
    - multi: se mais de um documento será atualizado
    - upsert: se o Mongo deve criar esse documento se ele não existir

Agora se quisermos excluir um documento (o mesmo acima, por exemplo), as modificações são bem simples também:

Código 5.57: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
$mongo = new MongoDB\Driver\Manager( 'string de conexao' );
$bulk = new MongoDB\Driver\BulkWrite;

$filter = ['_id' => "abc-123-def-456"];
$bulk->delete($filter);

$mongo->executeBulkWrite('banco.clientes', $bulk);
```

Note que agora só temos o $filter que só contém o _id, porque é a única informação que importa em uma deleção, certo?

E com isso concluímos esse tópico de PHP + MongoDB!

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Aulas Grátis de Node.js + MongoDB

Nesta playlist você encontra mais de 20 aulas gratuitas para aprender a usar Node.js com MongoDB.
https://www.youtube.com/playlist?list=PLsGmTzb4NxK0_CENI1ThoFUNeyIgsZ32V

Mecanismo de Busca em Node.js com MongoDB
Neste vídeo eu mostro passo a passo como construir uma aplicação Node.js usando o banco de dados MongoDB
https://www.youtube.com/watch?v=qLO8Q870fmc

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

6 Gerenciamento Básico

Some of the best programming is done on paper, really.
Putting it into the computer is just a minor detail.
— Max Kanat-Alexander

Tem alguns anos que uma nova corrente dentro da tecnologia da informação tem defendido um termo conhecido como DevOps, ou Development + Operations, onde tenta-se diminuir o gap de comunicação e colaboração entre os times de desenvolvimento da aplicação e operação da aplicação. Não é o intuito deste livro entrar no mérito de DevOps mas um bom ponto para começar a entender mais como DevOps pode e deve funcionar é os desenvolvedores das aplicações terem um noção, mesmo que básica, de como a operação das suas aplicações funciona.

Sendo assim, este capítulo de gerenciamento básico de MongoDB explora alguns conceitos essenciais de operação de um servidor MongoDB que todo desenvolvedor precisa conhecer.

Alterando metadados

Nos bancos relacionais temos duas categorias de comandos: DDL e DML. Comandos DML ou Data Manipulation Language são o equivalente ao CRUD que fizemos nas duas partes anteriores desta série. Já os comandos DDL ou Data Definition Language, são os comandos CREATE, ALTER, DROP, etc que os bancos relacionais possuem para modificar os metadados.

O MongoDB não possui uma diferenciação de dados e metadados tão clara quanto nos bancos relacionais, o que impacta diretamente no que você pode e não pode fazer depois que sua base já está em produção. Mudar o nome da sua base de dados, por exemplo, não é possível.

Apesar disso, mudar nome de coleções é possível, embora não recomendado durante horários de muito acesso ao seu banco, pois todas as queries em andamento àquela coleção são dropadas. Para mudar o nome de uma coleção, conecte-se à sua base de dados usando o utilitário mongo,

como vimos nos artigos anteriores, e depois use a função renameCollection, passando o novo nome da coleção:

Código 6.1: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
C:\mongodb\bin> db.clientes.renameCollection("clientes")
```

Note que o MongoDB não é case-sensitive nos nomes de databases, mas nos nomes de coleções, nomes de campos e valores de campos, sim.

Nos bancos relacionais, quando queremos adicionar uma nova coluna, devemos fazer um ALTER TABLE, certo?

No MongoDB não há necessidade disso, uma vez que o schema é flexível e conforme você adiciona novos documentos com novos campos, eles passam a surgir e pronto. Mais tarde, se quiser adicionar novos campos em documentos já existentes, basta usar um comando 'update' usando o update-operator $set, como já vimos anteriormente:

Código 6.2: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.clientes.updateOne({_id: ObjectId("123abc")}, {$set: {novoCampo: valor}})
```

Novamente falando dos bancos relacionais, quando queremos remover uma coluna, também temos de fazer um ALTER TABLE. No MongoDB não é necessário, basta executar um update passando o update-operator $unset, como abaixo:

Código 6.3: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.clientes.updateOne({_id: ObjectId("123abc")}, {$unset: {campovelho}})
```

E por fim, caso deseje apenas renomear um campo já existente (coisa que também iria requerer um ALTER TABLE no SQL), basta fazermos um update novamente, usando o update-operator $rename:

Código 6.4: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.clientes.updateMany({}, {$rename: {nomevelho: nomenovo}})
```

Note que usei aqui um updateMany e que como filtro passei um objeto vazio. Isso fará com que o update seja realizado sobre TODOS os documentos da coleção clientes. Outro ponto de atenção é que o $rename permite múltiplas renomeações no mesmo update, apenas informando todos os campos que serão renomeados e seus novos nomes, no formato JSON tradicional (que nem em um $set).

A título de curiosidade, e para evitar surpresas, o comando $rename é apenas um atalho para o uso do comando $unset no nome antigo, seguido de um $set com o novo nome e valor antigo. Por causa disso, certifique-se de que não ocorram colisões de nomes na sua coleção para evitar surpresas indesejadas.

Como o MongoDB não possui um schema rígido a ser seguido, geralmente atualizações de estrutura não são um grande problema por aqui e na maioria dos casos são realizadas a nível de documento (renomear campos, por exemplo).

As poucas tarefas de manipulação de coleções possíveis estão listadas abaixo:

- **db.<nome_da_collection>.drop()** : exclui a respectiva coleção e todos seus documentos;
- **db.<nome_da_collection>.renameCollection()** : muda o nome da collection para o novo nome passado por parâmetro.
- **db.<nome_da_collection>.dataSize():** retorna o tamanho da coleção;

- **db.<nome_da_collection>.stats()** : retorna diversas estatísticas;

Existem outras funções relacionadas a índices e outros recursos mais avançados, que veremos mais tarde. Além disso, é possível fazer backup de coleções, como veremos a seguir.

Analisando Comandos

O MongoDB fornece uma função chamada **explain** que quando usada no final de qualquer expressão básica sobre as coleções traz uma série de estatísticas que você pode analisar e gerar insumos para melhorias no banco de dados.

A sintaxe do comando é a seguinte (o find é apenas um exemplo de expressão):

Código 6.5: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db.customers.find().explain("executionStats")
```

Dentre as muitas informações trazidas que lhe podem ser úteis, sem sombra de dúvida o executionTime é uma delas, pois diz exatamente quanto tempo uma instrução demora para ser executada, o que lhe permite tomar decisões importantes como criar novos índices, remodelar seus documentos, etc.

Outras informações incluem o plano de execução da consulta (como que a consulta foi executada no banco de dados, quais índices usou, que características levou em conta para tomar suas decisões, etc), o número de documentos retornados (nReturned), o número de documentos examinados (docsExamined) e muito mais.

Autenticação

Durante todo este livro nossas bases de dados locais estavam operando com as configurações default do MongoDB, ou seja, sem segurança alguma. Espera-se que para projetos em produção você utilize soluções em nuvem

profissionais, que geralmente já vêm associadas a configurações de segurança mais avançadas do que vou mostrar aqui.

No entanto, é válido mesmo para seus projetos locais que o seu banco de dados possua credenciais de acesso e não fique simplesmente aberto para evitar até mesmo acidentes, de uma aplicação escrever no banco da outra porque todas usam o usuário default do sistema.

Primeiramente, para que seu servidor deixe de aceitar conexões anônimas, você deve criar um usuário admin na base de dados admin, uma base default que já vem no MongoDB.

Para criar um usuário admin, primeiro certifique-se de que seu servidor de MongoDB está executando. Depois conecte-se nele usando o cliente 'mongo'. Assim que se conectar, use o comando 'use admin' para apontar a variável db para a base de dados admin, e uma vez nela, execute o seguinte comando:

Código 6.6: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.createUser({user: "admin", pwd: "senha", roles: ["readWrite",
"dbAdmin"]})
```

Se tudo der certo, o usuário será criado com o nome e senha que você especificou, além das roles que são as permissões que esse usuário vai ter. Caso fosse um usuário de uso da sua aplicação, apenas a role "readWrite" seria o suficiente, mas como é um usuário admin, adicione a role "dbAdmin" também.

Como o servidor já estava rodando quando você criou o usuário, será necessário encerrar o servidor e executar novamente para que passem a valer as novas configurações de acesso, eliminando assim o acesso anônimo. Para fazer isso, suba o servidor novamente mas agora usando o seguinte comando:

Código 6.7: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
C:\mongo\bin> mongod --dbPath caminhoDoSeuBanco --auth
```

A flag --auth indica que esse servidor agora obriga todas as conexões a serem autenticadas. Para um maior segurança, recomenda-se ainda o uso de SSL na conexão, que foge do escopo deste livro.

Uma vez com o servidor rodando com autenticação e um usuário admin, sugere-se que você crie um novo usuário, com role de "readWrite", para cada base de dados individual. Para poder criar estes usuários, você vai ter de se conectar no seu servidor usando as credenciais novas de acesso e apontando para a base admin como sendo a base de autenticação, como abaixo:

Código 6.8: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
C:\mongo\bin> mongosh admin -u <usuario> -p <senha>
```

Uma vez autenticado como admin você pode se conectar nas bases das suas aplicações com o comando 'use' e criar os demais usuários.

Futuramente querendo alterar a senha de seus usuários, conecte-se no servidor usando um usuário que tenha permissão dbAdmin, e, após executar o comando 'use' na base de dados onde reside o seu usuário e use a função abaixo para mudar a senha:

Código 6.9: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db.changeUserPassword("usuario", "novaSenha")
```

Já querendo excluir um usuário que não tenha mais razão de existir, use o comando abaixo na respectiva base:

Código 6.10: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
> db.dropUser("usuario")
```

Backup com MongoDump

Uma das maneiras de fazer backup da sua base de dados MongoDB é usando o utilitário de linha de comando mongodump, que no passado vinha no mesmo download do Community Server mas que agora é um projeto separado, o MongoDB Database Tools.

Para baixá-lo, acesse https://www.mongodb.com/try/download/database-tools?tck=docs_databasetools

Basta extrair o zip em uma pasta da sua máquina e navegar até ela para usar o utilitário que desejar.

Antes de fazer o dump, sugiro criar uma pasta backup na mesma pasta do MongoDB, no mesmo nível da pasta bin.

Considerando que você está rodando um servidor de MongoDB localmente e sem segurança (apenas subiu um mongod como ensinado no início deste livro) o processo é muito simples. Abra seu terminal de linha de comando e navegue até a pasta bin do MongoDB. Execute o seguinte comando (considerando que tenho uma pasta backup dentro de MongoDB):

Código 6.11: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongodump --out "C:\mongodb\backup"
```

O parâmetro '--out' define a pasta onde serão salvas pastas com os nomes das databases e dentro delas os arquivos .bson contendo as collections. Os arquivos .bson contém todos os documentos JSON da coleção em questão, mas em formato binário.

Já se o seu servidor for remoto e/ou possuir credenciais de acesso (um cenário bem comum), você deve passar alguns parâmetros adicionais ao executar o utilitário mongodump:

-h <hostname>:<port>: o parâmetro -h permite especificar o endereço completo do host onde está hospedado seu banco MongoDB, incluindo porta. Ex:

Código 6.12: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongodump -h tatooine.mongodb.umbler.com:27017 --out
"C:\mongodb\backup"
```

-u <username>: o parâmetro -u permite especificar o nome de usuário que será utilizado para conectar na base a ser feito o dump.

-p <password>: a senha do usuário definido no parâmetro -u. Ex:

Código 6.13: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
C:\mongodb\bin> mongodump -h tatooine.mongodb.umbler.com:27017 -u
luiztools -p mudar123 --out "C:\mongodb\backup"
```

Restauração com MongoRestore

Uma vez que você tenha feito seu backup, pode restaurá-lo em outro servidor usando o mongorestore, que no passado vinha no mesmo download do Community Server mas que agora é um projeto separado, o MongoDB Database Tools.

Para baixá-lo, acesse https://www.mongodb.com/try/download/database-tools?tck=docs_databasetools

Basta extrair o zip em uma pasta da sua máquina e navegar até ela para usar o utilitário que desejar.

Para realizar este exemplo da maneira mais didática possível, sugiro que tenha uma conta criada em algum provedor de nuvem que ofereça serviços de MongoDB, como a Umbler (que te dá créditos para usar de graça, suficientes para este livro inteiro) ou o MongoDB Atlas, que é um serviço exclusivo de MongoDB (dão 500MB de graça pra testar). Este último eu explico como criar uma conta em mais detalhes em uma seção a seguir

Acesse a pasta bin da sua instalação de MongoDB Database Tools para encontrar o utilitário mongorestore. Use os seguintes parâmetros junto ao mongorestore para orientá-lo corretamente:

-h <hostname>:<port>: para definir endereço do servidor e porta

-d <database>: para definir o nome da base de dados no servidor de destino. Se a base não existir, ela será criada (se o usuário utilizado possuir permissão para isso)

-u <username>: para definir o nome de usuário no servidor de destino

-p <password>: a senha do usuário definido em -u

Além destes parâmetros, a última informação para executar o mongorestore corretamente é o caminho até a pasta de backup da sua database, como abaixo:

Código 6.14: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongorestore -h tatooine.mongodb.umbler.com:27017 -u luiztools -p mudar123 "C:\mongodb\backup\database"
```

Opcionalmente, você pode passar um parâmetro -c informando apenas uma coleção que deseja restaurar (ao invés de restaurar a database inteira).

Importando dados com MongoImport

O MongoImport é outro utilitário de linha de comando que no passado vinha no mesmo download do Community Server mas que agora é um projeto separado, o MongoDB Database Tools.

Para baixá-lo, acesse https://www.mongodb.com/try/download/database-tools?tck=docs_databasetools

Basta extrair o zip em uma pasta da sua máquina e navegar até ela para usar o utilitário que desejar.

Ele permite importar dados de diversas fontes para dentro do MongoDB, dentre elas, arquivos CSV, um formato muito comum de exportação de dados de diferentes sistemas.

Para realizar a importação de um arquivo CSV é muito simples, mas tenha algumas coisas em mente antes:

- um arquivo CSV pode ser importado para uma coleção;
- cada linha do CSV vira um documento na coleção;
- cada "coluna" da linha vira um campo no documento;
- o ideal é que seu CSV tenha o nome dos campos na primeira linha do arquivo, sem acentos e sem espaços em branco dentro dos nomes;
- se o campo for um número válido, ele será importado como um número, caso contrário será uma string (cuidado com as diferenças de separador decimal, que no MongoDB é o ponto '.' ao invés da vírgula brasileira);

Ou seja, não espere nada diferente disso aí, sem modelagens muito mirabolantes.

Para realizar uma importação, execute o seguinte comando:

Código 6.15: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
mongoimport -h server:port -u user -p password -d databaseName -c collectionName --type csv --file caminhoDoArquivo.csv --headerline
```

E segue uma rápida explicação sobre os parâmetros deste comando:

- **-h:** hostname e porta do servidor onde os dados serão importados;
- **-u e -p:** usuário e senha que possua permissão de escrita no banco de dados;
- **-d:** nome da base de dados onde o CSV será importado;
- **-c:** nome da coleção do banco de dados onde o CSV irá parar (se ela não existir, será criada)
- **--type:** informa o tipo de arquivo. csv e tsv são formatos válidos, por exempo;
- **--file:** caminho completo até o arquivo que será importado;
- **--headerline:** flag que indica que a primeira linha do arquivo são os nomes dos campos;

A importação pode demorar mais ou menos dependendo de onde está o servidor (na nuvem demora mais, por exemplo) e da quantidade de registros a serem importados. Se quiser fazer uns bons testes, recomendo os CSVs do Portal da Transparência do Governo Federal, onde existem muitos arquivos para download com os gastos públicos.

Atenção: *o MongoDB espera que os arquivos possuam o encoding UTF-8. Caso seu arquivo esteja em outro formato (como ISO-8859-1, popular aqui no Brasil), terá de ser convertido. Em sistemas Unix, uma forma de fazer isso é usando o utilitário iconv como abaixo, onde converto um arquivo origem.csv do encoding ISO-8859-1 para UTF-8 criando uma cópia dele de nome destino.csv:*

Código 6.15: disponível em http://www.luiztools.com.br/livro-mongo-fontes

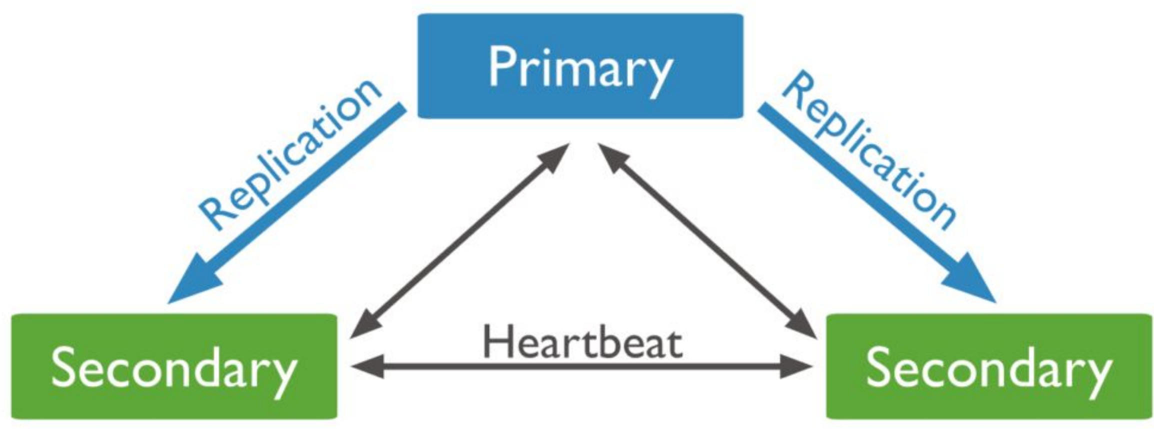
```
iconv -f ISO-8859-1 -t utf-8 origem.csv > destino.csv
```

Replicando Dados

No MongoDB chamamos o popular espelhamento de *replica set* . Um *replica set* é um conjunto de processos que mantém o mesmo conjunto de dados (*dataset*). No MongoDB, assim como em outros bancos modernos, permite fazer este espelhamento de maneira bem simples e é um dos pilares de segurança, uma vez que derrubar um servidor de banco de dados é uma das formas de tirar uma aplicação do ar, sendo que *replica sets* dificultam isso.

Não obstante, *replica sets* muitas vezes auxiliam na velocidade de leitura (pois diferentes usuários podem estar lendo de diferentes réplicas ao mesmo tempo) e podem auxiliar na velocidade de acesso, caso você possua réplicas em diferentes áreas geográficas. Outros usos para *replica sets* incluem backup (você mantém uma réplica que ninguém acessa, apenas para backup *near-realtime*), reporting (você mantém uma réplica apenas para leitura e extração de relatórios) e *disaster recovery* (você chaveia para ela, em outro continente, em caso de perder o datacenter principal).

Basicamente, a arquitetura de uma replica set é constituída de um primário e os secundários. Como mostra a imagem abaixo, exemplificando o recomendado que é 3 instâncias.



Basicamente o funcionamento é assim:

- somente o primário recebe escritas;
- todos recebem leituras;

- quando escrevem no primário, ele replica para todos os secundários;
- todos monitoram todos (heartbeat);
- se o primário cair, um secundário assume como primário e passa a receber as escritas;

Preferencialmente, todas instâncias devem possuir a mesma versão do MongoDB, para evitar problemas. Algumas funcionalidades até funcionam com duas instâncias, mas em caso de queda somente se tiver duas em pé é que uma será escolhida como primária, ou seja, permitirá escrita.

E com isso finalizamos o básico que você deve saber sobre replica sets antes de usá-las.

Criando um Replica Set

Replicar instâncias de MongoDB é muito simples, ao menos em um nível básico. Primeiro, suba com mongod uma instância de Mongo apontando os dados para uma pasta qualquer e passando o argumento replSet, como abaixo, que indica que esta instância faz parte do Replica Set "rs0".

Código 6.17: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongod --dbpath /replication/data/ -port 27018 --replSet "rs0"
```

Note que mudei a porta default, pois como vou subir mais de uma instância na mesma máquina (apenas para fins de estudo) precisarei usar portas diferentes. Você vai notar também que fica dando uns erros no terminal, dizendo que o replica set ainda não foi inicializado. Pode ignorar eles por enquanto.

Agora suba outra instância de mongod apontando os dados para outra pasta (não pode ser a mesma), com outra porta mas mantendo o argumento replSet para a mesma Replica Set.

Código 6.18: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongod --dbpath /replication/data2/ -port 27019 --replSet "rs0"
```

Agora suba a terceira instância, em uma outra porta e outra pasta, para que tenhamos acesso a todas funcionalidades do replica set.engine, and message

broker.

O próximo passo é inicializar o Replica Set com estas três instâncias. Para fazer isso, abra outra janela de terminal e se conecte via mongosh em apenas uma das instâncias, a que vai ser a primária, por exemplo, a primeira que subimos:

Código 6.19: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
mongosh -port 27018
```

Uma vez conectado nesta instância, rode o comando abaixo que inicializa a Replica Set com todas as réplicas que você possui. Aqui o recomendado é que se use os DNS públicos das instâncias e não os IPs, para maior flexibilidade.

Código 6.20: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
rs.initiate( {
    _id: "rs0",
    version: 1,
    members: [ { _id: 0, host : "127.0.0.1:27018" }, { _id: 1, host : "127.0.0.1:27019" }, {_id: 2, host: "127.0.0.1:27020"} ]
} )
```

Com isso, o Replica Set começará a funcionar depois de alguns segundos, sendo que vai notar pelas mensagens no terminal deles de que estão sincronizando dados (principalmente se um deles tinha dados e o outro não). O primary deve ser definido por eleição entre as réplicas, automaticamente. Se você quiser definir isso manualmente, adicione uma propriedade priority ao objeto member com um valor de 0 a 1000 (maior é melhor).

Se quiser adicionar outra replica mais tarde, pode se conectar no primário e usar o comando abaixo após já estar com a instância rodando como mostrei anteriormente (com replSet definido). No exemplo abaixo, eu subi ela em

localhost na porta 27021 (não use a palavra localhost, mas sim o IP 127.0.0.1).

```
rs.add({_id: 3, host: "127.0.0.1:27021"})
```

Por padrão, os secundários servem apenas como backup, ou seja, não podem ser acessados para leitura. Se quiser liberar a leitura em um secundário (lembrando que pode haver diferença mínima nos dados pois a replicação não é instantânea), use o comando abaixo na sessão do secundário:

Código 6.21: disponível em http://www.luiztools.com.br/livro-mongo-fontes

```
db.getMongo().setReadPref("primaryPreferred")
```

Já escrita é só no primário mesmo, não tem o que fazer. Agora se você se conectar a qualquer uma das instâncias da replica set notará que o console informa se você está no primário ou em um dos secundários.

Uma coisa bacana é que, se você já tiver dados em uma das instâncias quando criar a Replica Set, eles automaticamente serão replicados assim que o Replica Set for inicializado, ou seja, pode ser uma estratégia de backup bem interessante subir de vez em quando um Replica Set para espelhar seu banco.

No mais, todo dado que você adicionar no primário, a partir do momento que criou a Replica Set, serão replicados para TODOS secundários assim que possível (geralmente em poucos segundos, dependendo do volume e distância geográfica).

Caso o primário caia, uma nova eleição será feita entre os secundários e um deles vai assumir. Por isso a importância de fazer Replica Sets com no mínimo 3 membros, embora funcione somente para leitura com 2.

Usando um Replica Set

E ao usar via aplicação, o que muda? Se você se conectar diretamente ao primário (para leitura e escrita) ou a um secundário (para leitura somente), nada vai mudar e os dados apenas estarão sendo replicados em background.

Claro, se cair a instância que você está conectado, não vai adiantar estar replicado pois sua aplicação não conhece o Replica Set, mas apenas uma instância específica.

Agora, se você quer realmente aproveitar todos benefícios desta abordagem, o recomendado é se conectar informando o Replica Set, mudando sua connection string para algo similar ao abaixo:

Código 6.22: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
mongodb://127.0.0.1:27018,127.0.0.1:27019/?replicaSet=rs0
```

Caso tenha usuário e senha, adicione-os à frente da primeira instância como faria normalmente e adicione mais um parâmetro no final da URL para informar o banco de autenticação, como abaixo:

Código 6.23: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
mongodb://user:password@127.0.0.1:27018,127.0.0.1:27019/?
replicaSet=rs0&authenticationDatabase=myDb
```

Não é necessário listar todos os servidores da Replica Set na connection string, pois ao se conectar a um deles (o mais próximo e disponível) e ele informar que está na replicaSet informada na connection string, o client vai receber a informação de TODOS os demais membros da Replica Set, mesmo que alguns não estejam listados na connection string.

Isso porque a adição e remoção de membros do Replica Set acontece de maneira independente à aplicações que a usam, certo?

Ainda assim é recomendado informar mais de um membro na connection string pois pode ser que alguns membros estejam down no momento da conexão, aí o client vai tentar conectar no próximo.

Segurança em Replica Sets

Caso você esteja usando autenticação nos membros da sua Replica Set, e eu sugiro que o faça, é sempre mais fácil rodar as instâncias sem autenticação como eu fiz para fazer as configurações, no entanto, quando for colocar em produção vai precisar dela novamente. No entanto, depois de tudo

configurado, se você rodar as instâncias com --auth, vai receber o erro de "unauthorized replSetHeartbeat ...".

O processo de heartbeat entre os membros do Replica Set exige que eles confiem uns nos outros. Uma forma bem comum de fazer isso é através de keyfiles. Para criar um keyfile, use os comandos abaixo no terminal:

Código 6.24: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
openssl rand -base64 756 > caminho-pasta/key
chmod 400 caminho-pasta/key
```

Troque caminho-pasta para um caminho na pasta da sua instância de Mongo, deixe o key como está e copie o mesmo arquivo para as demais instâncias da Replica Set.

Agora quando for executar as suas instâncias, inclua o argumento --keyFile, como no exemplo abaixo, além do argumento --auth, é claro.

Código 6.25: disponível em http://www.luiztools.com.br/livro-mongo-fontes

---

```
mongod --dbpath /replication/data2/ -port 27019 --replSet "rs0" --keyFile caminho-pasta/key --auth
```

Com isso, as instâncias confiarão umas nas outras para replicação, uma vez que possuem a mesma chave e você terá um Replica Set seguro.

MongoDB na AWS

Quando o assunto é MongoDB na AWS (Amazon Web Services) você tem duas opções: criar uma instância de servidor Windows ou Linux, instalar o MongoDB dentro e configurá-lo na mão, como mostrei ainda neste capítulo. Ou então, usar um provedor de Database-as-a-Service (DbaaS) como o Atlas (antigo Mongolab ou Mlab). Nesta seção, vou mostrar como usar a solução do Atlas para subir um cluster MongoDB para sua aplicação, seja ele em qual linguagem for.

Aqui, estou pressupondo que você já possui um servidor na AWS com a sua aplicação, faltando agora subir o MongoDB em outro serviço, para juntar as

duas partes. Essa é a arquitetura de solução mais recomendada.

Por que na AWS e não em outro provedor de nuvem pública? Porque a AWS é o maior player deste mercado, seu uptime é absurdamente alto e é uma habilidade que frequentemente aparece em vagas de emprego.

O Atlas é o serviço de MongoDB em nuvem da própria empresa criadora do MongoDB, ou seja, é o melhor serviço de MongoDB existente (se você já usou o Mongolab/Mlab, saiba que foram adquiridos pelo Atlas).

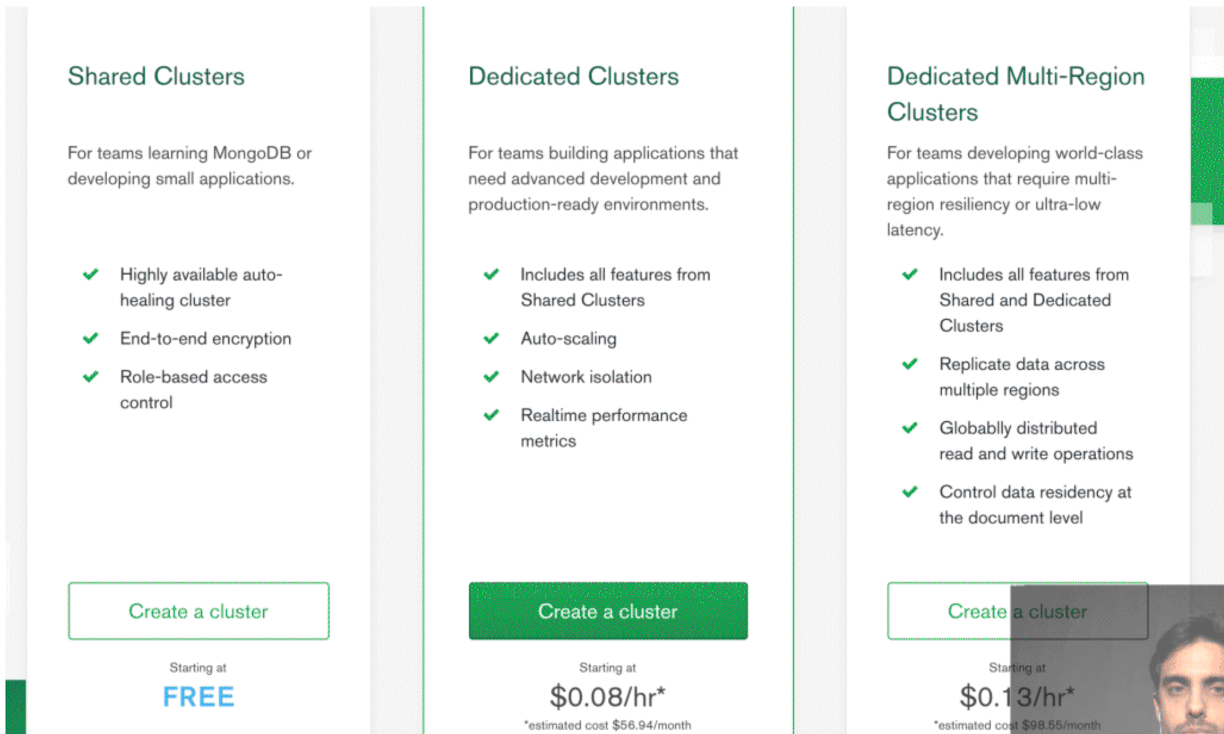
Eles possuem templates/imagens de servidores para os principais players de cloud do mercado (AWS, GCP e Azure) e com poucos cliques eles te montam um cluster próximo do seu servidor de aplicação, que é o que vou mostrar aqui nesta seção.

Criando o cluster

Primeiro, acesse o site do MongoDB e dentro de Cloud, escolha a opção Atlas. Crie uma conta (não precisa nem mesmo informar cartão de crédito) e escolha um dentre os 3 planos existentes no Atlas:

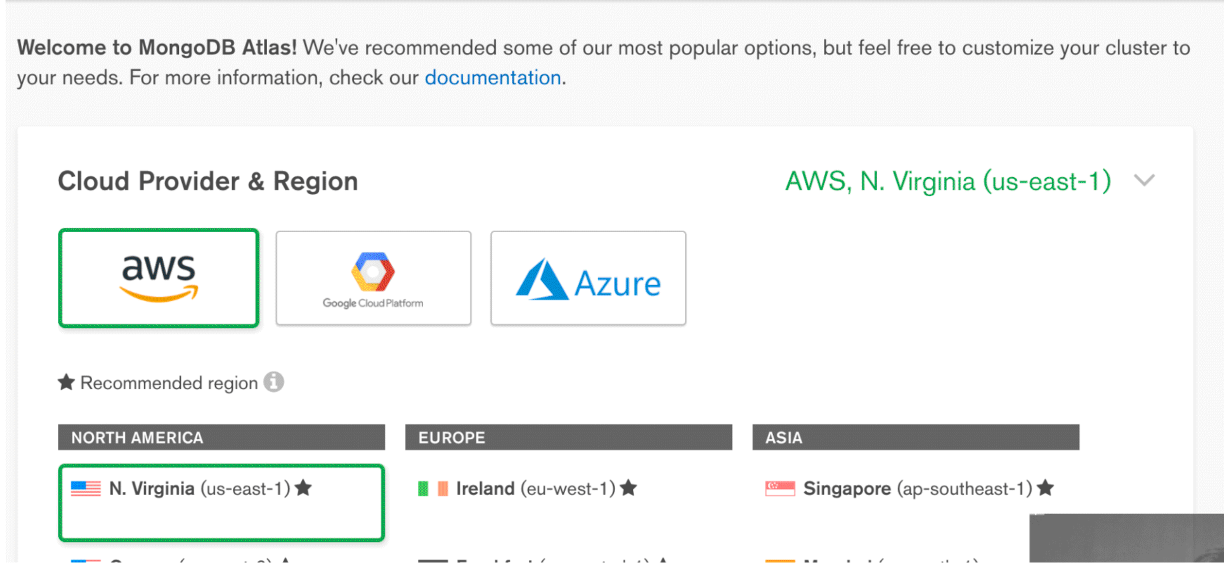
- **Shared Cluster:** parte de U$0/mês para um cluster de 3 instâncias em réplica com memória compartilhada e 512MB de espaço em disco, além de conexão criptografada e controle de acesso. Instâncias maiores custam a partir de U$9/mês.
- **Dedicated Cluster:** parte de U$56,94/mês para um cluster de 3 instâncias dedicado e recursos avançados como auto escalonamento, rede dedicada e métricas em tempo real.
- **Dedicated Multi-Region Clusters:** parte de U$98,55/mês para um cluster distribuído geograficamente.

Escolha a opção free, a da esquerda na imagem abaixo (desculpe meu rosto na imagem, tirei esse print durante a gravação de uma videoaula pro meu curso).

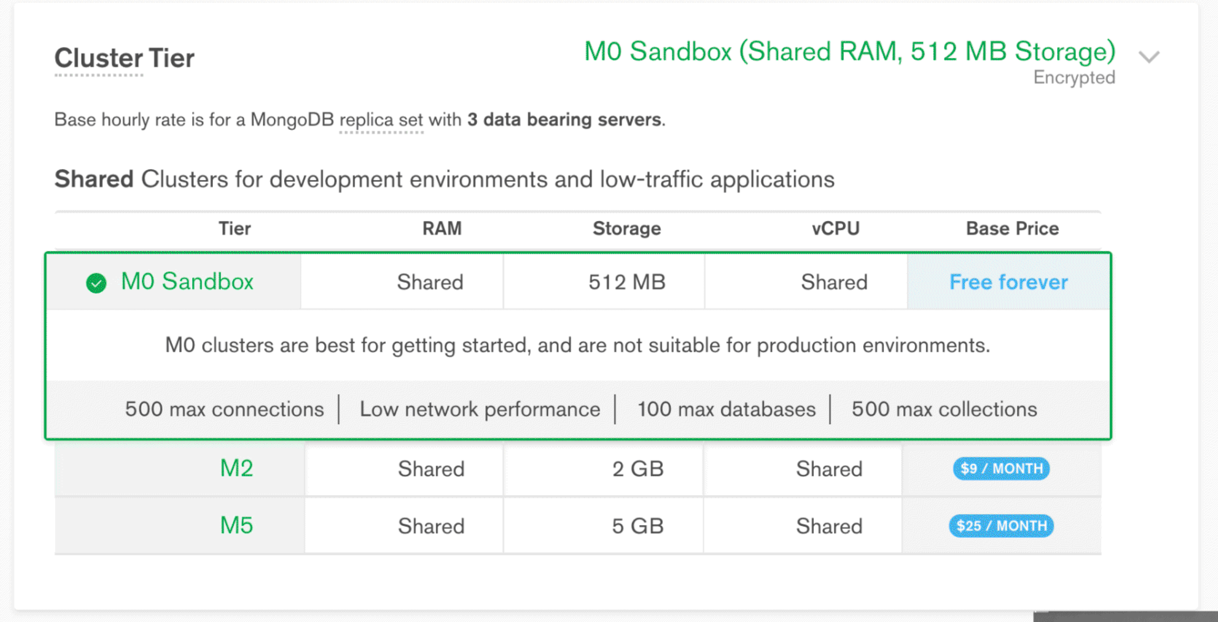


Em Cloud Provider & Region, escolha AWS na região de onde estão seus servidores. Neste passo-a-passo eu escolhi **N. Virginia** , pois é onde crio geralmente minhas aplicações.

É importante que, apesar da aplicação e do Mongo ficarem em instâncias separadas, que eles fiquem na mesma região por questões de performance. Aguarde alguns minutos até a criação do seu cluster gratuito.

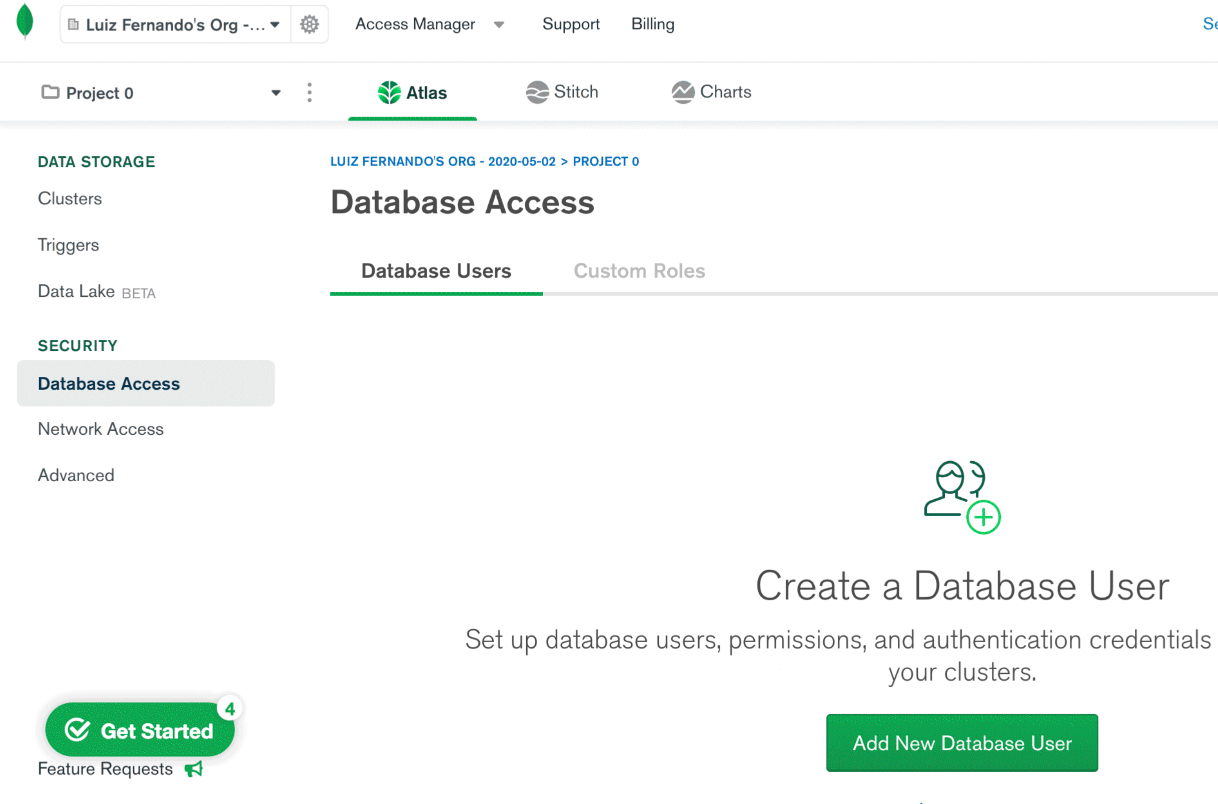


Logo abaixo, você tem mais configurações do cluster, sendo que a opção free é a M0 Sandbox, existindo opções pagas com 2GB e até 5GB de armazenamento. O legal desse plano inicial do Atlas é que ele é grátis pra sempre, ou seja, pode usar ele mesmo em produção, desde que não ultrapasse seus limites.



Configurando o cluster

Após o cluster ter sido criado, você terá acesso a um dashboard de monitoramento do mesmo, falarei dele mais tarde. Antes disso, acesse no menu da esquerda a opção Database Access e depois clique no grande botão verde de "Add New Database User", como abaixo.

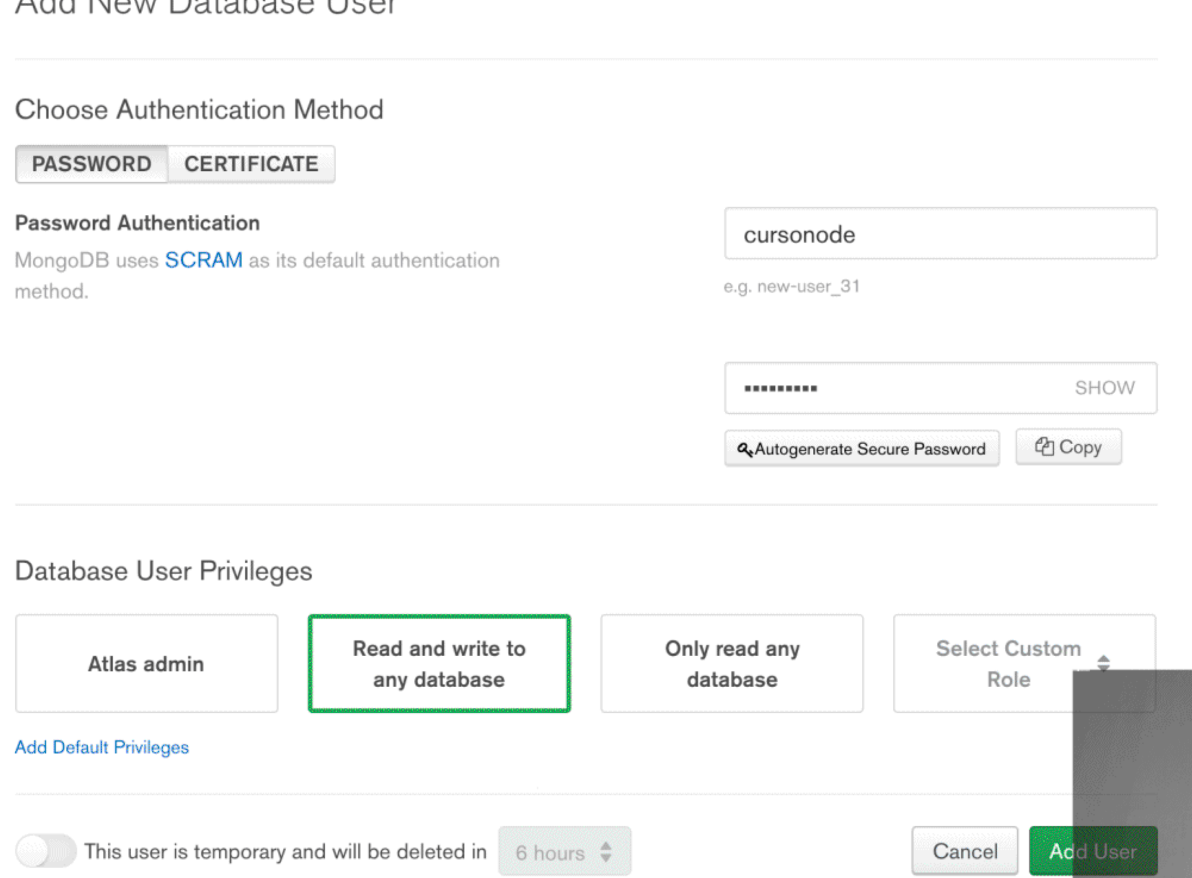


Isso irá abrir um formulário de criação de usuário, com alguns campos a serem configurados. Em Authentication Method, escolha Password, crie um username e um password para esse usuário. Em Database User privileges, temos 4 opções:

- **Atlas admin:** administrador geral de todo o cluster, evite esta opção;
- **Read and write to any database:** um usuário com permissão em todas bases deste cluster;
- **Only read any database:** um usuário com permissão de leitura em todas as bases deste cluster;
- **Select custom role:** personalizável;

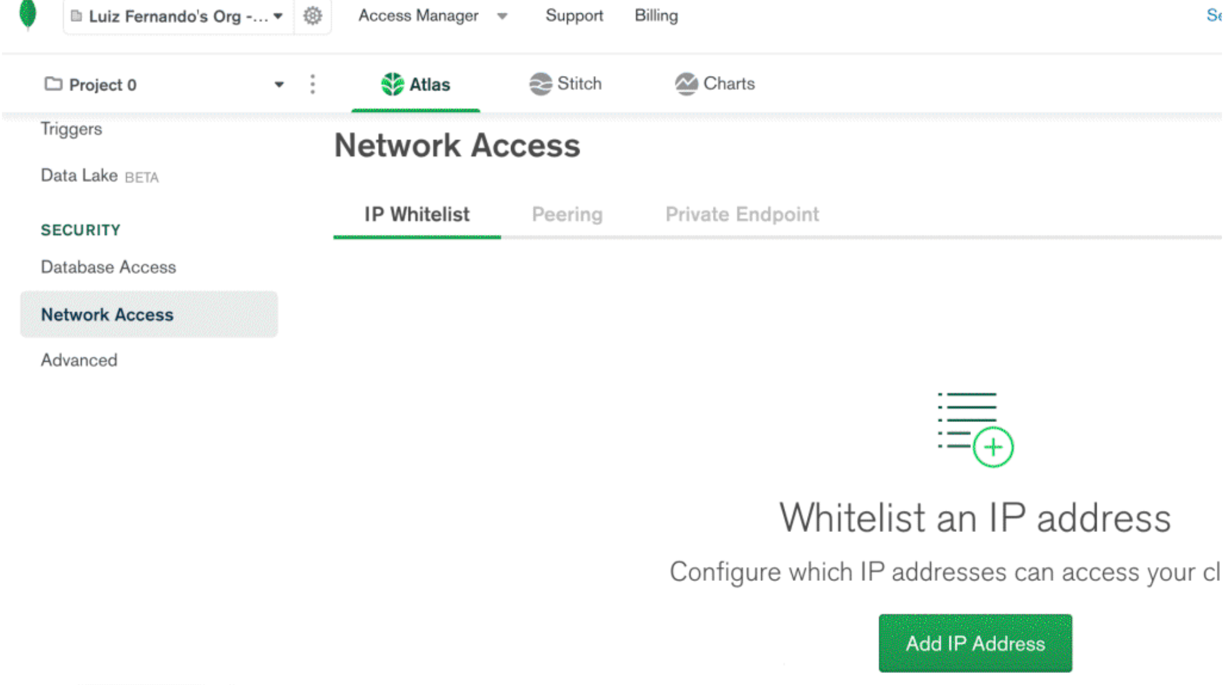
Para simplificar neste momento, sugiro a opção "Read and write any database", como mostrado na imagem abaixo. Mais tarde, dê uma estudada

nos privilégios existentes e crie um user para cada database, pois fica mais seguro assim.



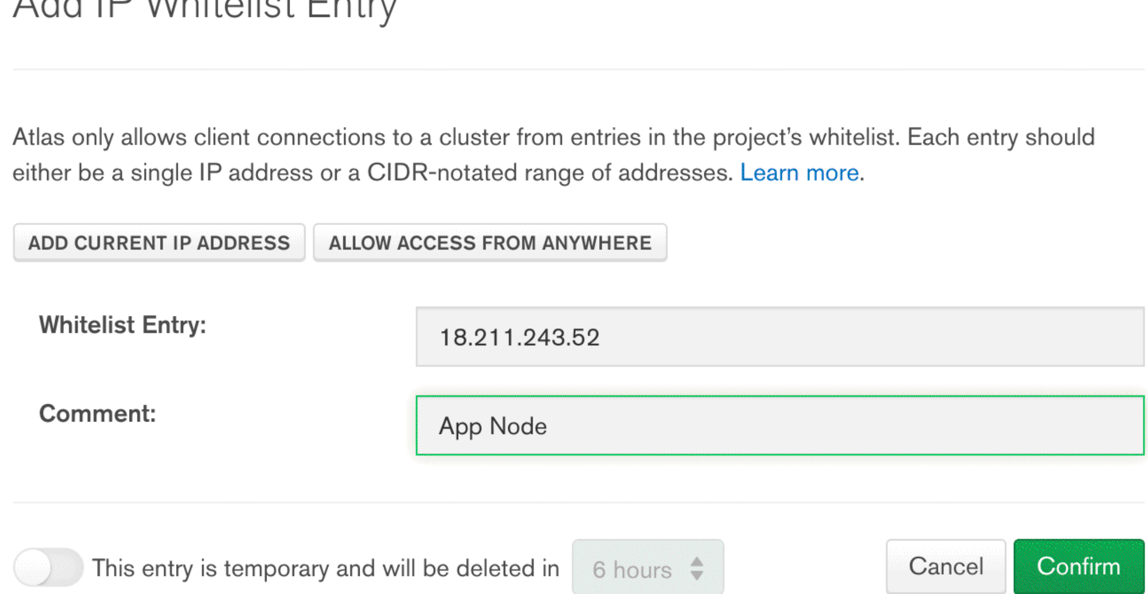
O próximo passo é permitir que a sua instância de aplicação na AWS possa acessar a instância de MongoDB. Por padrão, o cluster bloqueia qualquer tentativa de acesso externo a ele e para que seja possível a conexão entre nossas instâncias, devemos colocar nosso IP na whitelist do cluster.

Para fazer isso, acesse o menu Network Access na esquerda e clique em "Add IP Address".



Digite o IP estático do seu servidor de aplicação na AWS e deixe um comentário que explique o que é esse IP (sugiro citar que é do servidor de aplicação). De forma alguma recomendo usar a opção "Allow Access from Anywhere" pois ela abre o seu MongoDB para o mundo poder se conectar e isso não é legal.

Na imagem abaixo, estou liberando acesso para um servidor de aplicação Node.



Agora é hora de conectar no nosso cluster, para que nossa aplicação utilize de fato o nosso servidor de MongoDB.

Volte à tela inicial dos clusters e clique no botão Connect do seu cluster. Ele te dá algumas opções, queremos a **Connect your application** . As outras opções incluem informações de como se conectar via terminal de linha de comando e de como se conectar usando a aplicação MongoDB Compass, que nada mais é do que um client visual de MongoDB.

Selecione o driver e versão que estiver usando da sua linguagem e programação e copie a sua connection string. Note que ela estará sem a senha, que você deve preencher manualmente por uma questão de segurança.

Pronto, sua instância de MongoDB está pronta e acessível pela sua aplicação que está na AWS. Aliás, os dois serviços são agora vizinhos na AWS!

Para transferir os seus dados locais para este cluster na nuvem, basta usar os recursos que mostrei neste capítulo de import, restore, etc.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Curso de Node.js e MongoDB
No último módulo deste curso eu ensino em videoaulas passo a passo como fazer administração de servidores MongoDB, além de ensinar como subir um servidor na AWS.
https://www.luiztools.com.br/curso-nodejs

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

7 Boas práticas

I'm not a great programmer;
I'm just a good programmer with great habits.
- Kent Beck

Quando estamos começando com uma nova tecnologia sempre tudo é muito confuso, complexo e, por que não dizer, assustador?

Trabalho desde 2007 profissionalmente com programação e programo mesmo que não profissionalmente, desde um técnico em eletrônica que fiz em 2004 que incluía programação de microcontroladores para a indústria. Nesse ínterim tive de trocar sucessivas vezes de tecnologias, seja por questões acadêmicas, seja por questões de mercado e isso incluiu banco de dados, que comecei com MS Access, passei rapidamente por Firebird, usei MySQL, durante vários anos MS SQL Server, SQLite em apps e, agora, muito MongoDB.

E a cada troca eu volto a ser um calouro, tendo de aprender não apenas modelagem, comandos, todo o "ferramental" relacionado à tecnologia, mas o mais difícil de tudo: boas práticas!

E tudo fica pior quando estamos falando de uma tecnologia que não tem sequer uma década de vida ainda e até cinco anos atrás era coisa de devs malucos em eventos. São muitas as dúvidas sobre MongoDB e, mesmo com o todo-poderoso Google em nossas mãos, parece que essas respostas estão desconectadas e jogadas cada uma em um canto obscuro da Internet, fazendo com que dê um trabalho enorme encontrá-las.

Não trago aqui verdades universais ou respostas para todas as perguntas que você vai ter sobre Mongo, mas trago um apanhado de boas práticas oficiais da empresa que desenvolveu o Mongo, praticamente traduzidas ao pé da letra. Busquei selecionar as dicas mais práticas e aplicáveis em boa parte dos cenários de quem está iniciando com Mongo.

Boas práticas de Hardware

Para os desenvolvedores, hardware é algo que não recebe tanta importância (até apresentar problemas!). Não colocar o seu banco MongoDB em um servidor apropriado pode causar muitas dores de cabeça com a tecnologia. É como diz o ditado: o barato sai caro!

Ao contratar serviços em nuvem, como os da Umbler e outros players, procure se informar do tipo de hardware utilizado, como tecnologias de disco e memória, além de dimensionar corretamente a instância de servidor onde estará o MongoDB, é claro. Sobre este último, mais dicas serão apresentadas adiante.

Ah, e não use ambientes compartilhados de MongoDB para aplicações em produção. Nunca! Prefira sempre instâncias e containers dedicados para este serviço ficar executando.

Tenha memória RAM suficiente

Como a maioria dos bancos de dados, o MongoDB funciona melhor quando as suas coleções e índices mais utilizados cabem completamente na memória RAM do seu servidor. E, para saber o quanto de RAM cada componente ocupa, use a função stats(). No exemplo abaixo, mostro os stats da coleção 'customers':

```
> db.customers.stats()
{
   "size" : 715578011834,  // total size (bytes)
   "avgObjSize" : 2862,    // average size (bytes)
}
```

Para ter uma noção mais geral, use o comando db.serverStatus() para ver uma estimativa do tamanho do working set atual (dados mais utilizados).

A quantidade de memória RAM é o fator mais importante quando o assunto é hardware. O resultado de outras otimizações nem se compara com a melhoria de performance que ter RAM suficiente proporciona ao seu sistema. Se o seu working set excede a capacidade de memória de um único servidor uma dica é fazer sharding dele em vários servers.

Use discos adequados

A maioria dos acessos a dados em bases MongoDB não são sequenciais e, como resultado, são muito mais velozes em discos de acesso aleatório como SSDs, tanto SATA, PCIe e NVMe. Mesmo unidades flash domésticas possuem uma performance superior quando utilizados em uma máquina com boa quantidade de memória RAM (vide dica 1.1) frente aos discos rígidos de alta performance de servidores como SAS e SCSI.

No entanto, enquanto os arquivos de dados beneficiam-se dos SSDs, os arquivos de log do MongoDB são bons candidatos para serem armazenados em discos tradicionais devido ao seu perfil de escrita sequencial, sendo que, quando necessário, o uso de RAID-10 é o mais indicado para a maioria das aplicações.

Use múltiplos cores

Pode soar um tanto óbvio, mas o MongoDB realmente opera melhor em ambientes com CPUs rápidas e, especialmente, com a storage engine WiredTiger (a mais recente e recomendada na data que escrevo este artigo), pode-se tirar enorme proveito de multi-cores. Storage engines mais antigas como MMAPv1 não possuem esse mesmo benefício com múltiplos processadores, possuindo aumento de desempenho insignificante em ambientes mais modernos.

O uso de múltiplos cores, aliado ao modelo de concorrência da WiredTiger, fazem com que múltiplas operações concorrentes no seu banco de dados se tornem extremamente mais velozes do que em ambientes single core, na proporção de uma operação concorrente por core.

Boas práticas de aplicação

Ter o ambiente adequado para subir o seu banco de dados é só a primeira etapa rumo a tirar o máximo proveito do MongoDB. Geralmente a construção do ambiente adequado gera um custo direto que nos leva a crer (erroneamente) que é o requisito mais caro de qualquer projeto - quando na verdade não é. A construção correta da aplicação que consumirá o banco de dados muitas vezes é o que consome mais dinheiro a médio e longo prazo, embora seja um custo diluído ao longo do ciclo de vida do projeto.

Neste tópico, vamos ver algumas dicas de uso do MongoDB em suas aplicações, visando a maior performance possível do banco.

Atualize apenas os campos alterados

O comando replaceOne do MongoDB recebe por padrão o documento JSON, que irá substituir o atual com o mesmo _id usado como filtro, certo? No entanto, substituição de documentos inteiros gera um overhead desnecessário no banco de dados e não é recomendada.

A boa prática aqui é utilizar os operadores de update para alterar somente os campos que de fato foram atualizados pela sua aplicação. Ou seja, se somente o nome do cliente foi editado, envie apenas essa alteração ao banco de dados, diminuindo com isso o tempo de update e o uso de rede.

Relembrando os operadores de update:

- **$set:** muda o valor de um campo;
- **$unset:** remove um campo do documento;
- **$rename:** muda o nome de um campo do documento;
- **$inc:** quando um campo numérico precisa ser incrementado (ou decrementado, usando um valor negativo);
- **$mul:** quando um campo numérico precisa ser multiplicado;

Note que como o ACID do MongoDB é garantido somente a nível de documento, um banco com uma concorrência grande pode ter problemas de update caso você opte apenas por utilizar o operador $set (que é o mais simples). Especialmente em campos numéricos que precisem ser alterados,

dê preferência aos operadores $inc e $mul, que operarão sobre o valor atual do campo, sem que você tenha de conhecê-lo com antecedência o que poderia causar inconsistências caso o update de outro usuário seja mais veloz no mesmo campo.

Evite negações em queries

Como a maioria das bases de dados, o MongoDB não indexa a ausência de valores e condições de negação podem exigir que todos os documentos de uma coleção sejam analisados para dar um retorno à query. Se a negação é a única condição e ela não é seletiva (por exemplo, pesquisar uma coleção de pedidos onde 99% dos pedidos estão finalizados para encontrar aqueles que não estão), todos os registros serão escaneados em busca dos documentos que preencham a negação.

Embora não tenha como você fugir de algumas negações de vez em quando, experimente usar mais os operadores $e, $in e $all do que o operador $ne, sendo o mais seletivo possível se quiser uma performance maior nas consultas. Modelar o banco já pensando nas consultas que irá fazer ajuda bastante, como será discutido no tópico 3.

Analise o plano de execução

Para cada query importante da sua aplicação e/ou queries que estejam com mau desempenho, analise o plano de execução da mesma usando a função explain() no final da chamada.

Esta função exibe informações sobre como a query será (ou foi) executada, incluindo:

- Número de documentos retornados;
- Número total de documentos lidos;
- Quais índices foram utilizados;
- Se a consulta foi coberta, ou seja, se ela conseguiu retornar resultados sem ter de ler documentos;
- Se uma ordenação in-memory foi realizada, o que indica que a criação de um índice poderia ser benéfica;

- O número de entradas do índice que foram escaneadas;
- Quanto tempo a consulta levou para retornar, em ms;
- Quais outros planos de execução foram rejeitados pelo motor de consulta do Mongo, indicando ainda o tempo que essa decisão levou, geralmente 0ms (o que indica menos que 1ms)

A ferramenta MongoDB Compass, disponível nas assinaturas Enterprise e Professional do MongoDB, permite visualizar planos de execução de uma maneira bem prática e didática, sendo que cada estágio do pipeline de execução é exibido como um nó em uma árvore, como mostra a figura abaixo:

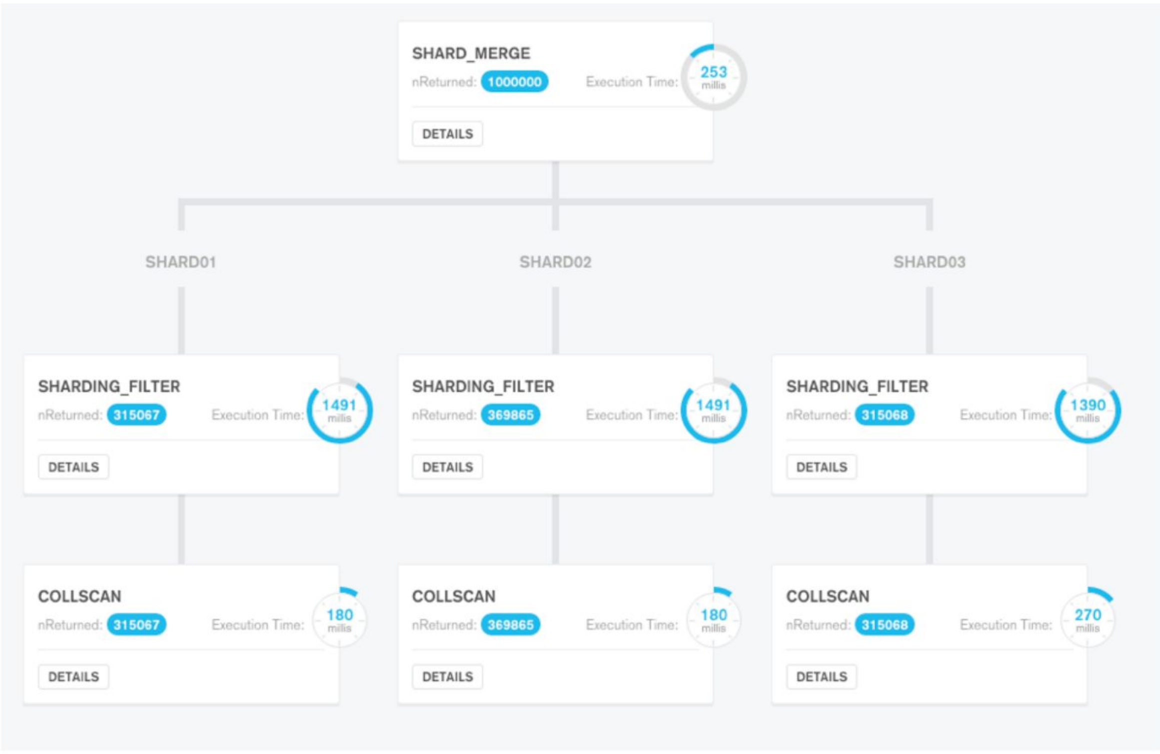


Figure 1: MongoDB Compass visual query plan for performance optimization across distributed clusters

Configure a garantia de escrita apropriada

Muito se discute sobre a qualidade da persistência de dados do MongoDB frente aos bancos relacionais mais maduros e líderes de mercado. No entanto, testes e mais testes comprovam que falhas de ACID geralmente são provocadas por má programação, e não pelo banco de dados em si.

Isso porque ao contrário dos bancos relacionais que possuem escrita bloqueante por natureza (e que você pode desabilitar manualmente com NOLOCKs), o MongoDB permite definir o nível de garantia de persistência com o que ele chama de “ *write concerns* ”. Essas configurações podem ser realizadas a nível de banco, de coleção, de conexão ou em uma granularidade extremamente fina: a cada operação.

Note que com grandes poderes vem grandes responsabilidades e definir o *write concern* correto cabe ao analista/arquiteto responsável pela aplicação, sendo que a norma padrão garante apenas performance genérica de escrita, mas que pode não atender ao seu caso específico, considerando a concorrência de leitura e escrita do seu banco.

As opções de *write concerns* são as seguintes:

- **Write Acknowledged:** opção padrão e mais veloz. O mongod confirma a execução da operação de escrita dos dados in-memory, permitindo ao cliente detectar rapidamente erros na operação como chaves duplicadas, falhas de comunicação, etc.
- **Journal Acknowledged:** o mongod confirma a operação somente depois que ela foi registrada no log com sucesso, o que permite que ela sobreviva a uma queda do serviço, garantindo maior durabilidade. Opção mais segura, mas mais lenta.
- **Replica Acknowledged:** válido apenas quando se está trabalhando com 'replica sets'. O mongod confirma que o dado não apenas foi escrito no log da réplica primary como nas demais réplicas (você pode configurar de quantas réplicas quer a confirmação).
- **Majority:** válido apenas para cenários com 'replica sets', definindo que a maioria das réplicas também devem confirmar que realizaram a escrita dos dados e dos logs, sempre incluindo a réplica primary.
- **Data Center Awareness:** mais avançada de todas opções, permite customizar a garantia de escrita em ambientes multi-server.

Caso não lembre como definir essas opções, consulte a documentação oficial do MongoDB, mas a nível de operação. Por exemplo, pode ser feito passando um segundo parâmetro para a função de escrita (insert, nesse

caso). Nesse segundo parâmetro você informa um objeto com o tipo de *write concern* (w:1 é o padrão) e se deve aguardar pela persistência nos logs ou não (j:true).

```
db.collection.insert({data}, {w:1, j:true})
```

Use os drivers mais recentes

MongoDB possui drivers de conexão para dezenas de linguagens de programação. Estes drivers são criados pelos mesmos engenheiros que mantém o kernel do banco de dados, mas tendem a ser atualizados de maneira mais constante do que o mesmo, tipicamente a cada dois meses. Sempre use a versão mais recente dos drivers para aproveitar ao máximo a estabilidade, performance e recursos implementados regularmente.

Boas práticas de Modelagem e Índices

O MongoDB usa um modelo de dados baseado em documentos binários chamado BSON, que é baseado no padrão JSON. Ao invés das tabelas de uma base relacional, o modelo de dados em documentos do Mongo é fortemente alinhado ao modelo de objetos usado nas linguagens de programação modernas e na maioria dos casos não há a necessidade de transações ou joins complexos devido às vantagens de se ter os dados relacionados ao objeto dentro do próprio objeto (lembra-se de agregação e composição na disciplina de OO e Engenharia de Software?). Note como isso é completamente diferente da divisão em tabelas e os relacionamentos entre elas para compor um objeto na aplicação a partir do banco.

No entanto, mesmo em tese sendo mais fácil de modelar informações em documentos do que em tabelas e relacionamentos, é necessário conhecer algumas boas práticas gerais e ter em mente que o analista/arquiteto do banco de dados é o único que pode realmente dizer qual a melhor abordagem a ser tomada na modelagem e criação de índices para uma aplicação em particular.

Armazene todos os dados de uma entidade em um único documento

Existem vários bons motivos para você seguir esta regra, mas vou citar os dois mais importantes: ACID e performance.

O MongoDB garante conformidade com ACID a nível de documento. Se os dados da sua entidade estiverem espalhados entre diversos documentos, não há garantia alguma da consistência e integridade entre eles. Se ele estiver espalhado entre diversas coleções, pior ainda: você corre o risco de perder a atomicidade também.

Além disso, quando os dados do seu registro são armazenados todos dentro de um mesmo documento, você consegue retornar todos os dados do mesmo com uma única consulta por _id no banco de dados, o que é a forma mais eficiente de se retornar dados em qualquer banco. Se pegarmos um exemplo de ecommerce moderno, além do schema instável que os grandes varejistas possuem e que complica a vida dos bancos relacionais, a tela de detalhes de um produto hoje em dia possui muitas informações que podem ser armazenadas juntamente com o produto, tais como:

- reviews do produto;
- versões do produto;
- informações de entrega;
- imagens;

Enquanto que, em um banco relacional, todas essas informações que possuem cardinalidade variável virariam outras tabelas e usaríamos chaves estrangeiras na tabela de produto, no Mongo podemos (e devemos) colocar todas em sub-documentos e campos multi-valorados sem problema algum. E esse é um cenário real e comum mesmo aqui no Brasil, entre os grandes varejistas que já estão utilizando bancos não-relacionais.

No entanto, em alguns casos não é muito prático armazenar todos os dados em um único documento, impactando negativamente em outras operações

como updates por exemplo. Lidar com esses trade-offs é responsabilidade do analista/arquiteto da aplicação.

Evite documentos muito grandes

Apesar da premissa anterior de salvar todas as informações de um registro no mesmo documento, tenha em mente que o tamanho máximo que o MongoDB suporta para um único documento é 16MB (na data que escrevo este artigo). Na prática, a maioria dos documentos ocupam apenas alguns KB, pois a analogia é a de que um documento seria uma linha na tabela de um relacional - não use documentos como se fossem tabelas inteiras.

Geralmente essa barreira de 16MB é um problema, caso você deseje armazenar binários em conjunto com seus dados, como vídeos e imagens. Neste caso use GridFS, uma convenção implementada por todos os drivers MongoDB que distribuem os binários em documentos menores indexados pelo documento principal.

Embora não seja uma recomendação oficial, não vejo (e nunca vi) muito sentido em armazenar binários no banco de dados, uma vez que você jamais vai consultá-los. Se é apenas para rápido retorno, usar arquivos estáticos na nuvem com CDN é muito rápido atualmente. No entanto, podem existir cenários onde isso é realmente necessário, então tome cuidado com a limitação.

Evite nomes de campos desnecessariamente longos

Os documentos BSON são schemaless, ou seja, não existem meta-informações dos documentos salvas em uma tabela de schemas como no caso dos bancos relacionais. Enquanto que isso te torna mais livre para criar documentos com diferenças entre eles e sem uma grande formalização dos dados, isso te cria um grande problema que é o armazenamento dessas meta-informações no próprio documento.

Todos os nomes de campos são armazenados em cada um dos documentos, o que faz com que o consumo de espaço em disco em um banco MongoDB geralmente seja muito maior do que em bancos relacionais, que fazem um

uso mais inteligente desse recurso, evitando repetições através de schemas e relacionamentos entre tabelas. Se você tem um milhão de clientes armazenados em documentos, além dos valores dos campos, eles possuem armazenado o nome dos campos.

Uma forma de minimizar esse problema é usar nomes curtos, geralmente apenas as iniciais dos campos e através de programação, fornecer APIs amigáveis para acesso e a manipulação dessas informações nos seus objetos de aplicação (como gets e sets do Java, por exemplo, referenciando os nomes curtos).

Em versões mais recentes do MongoDB, que possuem a storage engine WiredTiger, existe uma compressão automática de nomes grandes para que eles não prejudiquem tanto a performance do sistema.

Mantenha a manutenção dos índices em dia

O uso de índices em MongoDB segue os mesmos princípios dos bancos relacionais: melhoram a leitura e prejudicam a escrita dos dados. Sendo assim, caso possua índices que não estão sendo utilizados, remova-os, para garantir que eles não atrapalhem a escolha dos planos de execução pelo motor de consulta do Mongo e principalmente não prejudique a performance de inserts, updates, etc.

Para entender o uso e determinar se um índice merece ou não existir, uma agregação com $indexStats permite saber com que frequência os índices são atualizados, além de algumas ferramentas como MongoDB Compass lhe darem essa informação de maneira rica e visual.

Outra dica é sempre verificar se a sua coleção não deveria ter um índice composto ao invés de vários índices individuais. Caso você possua consultas que sempre usam os mesmos campos, como filtro, e estes campos possuem índices que não são usados individualmente, opte por condensá-los em um único índice composto que será muito mais performático.

Outras dicas gerais

Mais algumas dicas rápidas e muito úteis para quem está começando com MongoDB e não quer cometer alguns erros bem comuns:

- Evite campos multivalorados com muitos elementos, pois a performance em atualizações e inserções vai ficando cada vez mais onerosa. Prefira campos definidos na raiz do documento;
- evite expressões regulares sem início e/ou fim (ˆ e $, respectivamente), uma vez que os índices são ordenados pelos valores, o que faz com que todo o índice tenha de ser percorrido em expressões regulares ineficientes como essas;
- scripts complexos do banco de dados devem ser salvos em arquivos .js e versionados junto ao seu projeto da aplicação;
- na dúvida de como modelar seu banco? Faça-o pensando na experiência de consulta, que é o foco do modelo orientado à documentos;

Estas foram algumas dicas e boas práticas de uso do MongoDB. Embora não existam "verdades universais" quando o assunto são os bancos não-relacionais, as recomendações oficiais são sempre um bom "norte" para quem está começando com tecnologias que não conhecemos bem e é isso que quis trazer neste capítulo, sendo que todas essas dicas (e muitas outras), podem ser encontrados nos artigos técnicos do MongoDB .

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Node.js e MongoDB Tips
Nesta playlist você conhece várias dicas de Node.js e de MongoDB também.
https://www.youtube.com/playlist?list=PLsGmTzb4NxK0ce0Yvrpqw6nFx3iQYZBNb

---

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse https://www.luiztools.com.br/curso-nodejs

OceanofPDF.com

Seguindo em frente

A code is like love, it has created with clear intentions at the beginning, but it can get complicated.
— Gerry Geek

Este livro termina aqui.

Pois é, certamente você está agora com uma vontade louca de aprender mais e criar aplicações incríveis com MongoDB, que resolvam problemas das empresas e de quebra que o deixem cheio de dinheiro na conta bancária, não é mesmo?

Pois é, eu também! :)

Este livro é pequeno se comparado com o universo de possibilidades que o MongoDB nos traz. Como professor, costumo dividir o aprendizado de alguma tecnologia (como MongoDB) em duas grandes etapas: aprender o básico e executar o que foi aprendido no mercado, para alcançar os níveis intermediários e avançados. Acho que este guia atende bem ao primeiro requisito, mas o segundo só depende de você.

De nada adianta saber muita teoria se você não aplicar ela. Então agora que terminou de ler este livro e já conhece uma série de coisas incríveis que pode fazer com esta fantástica tecnologia, inicie hoje mesmo (não importa se for tarde) um projeto de aplicação que use Mongo. Caso não tenha nenhuma ideia, cadastre-se agora mesmo em alguma plataforma de freelancing. Mesmo que não ganhe muito dinheiro em seus primeiros projetos, somente chegarão os projetos grandes, bem pagos e realmente interessantes depois que você tiver experiência.

Me despeço de você leitor com uma sensação de dever cumprido. Caso acredite que está pronto para ainda mais tutoriais bacanas, sugiro dar uma olhada em meu blog https://www.luiztools.com.br .

Outras fontes excelentes de conhecimentos específico de MongoDB é o site oficial http://www.mongodb.org , lá tem tudo que você pode precisar!

Caso tenha gostado do material, indique esse livro a um amigo que também deseja aprender a programar para web com Node. Não tenha medo da concorrência e abrace a ideia de ter um sócio que possa lhe ajudar nos projetos.

Caso não tenha gostado tanto assim, envie suas dúvidas, críticas e sugestões para contato@luiztools.com.br que estou sempre disposto a melhorar.

Um abraço e até a próxima!

Curtiu o Livro?

Deixe sua avaliação na página do livro na Amazon: https://www.amazon.com.br/review/create-review/ref=cm_cr_dp_d_wr_but_top?ie=UTF8&channel=glance-detail&asin=B077711577

Me siga nas redes sociais: https://about.me/luiztools

Conheça meus outros livros: https://www.luiztools.com.br/meus-livros

Conheça meus cursos online: https://www.luiztools.com.br/meus-cursos

OceanofPDF.com