WebII

MongoDB: banco de dados baseado em documento

Relacional vs. NoSQL

Talvez um dos mais bem-sucedidos projetos de software em nossa história seja o **banco de dados relacional**. Seus princípios criados em 1970 continuam firmes e presentes em banco de dados comerciais e open sources, constituindo o **paradigma relacional**.

Este tipo de banco possui uma estrutura tabular e relacional baseada fortemente em esquemas e, nele, realizamos consultas e resolvemos seus relacionamentos através de uma linguagem padrão chamada SQL (*StructuredQuery Language*), mais notadamente o padrão ANSI 92 adotado por diversos bancos.

Com o tempo, um novo paradigma apareceu: o NoSQL. Esse termo foi cunhado em 1998 pelo desenvolvedor Carlo Strozzi e mais tarde reforçado

por Eric Evans para indicar bancos que não seguem o padrão SQL. Longe de ser um termo pejorativo, a ideia do NoSQL é a de bancos que não se baseiam em relacionamentos nem em esquemas, tão prezados nos bancos de dados relacionais. Nele, pela ausência de esquemas, a validação e integridade dos dados são de responsabilidade da aplicação.

Uma questão arquitetural?

Cada paradigma apresentado ataca problemas diferentes e sua escolha não é uma decisão puramente preferencial, mas de necessidade do projeto,

tangendo aspectos arquiteturais e mais abstratos. Realizar juízo de valor em cada uma das abordagens sem um contexto propriamente dito provavelmente resultará em debates inflamados. Precisamos de um contexto a partir do qual delimitaremos o que queremos resolver.

Delimitando o contexto: impedância

Tanto um banco de dados relacional quanto um NoSQL demandará do desenvolvedor a tradução dos dados retornados do banco para algo manipulável na linguagem de programação escolhida, em nosso caso, JavaScript.

Essa discrepância da estrutura dos dados armazenados no banco de dados e as estruturas de dados em memória é chamada de **impedância** e em algum momento o desenvolvedor precisará resolvê-la. Quanto menor for a impedância, menos trabalho de conversão o desenvolvedor terá entre as diversas camadas que compõem seu sistema.

Bancos NoSQL

Existem no mercado bancos que armazenam dados em estruturas idênticas ou semelhantes ao JSON, como o **Apache CouchDB** (http://couchdb.apache.org) , **JasDB** (http://www.oberasoftware.com) , **NeDB** (Node Embedded Database) (https://github.com/louischatriot/nedb) , **Terrastore** (https://code.google.com/p/terrastore) entre outros. Qual deles escolher?

MongoDB

O MongoDB é um banco de dados baseado em documento com alta performance e disponibilidade e de fácil escalabilidade. Porém, essas caracterísi

cas têm seu preço: nem todas as propriedades ACID (atomicidade, consistência, isolamento e durabilidade) tão prezadas nos bancos de dados relacionais são implementadas pelo banco.

O conceito de documento

Um dos pontos que torna o MongoDB ainda mais interessante na MEAN Stack é a maneira como trabalha com documentos, conceito fundamental

para este banco NoSQL.

Características do documento

Um documento pode ser armazenado em diferentes formatos hierárquicos como XML ou JSON com os dados associados a uma estrutura de chave e valor. É através de uma chave específica do documento que temos acesso ao valor associado. Porém, dos diversos formatos existentes no mercado, o MongoDB armazena seus documentos no formato **BSON** (Binary JSON), muito parecido com JSON. Comparemos a estrutura desses dois formatos:

```
// JSON
{
"nome" : "Jose Alves"
}
// BSON
{
"_id" : ObjectId("5303e0649fd139619aeb783e")
"nome" : "Jose Alves"
}
```

A grosso modo, a diferença entre BSON e JSON mora na quantidade dos tipos de dados suportados em cada uma deles: enquanto JSON possui seis tipos (Array, Boolean, null, Number, Object e String) o BSON,possui mais de 15 tipos! Veremos alguns desses tipos, porém há um que merece destaque neste momento: ObjectId.

Utilizando o mongo shell

O MongoDB possui o **mongo shell**, um cliente em linha de comando que nos permite interagir com suas instâncias. Para acessá-lo, basta digitarmos

no terminal o comando **mongo** seguido de dois parâmetros:

$ mongo --port 27017 --host localhost

Repare que no comando anterior passamos os parâmetros --port e --host com os valores 27017 e localhost, respectivamente. Esses valores já são o padrão do mongo shell justamente por eles também serem o padrão da instância do MongoDB que roda em sua máquina.

Criando o banco da aplicação

o comando **show dbs** que listamos todos os bancos da instância do MongoDB a que estamos conectados:

```
> show dbs
test 0.203125GB
```

Precisamos agora criar um banco que contenha os dados da nossa aplicação Por conveniência, criaremos um banco de mesmo nome, apenas começando com letra minúscula através do comando use banco

```
> use banco
switched to db banco
```

percebemos que o banco ainda não existe, porém, o MongoDB irá criálo. O mais interessante é que recebemos a mensagem switched to db contatooh. O mongo shell está nos avisando que o objeto db agora aponta para o banco contatooh. Será verdade? Se imprimirmos o objeto db, veremos que ele realmente aponta para nosso novo banco:

A variável db nos fornece um atalho para o banco em que estamos trabalhando no momento. Se por acaso usássemos o comando use test, nossa variável db apontaria para o banco test.

Collections e inserção de documentos

O shell do MongoDB permite criar variáveis nos moldes da linguagem JavaScript. Isso é fantástico, porque se a estrutura de dados BSON do MongoDB é parecida com JSON. Podemos criar um objeto JavaScript com a seguinte

estrutura:

```
> var contato = { "nome" : "Nome do Contato"}
```

A ideia agora é incluir este objeto em nosso banco. Não se preocupe se a nossa estrutura ainda não é um BSON, porque o próprio MongoDB resolverá essa impedância no momento da inclusão.

Temos um problema: queremos adicionar um contato a uma tabela chamada contatos, mas sabemos que o MongoDB não trabalha com tabelas e, sim, collections para agrupar documentos. Como criar uma collection para depois incluir nosso contato?

Podemos fazer as duas coisas de uma só vez! Vamos convencionar que o nome da collection que armazenará nossos contatos será contatos. Logo, nossa inclusão ficará assim:

```
> var contato = { "nome" : "Nome do Contato" }
> db.contatos.insert(contato)
WriteResult({ "nInserted" : 1 })
```

Buscando documentos

Já sabemos criar bancos e collections, e até inserir documentos nelas,porém ainda não verificamos se realmente nosso contato foi gravado. Para isso, pediremos à nossa collection que “ache” nosso documento através da função find:

```
> db.contatos.find();
{
"_id" : ObjectId("53e1534bf8a1b188b3f276d1"),
"nome" : "Nome do Contato"
}
```

O objeto cursor

Aprendemos a listar nossos documentos através da função find. Que tal guardarmos o resultado da função em uma variável para depois imprimirmos seu valor?

```
> var contatos = db.contatos.find()
> contatos
```

Funciona! Nossa variável contatos contém todos nossos contatos. Será mesmo? Vamos imprimi-la novamente:

> contatos

Desta vez, nada acontece! Por quê? Isso ocorre porque a lista retornada pelo método find da collection não é um array de objetos, mas um **cursor** criado pelo MongoDB, que buscará nossos dados apenas quando precisarmos.

Quando executávamos a função find diretamente ou quando guardávamos seu retorno em uma variável e depois a imprimíamos, o que o shell do MongoDB na verdade fazia era chamar a função next() do cursor, que retornava um contato a cada chamada.

Buscando com critério

Muitas vezes queremos encontrar apenas um documento, por exemplo, o primeiro que encontramos em nossa collection. Para isso existe a função**findOne** presente em toda collection:

```
> db.contatos.findOne()
{
"_id" : ObjectId("530b825db61fac75624ccfd1"),
"nome" : "Contato 1 Mongo",
"email" : "cont1@empresa.com.br"
}
```

a função findOne sempre retorna um objeto do banco, não um cursor

Critérios

Critérios são criados através da estrutura de dados JSON e são passados para as funções find e findOne da collection:

```
> var criterio = { "email" : "cont2@empresa.com.br" }
> var contato = db.contatos.find(criterio)
> contato
```

No exemplo anterior, criamos um critério que nada mais é do que um JSON com a chave email recebendo o valor que desejamos buscar no banco.

O MongoDB é inteligente para comparar a chave email com a mesma chave presente em todos os documentos de nosso banco, retornando apenas aqueles que atendem a o critério passado.

Alternativa like

E se quisermos encontrar todos os contatos que contenham o valor tato como parte do nome, ignorando maiúscula ou minúscula? Se você é conhecedor de expressão regular, já sabe a resposta. Podemos passar como critério expressões regulares:

```
> var criterio = { "nome" : /tato/i }
> var contatos = db.contatos.find(criterio)
```

Contando elementos

Podemos saber quantos contatos temos cadastrados no banco sem precisarmos retornar um cursor. Isso é feito através da função **count** chamada diretamente na collection pesquisada:

```
> db.contatos.count()
2
```

Query selectors

Qualquer critério passado para as funções find e findOne terão suas chaves consideradas como cláusulas AND. Mas se quisermos realizar um OR? Para isso, utilizaremos um dos vários **query selectors** do MongoDB:

```
> db.contatos.find({
"$or" : [
{"email" : "cont2@empresa.com.br"},
{"nome" : "Contato 1 Mongo"}
])
```

Podemos ainda obter todos os contatos que não contenham como e-mail cont2@empresa.com.br, através do *query selector* $ne:

```
> db.contatos.find({
"email" : {
"$ne" : "cont2@empresa.com.br"
}
});
```

Removendo documentos

A remoção de documentos no MongoDB é feita através da função remove chamada diretamente na collection alvo:

```
> db.contatos.remove({ "email" : "cont1@empresa.com.br" })
WriteResult({ "nRemoved" : 1 })
```

Repare que criamos o critério da mesma maneira que criamos para nossas consultas. A lógica é a mesma.

Atualizando documentos

Um banco de dados não armazena dados apenas, ele permite que esses dados sejam atualizados de acordo com algum critério. Por exemplo, precisamos atualizar o contato com e-mail cont3@empresa.com.br. Já aprendemos como obter contatos de acordo com um critério.

```
> var criterio = { "email" : "cont3@empresa.com.br" }
> var contato =  = { "nome" : "Nome do Contato" }
> db.contatos.update(criterio, contato)
WriteResult({ "nMatched" : 1, "nUpserted" : 0 })
```

Mongoose Object-Document Modeler

mongoose é uma biblioteca de ODM (*Object-Document Modeler*) criada pela equipe do MongoDB. Ela é uma camada entorno do driver do MongoDB que gerencia relacionamentos e executa validações, entre outras funcionalidades.

Mongoose funciona da seguinte maneira: no lado da aplicação, criamos **esquemas** que servem como molde para criação dos modelos da aplicação. Criamos **objetos** a partir desses **modelos** com auxílio do Mongoose e toda alteração realizada nesses objetos é persistida no banco através de métodos específicos presentes na instância do modelo ou diretamente no próprio modelo.

Instalação

```
npm install mongoose --save
```

Gerenciando a conexão

Nossa primeira tarefa será isolar o código de inicialização da conexão em seu próprio arquivo. Criaremos o módulo database.js dentro da pasta config, que terá como dependência o Mongoose. Ele receberá como parâmetro a URL do banco de nossa aplicação que passaremos para o Mongoose, este último responsável pela abertura e gerenciamento da conexão:

```
// config/database.js
var mongoose = require('mongoose');
module.exports = function(uri) {
mongoose.connect(uri);
mongoose.connection.on('connected', function() {
console.log('Mongoose! Conectado em ' + uri);
});
mongoose.connection.on('disconnected', function() {
console.log('Mongoose! Desconectado de ' + uri);
});
mongoose.connection.on('error', function(erro) {
console.log('Mongoose! Erro na conexão: ' + erro);
});
}
```

Precisamos garantir que a conexão seja fechada quando nossa aplicação for terminada. Vamos interagir com o objeto process globalmente disponível pelo Node.js e acessível em qualquer local de nossa aplicação. O objeto process possui o evento SIGINIT disparado quando nossa aplicação é terminada (por exemplo: CONTROL + C no terminal). É através do callback associado a este evento que pediremos ao Mongoose que feche nossa conexão através da função close:

```
process.on('SIGINT', function() {
mongoose.connection.close(function() {
console.log('Mongoose! Desconectado pelo término da aplicação');
// 0 indica que a finalização ocorreu sem erros
process.exit(0);
});
```

Mesmo sem termos terminado nosso módulo você deve estar se perguntando quem o chamará. A ideia é que nossa conexão seja iniciada com o servidor. É por isso que chamaremos nosso módulo database.js através de nosso server.js:

```
// /server.js
var http = require('http');
var app = require('./config/express')();
require('./config/database.js')('mongodb://localhost/contatooh');
http.createServer(app).listen(app.get('port'), function(){
console.log('Express Server escutando na porta ' +
app.get('port'));
});
```

Pool de conexões

A função mongoose.connect cria por padrão um pool com cinco conexões, porém esta quantidade pode não ser adequada para todas as aplicações. Podemos alterar esta quantidade passando uma configuração extra para a função:

```
mongoose.connect(uri, { server: { poolSize: 15 }});
```

Criando esquemas

Vimos que o MongoDB é um banco sem esquemas, não exigindo de seus documentos determinada estrutura. Contudo, isso não torna o esquema menos importante. É por isso que o Mongoose possui o objeto Schema, que define a estrutura de qualquer documento que será armazenado em uma collection do MongoDB. Ele permite definir tipos e validar dados.

Nosso primeiro passo será criar o módulo contato.js na pasta app/models. Nele, declaramos um objeto Schema através da função mongoose.Schema que recebe como parâmetro uma série de critérios. Por enquanto, passaremos um objeto vazio como critério:

```
// /models/contato.js
var mongoose = require('mongoose');
module.exports = function() {
    var schema = mongoose.Schema({
        nome: { type: String },
        email: { type: String }
    });
    return mongoose.model('Contato', schema);
};
```

Um Model é um objeto que corresponde a uma collection de nosso banco e utiliza o Schema usado em sua criação para validar qualquer documento que tenhamos na collection.
É por isso que a última linha do nosso módulo retornará um Model criado a partir do nosso Schema:

Adicionar obrigatoriedade

Queremos também que o nome e o email sejam obrigatórios. Isso é feito adicionando a chave required no objeto de configuração:

```
nome: {
    type: String,
    required: true
}
```

Por fim, não pode haver contatos com o mesmo e-mail. Garantimos isso adicionando mais uma chave de configuração, desta vez a index:

index: { unique: true }

Testando

Podemos realizar um teste para ver se tudo está funcionando. Primeiro, apagaremos o banco contatooh através do mongo shell:

```
$ mongo
> use contatooh
> db.dropDatabase()
```

Note que estamos conectados ao banco contatooh que acabamos de apagar! O Mongoose extraiu o nome do banco da URL de conexão criando para nós, inclusive criou a collection contatos, adotando como padrão o nome passado como primeiro parâmetro para a função mongoose.model, porém em *lowercase* e no plural. Para ficar ainda mais interessante, o índice de unicidade para a chave email também foi criado. Vejamos tudo isso através do mongo shell:

```
$ mongo
> show dbs
> use contatooh
switched to db contatooh
> show collections
contatos
system.indexes
> db.contatos.getIndexes()
// exibe detalhes do índice criado
```

Exercicio

Crie uma base e insira 5 valores

Buscar 1 elemento com criterio

Altere 1 valor

remova outro valor

monstre as mensagens do terminal para cada um dos comandos