04

Qual é o esquema?

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/mean-js/stages/10-alurapic.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo. Só não esqueça de baixar as dependências do projeto no terminal com o comando `npm install`.

Mostre seu documento!

Muito bem, temos uma conexão criada para o MongoDB através do Mongoose, mas e daí? O que fazemos com ela? Bem, nosso foco será a manipulação de **documentos**. O MongoDB não trabalha com o conceito de tabela como nos banco de dados relacionais, não há linhas e colunas como estamos acostumados. O que há são documentos que possuem uma estrutura de chave e valor como o JSON. Duvida? Vou abrir o **mongo shell** e criar um para vermos:

```
$ mongo
MongoDB shell version: 3.0.7
connecting to: test
```

Por padrão o MongoDB nos conecta ao banco test. Vou pedir para trocar para o alurapic:

```
use alurapic
switched to db alurapic
>
```

Podemos interagir com o alurapic através da variável `db`, um atalho para nosso banco. Tanto isso é verdade que se imprimirmos a variável no console temos:

```
> db
alurapic
>
```

Agora, vou criar um objeto Javascript que representa uma foto:

```
> var foto = { titulo: 'Leão', url : ''};
> foto
{ "titulo" : "Leão", "url" : "" }
>
```

Podemos usar toda aquela sintaxe conhecida do Javascript no mongo shell!

Documento arquivado

A questão agora é: onde eu gravo esse documento? No MongoDB, uma estrutra que lembra o conceito de tabela é a **coleção**. Em nosso exemplo, quero armazenar o documento foto na coleção `fotos`:

```
> db.fotos.insert(foto);
WriteResult({ "nInserted" : 1 })
```

Estamos acessando a propriedade `.fotos` do nosso banco `alurapic`. Por mais estranho que isso pareça, se essa propriedade não existir, o MongoDB a criará ao mesmo tempo em que cria uma collection de mesmo nome. Em seguida, a partir da coleção `fotos`, realizamos uma inclusão através da função `insert`. Eu sei, muita coisa acontecendo numa única linha! No final, o console indica que um documento foi inserido.

E agora? Será que gravou mesmo nossa foto? Podemos listar todos os documentos de uma coleção através da função `find`:

```
> db.fotos.find()
{ "_id" : ObjectId("5670728ae73ae83090993ccf"), "titulo" : "Leão", "url" : "" }
```

Curiosamente, quando listamos todos os documentos do banco, nossa foto que acabamos de gravar vem com uma propriedade que não havíamos adicionado, a `_id`. Isso acontece porque todo documento deve ter um identificador único por coleção e o MongoDB sempre criará um para nós toda vez que adicionamos um documento.

Agora, vejam que curioso:

```
> var foto2 = { titulo : 'Leopardo', url: '', grupo: 2};
> db.fotos.insert(foto2);
WriteResult({ "nInserted" : 1 })
```

Conseguimos gravar uma foto com uma estrutura diferente na coleção `fotos`. Isso é possível? Vamos listar:

```
> db.fotos.find();
{ "_id" : ObjectId("5670728ae73ae83090993ccf"), "titulo" : "Leão", "url" : "" }
{ "_id" : ObjectId("56707392e73ae83090993cd0"), "titulo" : "Leopardo", "url" : "", "grupo" : 2 }
```

Não tem esquema?

Sim é possível. Isso acontece porque o MongoDB não trabalha com esquemas. Em um banco de dados relacional, criamos esquemas para indicar que todo registro deve ter X colunas e que essas colunas são determinado tipo. Como você viu, eu posso gravar um documento com uma estrutura diferente na mesma coleção. Posso até gravar um documento no qual o título é um número.

Dando um jeitinho

Então esquema não é importante? Claro que são, mas no MongoDB quem deve garantir a consistência dos seus dados é sua aplicação, não o banco. Dessa forma, a criação de esquemas é responsabilidade da aplicação. É na tarefa de criação de esquemas que o Mongoose nos ajudará bastante, além de nos fornecer funções especializadas na manipulação de documentos.

O esquema de documentos

Então, em qual diretório criaremos os esquemas da aplicação? Vamos criá-los dentro da nova pasta `alurapic/app/models`. Veremos que um esquema do Mongoose no final gera um modelo que será usado pela aplicação. Criando o arquivo `alurapic/app/models/foto.js`:

```
// alurapic/app/models/foto.js

var mongoose = require('mongoose');

// cria o esquema
var schema = mongoose.Schema();
```

É através da função `mongoose.Schema` que criamos um esquema. Pense no `schema` como aquele que determinará quais atributos nosso modelo `Foto` deve ter, inclusive seus tipos e regras de validação. Mas olhando nosso código, onde está a definição dessas regras? Em nenhum lugar! Precisamos passar essas configurações para a função `mongoose.Schema`:

```
var schema = mongoose.Schema({

    titulo: {
        type: String,
        required: true
    },
    url: {
        type: String,
        required: true
    },
    grupo: {
        type: Number,
        required: true
    }
});
```

Passamos um objeto como parâmetro e suas propriedades representam os atributos que teremos em nosso documento. Em nosso exemplo, teremos `titulo`, `url` e `grupo`. Porém, para cada um desses atributos, podemos passar um outro objeto como parâmetro. É através desse objeto que dizemos o tipo do campo e se ele é obrigatório ou não. Por exemplo, o atributo `grupo` é do tipo `Number`, se passarmos uma `String` como parâmetro o mongoose não o aceitará. Inclusive, se tentarmos adicionar propriedades que não existem no esquema, elas serão ignoradas.

Precisamos de um modelo

Ótimo, criamos nosso primeiro schema, mas quando interagimos com o MongoDB através do Mongoose, não fazemos isso através dele, mas de um objeto modelo que segue a regra desse `schema`. É por isso que depois de criarmos um `schema`, precisamos solicitar ao mongoose que **compile** nosso esquema e gere um modelo para que possamos utilizar em nossa aplicação:

```
var schema = mongoose.Schema({

    titulo: {
        type: String,
        required: true
    },
    url: {
```

```
            type: String,
            required: true
        },
        grupo: {
            type: Number,
            required: true
        },
        descricao: {
            type: String
        }
    });

     // compilando um modelo com base no esquema
    mongoose.model('Foto', schema);
```

Pronto, a questão agora é que o módulo `alurapic/app/models/foto.js` precisa ser carregado assim que nossa aplicação subir. Isso não é problema para nós, porque aprendemos a carregar módulos através do `consign`. Vamos alterar `alurapic/config/express.js`:

```
var express = require('express');
var consign = require('consign');
var bodyParser = require('body-parser');

var app = express();

app.use(express.static('./public'));
app.use(bodyParser.json());

consign({cwd: 'app'})
    .include('models')
    .then('api')
    .then('routes')
    .into(app);

module.exports = app;
```

Não é por acaso que carregamos primeiro a pasta `models`. Como nossa API dependerá do modelo criado pelo mongoose, esses modelos precisam existir primeiro antes de serem criados. Até agora tudo bem?

Você deve estar pensando que acessaremos nosso modelo através de `app.models.foto`, mas não o acessaremos assim. Podemos em qualquer lugar da nossa aplicação requerer um modelo bastando importar o módulo do mongoose e em seguida chamar a função `mongoose.model('nomeDoModel')`. Veja que dessa vez, não passamos um esquema como parâmetro, justamente para indicar que queremos acessar um modelo já criado e não criar outro. Muito abstrato? Vamos ver isso na prática alterando `alurapic/app/api/fotos.js`:

Com o arquivo `alurapic/app/api/fotos.js` aberto, vamos remover o conteúdo de todas as funções da nossa API, inclusive remover as variáveis `fotos` e `ID_CONTADOR`, pois começaremos a realizar a integração como banco.

Seu arquivo deve estar assim:

```
// alurapic/app/api/fotos.js

module.exports = function(app) {
```

```
    var api = {};

    api.lista = function(req, res) {

    };

    api.buscaPorId = function(req, res) {

    };

    api.removePorId = function(req, res) {

    };

    api.adiciona = function(req, res) {

    };

    api.atualiza = function(req, res) {

    };

    return api;
};
```

Vamos agora importar o módulo mongoose e solicitar o modelo `Foto` :

```
// alurapic/app/api/foto.js

var mongoose = require('mongoose');

module.exports = function(app) {

    var api = {};

    // solicitando o modelo 'Foto'
    var model = mongoose.model('Foto');

    api.lista = function(req, res) {

    };

    api.buscaPorId = function(req, res) {

    };

    api.removePorId = function(req, res) {

    };

    api.adiciona = function(req, res) {

    };

    api.atualiza = function(req, res) {
```

```
    };

    return api;
};
```

Cuidado, se criamos no mongoose um model com o nome `Foto`, temos que usar esse mesmo nome quando pedirmos o model ao mongoose. Você não vai esquecer? Nem eu! :).

```
// alurapic/app/api/foto.js
// código anterior omitido
    api.lista = function(req, res) {
        model.find(function(error, fotos) {

        });
    };
// código posterior omitido
```

Interessante, também usamos a função `find` com um model do mongoose para buscarmos documento no Banco. Seu parâmetro é uma função cujo primeiro parâmetro nos dá acesso ao erro que possa ocorrer e o segundo o retorno do banco, em nosso caso, uma lista de fotos. Sempre que a operação for realizada sem problema algum, o parâmetro `error` será `undefined`.É por isso que precisamos levar em consideração esse parâmetro para sabermos que resposta enviamos:

```
// alurapic/app/api/fotos.js
// codigo anterior omitido
    api.lista = function(req, res) {

        model.find(function(error, fotos) {
            if(error) {
                console.log(error);
                res.status(500).json(error);
            }
            res.json(fotos);
        });

    };
// código posterior omitido
```

Enviamos como resposta uma lista de fotos e caso algum erro aconteça, enviamos o código 500 (internal server error) sinalizando para a aplicação Angular que houve um problema. Já podemos reiniciar o servidor e verificar se a lista de fotos que cadastramos no MongoDB é exibida. Confie em mim, ela será exibida!

Podemos melhorar um pouco nosso código, porque a função `find` retorna uma **promise**. Lembra do nosso curso de Angular? Usamos promises o tempo todo, inclusive você aprendeu a criar um serviço que retornava uma promise. Você deve lembrar que uma promise possui a função `then` e que ela recebe dois parâmetros. A primeira, é a função que será chamada quando nossa promise for resolvida, a segunda caso ocorra algum problema. É através da primeira que teremos acesso à todas as fotos retornadas do banco:

```
    api.lista = function(req, res) {

        model.find()
```

```
        .then(function(fotos) {
            res.json(fotos);
        }, function(error) {
            console.log(error);
            res.sendStatus(500);
        });

    };
```

Veja que agora temos bem separado a lógica do nosso código da parte que trata erros. Mais um teste demonstra que tudo continua funcionando. Pronto para implementar o restante da nossa API usando o Mongoose? No próximo capítulo faremos isso.