Utilizando MEAN Stack para o desenvolvimento de API REST

Adriano F. De Araújo1, Leonardo Sommariva2

1Departamento de Sistemas e Computação  
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brazil

2Departamento de Sistemas e Computação  
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brazil

flachadriano@gmail.com, lsommariva@gmail.com

**Abstract.** This paper presents the MEAN Stack framework, developed to make easier to build web applications, focusing on develop front-end and back-end with one programming language, the JavaScript. This framework a few others to do it, AngularJS é used to front-end, back-end use MongoDB, Express and NodeJS.

**Resumo.** Este artigo apresenta o framework MEAN Stack, desenvolvido para facilitar o desenvolvimento de aplicações web visando o desenvolvimento de front-end e back-end com apenas uma linguagem de programação, o JavaScript. Para isto é realizada a utilização de diferentes outros frameworks em conjunto, sendo o AngularJS para o front-end e para o back-end é utilizado MongoDB, Express e NodeJS.

# 1. MEAN Stack *framework*

Durante muitos anos JavaScript foi considerada por muitos como uma linguagem para amadores, porém sua arquitetura de desenvolvimento e potencial fez com que seus desenvolvedores mostrassem o poder desta linguagem. Com o surgimento do AJAX vislumbrou-se a possibilidade de transformar *sites* simples em aplicações *web*, o que inspirou o desenvolvimento de bibliotecas utilitárias, como jQuery e Prototype, para agilizar o desenvolvimento dessas aplicações. Google contribuiu para o contínuo crescimento da linguagem com o Chrome V8(HAVIV, 2014)*.* Este último, lançado em 2008, é uma máquina interpretadora de código JavaScript feita em C++, possibilitando o desenvolvimento de código JavaScript em processadores que suportem a linguagem C++ (GOOGLE, 2015).

Hoje em dia, *JavaScript* tornou-se ubíquo para o desenvolvimento de aplicações *web* *client-side*, porém para o desenvolvimento do servidor dessas aplicações *web*, muitas linguagens, *frameworks* e APIs entram em voga. Várias dessas opções atenderam as expectativas e estão decolando entre os desenvolvedores e empresas, enquanto outras ficaram obsoletas com o tempo. Em 2009, as pessoas já haviam se dado conta do potencial que JavaScript tinha como linguagem para o desenvolvimento de aplicações para o navegador, quando Ryan Dahl vislumbrou o potencial que esta linguagem tinha para o desenvolvimento de aplicações no servidor, então nascia o Node (BROWN, 2014).

Conforme Almeida (2015), O acrônimo MEAN foi cunhado em 2013 por Valeri Karpov do time do MongoDB para denotar o uso de uma stack completa para desenvolvimento de aplicações incluindo MongoDB, Express, AngularJS e Node.js.

A letra M do termo MEAN denota o MongoDB, um banco de dados orientado a documentos, que traz um novo conceito que armazenamento de dados, onde não há um esquema fixo definindo como cada dado armazenado deve ser (CHODOROW, 2013). A forma de armazenamento utilizada é muito similar ao JavaScript Object Notation (JSON) o que ajuda a realizar o armazenamento e reaver os dados, pois JSON é o formato comumente utilizado para prover e consumir APIs. Este formato de armazenamento realiza poucas validações em relação aos dados recebidos, tendo a aplicação a maior parte da responsabilidade de validar estes dados (ALMEIDA, 2015). Mantendo funcionalidades disponibilizadas por um banco relacional, como por exemplo índices e ordenação.

Express, criado em 2009, é responsável pela organização da aplicação no lado do servidor, utilizando a arquitetura MVC (ALMEIDA, 2015). Inspirado no *framework* Sinatra, desenvolvido em Ruby, que preza por desenvolvimento rápido, eficiente e manutenível. Seguindo esta ideia, Express disponibiliza uma camada mínima para o desenvolvimento da aplicação, porém, sua grande força está em permitir que sejam acoplados *middlewares*, que são responsáveis por executar alguma tarefa maior para a aplicação. Permitindo assim, que o framework evolua constantemente através de seus *middlewares*, assim como ocorre com Sinatra (BROW, 2014).

AngularJS implementa a letra A do MEAN Stack, que é responsável pelo desenvolvimento de aplicações no lado do cliente utilizando conceito de *Single Page Application* (SPA) onde a aplicação necessita ser carregada completamente apenas uma vez, sendo as demais chamadas realizada apenas para buscar partes necessárias para realizar a apresentação dos dados ao usuário (ALMEIDA, 2015). Este *framework* preza por desenvolver a aplicação de forma declarativa, extendendo as *tags* HyperText Markable Language (HTML), as tags adicionadas através deste framework são responsáveis por modularizar a aplicação (BRANAS, 2014). Este artigo focará no desenvolvimento da parte de servidor de uma aplicação *web*, sendo assim não será abordado o funcionamento do angular no MEAN Stack.

NodeJS é uma plataforma para aplicações JavaScript que roda sobre o Chrome V8. Nesta plataforma é possível utilizar bibliotecas desenvolvidas pela comunidade através do gerenciador de pacotes NPM. NodeJS seria o *core* da aplicação *web*.

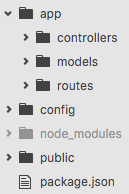
# 2. MongoDB

A primeira etapa é instalar o NodeJS, basta acessar o *site* <https://nodejs.org/en/download/>, baixar o instalador e executá-lo. Para verificar se tudo ocorreu conforme esperado, basta executar no terminal o seguinte comando: *node -v*. Deve ser apresentada uma mensagem com a versão instalada do NodeJS.

Conforme mencionado no item 1, o NPM é o gerenciador de pacotes utilizado pelo NodeJS para baixar as bibliotecas necessárias para o desenvolvimento das aplicações. Porém, o NPM precisa ter uma configuração em cada aplicação que for utilizar. Aconselha-se que para cada aplicação desenvolvida seja criada uma nova pasta no sistema. Para configurar o NPM é necessário abrir o terminal, acessar a pasta onde a aplicação será desenvolvida e executar o comando: *npm init*. Serão realizadas algumas perguntas e ao final será criado um arquivo com nome *package.json*. Para este artigo resultado é apresentado no quadro 1.

Quadro 1. package.json

Após configurado o NPM, pode ser solicitada a instalação da primeira biblioteca, que será o express, para isto basta executar o comando: *npm install* [*express@4.13.4*](mailto:express@4.13.4) *–-save*. A chave *install* informa que está sendo solicitada a instalação de uma biblioteca, em seguida, *express* é o nome da biblioteca que deve ser localizada, *@4.13.4* denota qual versão da biblioteca deve ser instalada, para este artigo será utilizada a versão 4.13.4, foi adicionado ao fim do comando a informçaão –-*save* que informa ao NPM que esta biblioteca deve ser salva como uma dependência da aplicação. Caso verifique novamente agora o arquivo *package.json*, terá uma nova chave (*dependencies*), com o *express* dentro e a versão especificada.

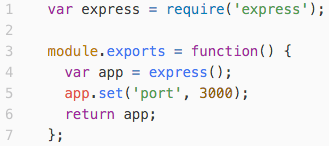
Almeida (2015) sugere que seja criada uma organização de pastas no diretório onde o projeto será desenvolvido, conforme o quadro 2.

Quadro 2. Organização das pastas

Ainda conforme Almeida (2015) cada pasta deve conter os seguintes conteúdos:

* controllers: controladores chamados pelas rotas da aplicação
* models: *models* que representam o domínio do problema
* routes: rotas da aplicação
* views: *views* da máquina geradora de templates
* config: configuração do express, banco de dados, etc
* public: todos os arquivos acessíveis diretamente pelo navegador

Agora é necessário realizar a configuração do servidor. Para isto, será necessário criar um arquivo na pasta *config*, que será responsável por configurar o *express*, que realizará o tratamento das requisições recebidas pela API, deverá ser criado um arquivo com o nome *express.js* nesta pasta. No quadro 3 pode ser observado o conteúdo que este arquivo deve conter, onde na primeira linha é realizada a importação do *framework express.* Na terceira linha, pode ser verificado que é realizada uma chamada *module.exports* que recebe uma *function* como atribuição, ao fazer isto, está sendo informado ao *node* que ao realizar a importação deste arquivo em outro lugar deverá ser retornado o que estiver dentro de *exports*. Na linha 4 está sendo realizada chamada ao *express* para inicializar uma nova aplicação. Na linha 5, é feita uma chamada ao método *set* da aplicação que está sendo inicializada, este método serve para armazenar alguma informação dentro da aplicação, neste caso está guardando o valor *3000* com a chave *port.*

Quadro 3. arquivo config/express.js

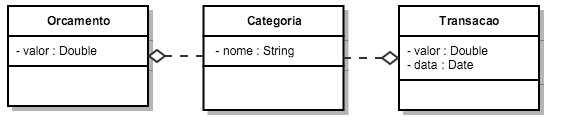
Agora que já foi criado o arquivo que instancia uma nova aplicação do *express*, é necessário criar um servidor *node* para levantar esta aplicação. Para isto deve ser criado um arquivo chamado *server.js* na pasta raiz do projeto, seu conteúdo deve ser conforme o quadro 4. Onde na primeira linha é realizada importação da biblioteca *http*, que é responsável por interagir com a rede do computador. Na segunda linha é feita a importação das configurações da aplicação que está sendo criada. Na linha 4, é chamado o método *createServer* da bilbioteca *http*, isto faz com que tenha uma instância de servidor do node, ao chamar em seguida o método *listen*, deve ser informado qual porta de rede a aplicação deve ouvir e se há alguma *callback* para ser executada após a aplicação iniciar. Como pode ser verificado, como primeiro parâmetro foi enviado *app.get(‘port’)*, na configuração da aplicação, foi chamado o método *set*, que guarda algum valor na aplicação, agora é realizada chamada ao *get* para buscar o valor guardado na aplicação, então como primeiro parâmetro está sendo enviado o valor *3000*. Como segundo parâmetro foi enviado uma *function* que irá imprimir no console do servidor ‘Express Server escutando na porta 3000’ se tudo tiver sido iniciado conforme o esperado.

Quadro 4. arquivo server.js

Para verificar se a configuração foi implementada corretamente, basta acessar o terminal, entrar na pasta raiz do projeto e executar o seguinte comando: *node server*. Deve ser executado a mensagem conforme esperado.

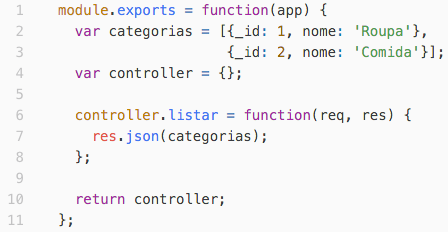
**3. Desenvolvendo a aplicação**

Foi criada a infraestrutura para uma aplicação MEAN, então agora faz-se necessário definir o tipo de aplicação que será desenvolvida. Será uma aplicação *web* para controle financeiro pessoal, onde terá categorias, que são responsáveis por categorizar os tipos de despesas, terão orçamentos, que indicam qual o valor máximo desejado para gastar mensalmente com as categorias e por fim as transações, que são as despesas. Pode ser verificado um diagrama no quadro 5.



Quadro 5. UML

Para iniciar o desenvolvimento será utilizada a categoria, pois não tem dependência entre os demais objetos. Então para isto deve ser criado um arquivo com o nome *categoria.js* na pasta controllers. Este arquivo terá o código responável por executar as rotas chamadas pelo navegador. Neste primeiro momento não será realizada a persistência dos dados no banco. Sendo assim, como primeira etapa será implementado o método responsável por listar algumas categorias fixas, conforme pode ser verificado no quadro 6.

 Quadro 6. controllers/categoria.js

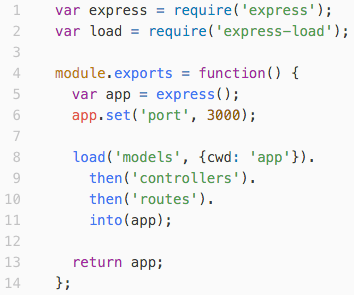
Assim como ocorreu com o arquivo *express.js*, para este arquivo também será exportada uma *function* que ao final retorna um objeto. Neste objeto, que será chamado de controller, deverá conter outras *functions* que executarão a ação responável por alguma rota chamada pelo navegador. Como pode ser verificado na linha 6 do quadro 6, foi adicionado para a chave *listar* uma *function* que recebe dois parâmetros, todas as funtions de *controller* no *express* podem receber estes dois parâmetros, *req* identifica os dados da requisição enviada pelo navegador ao acessar algum endereço da aplicação, enquanto *res* contém os dados de resposta para a requisição. Na linha 7 é chamado o método *json* do objeto *res*, isto fará com que a resposta enviada a solicitação seja um objeto JSON.

Após o desenvolvimento do *controller* é necessário identificar qual rota da aplicação corresponderá a este método do *controller*, para isto é necessário criar um novo arquivo, desta vez na pasta *routes*, deve ter o nome *categoria.js* e seu conteúdo deve ser idêntico ao quadro 7.

 Quadro 7. routes/categoria.js

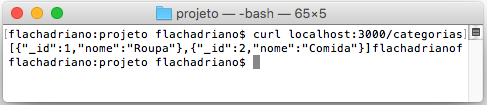
Assim como nos *controllers*, as rotas também podem receber como parâmetro a instância da aplicação que está sendo desenvolvida, utilizando esta instância, na linha dois é armazenado em uma variável o *controller* de categoria. Na linha 4, está sendo chamado o método *route* de *app*, que irá criar um recurso atrelado à esta rota, então a partir disto pode ser realizada chamada a recursos REST (get, put, post, update). Após obter este recurso é adicionado uma ligação com o método *get* enviando como parâmetro qual método do controller corresponde à rota especificada.

Após implementação do *controller* e da rota, a aplicação ainda não rodará conforme esperado, pois de alguma forma na linha 2 do quadro 7, foi acessado um *controller* da aplicação, porém não foi especificado a configuração de onde este *controller* é localizado. Para isto é necessário alterar o conteúdo do arquivo *express.js* que está na pasta *config*, conforme pode ser verificado no quadro 8.

 Quadro 8. config/express.js

Na linha dois pode ser verificado que está sendo importada uma nova biblioteca, porém ela ainda não foi instalada pelo NPM, para isto basta executar no terminal o seguinte comando na pasta raiz do projeto: *npm install* [*expres-load@1.1.15*](mailto:expres-load@1.1.15) *–-save*. Agora que está importada, pode-se verificar sua utilização entre as linhas 8 e 11, nestas linhas está sendo indicado ao *express-load* que os arquivos que estiverem nas pastas *models*, *controllers* e *routes* devem ser carregados para dentro da aplicação, isto significa que não será necessário executar o método *require* em nenhum dos arquivos da aplicação para acessar outro arquivo da mesma aplicação, isto será carregado na inicialização do *express*.

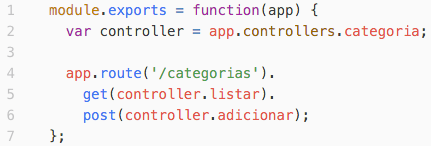
Agora a aplicação está funcional, para testar o que foi desenvolvido será utilizado cURL, que é uma biblioteca de código aberto para transferência de dados com a sintaxe de Uniform Resource Locator (URL) (CURL, 2016). Para utilizar esta biblioteca é necessário realizar o download, no endereço <https://curl.haxx.se/download.html>. Em seguida, após acessar a pasta da aplicação pelo terminal e executar novamente o comando: *node server*. Pode ser executado o cURL para verificar se as categorias estão sendo retornadas quando acessada a rota correspondente. Para executá-lo é necessário abrir um novo terminal e executar o comando: *curl localhost:3000/categorias*. O resultado deve ser idêntico ao quadro 9.

 Quadro 9. executando aplicação

Ao utilizar a aplicação, um usuário deve poder não só visualizar as categorias disponíveis, mas também criar, alterar e deletá-las. Para que isto ocorra é necessário desenvolver novos métodos no *controller* que corresponderão às rotas. Primeiramente será desenvolvida a rota para adicionar as categorias, sendo assim, será implementado o método responsável por adicionar uma nova categoria na lista de categorias disponíveis, para isso deve ser alterado o arquivo de *controller* de categoria, implementando um novo método (linha 4 do quadro 10) para adicionar mais um item nas lista de categorias. Como pode ser visto no quadro 10, na linha 6, é adicionado o corpo (*body*) da requisição dentro da lista de categorias. Na linha 7 é retornado o status 201, informando que a categoria foi criada com sucesso e o objeto que foi adicionado à lista.

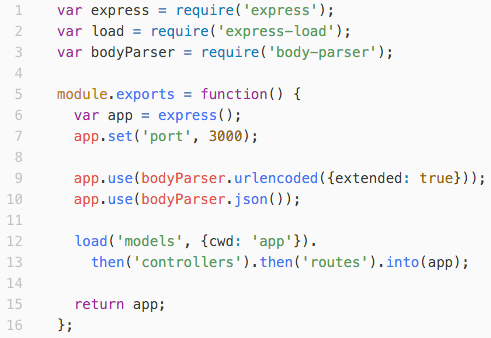
 Quadro 10. executando aplicação

Deve ainda ser ajustado o arquivo de rotas de categoria (*categoria.js*) para responder a rota de criação de categoria com este novo método, conforme é apresentado no quadro 11.

 Quadro 11. executando aplicação

Como pode ser verificado na linha 6 do quadro 11, foi encadeado a chamada do método *post* após o método *get*, quando se tem um recurso de rota, qualquer método REST pode ser chamado em seguida, neste caso está sendo informado que a rota */categorias* quando for chamada através do método *post* deve ser executada a *function* adicionar.

O método adicionar espera que no corpo da requisição retorne dados em JSON, porém por padrão isto não ocorre, portanto será necessário configurar o *express* para que isto ocorra. Foram adicionadas as linhas 3, 9 e 10, conforme pode ser verificado no quadro 12 para realizar essa configuração. Na linha 2 é importada uma biblioteca que ainda não existe no projeto, portanto faz-se necessário solicitar a instalação ao NPM através do comando: *npm install* [*body-parser@1.15.0*](mailto:body-parser@1.15.0) *–-save*. Como apresentado anteriormente, uma das forças do express é sua gama de plugins disponíveis, na linha 9 é utilizado o método *use* da aplicação, este método permite que sejam adicionados módulos externos à aplicação, nesta linha, a biblioteca está configurando o express para que aceite URLs extendidas e na linha 10 está informando que deve converter estes dados dessas URLs para um objeto JSON que ocupará o corpo da requisição.

 Quadro 12. arquivo config/express.js

Assim, após estas alterações, pode ser executado o comando no cURL para criar uma categoria. Reinicie o servidor da aplicação que está executando e execute em outro terminal o seguinte comando: *curl –-data “\_id=4&nome=Livros” localhost:3000/categorias*. Se, após este comando, for executado novamente a listagem de categorias, irá apresentar ao final da lista a categoria que foi adicionada.

Ainda restam as rotas para alterar uma categoria, buscar e deletar. Sua implementação pode ser encontrada no quadro 13 e devida configuração das rotas no quadro 14. A implementação de orçamento e transações pode ser encontrados no endereço https://github.com/flachadriano/pos-desenv-web/tree/master/artigo/projeto/memoria.

Quadro 13. arquivo controllers/categoria.js

 Quadro 14. arquivo routes/categoria.js

**4. Persistindo os dados**

Conforme mencionado na seção 1, será utilizado o banco de dados *MongoDB* para persistência dos dados. Desta forma, será necessário realizar a instalação deste banco no computador, para isto basta acessar o site *[https://www.mongodb.org/downloads#production](https://www.mongodb.org/downloads" \l "production)*, baixar o arquivo, clicar no *link* de instruções de instalação e segui-las. Para verificar se foi instalado corretamente, deverá ser aberta uma nova janela do terminal e executado o comando: *mongo*.

Agora o *MongoDB* está instalado, porém é necessário acoplar um driver ao *NodeJS* para que o express consiga acessar o banco de dados, este *middleware* se chama *mongodb* e pode ser instalado pelo NPM, bastando executar o comando: *npm install* [*mongodb@2.1.10*](mailto:mongodb@2.1.10) *–-save*.

Na seção 1 também foi mencionado que por usar uma estrutura JSON para armazenar as informações, o MongoDB passa a maior parte das responsabilidades de validar as informações para a aplicação. Porém, isto não significa que não tenha uma biblioteca que auxilie na realização destas validações, neste momento entra outra bilblioteca, o *mongoose*.

Conforme Almeida (2015), m*ongoose* é uma biblioteca *Object-Document Modeler* (ODM) criada pela equipe do *MongoDB*. Ela é a camada entorno do driver do *MongoDB* que gerencia relacionamentos e executa validações, entre outras funcionalidades.

Para instalar esta biblioteca será utilizado novamente utilizado o NPM, abrindo um terminal e executando o comando: *npm install* [*mongoose@4.4.8*](mailto:mongoose@4.4.8) *–-save*. Ainda se faz necessário executar algumas configurações no *mongoose* para que seja possível se conectar ao MongoDB, para isto deve ser criado um arquivo *database.js* na pasta *config*, conforme o quadro 15.

Quadro 15. arquivo routes/categoria.js

No quadro 15, na primeira linha está sendo realizada a importação da bilioteca *mongoose* que controlará as conexões com o banco de dados *MongoDB*. Na linha 4, está sendo informado ao *mongoose* qual a URL para conexão com o *MongoDB*. É possível interceptar algumas mudanças de status da biblioteca *mongoose*, dentre eles estão: *connected*, *disconnected* e *error*, onde o primeiro será executado quando a conexão for realizada com sucesso, o segundo quando for desconectado do *MongoDB* e o terceiro caso ocorra algum erro para conectar. Na linha 16 é utilizada a variável *process*, porém ela não é recebida como parâmetro em nenhuma lugar do arquivo, isto ocorre pois esta variável está disponível em qualquer momento da aplicação, como uma variável global, ela contém o processo da aplicação, e interceptando o status SIGINT é possível identificar quando a aplicação é encerrada, quando isto ocorrer deve ser desconectado o *mongoose* do *MongoDB*, o que é realizado nas linhas 17 a 21.

Para finalizar a configuração do MongoDB para esta aplicação é necessário chamar este arquivo, que acabou de ser criado, enviando a url com a qual o *mongoose* deve se conectar ao banco. Para isto, é necessário alterar o arquivo *server.js*, adicionando na terceira linha do arquivo o seguinte código: *require('./config/database.js')('mongodb://localhost/mymony');*. Desta forma, ao inicializar a aplicação, o *mongoose* conectará no *MongoDB* utilizando o banco *mymony*.

No quadro 16 é apresentado o modelo para a entidade *Categoria*, onde na primeira linha está sendo realizada a importação da biblioteca *mongoose*, na linha 4 está sendo criado um esquema, que definirá as validações para os atributos da entidade, para este caso terá o atributo *nome* (linha 5) que será do tipo *String* (linha 6), deverá ser obrigatório e gerar um índice de chaves únicas (linhas 7 e 8 respectivamente). Na linha 11 este esquema é registrado no *mongoose* como um modelo que terá a chave *Categoria*.

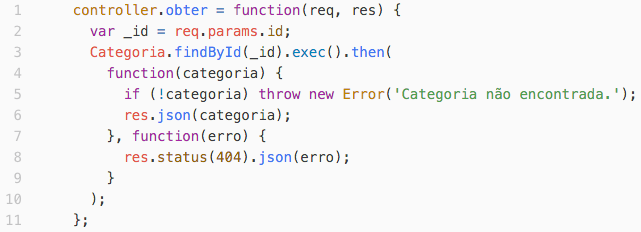
Quadro 16. arquivo models/categoria.js

Agora já está definido o modelo de *Categoria* e como seu atributo deve se comportar, porém isto ainda não está sendo utilizado no *controller,* que é onde são realizadas as interações com a unidade armazenadora dos dados. Para isso deve ser realizada a importação do modelo no *controller.* Deve ser alterado o arquivo *categoria.js* da pasta *controllers*, adicionado o seguinte código: *var Categoria = app.models.categoria;,* na segunda linha do arquivo.

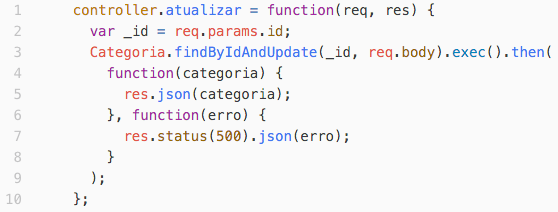
Quadro 17. arquivo controllers/categoria.js

A partir de agora o *controller* terá acesso ao modelo validador dos dados de categorias provido pelo *mongoose*. Conforme pode ser verificado no quadro 17, ao executar o método *find*, está sendo solicitado que sejam buscados todos os documentos salvos na coleção de *Categoria*, ao encadear com o método *exec*, está sendo realizada busca de todas as categorias e retornando uma promise. Com esta promise podem ser enviadas as *functions* responsáveis por tratar o retorno da consulta, onde a primeira representa um retorno de sucess e a segunda de falha.

Para obter um registro em específico foi realizada alteração no método *obter* do *controller* de categoria, o mongoose disponibiliza um método para buscar objetos pelo seu identificador que é *findById* conforme pode ser verificado no quadro 18, linha 3, em seguida é aplicado o método *exec* para realizar o tratamento do retorno, conforme realizado no método para listar as entidades de categoria.

Quadro 18. arquivo controllers/categoria.js

Para atualizar um registro é utilizado o método *findByIdAndUpdate*, que como seu nome demonstra, irá buscar um registro com o identificador enviado como primeiro parâmetro e fará atualização dos dados com o valor enviado como segundo parâmetro.

Quadro 19. arquivo controllers/categoria.js

Por fim, é necessário ajustar a forma como os dados vão ser removidos do banco, para isto é necessário alterar a *function* com esta responsabilidade. Novamente no arquivo *categoria.js* da pasta *controllers* deve ser alterado para ficar conforme o que é apresentado no quadro 20. Onde para remover um objeto deve ser enviado um outro objeto, que define quais os filtros devem ser utilizado para busca, como tem-se apenas o identificador do objeto, somente isto é enviado para a busca, porém se tivesse acesso a outro valor, como por exemplo o nome da categoria, poderia ser enviado este dado para filtrar e deletar.

Quadro 20. arquivo controllers/categoria.js

Agora o *controller* de *Categoria* foi alterado para persistir os dados em banco. Para testar estas alterações é necessário inicializar o banco de dados MongoDB e reiniciar a aplicação. Utilizando novamente a biblioteca cURL podem ser realizados os testes necessários utilizando os seguintes comandos:

* *curl localhost:3000/categorias*: Apresentará a listagem de todas as categorias;
* *curl –data “nome=Livros” localhost:3000/categorias*: Executará a ação de criar um objeto de categoria no banco de dados, dentro das aspas após *–-data* são enviados os dados para criação do documento no banco;
* *curl -X PUT –-data “nome=Cinema” localhost:3000/categorias/:\_id*: Para este comando é necessário enviar o parâmetro no lugar da chave *:\_id*, que deverá ser o identificador de alguma categoria criada anteriormente. Para obter este valor, deve ser executado o primeiro comando desta lista e escolher o valor de alguma das chaves *\_id;*
* *curl -X DELETE localhost:3000/categorias/:\_id*: É necessário enviar algum valor no lugar da chave *:\_id* para que seja removido o objeto com este identificador. Para obter este valor pode ser executado o primeiro comando desta lista e utilizado o valor que estiver em alguma das chaves *\_id*.

Conforme pode ser verificado no *controller* e nos comandos *curl*, assim como no banco relacional, no *MongoDB* também há a possibilidade de se trabalhar com um identificador único para cada coleção de dados, este identificador não precisa ser configurado pela aplicação, o que não foi feito quando o modelo de categoria foi criado. Então, ao executar o comando *curl –data “nome=Livros” localhost:3000/categorias* no terminal e em seguida executar *curl localhost:*3000/categorias, deve apresentar um resultado parecido com o seguinte: [{“\_id”:”56f1b243f322602d0cc4de13”,“nome”:”Livro”,”\_\_v”:”0”}]. Esta chave \_id foi criada automaticamente pelo MongoDB.

As alterações necessárias para os *controllers* de *Orçamento* e *Transação* podem ser obtido através do endereço [*https://github.com/flachadriano/pos-desenv-web/tree/master/artigo/projeto/db*](https://github.com/flachadriano/pos-desenv-web/tree/master/artigo/projeto/db), juntamente com os respectivos modelos.

# Referências

BROWN, Ethan. **Web Development with Node & Express:** Leveraging the JavaScript Stack. Sebastopol: O’Reilly Media, 2014. 306 p.

HAVIV, Amos Q. **MEAN Web Development:** Master real-time web application development using a mean combination of MongoDB, Express, AngularJS, and Node.js. Birmigham: Packt Publishing, 2014. 456 p.

ALMEIDA, Flávio. **MEAN:** Full stack JavaScript para aplicações web com MongoDB, Express, Angular e Node. São Paulo: Casa do Código, 2015. 377 p.

GOOGLE. **Chrome V8:** Google's high performance, open source, JavaScript engine. 2015. Disponível em: <https://developers.google.com/v8/>. Acesso em: 24 mar. 2016.

CHODOROW, Kristina. **MongoDB:** The Definitive Guide. 2. ed. Sebastopol: O’Reilly Media, 2013. 410 p.

BRANAS, Rodrigo. **AngularJS Essentials:** Design and construct reusable, maintainable, and modular web applications with AngularJS. Birmigham: Packt Publishing, 2014. 164 p.

NODEJS. **NodeJS.** 2016. Disponível em: <https://nodejs.org>. Acesso em: 24 mar. 2016.

\_\_\_\_\_\_. **NodeJS v5.9.1 Documentation.** 2016. Disponível em: <https://nodejs.org/api/modules.html>. Acesso em: 25 mar. 2016.

CURL. **CURL:** groks those URLs. 2016. Disponível em: <https://curl.haxx.se/>. Acesso em: 25 mar. 2016.