

BRUNNO ROMERO TENORIO

PORTAL DE VENDA DE PASSAGENS COM *NODE.JS*

**UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG**

2014

BRUNNO ROMERO TENORIO

PORTAL DE VENDA DE PASSAGENS COM *NODE.JS*

Trabalho de Conclusão de Curso Sistemas de Informação da Universidade do Vale do Sapucaí como requisito para obtenção do título de bacharel em SISTEMAS DE INFORMAÇÃO

Orientador: Prof. MSc. Roberto Ribeiro Rocha

**UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2014**

Romero, Brunno

Portal de venda de passagens com *Node.js* / Brunno Romero Tenorio,
– Pouso Alegre – MG: Univás, 2014.
55 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade do Vale
do Sapucaí, Univás, Sistemas de Informação.
Orientador: Prof. MSc. Roberto Ribeiro Rocha

1. Node.js. 2. Socket.io. 3. Web Standards.

BRUNNO ROMERO TENORIO

PORTAL DE VENDA DE PASSAGENS COM *NODE.JS*

Trabalho de conclusão de curso defendido e aprovado em 25/11/2014 pela banca examinadora constituída pelos professores:

Prof. MSc. Roberto Ribeiro Rocha
Orientador

Prof. Ednardo David Segura
Avaliador

Prof. André Luiz Martins de Oliveira
Avaliador

De Brunno Romero Tenorio.

Dedico este trabalho a minha mãe, pessoa em que sempre acreditou em mim e me motivou a nunca desistir.

...

AGRADECIMENTOS

Agradeço inicialmente a minha mãe Marinez Romero, por ter sido mãe e pai, por todo apoio, motivação e tudo que fez por mim. Só tenho a te agradecer por ser essa pessoa especial que sempre acreditou em mim e apoiou-me. Te amo muito.

Ao meu melhor amigo, meu parceiro, meu irmão Gustavo Alexandre, por sempre estar ao meu lado, nos momentos bons e ruins, e sempre me apoiar em tudo. Sem você, eu não seria nada nessa vida.

Ao professor Edy Segura, pela motivação e toda paixão transmitida pelo desenvolvimento web, fator decisivo para minha escolha vocacional. Se hoje sou desenvolvedor, pode ter certeza, foi graças a todo seu entusiasmo e didática. Muito obrigado pela paciência e por tudo que me foi ensinado.

Ao professor Dr. Estevan, pela atenção e paciência e especialmente pela ajuda fornecida com os polinômios utilizados para o funcionamento do Data Mining. Muito obrigado pela atenção e por tudo que me foi ensinado.

Ao professor e orientador Ms. Roberto Ribeiro Rocha, pelo apoio e considerações, no decorrer deste trabalho, visando à sua qualidade, deixo aqui meu muito obrigado.

À turma de Sistema de Informação 2014, pelas amizade e apoio recibidos. Só tenho a agradecer a todos vocês.

“O segredo é cair sete vezes, e levantar oito.

(Paulo Coelho)

ROMERO, Brunno. **Portal de venda de passagens com Node.js**. 2014. Monografia – Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, Pouso Alegre – MG, 2014.

RESUMO

Com o crescimento do mercado de serviços online, o qual se deu devido ao aumento do número de usuários conectados a internet juntamente com o ganho de confiança dos brasileiros em efetuar transações online, as empresas de transporte rodoviário perceberam a necessidade de disponibilizar aos seus clientes a venda de passagens online, em um portal. Diante do exposto, esta pesquisa tem como objetivo geral desenvolver um portal de venda de passagem com um módulo de apoio a tomada de decisão. Seus objetivos específicos são: a) Pesquisar as dificuldades e problemas das aplicações de venda de passagem rodoviária existente na internet; b) Utilizar as tecnologias Node.js e Socket.io para desenvolver o portal de venda de passagem; e c) Desenvolver um módulo de apoio à tomada de decisão. O desenvolvimento desta plataforma deu-se por meio de uma pesquisa aplicada com o uso do Node.js juntamente com o Socket.io. O Node.js é uma plataforma JavaScript que trabalha no lado servidor juntamente com o Socket.io, que é um módulo para Node.js que facilita a manipulação e gerenciamento de Web Sockets, e fornece recursos para o funcionamento *cross-browser* das aplicações. Para atender também a parte gerencial, foi desenvolvido, ainda, um módulo fornecendo gráficos utilizando a API Google Charts e relatórios de previsão de demanda e vendas, utilizando técnicas de Data Mining. Para o desenvolvimento do projeto foi aplicado, o uso de outras tecnologias como HTML5 e CSS3 para o desenvolvimento responsivo das páginas da aplicação, fornecendo suporte não só para desktop bem como para tablets e celulares. Os resultados do projetos foram vários, desde a exibição de uma nova forma de estruturação de aplicações Node.js ao fornecimento de uma matéria que possa servir como base para futuros trabalhos. Ao finalizar a pesquisa, foi notado a robustez do Node.js no desenvolvimento de aplicações web, possibilitando desenvolver aplicações complexas e escaláveis.

Palavras-chave: Node.js. Socket.io. Web Standars.

ROMERO, Brunno. **Portal de venda de passagens com Node.js**. 2014. Monografia – Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, Pouso Alegre – MG, 2014.

ABSTRACT

With the growth of online services market, which was due to the increased number of users connected to the Internet along with gain confidence Brazilians in making online transactions, trucking companies have realized the need to provide its customers selling tickets online in a portal. Given the above, this research has as main objective to develop a portal for the sale of passage with a module to support decision making. Its specific objectives are: a) Find the difficulties and problems of applications of selling existing road crossing on the Internet; b) Using Node.js and Socket.io technologies to develop the sales portal of passage; c) Develop a module to support decision making. The development of this platform was performed by means of an applied research using Node.js together with Socket.io. Node.js JavaScript is a platform that works in conjunction with the server side Socket.io, which is a module for Node.js that facilitates managing Web manipulação and sockets, and provides facilities for the operation cros-browser applications. To also meet the managerial part, was also developed one providing graphics using the Google Charts API and reports of demand forecasting and sales, using Data Mining module. For the development of the project was implemented as instruments, the use of other technologies such as HTML5 and CSS3 for the development of responsive pages of the application, providing support not only for desktop as well as mobile phones and tablets. The results of several projects were, from the view of a new way of structuring Node.js applications to providing a story that can serve as a basis for future work. When you finish the survey was completed on the robustness of the Node.js web application development, enabling scalar and develop complex applications. ...

Key words: Node.js. Socket.io. Web Standard.

LISTA DE FIGURAS

Figura 1 – Exemplo de código Node.js	16
Figura 2 – Exemplo Event-loop	17
Figura 3 – Exemplo MVC	20
Figura 4 – Ajax Polling	26
Figura 5 – Ajax Longpolling	27
Figura 6 – Exemplo WebSocket	27
Figura 7 – Instalação Express	32
Figura 8 – Intalalação Mongoose	33
Figura 9 – Árvore	33
Figura 10 – Package.json	35
Figura 11 – Árvore Completa	36
Figura 12 – Importação dos módulos iniciais	37
Figura 13 – Conexão Banco	38
Figura 14 – Configuração Diretório	38
Figura 15 – Configuração App	39
Figura 16 – Configuração Estatica	39
Figura 17 – Árvore Completa	39
Figura 18 – Carregamento dos Arquivos	40
Figura 19 – Adição Servidor	40
Figura 20 – Exemplo de Rotas	41
Figura 21 – Exemplo de Controller	42
Figura 22 – Serviço Vazio	44
Figura 23 – Modelo Banco	45
Figura 24 – Árvore View	46
Figura 25 – Inclusão do cabeçalho	47
Figura 26 – Exemplo de página	47
Figura 27 – Exemplo de página continuação	48
Figura 28 – Formulário de venda	50
Figura 29 – Árvore Completa Resultado	51
Figura 30 – Gráfico Data Mining	52

LISTA DE CÓDIGOS

LISTA DE SIGLAS E ABREVIATURAS

AJAX	<i>Asynchronous JavaScript and XML</i>
API	<i>Application Programming Interface</i>
CSS	<i>Cascading Style Sheets</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>HyperText Transfer Protocol</i>
SD	Sistema Distribuído ou <i>Distributed System</i>
SQL	<i>Structured Query Language</i>
URL	<i>Uniform Resource Locator</i>
WS	<i>WebSockets</i>
DOM	<i>Document Object Model</i>
W3C	<i>World Wide Web Consortium</i>
SO	Sistema Operacional
NO-SQL	<i>Not Only Structured Query Language</i>
NPM	<i>Node Package Manager</i>
MVC	<i>Model View Controller</i>
MVR	<i>Model View Route</i>
JSON	<i>JavaScript Object Notation</i>
XML	<i>eXtensible Markup Language</i>
IBOPE	Instituto Brasileiro de Opinião Pública e Estatística

SUMÁRIO

1	INTRODUÇÃO.....	13
2	QUADRO TEÓRICO.....	15
2.1	Node.js	15
2.1.1	Event-Loop.....	16
2.1.2	Event-driven	17
2.1.3	<i>Threads non-block</i>	18
2.1.4	V8	18
2.2	NPM	19
2.3	Express	19
2.4	MVC	20
2.5	Web Standards	21
2.6	HTML 5	21
2.7	EJS	22
2.7.1	jQuery	22
2.7.2	Bootstrap	23
2.7.3	Web Real-time	24
2.7.4	Socket.io	24
2.8	Ajax polling	25
2.9	Ajax Long-polling	26
2.9.1	Web socket	27
2.9.2	MongoDB.....	28
2.10	Google Charts.....	29
3	QUADRO METODOLÓGICO	30
3.1	Tipo de Pesquisa	30
3.2	Contexto	30
3.3	Procedimentos	31
3.3.1	Configuração do ambiente	31
3.3.2	Configuração da <i>stack</i>	37
3.3.3	Rotas	40
3.3.4	<i>Controllers</i>	41
3.3.5	Serviços	43
3.3.6	<i>Models</i>	44
3.3.7	<i>Views</i>	45
4	RESULTADOS	49
4.1	Pesquisa acerca das dificuldades e problemas das aplicações de venda de passagem rodoviária existente na internet.....	49
4.2	Utilização das tecnologias Node.js e Socket.io para desenvolver o portal de venda de passagem.....	50
4.3	Desenvolvimento de um módulo de apoio a tomada de decisão.	51
5	CONCLUSÃO	53

Referência	55
-------------------------	-----------

1 INTRODUÇÃO

As empresas de transporte rodoviário passam por problemas recorrentes devido à venda indiscriminada de passagem, que são vendidas em excesso ou duplicidade, assim extrapolando a capacidade dos veículos, causando transtorno aos usuários. Por não haver uma forma de se prever a necessidade de veículos adicionais, muitas vezes as empresas acabam deixando de atender grandes quantidades de pessoas.

O presente trabalho apresentará o uso de uma aplicação *real-time* para venda de passagem *online*, juntamente com um módulo *data mining* para prever quando serão necessários veículos adicionais nas rotas.

A velocidade ao se obter informação sempre foi um diferencial para todas as empresas, essa grande necessidade de troca e busca de dados juntamente com a velocidade no tráfego, tem contribuído para o surgimento de novas técnicas de *web real-time* ¹.

Web *real-time* é um paradigma inovador que está mudando a forma de ver a internet, que é destinado à troca de mensagem sem a necessidade de solicitações, de forma que, quando uma informação sofre alteração, ou um determinado servidor recebe uma mensagem que afete algum usuário conectado, a aplicação do usuário é automaticamente atualizada, sem a necessidade de pedidos de verificação, tornando a solução ideal para aplicações de baixa latência ².

O paradigma *real-time*, nasceu em 2006, resultado do projeto de um anônimo italiano, filiado pela empresa Mc2labs, e que passou a ser usado comercialmente no final de 2011 pelo Twitter. Sua implementação tornou-se possível após a evolução dos navegadores, que disponibilizaram suporte a essa nova técnica, assim possibilitando estabelecer uma conexão contínua entre um navegador e um servidor através da qual ambas as partes podem começar a enviar dados a qualquer momento.

Fette (2013) aponta as vantagens do uso de aplicações *real-time* na interligação de empresas, visando a necessidade das informações em curto espaço de tempo, concluindo a eficiência comparado ao paradigma de solicitação/resposta do HTTP ³ usado atualmente.

O protocolo HTTP é o que prevalece hoje em dia na internet, e seu funcionamento é feito sob solicitações, assim quando uma aplicação necessita de atualizações é disparada

¹ Informação em tempo real pela internet.

² A latência é uma medida fundamental de desempenho da rede, pois mede a quantidade de tempo entre o início de uma ação e seu término.

³ Hyper Text Transfer Protocol

uma requisição ao servidor que espera sua resposta ser processada.

Possuindo todos os dados em tempo real, permite que as empresas tenham um controle efetivo sobre sua situação atual, fornecendo aos seus gerentes as informações necessárias para a tomada de decisão.

A tomada de decisão sempre foi um ponto crítico de todas as empresas que trabalham com densa massa de dados, pois requer pessoas qualificadas e, em certos casos, devido ao grande número de informações distintas, torna-se impossível analisar todos os dados e tomar a melhor decisão; deixa-se assim, uma brecha para o uso do *data mining*, para automatizar o processo de interpretação dos dados e analisar as possibilidades de decisões.

O presente trabalho tem como objetivo geral:

- Desenvolver um *software* de venda de passagem, em tempo real, com um módulo de *Data Mining* para apoio a tomada de decisão;

e objetivos específicos:

- Pesquisar as dificuldades e problemas das aplicações de venda de passagem rodoviária existente na internet;
- Utilizar as tecnologias Node.js e Socket.io para desenvolver o portal de venda de passagem;
- Desenvolver um módulo de apoio à tomada de decisão;

Aplicações *real-time*, juntamente com técnicas de *data mining*, são assuntos pouco recorrentes em trabalhos acadêmicos, principalmente no âmbito nacional. Com isso, esse trabalho tende a oferecer à academia uma forma diferenciada de modelagem e desenvolvimento, que atende a requisitos cuja aplicação tem a necessidade de atualizações constantes e de análise de grandes volumes de dados.

O tema foi escolhido baseado na vontade de adquirir conhecimento em um novo paradigma de desenvolvimento de software, juntamente com a vontade de inspecionar grande quantidade de dados através da programação.

Este trabalho visa oferecer à comunidade mais conforto ao efetuar a aquisição de passagens rodoviárias, e evita a necessidade de deslocamento do usuário, assim ajudando as empresas de transporte rodoviário a oferecer um serviço de maior eficácia e segurança aos seus clientes, evitando transtornos e problemas com a venda de passagem.

2 QUADRO TEÓRICO

Neste capítulo serão apresentadas as tecnologias utilizadas no desenvolvimento deste trabalho.

2.1 Node.js

The Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices (Node.js, 2013).

Segundo Pereira (2013a), os sistemas para web desenvolvidos sobre plataformas bloqueantes possuem uma característica em comum, eles paralisam um processamento enquanto utilizam um *I/O*¹ no servidor. Essa paralisação é conhecida como modelo *Blocking-Thread*².

Um sistema bloqueante enfileira cada requisição e depois a processa, não permitindo múltiplos processamentos. Enquanto uma requisição é processada, a fila fica em espera, mantendo por um período de tempo uma fila de requisições ociosas.

Esta é uma arquitetura comum existente em diversos sistemas, mostrando, de certa forma, ser ineficiente, pois perde-se grande parte do tempo ao manter essa fila ociosa enquanto se executa um processo, tal como consultar o banco de dados, carregamento de um arquivo, exemplos de tarefas que gastam uma grande parte desse tempo.

No final de 2009, Ryan Dahl com a ajuda inicial de 14 colaboradores criou o *Node.js*, visando o aumento de acessos nos sistemas web e à frequência de gargalos nos servidores web no mundo inteiro, a partir daí as empresas sentiram necessidade de fazer um upgrade nos hardwares dos servidores, e com o objetivo de resolver esses problemas, surgiu o *Node.js*.

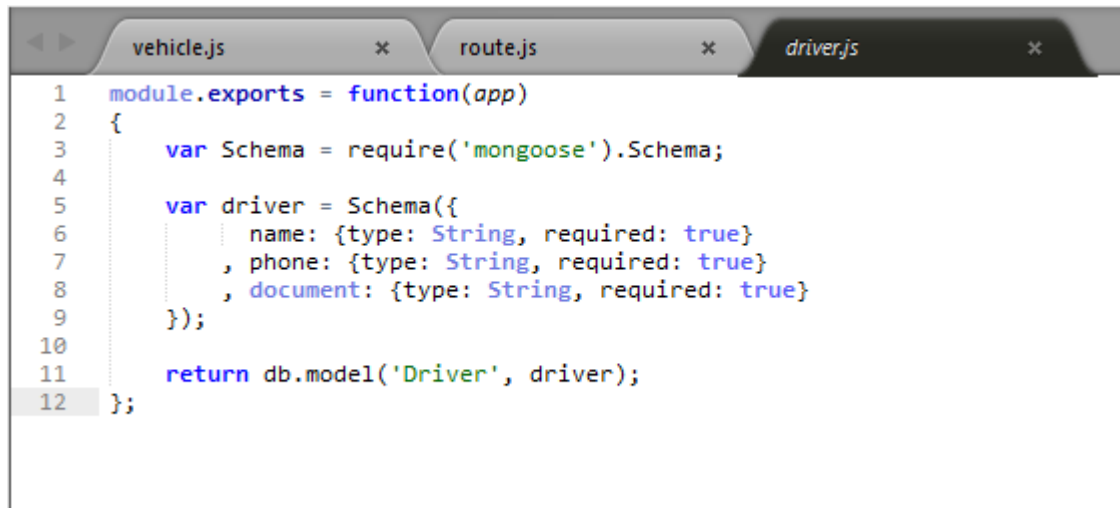
Segundo Pereira (2013b), esta tecnologia possui um modelo inovador, sua arquitetura é totalmente *non-blocking thread*, apresentando uma boa performance, com consumo de memória para máxima utilização, e de forma eficiente.

¹ Entrada e Saída.

² Thread bloqueante.

Usuários de sistemas *Node.js* estão livres de aguardarem por muito tempo o resultado de seus processos, e, principalmente, não sofrerão de *dead-locks* ³.

O *JavaScript* é a sua linguagem de programação, visando a toda facilidade e versatilidade da sintaxe dessa poderosa linguagem; tudo isso só foi possível graças à *engine JavaScript V8*, a mesma utilizada no navegador Google Chrome. Na Figura 1, tem-se uma imagem de seu código.

A screenshot of a code editor with three tabs: 'vehicle.js', 'route.js', and 'driver.js'. The 'driver.js' tab is active, showing the following JavaScript code:

```
1 module.exports = function(app)
2 {
3     var Schema = require('mongoose').Schema;
4
5     var driver = Schema({
6         name: {type: String, required: true}
7         , phone: {type: String, required: true}
8         , document: {type: String, required: true}
9     });
10
11     return db.model('Driver', driver);
12 };
```

Figura 1 – Exemplo de código Node.js **Fonte:** Elaborado pelo autor

Na Figura 1 é exibido um exemplo de modelo para persistência de dados no banco *MongoDB*.

Na JSConf Europeia em 2009, Ryan Dahl, um jovem programador, apresentou um projeto que ele estava trabalhando. Este projeto foi uma plataforma que combinado motor V8 JavaScript do Google, um ciclo de eventos, e uma API baixo nível de I/O. Este projeto não era como as outras plataformas de JavaScript do lado do servidor. O projeto recebeu aplausos de pé e desde então tem sido recebido com um crescimento sem precedentes, popularidade e adoção. Desde a sua introdução, *Node.js* tem recebido atenção de alguns dos grandes do setor. (PEREIRA, 2012)

Tudo começa quando é iniciada a aplicação e é executado o *Event-Loop* descrito a seguir.

³ Interbloqueio, ocorre quando dois processos tentam realizar mesmo procedimento, assim causando um travamento.

2.1.1 Event-Loop

O *Event-Loop* é o agente responsável por escutar e emitir eventos no sistema. Quando um evento está em execução, é possível programar qualquer lógica dentro dele e isso tudo acontece graças ao mecanismo de função *callback*⁴ do JavaScript.

Segundo Pereira (2013c), o design *event-driven*⁵ do *Node.js* foi inspirado pelos frameworks *Event Machine*⁶. Porém, o *Event-loop* do *Node.js* é mais performático porque seu mecanismo é nativamente executado de forma não-bloqueante.

Isso faz dele um grande diferencial em relação aos seus concorrentes que realizam chamadas bloqueantes para iniciar os seus respectivos *Event-loops*.

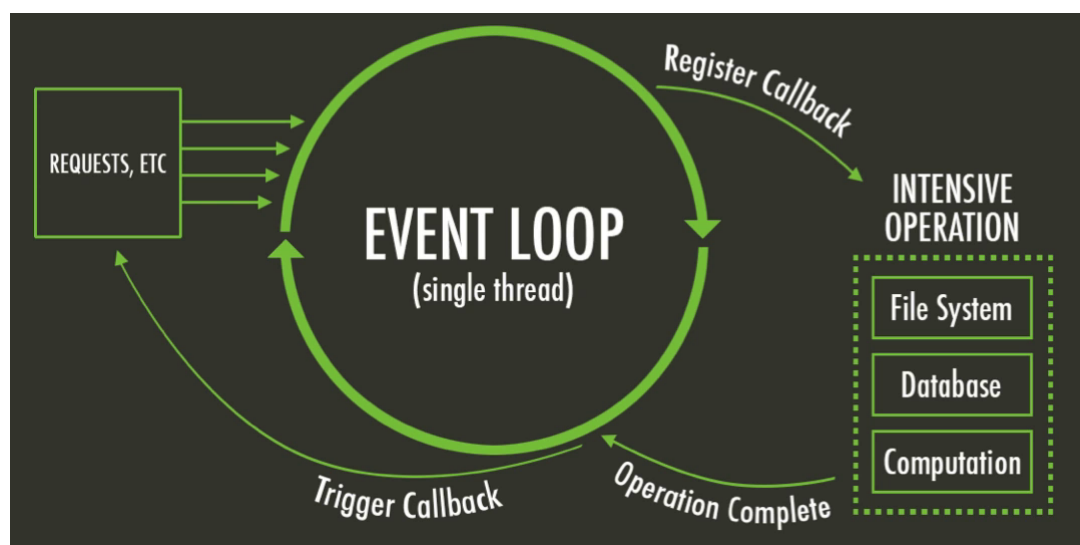


Figura 2 – Exemplo Event-loop. Fonte: www.nodejs.com

Na figura 2, foi exibido um exemplo do funcionamento do *Node.js*, quando o *event loop* recebe as requisições, enviando e criando os *callbacks* e processa os retorno.

2.1.2 Event-driven

O *Event-driven*, ou programação orientada a eventos, é descrita por Silvestre (2011), como sendo um paradigma de programação que não segue um fluxo de controle padronizado, sabendo-se que seus fluxos de controle são guiados por sinais externos.

⁴ Um *callback* é um trecho de código passado como parâmetro para um método.

⁵ Orientação a Eventos.

⁶ *Framework* orientado a eventos da linguagem Ruby.

Quando o usuário interage e faz uma requisição ao servidor, ele verifica e ativa o evento relacionado à solicitação feita a ele, assim fornecendo uma maior flexibilidade e tornando mais intuitivo o desenvolvimento baseado nesse paradigma.

2.1.3 *Threads non-block*

Threads non-block ou *Threads* não bloqueantes, em arquiteturas *multicore*⁷ e *multiprocessadas*, um recurso de programação conhecido como anidade, permite associar *Threads* a um ou mais processadores (KUNG, 2010). O objetivo deste recurso é explorar a localidade de referência aos dados na memória *cache*.

Este recurso permite utilizar a memória *cache* de maneira mais eficiente, pois ocorre menos *cache miss*⁸, pois uma *Thread* em específico será sempre executada sobre o mesmo processador ou grupo de processadores, os servidores mais novos conseguem re-aproveitar *Thread* ociosas, não é mais necessário associar cada *Thread* a uma conexão (PEREIRA,2012).

A diferença é que a *Thread* só é associada a uma conexão, quando esta tem uma requisição pendente, mostrando assim sua eficiência, pois a execução de uma *Thread* não força o travamento de toda aplicação. Essa técnica também é chamada de *Threads* assíncronas, e são muito utilizadas em aplicações de grande porte que dispõem de pouco recurso de *hardware* (PEREIRA,2012).

2.1.4 V8

A V8 é uma *engine*⁹ JavaScript *open-source*, desenvolvida em C++ que é usada no navegador Google Chrome. A V8 compila e executa o código JavaScript, diferente de outras *engines* que faz apenas a interpretação, assim torna possível lidar com alocação de memória para objetos e limpeza dos objetos obsoletos, esse é um dos grandes segredos do desempenho da V8 (PEREIRA,2013).

⁷ Multicore consiste em colocar dois ou mais núcleos de processamento em um único chip.

⁸ O *cache miss* ocorre quando o *cache* é consultado e não contém uma referência.

⁹ Motor

Em estudos realizados pela Universidade de *Stanford*, elegeram-se a V8 como *engine* mais completa e com maior desempenho, deixando para trás fortes concorrentes como Gecko, *engine* utilizada no navegador Mozilla Firefox.

O *Node.js* foi a plataforma adotada para o desenvolvimento de todo o *back-end* deste trabalho, o mesmo foi escolhido por ser uma linguagem nova, que está, cada vez mais, ganhando espaço no mercado de desenvolvimento web e por ter um maior aproveitamento do *hardware*.

2.2 NPM

NPM é o nome reduzido para *Node Package Manager* ou gerenciador de pacotes do *Node.js*, ele é um repositório online para publicações de projetos *open-source* para *Node.js*. Ele possui um terminal de linhas de comandos através do qual é possível efetuar o gerenciamento de versões e instalações de dependências (Node.js,2012).

Para efetuar a instalação de novos módulos usando o *NPM* é bem simples, basta utilizar o comando `NPM INSTALL [nome modulo]`, o mesmo será baixado e salvo no seu projeto ou pode ser instalado de uma forma global. O *NPM* também pode ser usado para instalar dependências ou módulos específicos de seu projeto, executando o comando no mesmo diretório onde encontra seu arquivo *packages.json* ¹⁰.

Segundo Pereira (2013, p.97), o *NPM* foi desenvolvido pelo próprio criador da linguagem, visando disponibilizar um repositório, para criação e armazenamento de novos módulos para *Node.js*.

2.3 Express

O Express é um módulo para *Node.js* que surgiu em 2009, logo após a grande explosão da linguagem, o mesmo veio para simplificar e aumentar o reuso de códigos desenvolvidos em *Node.js*, em aplicações em grande escala.

Segundo Teixeira (2012, p.3), utilizando-se apenas a *API HTTP* nativa, geraria códigos gigantescos, assim podendo gerar muitos *callback hell* ¹¹. Visando essa necessidade, surgiu o módulo *Express Framework*, que é um módulo para desenvolvimento de

¹⁰ Arquivo que contém as dependências da aplicação.

¹¹ Muitos callbacks alinhados.

aplicações web. Sua filosofia de trabalho foi inspirada pelo *framework* Sinatra da linguagem Ruby.

O Express possui uma lógica de roteamento de url via *callbacks* que é uma excelente *Interface Restfull*, suporta o modelo MVR ¹² integração com *SQL* ¹³ e *No-SQL* e possui nativamente um *Middleware* que pode ser configurado de maneira extremamente flexível; ele possui suporte a *helpers* dinâmicos e aceita integração com várias *Template Engines*.

O Express foi usado para criação das rotas e organização do código para trabalhar no padrão *MVC*, mesmo o *MVC* ainda não sendo nativo no Framework, ele foi usado para a organização e adaptado para o uso.

2.4 MVC

O Padrão de arquitetura *MVC* tem como objetivo principal organizar uma aplicação separando a lógica de negócio da interação com o usuário, visando aumentar a flexibilidade e o reuso de código. Segundo Reenskaung (1979, s/p), suposto criador do padrão, o *MVC* é dividido em três partes, sendo elas o *Model* ou Modelo que é responsável pelos objetos do domínio da aplicação, a *View* que é a camada de visualização, onde o usuário vai interagir com a aplicação, e o *Controller* que contém os objetos os quais definem como a interface reage a uma entrada do usuário.

De outro ponto de vista, Segundo Microsoft (2012), o padrão Model View Controller (*MVC*) é uma arquitetura projetada para separar os componentes de uma aplicação. Esta separação permite um maior controle sobre partes individuais da aplicação, tornando mais fácil desenvolver, testar e manter.

¹² Model View Rota.

¹³ Structured Query Language.

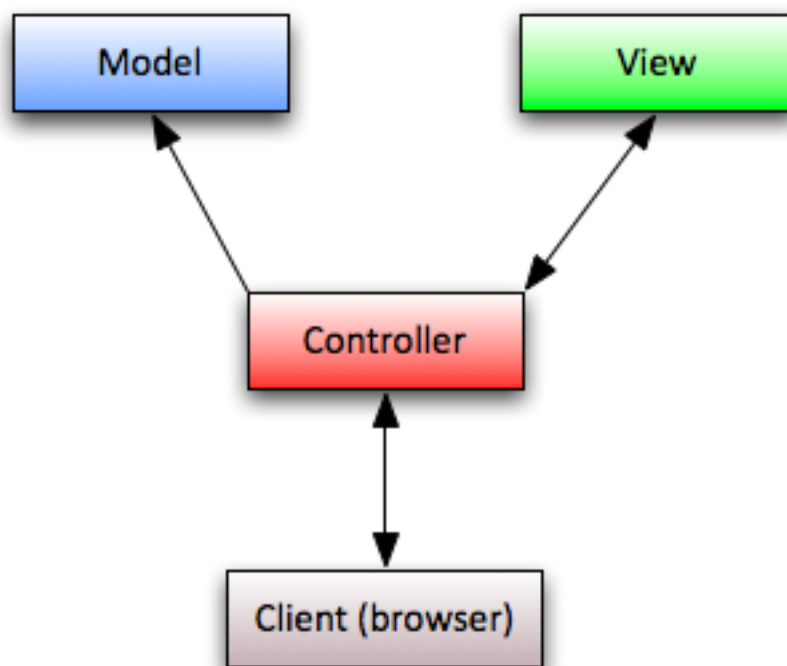


Figura 3 – Exemplo MVC. Fonte: www.caelum.com.br/mvc-java-fj11/mvc.jpeg

Na Figura 3, foi exibida a Estrutura do *MVC*, onde o cliente faz a chamada do *controller* sendo o mesmo responsável pela chamada dos *serviços*, *model*, e *views*.

2.5 Web Standards

Segundo a W3C (2014), a *Word Wide Web Consortium* mais conhecida como *W3C*, junto a outros grupos de padronização, tem estabelecido tecnologias, para criar padrões e tutoriais para conteúdos baseados na web. Essas tecnologias, chamadas de *Web Standards*, são cuidadosamente projetadas para oferecer o máximo de benefícios para o maior número de usuário da web, garantindo a viabilidade a longo prazo de qualquer documento público na internet.

Segundo a W3C (2014), Apesar dos principais desenvolvedores dos navegadores de internet estarem envolvidos com a padronização da web, desde a criação da *W3C*, por muitos anos essa conformidade não era observada. A cada nova versão de um navegador lançada no mercado, havia inúmeras falhas de suporte aos padrões, fragmentando o desenvolvimento de conteúdos para a internet e levando os designers, desenvolvedores

web, usuários e todas as empresas similares a procurarem por horas uma solução que funcionasse em todos os navegadores.

Graças ao desenvolvimento e aceitação dos padrões, atualmente a experiência de projetar, desenvolver e consumir conteúdo da internet é mais prática e rápida, garantindo uma considerável redução do custo e tempo na produção, enquanto torna os sites acessíveis à maioria das pessoas (W3C, 2013).

Serão utilizados para construir este trabalho, esses padrões e os conceitos que formam a base para o desenvolvimento de conteúdo para a internet: HTML5, JavaScript e CSS.

2.6 HTML 5

A sigla *HTML*, abreviação para a expressão inglesa *HyperText Markup Language*, que significa Linguagem de Marcação de Hipertexto, é uma linguagem de marcação utilizada para produzir páginas na Web.

Segundo a W3C (2014), o *HTML5* é a nova versão do *HTML4*, sendo agora uma recomendação. Um dos principais objetivos do *HTML5* é facilitar a manipulação do elemento possibilitando o desenvolvedor a modificar as características dos objetos de forma não intrusiva e de maneira que seja transparente para o usuário final.

Segundo a W3C (2013), ao contrário das versões anteriores, o *HTML5* fornece ferramentas para a *CSS* e o *JavaScript* fazerem seu trabalho da melhor maneira possível. O *HTML5* permite por meio de suas APIs a manipulação das características destes elementos, de forma que o *website* ou a aplicação continue leve e funcional.

O *HTML5* também cria novas tags e modifica a função de outras. As versões antigas do *HTML* não continham um padrão universal para a criação de seções comuns e específicas como rodapé, cabeçalho, sidebar, menus e etc.

Há outros elementos e atributos que sua função e significado foram modificados e que agora podem ser reutilizados de forma mais eficaz. Por exemplo, elementos como `` ou `<i>` que foram descontinuados.

O HTML 5 é usado em todas as páginas do projeto, junto com o EJS que trabalha como template engine.

2.7 EJS

O *EJS* é uma *template engine open-source* para *JavaScript*, que surgiu em 2010, desenvolvida pela empresa Jupiter4. Ele é muito parecido com Razor da plataforma .NET ou JSP que é utilizado na plataforma *Java*, o mesmo possui *Helpers* inteligentes que ajudam na geração de códigos HTML automáticos e ferramentas para iteração com variáveis.

O *EJS* proporciona também códigos mais limpos, pois permite separar seu HTML em partes, sendo possível importar um arquivo a qualquer momento utilizando a TAG correspondente à importação e todos os *Helpers* geram código em HTML5, fornecendo assim, um código mais limpo, semântico e organizado.

O *EJS* aceita o recebimento de variáveis, através de sua chamada no *back-end*. Ele foi usado em todas as páginas deste trabalho para otimização e organização.

2.7.1 jQuery

Segundo JQUERY (2012), é uma biblioteca JavaScript que simplifica a identificação de tags em documentos HTML, a manipulação de eventos, o desenvolvimento de animação e de iterações Ajax, facilitando o desenvolvimento web.

O *jQuery* se destina a adicionar interatividade e dinamismo às páginas web e proporciona facilidades como: adicionar efeitos visuais e animações, acessar e manipular o *DOM*¹⁴, buscar informações no servidor, sem necessidade de recarregar a página, prover interatividade, alterar conteúdos, modificar apresentação e estilo e simplificar tarefas específicas de JavaScript.

Ele foi criado com a preocupação de ser uma biblioteca em conformidade com os padrões web, ou seja, compatível com qualquer sistema operacional, navegador e com suporte total para a biblioteca CSS3. Essa biblioteca foi criada e está em acordo com as diretrizes do W3C, mas cabe ao desenvolvedor escrever os scripts de maneira a atender essa conformidade.

Seu uso foi em todo o front-end deste trabalho para fornecer dinamicidade às páginas e efetuar chamadas ao *back-end* da aplicação.

¹⁴ Document Object Model

2.7.2 Bootstrap

O *Bootstrap* é um *framework CSS* ¹⁵, que contém vários componentes personalizados para criação de sites e aplicativos web. Ele foi criado em 2011 pela equipe de desenvolvimento do *Twitter*, oferecendo muitos recursos para serem utilizados em aplicações web.

É sempre importante pensar na navegação do usuário ao desenvolver uma aplicação, desenvolver padrões e estar preparado para novas telas e mudanças. O *Bootstrap* foi construído para ajudar desenvolvedores a serem mais produtivos. Para isso, foi construído de maneira elegante e flexível, eliminando processos trabalhosos e permitindo maior foco no objeto a ser desenvolvido.

A solução do *Bootstrap* visa criar todos os elementos de forma modular e consentir que sejam utilizados sem dificuldades, quando necessários. Isso contribui para a customização e criação de novas telas com facilidade e torna o desenvolvimento mais produtivo.

Segundo Junior (2012) “O *Bootstrap* é responsivo, possui uma ótima documentação e traz dezenas de componentes funcionais, totalmente pronto para uso, sem falar nos plug-ins em jQuery” e realça: “Muita gente pode não concordar com a ideia de ter componentes prontos em um projeto, eu também concordo. Porém o *Bootstrap* em uma segunda vista, se mostrou muito eficiente sem desvalorizar o trabalho de um Designer”.

Isso faz com que o *Bootstrap* seja uma ferramenta para facilitar o desenvolvimento de telas, que até os mais desprovidos de habilidades de design consigam criar interfaces bonitas, oferecendo diversas classes e componentes prontos para o uso.

O *Bootstrap* foi utilizado para ajudar no desenvolvimento das interfaces, aplicando seu amigável *CSS* em todos os componentes.

2.7.3 Web Real-time

Kirkpatrick (2012) descreve em seu trabalho, a web *real-time*, sendo a busca de informações para utilizar, da forma mais rápida possível, ao contrário do paradigma de requisição, que solicita verificações periódicas de atualizações. O paradigma web *real-*

¹⁵ Cascading Style Sheets.

time mostra uma forma eficiente para troca de mensagem, em uma questão de minutos ou segundos.

A *web real-time* foi adotada por várias redes sociais, a fim de compartilhar informações com todos os usuários conectados, sem a necessidade de requisição das mesmas, mostrando-se bastante eficiente para realização desses trabalhos.

As arquiteturas deste tipo de sistema baseiam-se na arquitetura de computação distribuída em que “o modelo de comunicação que é baseado na troca assíncrona de mensagens, conhecidas como eventos” (SILVESTRE, 2011).

Existem inúmeros benefícios no uso do paradigma *web real-time*, entre eles o desacoplamento espacial e temporal dos envolvidos na comunicação, tornando-se atraente para aplicações que requerem uma melhor gerência, fornecendo assim mais flexibilidade e controle das conexões envolvidas.

Outro benefício é a grande diminuição do tráfego entre os servidores e a diminuição exponencial do uso dos recursos do servidor, assim, sendo possível proporcionar o desenvolvimento de uma aplicação com a necessidade de pouco recurso, devido ao seu funcionamento baseado em eventos e a possibilidade de uso de *Threads* não bloqueantes.

Este modelo de desacoplamento em que a informação circula livremente tem alguns problemas. Segundo Chakinala (2012 p.10), “um problema natural neste tipo de cenário é encontrar um desenho de um mecanismo eficiente que permita a distribuição de dados, desde a fonte até a origem, satisfazendo todos os requisitos da comunicação”.

2.7.4 Socket.io

A popularidade do *Node.js* veio por ele fornecer bibliotecas de baixo nível que suportam diversos protocolos (HTTP, HTTPS, DNS, TCP, FTP) como já exibido no primeiro capítulo deste trabalho. O recente protocolo *WebSockets* também é compatível com *Node.js* e ele permite desenvolver sistemas de conexão persistente utilizando *JavaScript*, tanto no cliente quanto no servidor.

O único problema em utilizar este protocolo, é que nem todos os *browsers* suportam esse recurso, tornando inviável desenvolver uma aplicação *realtime cross-browser*¹⁶.

Diante desse problema, nasceu o *Framework Socket.io*. Ele resolve a incompatibilidade entre o *WebSockets* com os navegadores antigos, emulando, por exemplo, *Ajax*

¹⁶ Aceita em todos os navegadores.

polling e Ajax long-polling ou outros transportes de comunicação em *browsers* que não possuem *WebSockets*, de forma totalmente abstraída para o desenvolvedor.

O *Socket.io* foi elaborado para sanar todos os problemas com conexão bi-direcional oferecendo o máximo de praticidade ao desenvolvedor e o máximo compatibilidade com os navegadores.

Segundo Pereira (2013a), *Socket.io* funciona da seguinte maneira. É incluído um script no cliente que detecta informações sobre o *browser* do usuário para definir qual será a melhor comunicação com o servidor.

Se o navegador do usuário possuir compatibilidade com *WebSockets* ou *FlashSockets*, será realizada uma comunicação bidirecional. Caso contrário, será emulada uma comunicação unidirecional, que em curtos intervalos de tempo faz requisições AJAX no servidor.

2.8 Ajax polling

Segundo JQUERY (2010), o *Ajax Polling* consiste em frequentes requisições de usuário ao servidor *HTTP*, executando-as em pequenos intervalos de tempo, com a finalidade de atualizar as informações dos objetos usados na sessão. Possuindo um *Polling* rápido, como baixo tempo de consulta, isto entre um segundo ou menos, tem-se a impressão de existir uma comunicação persistente.

No entanto, esta técnica não é tão eficaz, uma vez que o cliente tem a necessidade de consultar o servidor a cada segundo, ainda que não exista nenhuma nova informação. Em consequência disso, o servidor tem seu processamento perdido, com várias requisições desnecessárias de vários clientes conectados, sem dizer a desproporção entre chamadas realmente úteis, as que devolvem alguma informação para as que não as devolvem (JQUERY,2010).

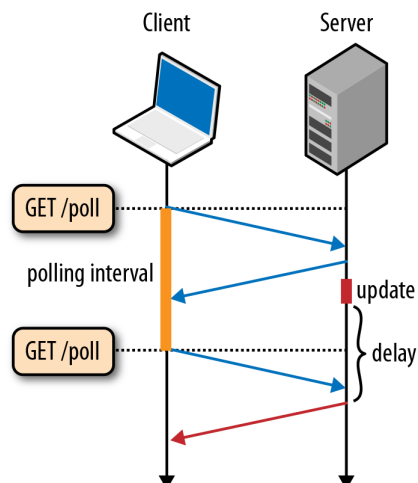


Figura 4 – Ajax Polling. Fonte: www.socket.io

Na figura 4, foi exibida uma imagem na qual mostra-se um exemplo do funcionamento do *ajax polling*.

2.9 Ajax Long-polling

Segundo JQUERY (2012), as soluções mais inteligentes foram desenvolvidas visando ao aprimoramento da técnica. Uma delas é a *long-polling*, uma evolução do método tradicional, citado anteriormente, funcionando de maneira bastante similar. Nele, em vez de o servidor responder imediatamente à requisição do cliente, ele mantém a conexão, o maior tempo possível, até que uma nova informação esteja disponível. Assim, o número de chamadas inúteis se reduz drasticamente. E imediatamente, após o cliente receber a informação, ele envia um novo pedido para o servidor, recomeçando o processo.

Porém, apesar da vantagem, a implementação sofre ainda alguns problemas, as conexões possuem um tempo limite em que podem ficar abertas, variando da configuração do servidor, que normalmente está sob o controle do administrador do *host* e a comunicação que é imediatamente fechada, após o servidor enviar as novas informações.

Na Figura 5, há um exemplo visual da técnica Ajax Long-Polling.

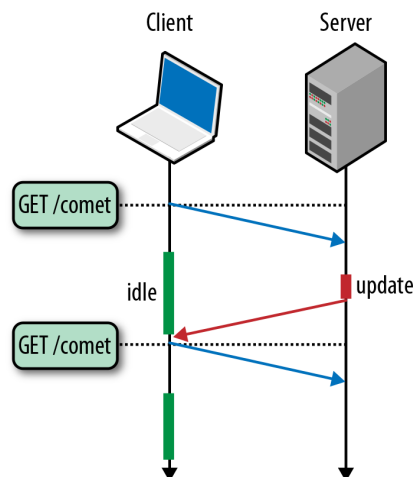


Figura 5 – Ajax Longpolling. Fonte: www.socket.io

2.9.1 Web socket

Websocket foi recentemente padronizado pela W3C e desenvolvido para superar as deficiências do *HTTP* em relação às aplicações tempo real, permite a clientes e servidores manterem uma conexão eficiente em modo *full-duplex* ¹⁷, sem a necessidade de soluções alternativas, como *plugins* de terceiros e outras técnicas pouco eficientes. Fornece, finalmente, aos desenvolvedores uma ferramenta destinada a permitir a construção de aplicativos que possam utilizar de um ambiente próximo a um tempo de resposta con-
dizente ao real. Na Figura 6, há um exemplo visual.

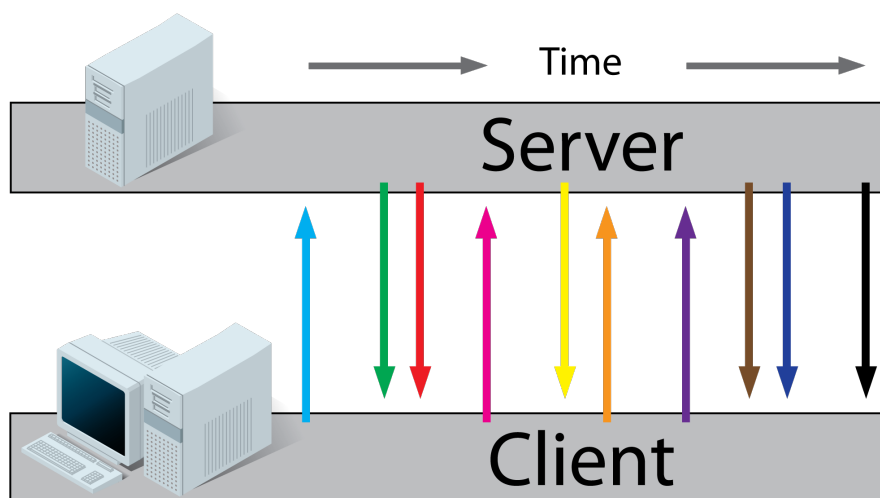


Figura 6 – Exemplo WebSocket. Fonte: www.caelum.com.br

¹⁷ É um sistema de comunicação composto por dois interlocutores que podem comunicar entre si em ambas direções.

O *WebSocket*, porém, ainda está em constante desenvolvimento, dificultando a sua implementação, este é mantido e encontrado até mesmo nos navegadores mais recentes. Sendo, portanto, uma ferramenta emergente, a fim de proporcionar uma experiência similar, ainda são utilizadas técnicas, ou plugins externos, para emular sua funcionalidade em navegadores nos quais ele não ainda não foi implementado.

O *WebSocket* "permite maior interatividade entre um navegador e um web site, viabilizando a disponibilização de conteúdos em tempo real. Isso se torna possível pelo provimento de um meio para o servidor enviar conteúdo para o navegador sem que tenha havido uma requisição prévia pelo mesmo, bem como pela possibilidade de envio/recebimento de mensagens mantendo a conexão aberta"(LOPES, 2013).

"O protocolo *WebSocket* especifica a interface de comunicação entre a aplicação web e o servidor". Enquanto "a interface de *WebSocket* API é a responsável pela comunicação entre o navegador e a aplicação web"(HAMALAINEN, 2012).

O protocolo é composto por duas partes o *Handshake*¹⁸ é o processo pelo qual duas máquinas afirmam uma a outra que a reconheceu e está pronta para iniciar a comunicação. e a transferência de dados. Segundo Fette (2013), "O handshake é o cabeçalho enviado ao cliente que informa que a transferência de dados vai começar". Esta transmissão de dados é denominada *FRAME*. "Sendo uma comunicação bidirecional cada lado da comunicação pode enviar dados a qualquer momento, diferente do *HTTP*, onde o servidor não consegue enviar dados para o cliente sempre o desejar"(FETTE, 2013).

2.9.2 MongoDB

O *MongoDB* é um banco de dados open-source, que segue o modelo *NoSQL*, o mesmo foi desenvolvido em *C++* e possuindo uma velocidade acima dos outros bancos *SQL*, acompanhado de uma característica escalável.

O *MongoDB* foi desenvolvido por volta de 2007, por uma empresa chamada 10gen que começou a projetá-lo junto a uma plataforma orientada a eventos, buscando criar uma base de dados rápida e altamente escalável para acompanhar suas aplicações.

Chodorow, Dirolf (2010), comentam que o *MongoDB* é uma base de dados escalável, flexível e poderosa, combinada com muitas das características mais úteis dos bancos de dados relacionais, como índices secundários, consultas de intervalo e classificação.

¹⁸ ou aperto de mão

Segundo Mongolab (2013), o *MongoDB* trabalha com o modelo de *MapReduce*, um modelo de programação paralela em forma de *cluster*, que fornece uma rápida recomposição da base de dados, trabalha também com o modelo *Sharding* que é a divisão dos dados horizontalmente com tendência a quebrar as tabelas e diminuir o número de linhas, separando-se, assim, em diferentes ambientes.

O *MongoDB* trabalha com arquivos no formato *JSON*, nos quais efetua a conversão do *JSON* em *BSON* que é o mesmo, em formato binário; diferente dos bancos *SQL* que armazenam os dados em linhas de dados relacionais. O *MongoDB* possui índice em qualquer atributo, sendo altamente replicável, fornece uma alta disponibilidade.

O *JSON* permite complexas estruturas de dados serem representadas em um simples texto, que pode ser facilmente lido pelo ser humano, geralmente é considerado mais fácil do que *XML*. Sabe-se que *XML*, *JSON* foi visto como uma maneira de trocar dados entre clientes e aplicações *web*, quando utilizado de maneira correta, pode descrever objetos, é essa simplicidade que influencia muitos programadores a utilizá-lo.

O *MongoDB* foi a base de dados usada pela aplicação, foi escolhida devido a sua grande capacidade escalável e por ser uma das bases de dados com o maior custo benefício da atualidade.

2.10 Google Charts

O *Google Charts* é uma ferramenta *open-source*, desenvolvida pela *Google* em 2013, que possibilita a geração de gráficos, desde de os mais simples aos mais complexos, como mapas e árvores hierárquicas, fornecendo um grande leque de opções de gráficos dos mais variados tipos.

Segundo Fette (2013), Os gráficos são processados usando tecnologia *HTML5/SVG* para fornecer compatibilidade *cross-browser*, incluindo versões mais antigas do *IE* e portabilidade entre plataformas para *iOS* e *Android*.

Todos os tipos de gráficos fazem uso do *Data Tables* para o preenchimento de dados, tornando-se mais fácil a alternância entre estes tipos. *DataTable* fornece métodos para a classificação, modificação e filtragem de dados, e pode ser preenchido diretamente de sua página web, de um banco de dados ou qualquer provedor de dados.

3 QUADRO METODOLÓGICO

Neste capítulo são apresentados os procedimentos utilizados para a realização do projeto de pesquisa. Segundo Fulgêncio (2009), a metodologia tem como objetivo descrever os procedimentos necessários para a realização de uma pesquisa.

Os procedimentos escolhidos serão apresentados, iniciando-se pelo tipo de pesquisa.

3.1 Tipo de Pesquisa

Para o desenvolvimento do presente trabalho, existe a necessidade do uso de uma metodologia de pesquisa, com o objetivo de investigar as possíveis melhorias no processo de venda de passagem rodoviária e previsão de demanda de veículos. Visando à metodológica foi usado o tipo de pesquisa aplicada.

A pesquisa aplicada, para Fulgêncio (2010, p. 476), "é uma investigação original concebida pelo interesse de adquirir novos conhecimentos" para Franceschini et al. (2012, p.58): "A pesquisa aplicada por metodologia entende-se a aplicação de técnicas e procedimentos para a coleta, descrição e interpretação de dados, os quais devem gerar informações".

3.2 Contexto

As empresas de transporte rodoviário são empresas que fornecem o serviço de transporte coletivo, e através do processo de licitação são fechados contratos com municípios ou estado para que haja o transporte da população.

Essas empresas têm traçadas rotas que cobrem praticamente todo o território nacional e alguns países de fronteira, sendo cerca de 10 mil dessas empresas para atender a uma massa de 36 milhões de brasileiros, que dependem do transporte, abrangendo uma taxa de 15 milhões de embarques por dia, segundo o IBOPE ¹.

¹ Instituto Brasileiro de Opinião Pública e Estatística

Segundo o IBOPE (2014), a internet está ao alcance de 58% da população brasileira e cerca de 30% dessas pessoas efetuam transações e compras utilizando seus serviços disponíveis, um dado apontando que o brasileiro, aos poucos, perde o receio de efetuar esses tipos de transações, o que possibilita o surgimento de novas soluções no mercado.

Observando o grande número de pessoas com acesso à internet, o autor deste trabalho decidiu focar a pesquisa em soluções para a venda de passagem e previsão de demanda de transporte rodoviário, procurando assim, gerar um software que, uma vez escrito, fosse capaz de atender o maior número possível de empresas nesse segmento.

3.3 Procedimentos

Seguem os procedimentos utilizados para o desenvolvimento do presente trabalho.

3.3.1 Configuração do ambiente

Para a realização deste trabalho, foi configurado o ambiente de desenvolvimento que consiste na instalação das ferramentas e dos módulos necessários para a organização da aplicação no padrão MVC e na instalação e configuração do banco de dados *MongoDB*.

Inicialmente, foi efetuada a instalação do *MongoDB*, obtido em seu próprio site ², logo após seu download foi extraído o conteúdo em uma pasta, e seu executável responsável pela execução do banco de dados, foi adicionado na inicialização do sistema operacional, assim, toda vez que o mesmo é carregado, automaticamente o banco de dados já é acionado, ficando disponível para aplicação.

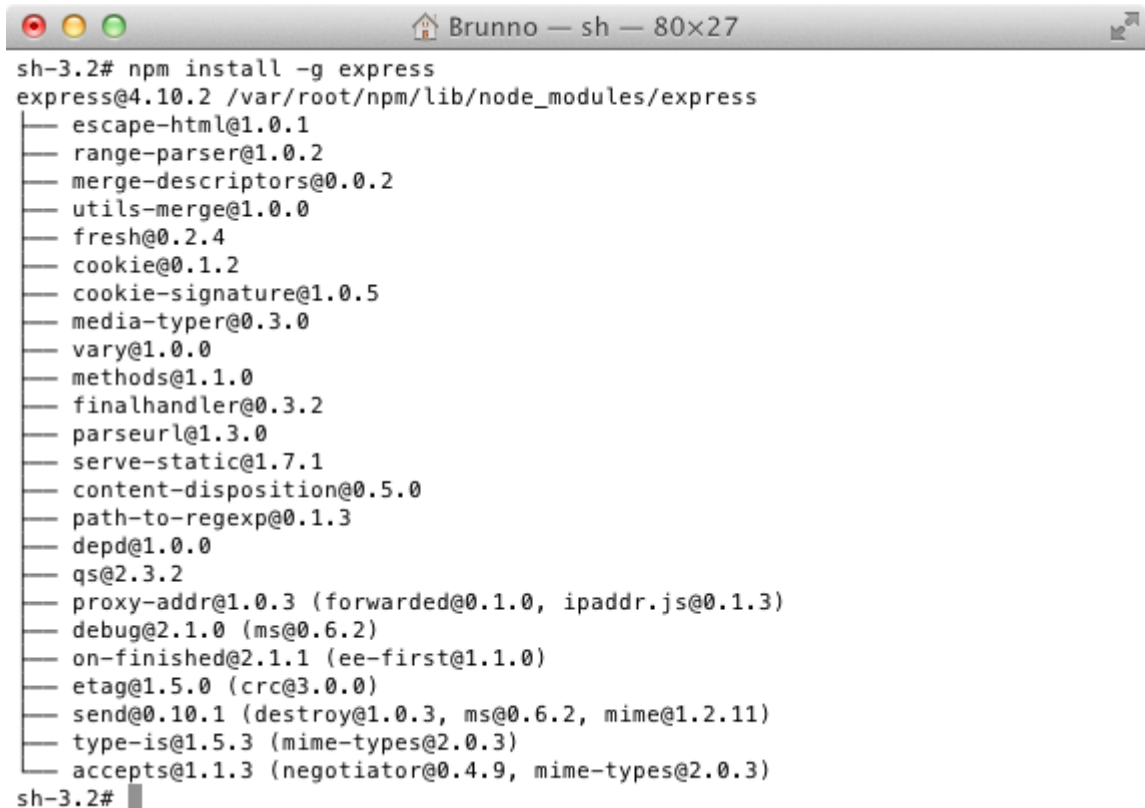
Na continuação, foi utilizado o próprio terminal do MongoDB disponível na sua pasta, com nome de *mongod.exe*, e a partir dele, foi possível executar comandos para visualização de seus bancos, execuções de consultas para obtenção de dados, criação de *collections* e alterações nos conteúdos.

Depois da configuração do banco, foi instalado o *Node.js*, obtido em seu próprio site ³. Após sua instalação foi automaticamente configurado seu compilador e gerenciador de pacotes NPM, habilitando, assim, seus comandos de terminal.

² www.mongodb.org

³ www.nodejs.org

Usando linha de comando *Node.js*, foi instalado o Express, módulo utilizado para geração da árvore de diretórios, gerenciamento de rotas e organização da aplicação. Sua instalação foi realizada utilizando o comando "npm install -g express", conforme exibido na Figura 7.



```
sh-3.2# npm install -g express
express@4.10.2 /var/root/npm/lib/node_modules/express
├── escape-html@1.0.1
├── range-parser@1.0.2
├── merge-descriptors@0.0.2
├── utils-merge@1.0.0
├── fresh@0.2.4
├── cookie@0.1.2
├── cookie-signature@1.0.5
├── media-typer@0.3.0
├── vary@1.0.0
├── methods@1.1.0
├── finalhandler@0.3.2
├── parseurl@1.3.0
├── serve-static@1.7.1
├── content-disposition@0.5.0
├── path-to-regexp@0.1.3
├── depd@1.0.0
├── qs@2.3.2
├── proxy-addr@1.0.3 (forwarded@0.1.0, ipaddr.js@0.1.3)
├── debug@2.1.0 (ms@0.6.2)
├── on-finished@2.1.1 (ee-first@1.1.0)
├── etag@1.5.0 (crc@3.0.0)
├── send@0.10.1 (destroy@1.0.3, ms@0.6.2, mime@1.2.11)
├── type-is@1.5.3 (mime-types@2.0.3)
└── accepts@1.1.3 (negotiator@0.4.9, mime-types@2.0.3)
sh-3.2#
```

Figura 7 – Instalação Express. Fonte: Elaborado pelo autor

O comando NPM ativa o gerenciador de pacotes do node; o comando INSTALL ativa o instalador de pacotes que busca o nome Express em sua base de dados de módulos; chegando a URL, efetua o download e instalando de maneira global.

Depois da instalação do Express, foi configurado o *Mongoose*, módulo responsável pela conexão com *MongoDB*, criação do mapeamento de objetos para MongoDB e para utilização do banco de dados, de forma *schema-less*, ajudando na criação dos modelos a serem persistidos no banco. O mesmo também foi instalado, de maneira global, conforme a Figura 8.

```
Brunno — sh — 80x25
sh-3.2# npm install -g mongoose

> kerberos@0.0.4 install /var/root/npm/lib/node_modules/mongoose/node_modules/mongod
ngodb/node_modules/kerberos
> (node-gyp rebuild 2> builderror.log) || (exit 0)

\
> bson@0.2.15 install /var/root/npm/lib/node_modules/mongoose/node_modules/mongo
db/node_modules/bson
> (node-gyp rebuild 2> builderror.log) || (exit 0)

mongoose@3.8.19 /var/root/npm/lib/node_modules/mongoose
├── regexp-clone@0.0.1
├── sliced@0.0.5
├── muri@0.3.1
├── hooks@0.2.1
├── mpath@0.1.1
├── mpromise@0.4.3
├── ms@0.1.0
├── mquery@0.8.0 (debug@0.7.4)
└── mongodb@1.4.12 (readable-stream@1.0.33, kerberos@0.0.4, bson@0.2.15)
sh-3.2#
```

Figura 8 – Instalação Mongoose. Fonte: Elaborado pelo autor

Depois de serem instalados os módulos globais, foi gerado, a partir do Express, o projeto base, contendo a árvore de estrutura inicial para o desenvolvimento da aplicação, como sua árvore de diretórios e seu *package.json*.

Para a geração da estrutura básica, foi utilizado o comando "Express node-transportcont -ejs". O comando *Express* é responsável por ativar o módulo *Express* junto com o nome do projeto e o comando *-ejs*, que ativam o projeto para o uso da *EJS template engine*. Após a finalização, foi criada a árvore de diretórios, conforme a Figura 9.

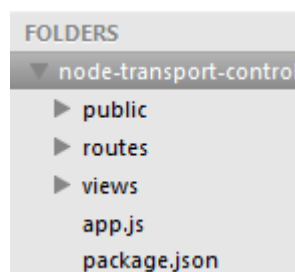


Figura 9 – Árvore. Fonte: Elaborado pelo autor

A árvore de diretórios é gerada visando manter a aplicação, de forma organizada e escalável, possibilitando a adição de novos diretórios em sua estrutura. A árvore inicial consiste nos seguintes diretórios e arquivos:

- *Public*: Responsável pelo conteúdo estático da aplicação, tais como arquivos de *front-end*, imagens, dentre outros.
- *App.js*: Responsável pela inicialização do servidor, através do comando `node app.js`.
- *Routes*: Diretório responsável por armazenar as rotas da aplicação.
- *Views*: Diretório responsável pelas telas da aplicação.

Além dos diretórios citados, foram criados outros, conforme será exibido abaixo. Junto a essa série de diretórios, foi criado também um arquivo chamado *package.json*, arquivo que contém as principais informações da aplicação, como versão do projeto, controle dos módulos a ser buscado pelo comando "`NPM INSTALL`", sendo assim, possível atualizar módulos ou bloquear a atualização.

Na Figura 10, tem-se o arquivo *package.json* que será detalhado.

```

{
  "name": "TCC-Transport ",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "3.4.7",
    "express-load": "1.1.8",
    "ejs": "0.8.5",
    "socket.io": "0.9.16",
    "mongoose": "3.8.4"
  },
  "description": "first",
  "main": "web.js",
  "devDependencies": {},
  "repository": {
    "type": "git",
    "url": "git@heroku.com:TCC-Transport:"
  },
  "keywords": [
    "7478564buh"
  ],
  "author": "brunno",
  "license": "ISC",
  "subdomain": "buh159-TCC-Transport",
  "engines": {
    "node": "0.10.x"
  }
}

```

Figura 10 – Package.json. Fonte: Elaborado pelo autor

Inicialmente, foi configurado o atributo *name* que descreve o nome do projeto, a ser chamado via função `require('nome')`. Em seguida, é adicionada uma descrição no atributo *description* (descrição da aplicação), logo depois, foi configurado o autor ou desenvolvedor da aplicação.

A versão da aplicação é o atributo principal do *package.json*, pois sem ela é impossível instalar os arquivos via comando NPM.

Nas configurações realizadas acima foi setado o atributo *private* como falso, dizendo que a aplicação é *open-source* e futuramente pode ser liberada para download no site do NPM.

A configuração de versão é dividida em três níveis: Maior(0) Menor(0) e Patch(1).

O primeiro módulo foi configurado de forma a aceitar a atualização de *patch* setado pelo caractere `*` que geralmente recebe apenas melhorias e correções.

Foi utilizada também a opção `"*"`, que permite qualquer tipo de atualização, se houver alguma mudança ou depreciação de algo em uso na aplicação, poderá causar o travamento da aplicação.

O ambiente básico para desenvolvimento node foi configurado, mas para a realização do presente trabalho foi criado um ambiente *MVC* com alocação de serviços. Para a configuração desse ambiente foram adicionados mais quatro diretórios:

- *Model*: Responsável pela persistência dos modelos no banco de dados.
- *Controller*: Consiste nos controladores da *view*, toda chamada para persistência de dados ou busca de informação é realizada sobre esse arquivo.
- *Service*: Diretório responsável pelos serviços, nele consiste a conexão com a base de dados, possibilitando a recuperação de um modelo ou sua inserção.
- *Socket*: Que consiste na configuração do *socket* da aplicação.

Na Figura 11 tem-se a imagem da árvore de diretórios completa.

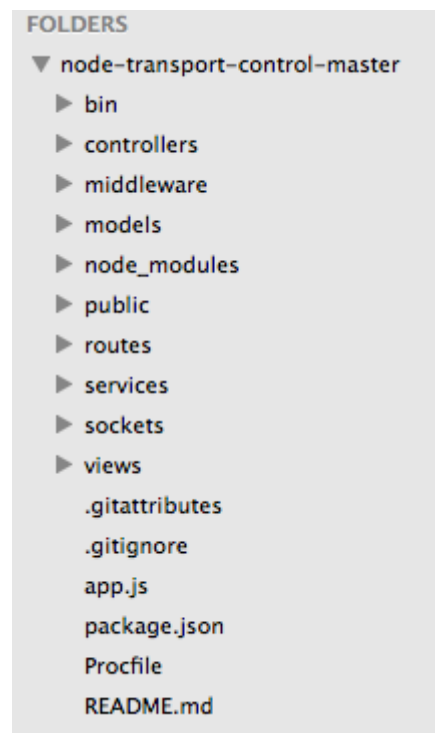


Figura 11 – Árvore Completa. Fonte: Elaborado pelo autor

Depois da criação dos diretórios, na árvore da aplicação, foi necessário adicionar a chamada de cada diretório pelo Express-Load na stack de configuração, comentário a ser feito no próximo tópico.

3.3.2 Configuração da *stack*

A stack é o local onde se concentra toda a configuração de inicialização, tais como conexão com banco de dados, arranque do servidor, configurações de diretórios, dentre outras.

Sua configuração foi dividida em trechos de código para melhor explicar sua aplicação no projeto, inicialmente foi efetuado o detalhamento das chamadas dos módulos, conforme a Figura 12.

```
1 var express = require('express')
2   , app = express()
3   , load = require('express-load')
4   , error = require('./middleware/error')
5   , mongoose = require('mongoose')
6   , server = require('http').createServer(app)
7   , io = require('socket.io').listen(server);
```

Figura 12 – Require Inicial. Fonte: Elaborado pelo autor

Na linha 1 foi carregado o módulo *Express* na variável *express*, possibilitando usar os recursos do módulo, como criação de servidores, carregamento de módulos via diretório.

Na linha 2 foi executado o Express e atribuído a variável *app*, tornando disponível as configurações básicas de um servidor node.

Na linha 3 foi carregado o módulo de *load* do *Express* que permite fazer o carregamento de diretórios, tornando possível acessar seus arquivos em uma estrutura modular, recurso utilizado para emular o funcionamento do modelo *MVC*, devido a ele não ser nativo no *Express*.

Na linha 4 foi carregado o *middleware*, módulo que contém configurações de permissões, tratamento de erros do servidor e tratamento de erros *HTTP*, que será detalhado abaixo.

Na linha 5 foi efetuada a chamada do módulo *mongoose*, tornando disponível na variável *mongoose* os métodos como conexão com banco de dados, criação de esquemas para persistência de dados, entre outros métodos utilizados, no decorrer da aplicação.

Na linha 6 foi carregado o módulo *HTTP*; após o carregamento foi chamado seu método *createServer*, passando a variável com as configurações previamente definidas na chamada do *Express*. Tornando assim disponível um servidor *HTTP*, com as configurações básicas definidas, efetuar a chamada do módulo *Express*.

Na linha 7 foi carregado o módulo *Socket.io* e depois foi chamado o método *listen*, que torna a aplicação disponível com a configuração importada acima do módulo *HTTP*.

```
9  global.db = mongoose.connect(
10    'mongodb://localhost/node-transport',function(error){
11    if (error)
12    {
13      console.log(error);
14    }
15    else
16    {
17      console.log("Banco OK!");
18    }
19  });
```

Figura 13 – Conexão Banco. Fonte: Elaborado pelo autor

Na linha 9 foi declarada uma variável global com o nome *db*, visando sua disponibilidade no decorrer de toda aplicação, logo após foi efetuada a chamada do método *connect* passando por parâmetro a url do *MongoDB*.

Em seguida, via *callback* da conexão, foi adicionada uma verificação para detectar se houve retorno de erro na variável *error*, caso contrário apenas será exibido que a conexão foi efetuada.

```
17  app.set('views', __dirname + '/views');
18  app.set('view engine', 'ejs');
```

Figura 14 – Configuração Diretório. Fonte: Elaborado pelo autor

Na linha 17 foi setado o caminho das views, sendo a constante *__dirname*, o nome do diretório base e o *"/views"* o diretório onde se encontram as telas da aplicação.

Na linha 18 foi setado a *template engine EJS*, fazendo com que, antes de retornar a *view* para ser renderizada pelo *browser*, ela passe pelo template engine para que seja montada, adicionado cabeçalho, rodapé e outros tratamentos com variáveis para que depois a *view* seja retornada para o *browser*.

```
20 app.use(express.cookieParser('      '));
21 app.use(express.session());
22 app.use(express.json());
23 app.use(express.urlencoded());
24 app.use(express.methodOverride());
25 app.use(app.router);
```

Figura 15 – Configuração App. Fonte: Elaborado pelo autor

Na linha 20 foi acionado o uso de cookies com nome de "node-transport", fazendo assim o armazenamento.

Na linha 21 foi acionado o recurso da *session*, sessão de usuário conectado junto a aplicação, permitindo o armazenamento de dados e fazendo o controle de acesso da aplicação.

Na linha 22 foi habilitado o tráfego de *Json* na aplicação.

Na linha 23 foi ativado o *urlencoded*, ativando assim um recurso que converte todos os caracteres não alfa-numéricos em um sinal de por cento, seguido por dois dígitos hexadecimais.

Na linha 25 foi ativado o recurso que envia automaticamente todas as chamadas do browser ao servidor, ao serviço de rotas, onde ela é filtrada e tratados seus paramentos, se necessário.

```
27 app.use(express.static(__dirname + '/public'));
```

Figura 16 – Configuração Estatica. Fonte: Elaborado pelo autor

Na linha 27 foi habilitado o diretório estático da aplicação, onde contém os arquivos *CSS*, *JavaScript* e imagens que serão visualizados pelo cliente, no lado *front-end* da aplicação.

```
29 app.use(error.notFound);
30 app.use(error.serverError);
```

Figura 17 – Árvore Completa. Fonte: Elaborado pelo autor

Na linha 29 foi setado o nosso tratamento de erros, no caso de página não encontrada ou acesso às rotas inexistentes na aplicação. Seu funcionamento será detalhado no parágrafo sobre *middleware*.

Na linha 30 foi setado o tratamento de erros, no caso de falhas no servidor para que seja exibido uma página mais amigável ao usuário. Seu funcionamento será detalhado no parágrafo sobre *middleware*.

```
32 load('models')
33   .then('services')
34   .then('controllers')
35   .then('routes')
36   .into(app);
```

Figura 18 – Load Arquivos. Fonte: Elaborado pelo autor

Nas linhas 32 a 36 foi chamado método `load()` do Express para carregar os respectivos diretórios, para que a aplicação possa funcionar no modelo MVC, tornando disponível na variável global da aplicação a `app`, as funções contidas nos arquivos dos diretórios.

```
38 server.listen(3000,function(){
39   console.log("Transport Control Online.");
40 });
```

Figura 19 – Adição Servidor. Fonte: Elaborado pelo autor

Na linha 38 foi iniciado o servidor na porta 3000, passado também um *callback* para que notifique no terminal, via comando `console.log()`. A aplicação está disponível para conexão.

3.3.3 Rotas

As rotas foram utilizadas nesse trabalho para efetuar o redirecionamento das chamadas ao servidor, enviando os dados da chamada ao seu respectivo controlador, criando um ambiente organizando e possibilitando o desacoplamento da aplicação.

O sistema de rotas, basicamente consiste em métodos que recebem como argumento o servidor *app*, contendo o sistema de rotas ativado acima na *stack* de configuração e dentro desse método é tratada e redirecionada cada chamada.

Para a realização do presente trabalho foram utilizados chamadas GET e POST. As chamadas GET para simples navegação e chamadas para renderização de páginas, também as chamadas POST para envio de objetos JSON.

Na Figura 20, tem-se um exemplo de um arquivo contendo as rotas.

```
1 module.exports = function(app)
2 {
3     var users = app.controllers.users;
4     app.get('/signin', users.signin);
5     app.post('/newUser', users.newUser);
6     app.get('/myTravels', users.myTravels);
7     app.post('/login', users.login);
8     app.post('/logout', users.logout)
9 };
```

Figura 20 – Exemplo Rotas. **Fonte:** Elaborado pelo autor

Na linha 1 foi criado o módulo recebendo como parâmetro o servidor *app*.

Na linha 3 foi carregado na variável *users* no *controller* setado acima na *stack* de configuração, deixando assim disponível seus métodos na variável *textitusers*.

Na linha 4 foi adicionada uma rota sinalizando que, quando o servidor receber uma chamada na rota *signin*, será redirecionado para o método *signin* do *controller*, carregá-lo logo acima.

3.3.4 *Controllers*

Os *controllers* são os responsáveis por renderizar e fornecer suporte às *Views* da aplicação, nele encontra de simples validações a chamadas de serviços para persistência ou recuperação de dados.

Após ser chamado pela rota, o *controller* recebe como parâmetro um objeto *req*, contendo os dados da requisição e um objeto *resp*, contendo os dados para o retorno a chamada.

Na Figura 21, tem-se a imagem de um controller utilizado na aplicação.

```

1  module.exports = function(app)
2  {
3
4      var Users = app.services.users;
5      var isLogged = require('../middleware/isLogged');
6
7      var UsersController =
8      {
9
10         login: function(req,res)
11         {
12             var email = req.body.email;
13             var password = req.body.password;
14
15             var user = Users.getLogin(email,password, function(user){
16
17                 if (user)
18                 {
19                     req.session.user = user;
20                     res.redirect('/')
21                 }
22                 else
23                 {
24                     params =
25                     {
26                         error: "Usuário não localizado. :/"
27                     };
28
29                     res.render('home/index', params);
30                 }
31             });
32         }
33     };
34
35     return UsersController;
36
37 };
38

```

Figura 21 – Exemplo Controller. Fonte: Elaborado pelo autor

Na linha 1 foi criado o módulo recebendo como parâmetro o servidor app.

Na linha 4 foi declarada a variável *Users* recebendo os serviços de usuários, sendo os métodos de recuperação, deleção e persistência de usuários.

Na linha 5 foi declarado uma variável que recebe a chamada do *middleware*, onde localiza o método que é verificado o usuário na sessão e outros tratamentos.

Na linha 8 foi declarado a classe *UsersController* que contém os métodos do controller, ela será retornada quando for efetuada a chamada *app.controller.user*.

Na linha 10 foi criado o método responsável pelo redirecionamento para a tela de cadastro de usuários da aplicação, esse método é chamado pela rota de usuário que envia como parâmetro o Req e Resp que será tratado abaixo.

Na linha 14 foi efetuada a chamada do *middleware*, enviando como parâmetro o *req*, arquivo, contendo a requisição do cliente ao servidor, a mesma é processada pelo

middleware e se o usuário estiver na sessão, ele é setado a variável *param* instanciada na linha 14.

Na linha 18 foi chamado o método *render()*, responsável por retornar o caminho das *Views* e o parâmetro que ela receberá para ser populada ou tratada.

Na linha 21 foi criado o método responsável por efetuar o login e o redirecionamento para página inicial da aplicação, esse método é chamado pela rota de usuário que envia como parâmetro o *Req* e *Resp* da mesma forma que na linha 10.página

Nas linhas 23 e 24 foram declarados as variáveis *email* e *password* que recebem os dados enviado via parâmetro *Req*.

Na linha 26 foi chamado o método *getToLogin()* do serviço de usuário, e em seu *callback* recebe o usuário como parâmetro.

Nas linhas 23 a 30 foi adicionado uma verificação que, se o usuário for localizado, ele é setado na sessão e redirecionado para pagina *index* da aplicação.

Nas linhas 31 a 37 se não for localizado o usuario, adiciona-se uma mensagem via parâmetro *error* que será tratada pela *View*, após ser retornado.

3.3.5 Serviços

Os serviços são responsáveis por buscar os dados e retornarem aos *controllers*, esses métodos são os de busca, criação, alteração e deleção, conforme exibido na Figura 22.

```

1  module.exports = function(app)
2  {
3      var User = app.models.user;
4      var UsersService =
5      {
6          get: function(_id, callback)
7          {
8              },
9          create: function(users, callback)
10         {
11             },
12         update: function(users)
13         {
14             },
15         delete: function(users)
16         {
17             }
18     };
19     return UsersService;
20 };

```

Figura 22 – Serviço Vazio. Fonte: Elaborado pelo autor

3.3.6 Models

Os *Models* são arquivos onde contém *schemas* para padronizar a persistência dos objetos no banco. Nesses *Models* foram setadas configurações como tipo de dado persistido, campo obrigatório.

Foram criados os *Models*, seguindo o padrão da Figura 23.

```

1  module.exports = function(app)
2  {
3      var Schema = require('mongoose').Schema;
4
5      var myTravel = Schema({
6          destination: String
7          , origin: String
8          , date: Date
9          , price: String
10     });
11
12     var user = Schema({
13         user: {type: String, required: true}
14         , email: {type: String, required: true}
15         , password: {type: String, required: true}
16         , myTravels: [myTravel]
17     });
18
19     return db.model('user', user);
20 };

```

Figura 23 – Modelo Banco. Fonte: Elaborado pelo autor

Na linha 3 foi declarada uma variável com nome de *Schema*, recebendo o objeto *Schema* chamado do módulo *mongoose*, tendo assim disponível a geração de modelos para serem salvos e recuperados do *MongoDB*.

Nas linhas 5 a 10 tem-se um exemplo da geração dos modelos, onde é chamado o *Schema*, e disponibilizada a opção de setar tipo e obrigatoriedade dos campos, gerando assim um objeto modelo.

Nas linhas 12 a 17 foi efetuado o mesmo processo acima, exceto na linha 16, que foi adicionada uma lista de objetos *myTravel*, por não existir relacionamento, o objeto *user* passa a conter uma lista de objetos *myTravel*, assim, quando for necessário recuperar um usuário, conter-se-á uma lista de objetos para que possa ser trabalhada.

Na linha 19 foi retornada a adição, no banco de dados, do objeto *user* com o nome *User* para que possa ser utilizado no decorrer da aplicação.

3.3.7 Views

Para criação das views foi utilizado *HTML5* e o *EJS* como *template engine*. Inicialmente, foi criado o *template* básico e utilizado em toda a aplicação; dividido em um

Header e *Footer*, são importados em todas as páginas da aplicação.

Na Figura 24, tem-se uma imagem da árvore de diretórios das views.

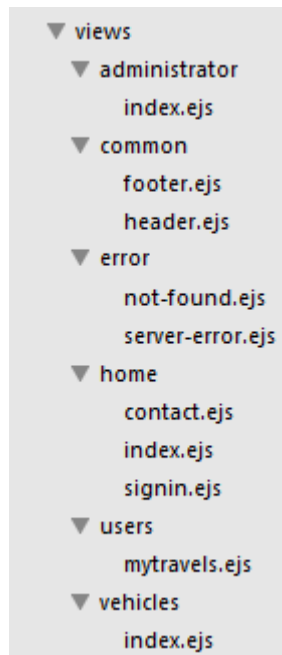


Figura 24 – Árvore View. Fonte: Elaborado pelo autor

Na divisão da árvore há um diretório common, onde estão os arquivos que são comuns em todas as *views* da aplicação, contendo chamadas a *JavaScript* como *jQuery* e nosso *JavaScript* global.

Logo depois, existe um diretório error onde contém todas as páginas de erro da aplicação, os mesmos estão nomeados com o nome do erro, seguido da extensão do *EJS*.

E depois existem os demais diretórios referentes às *views* de cada módulo da aplicação, todos seguindo o mesmo padrão importando o cabeçalho, *layout* e rodapé que será exibido na Figura 25.


```

1  <% include ../common/header %>
2
3  <div class="container">
4      <address>
5          <strong>BRDev, Inc.</strong><br>
6          255 Lodonio F Cortez, Ap 6<br>
7          Pouso Alegre, MG 37550-000<br>
8          <abbr title="Phone">P:</abbr> (35) 3422-6804
9      </address>
10
11     <address>
12         <strong>Email</strong><br>
13         <a href="mailto:#">brunnoromerotenorio@gmail.com</a>
14     </address>
15 </div>
16
17 <script src="/javascripts/home/contact.js"></script>
18
19 <% include ../common/footer %>

```

Figura 25 – Header Include. Fonte: Elaborado pelo autor

Na linha 1 foi importado o nosso cabeçalho via include, comando disponibilizado pela template engine, gerando assim todo o cabeçalho da aplicação.

No cabeçalho há o carregamento dos arquivos *CSS* e *Jquery*, tratamento de alertas enviados pelo servidor e a inicialização de nosso *layout* conforme exibido nas Figuras 26 e 27.

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width, user-scalable=no">
6      <title>Passage Online</title>
7      <link href="/stylesheets/bootstrap.css" rel="stylesheet">
8      <link href="/stylesheets/common/index.css" rel="stylesheet">
9
10     <script src="/javascripts/common/jquery-1.7.1.min.js"></script>
11     <script src="/javascripts/common/bootstrap.min.js"></script>
12     <script src="/javascripts/common/Socket.io.min.js"></script>
13     <script src="/javascripts/common/Global.js"></script>
14
15 </head>
16
17 <body>
18     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
19         <div class="container">
20             <div class="navbar-header">
21                 <button type="button" class="navbar-toggle" data-target=".navbar-collapse">
22                     <span class="sr-only">Toggle navigation</span>
23                     <span class="icon-bar"></span>
24                     <span class="icon-bar"></span>
25                     <span class="icon-bar"></span>
26                 </button>
27                 <a class="navbar-brand" href="/">Ticket Online</a>
28             </div>

```

Figura 26 – Header 1. Fonte: Elaborado pelo autor

```

76 <div class="body">
77   <section class="content-wrapper main-content clear-fix">
78
79     <div id="notify" class="alert alert-warning fade in" role="alert">
80       <button type="button" class="close" data-dismiss="alert">
81         <span aria-hidden="true">x</span><span class="sr-only">Close</span>
82       </button>
83       <strong>Atenção:</strong>
84     </div>
85
86     <% if(typeof error != 'undefined') { %>
87     <div class="alert alert-danger fade in" role="alert">
88       <button type="button" class="close" data-dismiss="alert">
89         <span aria-hidden="true">x</span><span class="sr-only">Close</span>
90       </button>
91       <strong>Erro:</strong> <%- error %>
92     </div>
93     <% } %>
94
95     <% if(typeof info != 'undefined') { %>
96     <div class="alert alert-info fade in" role="alert">
97       <button type="button" class="close" data-dismiss="alert">
98         <span aria-hidden="true">x</span><span class="sr-only">Close</span>
99       </button>
100      <strong>Informação:</strong> <%- info %>
101    </div>
102    <% } %>
103
104    <% if(typeof success != 'undefined') { %>
105    <div class="alert alert-success fade in" role="alert">
106      <button type="button" class="close" data-dismiss="alert">
107        <span aria-hidden="true">x</span><span class="sr-only">Close</span></button>
108      <strong>Opa:</strong> <%- success %>
109    </div>
110    <% } %>

```

Figura 27 – Header 2. Fonte: Elaborado pelo autor

Da linha 83 em diante funciona o controlador de mensagens enviadas pelo servidor, para que seja exibida em um *box* destinado a mensagens de erro, o envio é via variável *param* com o objeto do tipo de mensagem, conforme foi especificado.

4 RESULTADOS

Neste capítulo serão discutidos, por meio de uma explicação teórico-prática, os resultados obtidos durante o desenvolvimento do projeto.

O objetivo principal consiste em desenvolver uma aplicação web para venda de passagens rodoviária, com um módulo de mineração de dados, para previsão de demanda de veículos adicionais.

Os resultados, após o desenvolvimento da aplicação, foram muitos, e variam da simplicidade para a aquisição das passagens, sem a necessidade de deslocamento do usuário aos relatórios de apoio à tomada de decisão.

Os resultados serão apresentados e discutidos abaixo, por meio dos objetivos específicos.

4.1 Pesquisa acerca das dificuldades e problemas das aplicações de venda de passagem rodoviária existente na internet.

Visando às dificuldades e complexidade dos portais de venda de passagem online, o presente trabalho propôs o desenvolvimento de um aplicativo web descomplicado e intuitivo, fornecendo ao usuário uma experiência rica que lhe possibilite a aquisição de passagem com poucos cliques.

Para o desenvolvimento do portal foram pesquisados vários sites e aplicativos através dos quais era fornecido o serviço de venda de passagens, efetuando assim o mapeamento das necessidades e possibilidades de melhorias nos serviços, hoje existentes na internet.

Logo no início, foi notada uma inflexibilidade dos sites referentes à responsabilidade, demonstrando dificuldades para os usuários de dispositivos móveis como tablets e smartphones.

Segundo Rocha (2013), ao fornecer ao usuário uma experiência rica na utilização das aplicações web, os retornos são muitos e variam de uma permanência maior do usuário junto à aplicação a recomendações pela facilidade e simplicidade de uso.

Para fornecer uma melhor experiência ao usuário foi utilizado o *Bootstrap* abordado na seção 2.8, visando a seus componentes personalizados e *templates* responsivos

que suportam praticamente todos os dispositivos disponíveis, tornando assim nosso portal disponível sem a necessidade de zoom ou qualquer outro tipo de tratamento do dispositivo.

Outra dificuldade mapeada foi a complexidade e burocracia para efetuar a aquisição da passagem, sendo necessário o preenchimento de extensos formulários com informações redundantes e confusas. Para a resolução do problema, foram filtradas as informações que são realmente necessárias para a aquisição da passagem e assim foram gerados formulários reduzidos e completos.

Como resultado, segue abaixo uma série de imagens mostrando a simplicidade para a aquisição de uma passagem utilizando o portal.

1. Preencha os dados de quem vai viajar

Origem:	Pouso Alegre	Saída:	03/10/2014	Chegada:	03/10/2014
Destino:	Santa Rita do Sapucaí	Horário:	09:50	Horário:	10:25

Preencha com os dados dos passageiros:

Poltrona:	Passageiro:	Documento:
08	Nome completo	Número de documento com foto (RG, CNH)

2. Escolha a forma de pagamento

Cartão de Débito

Itaú, Bradesco, Nubank, Santander

Cartão de Crédito

VISA, MasterCard, DISCOVER, elo, American Express

Figura 28 – Formulário de venda. Fonte: Elaborado pelo autor

Segundo Moura (2014), reduzir o número de campos substituindo por informações arrastáveis, com funcionamento iterativo, causa um impacto menor ao usuário ao utilizar a aplicação, quebrando assim o tabu da aquisição de produtos e serviços disponíveis na internet.

4.2 Utilização das tecnologias Node.js e Socket.io para desenvolver o portal de venda de passagem

O *backend* da aplicação foi todo desenvolvido usando o *Node.js* e *Socket.io*, a aplicação foi organizada no modelo MVC, conforme exibido na seção 3.6. Conseguindo assim organizar a aplicação e desacoplar os módulos.

Para essa organização foi utilizado o *express-load* citado no quadro teórico, na seção 2.3 fazendo o carregamento dos arquivos contidos dentro das respectivas pastas, conforme apresentado na seção 3.3 do quadro metodológico.

Como resultado foi gerado uma *workspace* organizada e de fácil manutenção, conforme mostrado na imagem abaixo.

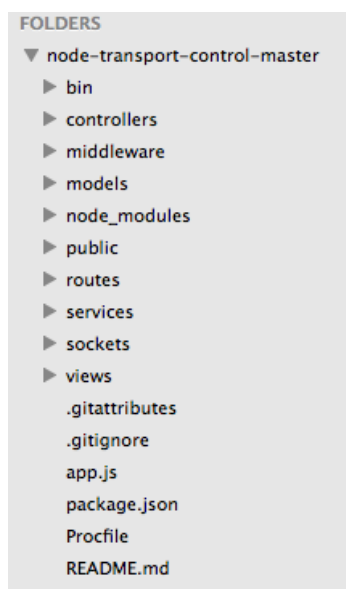


Figura 29 – Árvore Completa Resultado. Fonte: Elaborado pelo autor

O *Socket.io* foi utilizado em toda aplicação recebendo requisições e atualizando, com seu *broadcast*, todos os usuários conectados na aplicação, gerando como resultado uma aplicação em tempo real que fornece aos usuários uma melhor experiência de navegação.

4.3 Desenvolvimento de um módulo de apoio a tomada de decisão.

O módulo de apoio à tomada de decisão foi desenvolvido fornecendo ao gerente relatórios de venda e gráficos de demanda de veículo. Inicialmente foram mapeadas todas

as informações contidas na base de dados e apontada como resultado a opção de gerar relatórios de vendas por período.

Foi criada também a possibilidade de geração de um relatório de demanda de veículos na qual foram utilizadas técnicas de *Data Mining*.

Segue na Figura 30 a imagem do relatório de previsão de demanda onde foram mapeadas as demandas do ano de 2013, gerando, assim, as demandas de 2014; o relatório foi pintado, nas cores azul claro a azul escuro representando, da necessidade baixa à alta.

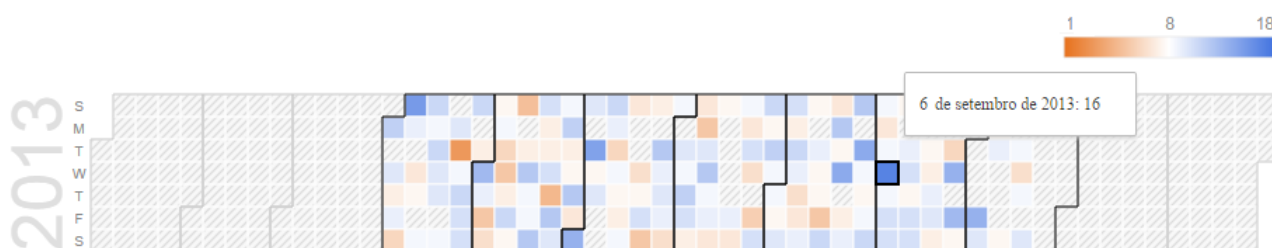


Figura 30 – Gráfico com resultado Data Mining. **Fonte:** Elaborado pelo autor

Com os relatórios em mão, os gerentes e administradores da empresa podem ter um maior controle de seus veículos e de suas vendas, tendo, assim, mais efetividade em suas tomadas de decisões.

A aplicação disponibiliza também um controle de transações não efetuadas, proporcionando o contato com o cliente para futuros *feedback* referentes aos motivos da não concretização das compras no portal e também o recurso e envio de e-mail comunicando se a compra foi ou não concretizada com sucesso.

5 CONCLUSÃO

O comércio eletrônico tem crescido, cada vez mais no Brasil, em consequência da significativa expansão do acesso à internet, percebendo-se como uma das principais causas o aumento das vendas de tablets e smartphones. As expectativas para o setor são positivas, pois o consumidor brasileiro está mais confiante para comprar utilizando a internet.

O comércio de serviços tem sido um dos mais crescentes. Visando a esse mercado, foi feito um estudo, em cuja conclusão percebeu-se que o transporte rodoviário ainda é o mais carente, tendo em vista serviços como compra de passagens aéreas, pacotes turísticos, dentre outros. Face a isso, foi criado um portal onde é possível efetuar a aquisição de passagens, via computadores e dispositivos móveis, gerando mais comodidade para os clientes.

Pensando também no controle das empresas, foi efetuado um estudo mapeando as suas maiores necessidades, deste estudo pode-se concluir que o controle de venda e mapeamento de demanda são os fatores mais críticos. Com o objetivo de se resolver isso foi proposto o desenvolvimento de um módulo de apoio à tomada de decisão, fornecendo e possibilitando relatórios e gráficos de venda por cartão e um relatório de previsão de demanda, onde se utilizaram técnicas de mineração de dados para o levantamento das informações.

Para a realização do presente trabalho foram utilizadas diversas ferramentas novas, porém poderosas para a época, como Bootstrap, Google Charts e principalmente NodeJS que contribuem para o cunho acadêmico e científico deste trabalho. Além de oferecer um rico e preciso conhecimento sobre as ferramentas e tecnologias, funcionando corretamente em diversos ambientes. Vale acrescentar que, apesar do breve tempo de desenvolvimento, muito esforço e empenho foram dedicados para a conclusão deste trabalho.

Além do tempo, fator limitador de qualquer tarefa, foram experimentados outros problemas durante a realização deste trabalho, como divergências entre browsers diferentes, domínio das ferramentas utilizadas, abstração da orientação a eventos e problemas para conciliar o desenvolvimento com a documentação.

A utilização de um banco de dados orientado a documento não foi uma boa escolha para a aplicação, pois a mesma tem uma grande necessidade de integridade dos dados,

ficando como sugestão para futuros trabalhos sua implementação utilizando uma base de dados relacional, que traga mais segurança e integridade para os dados persistidos na aplicação.

Mesmo com todas estas dificuldades, concluiu-se todos os objetivos propostos, sempre buscando a melhor forma de se resolverem os problemas encontrados, agregando diversos conhecimentos, não só técnicos como também de organização, do tempo e cumprimento do cronograma para desenvolvimento do trabalho.

Finalmente, espera-se que o trabalho deixe legado, para a comunidade acadêmica, ficando como incentivo para novos desenvolvedores.

REFERÊNCIAS

- FETTE, W. Aplicações real-time. *www.w3c.com*, p. s.p, 2013.
- KIRKPATRICK, C. *Web Real-time asynchronous*. Dissertação (Mestrado) — MIT, 2012.
- KUNG, F. Threads não bloqueantes. *http://blog.caelum.com.br/servidores-web-e-nio/*, p. s.p, 2010.
- MICROSOFT. Entity framework. *www.msdn.com*, p. s.p, 2012.
- PEREIRA, C. R. Nodejs a nova arma da google. *http://blog.caelum.com.br/nodejs-google-code/*, p. s.p, 2012.
- PEREIRA, C. R. Aplicações web real-time com node.js. p. 77, 2013.
- PEREIRA, C. R. Aplicações web real-time com node.js. p. 77, 2013.
- PEREIRA, C. R. Aplicações web real-time com node.js. p. 77, 2013.
- SILVESTRE, G. A web do futuro. *www.w3c.com*, p. s.p, 2011.

biblio.bib)