04

Cordão umbilical servidor e banco

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/mean-js/stages/09-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` .

Menor impedância

Conseguimos fazer a comunicação da nossa aplicação Angular com nosso server. Durante esta comunicação, tanto um quanto outro enviam e recebem dados. Um ponto curioso a se destacar é que **trafegamos a estrutura de dados JSON**. Essa estrutura tão difundida no mundo Javascript é de fácil manipulação, inclusive tanto nosso servidor quando nosso cliente Angular sabem transformar esse formato textual em objetos que podem ser manipulados.

Sabemos que precisamos ainda implementar a persistência dos nossos dados, porque a cada reinicio do servidor perdemos nossas modificações. A pergunta que lhe faço é a seguinte: qual banco utilizaremos? A escolha de um banco de dados não é uma questão de gosto, mas envolve decisões arquiteturais que nem sempre são triviais. A boa notícia é que não precisaremos meditar muito para escolhermos o nosso banco, pois no acrônimo MEAN o M significa **MongoDB**.

O MongoDB (https://www.mongodb.org/) é um banco noSQL orientado a documento que armazena seus dados em uma estrutura de dados extremamente parecida com JSON e dependendo da maneira que interagimos com esse banco através do nosso código no backend podemos tratá-lo como tal. Dentro desse contexto, gravaremos nossos dados no formato "JSON", leremos através da nossa aplicação Node.js esse dado que será enviado diretamente para nossa aplicação Angular que entende com facilidade essa estrutura. Repare que durante esse trâmite não gastaremos tempo lidando com conversões, inclusive quando formos atualizar os dados, enviaremos o JSON atualizado que será recebido pelo backend e que será gravado diretamente no banco! Perfeito. No final das contas, nossa aplicação terá menor impedância, que é a discrepância das estrutura de dados do banco e essa estrutura de dados em memória.

Mongoose, que nome estranho!

Para interagirmos com o MongoDB diretamente do nosso servidor Node usaremos o módulo Mongoose (http://mongoosejs.com/). Sua instalação é direta através do `npm` :

```
npm install mongoose@4.3.1 --save
```

Construindo o cordão umbilical entre servidor e banco: a conexão

Agora, antes de pensarmos em qualquer operação com o banco, precisamos criar uma conexão para ele dentro da nossa aplicação. Ai eu te pergunto: em qual pasta guardaremos essa configuração? Se você respondeu `alurapic/config` você acertou. Vamos criar o arquivo `alurapic/config/database.js` e importar o módulo do Mongoose:

```
// alurapic/config/database.js

var mongoose = require('mongoose');
```

Nosa estrutura de projeto está bem legal. Conseguimos ver claramente onde está o código que roda no cliente e aquele que roda em nosso servidor, inclusive onde ficam as suas configurações. Temos duas grandes separações `app` (server) e `public` (cliente).

O próximo passo é abrirmos uma conexão com o MongoDB através do Mongoose:

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/alurapic');
```

Usamos a função `mongoose.connect` que recebe como parâmetro uma string de conexão. Essa string deve sempre começar com `mongodb://` seguida do endereço do banco, em nosso caso, `localhost` e o nome do banco, que chamaremos de `alurapic` . Se tentarmos conectar em um banco que não existe, o mongoose solicitará ao MongoDB que crie um novo para nós assim que interagirmos com ele.

Excelente, já configuramos a conexão com o MongoDB, mas será que se reiniciarmos nosso servidor já nos conectaremos? Claro que não, porque não chamamos o módulo `database.js` em nenhum lugar da nossa aplicação. Onde o chamaremos? Que tal em `server.js` ? Além dele subir nosso servidor, ele também abrirá a conexão com o banco. Vamos lá:

```
// alurapic/server.js
var http = require('http');
var app = require('./config/express')

//novidade aqui
require('./config/database');

http.createServer(app)
.listen(3000, function() {
    console.log('Servidor iniciado na porta 3000');
});
```

Excelente, nosso servidor sobe e não recebemos nenhum erro. Mas será que realmente temos uma conexão com o banco? Que tal se quando o Mongoose se conectar com o MongoDB ele nos avisasse?

```
// alurapic/config/database.js

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/alurapic');

mongoose.connection.on('connected', function() {
    console.log('Conectado ao MongoDB')
});
```

Reiniciando o servidor e subindo-o mais uma vez vemos a seguinte mensagem exibida no console:

```
consign v0.1.2 Initialized in app
+ ./api/foto.js
+ ./api/grupo.js
+ ./routes/foto.js
```

```
+ ./routes/grupo.js
Servidor iniciado
Mongoose! Conectado ao MongoDB
```

Configuração genérica

Perfeito, mas se quisermos usar esse arquivo de configuração com qualquer projeto? Veja que a `URI` do banco esta fixa. Que tal alterar nosso módulo fazendo com que ele receba a URI como parâmetro?

```
// alurapic/config/database.js

module.exports = function(uri) {

    var mongoose = require('mongoose');

    mongoose.connect('mongodb://' + uri);

    mongoose.connection.on('connected', function() {
        console.log('Conectado ao MongoDB')
    });
};
```

Agora, precisamos passar essa uri como parâmetro lá na chamada do módulo em `alurapic/server.js` :

```
var http = require('http');
var app = require('./config/express');
// erro intencional aqui, "ocalhost"
require('./config/database')('ocalhost/alurapic');

http.createServer(app)
.listen(3000, function() {
    console.log('Servidor iniciado');
});
```

Cercando de todos os lados

Reiniciando e testando. Nosso servidor sequer levanta e ainda recebemos uma mensagem de erro no terminal:

```
/Users/flavioalmeida/Desktop/alurapic/node_modules/mongoose/node_modules/mongodb/lib/server.js:235
        process.nextTick(function() { throw err; })
                                      ^
Error: getaddrinfo ENOTFOUND ocalhost ocalhost:27017
```

Pois é, escrevi o endereço do servidor errado, vou corrigir. No entanto, parece que erros de conexão podem evitar que nosso servidor suba. Podemos resolver isso fazendo com que o Mongoose escute por qualquer erro de conexão que existir, evitando assim que nossa aplicação termine:

```
// alurapic/config/database.js
```

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

    var mongoose = require('mongoose');

    mongoose.connect('mongodb://' + uri)

    mongoose.connection.on('connected', function() {
        console.log('Conectado ao MongoDB')
    });

    mongoose.connection.on('error', function(error) {
        console.log('Erro na conexão: ' + error);
    });
};
```

Testando mais uma vez, recebemos a mensagem de erro no terminal (menos verbosa) e nosso server continua rodando.

```
consign v0.1.2 Initialized in app
+ ./api/foto.js
+ ./api/grupo.js
+ ./routes/foto.js
+ ./routes/grupo.js
Servidor iniciado
Erro na conexão: MongoError: getaddrinfo ENOTFOUND ocalhost ocalhost:27017
```

Agora basta corrigirmos nosso endereço para nossa conexão ser efetuada com sucesso:

```
// alurapic/server.js

var http = require('http');
var app = require('./config/express');
require('./config/database')('localhost/alurapic'); // endereço corrigido

http.createServer(app)
.listen(3000, function() {
    console.log('Servidor iniciado');
});
```

Tá fincando bom o negócio. E se acontece uma desconexão com banco? Também é interessante registrar essa informação:

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

    var mongoose = require('mongoose');

    mongoose.connect('mongodb://' + uri)

    mongoose.connection.on('connected', function() {
        console.log('Conectado ao MongoDB')
    });

    mongoose.connection.on('error', function(error) {
        console.log('Erro na conexão: ' + error);
    });
```

```
    mongoose.connection.on('disconnected', function() {
        console.log('Desconectado do MongoDB')
    });
};
```

Será que quando finalizarmos nosso servidor receberemos a mensagem de desconexão? Vamos tentar parar. Nenhuma mensagem é exibida. Será que nossa conexão continua aberta? Não, mas dependendo da sua aplicação, onde ela for hospedada pode ser que isso aconteça. Que se quando alguém matar nossa aplicação terminando-a no terminal a gente fechasse a conexão?

```
// alurapic/config/database.js

module.exports = function(uri) {

    var mongoose = require('mongoose');

    mongoose.connect('mongodb://' + uri)

    mongoose.connection.on('connected', function() {
        console.log('Conectado ao MongoDB')
    });

    mongoose.connection.on('error', function(error) {
        console.log('Erro na conexão: ' + error);
    });

    mongoose.connection.on('disconnected', function() {
        console.log('Desconectado do MongoDB')
    });

    process.on('SIGINT', function() {
        mongoose.connection.close(function() {
            console.log('Aplicação terminada, conexão fechada')
            process.exit(0);
        });

    })
};
```

A variável `process` é uma variável globalmente acessível que nos dá acesso a várias informações do sistema operacional. Com ela, escutamos o evento de término da aplicação pelo sinal `SIGINT`. Quando a aplicação é terminada, chamamos `mongoose.connection.close` para fecharmos a conexão, que por conseguinte dispara o evento `disconnected`. Usamos `process.exit(0)` para indicar que foi um término de aplicação esperado, não decorrente de um erro.

Perfeito, agora estamos preparamos para interagir com o MongoDB através do Mongoose!