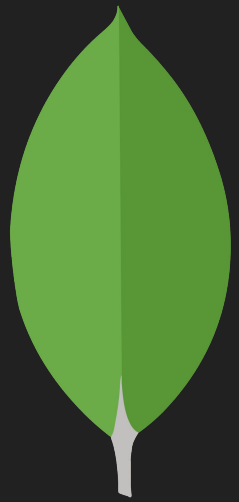


# Curso de MongoDB

## do básico ao avançado

# Relacional x Não relacional

- Os bancos **relacionais** demandam uma forte configuração de: tabelas, colunas e relações entre tabelas para o seu funcionamento;
- Os **não relacionais** não são rigorosos quanto a isso, podemos criar colunas quando um dado é inserido;
- O que gera **flexibilidade** para o não relacional e também pode ser sinônimo de **desorganização**;
- Apesar do nome, o não relacional **pode ter relações entre collections**;

# E o MongoDB?

- É o banco não relacional **mais utilizado**;
- Os dados são inseridos em formato de objeto (**JSON**);
- Os comandos, em vez de queries, são **métodos**;
- Podemos criar **relações entre entidades**;
- Facilmente adaptado para diversas linguagens através de **drivers**;
- Possui uma **proximidade com JavaScript**;

# Instalação MongoDB Windows

- Vamos precisar instalar duas ferramentas no Windows;
- **MongoDB**, que é o sistema do banco de dados;
- **MongoDB tools**, ferramenta para importação e exportação de bancos;
- As duas possuem **instaladores** e estes estão disponíveis na documentação oficial;
- Vamos lá!

# Instalação do CMDER

- O **cmder** é um simulador de terminal para Windows, que deixa o terminal bem parecido com o do Linux, o que é interessante para o curso;
- Este software será o escolhido para as aulas, **então é sugerido fazer a instalação**;
- Lembrando que **a maioria dos servidores de aplicações é Linux**, então isso é vantajoso para você também;

# Variáveis PATH

- Para poder **executar o MongoDB no terminal**, precisamos inserir variáveis PATH;
- Vamos **colocar os binários tanto do MongoDB como do tools**, para ter acesso as duas ferramentas;
- Vamos lá!

# Instalação MongoDB Linux

- Para **instalar no Linux** vamos seguir a documentação oficial;
- Lá os passos são simples e resumidos;
- Além de instalar é preciso sempre iniciar o serviço do MongoDB antes das aulas;
- Que pode ser feito via o comando: **mongod**

# Instalação do VS Code

- O **VS Code** será o editor de código oficial do curso;
- Ele também é o **mais utilizado atualmente**, pela maioria dos desenvolvedores;
- Sua instalação é super simples e **compatível com diversos SO's**;
- Vamos lá!

# Principais entidades

- **Database**: é onde ficam as nossas collections e dados;
- **Collections**: são como as tabelas nos bancos relacionais, nelas vamos inserir os dados;
- **Documents**: são os dados, no MongoDB tem esta nomenclatura;
- **Collections podem ser criadas livremente** a qualquer momento e **não possuem colunas fixas** para os dados;

# MongoDB e JSON

- O tipo de dado inserido na tabela é o **BSON**, uma variação de JSON;
- O BSON é semelhante ao JSON, porém com **recursos a mais**;
- A **forma de criar um BSON é igual ao JSON**, veja:

```
{
    nome: “Matheus”,
    idade: 30
}
```

# Primeiro mergulho

- Vamos agora dar o nosso **primeiro mergulho no MongoDB**;
- Nossa missão será **criar um banco de dados**;
- Criar uma **collection**;
- **Inserir um dado** e resgatar este mesmo dado por meio de um **find**;
- **Obs**: Aprenderemos todos estes comandos em detalhes posteriormente!
- Vamos lá!

# Exercício 1

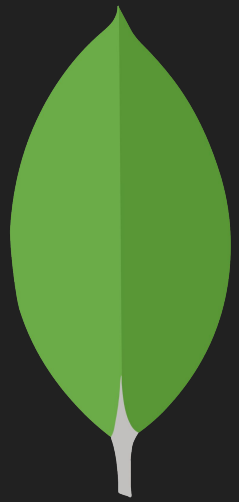
- Insira um dado na nossa collection;
- Faça a seleção de dados para ver o dado inserido (comando **find**);

# MongoDB e os drivers

- Quando utilizarmos MongoDB e alguma linguagem, **precisamos utilizar o driver da mesma**;
- Todos estão disponíveis na **documentação oficial**;
- Os comandos do **shell** são os mesmos que o **driver de JavaScript (Node JS)**, o que facilita muito para as aplicações **MERN / MEAN / MEVN**;
- **Ps:** focaremos no shell!
- Vamos explorar os drivers!

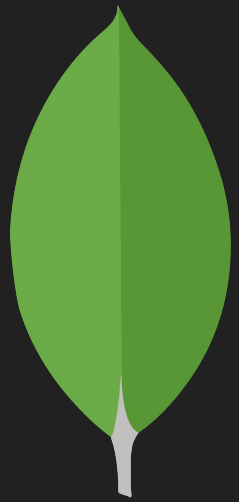
# Como tirar o máximo proveito

- Faça todos os **exercícios**;
- Faça **anotações** dos conceitos aprendidos;
- **Aplique os exemplos** em outras situações;
- Após o curso crie seus próprios **projetos**;
- **Extra:** assistir e depois agir;



# Curso de MongoDB

## Conclusão da seção



# Gerenciamento de DBs

## Introdução da seção

# Verificar todos os bancos

- Podemos verificar os bancos do sistema com: **show dbs**
- Este comando **mostra todos os DBs criados até o momento**;
- Note que há alguns bancos do próprio Mongo já criados;
- **Obs:** faça o download dos arquivos do curso!

# Criando banco de dados

- Para criar um banco utilizamos a instrução: **use <nome>**
- Isso fará com que um banco seja inicializado, porém ele só será registrado de fato quando houver algum dado nele;
- Podemos checar o banco atual com o comando: **db**
- O comando use também serve para **mudar de banco**;

# Exercício 2

- Crie um banco de dados;
- Utilize o comando de verificação de bancos para exibir todos eles;

# Criando collections

- **Não precisamos explicitamente criar uma collection**, basta inserir um dado em alguma;
- Com o comando: **db.<collection>.insertOne(<dados>)**
- A collection será criada automaticamente, e também o banco de dados persistirá no sistema;
- Note que a instrução db vai referir sempre ao banco atual!

# Encontrando dados

- Para buscar dados utilizamos o comando: **find**
- Este comando recebe um filtro, para selecionarmos dados específicos;
- Exemplo: **db.<collection>.find({nome: “João”})**
- Neste caso **buscamos por um document** com uma chave nome e um valor João;

# A função pretty

- A função **pretty** pode ser adicionada a alguns comandos;
- O resultado é um **retorno de dados melhor formatado**;
- Desta forma fica mais legível e conseguimos entender melhor o que retornou;
- Ela é muito **utilizada com find**;

# Criação de collection implícita

- Há a possibilidade de criar a collection com um comando também;
- Exemplo: **db.createCollection(“nome”, { opções })**
- Podemos definir alguns **parâmetros de configuração** como: número máximo de registros, tamanho máximo da collection e etc;

# Exibir todas as collections

- Para exibir todas as collections utilizamos: **show collections**
- Este comando de verificação nos ajuda a **entender melhor o banco de dados**;
- Lembrando que para uma collection ser criada de fato, ela precisa ter algum dado inserido!

# Exercício 3

- Crie uma collection com dados de nome de pessoas e salários;
- Utilize o comando find para verificar os registros dela;
- Verifique todas as collections do banco;

# A Chave _id

- Todo registro inserido no banco vem com uma chave chamada **_id**;
- Esta chave tem como objetivo criar um **identificador único para todo registro**;
- Ele consegue ser único pois é baseado no tempo em que é criado, mesmo que os dados sejam inserido simultaneamente, **ids serão distintos**!
- Outra funcionalidade interessante é que ele possui um **índice**, agilizando consultas por esta chave;

# Removendo collections

- Podemos remover collections quando elas não forem mais necessárias ou se errarmos o nome, por exemplo;
- O comando é: **db.<collection>.drop()**
- Após a execução **todos os dado serão removidos também**, então tome cuidado!

# Removendo bancos de dados

- Podemos remover os bancos também!
- O comando é: **db.dropDatabase()**
- Após a execução do comando, **todos os dados e collections serão excluídos** do sistema;

# Exercício 4

- Remova primeiramente a collection de salarios;
- E depois remova o banco em que ela se encontra;

# Importar bancos em JSON

- Um formato muito encontrado de bancos de MongoDB é **.json**;
- Vamos utilizar uma funcionalidade do tools para realizar a importação;
- O comando é: **mongoimport <arquivo> -d <database> -c <collection>**
- Desta maneira **criamos um banco de dados** nomeado no comando e também uma **collection**;
- E é claro, os dados importados!

# Exportar bancos em JSON

- Outra ação comum é **exportar bancos de dados**;
- Para esta ação utilizamos: **mongoexport -c <collection> -d <database> -o <output>**
- Onde definimos qual collection estamos exportando, qual banco e qual o arquivo de saída;

# Exportar muitas collections

- Quando o banco possui mais de uma collection a melhor opção de exportação é o **mongodump**;
- Utilizamos assim: **mongodump -d <banco> -o <diretorio>**
- Onde o **-o** criará uma pasta, com os arquivos de cada collection;

# Importar dados do mongodump

- Para importar os arquivos gerados do mongodump, utilizamos o **mongorestore**;
- O comando é o seguinte: **mongorestore <diretorio>**
- Não precisamos informar uma flag para o diretório, apenas o caminho relativo a ele

# Status do MongoDB

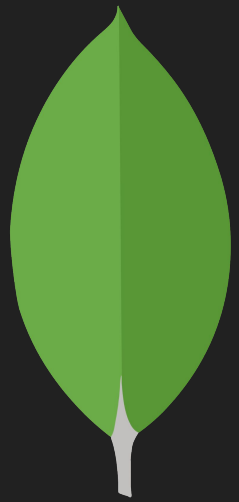
- Podemos checar algumas informações como: **quantidade de consultas rodando, consumo de rede e outros dados**;
- O comando é **mongostat**
- Teremos uma aba do terminal ocupada, atualizando todo o segundo trazendo informações em tempo real;

# Forma simples de remover bancos

- Podemos criar um **loop no nosso terminal** que vai remover todos os bancos não necessários;
- Ou seja, precisamos preservar: **admin, config e local**;
- Os outros podem ser removidos da nossa base;
- Vamos criar o snippet!

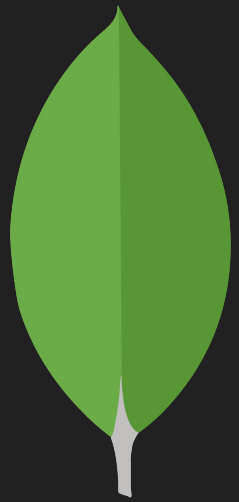
# Tarefa #01

1. Crie um banco de dados;
2. Crie apenas uma collection e insira três dados;
3. Utilize um find para exibir os dados;
4. Exporte o banco criado;
5. Importe o banco, pelo dump, em um outro novo banco;



# Gerenciamento de DBs

## Conclusão da seção



# Inserção de dados (Create)

## Introdução da seção

# O que é CRUD?

- **CRUD** é uma sigla para **C**reate, **R**ead, **U**pdate e **D**elete;
- É onde inserimos dados, resgatamos/lemos dados, atualizamos dados e deletamos/removemos dados;
- As **quatro operações básicas de banco de dados**, tanto relacionais quanto não relacionais;
- **Praticamente toda aplicação** terá um CRUD;
- Por isso é importantíssimo o aprendizado deste assunto;

# Tudo é documento

- Sempre que vamos trabalhar com MongoDB é comum adicionarmos várias entidades com chaves **{ }**
- Sempre que adicionamos chave a algum local, chamamos de **document** (documento)
- Ou seja, é bem comum ouvir: **inserir um document na collection**;
- Onde em SQL seria inserir o dado na tabela;

# Inserindo dados

- Para inserir um document utilizamos o método **insertOne**
- Desta maneira: **db.<collection>.insertOne({<dados>})**
- Onde **collection** é o nome da collection que vamos inserir dados;
- E dados representa **o conjunto de chaves e valores** do dado em questão;

# Exercício 5

- Crie uma collection chamada provas
- Crie 2 dados com nome do aluno e um array chamado notas, com notas de provas;
- Exiba todos os dados;

# Não há relação entre dados

- Em uma collection **não precisamos respeitar as chaves dos outros documents**;
- Ou seja, em um banco relacional precisamos adicionar dados das colunas e **em MongoDB não existe essa regra**;
- Então **podemos ter documents totalmente diferentes** em uma collection;
- Vamos ver isso na prática!

# Inserindo vários dados

- Podemos também inserir vários dados de uma vez só, com o método **insertMany**
- A sintaxe é a seguinte: **db.<collection>.insertMany([<dados…>])**
- Note que vamos precisar de um **array**, que vamos inserir os documents, separados por vírgula;

# Exercício 6

- Crie uma collection chamada mercado;
- Com o comando de inserção de dados múltiplos, insira produtos com nome, preço e disponibilidade;

# O método insert

- Existe um método chamado **insert**, que serve para inserir um ou mais dados;
- Exemplo: db.<collection>.insert()
- Porém ele é mais antigo, **e os métodos mais atuais são insertOne e insertMany**;
- Então é interessante utilizá-los em nossas aplicações, em vez de insert;
- Vamos ver o insert na prática!

# Voltando ao _id

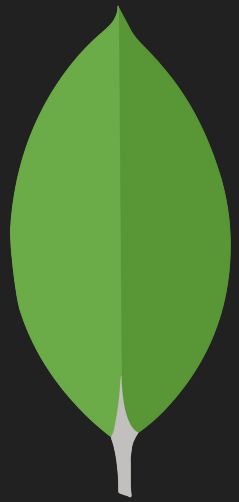
- Já sabemos que o **_id é único** e criado em todos os documents da collection;
- Porém não necessariamente precisamos deixar a cargo do MongoDB isso, **podemos criar o nosso próprio id**;
- Exemplo: **db.<collection>.insertOne({_id: “meuid”, nome: “Matheus”})**
- Desta forma é possível personalizar este campo!

# Write Concern

- O **Write Concern** é uma configuração que pode ser inserida no insertMany;
- Podemos limitar o tempo de execução da inserção;
- Retornando um **erro de time out** caso exceda o mesmo;
- Exemplo: **{w: “majority”, wtimeout: 100}**
- A inserção tem 100ms para ser executada;

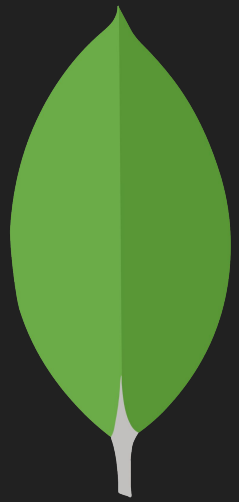
# Tarefa #02

1. Crie um banco de dados chamado dadosDeCarros;
2. Crie uma collection chamada carros;
3. O id deve ser personalizado!
4. Insira 4 carros com os seguintes dados: marca, modelo, ano fabricação, kilometragem rodada
5. Visualize todos os dados, com pretty;



# Inserção de dados

## Conclusão da seção



# Seleção de dados (Read)

## Introdução da seção

# Banco dessa seção

- Nesta seção vamos trabalhar com o banco **books.json**
- Ele se encontra nos arquivos do curso!
- **Vamos importar ele** primeiramente para depois criar nossas seleções de dados!

# Encontrar todos os dados

- Para encontrar todos os dados vamos utilizar o comando find;
- Exemplo: **db.books.find({})**
- O **document vazio neste método é opcional**, sem ele todos os dados serão retornados também;
- Podemos utilizar o **pretty** com o comando de find!

# Mais sobre o pretty

- O método pretty não nos retorna todos os dados, e sim um **cursor**;
- Por isso precisamos digitar **it** para receber mais registros;
- É uma forma de retornar menos dados, uma espécie de paginação;
- O **cursor também um objeto**, que possui outros métodos em MongoDB;
- Geralmente modificando a forma que os dados são retornados;

# Encontrar dado com valor específico

- Para encontrar um dado específico podemos **definir um document dentro do find**;
- O primeiro argumento da opção também é chamado de **filtro**;
- Exemplo: **db.books.find({ pageCount: 362 })**
- Todos os livros com 362 páginas serão retornados!

# Exercício 7

- Selecione o livro com o título de: MongoDB in Action
- Selecione os livros do autor: Jason R. Weiss

# Encontrar dado entre valores

- Para esta função vamos utilizar o operador **$in**;
- Teremos uma seção exclusiva para operadores posteriormente!
- Exemplo: **db.books.find({ catogories: { $in: [“Java”, “Internet”] } })**
- Precisamos criar uma **lista de valores** que queremos buscar;
- Todos estes registros que contiverem um destes valores será retornado;

# Múltiplas condições

- Dados podem ser encontrados baseados em múltiplas condições;
- Basta **adicionar uma vírgula no document** e inserir o próximo requisito;
- Exemplo: **db.books.find({ pageCount: 592, _id: 63 }).pretty()**
- Neste caso acima buscamos por um livro com 592 páginas e que tenha o id igual a 63;
- **Obs:** esta consulta também é semelhante ao operador AND em SQL;

# Maior que algum valor

- Outro operador interessante é o que vai buscar valores maiores que um determinado, o operador é o **$gt (greater than)**;
- Exemplo: **db.books.find({ pageCount: { $gt: 450 }}).pretty()**
- Neste exemplo buscamos livros que tem mais que 450 páginas;
- Note que inserimos um **novo document** para determinar o $gt;

# Exercício 8

- Selecione os livros que tem mais de 800 páginas;
- Selecione os livros que tem o id maior que 122;

# Menor que algum valor

- Assim como o maior que, temos o menor que, que busca valores menores que o determinado;
- O operador é o **$lt (less than);**
- Exemplo: **db.books.find({ pageCount: { $lt: 120 }}).pretty()**
- Aqui buscamos livros com menos que 120 páginas, **note que a aplicação é a mesma que $gt**;

# Operador $or

- O operador **$or** é utilizado para resgatar dados que possuem um valor ou outro;
- Exemplo: **db.books.find({ $or: [{pageCount: {$gt: 500}, _id: {$lt: 5}}]}).pretty()**
- Neste caso buscamos por livros com mais de 500 páginas e com id menor que 5;
- Note que **combinamos vários operadores**;

# Operador and e or na mesma consulta

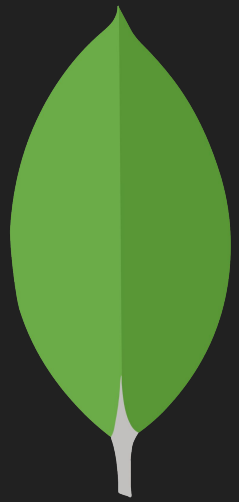
- Como visto anteriormente **podemos unir vários operadores**;
- Exemplo: **db.books.find({ status: "PUBLISH", $or: [{pageCount: 500}, {authors: "Robi Sen"}] }).pretty()**
- Aqui buscamos livros publicados e que tenham 500 páginas ou o autor seja Robi Sen;
- Perceba que podemos adicionar filtros bem específicos as consultas!

# Contando número de resultados

- Para contar o número de retornos de uma consulta, podemos utilizar o método **count**;
- Exemplo: **db.books.find({ pageCount: {$gt: 600 }}).count()**
- Aqui temos quantos livros existem acima de 600 páginas no nosso banco;

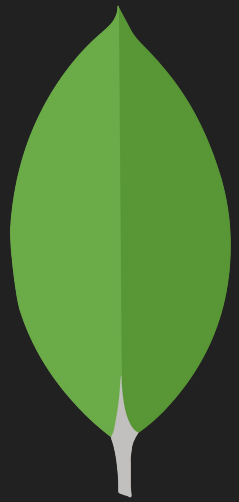
# Tarefa #03

1. Selecione os livros da categoria Java;
2. Selecione os livros com menos de 100 páginas;
3. Selecione os livros da categoria Microsoft com mais de 300 páginas;
4. Conte quantos livros estão na categoria Web Development;
5. Utilize o operador $or para resgatar livros de Bret Updegraff ou que são da categoria mobile;



# Seleção de dados (Read)

## Conclusão da seção



# Atualização de dados (Update)

## Introdução da seção

# Atualizando um dado

- Para atualizar um dado utilizamos o método updateOne;
- Primeiro realizamos o filtro e depois inserimos o que precisa ser atualizado;
- Exemplo: **db.books.updateOne({_id: 314}, { $set: { pageCount: 1000 }})**
- Aqui atualizamos as páginas do livro com id 314 para 1000;
- O operador **$set** é onde ficam os valores a serem atualizados;

# Exercício 9

- Altere o título do livro com id 20, para “Meu primeiro update”;
- Encontre o registro e veja se foi corretamente modificado;

# Atualizando vários itens

- Para atualizar diversos itens utilizamos **updateMany**;
- Este método tem a mesma lógica de execução de que updateOne;
- Exemplo: **db.books.updateMany({categories: “Java”}, {$set: {status: “UNPUBLISHED”}})**
- Neste update atualizamos todos os dados da categoria Java, alteramos o status destes registros;

# Adicionando dados com update

- O update pode servir para adicionar um dado ao document;
- Basta **inserir um valor para uma chave** que não existe no mesmo;
- Exemplo: **db.books.updateMany({authors: "Vikram Goyal"}, { $set: {downloads: 1000} })**
- Aqui adicionamos a chave downloads com o valor de 1000 a todos os livros de Vikram;

# Exercício 10

- Adicione a todos os livros com mais de 500 páginas o dado:
- bestseller: true;
- Depois faça uma seleção para verificar se houve a modificação;

# Trocando todo o documento

- Podemos trocar todos os dados do documento com o **replaceOne**;
- Ou seja, haverá uma substituição de dados;
- Exemplo: **db.books.replaceOne({_id: 120}, {foi: "substituido"})**
- Neste caso trocamos todos os dados do registro com id 120 para o document do segundo argumento;

# Adicionar item a um array

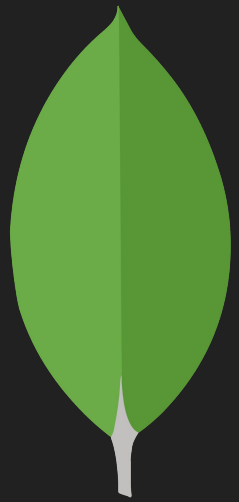
- Se tentarmos atualizar um array diretamente vamos substituir ele;
- Para adicionar um item vamos precisar do operador **$push**;
- Exemplo: **db.books.updateOne({_id: 201}, { $push: {categories: "PHP"}})**
- Neste caso adicionamos a categoria PHP ao livro com id 201;

# Atualizar todos os itens

- Para atualizar todos os itens podemos utilizar o **updateMany**;
- Porém **não colocamos filtro nenhum**;
- Exemplo: **db.books.updateMany({}, { $set: {atualizado: true} })**
- Assim todos os itens serão atualizados!

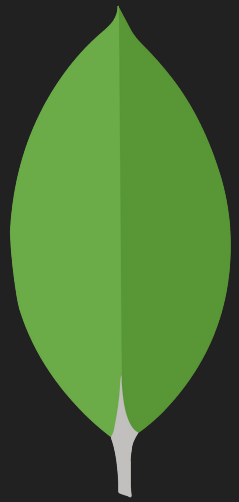
# Tarefa #04

1. Após qualquer atualização, faça uma seleção de dados para verificar se o registro está correto;
2. Atualize o status do livro Flex 4 in Action para OUT OF STOCK;
3. Atualize todos os livros que tem menos de 100 páginas com a chave short_book e o valor true;
4. Adicione a categoria Many Pages aos livros da categoria Java com mais de 500 páginas;



# Atualização de dados (Update)

## Conclusão da seção



# Deletando de dados (Delete)

## Introdução da seção

# Remover um item

- A remoção de itens é bem **parecida com a atualização**;
- Baseado em um **filtro**, podemos deletar um elemento;
- Exemplo: **db.books.deleteOne({_id: 20})**
- Neste caso deletamos o item de id igual a 20;

# Remover mais de um item

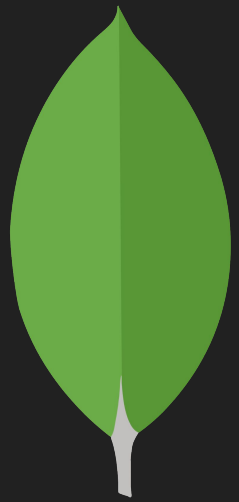
- Para remover mais de um item utilizamos **deleteMany**;
- Exemplo: **db.books.deleteMany({categories: “Java”})**
- Neste caso, removemos todos os livros da categoria Java;

# Exercício 11

- Remova o livro com o isbn igual a 1933988320
- Remova todos os livros que pertecam a categria PowerBuilder;
- Faça um seleção de dados para verificar se estes registros foram removido;

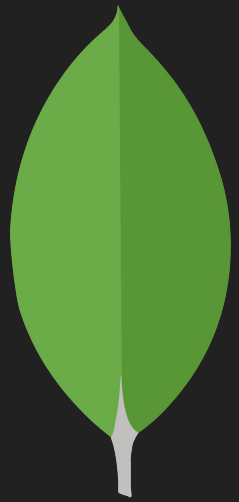
# Remover todos os itens

- Para remover todos os itens basta utilizar o **deleteMany**;
- Porém passando um **filtro vazio**;
- Exemplo: **db.books.deleteMany({})**



# Deletando de dados (Delete)

## Conclusão da seção



# Tipos de dados

## Introdução da seção

# O que são os tipos de dados?

- **Categorias** para os dados inseridos no sistema;
- Escolher os melhores tipos de dados nos ajuda a ter um **projeto melhor estruturado**;
- **O MongoDB tem alguns tipos diferentes do SQL**, por isso a importância de conhecer os mais relevantes;
- **Alguns deles são:** strings, numbers, boolean, array e etc;

# Textos

- Os textos, mais conhecidos como **strings**;
- São valores que inserimos entre aspas nos nossos documents;
- Exemplo: **db.strings.insertOne({nome: "Matheus"})**
- As strings podem ser inseridas em **aspas simples ou duplas**, não há diferenciação;

# Verificando tipo de dado

- Podemos verificar o **tipo de dado** de uma forma simples no MongoDB;
- Utilizamos o **findOne** em algum registro e **encapsulamos em uma variável**;
- Depois é só utilizar o operador **typeof** com o dado que queremos verificar;
- Exemplo: **typeof variavel.nome**
- Receberemos um **retorno** do tipo de dado;
- Obs: alguns dados são considerados objetos!

# Arrays

- Os arrays, também conhecidos como **listas**, são utilizados para inserir vários itens em um campo;
- Basta adicionar os valores entre **[ ]**, e **separados por vírgula**;
- Exemplo: **db.arrays.insertOne({carros: ["BMW", "Ferrari", "Fusca"]})**
- Os valores podem ser em **qualquer tipo de dado**;
- Mas geralmente um array tem um único tipo;

# Datas

- As datas em Mongo são salvas no formato **ISO**;
- Podemos criar uma nova data com **new Date()**;
- Exemplo: **db.dates.insertOne({data: new Date()})**
- A ação acima salva corretamente uma data em um document;

# Document

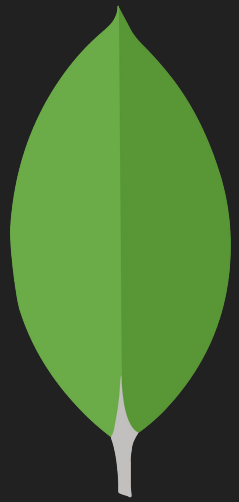
- O document é parecido com o **objeto de JavaScript**;
- Guarda dados com **chaves e valores**;
- Exemplo: **db.documents.insertOne({nome: "Matheus", desc: {profissao: "Programador", hobbies: ["Estudar", "Ler", "Caminhar"]}})**
- Aqui criamos um document com uma string e um array;
- Os itens também são **separados por vírgula**;

# Booleano

- O booleano é um dado que só aceita dois valores: **true e false**;
- Exemplo: **db.bools.insertOne({nome: "Matheus", trabalhando: true})**
- Neste caso conseguimos definir se uma pessoa do banco de dados está trabalhando ou não;

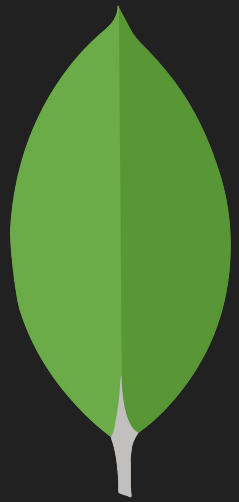
# Números

- Todos os números para o MongoDB são classificados como **doubles**;
- A não ser que explicitamente declararmos eles como inteiros;
- Exemplo: **db.numbers.insertOne({double: 12.2, outro_double: 50, inteiro: NumberInt("5")})**
- Neste caso, os dois primeiros são categorizados como decimais e o terceiro como inteiro;



# Tipos de dados

## Conclusão da seção



# Operadores de query (Select)

## Introdução da seção

# O que são operadores de query?

- São funcionalidades do MongoDB para deixar nossas **queries mais precisas**;
- Os operadores são **divididos em tipos**, como: de comparação e lógicos;
- Veremos os mais importantes nesta seção;
- A sintaxe deles é: **$nome**;
- Onde geralmente definimos um document, especificando o que queremos filtrar;

# Exercício 12

- Nesta seção vamos utilizar um **banco diferente**;
- O banco está na pasta da seção 8 e se chama restaurant.json;
- Crie o banco e a collection para este arquivo;

# Operador $eq

- O operador **$eq** verifica se um registro é igual ao que estamos especificando no document;
- Exemplo: **db.restaurants.findOne({ rating: {$eq: 5} })**
- Neste caso buscamos um restaurante com nota igual a 5;
- Note que este operador é **facilmente substituido pelo filtro normal**, sem operador;

# Operador maior e maior ou igual

- Os operadores **$gt** e **$gte** verificam se um dado é maior e maior ou igual a algum valor específico;
- Exemplo: **db.restaurants.findOne({ rating: {$gte: 4} })**
- Aqui buscamos por restaurantes de nota 4 ou maior;
- Se trocássemos para $gt, buscariamos apenas por notas maiores que 4;

# Exercício 13

- Selecione restaurantes que tem nota maior ou igual a 3;
- E também que o tipo de comida é Breakfast;

# Operador menor e menor ou igual

- Os operadores **$lt** e **$lte** verificam se um dado é menor e menor ou igual a algum valor específico;
- Exemplo: **db.restaurants.findOne({ rating: {$lt: 2} })**
- Aqui buscamos por restaurantes de nota menor que 2;
- Se trocássemos para $lte, buscariamos por notas 2 e menores;

# Operador $in

- Os operador **$in** verifica registros que se encaixam em apenas um dos passados como lista na consulta;
- Exemplo: **db.find({type_of_food: {$in: ["Pizza", "Chinese"]}})**
- Neste caso, procuramos por restaurantes que servem pizza ou comida chinesa;

# Operador $ne

- Os operador **$ne** (not equal) trás resultados que não são iguais ao informado, é o inverso de $eq;
- Exemplo: **db.restaurants.findOne({ rating: {$ne: 5} })**
- Neste caso retornamos o primeiro restaurante que não é nota 5;

# Operador $exists

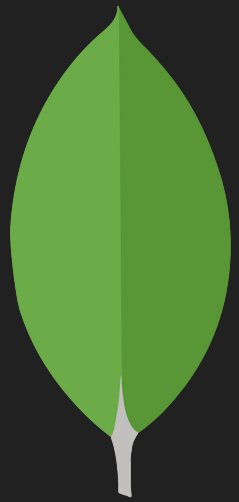
- O **$exists** retorna apenas os dados que possuem determinado campo;
- Exemplo: **db.restaurants.find({high_score: {$exists: true}})**
- Neste caso retornamos só os registros que possuem high_score;

# Exercício 14

- Adicione um campo que qualifica os restaurantes ruins, ou seja que tem nota menor ou igual 2;
- Depois faça uma seleção dos mesmos com exists, baseado neste novo campo;

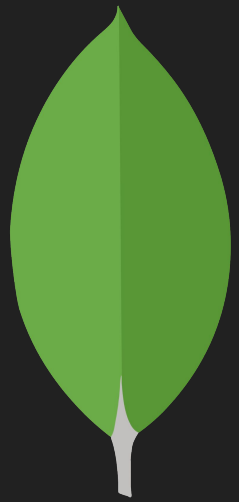
# Operador $text

- O **$text** faz uma busca sobre o texto do campo que foi informado no filtro;
- Exemplo: **db.restaurants.find({$text: {$search: "pizza"}}).pretty()**
- Porém **é preciso criar um índice** em algum dos campos, se não ele não funciona;
- **Obs:** Teremos uma seção exclusiva de index!



# Operadores de query (Select)

## Conclusão da seção



# Relacionamentos

## Introdução da seção

# O que são relacionamentos?

- São **registros que possuem ligações entre si**;
- Tipos de relação: **one to one, one to many e many to many**;
- Onde cada uma possui um método diferente de ser aplicado em MongoDB;
- Além de uma forma especial, graças a flexibilidade dos documents, que é a **embedded document**;
- Vamos ver cada uma delas!

# Embedded documents

- **Embedded documents** é uma forma simples de fazer relacionamento entre documents;
- A ideia é inserir um document dentro do registro principal;
- Este recurso **funciona bem para One to One e One to Many**, porém não para Many to Many;
- Vamos ver na prática!

# One to One

- A relação One to One é quando **um registro possui uma ligação única com outro**, e o inverso também é verdadeiro;
- **Exemplo:** Nosso sistema permite o cadastro de um único endereço por usuário, então podemos dizer que o endereço é único para cada usuário;
- E agora vamos trabalhar com **duas collections**;
- Precisamos inserir uma informação que faça referência ao registro, como o **id**, vamos ver na prática!

# One to Many

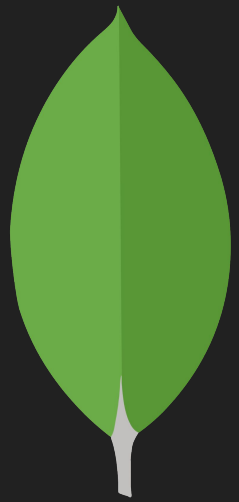
- A relação **One to Many** é quando um **registro pode possuir mais vínculos com uma outra collection**, porém o inverso é falso;
- **Exemplo:** Um usuário pode fazer várias compras, mas uma compra pertence a apenas um usuário;
- Desta maneira a collection de compras contém em cada compra uma referência ao usuário;
- Que será o **_id;**

# Many to Many

- A relação **Many to Many** acontece quando os registros das duas collections possuem mais de uma relação entre si;
- **Exemplo:** Temos alunos e cursos, um curso pode ter vários alunos matriculados e um aluno pode estar fazendo vários cursos;
- Normalmente se cria uma **estrutura intermediária**, ou seja, uma collection;
- Esta collection **contém apenas os ids** de cursos e alunos;

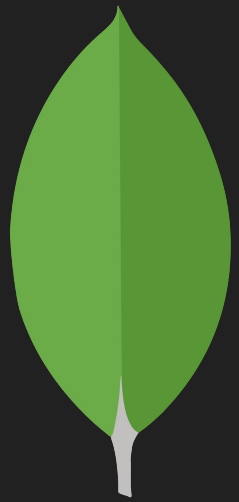
# Por que criar novas collections?

- Uma dúvida que pode ter ficado é: **por que não fazer tudo com embedded documents?**
- Pois há um **limite de 16mb por document**;
- Ou seja, em projetos grandes isso se tornará um problema;
- Então **subdividir em collections trará mais benefícios a longo prazo**;
- Apesar de lidar com as consultas ser mais trabalhoso;



# Relacionamentos

## Conclusão da seção



# Seleção de arrays e documents

## Introdução da seção

# Seleção em embedded documents

- Para resgatar um dado que está em um **document em um outro document**, vamos precisar de uma sintaxe diferente:
- Exemplo: **find({ “chave1.chave2”: “valor” })**
- Precisamos **colocar as duas chaves entre aspas** e depois seguir com o valor, como é comum na busca por chaves;
- Vamos ver na prática!

# Seleção em embedded com operador

- **A lógica para utilizar operadores é a mesma**, colocar as chaves entre aspas;
- Exemplo: **find({ “chave1.chave2”: { $gt: 20 })**
- Note que esta abordagem é a mesma de acessar propriedades em objeto JS;
- Vamos ver na prática!

# Exercício 15

- Selecione pessoas por dois campos: peso e idade, que ficam em características;
- Em peso utilize o operador $in;
- Em idade utilize o operador $gt;

# Encontrar item específico de array

- Para encontrar item específico em array podemos utilizar o **valor final**;
- Exemplo: **db.alunos.find({ notas: 8 })**
- Todos os alunos com nota 8 serão retornados;
- Para valores exatos, precisamos colocar o **array inteiro**:
- Exemplo: **db.alunos.find({notas: [10, 8, 6, 5]})**
- Somente alunos que tiraram as quatro notas acima serão retornados;

# Alguns valores do array

- Para encontrar elementos que contenham apenas alguns valores, utilizaremos o **$all**;
- Exemplo: **db.alunos.find({matematica: {$all: [8, 7]}})**
- Neste caso, retornamos todos os alunos que tiraram 8 e 7 em duas de algumas provas;

# Consulta pelo tamanho do array

- Para consultar um array pelo tamanho, utilizamos o **$size**;
- Exemplo: **db.alunos.find({matematica: {$size: 4}})**
- Aqui todos os arrays de quatro elementos serão retornados da consulta;

# Seleção de array de documentos

- Para fazer uma seleção específica é necessário colocar **todas as características** do elemento em find;
- Exemplo: **db.produtos.find({"variacoes": { cor: "Verde", tamanho: "G", qtd: 48 }})**
- Neste caso, apenas o item que corresponder a todos os critérios será retornado;

# Array de documentos e operadores

- Para utilizar operadores vamos precisar colocar a **propriedade entre aspas** e em seguida o operador;
- Exemplo: **db.produtos.find({ "variacoes.qtd": { $gt: 30 }})**
- Neste caso, apenas produtos com mais de 30 unidades serão retornados;

# Utilizando o $elemMatch

- Para trazer registros por múltiplos critérios utilizamos o operador **$elemMatch**;
- Exemplo: **db.produtos.find({"variacoes": { $elemMatch: {tamanho: { $gt: 40 }, cor: "Azul"}}}).pretty()**
- Neste caso, recebemos resultados que possuem variações com tamanho maior que 40 e também cor Azul;

# Exercício 16

- Busque pelas roupas com tamanho menor que 48;
- E também que tenham mais ou 30 itens em estoque;

# Retornando campos específicos

- Nem sempre precisamos do retorno de todos os campos dos documents;
- Então podemos **escolher apenas os que são necessários**;
- Exemplo: **db.pessoas.find({}, {nome: 1, idade: 1})**
- Neste caso retornamos nome, idade e o _id (que sempre é retornado por default);

# Removendo o id do retorno

- Para remover o _id do retorno de dados específicos, **precisamos deixar isso explícito**;
- Exemplo: **db.pessoas.find({}, {_id: 0, nome: 1, idade: 1})**
- Colocamos o id como 0, agora ele não é mais retornado;
- Teremos apenas o nome e a idade, nesta consulta;

# Excluir campos

- Em algumas ocasiões será mais interessante **excluir campos**, do que adicionar todos os necessários;
- Exemplo: **db.pessoas.find({}, {altura: 0, cor_dos_olhos: 0})**
- Neste caso removemos dois campos, todos os outros serão retornados;

# Exercício 17

- Traga a seleção de todas pessoas;
- Remova o _id e o nome dos dados que retornam;

# Retornar campos de embedded

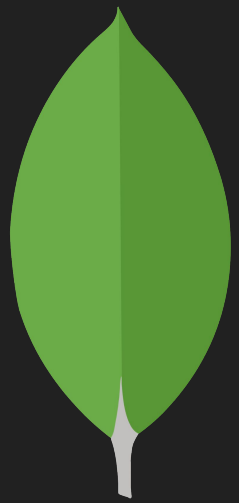
- Para retornar campos específicos de embedded documentos **precisamos selecionar a propriedade entre aspas**;
- Exemplo: **db.pessoas.find({}, {nome: 1, "caracteristicas.peso": 1})**
- Neste caso, apenas o peso é retornado com o documento de características;

# Suprimir campos de embedded

- Para suprimir campos específicos de embedded, vamos **selecionar o campo entre aspas e colocar o valor 0;**
- Exemplo: **db.pessoas.find({}, {"caracteristicas.cor_dos_olhos": 0})**
- Neste caso suprimimos a cor_dos_olhos do document de caracteristicas;

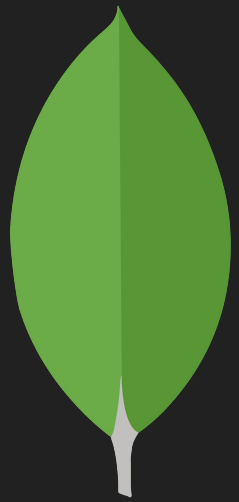
# Tarefa #05

1. Na collection alunos adicione 4 notas de química para todos os alunos;
2. Adicione um novo aluno, chamado Josias, este tem apenas duas notas em química;
3. Resgate todos os alunos que tem alguma nota 10 em química;
4. Resgate os alunos que tem apenas 2 notas de química no sistema;
5. Faça uma atualização do aluno com duas notas, e adicione um campo chamado “falta_provas”, como true e retorne todos os dados com pretty;



# Seleção de arrays e documents

## Conclusão da seção



# Operadores de Update

## Introdução da seção

# Operador $inc

- O operador **$inc** pode acrescentar ou diminuir uma quantidade especificada a um valor;
- Exemplo: **db.blog.updateOne({author: "Matheus Battisti"}, {$inc: {postCount: 2}})**
- No exemplo, adicionamos 2 a quantidade de postCount;
- O valor pode ser negativo também!

# Operador $min

- O operador **$min** atualiza um valor, caso o especificado do operador seja menor que o do registro;
- Exemplo: **db.blog.updateOne({ author: "Maicon Santos"}, {$min: { postCount: 0, likesReceived: 0}})**
- Neste caso, zeramos os posts e os likes de Maicon;

# Operador $max

- O operador **$max** faz o inverso de $min, ou seja, atualiza o valor se ele for maior que o do campo;
- Exemplo: **db.blog.updateOne({ author: "Matheus Battisti" }, {$max: {maxPosts: 250}})**
- Neste caso, aumentamos o número de posts permitidos por meio de $max;

# Operador $mul

- O operador **$mul** multiplica o número de alguma propriedade por um outro número definido;
- Exemplo: **db.blog.updateOne({ author: "Matheus Battisti" }, {$mul: {maxPosts: 2}})**
- Aumentamos novamente o número máximo de posts, mas dessa vez pelo operador $mul;

# Operador $rename

- O operador **$rename** renomeia um campo, por outro nome que definimos;
- Exemplo: **db.blog.updateMany({}, {$rename: {author: "author_fullname"}})**
- Neste caso, atualizamos o nome do campo “author” para “author_fullname”

# Operador $unset

- O operador **$unset** tem como objetivo remover um campo de um item;
- Exemplo: **db.blog.updateMany({}, {$unset: {active: ""}})**
- Neste caso, removemos o campo active de todos os registros;

# Operador $addToSet

- O operador **$addToSet** adiciona um ou mais valores em arrays, apenas se eles já não estiverem lá;
- Ou seja, não duplica os elementos;
- Exemplo: **db.blog.updateOne({author_fullname: "Matheus Battisti"}, {$addToSet: {categories: { $each: ["PHP", "Vue"]}}})**
- Neste caso só foi adicionado Vue, pois PHP já existia;

# Operador $pop

- O operador **$pop** remove o último ou o primeiro elemento de um array;
- Para remover o primeiro utilize -1, para remover o último 1;
- Exemplo: **db.blog.updateOne({author_fullname: "Matheus Battisti"}, {$pop: {categories: -1}})**
- Neste caso, removemos o primeiro elemento de categories;

# Operador $push

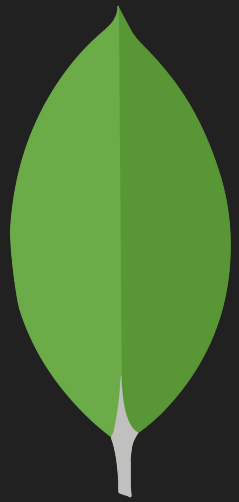
- O operador **$push** adiciona um ou mais valores a um array;
- Exemplo: **db.blog.updateOne({author_fullname: "Matheus Battisti"}, {$push: {categories: "Linux"}})**
- Adiciona o valor Linux ao array categories;

# $push para mais itens

- Para adicionar mais itens com o **$push**, precisamos do **$each**;
- Exemplo: **db.blog.updateOne({author_fullname: "Matheus Battisti"}, {$push: {categories: { $each: ["HTML", "CSS"]}}})**
- Adicionamos HTML e CSS ao array de categories;

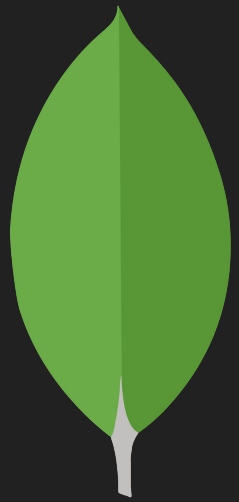
# Operador $pullAll

- Para remover vários itens de um array utilizamos o **$pullAll**;
- Exemplo: **db.blog.updateOne({author_fullname: "Maria Marin"}, {$pullAll: { categories: ["Linux", "Docker"]}})**
- Aqui removemos as categorias Linux e Docker de Maria;



# Operadores de Update

## Conclusão da seção



# Utilizando Indexes

## Introdução da seção

# O que é um índice?

- Os **índices (jndexes)** são recursos que podem aumentar a eficiência de uma query, ou seja, deixá-la **mais rápida**;
- Podemos adicionar índices a um dado em uma collection;
- O dado **_id** já vem com um índice;
- Os **dados com índices são checados primeiro** na hora de uma seleção, por exemplo;

# Importando o banco da seção

- Nesta seção vamos utilizar um banco com **mais dados**;
- O nome é **city_inspections.json** e está nos arquivos do curso;
- Vamos importá-lo!

# Criar um índice

- Para criar um índice vamos utilizar o comando **createIndex**;
- Exemplo: **db.inspections.createIndex({ certificate_number: 1 })**
- Aqui criamos um índice no dado certificate_numer;
- A partir de agora, todas as consultas que utilizem ele, **serão mais rápidas**;

# Criar um índice

- Para criar um índice vamos utilizar o comando **createIndex**;
- Exemplo: **db.inspections.createIndex({ certificate_number: 1 })**
- Aqui criamos um índice no dado certificate_numer;
- A partir de agora, todas as consultas que utilizem ele, **serão mais rápidas**;

# Criar um índice em campo de documento

- Podemos também criar **índices para campo de embedded documents**!
- Exemplo: **db.inspections.createIndex({ "address.city": 1 })**
- Agora o campo cidade de address, tem um índice;
- Consequentemente as consultas que envolvem este campo, serão mais rápidas;

# Verificando index de collections

- Podemos checar quais **índices uma collection possui**;
- Exemplo: **db.inspections.getIndexes()**
- Assim todos os índices criados serão retornados;
- Recebemos informações importantes também, como o campo do índice;

# Listar índices de um banco

- Para listar os índices de um banco temos que realizar uma **operação um pouco mais complexa**;
- Primeiramente vamos fazer um **loop em todas as collections com forEach**;
- E depois vamos utilizar o comando **getIndexes** para resgatar todos, e exibir;
- Vamos ver na prática!

# Removendo índices

- Para remover índices vamos utilizar **dropIndex**;
- Exemplo: **db.inspections.dropIndex({ certificate_number: 1 })**
- Desta maneira teremos a queda na performance deste campo;
- E o index removido da collection;

# Remover todos os índices

- Para remover todos os índices de uma collection utilizamos **dropIndexes**;
- Exemplo: **db.inspections.dropIndexes()**
- Todos os índices da collection serão removidos;
- **Com exceção do _id**, este não é excluído;

# Plano da consulta

- Podemos obter informações interessantes e saber como o MongoDB fez uma consulta com o método **explain()**;
- Exemplo: **db.inspections.find({certificate_number: 3030353}).explain()**
- Aqui vamos entender como ele encontrou o certificado número 3030353;
- Se usou algum índice ou não e etc;

# Índices compostos

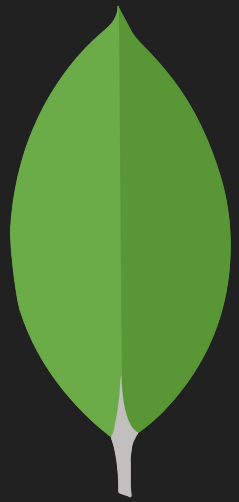
- O MongoDB possui a possibilidade de criar **um índice para múltiplos campos**;
- Exemplo: **db.inspections.createIndex({ certificate_number: 1, date: 1 })**
- Isso favorece as buscas quando os dois são incluídos na consulta, por exemplo;

# Índices de texto

- São índices que facilitam a **busca de texto em um campo**;
- Exemplo: **db.inspections.createIndex({ business_name: "text" })**
- Podemos ter apenas um índice de texto por collection;

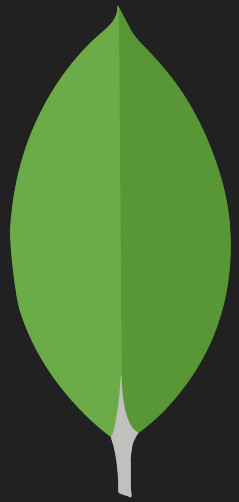
# E por que não criar muitos índices?

- O exagero de índices pode fazer até o **papel contrário**;
- Pois **os índices desnecessários vão ocupar o lugar dos que precisamos**;
- Tornando o efeito nulo;
- Por isso **precisamos planejar bem** em quais campos devemos adicionar os índices;



# Utilizando Indexes

## Conclusão da seção



# Aggregation

## Introdução da seção

# O que é Aggregation?

- Aggregation é um **framework** do MongoDB;
- O seu principal objetivo é **agregar resultados** (aggregate functions do SQL);
- Para retornar resultados que **não temos somente a partir dos dados**;
- Isso nos permite criar relatórios mais completos dos dados do sistema;

# O que é pipeline?

- **Pipeline** é um termo que está ligado ao aggregation;
- É o modo que construímos o resultado da nossa aggregation;
- **Unir diversos métodos** gera uma pipeline;
- Podemos entender pipile line como **segmentação**;
- Conforme as aulas vão seguindo, nossa pipeline se tornará mais complexa;

# Utilizando o $bucket

- O **$bucket** tem como objetivo agrupar resultados;
- **Definiremos como uma grupo deve ser classificado**, baseado em um campo;
- E receberemos uma **contagem de dados neste grupo**, por exemplo;
- Vamos ver na prática;

# Utilizando o $bucketAuto

- O **$bucketAuto** define os buckets de uma forma mais automatizada;
- É uma espécie de **$bucket rápido**;
- Definimos um **campo e quantos buckets queremos receber**;
- E o Mongo se encarrega em dividir os dados;
- Vamos ver na prática!

# Utilizando o $collStatus

- O operador **$collStatus** tem como objetivo retornar dados de uma collection;
- Vamos receber informações como: **banco, collection, horário atual, contagem de registros;**
- Mas podemos resgatar dados mais avançados como: **shards, quantidade de queries executadas e mais;**
- Vamos ver na prática!

# Utilizando o $sort

- Com o operador **$sort** podemos ordenar os resultados;
- Baseando em algum campo (**1 crescente, -1 decrescente**);
- É com este operador que faremos filtros de ordenação de preço, por exemplo;
- Vamos ver na prática!

# Utilizando o $limit

- Com o operador **$limit** podemos limitar o número de resultados retornados;
- Passamos um parâmetro com o **número limite**;
- Este operador permite criar a funcionalidade: número de resultados;
- Vamos ver na prática!

# Utilizando o $match

- Com o **$match** é possível determinar um filtro para os resultados;
- Por exemplo: que sejam do autor Gavin King;
- Então **além dos operadores de agrupamento**, só retornam os livros que são de Gavin;
- Vamos ver na prática!

# Utilizando o $out

- O **$out** nos permite criar uma collection a partir da aggregation;
- Ou seja, os **retornos da agregação serão inseridos em uma nova collection**;
- Tendo assim só os dados filtrados;
- Vamos ver na prática;

# Utilizando o $project

- O **$project** é uma opção para resgatar apenas os campos que precisamos com aggregation;
- Exemplo: **{ $project: { title: 1 } }**
- Neste caso, apenas o título será retornado no aggregation;

# Utilizando o $sample

- O **$sample** retorna uma amostragem aleatória, definida por uma quantidade no operador;
- Exemplo: **{ $sample: { size: 10 } }**
- Neste caso, 10 itens aleatórios da condição imposta serão retornados;

# Utilizando o $skip

- O **$skip** pula um determinado número de dados;
- Exemplo: **{ $skip: 5 }**
- Neste caso, os 5 primeiros dados que viriam nesta aggregation são substituídos pelos 5 próximos;

# Utilizando o $unwind

- O **$unwind** desconstrói um array;
- **Permitindo trabalhar com o resultado de cada item** do array desconstruído;
- Ou seja, cada item do array se torna um item no retorno da query;

# Utilizando o $sortByCount

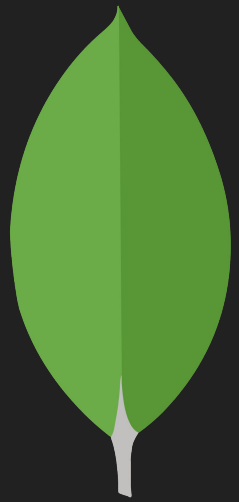
- O **$sortByCount** oderna os resultados por um campo específco;
- Se trouxermos dados a grupos, podemos selecionar pelo **número de ocorrências de cada grupo**;

# Utilizando o $unset

- O **$unset** vai remover um ou mais campos do retorno da nossa agreggation;
- Se for apenas **um campo**, basta inserir o valor como string;
- Se for **vários campos**, inserir um array com o nome dos mesmos;

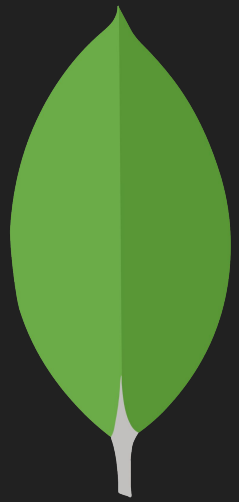
# Count em aggregation

- Para inserir o **count** em uma aggregation, precisamos adicionar mais um passo;
- Ou seja, implementá-lo na nossa **pipeline**;
- A sintaxe é: **{ $count: “Nome” }**
- E a aggregation retorna apenas a contagem de dados;



# Aggregation

## Conclusão da seção



# Mongoose

## Introdução da seção

# O que é Mongoose?

- É um framework de ODM (**Object Data Model**) para MongoDB;
- A ideia principal é criar um **Model** para cada uma das collections;
- E por meio deste definir seus **campos e tipos de dados**;
- Assim teremos um maior controle e organização pela aplicação;
- Baseado nestes models poderemos executar as queries, para realizar um CRUD, por exemplo;
- Inclusive algumas ações são **mais simples pela ODM** do que pelo driver;

# Criando o projeto

- Vamos nesta aula iniciar o projeto para aprender os **conceitos fundamentais de Mongoose**;
- Primeiramente vamos precisar criar com **npm init**;
- E depois instalar dois pacotes necessários: **Mongoose e Nodemon**;
- Por fim, vamos **criar um script** para inicializar o projeto mais rapidamente;
- **Obs:** é necessário que o Mongo esteja ativo na máquina!

# Conexão via Mongoose

- Para criar a conexão vamos precisar **importar o Mongoose** e utilizar o método **connect**;
- Em conect vamos **passar o endereço do servidor**, no nosso caso: localhost;
- **Depois poderemos iniciar a conexão** e testar se a mesma funcionou;
- Vamos lá!

# Criando um Schema

- O **Schema** é a estrutura dos dados que vamos inserir no banco;
- Também **representa a collection**;
- Nesta classe vamos **definir os campos e seus tipos de dados**;
- Posteriormente criaremos um **Model** baseado neste Schema;
- Teremos **um Schema para cada collection**;

# Criando um Model

- Partindo de um Schema, **temos a possibilidade de criar um Model**;
- **Ele será responsável pela execução de outros métodos** do Mongoose, como os do CRUD;
- Após termos o Model, **podemos também estruturar um dado**, instanciando uma nova classe do mesmo;

# Salvando dados

- Para salvar dados podemos utilizar o método **save**;
- Este método representa o **C do CRUD**;
- **Temos a possibilidade de verificar se algum erro ocorreu**, por meio de uma função anônima;
- Podemos verificar os dados sendo inseridos pelo **shell**;

# Encontrando dados

- Para resgatar um dado podemos utilizar **findOne**;
- Assim como no shell, o método **retorna um único valor** e recebe um filtro como parâmetro;
- Podemos imprimir o dado com **console.log**;

# Inserindo vários dados

- Para inserir múltiplos dados podemos utilizar **insertMany**;
- Método **semelhante ao do shell**;
- Enviamos um **array de documents**, que serão inseridos após a execução;
- Vamos ver na prática!

# Deletar dados

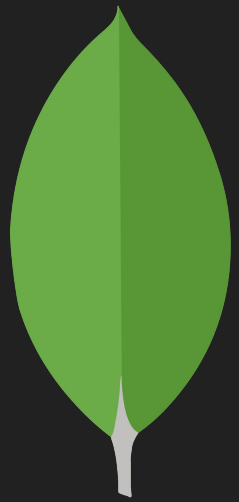
- Para deletar um dado podemos utilizar **deleteOne**;
- Da mesma forma que no shell, passaremos um **filtro** para o método;
- E então o registro encontrado será removido;
- Vamos ver na prática!

# Atualizar dados

- Para atualizar um dado, utilizamos o método **updateOne**;
- Assim como no shell, vamos passar um **filtro** e depois o que será modificado (**$set**);
- Assim o dado é atualizado no banco;
- Vamos ver na prática!

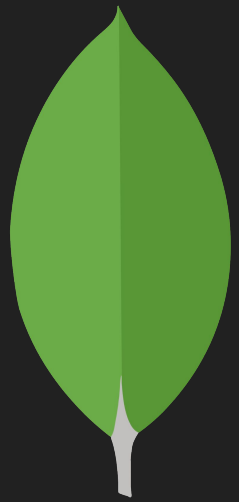
# Utilizando o where

- O **where** é um método do Mongoose que facilita a utilização de múltiplos filtros;
- Podemos **unir vários where's** para deixar a busca mais precisa;
- Assim os dados que estiverem de acordo com o filtro serão retornados;
- Vamos ver na prática!



# Mongoose

## Conclusão da seção



# MongoDB Atlas

## Introdução

# O que é MongoDB Atlas?

- É um **serviço em Cloud** dos criadores de MongoDB para hospedar banco de dados;
- Chamado de **DBasS** (DataBase as a Service);
- Há diversos recursos que deixam a utilização interessante, como a **clusterização**;
- O **preço** acaba sendo bem interessante;
- E também a **otimização para MongoDB**;

# Criando conta no Atlas

- Primeiramente vamos entrar no **site do Atlas**;
- E depois podemos realizar a conta por **auth do Google** ou com email/senha;
- Obs: apesar de ser pago, há um **free tier**;
- Vamos criar a conta e conhecer a **interface** do Atlas!

# Criando projeto no Atlas

- Para começar a utilizar o Atlas vamos precisar criar um **projeto**;
- Vamos selecionar o **free tier**;
- Ele permite que **criar os nossos bancos de dados**, que é o próximo passo;
- Precisamos também **criar acesso a um usuário e ao nosso IP**;
- Na área do projeto é possível **monitorar as ações do banco de dados**;
- Futuramente vamos ver a nossa aplicação funcionando lá! =)

# Clonando o projeto

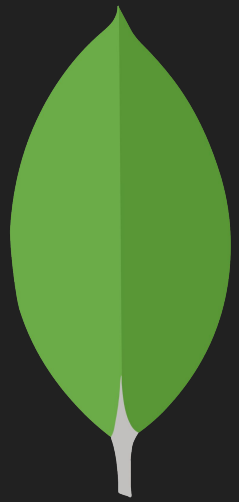
- Para esta seção vamos utilizar nosso projeto de CRUD (**Notes**);
- **Vamos prepará-lo em uma nova pasta** e começar as fazer as **modificações necessárias** para o Atlas;
- Bora lá!

# Utilizando o .env

- Vamos utilizar uma nova extensão para **salvar dados sensíveis**;
- Pois **não é seguro armazenar usuário e senha** na aplicação;
- O nome da extensão é **dotenv**;
- Podemos **adicionar este arquivo no .gitignore** e agora ele não será levado para o repositório;
- **Nossa aplicação fica mais segura**, pois não transmite usuário e senha;
- Vamos ver na prática!

# O último passo

- Agora basta **adicionar o usuário e senha do banco de dados** que vem do Atlas, no .env;
- Em seguida vamos utilizar estes dados **na parte da conexão** em nosso app;
- Ela conectará por aquelas credenciais e **estaremos utilizando o serviço em Cloud**;
- Vamos ver na prática!



# MongoDB Atlas

## Conclusão