MEAN Stack: Desenvolvendo Aplicações Web Utilizando Tecnologias Baseadas em JavaScript

Marcio H. Gimenes Bera, Anderson Fernandes Mine, Luiz Fernando B. Lopes

Faculdade Cidade Verde (FCV)
CEP: 87020-320 – Maringá – PR – Brasil

marciobera@hotmail.com, burnes@professorburnes.com,
prof_braga@fcv.edu.br

Abstract. *From the time that the Internet became part of everyday life in the mid-90s, JavaScript is a language that has never ceased to exist. At first, very popularly known for validating forms or provide some type of interactivity with the user, these days it is a trend with regard to agile applications and gain performance. This article discusses the use of JavaScript technologies like NoSQL MongoDB database, the backend framework Express.js, the frontend framework Angular.js and Node.js, and also shows cases of successful companies that is already using such technology.*

Resumo. *Desde a época em que a internet começou a fazer parte do cotidiano, em meados dos anos 90, JavaScript é uma linguagem que nunca deixou de existir. No começo, muito conhecida popularmente por validar formulários ou proporcionar algum tipo de interatividade com o usuário, nos dias de hoje é uma tendência no que diz respeito a aplicações ágeis e ganho de performance. Este artigo aborda a utilização de tecnologias JavaScript, como o banco de dados NoSQL MongoDB, o framework backend Express.js, o framework Angular.js e o Node.js, e também mostra casos de sucesso de empresas que já vem utilizando tal tecnologia.*

1. Introdução

O *JavaScript* é uma das linguagens mais populares do mundo [RedMonk 2015]. É uma linguagem que vem despontando no mercado com uma grande quantidade de ferramentas, eventos e novos desenvolvedores. *JavaScript* é uma linguagem que nativamente já vem interpretadas pelos *browsers*, mas não se limita somente a aplicações web [Rauschmayer 2012].

JavaScript está presente em aplicações para *Smart TV's* [Handstudio Co. 2013], *Smartphones* [Firdaus 2014], também é possível desenvolver *widgets* para sistemas operacionais como o *Windows 8* e *Windows Phone* [Microsoft 2014], além de plugins para softwares gráficos como *Photoshop* [Adobe 2014], *Illustrator* [Adobe 2015] e aplicações Acrobat [Adobe 2012]. Também é utilizado para games no Unity e HTML5. Além disso, conta com *frameworks* para aplicações desktop, como o caso do *AppJS*

ISBN: 978-85-68323-04-5

[Miliani and Benvie 2012]. Enfim, nos dias atuais, *JavaScript* já se encontra presente em diferentes plataformas e dispositivos.

Os benefícios em utilizar uma aplicação baseada em *JavaScript* abrange dois fatores principais: (i) **Escalabilidade**; que se define pela disponibilidade de serviços requisitados a proporcões crescentes de novos usuários; e (ii) **Performance**; o desempenho obtido levando em conta a escalabilidade e recursos consumidos. Outro benefício que justifica a utilização de *JavaScript* é por ser uma linguagem orientada a eventos, fazendo com que suas operações sejam executadas somente quando as mesmas são requisitadas em alguma parte da aplicação.

Existe uma série de *frameworks* baseados em *JavaScript*, para desenvolvimento de aplicações escaláveis web, destaque para o *MEAN Stack*, que é um conjunto de tecnologias *JavaScript*, que visa aproveitar todos os benefícios que o *JavaScript* proporciona. O MEAN trabalha desde o desenvolvimento da interface da aplicação (*fontend*), até o banco de dados e a programação que fica no lado do servidor (*backend*).

Este artigo está dividido da seguinte forma: A Seção 2.1 apresenta os conceitos de *MEAN Stack* e cada tecnologia que o compõe; a Seção 3 apresenta um passo a passo, auxiliando por onde começar a instalação das tecnologias; a Seção 4 apresenta uma aplicação utilizando MEAN; a Seção 5 apresenta projetos atuais que utilizam MEAN e o caso de sucesso do *PayPal*; e por fim, a Seção 6 apresenta uma breve discussão sobre o conteúdo desenvolvido neste artigo.

2. Background

Esta Seção apresenta a fundamentação teórica necessária para compreender as tecnologias que compõe o *MEAN Stack*. Desta forma, é apresentado um breve resumo sobre os conceitos do *MEAN Stack*, e posteriomente, as tecnologias envolvidas.

2.1. *MEAN Stack*

Com a crescente evolução da internet nos últimos anos, as técnolocias escaláveis vieram para acompanhar tal evolução e proporcionar aplicações mais estáveis aos usuários. Assim, tecnologias como Computação em Núvem (*Cloud Computing*) fornecem um meio de prover serviços escaláveis, que proporcionam a confiabilidade das aplicações em relação a demanda de dados e de acessos. Porém, o desempenho da aplicação e consumo de tráfego de dados estão relacionados diretamente com a tecnologia utilizada no desenvolvimento da aplicação.

Neste cenário, algumas tecnologias se destacam, como o *MEAN Stack*, que é a união de tecnologias baseadas em *JavaScript*, com o objetivo de proporcionar aplicações escaláveis e com auto desempenho. A sigla MEAN é oriunda das inicias de tais tecnologias, tal como:

- **M**ongoDB: Banco de dados *NoSQL* baseado em *JavaScript*;
- **E**xpress.js: *Framework* baseado em *JavaScript* para desenvolvimento do backend da aplicação;

ISBN: 978-85-68323-04-5

- **A**ngular.js: *Framework* baseado em *JavaScript* para desenvolvimento do *frontend* da aplicação; e
- **N**ode.js: Servidor baseado em *JavaScript* que interpreta o Express.js no *backend* da aplicação.

Básicamente, a arquitetura MEAN opera da seguinte forma:

- O usuário acessa um aplicação pelo *browser*;
- O *Angular.js* envia a requisição via *REST* para o *Express.js* no formato JSON;
- O Express converte esses dados para o formato BSON e os envia para o *MongoDB*;
- O *MongoDB* retorna os dados BSON para o *Express.js*;
- O *Express.js* converte os dados para JSON e retorna ao *Angular.js*; e
- O *Angular.js* retorna os dados para o usuário.

Todas essas iterações acontecem de maneira rápida e dinâmica. A Figura 1 apresenta uma visão geral da arquitetura MEAN.

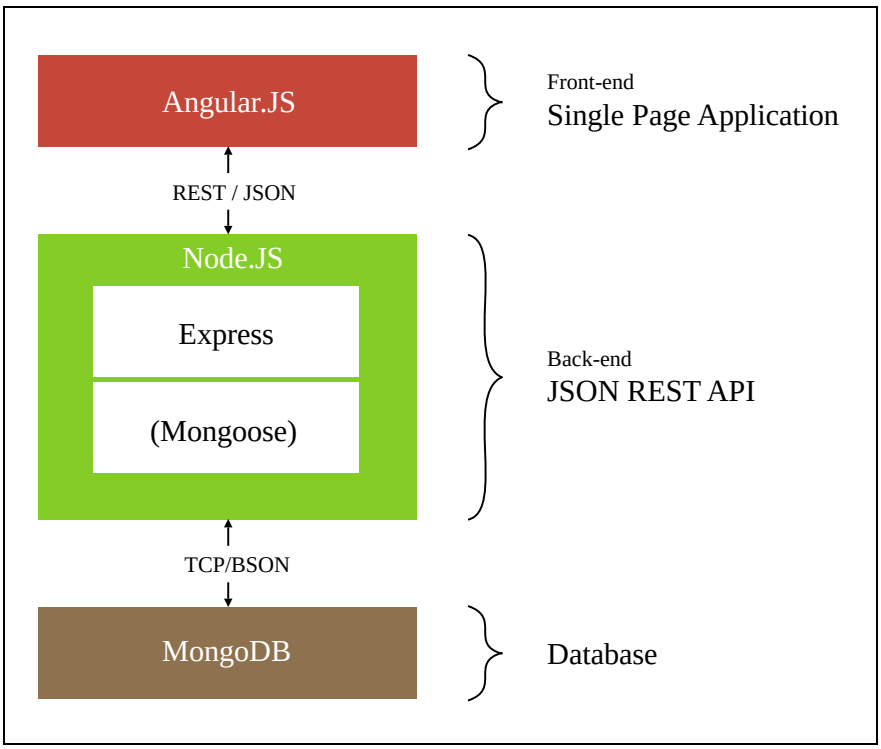


Figura 1: Arquitetura MEAN, adaptada de [Gotthilf 2014].

Nas Subseções a seguir, cada tecnologia será abordada individualmente, para uma compreensão mais ampla.

ISBN: 978-85-68323-04-5

2.1.1. MongoDB

MongoDB, com a nomenclatura derivada da palavra "humongous" (gigantesco), é um banco de dados open-source *NoSQL* (Not Only SQL) que armazena documentos em coleções (*collections*) no formato ***J**ava**S**cript **O**bject **N**otation* (JSON) [Sevilleja and Lloyd 2015] - formato padrão de objetos do *JavaScript*. A família de bases de dados NoSQL, como o próprio nome sugere, não utilizam só SQL, e isso é um dos motivos pelos quais essas bases de dados tem crescido nos últimos anos, principalmente no que diz respeito a facilidade do uso e performance [Bretz and Ihrig 2014].

Os motivos para a utilização do *MongoDB* juntamente com outras tecnologias *JavaScript* se dá pelos fatores:

- O tratamento de dados como objetos JSON;
- Possui uma boa aceitação por parte dos desenvolvedores [DB-Engines 2015];
- Possui um excelente desempenho com grandes quantidades de dados [Politowski and Maran 2014];
- Por proporcionar a execução de comandos *JavaScript* nas consultas; e
- Por possuir uma grande interação com *Node.js*, através dos próprios drivers e com a utilização da api *Mongoose*.

De acordo com [MongoDB 2015], as vantagens da utilização de documentos são:

- Documentos (ou seja, objetos) correspondem aos tipos de dados nativos em muitas linguagens de programação;
- Documentos e *arrays* incorporados, reduzem a necessidade de utilizar *joins*; e
- *Schema* dinâmico, suporta polimorfismo de forma mais fácil.

2.1.2. Express.js

Express.js é um *framework* para a construção de aplicações web baseado em *Node.js* [Sevilleja and Lloyd 2015]. Ele auxilia na organização da aplicação no lado do servidor. De acordo com [StrongLoop 2015] o *Express* é um *framework* de aplicações web, flexível e pequeno para *Node.js*.

Com o *Express* é possível instanciar servidores web e receber requisições HTTP de maneira trivial, além de permitir criar um conjunto de diretórios com uma arquitetura padrão, e também organizar as rotas dos arquivos para as *views* da aplicação [Bonfim and Liang 2014]. Com uma infinidade de métodos utilitários HTTP e *middleware* à sua disposição, é possível criar uma API robusta, de mandeira rápida e fácil [StrongLoop 2015].

ISBN: 978-85-68323-04-5

2.1.3. Angular.js

Angular.js é um *framework JavaScript* MVC para web que acelera o desenvolvimento *frontend*, além de fornecer uma estrutura consistente e escalável, que minimiza o esforço no desenvolvimento de aplicações complexas e de grande porte [Seshadri and Green 2014].

O padrão MVC (Model-View-Controller, ou Modelo-Visão-Controlador) é uma maneira de separar unidades lógicas e responsabilidades no desenvolvimento de aplicações de grande porte [Seshadri and Green 2014]. Uma arquitetura MVC é separada em três partes distintas e modulares. *Model* corresponde aos dados por trás da aplicação, que geralmente são acessados pelo servidor. *View* corresponde as visões disponíveis aos usuários, ou seja, a interface da aplicação. *Controller* corresponde a lógica de negócios da aplicação, que faz a interação dos dados entre a *View* e o *Model*.

Algumas vantagens que o *Angular.js* possui são:

- ***Single Page Application* (SPA)**: Aplicação de página única, ou seja, cada parte da página é carregada de forma independente;
- ***Two-Way Data Biding***: Uma das caracteríscias mais fortes do *Angular* é a sua orientação a dados por meio de *data-binding*. Sempre que algum dado é alterado no lado do servidor (ou seja, no *Model*), a *View* recebe estes dados, e vice-versa (Por isso é conhecido com *data-biding* uni e bidirecional); e
- **Injeção de dependências**: ocorre uma solicitação de dependências de um determinado controlador ou serviço, ao invés de criar uma instância por meio de um operador new ou chamar de forma explicita uma função. Outra parte do seu código ficará responsável por descobrir como criar essas dependências e disponibilizá-las quando forem solicitadas [Seshadri and Green 2014]. Além disso, fornece recursos extras necessários na aplicação de forma transparente ao usuário, de modo que o desenvolvedor somente deverá solicitar um recurso, que será injetado pelo *framework* e ficará disponível para uso.

2.1.4. Node.js

Node.js é um *framework backend JavaScript* para desenvolvimento web que tem se tornado muito popular nos últimos anos. Criado em 2009 por Ryan Dahl com o objetivo de porporcionar aplicações escaláveis, começou a ser aplicada em pequenos projetos de desenvolvimento e, desde então, penetrou no mercado. Hoje, pode ser visto em grandes empresas como Microsoft, eBay, LinkedIn, Yahoo, WalMart, Uber, Oracle, e vários outros [Pinkham 2015].

Node foi construído em cima da *engine* do Google **V8 JavaScript** (a mesma utilizada no *Chrome*), e *libuv* da Joyent. *Node* é uma plataforma guiada a eventos, por consequência ele acaba tendo entradas e saídas não bloqueantes (*non-blocking I/O*) de forma assíncrona. Ou seja, os métodos da aplicação são carregados de maneira independente, e paralelamente.

ISBN: 978-85-68323-04-5

O *Node* trabalha com *web-sockets*, muito utilizado em *chats* e salas de bate-papo para receber notificações em tempo real, *hangouts*, etc. Ou seja, ele deixa um canal aberto, onde as mensagens são compartilhadas e todas as conexões recebem os dados.

Na próxima Seção será demonstrado por onde começar a utilizar o *MEAN Stack*, bem como instalar cada tecnologia que o compõe.

3. Por Onde Começar?

O primeiro passo para começar uma aplicação *MEAN*, é instalando suas tecnologias. O instalador do *Node.js* pode ser encontrado no seu site oficial (http://www.nodejs.org), com versões para Windows, Linux e Macintosh.

O segundo passo é instalar o banco de dados *MongoDB*. O instalador do *MongoDB* pode ser encontrado no seu site oficial (http://www.mongodb.org), com versões para Windows, Linux, Macintosh e Solaris.

Com o *Node.js* e *MongoDB* devidamente instalados, iniciaremos a instalação dos drivers das demais tecnologias por meio de linha de comando, utilizando o *Node Package Manager* (NPM). NPM é um repositório online (https://www.npmjs.com) para publicação de projetos open source em *Node.js*. Por meio da utilização do comando `npm`, é possível interagir com tais projetos, facilitando assim a instalação, gerenciamento de versão e gerenciamento de dependências. No site do NPM é possível buscar pelos projetos publicados, e a descrição de cada projeto instrui a instalá-los por meio do comando `npm` no terminal.

Um diretório de preferência do usuário deve ser escolhido, e posteriormente acessado via terminal. Todas as tecnologias que forem instaladas, estarão contidas em uma pasta nomeada `node_modules` no diretório escolhido. Com o diretório devidamente selecionado, os passos a seguir defirão a configuração do *webserver*.

A instalação do driver do *Express.js* pode ser realizada utilizando a seguinte linha de comando:

```
npm install express
```

A instalação do driver do *MongoDB* pode ser realizada utilizando a seguinte linha de comando:

```
npm install mongodb
```

Em seguida, a instalação do driver que será utilizado para acessar o *MongoDB*. O módulo utilizado neste artigo é o *mongoose* (será falado mais sobre ele adiante), e ele pode ser instalado utilizando a linha de comando, como segue:

```
npm install mongoose
```

Finalmente, a última tecnologia que compoõe o MEAN e que falta instalar é o *Angular.js*. A linha de comando para a sua instalação é:

```
npm install angular
```


ISBN: 978-85-68323-04-5

Antes de iniciar um projeto utilizando estas tecnologias, é importante compreender a importância do gerenciamento de dependências. O gerenciamento de dependências permite identificar as características da aplicação desenvolvida, bem como as tecnologias que estão sendo utilizadas, suas versões, o(s) autor(es) da aplicação, a versão da aplicação, e muitas outras. O arquivo que armazena estas informações é convencionalmente chamado de `package.json`. Através do NPM é possível criá-lo de forma bem intuitíva. A linha de comando para tal, é:

```
npm init
```

Desta forma, incia-se a criação do gerenciador de dependências da aplicação. Após entrar com a linha de comando npm init, o console irá esperar a interação com o usuário para preencher alguns requisitos da aplicação, a ordem é basicamente a seguinte:

- **name**: se refere ao nome da aplicação. Sugestivamente, ele indica o nome *node*. Se esta informação não for preenchida, o nome sugerido será aplicado;
- **version**: referente a versão da aplicação. Sugestivamente, ele indica a versão 1.0.0. Se esta informação não for preenchida, a versão sugerida será aplicada;
- **description**: referente a uma breve descrição da aplicação;
- **entry point**: referente ao arquivo no qual o servidor *Node.js* será inicializado. Sugestivamente, ele indica o nome `index.js`. Se esta informação não for preenchida, a versão sugerida será aplicada;
- **test command**: se refere a determinados *scripts* que podem ser instanciados para ser executado quando rodar o npm test da aplicação. Além disso, o NPM fornece outros comandos, como `start`, `preinstall`, `uninstall` e outros;
- **git repository**: referente ao endereço do repositório git para controle de versão da aplicação;
- **keywords**: se refere as palavras-chave que identificam a aplicação;
- **author**: referente ao(s) nome(s) do(s) autor(es) da aplicação;
- **license**: se refere a licença da aplicação. Sugestivamente, ele indica a sigla ISC, referente a licensa de software grátis, da *Internet Software Consortium*. Se esta informação não for preenchida, a versão sugerida será aplicada; e
- **Is this ok?**: referente aos dados informados anteriormente. Se estiverem corretos, deve-se digitar `yes` ou apertar a tecla `enter`, caso contrário, deve-se digitar `no`.

Desta forma, o arquivo `package.json` deve ser criado corretamente no diretório escolhido para iniciar a aplicação. As próximas Subseções apresentaram maneiras de configurar um *webserver*, utilizando *Node.js* e *Express.js*.

ISBN: 978-85-68323-04-5

3.1. Configurando o WebServer com Node.js

A criação de um *webserver* com *Node.js* é essencial para conhecer a configuração da aplicação. Nesse artigo foi utilizado um *webserver* criado com o *Express.js*, que será apresentado na próxima Subseção. Porém, a configuração com do *webserver* com o *Node.js* serve de base para a compreensão de algumas configurações necessárias. Dessa forma, a Figura 2, adaptada da documentação oficial do site do *Node.js*, apresenta o código necessário para configurar o *webserver*.

Figura 2: Criando um WebServer com Node.js

Na Figura 2, a função `require` inclui o módulo *http*, e a variável `http` recebe esta requisição. Em seguida, a função `createServer`, instanciada a variável `http`, cria o *webserver*. A função `createServer` possui uma função de *callback* como parâmetro, que é um *event loop*, ou seja, essa função ficará executando, até que o servidor seja fechado. Esta função possui dois parâmetros: o `request`, referente ao dado que o browser envia ao *webserver*, e o `response`, referente à resposta que o *webserver* retorna ao *browser*. Assim, se o servidor for criado corretamente, a função `createServer` retornará a função `writeHead`, com o código HTTP 200 para informar que a solicitação obteve sucesso. Em seguida, a função end finaliza as requisições com a mensagem "Olá Mundo". A função `listen` (do inglês, "escultando") estânciada na função `createServer`, fica monitorando a porta 8124. Pode-se executar o *webserver* de duas maneiras. Uma delas é através da linha de comando `npm start`. Já a outra forma, é preciso executar o comando node juntamente o nome do arquivo configurado, nesse caso o arquivo `webserver.js`. A Figura 3 apresenta o retorno que o terminal recebe ao executar a aplicação.

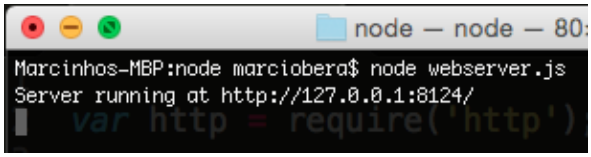


Figura 3: Executando uma Aplicação com *Node.js*

Agora, o usuário pode acessar o *webserver* rodando localmente, através da porta configurada. A Figura 4 apresenta o resultado do acesso ao *webserver* pelo *browser*.

ISBN: 978-85-68323-04-5

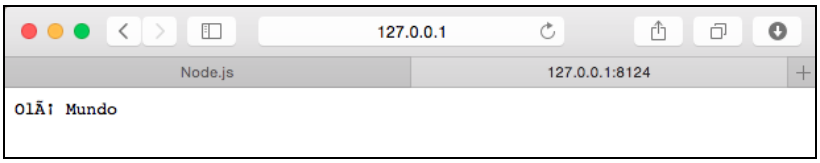


Figura 4: Acessando um *WebServer* com *Node.js* Configurado

Para finalizar a execução do servidor no terminal, basta digitar CTRL + C.

3.2. Configurando o WebServer com Express.js

A Subseção anterior, apresentou como criar um *webserver* com *Node.js*. Essa Seção apresenta como criar um *webserver* utilizando o *Express.js*. A Figura 5 apresenta o código necessário para tal.

Figura 5: Criando um *WebServer* com *Express.js*

Nesta Figura, pode-se perceber que a função `require` está chamando o módulo *express*, onde este se encarrega por toda a criação necessária para o suporte do módulo *http*. Em seguida, a variável `app` recebe a função `express`, para instanciar o *webserver*. Assim, a variável `app` é instanciada com a função `get`, onde está passa no primeiro parâmetro o caminho da raiz da aplicação (*root*), e no segundo parâmetro a função do *event loop*, que recebe as requisições via *browser* pelo parâmetro `req` e envia as respostas pelo parâmetro `res`. Dentro do *event loop* foi criada uma mensagem de retorno no console da aplicação, mes caso no terminal, e a mensagem '*Hello Word :)*' é enviada como resposta para o *browser*. Ainda, a variável `app` está instânciada com a função `listen`, para monitorar a porta 3000. A Figura 6 apresenta o código chamado no terminal para rodar a aplicação.

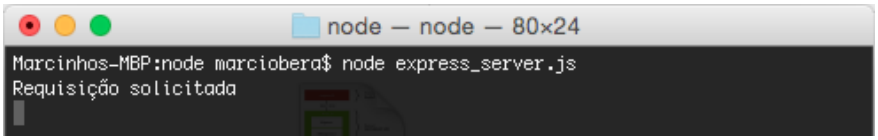


Figura 6: Rodando uma aplicação com com *Express.js*

O comando `node` é utilizado para rodar o arquivo que contém a configuração do *webserver*, nesse caso o arquivo é `express_server.js`. Na Figura 6 ainda, pode-se perceber que após o usuário acessar o endereço do *webserver* com a determinada porta

ISBN: 978-85-68323-04-5

configurada, o console recebe a mensagem que foi declarada no *event loop* do método `get`. A Figura 7 apresenta o retorno que o usuário recebe ao acessar a aplicação.

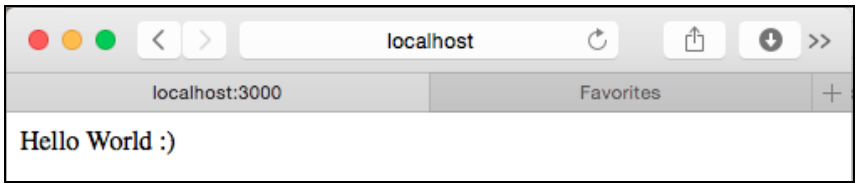


Figura 7: Acessando um *WebServer* com *Express.js* Configurado

As duas formas de configurar o *webserver* são validas, porém utilizando o *Express.js* muita coisa já vem pronta, facilitando assim o desenvolvimento da aplicação. Se o *Express* for instalado globalmente na máquina, é possível criar uma aplicação pronta apenas digitando a linha de comando `express`. A Seção 4 apresenta a construção e configuração de uma aplicação básica utilizando os recursos disponíveis do *MEAN Stack*.

4. Aplicação Utilizando MEAN

Esta Seção apresenta uma aplicação básica do *framework MEAN Stack*, utilizando as técnologias que o compõe. A Seção 3 é fundamental para acompanhar as etapas seguintes desta Seção.

4.1. Arquitetura Inicial da Aplicação

A definição da arquitetura da aplicação é algo muito importante. Os arquivos que podem ser acessados pelos usuários, devem estar em uma pasta pública. Já os arquivos que pertencem ao controle da aplicação, devem estar em um diretório que não pode ser acessado diretamente pelo *browser*. A organização da arquitetura é muito pessoal, e depende do tipo de aplicação realizada. Neste artigo, foi utilizando o padrão MVC, explorando também os benefícios advindos dos módulos, porém com rotas para diretórios diferentes no acesso via *browser*. Por meio do *Express.js*, pode-se criar rotas estáticas, nas quais mapeiam um diretório interno da aplicação, e o *browser* interpreta esta roda de acordo com as definições propostas. A função `static` do *Express.js* é responsável por servir os arquivos estáticos da aplicação. Por exemplo, os módulos baixados por meio do NPM, ficam na pasta `node_modules`, porém, por meio da função `use`, pode-se definir as rotas que serão utilizadas para acessar determinados diretórios, simulando para o *browser* onde estes diretórios estão. Ainda, neste artigo, foi definido que na pasta `public` estarão os arquivos públicos, e na pasta `api` estarão os arquivos de controle da aplicação. A Figura 8 apresenta a arquitetura inicial da aplicação.

ISBN: 978-85-68323-04-5

Figura 8: Arquitetura Inicial da Aplicação

Ainda, na Figura 8, pode-se observar que um arquivo `index.html` foi criado na pasta pública, dentro de uma pasta chamada `views`. Quando o usuário acessa o diretório raiz do *webserver*, a função get envia o arquivo `index.html`, do diretório que foi mapeado, como resposta para o *browser*, por meio da função `sendFile`. O objeto global `dirname` do *Node.js* indica o nome do diretório onde o *script* está sendo executado. Os módulos baixados via NPM, estão dentro do diretório padrão `node_modules`, mas podem ser mapeados como se estivessem no diretório `public/assets`.

4.2. Criando a Interface da Aplicação com *Angular.js*

A aplicação desenvolvida para este artigo, basicamente irá fazer um CRUD (*Create*, *Read*, *Update* e *Delete*) de cadastros de usuários. Primeiramente, a interface estática da aplicação foi criada. As interações da interface foram desenvolvidas com *Angular.js*, e a parte visual da aplicação foi desenvolvida com *Bootstrap*. O módulo do *Bootstrap* foi incluído via NPM. A explicação sobre a construção do template utilizado estão fora do escopo do artigo, porém, seu código fonte está disponibilizado no *GitHub (https://github.com/marciobera/angular-crud)*. A Figura 9 apresenta o template estático que será utilizado na aplicação.

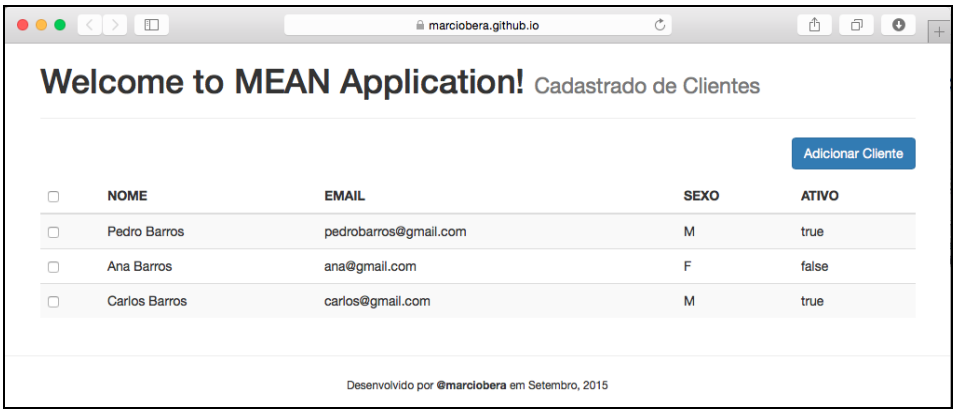


Figura 9: Template Estático da Aplicação

ISBN: 978-85-68323-04-5

Nesta intarface, o arquivo `app.js` contém um *array* de objetos json intitulado "cadastros", onde estão contidos os cadastros dos clientes com os seguintes atributos: `nome`; `email`; `sexo`; e `ativo`. Este objeto é tomado pelo *Angular.js*, e através da diretiva `ng-repeat`, são inseridos na tabela de cadastro de cliente. A Seção 4.5 apresenta a interação do *template* em *Angular.js* com os dados dinâmicos advindos do *MongoDB*.

4.3. Configurando o Banco de Dados da Aplicação com *MongoDB*

4.3.1. Configurando o *Mongoose*

Para configurar corretamente o *MongoDB* em aplicações MEAN, é necessário incluir o módulo *mongoose* no *webserver* da aplicação. Na Seção 3 apresenta como instalá-lo via NPM, porém se o *mongoose* não foi instalado após criar o `package.json`, é necessário instalá-lo utilizando a seguinte linha de comando:

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

A declaração `--save` no final da linha de comando, indica que além de instalar o módulo *mongoose*, ele será salvo nas dependências do arquivo `package.json`. Caso ele já tenha sido instalado anteriormente, não é necessário repetir esta instalação. O módulo *mongoose* deve ser adicionado ao arquivo do *webserver* para que se possa manipular os dados do *MongoDB*. Para tal, é necessário inserir o seguinte código no arquivo do *webserver*:

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

Em seguida, é necessário configurar a conexão ao *MongoDB* via *mongoose*. Para tal, o seguinte código deve ser adicionado no arquivo do *webserver*:

```
mongoose.connect('mongodb://localhost:27017/appmean');
```

A variável `mongoose` criada anteriormente, a fim de requerer o módulo *mongoose*, deve receber a conexão do *MongoDB*. Assim, a URL de conexão deve ser iniciada com o protocolo `mongodb://`, em seguida o endereço padrão do *webserver*, neste caso `localhost`, seguido da porta padrão utilizada pelo *MongoDB*, 27017. Após isso, deve-se inserir o nome da base de dados, neste caso o nome definido foi `appmean`. No *MongoDB* não é necessário criar as bases de dados préviamente, pois elas são criadas de acordo com as *collections* que são inseridas. Exemplos serão apresentados no momento de configurar o model da aplicação.

4.3.2. Configurando o *Body Parser*

Um outro módulo importante e necessário para se trabalhar com manipulação de objetos JSON é o *Body Parser*. Ele tem o objetivo de garantir que os dados enviados sejam objetos JSON. A sua instalação via NPM pode ser realizada da seguinte forma:

```
npm install body-parser --save
```

No arquivo principal do *webserver*, ele deve ser incluído com o seguinte código:

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


ISBN: 978-85-68323-04-5

E mais abaixo, ele deve ser instância-lo à aplicação, estipulando o tipo de dado que se irá trabalhar. Assim, também é necessário incluir o seguinte código:

```
app.use(bodyParser.json());
```

A Figura 10 apresenta a evolução do arquivo *webserver.js* com a inclusão dos códigos acima mencionados.

Figura 10: Evolução do Arquivo *webserver.js*

4.3.3. Definindo o *Model* de Cliente da Aplicação

No diretório `api/models` da aplicação, deve ser criado um arquivo para o *model*, neste caso o `cliente.js`. Para cada *model* da aplicação, um novo arquivo deve ser criado, com suas respectivas configurações. No arquivo `cliente.js` estará contido um modelo padrão para a *collection* de clientes. A Figura 11 apresenta um modelo da *collection* de clientes.

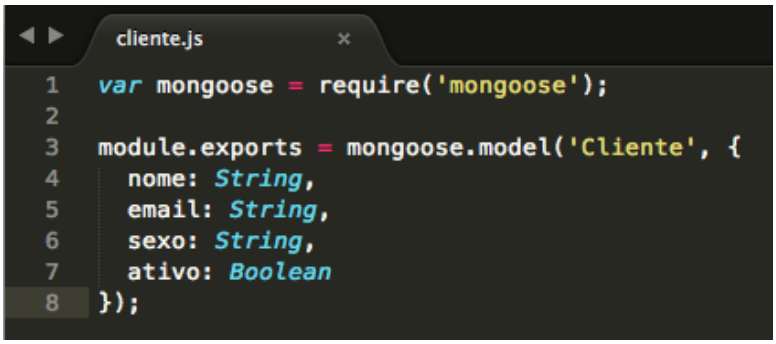


Figura 11: Definindo o Modelo da *Collection* de Clientes

Nesta Figura, pode-se perceber que, primeiramente, é requirido o módulo *mongoose*, para evitar alguma possibilidade de erro de dependência. Em seguida, é exportado um módulo *mongoose*, referente a um modelo de *collection*. A função `model`, tem por objetivo definir um modelo para uma determinada *collection*. Assim, o primeiro parâmetro desta função é o nome da *collection*, e o segundo parâmetro é a sua estrutura. No exemplo da Figura 11, está sendo exportado um módulo *mongoose*, referente a um modelo de *collection*, neste caso o *model* "`Cliente`". Em seguida, está sendo definido a sua estrutura. Assim, a estrutura da *collection* é definida pelo nome,

ISBN: 978-85-68323-04-5

seguido do *BSON Type*. O Tipo de Dados BSON também pode ser referenciado por números. No caso da *collection* "`Cliente`", é a seguinte: `nome`: *String*; `email`: *String*; `sexo`: *String*; e `ativo`: *Boolean*.

4.3.4. Definindo o *Controller* de Cliente da Aplicação

Neste momento, será definido o *controller* deste *model*, onde este irá controlar a manipulação dos dados. Assim, o arquivo de controle deve ser criado no diretório `api/controller`. Neste caso, o nome sugerido foi `cliente-controller.js`. Básicamente, este controle irá realizar as seguintes operações do tipo CRUD:

- **Criar (*Create*)**: deverá incluir um novo cliente no banco de dados;
- **Listar (*Read*)**: deverá listar os clientes cadastrados no banco de dados;
- **Atualizar (*Update*)**: deverá atualizar um determinado cliente existente no banco de dados; e
- **Deletar (*Delete*)**: deverá deletar um ou mais clientes do banco de dados.

A Figura 12 apresenta o *Controller* do *Model* de Clientes da aplicação. Este arquivo, incia-se com a inclusão do *Model* de Clientes, relacionado à variável `Cliente`.

ISBN: 978-85-68323-04-5

Figura 12: Definindo o *Controller* do *Model* de Clientes

De acordo com a Figura 12, as seguintes operações CRUD, foram estabelecidas e códificadas:

- **Criar**: Da linha 3 à 8 do arquivo apresentado na Figura 12, é criado o método `criar` do *controller*. Este método basicamente irá inserir um novo cliente ao banco de dados. Assim, um novo Cliente é criado na linha 4, e como parâmetro é passado `res.body`. A variável `res` vem dos parâmetros da *function*, onde o *Body Parser* se encarrega de passar o objeto criado. Por esse motivo, a variável `res.body` é chamada, onde `body` é o objeto JSON enviado via *browser*. Em seguida, o `cliente` novo é instânciado a uma função `save` proveniente do *mongoose*, onde está salvado o cliente, e como parâmetro a função `save` retorna algo. Este retorno é instânciado ao parâmetro `res` da função de *callback* do método `criar`, e retorna os resultados para o *browser* como objetos JSON;
- **Listar**: Da linha 10 à 14 do arquivo apresentado na Figura 12, é criado o método `listar` do *controller*. Este método irá listar os clientes existentes no banco de dados. Para isso, o Cliente definido no inícil da aplicação é instânciado à função

ISBN: 978-85-68323-04-5

`find` do *mongoose*. No primeiro parâmetro da função `find` é definido o nome dos atributos que devem ser buscados. Neste caso, como não está sendo buscado algo específico e sim todos os cadastros, o objeto passado está vazio. No segundo parâmetro é passado a função de *callback*, que retornará ao *browser* os resultados encontrados;

- **Atualizar**: Da linha 16 à 24 do arquivo apresentado na Figura 12, é criado o método `atualizar` do *controller*. Este método irá atualizar um cliente existente no banco de dados. Para isso, o Cliente definido no início da aplicação é instânciado à função `update` do *mongoose*. Esta função tem três parâmetros: o primeiro é o identificador do cliente, onde o mongoose identifica qual cliente será atualizado no *MongoDB*; o segundo é o conteúdo que será atualizado, neste caso um objeto vindo do *Body Parser*; e o terceiro é a função de *callback*, que pode retornar as mensagens de erro ou sucesso, em caso de atualização; e
- **Deletar**: Da linha 26 à 33 do arquivo apresentado na Figura 12, é criado o método `deletar` do *controller*. Este método irá remover um cliente existente no banco de dados. Para isso, o Cliente definido no inícil da aplicação é instânciado à função `findByIdAndRemove` do *mongoose*. Esta função tem como objetivo encontrar um cliente atráves de seu identificador, e removê-lo da *collection*. Ainda, essa função possui dois parâmetros: o primeiro é o identificador do cliente, referente à qual cliente será removido da collection; e o segundo é a função de *callback*, com o retorno do que ocorreu.

A próxima Subseção apresenta como testar e criar as rotas REST para manipular os dados da aplicação.

4.4. Criando Rotas REST Para Manipular os Dados da Aplicação

Para testar se o *model* e o *controlller* criado é efetivo, foi necessário utilizar uma aplicação REST (*Representational State Transfer*), que é um tipo de trânsferência de estado representacional do HTTP. Por meio de REST é possível se comunicar com o *webserver* informando qual operação deseja-se realizar, pelo tipo de acesso que é enviado via *browser*. Atualmente, o HTTP/1.1 suporta os seguintes tipos: OPTIONS; GET; HEAD; POST; PUT; DELETE; TRACE; e CONNECT.

Os sistemas que seguem os princípios REST são chamados de *RESTful*. Existem diversos programas e plugins para *browsers*, para testar as aplicações *RESTful*. Eles auxiliam a verificar se os dados estão sendo retornados corretamente, se estão sendo cadastrados corretamente, atualizados e deletados respectivamente. Para testar os exemplos desta aplicação, foi utilizado o plugin para *Safari*, chamado de *CocoaRestClient*. Para *Chrome*, existe o *Postman*, *Advanced REST client* entre outros, e também existe alguns que são para *Desktop*.

Os tipos que foram utilizados nesta aplicação foram:

- **GET**: para retornar os clientes cadastrados;
- **POST**: para cadastrar os clientes;

ISBN: 978-85-68323-04-5

- **PUT**: para atualizar um determinado cliente; e
- **DELETE**: para remover um determinado cliente.

Para essa integração funcionar, deve-se estipular as rotas no *webserver*. Para tanto, é necessário que o *webserver* inclua os *controllers* existentes. No caso desta aplicação, há apenas um *controller*. Assim, o seguinte código deve ser adicionado:

```
var cadastroController = require(__dirname +
            '/api/controllers/cliente-controller.js');
```

Desta forma, a variável `cadastroController` poderá informar qual operação deverá ser realizada para uma determinada rota.

Em seguida, deve-se criar as rotas por meio das funções `get`, `post`, `put`, `delete` do *Express.js*. Para as requisões do tipo GET, a seguinte rota deve ser criada:

```
app.get('/api/cliente', cadastroController.listar);
```

Quando a rota '`/api/cliente`' do *webserver* é acessada com o tipo GET, ela retornará a lista de cadastros. Por este motivo, a variável `cadastroController.listar`, se refere ao *controller* `listar`, criado no arquivo `cliente-controller.js`.

Para as requisições do tipo POST, a seguinte roda deve ser criada:

```
app.post('/api/cliente', cadastroController.criar);
```

Quando a rota '`/api/cliente`' do *webserver* é acessada com o tipo POST, um novo cliente será adicionado à *collection* Cliente. Por este motivo, a variável `cadastroController.criar`, se refere ao *controller* `criar`, criado no arquivo `cliente-controller.js`.

Para as requisições do tipo PUT, a seguinte roda deve ser criada:

```
app.put('/api/cliente', cadastroController.atualizar);
```

Quando a rota '`/api/cliente`' do *webserver* é acessada com o tipo PUT, um determinado cliente será atualizado na *collection* Cliente. Por este motivo, a variável `cadastroController.atualizar`, se refere ao *controller* `atualizar`, criado no arquivo `cliente-controller.js`.

Finalmente, para as requisições do tipo DELETE, a seguinte roda deve ser criada:

```
app.delete('/api/cliente/:id',
            cadastroController.deletar);
```

Quando a rota '`/api/cliente/:id`' do *webserver* é acessada com o tipo DELETE, um determinado cliente será removido da *collection* Cliente, desde que um identificador seja passado como parâmetro, por esse motivo existe o parâmetro `:id` na rota criada. Assim, a variável `cadastroController.deletar`, se refere ao *controller* `deletar`, criado no arquivo `cliente-controller.js`.

ISBN: 978-85-68323-04-5

A Figura 13 apresenta a versão final do *webserver*, com as rotas devidamente criadas.

Figura 13: Definição das Rotas no *WebServer*

O próximo passo é integrar a interface *Angular.js* com as rotas do *webserver*.

4.5. Interagindo o *Angular.js* com os dados advindos do *MongoDB*

O serviço `$http` do *Angular.js* permite a realização de requisições utilizando `XMLHttpRequest` ou via `JSONP` [Branas 2014]:

- `get(url, config);`
- `post(url, data, config);`
- `put(url, data, config);`
- `delete(url, config);`
- `head(url, config);` e
- `jsonp(url, config).`

Desta forma, é posível interagir o *Angular.js* com o *MongoDB*, via REST. Para tanto, algumas modificações no template criado serão necessárias, e somente estas serão abordadas nesta Seção.

4.5.1. Definindo os Serviços *Angular.js* da Aplicação

As iterações diretas com o *webserver* via REST, neste artigo, serão realizadas utilizando um arquivo de serviço da aplicação, por meio da função `factory` do *Angular.js*. Esta função irá criar os serviços para as operações de manipulação dos dados que estão disponíveis. Para tanto, um novo arquivo chamado `clientesAPIService.js` será criado no diretório `public/assets/js/`. A Figura 14 apresenta a API de serviços das operações disponíveis na aplicação.

ISBN: 978-85-68323-04-5

clientesAPIService.js

```
app.factory("clientesAPI", function($http){

    var _listarClientes = function(){
        return $http.get("http://localhost:3000/api/cliente/");
    };

    var _cadastrarCliente = function(cliente){
        return $http.post("http://localhost:3000/api/cliente/", cliente);
    };

    var _atualizarCliente = function(cliente){
        return $http.put("http://localhost:3000/api/cliente/", cliente);
    };

    var _deletarCliente = function(cliente){
        console.log(cliente);
        return $http.delete("http://localhost:3000/api/cliente/" + cliente._id);
    };

    return {
        listarClientes: _listarClientes,
        cadastrarCliente: _cadastrarCliente,
        atualizarCliente: _atualizarCliente,
        deletarCliente: _deletarCliente
    }

});
```

Figura 14: API de Serviços da Aplicação com *Angular.js*

De acordo com a Figura 14, a variável `app`, percentente ao arquivo `app.js` do mesmo diretório, se refere ao módulo da aplicação, intitulado `meuApp`. Assim, o serviço do tipo `factory` está sendo instânciado a tal aplicação. O primeiro parâmetro da função `factory` é o nome do serviço. Neste caso, o nome deste serviço foi definido como `clientesAPI`. O segundo parâmetro, é uma função. Com isso, é possível criar métodos dentro deste serviço `clientesAPI`, onde estes poderão ser acessados por *controllers* do *Angular.js*. Assim, dentro deste serviço foram criados quatro métodos:

- `_listarClientes`: este método tem por objetivo retornar os clientes existentes no banco de dados da aplicação. Assim, o serviço `$http` instânciado a função REST `get` do *Angular.js*, acessa a rota criada no *webserver* que retorna os clientes cadastrados;
- `_cadastrarCliente`: este método tem por objetivo cadastrar um determinado cliente no banco de dados da aplicação. Assim, o serviço `$http` instânciado a função REST `post` do *Angular.js*, acessa a rota criada no *webserver* e envia o objeto cliente, recebido pelo parâmetro de sua própria função;
- `_atualizarCliente`: este método tem por objetivo atualizar um determinado cliente existente no banco de dados da aplicação. Assim, o serviço `$http` instânciado a função REST `put` do *Angular.js*, acessa a rota criada no *webserver*, e envia o objeto cliente, recebido pelo parâmetro de sua própria função; e

ISBN: 978-85-68323-04-5

- `_deletarCliente`: este método tem por objetivo excluir um determinado cliente existente no banco de dados da aplicação. Assim, o serviço `$http` instânciado a função REST `delete` do *Angular.js*, acessa a rota criada no *webserver*, passando o identificador do cliente junto à rota, recebido pelo parâmetro de sua própria função.

4.5.2. Acessando os Serviços *Angular.js* da Aplicação

O serviço `clientesAPI` criado, retorna um objeto contendo os respectivos nomes dos módulos, referênciando-os sem o caracter "_" (underline). Isso significa que, os *controllers* do *Angular.js*, poderão acessar os serviços da seguinte maneira: Nome do Serviço + Nome do método. Por exemplo:

```
clientesAPI.listarClientes()
```

A Figura 15 apresenta a primeira parte do arquivo `app.js`, onde este está acessando o método `listarClientes()` do serviço `clientesAPI`. Para este serviço ser visto pelo *controller*, é necessário incluí-lo no parâmetro da função do respectivo controlador. Anteriormente, o objeto `$scope.cadastros` estavam recebendo valores estáticos, agora irá receber os valores existentes no *MongoDB*. Por este motivo, foi instânciado à ele um *array* vazio.

app.js

```
1  var app = angular.module('meuApp', []);
2
3  app.controller("cadastroController", function($scope, clientesAPI){
4      $scope.nomeApp = "MEAN Application";
5      $scope.showForm = false;
6      $scope.checkedAll = false;
7      $scope.contaItens = 0;
8      $scope.cadastros = [];
9
10     //Requisições
11     var listarClientes = function() {
12         clientesAPI.listarClientes().success(function(data){
13             $scope.cadastros = data;
14         });
15     }
16
17     listarClientes();
```

Figura 15: Configurando e Acessando os Métodos Disponíveis no Serviço `clientesAPI`

Uma função foi criada, com o nome `listarClientes`, onde esta é acessada pelo próprio controlador da aplicação. Esta função irá acessar o método `listarClientes()` do serviço `clientesAPI`. Em caso de sucesso, a função `success` do Angular.js irá retornar ao objeto `$scope.cadastros` os objetos retornados. Em seguida, esta função é chamada, a fim de carregar os objetos retornados

ISBN: 978-85-68323-04-5

ao *template*. Esta função, não pertence ao escopo, ou seja, não pode ser acessada diretamente pelo template.

A Figura 16 apresenta a segunda parte do arquivo `app.js`. Nesta parte, duas funções são criadas: `salvarCliente` e `deletarClientes`. Ambas podem ser acessadas diretamente pelo escopo, ou seja, pelo *template*.

app.js

```
18
19    $scope.salvarCliente = function (cadastro) {
20        var salvarCliente = clientesAPI.cadastrarCliente(cadastro);
21        if(cadastro._id){
22            salvarCliente = clientesAPI.atualizarCliente(cadastro);
23        }
24        salvarCliente.success(function(data){
25            delete $scope.cadastro;
26            $scope.showForm = false;
27            $scope.cadastroForm.$setPristine();
28            listarClientes();
29        });
30    };
31
32    $scope.deletarClientes = function (cadastros) {
33        $scope.cadastros = cadastros.filter(function (cadastro) {
34            if (cadastro.warning){
35                clientesAPI.deletarCliente(cadastro).success(function(data){});
36            }else{
37                return cadastro;
38            }
39        });
40        if($scope.cadastros.length == 0){
41            $scope.checkedAll = false;
42        }
43    };
44
```

Figura 16: Acessando os Métodos Disponíveis no Serviço `clientesAPI`

A função `salvarCliente` tem uma particularidade. Ela serve tanto para adicionar um novo cliente, como para atualizar um respectivo cliente existente. Ela inicia criando uma variável `salvarCliente`, que recebe o método `cadastrarCliente` do serviço `clientesAPI`. Caso, exista algum identificador vinculado ao parâmetro cadastro, esta variável passa a receber o método `atualizarCliente` do serviço `clientesAPI`. Após realizada esta verificação, a variável criada é instânciada a uma função `success` do *Angular.js*. Dentro desta função existe alguns eventos. Primeiramente, é deletado o objeto de escopo cadastro (que só é utilizado no *template* quando está vinculado ao formulário de cadastros). O segundo evento `showForm` é utilizado para o template saber se deve mostrar a listagem dos clientes ou o formulário de cadastro. Em seguida o formulário de cadastro é instânciado a uma função do *Angular.js* chamada de `setPristine`. Esta função tem vinculo com as validações realizadas no formulário, ou seja, nesse momento é solicitado que ela remova as validações realizadas referente aos campos que foram utilizados. E por fim, os dados são recarregados.

ISBN: 978-85-68323-04-5

A função `deletarClientes` pode nesse momento fazer com que seja excluído mais de um cliente. O objeto de escopo cadastros receberá somente os cadastros que não serão excluídos. Isso acontece, porque a função `filter` do *Angular.js* está filtrando todos os cadastos passados por parâmetro na função, e estes são submetidos a uma verificação. Assim, o uma condição verifica se no objeto filtrado existe uma classe `warning`. Esta classe é ativada ao objeto, no momento em que o usuário ativa o item *checkbox* do respectivo usuário, assim, este atributo passa a ser *true* para este usuário. Vale lembrar que este atributo só existe no escopo do *template*, ele não existe no banco de dados. Assim, os respectivos objetos marcados no template são submetidos ao método `deletarCliente` do serviço `clientesAPI`. Em caso de sucesso, nada está sendo retornado, mas poderia retornar alguma mensagem, caso fosse necessário. Portanto, se o item `warning` do objeto deste escopo for *false* ou não existir este item neste objeto, tal objeto é retornado ao *array* de objetos cadastros do escopo da aplicação. Por fim, uma outra condição verifica se o *array* de objetos cadastros é vazio. Caso seja, a variável `checkedAll` recebe *false*. Esta variável só auxilia o template a desmarcar o *checkbox* que seleciona todos os itens cadastrados, para uma possível exclusão mútua.

4.6. Considerações Finais

Esta Seção apresentou uma aplicação básica realizada com todas as tecnologias que compõe o *MEAN Stack*. Assim, foi possível compreender passo a passo a utilização e a aplicação de tecnologias, no contexto de uma aplicação CRUD. A Figura 17 apresenta a tela final da aplicação em *Angular.js*, com os dados retornados do *MongoDB*, via as rotas da API REST do *Express.js*, junto ao *Node.js*.

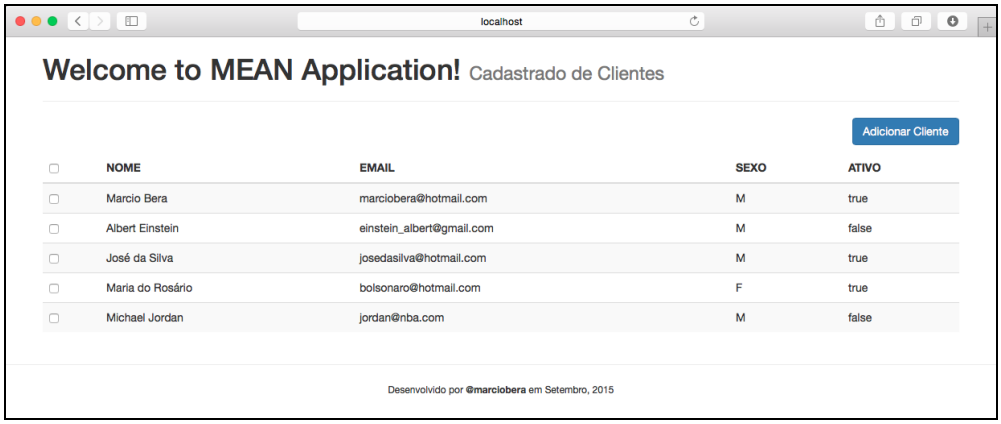


Figura 17: Tela Final da Aplicação MEAN

Todo o conteúdo desenvolvido nesta Seção, está disponível para download no *GitHub* (https://github.com/marciobera/aplicacao-mean-artigo-pos-mobile).

5. Casos de Sucesso

Muitas empresas e grandes aplicações estão utilizando ou migrando para as tecnologias baseadas em *JavaScript*. Empresas como *Ebay*, *Linkedin*, *Adobe*, *Shutterstock*, *PayPal*,

ISBN: 978-85-68323-04-5

Yahoo!, *Walmart*, *Microsoft*, *Uber*, *Trello*, *Academia.edu*, *Wikpedia*, *Grupon*, *The New York Times*, *Amazon*, dentre outras, utilizam tecnologias *JavaScript* em suas aplicações.

5.1. Migração da *PayPal* para Aplicação com *Node.js*

Esta Seção, irá apresentar especificamente o caso de sucesso da empresa *PayPal*, que migrou de uma aplicação em *Java*, para uma aplicação baseada em *JavaScript*. As informações aqui descritas, foram retiradas do próprio site dos engenheiros da *PayPal* 11, em Novembro de 2013.

5.2. Indícios que Levaram os Engenheiros a Mudar de Tecnologia

Antes da migração, os engenheiros do *PayPal* criavam as interfaces da aplicação (utilizando HTML, CSS e *JavaScript*), e solicitavam a criação de persistência dos dados em *Java* para outros desenvolvedores. Com a chegada de plataformas *full-stack* isso acabou sendo resolvido. O *Node.js* foi um divisor de águas, permitindo desenvolver as aplicações tanto no *frontend* quando no *backend*. Assim, passaram a utilizar *Express.js* por conta da flexibilidade que o *framework* permite desenvolver. O único problema encontrado foi que a consistência do desenvolvimento poderia ser complexa. Porém isso foi resolvido por meio da utilização de padrões de desenvolvimento.

Começaram então a desenvolver pequenas aplicações, e consequentemente aplicações paralelas em *Java*, a fim de mitigar o risco pela mudança de plataforma. Este cenário, proporcinou alguns dados interessantes. Enquanto uma parte da equipe continuava desenvolvendo em *Java*, uma pequena parte da equipe desenvolvia as mesmas funcionalidades em *Node.js*. Alguns detalhes se destacaram após ocorrer alguns testes com ambas plataformas:

- A aplicação com *Node.js* foi construída duas vezes mais rápida que a com *Java*, e com um número menor de pessoas;
- A aplicação com *Node.js* foi escrita com 33% de linha de código a menos; e
- A construção da aplicação com *Node.js* teve 40% de arquivos a menos.

Estas evidências ajudaram a encorajar a equipe, mostrando que poderiam se mover mais rápido com *JavaScript*. Até mesmo os engenheiros *Java* que no início do projeto estavam incertos sobre *Node.js*, consideraram mudar-se para desenvolver paralelamente com a nova tecnologia, o que proporcionou o dobro da produtividade vista anteriormente.

5.3 Performance da aplicação *Java* vs *Node.js*

As duas aplicações construídas eram idênticas, porêm com linguagens diferentes e construídas por aproximadamente a mesma equipe de desenvolvedores. De um lado o *framework Java* baseado em *Spring*, e de outro construído sobre *Kraken.js* utilizando *Express.js*, *Dust.js* entre outros códigos *open source*. A aplicação continha três rotas e cada rota feita entre duas a cinco requisições de API, orquestrando os dados e renderizando a página utilizando a API *Dust*.

ISBN: 978-85-68323-04-5

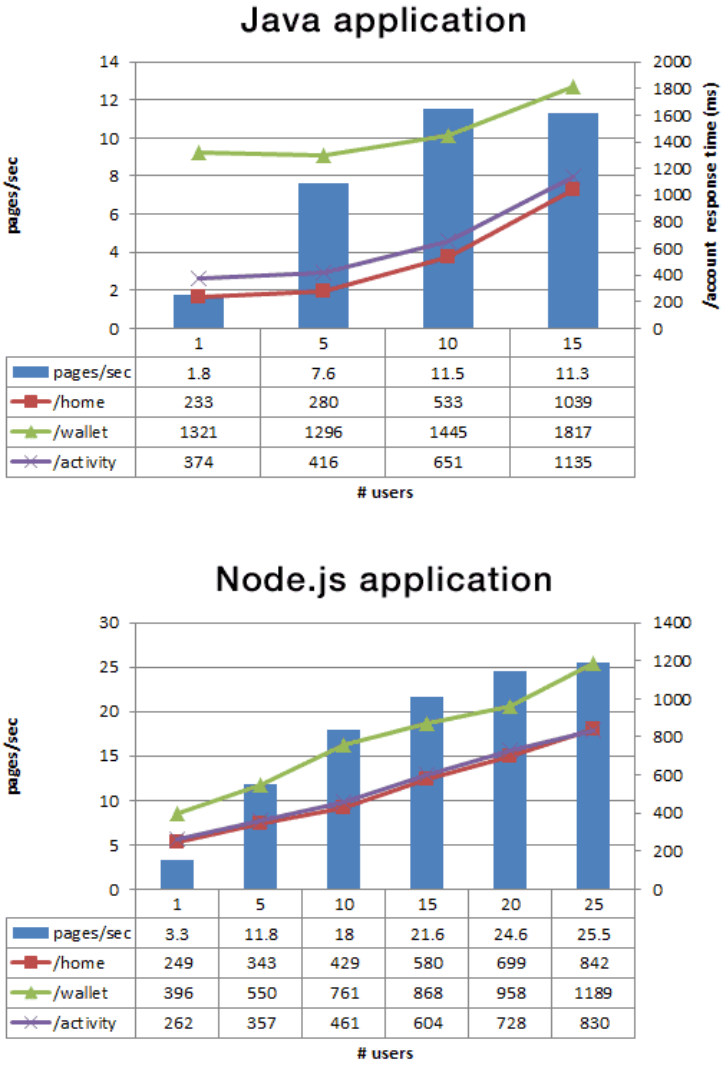
O conjunto de testes aconteceu usando o *hardware* em produção, que testou as rotas e os dados coletados sobre o rendimento e tempo de resposta. A Figura 18 apresenta o conjunto de testes realizados.

É perceptível que a aplicação em *Node.js* obteve:

- **O dobro de pedidos por segundo contra a aplicação *Java***. Isso porque os resultados iniciais de desempenho estavam utilizando um único núcleo para a aplicação *Node.js*, contra cinco núcleos da aplicação em *Java*; e
- **Diminuição de 35% no tempo de resposta para a mesma página**. Resultou em uma página sendo apresentada em 200ms mais rápido, algo notável até para os usuários.

Os dados apresentados, são muito satisfatórios para a equipe do *PayPal*. Todas as aplicações a serem desenvolvidas posteriormente seriam feitas em *Node.js*, de acordo com seus engenheiros.

ISBN: 978-85-68323-04-5



	1	5	10	15
pages/sec	1.8	7.6	11.5	11.3
/home	233	280	533	1039
/wallet	1321	1296	1445	1817
/activity	374	416	651	1135

	1	5	10	15	20	25
pages/sec	3.3	11.8	18	21.6	24.6	25.5
/home	249	343	429	580	699	842
/wallet	396	550	761	868	958	1189
/activity	262	357	461	604	728	830

Figura 18: Performance da Aplicação do *PayPal*: *Node.js* vs *Java*

5. Conclusão

É fato que na última década as tecnologias evoluiram de maneira surpreendente. Paralelamente, os usuários passaram a ser mais exigentes. Na medida que a tecnologia evolui, os usuários buscam utilizar aplicações que lhe façam poupar mais tempo, lhe dê conforto e alguns outros fatores.

O caso de sucesso apresentado neste artigo, mostra uma visão de engenheiros, onde a utilização de novas tecnologias fizeram com que a produtividade da equipe fosse aumentada significativamente, além do desempenho que a aplicação forneceu aos usuários.

ISBN: 978-85-68323-04-5

Este artigo também, apresentou o uso de tecnologias *JavaScript*, que tem se destacado no cenário de desenvolvimento, dentre muitas outras empresas e aplicações de grande escala.

Uma aplicação básica, desenvolvida utilizando as tecnologias que compõe o *MEAN Stack*, também foi apresentada neste artigo. A construção passo a passo de cada parte da aplicação foi comentada e apresentada por imagens reais, que fornecem aos leitores um possível conforto sobre a sua utilização.

Portanto, de acordo com a necessidade de evolução de tecnologias, levando em consideração os resultados de produtividade e performance das tecnologias que compõe o *MEAN Stack*, é notório que esta tecnologia pode proporcionar aplicações robustas e de grandes escalas, podendo ser uma solução viável para suportar o crescente número de usuários, explorando os benefícios da escalabilidade.

Referências

Adobe, S. I. (2012). Javascript for acrobat.

Adobe, S. I. (2014). Adobe photoshop cc 2014 javascript scripting reference.

Adobe, S. I. (2015). Adobe illustrator cc 2015 scripting reference: Javascript.

Bonfim, F. L. and Liang, M. (2014). Aplicações escaláveis utilizando mean stack. Ciências Exatas da Universidade Federal do Paraná.

Branas, R. (2014). AngularJS Essentials. Community Experience Distilled. Packt Publishing.

Bretz, A. and Ihrig, C. J. (2014). Full Stack JavaScript Development with MEAN. SitePoint Pty. Ltd.

DB-Engines (2015). Db-engines ranking.

Firdaus, T. (2014). 10 frameworks to build mobile application with html, css & javascript.

Gotthilf, Y. (2014). Introduction to the mean stack.

Handstudio Co., L. (2013). Samsung SmartTV Application Development. Wiley, 1th edition.

Microsoft (2014). Winjs: The windows library for javascript.

Milani, M. and Benvie, B. (2012). Build desktop applications for linux, windows and mac using html, css and javascript.

MongoDB, I. (2015). Introduction to mongodb.

Pinkham, A. (2015). Projects, applications, and companies using node.

Politowski, C. and Maran, V. (2014). Comparação de performance entre postgresql e mongodb. Departamento de Ciências Exatas e Engenharias.

ISBN: 978-85-68323-04-5

Rauschmayer, D. A. (2012). The Past, Present, and Future of JavaScript. O'Reilly, 1th edition.

RedMonk (2015). The redmonk programming language rankings: January 2015.

Seshadri, S. and Green, B. (2014). Desenvolvendo com AngularJS. O'Relly.

Sevilleja, C. and Lloyd, H. (2015). MEAN Machine: A beginner's practical guide to the JavaScript stack. Leanpub.

StrongLoop, I. (2015). Expressfast, unopinionated, minimalist web framework for node.js.

ISBN: 978-85-68323-04-5