MEAN

JavaScript de ponta a ponta

Manoel Braz de Andrade Maciel[1] Edson Angoti Junior[2]

[1]Estudante de Sistemas para Internet – IFTM – Campus Uberlândia Centro – manoelbraz.maciel@gmail.com [2]Professor do IFTM – Campus Uberlândia Centro – angoti @ iftm.edu.br

RESUMO: A rápida evolução da linguagem JavaScript, levou-a aos três níveis básicos da programação de sistemas baseados em redes de computadores e, mais especificamente, em sistemas para a Internet: o lado do servidor, o lado do cliente e a persistência dos dados. A Stack MEAN — MongoDB, ExpressJS, AngularJS e NodeJS — cuja integração é o objeto do presente artigo, é um conjunto de tecnologias que, diferentemente do Java e do XAMP – que têm o servidor, o cliente e a persistência de dados programadas em diferentes linguagens – utiliza a linguagem JavaScript de ponta a ponta: no servidor, no cliente e na persistência dos dados. A integração evolutiva de cada uma dessas partes: o lado do servidor, o lado do cliente e a persistência dos dados, com a stack MEAN visa facilitar o desenvolvimento de sistemas baseados em redes de computadores e para a Internet. Neste artigo é mostrado o passo a passo de como acontece a integração desses quatro componentes. Começando com um servidor simples desenvolvido com o Node apenas. Em seguida, o Node é integrado com o Express, e o servidor simples evolui e passa a servir páginas estáticas. Depois é criada uma API persistida em RAM, para ser consumida pelo Angular. Logo após o Angular é integrado a todo esse backend. E, finalmente, os dados passam a ser persistido no banco de dados MongoDB. O objetivo é obter um maior conhecimento e controle de cada uma das partes componentes desse sistema, proporcionando uma maior facilidade de evolução e manutenção de sistemas desenvolvidos com a stack MEAN.

Palavras-Chave: JavaScript, MEAN, MongoDB, ExpressJS, AngularJS, NodeJS, npm, Bower.

MEAN: End-to-end JavaScript

ABSTRACT: The rapid evolution of the JavaScript language has led to the three basic levels of programming of systems based on computer networks and more specifically in systems for the Internet: the server side, the client side and the persistence of the data. Stack MEAN - MongoDB, ExpressJS, AngularJS and NodeJS – whose integration is the object of this article, is a set of technologies that, unlike Java and XAMP – that have server, client and data persistence programmed in different languages the stack MEAN uses the end-to-end JavaScript language:

server, client and data persistence. The evolutionary integration of each of these parts: server side, client side and data persistence with the MEAN stack aims to facilitate the development of systems based on computer networks and the Internet. In this article we will show step by step how to integrate these four components. We start from a simple server with Node only. Then we integrate with Express, and it starts to serve static pages. Then we created a persistent API in RAM, to be consumed by the Angular. Soon after, the Angular is integrated with this backend. And, finally, the data is being persisted in the MongoDB database. The goal is to gain a greater knowledge and control of each of the component parts of this system, providing greater ease of evolution and maintenance of systems developed with the MEAN stack.

Keywords: JavaScript, MEAN, MongoDB, ExpressJS, AngularJS, NodeJS, npm, Bower.

1 INTRODUÇÃO

A linguagem de programação JavaScript foi criada em 1995, por Bredan Eich para o navegador Netscape. Ela já foi considerada de brinquedo e para amadores, mas é hoje uma das mais utilizadas no mundo, quase todos os dispositivos capazes de se conectar à Internet possuem um interpretador para a linguagem JavaScript. É uma linguagem de processamento assíncrono, orientada a eventos, que possui suporte à orientação a objetos e que também implementa programação funcional. A rápida evolução desta linguagem e sua ampla disponibilidade, desencadearam sua utilização em todas as etapas do desenvolvimento de aplicações de rede e, mais especificamente, em sistemas para a Internet, culminando com o surgimento de muitas ferramentas com a promessa de facilitar e acelerar o desenvolvimento, uma dessas ferramentas é a stack MEAN – MongoDB, ExpressJS, AngularJS e NodeJS, cuja a integração de suas partes é o objeto do presente artigo.

1.1 O lado servidor

Em 2009, o JavaScript aparece no lado do servidor, durante a JSConf Europa, quando Ryan Dahl apresenta o NodeJS, uma plataforma para construir aplicações de rede que utiliza o motor JavaScript V8 de alto desempenho, do navegador Google Chrome. Nessa função, o NodeJS recebe a ajuda de um módulo especial chamado Express, este é um framework que permite construir

aplicações Web e expor APIs do tipo RESTful baseadas também em JSON. Os dois juntos

desempenham o fundamental papel de manipuladores de requisição, respondendo exatamente com aquilo que o cliente solicitou de forma rápida e eficaz.

1.2 O lado cliente

O JavaScript possui um sem número de bibliotecas, sendo a mais famosa delas talvez seja o jQuery. A outra que está ficando tão famosa quanto é o AngularJS, ele facilita muito a sua utilização no lado do cliente, sem a manipulação do DOM.

1.3 A persistência de dados

A linguagem JavaScript está presente na persistência de dados, em vários sistemas, um deles é o MongoDB, um banco de dados NoSQL que utiliza uma representação de documentos muito parecida com o JSON, formato familiar a programadores JavaScript.

1.4 A stackfull

Essas quatro tecnologias juntas – MongoDB, AngularJS, ExpressJS e NodeJS – formam o que ficou conhecido como MEAN, que é algo mais que um acrônimo, é uma plataforma baseada em JavaScript que permite o desenvolvimento ágil, com alto desempenho e escalabilidade, sendo ideal para construir aplicações responsivas com uma grande base de usuários, como a grande parte das modernas aplicações de rede e, mais especificamente, sistemas para a Internet. Ela ainda conta com diversos gerenciadores de dependências como o npm, no lado servidor e o Bower no lado do cliente, os quais serão utilizados no decorrer do desenvolvimento da nossa aplicação de demonstração da integração da stack.

2 REFERENCIAL TEÓRICO

2.1 MongoDB

MongoDB é um banco de dados de código aberto, gratuito, de alta performance, sem

esquemas e orientado a documentos, lançado em fevereiro de 2009 pela empresa 10gen. Foi escrito na linguagem de programação C++, o que o torna portável para diferentes sistemas operacionais. O Mongo é um banco de dados NoSQL líder, capacitando as empresas a serem mais ágeis e

escaláveis. É projetado para facilidade de desenvolvimento e dimensionamento. Ele usa um modelo de documento muito parecido com JSON (JavaScript Object Notation), familiar a programadores JavaScript (MONGODB).

2.2 Express

O Express é um Framework web rápido, flexível e minimalista para Node.js. É um servidor web bastante transparente que não apenas permite construir aplicações web como também expor APIs do tipo RESTful baseadas em JSON. O ExpressJS é bastante modular e processa ou modifica toda e qualquer solicitação vinda do navegador usando um elemento chamado **middleware**. Existe middleware para autenticação, manipulação de cookies, registro de eventos (log), tratamento de erros e muitas outras funções. Ou seja, o Express intercepta a requisição, aplica o middleware correspondente e passa para a próxima porção de código, até todos os disponíveis tenham sido aplicados (EXPRESSJS).

2.3 Angular

O Angular é um framework MVW client-side que trabalha com tecnologias já estabelecidas: HTML, CSS e JavaScript. O foco do Angular é a criação de Single Page Applications (SPA), embora isso não o impeça de ser utilizado em outros tipos de aplicação. O Angular guia o desenvolvedor de maneira implacável na organização de seu código. As aplicações com o Angular são compostas por componentes. Um componente é a combinação de um modelo HTML e um código que controla uma parte da tela. Em outras palavras, uma porção do código HTML delimitado por uma tag pode ser controlado por uma porção de código JavaScript referenciado dentro dessa tag. O AngularJS deixa isso bem explícito quando é criado o módulo *controller* onde os dados serão tratados e o *template* HTML com as diretivas onde os dados serão expostos (ANGULARJS).

2.4 Node

O Node é um ambiente de execução JavaScript construído sobre o motor JavaScript V8 do Chrome. Ou seja, o Node.js é executado sobre essa plataforma da Google que foi desenvolvida em C++ para ser um ambiente extremamente rápido. O Node usa um modelo de entrada e saída sem

bloqueio, orientado a eventos, que o torna leve e eficiente. Essas são características perfeitas para solucionar os problemas de intenso tráfego de rede e aplicações que respondam em tempo real, que são frequentemente os maiores desafios das aplicações web hoje em dia. O Node é o coração da stack MEAN. Ele será o eixo central da aplicação de exemplo que será desenvolvida para demonstrar a integração da stack (NODEJS).

2.5 npm

O npm é o gerenciador de dependências que atua no lado do servidor. O ecossistema de pacotes do Node, npm, é o maior ecossistema de bibliotecas de código aberto do mundo, segundo o seu site oficial. Ele será utilizado para gerenciar os módulos necessários para a integração do Express e do MongoDB. Ele será o responsável por gerenciar e prover essas dependências (NPMJS).

2.6 Bower

O bower é o gerenciador de dependências que atua no lado do cliente, provendo não só as dependências JavaScript, mas também as CSS, entre outras. Ele será utilizado para gerenciar principalmente os módulos necessários para a integração com o Angular. Ele será o responsável por gerenciar e prover essas dependências (BOWER).

3 A INTEGRAÇÃO DA STACK

A stack MEAN, como já foi dito, possui quatro componentes: MongoDB, Express, Angular e Node. A princípio pode parecer mais difícil usar quatro tecnologias diferentes para desenvolver aplicativos mas, a stack conta a seu favor, com o fato de que todas elas usam a mesma linguagem: JavaScript. É fato que a maioria dos desenvolvedores de aplicativos para a internet já teve algum

contato com JavaScript ao longo da sua formação e/ou carreira. Portanto, há uma grande possibilidade de que os possíveis candidatos a utilização da stack MEAN já tenha alguma familiaridade com essa linguagem. O outro fato é que todos os lados da aplicação usarão JavaScript: backend, frontend e persistência de dados, facilitando o desenvolvimento quando feito em equipe. Um posto ocasionalmente vago pode facilmente ser suprido por um outro membro, mesmo que do outro lado da equipe. Como objetivo principal do presente estudo, será analisado como ocorre a

integração desses quatro componentes. Será desenvolvida uma aplicação simples, que paulatinamente será integrada por cada um dos quatro componentes. Começando com o Node, depois o Express, em seguida o Angular e, finalmente o MongoDB. Para conter todo o projeto, primeiramente será criada uma pasta chamada Contacts, que será a pasta raiz do projeto.

3.1 O Node

O estudo da integração da stack será iniciado, criando um servidor muito simples com apenas o Node. O Node é o coração da stack. Tendo-o instalado em um Sistema Operacional, com algumas poucas linhas de código, e utilizando o módulo 'http' interno a ele, é possível obter um Servidor Web muito simples, mas completamente funcional. Dentro da pasta raiz do projeto será criado um arquivo chamado server.js, com o seguinte código:

```
// server.js

const http = require('http');

const host = 'localhost';
const port = 3001;

const server = http.createServer(function(request, response){
 response.writeHead(200, {'content-Type': 'text/html'});
 response.write('<h1>Hello ... World!</h1>');
 response.end();
});

server.listen(port, host, function(){
```

```
console.log('Servidor ... Respondendo em: %s: %s!', host, port);
});
```

Esse é o código do servidor Web inicial, extremamente simples criado com o Node. Esse servidor, apesar de muito simples, já mostra algumas funcionalidades interessantes do Node. Há uma constante 'http' que através da função require é carregada com o módulo http interno ao Node. A função require executa o código do módulo http, disponibilizando nessa constante tudo o que estiver sendo exportado pelo módulo http. A função createServer do módulo http, cria o servidor e o

carrega na constante server, e disponibiliza os objetos 'request' e 'response' que serão manipulados durante a execução da função anônima de callback, essa função será executada toda vez que o servidor receber uma requisição de um cliente – normalmente um browser ou uma outra aplicação capaz de fazer essa solicitação. Ele vai ficar aguardando por solicitação no endereço especificado pela sua função listen. Esses dois objetos são fundamentais no funcionamento do servidor: o 'request' fornece os dados da requisição e é através do 'response' que o servidor disponibilizará as respostas necessárias para satisfazer uma solicitação. O que determina o que faz cada parâmetro, não é o nome, mas a sua posição dentro da função de callback.

Com o comando a seguir sobe o servidor simples:

C:\tcc\Contacts>**node server.js**

Será exibida no console a seguinte resposta:

Servidor ... Respondendo em: localhost: 3001!

Abrindo-se um navegador e apontando para o endereço 'http:\\localhost:3001', será recebida a resposta do servidor recém-criado que estará sempre aguardando por solicitação, até que seja derrubado por um Ctrl + C, ou outra maneira capaz de interromper a sua execução. A resposta é com toda a simplicidade possível um "Hello … World!". Porém este servidor é muito limitado, pois responde sempre com o mesmo conteúdo a qualquer que seja a solicitação e, normalmente, é preciso de muito mais do que isso em uma aplicação real. Claro que tudo poderia ser resolvido diretamente com o Node. Mas, fica claro também, que tudo deveria ser minunciosamente desenvolvido dentro do Node. O que, além de penoso, demandaria um conhecimento profundo dele e da linguagem JavaScript. E é exatamente para isso que serve a 'stack MEAN'. A maior parte das

necessidades de uma aplicação já foram desenvolvidas e estão disponíveis em módulos. Um exemplo já utilizado, é o módulo 'http' com o qual foi criado o servidor. No próximo tópico será abordado um outro módulo: o Express.

3.2 O Express

O Express é um módulo externo ao Node e, diferentemente do módulo 'http' que é interno e foi utilizado sem a necessidade de uma instalação, é necessário instalá-lo na aplicação de exemplo, tornando-o uma dependência. Por isso antes de se falar do Express, é necessário conhecer o gerenciador de dependências do lado servidor do projeto: **o npm**.

3.2.1 npm – o gerenciador de dependências no lado do servidor

O **npm** faz parte do pacote do Node, ele gerencia a limitação inicial do Node, provendo os módulos necessários para estender as funcionalidades do servidor básico de maneira muito simples e relativamente fácil. Um desses módulos e talvez o mais importante é o **Express**.

3.2.2 O comando **init** e o arquivo **package.json**

Para inicializar um novo projeto, o npm disponibiliza o comando init, que deverá ser disparado dentro da pasta do projeto:

C:\tcc\Contacts>**npm init**

Esse comando criará o arquivo package.json, o qual conterá, entre outras coisas, o nome do projeto, autor, scripts, etc. E as informações necessárias para que o **npm** possa gerenciar as dependências da aplicação. Ele faz uma série de perguntas que podem ser todas respondidas com Enter para que ele assuma os valores padrão. No final teremos um arquivo com essas informações disponíveis:

```
{
"name": "contacts",
"version": "1.0.0",
"description": "MEAN TCC - 2017",
"main": "server.js",
```

```
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1",
"start": "node server.js"
},
"author": "Manoel Braz Maciel",
"license": "ISC",
}
```

Esse é o conteúdo do arquivo package.json. Esse é um arquivo muito importante e será utilizado durante todo o desenvolvimento do projeto e algumas peculiaridades das suas informações serão comentadas no momento da sua utilização. Por enquanto, só estão fora do padrão do npm, a descrição e o autor, as outras informações foram automaticamente acrescentadas por ele. Para o

nome do projeto ele assume o nome da pasta, em letras minúsculas. O script de test não será utilizado. O script start, permite que seja digitado 'npm start' dentro do diretório raiz do projeto, ao que o npm responde executando 'node server.js', o que significa subir o servidor. O servidor é interrompido normalmente com um Ctrl + C.

3.2.3 Instalando o Express

O **Express** é um dos módulos facilmente instaláveis com o **npm**, com apenas uma linha de comando, estará disponível o auxiliar necessário para prover várias funcionalidades da aplicação, dentre elas, as **variáveis de ambiente**, os **middlewares** e as **rotas.** Cada uma delas será criada, configurada e detalhada oportunamente, e tornarão esse servidor básico altamente operacional. O Express é instalado com o seguinte comando:

C:\tcc\Contacts>***npm install express*** *--save*

Esse comando instalará o módulo '**express**' no projeto. O **npm** criará uma pasta chamada **node_modules**, e nela instalará uma série de dependências, inclusive o express. O --save instruí o npm para incluir o Express nas dependências do projeto no arquivo package.json, para que no futuro, se a aplicação for colocada em produção, o comando 'npm install' saberá que deverá instalar esse módulo na aplicação. Dessa forma não haverá a necessidade de levar a pasta **node_modules** para o ambiente de produção. O que é inclusive recomendado para evitar incompatibilidades por acaso existentes dentro de ambientes de desenvolvimento e produção diferentes. Tendo o **express**

instalado no projeto, a aplicação pode torna-se um pouco mais elaborada, configurando-a para servir inicialmente, páginas estáticas.

3.2.4 A pasta config e o arquivo express.js

Será criada uma pasta de configuração dentro da pasta raiz do projeto chamada ‘config’ e dentro dela será criado um arquivo chamado express.js, com o seguinte código:

```
// ./config/express.js

const express = require('express');
```



```
const app = express();

// enviroment variables
app.set('host', 'localhost');
app.set('port', 3002);

// middlewares
app.use(express.static('./public'));

module.exports = app;
```

O código desse arquivo será executado quando ele for carregado pelo arquivo server.js. Ele por sua vez, através da função require, carrega o módulo ‘express’ na constante **express.** Esse código é executado em seguida, carregando o resultado na constante **app**. Assim, a constante app conterá o express executando como uma função. Em seguida são configuradas as variáveis de ambiente host como localhost e port como 3002, através da função set de app: app.set(‘host’, ‘localhost’) e app.set(‘port’, 3002). Ou seja, é o Express que é configurado com essas variáveis. Mais a frente será mostrado que é o próprio express que responderá por requisições neste endereço.

Com a linha: app.use(express.static(‘./public’), é utilizando um middleware interno do express que tem como função dentro da aplicação, servir páginas estáticas, passando como parâmetro o diretório em que estas páginas estarão hospedadas. E, finalmente é exportada a constante app, contendo essas

configurações, ela será recebida no arquivo server.js oportunamente. A seguir o servidor outrora simples será incrementado para receber essas informações contidas em app. Essa constante será de suma importância no decorrer do desenvolvimento do projeto. Praticamente toda a parte de "backend" da aplicação girará em torno dela.

3.2.5 As alterações no arquivo server.js

As seguintes alterações serão realizadas no arquivo 'server.js':

```
// server.js

const http = require('http');
const app = require('./config/express');
```



```
const host = app.get('host');
const port = app.get('port');

const server = http.createServer(app);

server.listen(port, host, function(){
console.log('Servidor ... Respondendo em: %s:%s!', host, port);
});
```

A primeira linha não mudou. Na segunda linha, uma nova constante **app** recebe através da função require a constante app, exportada pelo arquivo express.js, que foi criado anteriormente na pasta config. Em seguida as constantes address e port recebem através da função app.get o nome de host e a porta configuradas com a função app.set, no arquivo de configuração, em: './config/express.js'. Então app é passada como parâmetro para a função 'createServer' e se comportará como um listener escutando o endereço e porta especificados e respondendo com os arquivos estáticos que se encontrarem na pasta './public'. O './' significa o diretório em que o servidor foi executado. Houve uma ligeira evolução do nosso servidor com a adição do Express. Ele, como foi dito um pouco acima, responde agora páginas estáticas. É o próprio Express que está aguardando por solicitações no endereço e porta configurados e servindo em resposta a estas solicitações, páginas estáticas através de um dos seus middlewares internos. Ou seja, é o próprio Express que está desempenhando essas importantes funções. A pasta public e o arquivo estático ainda não foram criados. É isso que

será feito em seguida.

3.2.6 – Criando o arquivo estático de resposta: index.html

Vamos agora criar a pasta public e dentro dela o arquivo index.html com o seguinte código:

```
<!DOCTYPE html>
<html>

<head>
 <meta charset="UTF-8">
 <title>Contacts</title>
</head>

<body class="container">
```



```
 <div class="jumbotron">
  <h1>Contacts</h1>
 </div>

 <footer class="jumbotron">
 &copy; Manoel Braz Maciel – MEAN TCC - 2017
 </footer>
</body>

</html>
```

Se o servidor estiver em execução, deverá ser derrubado. Em seguida, o servidor deverá ser executado novamente com:

C:\tcc\Contacts>**npm start**

A resposta no console, refletirá as mudanças no comando e configuração de porta que foi aplicado ao servidor:

```
> contacts@1.0.0 start C:\tcc\Contacts
> node server.js

Servidor ... Respondendo em: localhost: 3002!
```

O servidor outrora extremamente simples, agora ligeiramente aperfeiçoado, responde em: http://localhost:3002 e http://localhost:3002/index.html, com o conteúdo do arquivo index.html, implantado na pasta public. Mais uma vez, espero que fique bem claro que, um site completo pode ser disponibilizado desta forma, colocando tudo dentro da pasta public, com toda a estrutura necessária como acontece quando é implantado um site estático no Apache, por exemplo. Mas, o nosso aplicativo ainda vai evoluir mais um pouco. Será criado no próximo tópico um backend um pouco mais complexo com *model, controller e route*, criando uma API que mais tarde será consumida pelo frontend do projeto com o Angular.

3.3 Criação da API

Nesta parte do desenvolvimento da aplicação que demonstra a integração dos componentes da stack MEAN, será criada uma API com o Express, que será posteriormente consumida pelo Angular. Será desenvolvido um CRUD de Contacts persistido inicialmente em memória RAM. Será primeiramente criado um backend meio que fictício, com o *Model* representado por um *array* de objetos em memória representando o banco de dados. Depois o *Controller* – que mais tarde terá que sofrer modificações para acessar os dados no banco. A *Route* porém, não sofrerá nenhuma modificação posterior, pois os URLs de acesso serão os mesmos. Para conter essa estrutura de backend será criada na pasta raiz do projeto uma outra pasta chamada *app*.

3.3.1 O Model

Depois de criada a pasta app na raiz do projeto, dentro da pasta app será criada a pasta model. E, dentro dela, um arquivo chamado 'contacts.js', com o seguinte código:

```
// ./app/model/contacts.js

var contacts = [
{ '_id': 1, 'name': 'Contact 1', 'email': 'contact1@domain.com'
}, { '_id': 2, 'name': 'Contact 2', 'email':
'contact2@domain.com' }, { '_id': 3, 'name': 'Contact 3',
'email': 'contact3@domain.com' }, { '_id': 4, 'name': 'Contact
4', 'email': 'contact4@domain.com' }
];

module.exports = contacts;
```

Este código é um array de objetos que está, neste momento, representando os dados da aplicação, e será acessado pelo controller simulando um acesso ao banco de dados do sistema. Os dados serão manipulados na memória RAM do computador, até o momento da integração com o banco de dados MongoDB. A última linha module.exports = contacts, é que disponibiliza o array de contatos. Esse array será posteriormente recebido no controller e manipulado por ele.

3.3.2 O Controller

Dentro da pasta app, deverá ser criada uma pasta controller e, dentro desta deverá ser criado um arquivo que também chamaremos de 'contacts.js' que deverá conter o seguinte código:

```
// ./app/controller/contacts.js

module.exports = function () {
```



```
var contacts = require('../model/contacts');
var id = contacts.length;

var controller = {};


controller.create = function (req, res, next) {
 var newContact = {};
 newContact._id = ++id;
 newContact.name = req.body.name;
 newContact.email = req.body.email;
 contacts.push(newContact);
 res.statusCode = 201;
 res.send('Contact created successfully ... !').end();
}


controller.retrieveAll = function (req, res, next) {
 res.statusCode = 200;
 res.json(contacts);
};
```

```
controller.retrieveOne = function (req, res, next) {
 var idContact = req.params._id;
 var contact = contacts.filter(function (contact) {
 return idContact == contact._id;
 })[0];
 if (contact) {
 res.statusCode = 200;
 res.json(contact);
 res.end();
 } else {
 res.statusCode = 404;
 res.end('Contact not found ... !');
 }
};

controller.update = function (req, res, next) {
```



```
 var idContact = req.params._id;
 var contactExists = contacts.filter(function (contact) {
return idContact == contact._id;
 })[0];
 if (contactExists) {
 var contact = req.body;
 contact._id = req.params._id;
 contacts = contacts.map(function (contactMapped) {
 if (contactMapped._id == contact._id) {
 contactMapped.name = contact.name;
 contactMapped.email = contact.email;
 }
 return contactMapped;
 });
 res.statusCode = 200;
 res.send('Contact updated successfully ... !').end();  }
else {
 res.statusCode = 404;
 res.end('Contact not found ... !');
 }
}
```

```
controller.delete = function (req, res, next) {
 var idContact = req.params._id;
 contacts = contacts.filter(function (contactFiltered) {
return idContact != contactFiltered._id;
 });
 res.statusCode = 200;
 res.send('Contact deleted successfully ... !').end();
};

return controller;

};
```

Uma porção de código desse arquivo será executada para cada operação de CRUD invocada: *create*, *retrieveAll*, *retrieveOne*, *update* e *delet*e. Este código posteriormente será substituído por operações no banco de dados, quando o projeto for integrado com o MongoDB.

3.3.3 O Route

Dentro da pasta app, deverá ser criada ainda uma outra pasta chamada route e, dentro desta deverá ser criado um arquivo também chamado ‘contacts.js’ que conterá o seguinte código:

```
// ./app/route/contacts.js

module.exports = function (app) {

var controller = require('../controller/contacts')();

app.route('/contacts')
 .post(controller.create) // cria um novo contato
 .get(controller.retrieveAll); // recupera todos os contatos

app.route('/contacts/:_id')
 .get(controller.retrieveOne) // recupera um único contato pelo seu id
.put(controller.update) // atualiza os campos de um contato pelo seu id
.delete(controller.delete); // exclui um contato pelo seu id

}
```

Esse código, que é uma função, recebe como parâmetro a constante app – foi dito anteriormente que o backend do projeto giraria praticamente todo em torno dessa constante. – que é, em última análise, o próprio Express. Ele carrega o arquivo contacts.js da pasta controller numa variável também chamada de controller e o executa. Usa então a função route do express, definindo que, para cada uma das duas URLs existentes na aplicação: `/contacts` e `/contacts/:_id`, haverá os verbos http e, cada um destes verbos: *post*, *get*, *put* e *delete*, quando chamado executará o código correspondente no controller. Cada uma dessas rotas será posteriormente chamada pelo código do Angular, no frontend. Este arquivo não sofrerá alterações na integração final com o MongoDB.

3.3.4 As mudanças no arquivo express.js

As seguintes mudanças deverão ser realizadas no arquivo ./config/express.js:

```
// ./config/express.js

const express = require('express');
const home = require('../app/route/contacts.js');
```



```
const app = express();

app.set('host', 'localhost');
app.set('port', 3003); // só muda o número da porta

// middlewares
app.use(express.static('./public'));

home(app);

module.exports = app;
```

As linhas de código destacadas indicam as mudanças iniciais necessárias no arquivo config.js para que a API funcione, ainda de forma precária. O arquivo das rotas será carregado na constante *home* e na seção de *middlewares* será executado, recebendo como parâmetro a constante app, a qual contém o próprio Express. Durante essa execução o express será configurado com as rotas da API, essas por sua vez já saberão como responder, ao serem solicitadas, executando a porção de código

correspondente do arquivo contacts.js da pasta controller, o qual acessa os dados no arquivo contacts.js da pasta model. A precariedade existe por que alguns verbos ainda não funcionam. Isso será devidamente solucionado mais a frente.

3.3.5 Os testes

Após o servidor ser derrubado com mais um Crtl + C e executado novamente através do comando correspondente:

C:\tcc\Contacts>**npm start**

A API poderá ser testada utilizando inicialmente um navegador para o verbo

get: http://localhost:3003/contatcts

O navegador renderizará o array de objetos com todos os quatro contatos existentes. Se, além disso, for passado um id especifico como parâmetro na url:

http://localhost:3003/contatcts/ 3

Apenas o contato de id 3 será retornado para o navegador.

3.3.6 Ainda testes: o aplicativo Postman

O aplicativo Postman poderá ser utilizado para testar o restante dos verbos. Mas por enquanto só o delete está disponível. O post e o put ainda não funcionam. Escolhendo-se o verbo *delete*, com a url: localhost:3003/contacts/x. Onde o x é o id do contato que se pretende deletar. Se for indicado 3 no lugar do x, o contato de id 3 será deletado. Uma mensagem de contato deletado com sucesso deve aparecer ou algum erro que por acaso ocorra.

3.3.7 Os verbos *post* e *put* e o *middleware body-parser*

Os verbos 'post' e 'put' no entanto, exigem mais algumas mudanças no arquivo de configuração do projeto, o express.js. Esses verbos necessitam que os dados para a criação ou atualização de um contato, sejam enviados no corpo da requisição, porém o servidor do projeto ainda não está preparado para lidar com isso. Para isso é necessário ativar o middleware body parser. Isto será realizado com mais algumas mudanças no arquivo de configuração da aplicação: o ./config/express.js.

3.3.8 Mais mudanças no arquivo express.js

Mais três linhas serão acrescentadas no arquivo express.js na pasta config, para que o servidor seja configurado para receber dados no corpo da requisição.

```
// ./config/express.js

const express = require('express');
const bodyParser = require('body-parser');
const home = require('../app/route/contatos.js')
const app = express();

app.set('host', 'localhost');
app.set('port', 3003);

// middlewares
app.use(express.static('./public'));
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
home(app);

module.exports = app;
```


Primeiro é necessário carregar o módulo externo *body-parser*, e isso é feito em uma constante chamada bodyParser, mais uma vez através da função require. Depois as duas linhas destacadas na seção de middleware, farão com que o servidor seja capaz de receber dados no formato json no corpo da requisição. Agora é possível testar os verbos *post* e *put*, configurando o aplicativo Postman para enviar os dados através da opção 'x-www-form-urlencoded'. Preenchendo as chaves e valores correspondentes antes de enviar a requisição. As urls serão: http://localhost:3003/contatcs, no caso do *post* e no caso do verbo *put* será: http://localhost:3003/contac t s/x, onde o x será substituído pelo id do contato que será atualizado. O resultado deverá ser um contato criado ou atualizado com sucesso, ou um erro, caso ele ocorra, completando assim os testes da API do projeto. Na próxima seção essa API será consumida pelo Angular no frontend do projeto.

3.4 O Angular

A partir deste ponto do estudo da integração da stack MEAN o Angular consumirá a API desenvolvida no tópico anterior. Mas o angular é uma dependência do lado do cliente e para lidar com isso no projeto será utilizado o Bower. Ele será o gerenciador de dependências do lado do cliente. Por isso será falado um pouco sobre ele antes de se entrar na integração do Angular.

3.4.1 O Bower

O Bower também será instalado como um módulo do Node através do npm com o seguinte comando:

C:\tcc\Contacts>**npm install -g bower --save**

Assim será obtido o gerenciador de dependências do lado cliente. O -g indica que ele será instalado de forma global e ficará disponível para todo o sistema. O --save instruí o npm para incluir o Bower nas dependências do projeto no arquivo package.json. Através dele será gerenciada as dependências do lado do cliente e será obtido, entre outras coisas, o Angular que vai proporcionar a ligação entre o cliente e o servidor, manipulando os eventos que desencadeiam a mágica interatividade do sistema com o usuário final. Antes disso porém, é necessários criar um arquivo de configuração na pasta raiz do projeto chamado .bowerrc, ele conterá a seguinte linha de código:

```
{ "directory": "public/lib" }
```


Essa linha indica o diretório onde o Bower colocará os arquivos de dependências do lado do cliente. Essa pasta não precisa ser criada, o próprio Bower se encarregará de criá-la no momento da instalação das dependências. Feito isso será criado o arquivo de configuração do Bower com o comando init:

C:\tcc\Contacts>**bower init**

Respondendo a todas as perguntas com enter, será criado na pasta raiz do projeto, um arquivo chamado bower.json, com quase as mesmas informações do arquivo package.json, do qual inclusive ele obtém alguns dados. Agora o projeto está pronto para se começar a instalar as dependências do

lado do cliente.

3.4.2 Instalando o Angular e outras dependências

Agora que o Bower já está instalado e configurado no projeto, com um comando apenas se instala as dependências necessárias para o lado cliente da aplicação:

C:\tcc\Contacts>**bower install angular angular-route bootstrap** --save

Com esta linha de comando o Bower instalará essas dependências e ainda o jQuery. Tudo será disponibilizado na pasta ‘./public/lib’, criada automaticamente, conforme foi configurado no arquivo ‘.bowerrc’. O angular-route será utilizado na criação das rotas do lado do cliente e o bootstrap ajudará a criar uma interface melhorada sem muito esforço. Agora todos esses arquivos precisam ser importados no código do arquivo index.html, com as seguintes mudanças:

```
<!DOCTYPE html>
<html>

<head>
<meta charset="UTF-8">
 <title>Contacts – Angular</title>
 <link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.min.css">
</head>

<body class="container">
```



```
 <div class="jumbotron">
 <h3>Contacts</h3>
 <h6>Versão 3.0.0</h6>
 </div>

 <footer>
 &copy; Manoel Braz Maciel - TCC 2017
 </footer>

 <!-- scripts importados -->
<script src="lib/angular/angular.min.js"></script>
```

```
<script src="lib/angular-route/angular-route.min.js"></script>

</body>

</html>
```

Para testar se tudo está funcionando corretamente, após realizar uma pequena mudança no número da porta no arquivo de configuração, derrubar e subir novamente o servidor, basta acessar o endereço:

http://localhost:3004/

Com o servidor em execução, claro. Pedir para acessar o código fonte da página. E todos os links devem remeter ao código da dependência respectiva, conforme a figura 1 abaixo.

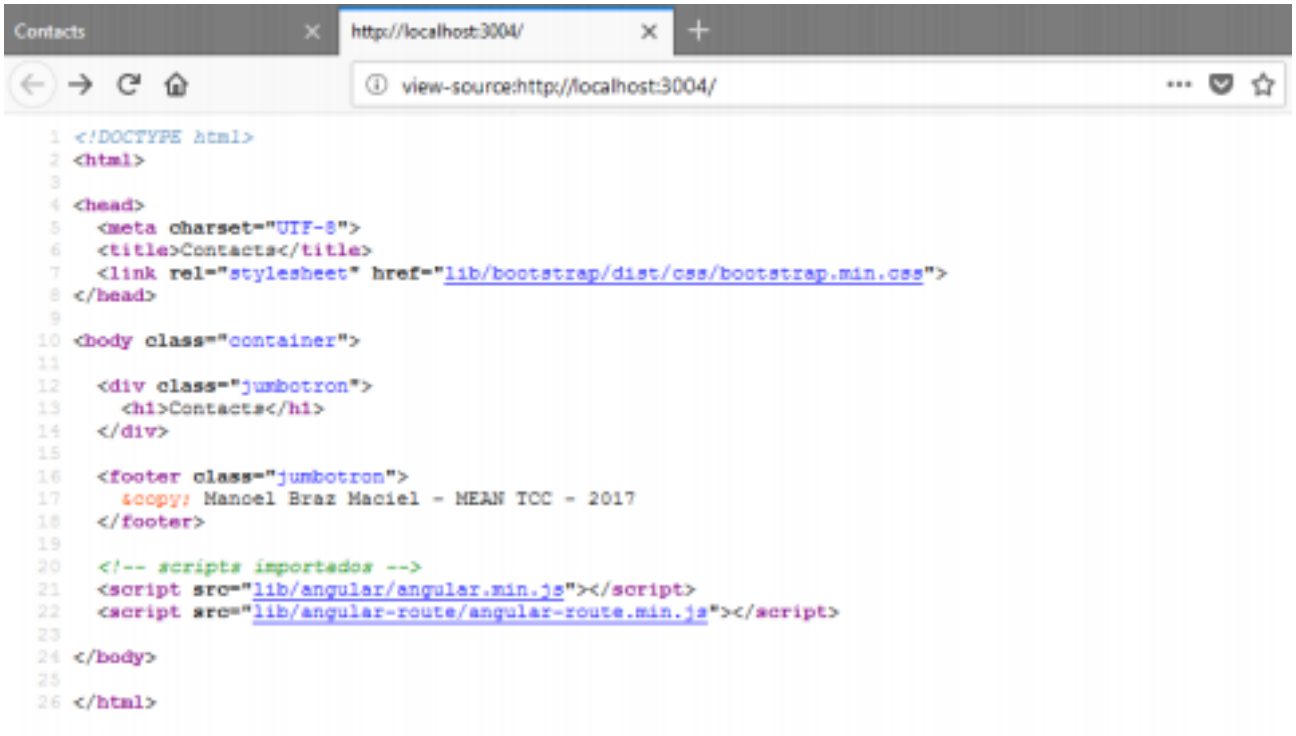

Figura 1 – O código-fonte da página inicial visto no navegador

Com tudo funcionando perfeitamente, será realizada a integração do Angular propriamente dita.

3.4.4 Integrando com o Angular

Para realizar a integração do Angular com o restante do sistema será criada dentro da pasta *public* uma pasta chamada *js*. Essa pasta conterá uma estrutura muito parecida com a que foi criada no lado do servidor. Primeiramente erá criado um arquivo chamado *app.js* nessa pasta, com o

seguinte código:

3.4.4.1 O módulo principal

```
// ./public/js/app.js

angular.module("contact", ['ngRoute']);
```

Esse código cria um módulo do Angular chamado ‘contact’, e injeta o módulo *ngRoute* – que será utilizado para configurar as rotas do lado cliente – como uma dependência desse módulo.

3.4.4.2 A configuração das rotas

Em seguida deve ser criada dentro da pasta *js*, uma outra pasta chamada *route* e dentro dela um arquivo chamado *configRoute.js* com o seguinte código:

```
// ./public/js/route/configRoute.js

angular.module('contact')

.config(function ($routeProvider, $locationProvider) {

$locationProvider.hashPrefix('');

$routeProvider.when('/contacts', {
templateUrl: 'partials/contacts.html',
controller: 'contactsCtrl'
});

$routeProvider.when('/contact', {
templateUrl: 'partials/contact.html',
controller: 'contactCtrl'
});

$routeProvider.when('/contact/:id', {
templateUrl: 'partials/contact.html',
controller: 'contactCtrl'
});

$routeProvider.otherwise({ redirectTo: '/contacts' });
```

```
});
```

Esse código cria as rotas e define os templates para cada uma e o *controller* que será carregado junto com esse template. São três rotas, dois templates, dois controllers e a rota padrão para qualquer outro endereço que for acessado e não satisfaça nenhuma das rotas anteriores.

3.4.4.3 Os códigos dos controllers

Em seguida será criada dentro da pasta js, uma outra pasta chamada ‘controller’ e dentro dela um arquivo chamado *contactsCtrl* com o seguinte código:

```
// ./public/js/controller/contactsCtrl.js

angular.module("contact")

.controller("contactsCtrl", function ($scope, $http) {

$scope.contacts = [];
$scope.message = {};

$scope.remove = function (contact) {
var url = '/contacts/' + contact._id;
$http.delete(url).then(loadContacts, error);
}

var loadContacts = function () {
$http.get("/contacts").then(success, error);
}

var success = function (contact) {
$scope.contacts = contact.data;
}

var error = function (error) {
console.log(error);
}

loadContacts();
```

```
});
```

e ainda dentro da pasta *controller*, deverá ser criado um outro arquivo chamado *contactCtrl.js* com o seguinte código:

```
// ./public/js/controller/contactCtrl.js
```



```
angular.module('contact')

.controller('contactCtrl', function ($scope, $routeParams, $http)
{ $scope.contact = {};
$scope.message = {};
$scope.operation = 'Editing an existing contact ...';

var init = function (id) {
var url = '/contacts/' + id;
if (id) $http.get(url).then(success, error);
else
$scope.operation = 'Creating a new contact ...';
}
success = function (contact) {
$scope.contact = contact.data;
}
error = function (error) {
$scope.message = "Could not perform operation ...!";
console.log(error);
}

$scope.salvar = function (contact) {
if (!contact._id) {
criarNovoContato(contact);
} else {
atualizarContato(contact);
}
}
```

```
var criarNovoContato = function (contact) {
var url = '/contacts';
$http.post(url, contact).then(listar, error);
};

const atualizarContato = function (contact) {
var url = '/contacts/' + contact._id;

$http.put(url, contact).then(listar, error);
};
```



```
listar = function () {
location.assign("#/contacts");
}

init($routeParams.id);

});
```

Esses são os códigos responsáveis pela interação entre frontend e backend. Para cada operação solicitada pelo usuário do sistema, os controllers vão disparar para o backend as solicitações para satisfazer as necessidades dessa operação, se for o caso. Isso será ainda detalhado mais a frente.

3.4.4.4 Criando os templates

Agora dentro da pasta *public* deverá ser criada uma pasta chamada *partials* e dentro dela primeiro será colocado um dos templates chamado *contatcs.html* com o seguinte código:

```
<!-- ./public/partials/contacts.html -->

<div class="alert alert-success" ng-if="message.text &&
contacts.length"> {{ message.text }}
</div>

<div class="alert alert-danger"
ng-if="!contacts.length"> No contacts registered ... !
```

```
</div>

<a href="#/contact" class="btn btn-primary">New
contact</a> <table class="table table-striped
table-responsive">
<thead>
<th>Name</th>
<th>Email</th>
<th>Action</th>
</thead>
<tbody>
<tr ng-repeat="contact in contacts">
```



```
<td>
<a href="#/contact/{{contact._id}}" title="Click to
edit">{{contact.name}}</a>
</td>
<td>
<a href="#/contact/{{contact._id}}" title="Click to
edit">{{contact.email}}</a>
</td>
<td>
<button class="btn btn-danger"
ng-click="remove(contact)">Delete</button> </td>
</tr>
</tbody>
</table>
<p class="text-info" ng-show="contacts.length > 0">Total contacts:
{{ contacts.length }}</p>
```

Essa parcial será carregada, junto com seu *controller*, quando o aplicativo estiver trabalhando com a *listagem* dos contatos. E ainda, dentro da pasta *partials*, deverá ser criado um outro arquivo chamado *contact.html*, com o seguinte código:

```
<!-- ./public/partials/contact.html -->

<div ng-class="message.class" ng-if="message">
{{ message.text }}
```

```
</div>
<label class="text-info">{{operation}}</label>
<form class="form-horizontal"
ng-submit="salvar(contact)"> <div class="form-group">
<label class="col-sm-2 control-label">Name</label>
<div class="col-sm-10">
<input type="text" class="form-control"
ng-model="contact.name" placeholder="Full name ... ">
</div>
</div>
<div class="form-group">
<label class="col-sm-2 control-label">Email</label>
<div class="col-sm-10">
```



```
<input type="email" class="form-control"
ng-model="contact.email" placeholder="Email ... ">
</div>
</div>
<div class="form-group">
<div class="col-sm-offset-2 col-sm-10">
<button type="submit" class="btn btn-danger">Save</button>
<a href="#/contacts" class="btn btn-primary">Contact
list</a> </div>
</div>
</form>
```

Esse arquivo será carregado com o seu controller, quando o sistema estiver trabalhando com um único contato; ou seja, na *criação* de um novo contato ou na *edição* de um contato existente.

3.4.4.5 As mudanças no index.html

E para completar essa integração, se faz ainda necessário realizar algumas mudanças no arquivo index.htm, para que ele possa interagir com os scripts importados do Angular e os scripts locais criados para realizar essa interação. Segue em destaque as mudanças necessárias no index.html:

```
<!DOCTYPE html>
```

```
<html ng-app="contact">

<head>
<meta charset="UTF-8">
<title>Contacts</title>
<link rel="stylesheet"
href="lib/bootstrap/dist/css/bootstrap.min.css"> </head>

<body class="container">

<div class="jumbotron">
<h1>Contacts</h1>
</div>
```



```
<div ng-view class="jumbotron">
<!-- renderização das partials -->
</div>

<footer class="jumbotron">
&copy; Manoel Braz Maciel - TCC MEAN - 2017
</footer>

<!-- scripts importados -->
<script src="lib/angular/angular.min.js"></script>
<script src="lib/angular-route/angular-route.min.js"></script>

<!-- scripts locais -->
<script src="js/app.js"></script>
<script src="js/route/configRoute.js"></script>
<script src="js/controller/contactsCtrl.js"></script>
<script src="js/controller/contactCtrl.js"></script>

</body>

</html>
```

Com o servidor em execução, o acesso com um navegador ao endereço:

http:\\localhost:3004

Carregará a página inicial index.html, trazendo para o *frontend* todo o conteúdo disponível na pasta

public e referenciado no código da página. A diretiva *ng-app* indica que todo o código envolvido pela *tag* html agora será gerenciado pelo Angular, através do arquivo *app.js* que define o módulo contact. O arquivo configRoute.js define as rotas que estarão disponíveis através do artefato $routeProvider, configurando o template e o controller que será carregado para cada rota. Esses templates serão carregados na tag gerenciada pela diretiva *ng-view*. A página index.html é carregada uma única vez como uma SPA (*Single Page Application*) e as mudanças subsequentes ocorrerão apenas na área gerenciada pela diretiva *ng-view*. Para cada interação realizada pelo usuário final da aplicação, o controller correspondente acessará o backend quando necessário, solicitando que essa operação seja realiza, recebendo o resultado quando for o caso, e atualizando a página somente na medida do estritamente necessário. Toda essa interação é possível graças ao mecanismo de *two way-databind* do Angular através do seu objeto global *$scope* que funciona como uma cola entre a

View e o Model, no lado do cliente. Para testar a aplicação basta acessar as funções de CRUD de contatos do sistema, que são praticamente intuitivas. A operação de edição – não tão intuitiva – recebe uma dica ao passar o mouse sobre os dados de um contato existente. No próximo tópico será abordado a integração final com o MongoDB. Quando a aplicação deixará de consumir objetos mantidos na memória RAM, para realizar a persistência no banco de dados.

3.5 O MongoDB

O MongoDB é um SGBD NoSQL que disponibilizará e persistirá os dados necessários para o funcionamento da aplicação, sem a necessidade de normalização. Provendo dados no lado servidor e cliente com praticamente a mesma disposição, sem a necessidade de qualquer processo de transformação. É tudo praticamente JSON. E a codificação é altamente facilitada pelo uso da biblioteca mongoose. O MongoDB já deverá estar instalado na máquina que rodará o servidor. A instalação é simples basta seguir as orientações do site oficial para cada sistema operacional. Para integrar o MongoDB ao restante do sistema será instalado o Mongo e o mongoose na aplicação com o npm, será realizada algumas mudanças no backend da aplicação: será trocado o model e o controller, será adicionado um arquivo de configuração do banco e realizada uma pequena mudança no servidor. Isso tudo será detalhado daqui em diante.

3.5.1 Instalando o MongoDB e o mongoose

O mongodb e o mongoose são mais dois módulos facilmente instaláveis com apenas uma linha de comando do npm:

C:\tcc\Contacts>**npm install mongodb mongoose** --save

Após a instalação dos dois últimos módulos necessários para a integração completa da stack MEAN, o arquivo package.json apresentará o seguinte conteúdo:

```
{
"name": "contacts",
"version": "1.0.0",
"description": "MEAN TCC",
"main": "server.js",
"scripts": {
```



```
"test": "echo \"Error: no test specified\" && exit 1",
"start": "node server.js"
},
"author": "Manoel Braz Maciel",
"license": "ISC",
"dependencies": {
"express": "^4.16.2",
"mongodb": "^3.0.0-rc0",
"mongoose": "^4.13.5"
}
}
```

As dependências foram todas acrescentadas automaticamente pelo npm quando da instalação de cada uma delas. Essa é a versão final do arquivo package.json. A pasta node-modules poderá ser deletada que, com um comando npm install, o npm vai instalar todas as dependências necessárias para o funcionamento do aplicativo no lado do servidor, criando a pasta node_modules novamente. Será realizado um retorno para o backend para conectar com o MongoDB.

3.5.2 Substituindo o model

O arquivo *contacts.js* da pasta *./app/model*, será substituído pelo arquivo *contact.js* com o

seguinte código:

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

module.exports = function() {
 const schema = mongoose.Schema({
 name: {
 type: String,
 required: true
 },
 email: {
 type: String,
 required: true,
 index: {
 unique: true
```



```
 }
 }
});

return mongoose.model("Contacts", schema);

}
```

Esse arquivo cria o esquema da base de dados, o qual define os campos do documento e seus tipos, bem como as restrições impostas e cada um dos campos. Os dois campos são requeridos, ou seja, não é possível criar um contato sem nome e e-mail. E não poderá haver dois e-mails iguais, o campo de e-mail é utilizado como índice, e por isso é único. Todos esses requesitos são proporcionados pelo uso da biblioteca mongoose, uma vez que o Mongo por si só não possui essas características. Mais uma vez destacando a modularidade do desenvolvimento. As coisas são acrescentadas conforme as necessidades de cada aplicação.

3.5.3 Trocando o controller

O código do arquivo *contacts.js* na pasta *./app/controller* deverá ser substituído pelo seguinte código:

```
// ./app/controller/contacts.js

module.exports = function () {

const Contact = require('../model/contact')();

var controller = {};

controller.create = function (req, res) {
var contact = new Contact();

contact.name = req.body.name;
contact.email = req.body.email;

contact.save(function (error) {
if (error)
res.send(error);
res.json({ message: 'Contact criado com sucesso ... !' });
```



```
});
};

controller.retrieveAll = function (req, res) {

Contact.find(function (err, contacts) {
if (err)
res.send(err);
res.json(contacts);
});

};

controller.retrieveOne = function (req, res) {
Contact.findById(req.params._id, function (error, contact) {
if (error) res.send(error);
res.json(contact);
});
};

controller.update = function (req, res) {
```

```
Contact.findById(req.params._id, function (error, contact) {
if (error) res.send(error);
contact.name = req.body.name;
contact.email = req.body.email;
contact.save(function (error) {
if (error) res.send(error);
res.json({ message: 'Contato atualizado com sucesso ... !'
}); });
});
};

controller.delete = function (req, res) {
Contact.remove({
_id: req.params._id
}, function (error) {
if (error) res.send(error);
res.json({ message: 'Contato excluído com sucesso ... !'
}); });
```



```
}

return controller;

}
```

Esse código proverá as operações de CRUD da aplicação diretamente com o banco de dados. Agora a aplicação passará a persistir os dados no Banco de dados MongoDB que será configurado e executado nos tópicos a frente.

3.5.4 Configurando o banco

Na pasta *config* deverá ser adicionado o arquivo *dbMongo.js* com o seguinte código:

```
// ./config/dbMongo.js

const mongoose = require('mongoose');

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

```
mongoose.connect(uri, { useMongoClient: true })
mongoose.Promise = global.Promise

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(error) { console.log(`Mongoose!
${error}`) });

process.on('SIGINT', function() {
mongoose.connection.close(function() {
console.log("Mongoose! desconectado pelo termino da
aplicação") process.exit(0)
})
})

}
```


Esse código utiliza o módulo *mongoose* para prover operações de conexão com o banco de dados e informar sobre os estados desta conexão. Para que as operações com o banco de dados possam ser realizadas, o MongoDB deverá estar instalado e em operação no sistema. Com ele instalado, vamos criar uma pasta na raiz do projeto chamada *data*. Essa pasta armazenará os dados da aplicação. O comando deverá ser executado de uma outra janela de execução, pois o *MongoDB* deverá permanecer em execução durante a operação do aplicativo. O seguinte comando executa o mongo indicando a pasta de armazenamento dos dados:

```
mongod --dbpath C:\tcc\Contacts\data
```

O Mongo responderá com uma série de mensagens, e a seguinte mensagem

```
final: I NETWORK [thread1] waiting for connections on port 27017
```

Significando que ele está pronto para as operações consecutivas necessárias.

3.5.5 Alterando o servidor Node

Uma pequena alteração deverá ser realizada no arquivo server.js

```
// server.js

const http = require('http');
const app = require('./config/express');
require('./config/dbMongo.js')('mongodb://localhost:27017/contact');

const host = app.get('host'); //
const port = app.get('port');

const server = http.createServer(app);

server.listen(port, host, function(){
console.log('MongoDB ... Respondendo em: %s:%s!', host,
port); });
```


Essa linha fará a conexão da aplicação com o Mongo. Uma base de dados chamada *contact* será criada automaticamente. Um documento chamado *contacts* será criado dentro dessa base de dados quando o primeiro contato for adicionado. Antes disso ambos não existem ainda.

3.5.6 Testes finais

Deverá ser realizada mais uma vez uma pequena alteração no número da porta do servidor para 3005. O servidor deverá se derrubado e subido novamente. O acesso ao endereço:

http:\\localhost:3005

Testará mais uma vez as operações do aplicativo. Agora persistindo dados no MongoDB.

Para verificar as alterações no banco de dados poderá ser utilizado o aplicativo de linha de comando do Mongo. Depois de serem cadastrados alguns contatos, os seguintes comandos podem confirmar se as operações estão sendo realizadas mesmo no banco.

C:\Users\Maciel>**mongo**

MongoDB shell version v3.4.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.1

2017-12-11T13:33:47.225-0700 I CONTROL [initandlisten]

> **show dbs**
admin 0.000GB
contact 0.000GB
local 0.000GB

> **use contact**
switched to db contact

O objeto **db** recebe o banco de dados contact.

> **db.contacts.find().pretty()**

Esse comando mostra todos os contatos cadastrados no banco de dados *contact*, no documento *contacts*.

5 RESULTADOS

Ao final dessa empreitada foi criada uma aplicação completa e funcional: um CRUD de Contatos. Um sistema que, possivelmente, pode servir de base para projetos futuros: uma prototipação rápida para uma tomada de decisão, uma apresentação para um cliente ou um projeto comercial mais robusto e confiável. O que certamente, proporcionou um conhecimento razoável do funcionamento da stack MEAN.

6 CONCLUSÃO

Existe amplamente disponível uma infinidade de engines que criam uma aplicação com a stack MEAN sem a necessidade de muita codificação básica por parte do desenvolvedor, basta apenas ir modificando as partes que se fizerem necessárias e, ir conferindo o resultado final. Isso muitas vezes resulta em uma aplicação satisfatória, de um relativamente fácil desenvolvimento. Mas, até um certo ponto essa aplicação pode ser considerada desconhecida. O que, muito

provavelmente, dificultará sua possível evolução e a sempre necessária manutenção. Por outro lado, integrar uma aplicação parte a parte até obter o resultado final desejado, nos proporciona o conhecimento da funcionalidade de cada uma de suas partes e um controle mais efetivo da aplicação como um todo, resultado que certamente, justifica o tempo e trabalho dispendido. Usar uma caixa-preta é muitas vezes cômodo e produtivo, mas o preço disso é pago outras tantas vezes pelo trabalho de ter que desvendar os mistérios da caixa, quando a aplicação desenvolvida começa a apresentar algum problema. Nesse momento, ou seja, quando o sistema começa a apresentar problemas, ter o conhecimento mais profundo de cada uma das suas partes e o conhecimento e controle da integração de todas elas, nos dá certamente uma folgada e bem-vinda confiança, com uma maior possibilidade de aplicar mudanças e obter resultados com uma certa facilidade, o que nesses casos, pode fazer uma grande diferença na hora de dar a necessária manutenção no sistema desenvolvido ou projetar evoluções futuras.

REFERÊNCIAS:

ALMEIDA, Flávio. **MEAN: Full Stack JavaScript para Aplicações Web com MongoDB, Express, Angular e Node.** São Paulo: Casa do Código, 2015.

ANGULAR. Disponível em <https://angularjs.org>. Acesso em 13 dez. 17.

BOWER. Disponível em <https://bower.io>. Acesso em 13 dez. 17.

EXPRESSJS. Disponível em <http://expressjs.com/pt-br>. Acesso em 13 dez. 17. MEJIA, Adrian. **Construindo uma Aplicação E-commerce com MEAN**. Novatec, 2016. MONGODB. Disponível em <https://www.mongodb.com>. Acesso em 13 dez. 17. MORAES, William Bruno. **Construindo Aplicacações com NodeJS**. Novatec, 2015. NODE.JS. Disponível em <https://nodejs.org>. Acesso em 13 dez. 17.

NPMJS. Disponível em <https://www.npmjs.com>. Acesso em 13 dez. 17.

SCHMITZ, Daniel; LIRA, Douglas. **AngularJS na Prática**. Leanpub book, 2016