**NÍCOLAS HENRIQUE VIEIRA TOLEDO**

**SEBASTIÃO BATISTA DE ANDRADE NETO**

**ESTATÍSTICAS DE REDES SOCIAIS**

**COM BIG DATA**

**UNIVERSIDADE DO VALE DO SAPUCAI**

**POUSO ALEGRE**

**2015**

**NÍCOLAS HENRIQUE VIEIRA TOLEDO**

**SEBASTIÃO BATISTA DE ANDRADE NETO**

**ESTATÍSTICAS DE REDES SOCIAIS**

**COM BIG DATA**

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

Orientador: Prof. Ednardo David Segura

**UNIVERSIDADE DO VALE DO SAPUCAI**

**POUSO ALEGRE**

**2015**

**NÍCOLAS HENRIQUE VIEIRA TOLEDO**

**SEBASTIÃO BATISTA DE ANDRADE NETO**

**ESTATÍSTICAS DE REDES SOCIAIS**

**COM BIG DATA**

Trabalho de conclusão de curso defendido e aprovado em \_\_/\_\_/\_\_\_\_ pela banca examinadora constituída pelos professores:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor EDNARDO DAVID SEGURA

Orientador

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor ARTUR LUIS RIBAS BARBOSA

Examinador

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor ROBERTO RIBEIRO ROCHA

Examinador

Dedicado às nossas famílias, amigos e professores que sempre nos incentivaram e apoiaram ao longo de nossas vidas.

**AGRADECIMENTOS**

Agradecemos primeiramente a Deus, pela força e determinação que nos foi concedida nos momentos mais difíceis de nossa jornada.

Ao Prof. Ednardo David Segura, orientador desta pesquisa, pelo auxílio e tempo dedicados a nossa orientação ao esclarecimento de nossas dúvidas.

Ao Inatel, por ter cedido o espaço para a realização das nossas reuniões com o orientador da pesquisa.

Ao Lucas Vilela, pela sua disponibilidade e seu grande auxílio no entendimento do tema abordado. À Profª. Dra. Joelma Pereira de Faria e ao Prof. Ms. José Luiz da Silva, pela atenção em corrigir nossos textos e nos orientar para que o trabalho atendesse as normas técnicas e ortográficas.

De Nícolas Henrique Vieira Toledo:

Agradeço a todos os meus familiares que me apoiaram em minha jornada, especialmente a meu pai, Donizette Felício de Toledo, que acreditou em meus sonhos e me ajudou a torná-los realidade. Pai, o senhor é a minha inspiração de vida, obrigado por tudo. Aos meus amigos e a minha namorada Aline Gomes Pereira Pinto, pela compreensão nos momentos de ausência durante o desenvolvimento desta pesquisa.

De Sebastião Batista de Andrade Neto:

Agradeço aos meus pais, que acreditaram em mim quando nem eu mesmo já acreditava. Eles foram a minha força para seguir em frente. Aos meus amigos, pela compreensão nos momentos de ausência no decorrer do curso.

*“Na adversidade, uns desistem, enquanto outros batem recordes”.*

*(Ayrton Senna)*

NETO, Sebastião Batista de Andrade; TOLEDO, Nícolas Henrique Vieira. **Estatísticas de Redes Sociais com Big Data**. Monografia - Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, 2015.

**RESUMO**

Atualmente, um grande volume de dados é gerado diariamente pela vasta quantidade de usuários constantemente conectados à internet. Estes dados são criados de diversas formas, e para finalidades distintas. Porém estas informações, que muitas vezes são ignoradas, podem trazer diversas vantagens competitivas. A partir desse pressuposto, parte-se para o conceito-chave desta pesquisa, o *Big Data*. Este conceito pode ser descrito como um conjunto de tecnologias e estratégias adotadas para o tratamento de grandes quantidades de dados que auxiliam a tomada de decisão. Desse modo, esta pesquisa apresenta o desenvolvimento de um sistema capaz de realizar a mineração, armazenamento, tratamento e demonstração em tempo real do grande volume de dados políticos gerados pela rede social *Twitter*. O tema político foi abordado nesta pesquisa, pois, além de ser um dos assuntos mais comentados do *Twitter* na atualidade, ele é um tema que abrange muitas das carências da população brasileira, que é o público-alvo principal desta pesquisa. Para o desenvolvimento do sistema, foram utilizadas tecnologias *Apache Storm*, MongoDB, Node.JS e o *framework Express*, visando a obtenção dos resultados que envolvem conceito de Big Data. Também foi feito o uso de tecnologias de comunicação com o *Twitter*, que são: O *Twitter Stream* API e a biblioteca *Twitter4j*. Para a demonstração das informações políticas em tempo real, foram usados recursos de *live stream*, mapas e gráficos. Ao final desta pesquisa, foi notado que todas as tecnologias usadas obtiveram o resultado esperado, manipulando um grande volume de dados em tempo hábil para a apreciação do usuário.

**Palavras-chave:** Big Data. Mineração de Dados. Apache Storm. Twitter4j. Tempo Real.

NETO, Sebastião Batista de Andrade; TOLEDO, Nícolas Henrique Vieira. **Estatísticas de Redes Sociais com Big Data**. Monografia - Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, 2015.

**ABSTRACT**

*Nowadays, the large number of users constantly connected to the internet daily generates a huge volume of data. These data are create in many ways, and for different purposes. However, this information, often ignored, could bring several competitive advantages. Based on this premise, is initiated the concept-key of this research, the Big Data. This concept can be described as a set of technologies and strategies used to deal with large amounts of data, that assists in decision-making process. In that way, this research presents the development of a system capable of perform mining, storage, processing and real-time demonstration of the huge volume of political data generated by Twitter. The political theme was approached in this research because, besides being one of the most commented subjects on Twitter nowadays, it is a topic that covers many of the needs from the Brazilian population, which is the main target audience of this research. To build the system, the technologies that they were used are Apache Storm, MongoDB, Node.JS and Express framework, aimed at obtaining results that involve the Big Data concept. It was also need the use of communication technologies with Twitter, which are the Twitter Stream API and the Twitter4j library. To show the political information in real time, was used live stream resources, maps and charts. At the end of this research, it was noted that all used technologies have met the expected result, handling a large volume of data in time for user's enjoyment.*

**Keywords:** Big Data. Data Mining. Apache Storm. Twitter4j. Real-Time.

**LISTA DE FIGURAS**

[Figura 1 - Exibição dos *tweets* em tempo real e por data 19](#_Toc435998452)

[Figura 2 - Exibição da localidade dos *tweets* 20](#_Toc435998453)

[Figura 3 - Exibição de estatísticas politicas, por meio de um gráfico 20](#_Toc435998454)

[Figura 4 - Os 3 v’s que definem o *Big Data* 26](#_Toc435998455)

[Figura 5 - *Data Mining* como um passo no processo da descoberta do conhecimento 29](#_Toc435998456)

[Figura 6 - Arquitetura não-bloqueante 37](#_Toc435998457)

[Figura 7 - Arquitetura bloqueante 37](#_Toc435998458)

[Figura 8 - Arquivo package.json 50](#_Toc435998459)

[Figura 9 - Árvore de diretórios da aplicação web 51](#_Toc435998460)

[Figura 10 - Arquitetura do sistema 57](#_Toc435998461)

[Figura 11 - Modelagem do *Back-end* 59](#_Toc435998462)

[Figura 12 - Árvore de diretórios das *Views* 76](#_Toc435998463)

[Figura 13 - Página inicial 78](#_Toc435998464)

[Figura 14 - *Partial View* de cadastro de usuário 81](#_Toc435998465)

[Figura 15 - Classes do *framework* Apache Storm 89](#_Toc435998466)

[Figura 16 - Diagrama de classes do *framework* Apache Storm 90](file:///C:\Users\Nícolas%20Vieira\Desktop\TCC\documents\life_cycle\Capa%20Dura\pesquisa_capa_dura_sem_correcoes_ortograficas.docx#_Toc435998467)

[Figura 17 - *Tweets* no MongoDB, no formato de documentos 107](#_Toc435998468)

[Figura 18 - Seção de manipulação das palavras-chave 114](#_Toc435998469)

[Figura 19 - Quantidade de tweets minerados até o dia 08/11/2015 116](#_Toc435998470)

[Figura 20 - *Tweets* da palavra-chave “Dilma” 118](#_Toc435998471)

[Figura 21 - *Tweets* da palavra-chave “Lula” 118](#_Toc435998472)

[Figura 22 - Aba da palavra-chave “Aécio Neves”, sem nenhum *tweet* recebido 119](#_Toc435998473)

[Figura 23 - Mapa da aplicação, em seu estado inicial 120](#_Toc435998474)

[Figura 24 - Mapa da aplicação, após 10 segundos de uso 120](#_Toc435998475)

[Figura 25 - Mapa da aplicação, após 20 segundos de uso 121](#_Toc435998476)

[Figura 26 - Botão de limpeza dos marcadores 121](#_Toc435998477)

[Figura 27 - Gráfico de menções no Twitter por palavra-chave 123](#_Toc435998478)

**LISTA DE CÓDIGOS**

[Código 1 - Adição do repositório CloarJS.org no arquivo pom.xml 52](#_Toc435998768)

[Código 2 - Dependências adicionadas no arquivo pom.xml 53](#_Toc435998769)

[Código 3 - Chamada dos módulos na *stack* 60](#_Toc435998770)

[Código 4 - Conexão com o Banco de dados 62](#_Toc435998771)

[Código 5 - Configuração de diretórios 62](#_Toc435998772)

[Código 6 - Configurações da variável app 63](#_Toc435998773)

[Código 7 - Configurações estáticas e de tratamento de erro da aplicação 63](#_Toc435998774)

[Código 8 - *Load* de diretórios da aplicação 64](#_Toc435998775)

[Código 9 - Inicialização do servidor na porta 3000 65](#_Toc435998776)

[Código 10 - Configuração da conexão *socket* 66](#_Toc435998777)

[Código 11 - Autenticação de usuário via *middleware* 68](#_Toc435998778)

[Código 12 - Tratamento de erros via *middleware* 68](#_Toc435998779)

[Código 13 - Exemplo de um arquivo de rotas 69](#_Toc435998780)

[Código 14 - Trecho de código exemplificando um *Controller* 71](#_Toc435998781)

[Código 15 - Trecho de código exemplificando um *Service* e sua consulta no banco 73](#_Toc435998782)

[Código 16 - Trecho de código exemplificando um *Schema* de um *Model* 75](#_Toc435998783)

[Código 17 - Código parcial da *View* da página inicial 77](#_Toc435998784)

[Código 18 - Código parcial exemplificando um *Header* 79](#_Toc435998785)

[Código 19 - Exemplo de uma *Partial View* 80](#_Toc435998786)

[Código 20 - Import do JavaScript do Google Maps API 81](#_Toc435998787)

[Código 21 - JavaScript controlador do mapa da aplicação (Parte 1) 82](#_Toc435998788)

[Código 22 - JavaScript controlador do mapa da aplicação (Parte 2) 83](#_Toc435998789)

[Código 23 - JavaScript controlador do mapa da aplicação (Parte 3) 83](#_Toc435998790)

[Código 24 - JavaScript controlador do mapa da aplicação (Parte 4) 84](#_Toc435998791)

[Código 25 - JavaScript controlador do mapa da aplicação (Parte 5) 84](#_Toc435998792)

[Código 26 - Funções de manipulação do gráfico (Parte 1) 86](#_Toc435998793)

[Código 27 - Funções de manipulação do gráfico (Parte 2) 87](#_Toc435998794)

[Código 28 - Funções de manipulação do gráfico (Parte 3) 87](#_Toc435998795)

[Código 29 - Funções de manipulação do gráfico (Parte 4) 88](#_Toc435998796)

[Código 30 - Criação da conexão com o MongoDB 91](#_Toc435998797)

[Código 31- Busca de todas as palavras-chave cadastradas no MongoDB. 92](#_Toc435998798)

[Código 32 - Método responsável por para salvar um documento no banco de dados 93](#_Toc435998799)

[Código 33 - Método execute da classe PolitistatusTwitterBolt 94](#_Toc435998800)

[Código 34 - Métodos declareOutputFields, getKeywords e setKeywords 95](#_Toc435998801)

[Código 35 - Construtor da classe PolitistatusTwitterSpout 95](#_Toc435998802)

[Código 36 - Implementações da biblioteca Twitter4j na classe PolitistatusTwitterSpout 96](#_Toc435998803)

[Código 37 - Criação do Twitter *Stream* 98](#_Toc435998804)

[Código 38 - Método de leitura das tuplas 99](#_Toc435998805)

[Código 39 - Método de encerramento de conexão com o Twitter Stream API 99](#_Toc435998806)

[Código 40 - Método configuração de tarefas 100](#_Toc435998807)

[Código 41 - Operações de banco de dados na classe de execução 101](#_Toc435998808)

[Código 42 - Método openDbConnection 101](#_Toc435998809)

[Código 43 - Método getAllUserKeywords 102](#_Toc435998810)

[Código 44 - Lógica da classe de execução 102](#_Toc435998811)

[Código 45 - Método de construção da topologia do Storm 103](#_Toc435998812)

[Código 46 - Métodos de reconstrução da topologia do Storm 105](#_Toc435998813)

[Código 47 - Método da busca atualizada e envio dos tweets 109](#_Toc435998814)

**LISTA DE ABREVIATURAS E SILGAS**

ASCII - *American Standard Code for Information Interchange*

AJAX - Asynchronous JavaScript and XML

API - Application Programming Interface

CERN - *European Organization for Nuclear Research*

CRUD - *Create, Read, Update and Delete*.

CSS - *Cascading Style Sheets*

EJS - *Embedded* *JavaScript*

EUA - Estados Unidos da América

HP - Hewlett-Packard

HTML - *Hyper Text Markup Language*

HTTP - *Hyper Text Transfer Protocol*

IaaS - *Infraestructure as a Service*

IBM - *International Business Machines*

ICC - Inatel *Competence Center*

IDE - *Integrated Development Environment*

Inatel - Instituto Nacional de Telecomunicações

I/O - *Input/Output*

JAR - Java *Archive*

JS - *JavaScript*

JSON - *JavaScript Object Notation*

JVM - Java *Virtual Machine*

NoSQL - *Not Only* SQL

NPM - *Node Package Manager*

PaaS - *Plataform as a Service*

RAM - *Random Access Memory*

REST - *Representational State Transfer*

SaaS - *Software as a Service*

SQL - Structured Query Language

TI - Tecnologia da Informação

URI - *Uniform Resource Indentifier*

URL - *Uniform Resource Locator*

W3C - *World Wide Web Consortium*

WWW - *World Wide Web*

XHTML - *Extensible Hyper Text Markup Language*

XML - *Extensible Markup Linguage*

**SUMÁRIO**

[1 INTRODUÇÃO 16](#_Toc436338162)

[2 QUADRO TEÓRICO 22](#_Toc436338163)

[*2.1* *Big Data* 22](#_Toc436338164)

[2.2 Apache Storm 27](#_Toc436338165)

[2.2.1 *Data Mining* 28](#_Toc436338166)

[2.2.2 *Data Mining* em Redes Sociais 30](#_Toc436338167)

[2.2.3 Apache Maven 30](#_Toc436338168)

[*2.3* REST API 31](#_Toc436338169)

[2.3.1 Twitter *Stream* API 32](#_Toc436338170)

[2.3.2 Twitter4j 33](#_Toc436338171)

[2.4 Banco de dados NoSQL 33](#_Toc436338172)

[2.4.1 MongoDB 35](#_Toc436338173)

[2.5 Node.JS 36](#_Toc436338174)

[2.5.1 Express 38](#_Toc436338175)

[*2.6* *Cloud Computing* 39](#_Toc436338176)

[2.6.1 Amazon *Web Services* 40](#_Toc436338177)

[*2.7* *Open Web Plataform* 41](#_Toc436338178)

[2.7.1 HTML5 41](#_Toc436338179)

[2.7.2 CSS3 42](#_Toc436338180)

[2.7.3 JavaScript 43](#_Toc436338181)

[2.8 Bootstrap 43](#_Toc436338182)

[2.9 Google Maps API 44](#_Toc436338183)

[2.10 Highcharts 44](#_Toc436338184)

[3 QUADRO METODOLÓGICO 46](#_Toc436338185)

[3.1 Tipo de Pesquisa 46](#_Toc436338186)

[3.2 Contexto de pesquisa 47](#_Toc436338187)

[3.3 Instrumentos 47](#_Toc436338188)

[3.4 Procedimentos e Resultados 48](#_Toc436338189)

[3.4.1 Configuração do ambiente 49](#_Toc436338190)

[3.4.2 Desenvolvimento dos protótipos 54](#_Toc436338191)

[3.4.3 Desenvolvimento da aplicação 56](#_Toc436338192)

[3.4.4 *Back-end* 58](#_Toc436338193)

[3.4.5 *Front-end* 76](#_Toc436338194)

[3.4.6 *Framework* de *Data Mining* e *Big Data* 89](#_Toc436338195)

[3.4.7 Hospedagem na *Amazon Web Services* 106](#_Toc436338196)

[3.4.8 Integração das tecnologias - Mineração dos dados, geração do *Big Data* e demonstração dos dados políticos 106](#_Toc436338197)

[4 DISCUSSÃO DE RESULTADOS 112](#_Toc436338198)

[5 CONCLUSÃO 124](#_Toc436338199)

[REFERÊNCIAS 126](#_Toc436338200)

[ANEXO 129](#_Toc436338201)

[APÊNDICES 131](#_Toc436338202)

# INTRODUÇÃO

Atualmente, um grande volume de dados é visto circulando por todo o globo diariamente. Dia após dia, mais dados são criados por usuários pelas mais diversas razões. Eles podem ser desde uma simples foto de viagem postada em uma rede social, até uma página de protesto governamental da web. As possibilidades são gigantes. O problema, é que a maioria desses dados não está armazenada em uma base de dados estruturada. Desse modo, eles são considerados dados não estruturados (ANDRIOLE, 2015).

Os dados não estruturados são provenientes de documentos nos mais diversos formatos, tais como textos, imagens, vídeos, páginas web, dentre outros formatos conhecidos de arquivo (ANDRIOLE, 2015). Porém, há um questionamento crucial a ser feito: como armazenar de maneira confiável, escalável e segura uma quantidade de dados tão variada e que cresce de maneira tão rápida?

Com o surgimento deste problema, nasceu o conceito de *Big Data*. O *Big Data*, de maneira geral, consiste em um armazenamento de massas de dados não estruturados, de modo que estes sejam coletados, organizados, processados e apresentados de maneira mais rápida e segura possível (ALECRIM, 2015).

LaVelle et al (2010), afirmam que a tendência do *Big Data* está crescendo entre as organizações. As informações estratégicas são coletadas através de canais digitais não estruturados, como: redes sociais, aplicativos de *smartphones*, e tantos outros dispositivos emergentes baseados na internet. Devido à sua grande quantidade, estes dados devem ser armazenados em uma base de dados consistente e confiável.

O *Big Data* apresenta soluções realmente inovadoras, substituindo os tradicionais métodos de coleta e armazenamento de dados de todas as organizações que desejam obter um diferencial competitivo no mercado. Contudo, é necessário também mudar a cultura da organização para que o *Big Data* apresente o resultado esperado.

McAfee e Brynjolfsson (2012) concluíram que as ferramentas e filosofias relacionadas ao *Big Data* se disseminaram, mudando assim as ideias em longo prazo de gestores sobre a experiência, natureza, e a prática de gestão. Desse modo, líderes capacitados de todos os setores do mercado estarão usando o *Big Data* com a devida finalidade para qual ele foi concebido: uma revolução na gestão. Como em qualquer outra grande mudança de negócio, os desafios de tornar o *Big Data* operante em uma organização podem ser relativamente complexos em alguns casos, no entanto, é uma transição que é indispensável para os executivos nos dias atuais.

No entanto, para que se aplique o conceito de *Big Data*, é necessário existir uma grande quantidade de dados. Para que isso ocorra, é necessário realizar uma mineração de dados. Porém, antes de fazer a mineração, é necessário decidir a fonte e quais os tipos de dados que serão minerados. Essa fonte de dados, obrigatoriamente deverá ter um grande fluxo de dados diários. Um exemplo de fonte de dados confiável é a rede social Twitter, que, por meio de suas API’s de *stream*, possibilitam o acesso ao seu fluxo global de *tweets*.

Devido ao grande número de usuários, conectados através dos mais diversos dispositivos, redes sociais como o Facebook e o Twitter geram quantidades enormes de dados por dia. Esses dados são de extrema importância, se forem analisados visando um diferencial competitivo. Desse modo, a coleta e análise dessa massa de dados se tornam indispensáveis (FRANÇA et al, 2014).

Com a fonte de dados definida, é necessário decidir que tipo de dados serão minerados. De nada adianta o acesso aos dados do Twitter se o tema escolhido para a mineração de dados não gerar grandes quantidades de *tweets* por dia. Tomando por base o atual cenário brasileiro, a política é um exemplo de tema que está gerando grandes quantidades de *tweets* por dia.

Nagarajan, Sheth e Velmurugan (2011), afirmam que o compartilhamento de opiniões entre população tornou-se uma atividade comum nos dias atuais. Isso se deve ao crescimento rápido na popularidade das redes sociais, bem como o de aparelhos constantemente conectados à internet. Isto deu um acesso sem precedentes aos os dados de uma população, bem como capacidade de realizar análises dos dados sociais e o desenvolvimento aplicações socialmente inteligentes, seja no sentido de entrega de conteúdo direcionado, gestão de crises, organização de revoluções ou a promoção do desenvolvimento social nos países em desenvolvimento.

Analisando em um cenário nacional, mais precisamente no âmbito político, os presidenciáveis Dilma Rousseff e Aécio Neves, juntos, tiveram cerca de 1.585.369 de menções no Facebook e 2.715.438 de *tweets* nas eleições do ano de 2014 (SCUP IDEAS, 2015).

Geralmente, a população tende a expressar suas opiniões políticas com maior intensidade quando ocorrem eventos de grandes proporções nacionais, como uma eleição presidencial. A população também se expressa com certa intensidade no ano da posse oficial, onde as expectativas relacionadas ao candidato são criadas. Como as redes sociais são um meio de expressão de fácil acesso, quantidades enormes de opiniões, sugestões, críticas, e análises políticas são feitas diariamente (SILVA; FERREIRA JUNIOR, 2013).

Diante desta situação, esta pesquisa adotou a rede social Twitter como fonte de dados, e a política como tema base para a mineração. Esta decisão foi tomada devido a grande quantidade de *tweets* de embasamento político criados diariamente pelos usuários do Twitter.

Desse modo, devido ao grande interesse gerado pelas tecnologias e conceitos que envolvem o *Big Data*, a presente pesquisa tem como objetivo geral o desenvolvimento de um sistema capaz de armazenar e demonstrar em tempo real as opiniões dos usuários do Twitter, com foco naquelas que são relacionadas à política brasileira.

Para alcançar o objetivo geral desta pesquisa, foram colocados os seguintes objetivos específicos:

* Coletar, da rede social Twitter, dados relacionados à política brasileira com base em palavras-chave cadastradas pelos usuários de uma aplicação web;
* Processar os dados coletados através do Apache Storm, posteriormente salvando as informações importantes em uma base de dados;
* Apresentar na aplicação web os *tweets*, tanto em tempo real quanto os mais antigos. Apresentar também um mapa com as localidades dos *tweets* e um gráfico de quantidade de menções por palavra-chave cadastrada.

A proposta da pesquisa é desenvolver um sistema que seja capaz de demonstrar como a robustez das tecnologias relacionadas à *Big Data* podem influenciar nas mais diversas áreas do conhecimento humano. Para a geração do *Big Data*, foi definido que seria feita a mineração de dados na rede social Twitter. Porém, os dados a serem minerados devem ter um tema atual, e que seja responsável pela geração de um grande fluxo de *tweets*. Estes devem ser preferencialmente, feitos por usuários brasileiros. Para tal, foi definido que o tema de mineração de *tweets* será a política, pois, além de ser um tema de atual relevância no Brasil, ele é o responsável pela criação de uma grande quantidade diária de *tweets*, sendo capaz assim, de gerar o *Big Data* desta pesquisa.

Em outras palavras, a proposta desta pesquisa é desenvolver um sistema que seja eficiente o bastante para demonstrar para seu usuário as opiniões e estatísticas relacionadas à política brasileira em tempo real, por meio da rede social Twitter.

Para exemplificar os objetivos desta pesquisa, as Figuras 1, 2 e 3, demonstram as telas do sistema web que foi desenvolvido. Respectivamente, essas telas demonstram: a exibição dos *tweets* em tempo real e por data, um mapa de demonstração das localidades dos *tweets* e as estatísticas de menção por político, demonstrada por meio de um gráfico.

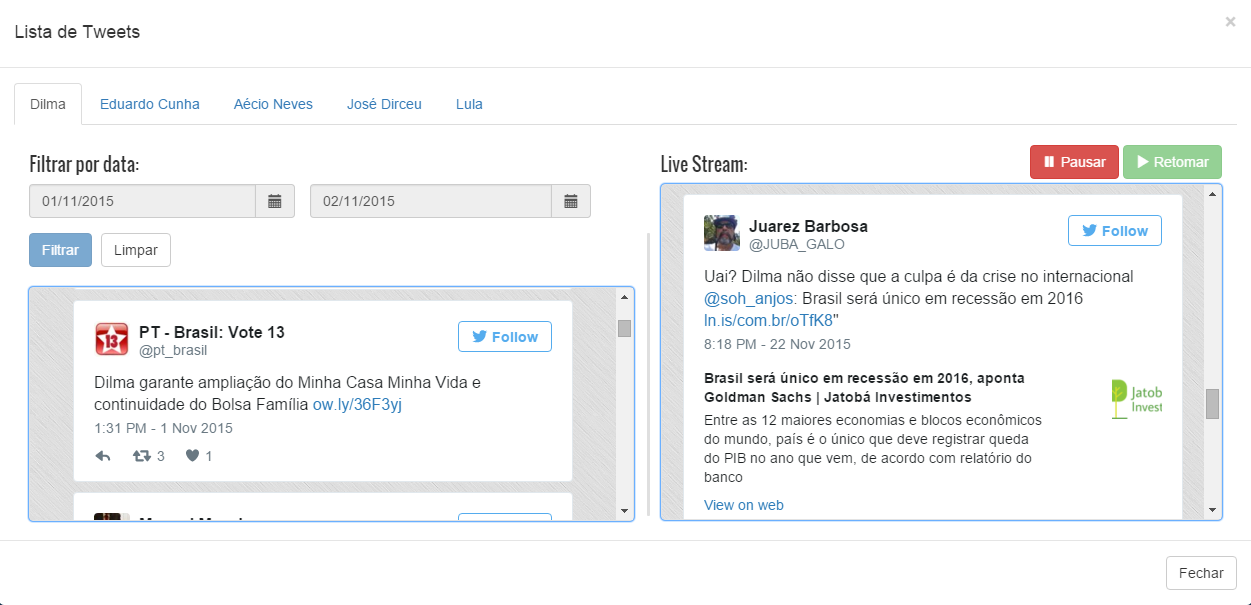


Figura 1 - Exibição dos *tweets* em tempo real e por data. Fonte: Elaborado pelos autores.



Figura 2 - Exibição da localidade dos *tweets*. Fonte: Elaborado pelos autores.



Figura 3 - Exibição de estatísticas politicas, por meio de um gráfico. Fonte: Elaborado pelos autores.

Em âmbito social, a pesquisa se torna relevante na área política, pela qual serão geradas informações para análise. Desse modo, são tomadas como base duas linhas de contribuição: a dos próprios políticos, a qual será feita a demonstração de onde a população se torna mais carente (através da análise e demonstração da localidade dos *tweets*), e da população de um modo geral, pois, através da exposição dos *tweets* pela aplicação, será feita a demonstração de como está o desempenho de seus representantes perante a sociedade.

No âmbito tecnológico, foi escolhida a rede social *Twitter* pelo fato de seus usuários gerarem grandes quantidades de dados diariamente. A análise correta destes dados pode trazer diferenciais competitivos para uma organização. O conceito de *Big Data* foi escolhido por ser um assunto de atual relevância e diferencial competitivo no mercado mundial. Combinando estes dois cenários, a contribuição será a integração entre as duas partes, servindo de “exemplo prático” para qualquer pessoa com o conhecimento tecnológico que queira fazer este tipo de integração.

Na perspectiva acadêmica, a pesquisa se torna relevante pois ainda não foi abordado no curso. Assim, a mesma pode ser arquivada na biblioteca a fim de servir de material de consulta para alunos do curso de sistemas de informação.

Esta pesquisa divide-se em cinco capítulos, nos quais são demonstradas e discutidas as ferramentas e tecnologias empregadas no desenvolvimento do sistema de estatísticas com *Big Data*. No capítulo dois, são apresentadas as teorias relacionadas às tecnologias que embasaram o desenvolvimento do sistema. No terceiro capítulo são demonstradas as metodologias de desenvolvimento executadas para que os objetivos desta pesquisa fossem alcançados. No quarto capítulo são demonstrados os resultados obtidos com o desenvolvimento desta pesquisa e as discussões acerca dos mesmos. No quinto e último capítulo, são demonstradas as conclusões obtidas pelos autores.

# QUADRO TEÓRICO

No quadro teórico são descritos o histórico e os conceitos que se relacionam com o tema, bem como as tecnologias que são aplicadas no desenvolvimento do trabalho.

## *Big Data*

A história de como os dados se tornaram tão vastos, gerando o que hoje conhecemos como *Big Data*, vai muito além do atual estado estabelecido. Há nove anos atrás já houve tentativas de quantificar a taxa de crescimento no volume de dados ou o que ficou conhecido como a “explosão de informação” (PRESS, 2013).

Aqui será descrito um resumo de como tudo aconteceu desde 1944, quando Freemont Rider, em sua obra “The Scholar and the Future of the Research Library”[[1]](#footnote-1), estimava que “As bibliotecas universitárias americanas dobravam o tamanho a cada dezesseis anos” (RIDER, 1944, apud PRESS, 2013). Ele especulava que em 2040, a Biblioteca de Yale teria cerca de 200 milhões de volumes e precisaria de cerca de seis mil pessoas trabalhando no serviço de catálogo (PRESS, 2013).

Hal B. Becker, em sua publicação “Can users really absorb data at today’s rates? Tomorrow’s?”[[2]](#footnote-2), de 1986, estimava que a densidade de caracteres alcançada por Guttenberg era de aproximadamente de 500 caracteres por polegada cúbica. Por volta do ano 2000, as memórias RAM armazenariam 1.25X10¹¹ bytes por polegada cúbica (PRESS, 2013).

Em 1997 Michael Cox e David Ellsworth publicaram “Application-controlled demand paging for out-of-core visualization”[[3]](#footnote-3), no qual eles começaram o artigo afirmando que o tamanho extenso dos conjuntos de dados penalizam as capacidades da memória principal, disco local e até discos remotos, gerando assim um desafio interessante para os sistemas computacionais. A esse desafio, eles deram o nome de “problema do Big Data”. De acordo com eles, quando conjuntos de dados não cabem mais na memória principal, ou quando os dados não cabem nem nos discos locais, a solução é adquirir mais recursos (PRESS, 2013).

A publicação de 1999 “*Visually exploring gigabyte data sets in real time”*[[4]](#footnote-4), de Steve Bryson, David Kenwright, Michael Cox, David Ellsworth e Robert Haimes, dizia que computadores poderosos são uma dádiva em diversos campos de investigação, porém em contrapartida, suas operações computacionais rápidas geram quantidades massivas de dados (PRESS, 2013).

Eles também afirmavam que tempos atrás, alguns megabytes de conjuntos de dados eram considerados grandes. Entretanto, na época já eram encontrados conjuntos de dados na casa dos 300 gigabytes. Porém se torna um desafio entender os resultados dos dados de computações de alto desempenho. É simplesmente difícil olhar para todos os números. Além do mais, o objetivo da computação é o conhecimento, e não números (PRESS, 2013).

Na publicação “*How Much Information?*”[[5]](#footnote-5), de Peter Lyman e Hal R. Varian feita em 2000, foi apontado que em 1999 o mundo produziu cerca de 1.5 exabytes de informação original, ou cerca de 250 megabytes para cada homem, mulher e criança na terra. Uma vasta quantidade de informação é criada e armazenada por indivíduos (democratização dos dados), e isso cresce velozmente. Chamando essa descoberta de “A dominância do Digital”, Lyman e Varian diziam que maioria da informação textual já “nascia digital” e em poucos anos, a sentença também se tornaria verdadeira para imagens (PRESS, 2013).

Também em 2000, Francis X. Diebold apresentou um documento intitulado “*Big Data Dynamic Factor Models for Macroeconomic Measurement and Forecasting*”[[6]](#footnote-6), onde ele afirma que “Recentemente, muita ciência boa, seja física, biológica ou social, foi forçada a confrontar, além de obter benefícios, do fenômeno designado *Big Data*” (DIEBOLD, 2000, apud PRESS, 2013).

*Big Data* refere-se como a explosão na quantidade (e às vezes na qualidade) da informação relevante disponível, em grande parte, o resultado de avanços recentes e inéditas em gravação de dados e tecnologia de armazenamento (PRESS, 2013).

Em 2001, Doug Laney, um analista no Meta Group, publicou uma nota de pesquisa intitulada “*3D Data Management: Controlling Data Volume, Velocity, and Variety.*”[[7]](#footnote-7) Uma década depois, os “3v's” se tornaram as três dimensões que definem o *Big Data*, mesmo que nas notas de Laney o termo em si não aparece (PRESS, 2013).

Em 2007 John F Gantz, David Reinsel e outros pesquisadores lançam um livro, intitulado “*The Expanding Digital Universe: A Forecast of Worldwide Information Growth through 2010*”[[8]](#footnote-8). Nele, era afirmado que em 2006, o mundo produziu 161 exabytes de dados, além de prever que entre 2006 e 2010, a informação adicionada anualmente ao universo digital aumentaria para 988 exabytes, ou dobrando a cada 18 meses. De acordo com os lançamentos de 2010 e 2012 do mesmo estudo, a quantidade de dados digitais ultrapassou esta previsão, atingindo 1.227 exabytes em 2010, e crescendo a 2.837 exabytes em 2012 (PRESS, 2013).

Em 2008 Randal E. Bryant, Randy H. Katz, e Edward D. Lazowska publicaram “*Big-Data Computing: Creating Revolutionary Breakthroughs in Commerce, Science and Society*”[[9]](#footnote-9).

Para Bryant; Katz e Lazowska (2008, p.1 e seg. apud Press 2013):

Assim como os motores de busca têm transformado a forma de acessar as informações, outras formas de *Big Data* podem e vão transformar as atividades das empresas, pesquisadores científicos, médicos e operações de defesa e inteligência da nossa nação[...] *Big Data* é, talvez, a maior inovação na computação na última década. Nós apenas começamos a ver o seu potencial para coletar, organizar e processar dados em todas as esferas da vida. Um modesto investimento por parte do governo federal poderia acelerar consideravelmente o seu desenvolvimento e implementação.

Em 2010 Kenneth Cukier informa em seu relatório “*Data, data everywhere*”[[10]](#footnote-10) que o mundo contém uma inimaginável quantidade de informação digital que está crescendo cada vez mais rapidamente, e que o efeito pode ser visto em todos os lugares. Cientistas e engenheiros de computação designaram termo para o fenômeno: “*Big Data*” (PRESS, 2013).

Na publicação de 2011 “*Big data: The next frontier for innovation, competition, and productivity*”[[11]](#footnote-11) feita por James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, e Angela Byers, foi estimado que até 2009, quase todos os setores da economia americana tinham, pelo menos, uma média de 200 terabytes de dados armazenados por empresa (o dobro do tamanho do *Data Warehouse*[[12]](#footnote-12) da varejista americana *Wal-Mart* em 1999). No total, o estudo estimou que 7,4 exabytes de novos dados foram armazenados pelas empresas e 6,8 exabytes por consumidores em 2010 (PRESS, 2013).

Na publicação de 2012 “*Critical Questions for Big Data in* Information, Communications, and Society**”**[[13]](#footnote-13), Danah Boyd e Kate Crawford definem *Big Data* como um fenômeno cultural, tecnológico e acadêmico que é formado da interação entre tecnologia (capacidade computacional e precisão algorítmica para coletar, analisar, unir e comparar dados), análise (identificação padrões, a fim de fazer reivindicações econômicas, sociais, técnicas e legais) e mitologia (crença de que grandes conjuntos de dados oferecem uma forma mais elevada de inteligência e conhecimento que podem gerar perspectivas que antes eram impossíveis, com verdade, objetividade e precisão) (PRESS, 2013).

De acordo com Soubra (2012), existem três propriedades que são capazes de definir a expansão do Big Data: Volume, Variedade e Velocidade. A junção destes é chamada de “os 3 v’s do Big Data”. Cada um deles pode ser definido da seguinte forma:

* Volume: quantidade de dados realmente grandes, que tem crescimento exponencial;
* Variedade: o grande volume de dados se deve ao fato da existência de uma grande variedade de informações. Os dados podem ser classificados em estruturados, semi-estruturados e não estruturados.
* Velocidade: o tratamento dos dados deve ser feito em tempo hábil. O volume de informações, com uma variedade de dados diversa, deve ser processado e analisado no menor tempo possível.

A Figura 4 demonstra as 3 fronteiras de crescimento do *Big Data* (3 v’s).

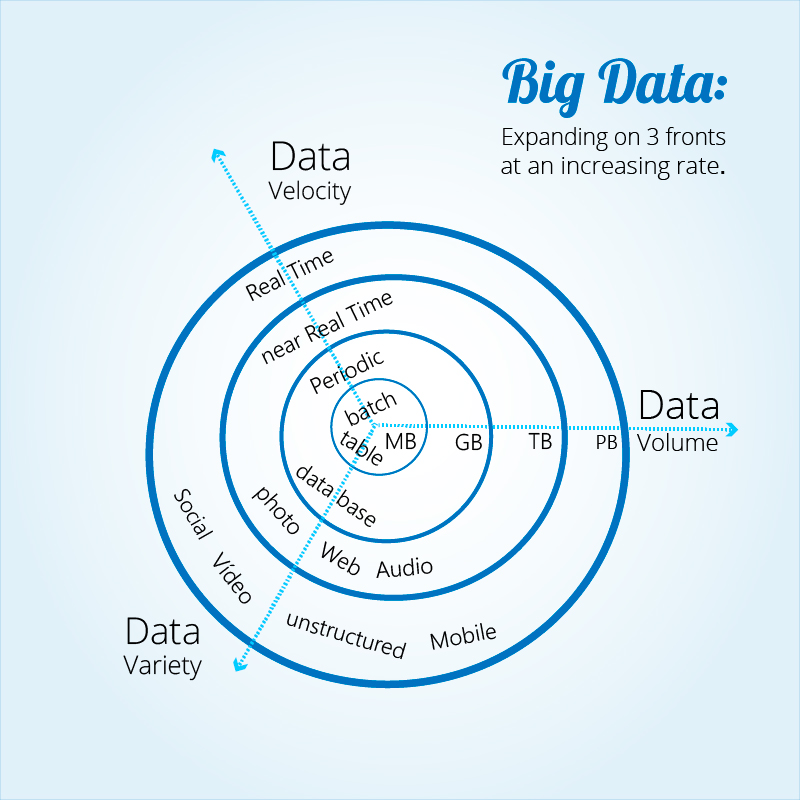
**

Figura 4 - Os 3 v’s que definem o *Big Data*. Fonte: Elaborado pelos autores.

A cada dia que se passa o volume de dados cresce de forma exponencial. Desse modo, se vê necessária a existência de uma abordagem que tenha a capacidade de armazenar de maneira segura uma quantidade de dados crescente em tamanho e variadade. Além disso, existe a necessidade de processar e apresentar essas informações em tempo hábil para a tomada de decisões. Para o atendimento de todas essas necessidades, foi criado o conceito de *Big Data* (ALECRIM, 2015).

## Apache Storm

De acordo com Leibiusky, Eisbruch e Simonassi (2012), o *Storm* é um sistema distribuído, confiável e tolerante a falhas desenvolvido para processar fluxos de dados. Sua estrutura consiste em diferentes tipos de componentes. Desta forma o trabalho é dividido entre os componentes, cada qual sendo responsável por uma tarefa de processamento simples e específica. O fluxo de entrada do *cluster*[[14]](#footnote-14) do Storm é tratado por um componente chamado *spout*.

Segundo Apache (2015), um *spout* pode ser denominado como a fonte de fluxos em uma topologia. Geralmente, eles irão ler tuplas (sequências de objetos) de uma fonte externa (uma API de rede social, por exemplo), posterirormente encaminhando-os para a topologia.

Continuando com Leibiusky, Eisbruch e Simonassi (2012), o *spout* por sua vez direciona os dados para um componente denominado *bolt*, o qual irá transformá-los de alguma maneira. Um *bolt* pode tanto persistir os dados em algum tipo de armazenamento, quanto direcioná-lo para outro *bolt*. Para melhor entendimento, pode-se imaginar o *cluster* do Storm como uma corrente de *bolts*, cada qual executa algum tipo de transformação nos dados expostos pelo *spout*.

Leibiusky, Eisbruch e Simonassi (2012) afirmam que, dentre as propriedades do Storm, as principais são:

* Simples de programar: a complexidade é reduzida;
* Suporte a várias linguagens de programação: maior facilidade desenvolver em linguagens baseadas na JVM[[15]](#footnote-15), no entanto qualquer linguagem é suportada desde que se implemente uma pequena biblioteca intermediária;
* Tolerante a falhas: o *cluster* do Storm cuida dos trabalhos, realocando as tarefas quando necessário;
* Escalável: o Storm pode reatribuir as tarefas para as novas máquinas, caso seja necessário;
* Confiável: todas as mensagens são obrigatoriamente processadas ao menos uma vez. Em caso de erros, a mensagem pode ser processada novamente. Assim, tem-se a garantia de nunca perder uma mensagem;
* Rápido: velocidade é um dos fatores-chave do *Storm*;
* Padronizado: pode-se obter exatamente a mesma semântica de mensagens para praticamente qualquer tipo de processamento.

O Storm torna-se uma solução eficiente, rápida, confiável e de baixo custo para operações relacionadas com *Big Data*. Grandes massas de dados devem ser processadas no menor tempo possível, com a garantia que nenhum dado se perca. A ferramenta possibilita este tipo de solução, através de seu *design* e suas funcionalidades (LEIBIUSKY, EISBRUCH e SIMONASSI, 2012).

### *Data Mining*

Para Han, Kamber e Pei (2012), *Data Mining* (ou mineração de dados) pode ser definida de diversas formas. O termo “mineração” é constantemente usado para caracterizar o processo em que se encontra um conjunto de “pepitas” de uma grande quantidade de matéria-prima. Assim, nasceu o termo *Data Mining*, que em suma, representa a extração de conhecimento a partir da análise de dados.

De acordo com Han, Kamber e Pei (2012), o processo de obtenção do conhecimento por meio da mineração de dados segue a sequência iterativa dos seguintes passos:

* Limpeza dos dados: remoção de lixo e dados inconsistentes;
* Integração dos dados: combinação de múltiplas fontes de dados;
* Seleção dos dados: os dados relevantes para a tarefa são obtidos do banco de dados;
* Transformação dos dados: os dados são transformados e consolidados em formas apropriadas para a mineração por operações de resumo ou operações de agregação;
* Mineração de dados: processo onde são aplicados métodos inteligentes para a extração de padrões de dados;
* Avaliação dos padrões: identificação dos padrões verdadeiramente interessantes que representam conhecimento baseados em medidas de interesse;
* Apresentação do conhecimento: técnicas de visualização e representação do conhecimento são usadas para apresentar o conhecimento extraído para os usuários.

A Figura 5 demonstra o processo de descoberta do conhecimento baseado em *Data Mining*.

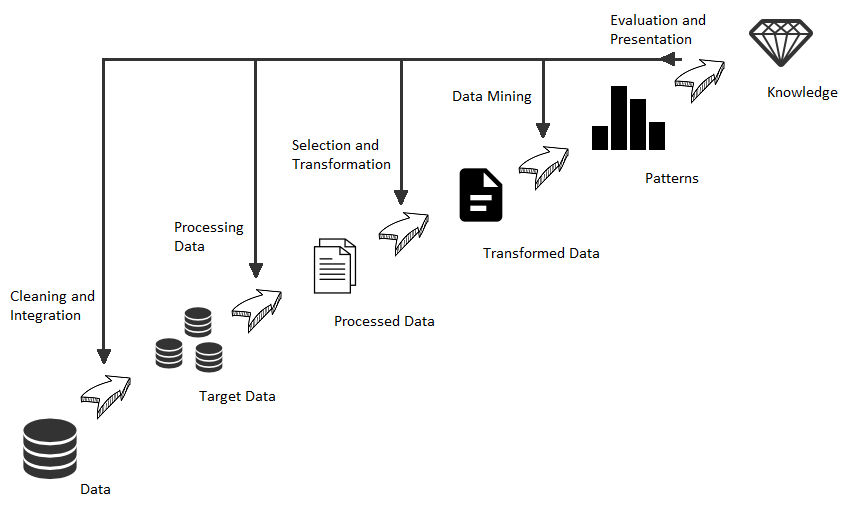


Figura 5 - *Data Mining* como um passo no processo da descoberta do conhecimento. Fonte: Elaborado pelos autores.

A mineração de dados é o processo de descobrir padrões e conhecimento de grandes quantidades de dados. As fontes de dados podem incluir bancos de dados, *data warehouses*, a web e outros repositórios de informações ou dados que serão transmitidos para o sistema de forma dinâmica (HAN, KAMBER e PEI, 2012).

### *Data Mining* em Redes Sociais

De acordo com Russel (2013), algumas das redes sociais eram consideradas inicialmente como uma simples “moda”. Posteriormente, passaram a ser consideradas como um “*mainstream*”[[16]](#footnote-16). Atualmente, chegaram ao reconhecimento de uma espécie de “grife”, chegando a mudar a maneira de vida da população dentro e fora da web.

Para Russel (2013), de um modo geral, as redes sociais, juntamente com *data mining*,análises e técnicas de visualização para explorar dados buscam responder as seguintes perguntas:

* “Quem conhece quem, e quem são os contatos comuns em suas redes sociais?”;
* “Com que frequência pessoas específicas se comunicam com outras?”;
* “Qual rede social gera mais valor para um nicho específico?”;
* “Como a geografia afeta suas conexões sociais em um mundo online?”
* “Quem são as pessoas mais influentes/procuradas em uma rede social?”;
* “Sobre o que as pessoas estão conversando? (É valioso?)”;
* “Quem são as pessoas interessadas, com base na linguagem humana, e o que eles utilizam no mundo digital?”.

A resposta a essas perguntas muitas vezes deram oportunidades valiosas e lucrativas a empresários, cientistas sociais e outros profissionais, promovendo assim, o entendimento de problemas e a descoberta de soluções (RUSSEL, 2013).

### Apache Maven

De acordo com Leibiusky, Eisbruch e Simonassi (2012) o Maven é uma ferramenta de gerenciamento e compreensão de projetos de software. Ele pode ser usado para gerenciar vários aspectos de um ciclo de desenvolvimento do projeto, a partir de dependências para o processo de compilação.

Os principais objetivos do Maven, de acordo com a Apache *Software Foundation*, é permitir que o desenvolvedor para compreenda a complexidade do esforço de desenvolvimento em um menor período de tempo. Para atingir este objetivo, existem várias áreas de preocupação que Maven tenta lidar, tais como:

* Tornar o processo de compilação fácil.
* Fornecer um sistema de construção uniforme.
* Fornecer informações sobre qualidade do projeto.
* Fornecer orientações para melhorar desenvolvimento de práticas.
* Permitir a migração transparente para novas funcionalidades.

Embora de acordo com Leibiusky, Eisbruch e Simonassi (2012) não é necessário ser um especialista em Maven para usar o Storm, porém torna-se útil saber o básico de como o Maven funciona.

## REST API

Para Massé (2011), *Representational State Transfer* (REST) é o nome de uma arquitetura web que pode ser implementada usando diferentes tecnologias. O estilo arquitetural REST é comumente aplicado ao desenvolvimento de API’s[[17]](#footnote-17) para *Web Services*[[18]](#footnote-18). Uma API em conformidade com a arquitetura REST é uma REST API.

Ainda de acordo com Massé (2011), algumas das melhores práticas para o design REST API estão implícitas no padrão HTTP, enquanto outras abordagens “pseudo-padrão” surgiram ao longo dos anos. No entanto, nos dias atuais recomenda-se continuar a buscar as respostas para uma série de questões, tais como:

* “Quando um *path* URI deve ser nomeado com substantivo no plural?”;
* “Qual método de requisição deve ser usado para atualizar o estado de recurso?”;
* “Como fazer para mapear operações não-CRUD aos URI’s?”;
* “Qual é o código de status de resposta HTTP mais apropriado para um determinado cenário?”;
* “Como gerenciar as versões de representação do estado de um recurso?”;
* “Como estruturar um *hyperlink* em JSON?”.

O conhecimento dos princípios do REST trazem benefícios não somente em desenvolvimento de aplicações web, mas também no desenvolvimento focado na evolução de uma API em um *Web Server*[[19]](#footnote-19) robusto (MASSÉ, 2011).

### Twitter *Stream* API

Para Twitter Developers (2015), as API’s de *streamig* do Twitter proporcionam aos desenvolvedores um acesso de baixa latência ao fluxo global de *tweets[[20]](#footnote-20)*. A implementação correta da API retorna os *tweets* e seus respectivos eventos, sem ocorrer qualquer sobrecarga nas extremidades da REST.

Continuando com Twitter Developers (2015), o Twitter oferece alguns tipos de serviço de *stream*, os quais podem ser citados:

* *Public Streams*: *streams* de dados públicos que fluem através do Twitter. É mais adequado para seguir usuários ou tópicos específicos, bem como mineração de dados;
* *User Streams*: *streams* de um único usuário. Retorna todos os dados de um usuário específico;
* *Site Streams*: *streams* multiusuário. Os dados são destinados a servidores, os quais se conectam ao Twitter através de muitos usuários.

As API’s de *stream* do Twitter facilitam o acesso às informações dos seus usuários. Com a implementação correta, torna-se possível fazer a mineração dos dados necessários para desenvolver uma aplicação integrada com o Twitter(TWITTER DEVELOPERS, 2015).

### Twitter4j

Para Twitter4j (2015), o Twitter4j é uma biblioteca Java não-oficial para o consumo do *Web Service* do Twitter. A principal característica da Twitter4j é a facilidade de integração entre a aplicação Java e o Twitter *Service*.

De acordo com Twitter4j (2015), são características da biblioteca Twitter4j:

* Funciona em qualquer plataforma Java na versão 5 ou posterior;
* Plataformas Android e Google *App Engine* incluídas;
* Dependência zero: não requer jar’s[[21]](#footnote-21) adicionais;
* Suporte a OAuth[[22]](#footnote-22) integrado;
* Suporte a gzip[[23]](#footnote-23);
* Compatível com a versão 1.1 do Twitter API.

A biblioteca Twitter4J possui uma série de funcionalidades simples de serem utilizadas, resultando em uma implementação rápida e sem maiores dificuldades (TWITTER4J, 2015).

## Banco de dados NoSQL

De acordo DevMedia (2013), o movimento NoSQL promove soluções de armazenamento de dados não relacionais. Suas diversas ferramentas resolvem problemas relacionados a grandes volumes de dados, execução de consultas com baixa latência e modelos flexíveis de armazenamento de dados (como XML[[24]](#footnote-24) ou JSON[[25]](#footnote-25)).

O objetivo do NoSQL não é de substituir os bancos de dados relacionais, mas sim apresentar algumas soluções, que em determinados momentos, são mais adequadas e eficientes. Assim, nada se impede de se trabalhar com as tecnologias baseadas em NoSQL e os tradicionais bancos de dados relacionais em um mesmo projeto (DEVMEDIA, 2013).

Para DevMedia (2013), os tipos de bancos de dados NoSQL são:

* Chave-Valor: armazenam objetos indexados por chaves, e possibilitam sua busca a partir de sua respectiva chave.
* Documentos: conjunto de documentos no formato JSON. Os documentos são tratados como objetos únicos, os quais possuem campos com os respectivos valores.
* Coluna: formado por colunas que contém um conjunto de informações, semelhante a uma tabela.
* Grafos: formado por dados distribuídos em forma de vértices e arestas, os quais possuem atributos tanto nas arestas quanto nos vértices.

Para Sadalage e Fowler (2012), o uso do NoSQL que é reconhecido em dias atuais, remonta a um encontro que ocorreu em 11 de junho de 2009 em São Francisco, EUA, organizado pelo desenvolvedor de software Johan Oskarsson. Um exemplo que foi desenvolvido usando BigTable[[26]](#footnote-26) e Dynamo[[27]](#footnote-27) havia inspirado diversos projetos a experimentar um armazenamento de dados alternativo, e discussões a respeito do NoSQL haviam se tornado uma característica das melhores conferências de software em torno dessa época.

O nome “NoSQL” vem do fato de não ser usado SQL[[28]](#footnote-28) como uma linguagem de consulta aos seus registros. Ao invés disso, o banco de dados NoSQL é manipulado através de *shell scripts*[[29]](#footnote-29) que podem ser combinados nos *pipelines* do UNIX[[30]](#footnote-30) (SADALAGE e FOWLER, 2012).

O banco de dados do tipo NoSQL tem como característica o armazenamento de suas tabelas como arquivos do tipo ASCII[[31]](#footnote-31). Cada uma de suas tuplas é representada por uma linha, que tem os seus campos separados por guias (SADALAGE e FOWLER, 2012).

Continando com DevMedia (2013), os bancos de dados NoSQL são implantados quando os bancos de dados relacionais já não estão suportando a demanda de dados, ou seja, estão apresentando lentidão em consultas e processamentos. Um NoSQL pode substituir um banco de dados relacional por completo, ou simplesmente, ser implantado como um “suporte”, realizando as consultas/processamentos em um grande volume de dados onde a velocidade é necessária. O tipo de NoSQL é escolhido após uma análise crítica do gestor de TI da organização, dependendo da necessidade da mesma.

### MongoDB

Segundo Chodorow (2013), o MongoDB é um banco de dados NoSQL poderoso, flexível e escalável. Ele combina a capacidade de dimensionamento com características tais como índices secundários, consultas por abrangência, classificação, agregações e índices geoespaciais.

O MongoDB não é um banco de dados relacional, mas sim um banco de dados orientado a documentos. Uma das principais razões para a migração dos tradicionais modelos relacionais é a necessidade de fazer um escalonamento de modo mais fácil (CHODOROW, 2013).

Um banco de dados orientado a documentos substitui o tradicional conceito de uma linha, oriundo dos bancos de dados relacionais, com um modelo mais flexível, o documento. Ao permitir a incorporação de documentos e *arrays*[[32]](#footnote-32), o banco de dados orientado a documento permite representar relações hierárquicas mais complexas com um único registro. Isso se encaixa naturalmente na forma como os desenvolvedores de linguagens orientadas a objetos pensam sobre seus dados (CHODOROW, 2013).

O MongoDB foi projetado para ser escalonável. O seu modelo de dados orientado a documentos permite separar os dados entre vários servidores de maneira mais fácil. Ele se encarrega automaticamente de balancear a carga de dados através do *cluster*. Assim, ele consegue redistribuir documentos automaticamente e rotear as solicitações dos usuários para as máquinas corretas. Esta característica permite que os desenvolvedores se concentrem no desenvolvimento da aplicação, não no escalonamento (CHODOROW, 2013).

Caso um *cluster* necessite de mais capacidade, novas máquinas podem ser adicionadas, e automaticamente, o MongoDB irá descobrir como os dados devem ser distribuídos entre elas (CHODOROW, 2013).

## Node.JS

De acordo com Pereira (2013), os sistemas web desenvolvidos sobre plataforma .NET, Java, PHP, Ruby ou Pythoncompartilham da mesma característica da paralisação de processamento enquanto utilizam um I/O[[33]](#footnote-33) no servidor. Essa paralisação é conhecida com modelo bloqueante (*Blocking-Thread*). O problema é que essas arquiteturas se tornam ineficientes em alguns casos, pois elas mantêm uma fila ociosa enquanto as operações de I/O são executadas, bloqueando totalmente o sistema. Desse modo, o aumento de acessos no sistema geraram *gap’s*[[34]](#footnote-34)mais frequentes, aumentando a necessidade de fazer um *upgrade* de *hardware* nos servidores, o que é financeiramente inviável.

Baseado neste problema que em 2009, Ryan Dahl e mais 14 colaboradores criaram o Node.JS. O Node.JS tem um modelo inovador, sendo sua arquitetura não-bloqueante (*non-blocking thread*), apresentando uma boa performance e baixo consumo de memória (PEREIRA, 2013).

A Figura 6 demonstra a arquitetura não-bloqueante do Node.JS, enquanto a Figura 7 demonstra a arquitetura bloqueante, proveniente de outras plataformas.

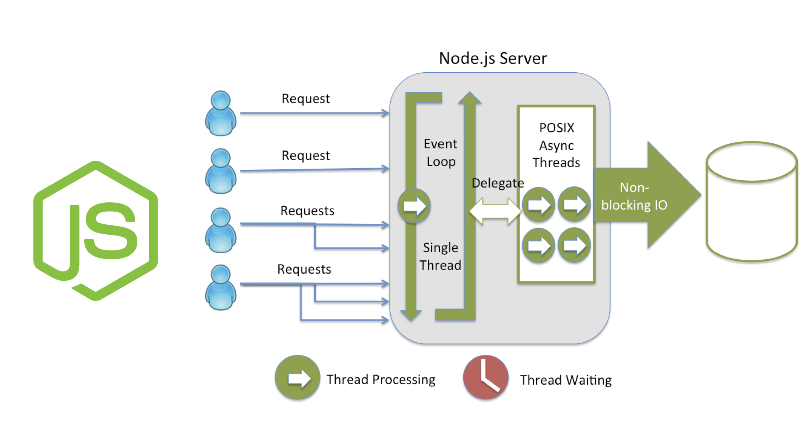


Figura 6 - Arquitetura não-bloqueante. Fonte: StrongLoop (2014).

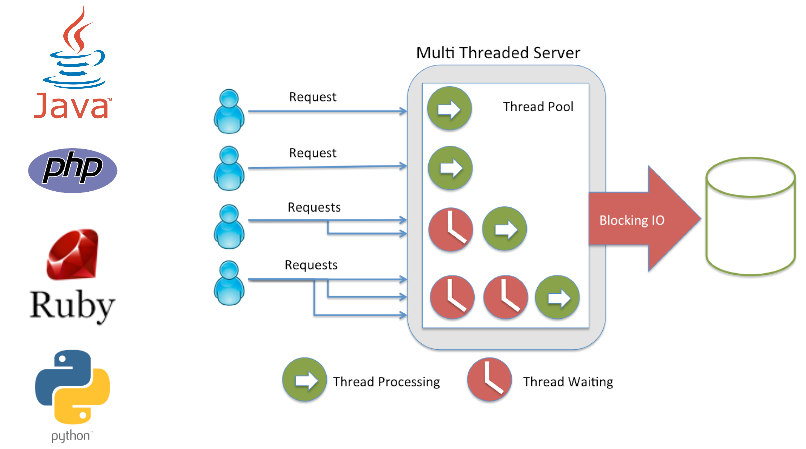


Figura 7 - Arquitetura bloqueante. Fonte: StrongLoop (2014).

O Node.JS é uma plataforma escalável e de baixo nível, pois a programação é feita diretamente com protocolos de rede e bibliotecas que acessam os recursos do sistema operacional (PEREIRA, 2013).

A linguagem de programação do Node.JS é o JavaScript. Isso se deu graças à *engine* JavaScriptV8, utilizada também no navegador Google Chrome (PEREIRA, 2013).

### Express

Para Brown (2014), o Express é um web *framework*[[35]](#footnote-35) Node.JS minimalista e flexível que fornece um conjunto robusto de recursos para a criação das mais diversas aplicações web.

De acordo com Brown (2014), o Express tem como características:

* Minimalista: a filosofia do Express é proporcionar uma camada mínima entre o desenvolvedor e o servidor;
* Flexível: como o Express fornece um *framework* minimalista, fica a cargo do desenvolvedor adicionar ao Express diferentes partes de sua funcionalidade quando for necessário, substituindo assim o que não atender às suas necessidades;
* Web *Framework*: possibilita a construção de páginas web, sites e aplicações web. É fácil imaginar que em poucos anos, não existirá mais a distinção entre eles;
* Suporte a aplicações web de página única: em vez de um site fazendo requisições de rede à medida que o usuário vai navegando por páginas diferentes, uma aplicação web de página única faz o *download* do site inteiro para o cliente. O desenvolvimento desse tipo de aplicação é facilitado pelo Express;
* Suporte a aplicações web Multi-páginas e Aplicações web híbridas: as aplicações web Multi-páginas usam a abordagem tradicional para o desenvolvimento de websites, enquanto as aplicações web híbridas usam em conjunto as aplicações Web de página única e as aplicações web Multi-páginas. O desenvolvimento de ambas é facilitado pelo Express.

O Express facilita o desenvolvimento de aplicações web que utilizam o Node.JS como servidor. Ele é minimalista, robusto e flexível, além de ter um eficiente sistema de roteamento, um executável para a geração de aplicações, dentre outras vantagens (NODEBR, 2013).

## *Cloud Computing*

Para Marinescu (2013), o ideal que a computação pode ser livremente distribuída como uma utilidade pública, do mesmo modo que a água e a eletricidade foi formulado na década de 1960 pelo cientista da computação e visionário John McCarthy, o mesmo que defendeu a lógica matemática na inteligência artificial. Cerca de quatro décadas depois, com a internet já consolidada, a causa da “utilidade computacional” foi resgatada por grandes empresas de TI, tais como Amazon, Apple, Google, HP, IBM, Microsoft e Oracle.

O movimento da computação em nuvem foi um movimento iniciado em algum período em meados da primeira década do novo milênio, cuja ideia principal é que o processamento de informações pode ser feito com mais eficiência por meio de métodos de armazenamento via internet, sendo assim alocados em um “*server farm*”, ou seja, um conjunto de servidores com capacidades computacionais extremamente superiores a de um computador convencional (MARINESCU, 2013).

Uma das principais vantagens da computação em nuvem é que ela oferece serviços computacionais e de armazenamento escaláveis e elásticos. Todos os recursos utilizados para estes serviços são medidos. Com isso, os usuários serão somente cobrados pelos serviços que eles consomem. Aplicações científicas e de engenharia, mineração de dados, jogos, redes sociais, e muitas outras atividades computacionais de uso intensivo de dados podem se beneficiar da computação em nuvem (MARINESCU, 2013).

A arquitetura da computação em nuvem oferece três tipos de serviços: Software como Serviço (SaaS), Plataforma como Serviço (PaaS) e Infraestrutura como Serviço (IaaS). Os serviços baseados em SaaS são mais apropriados ao usuário final. Os serviços baseados em PaaS são mais voltados a área de desenvolvimento, oferecendo assim poderosas ferramentas ao desenvolvedor. Já os serviços baseados em IaaS oferecem serviços de infraestrutura de TI completos, liberando assim a organização para focar nos negócios (MARINESCU, 2013).

A computação em nuvem já é uma realidade empresarial em dias atuais. O número de organizações que estão adotando o paradigma é crescente (MARINESCU, 2013).

### Amazon *Web Services*

De acordo com Murty (2008), o Amazon *Web Services* é um conjunto de *web services* disponibilizados pela Amazon[[36]](#footnote-36) que permitem aos desenvolvedores acessar e desenvolver sobre a sua plataforma tecnológica. Neste conjunto de *web services*, estão inclusos diversos serviços de infraestrutura, os quais têm a finalidade de expansão ou substituição da infraestrutura física tradicional de aplicações web (MURTY, 2008).

Estes serviços de infraestrutura fornecem armazenamento, capacidade computacional, um sistema de mensagens, um sistema de pagamento e um banco de dados que pode ser acessado por qualquer pessoa, desde que possua uma conta na Amazon.com e um cartão de crédito. A principal característica é: os usuários pagam somente pelo que usam (MURTY, 2008).

Esses serviços oferecem uma alternativa ao tradicional desenvolvimento de aplicações sob o hardware físico, pois eles fornecem componentes de infraestrutura escalável, confiável e rentável sem exigir muito esforço, ao contrário da montagem do hardware proprietário. Desse modo, o foco se torna a aplicação em vez da plataforma física (MURTY, 2008).

Outra vantagem desses serviços é a redução significativa do investimento inicial exigido para construir e oferecer um serviço. A partir do momento em que a gestão de infraestrutura está sob a responsabilidade da Amazon, o foco se torna o desenvolvimento da aplicação e o crescimento dos negócios. Desse modo, a aplicação web está liberada para competir e ter sucesso com base em suas ideias, inovação e execução (MURTY, 2008).

## *Open Web Plataform*

De acordo com W3C (2015), a *Open Web Plataform* é uma coleção de tecnologias abertas que podem ser implementadas sem a necessidade de qualquer aprovação ou pagamento de taxas de licença.

Seguindo esse conceito, nesta seção serão descritas as tecnologias da *Open Web Plataform* que serão utilizadas neste projeto.

### HTML5

Segundo W3C (2015), a WWW (*World Wide Web*) foi criada em 1989, por Tim Berners-Lee no CERN (*European Organization for Nuclear Research*[[37]](#footnote-37)) em Genebra, na Suíça. Berners-Lee estava trabalhando em uma seção de serviços de computação no CERN, quando ele teve a ideia de reunir e organizar as pesquisas de diversos cientistas do mundo todo. Mais do que somente reunir, ele propôs que os textos poderiam ter *links*[[38]](#footnote-38) entre si, ou seja, durante a leitura de uma pesquisa, seria possível rapidamente exibir parte de outro artigo que contém texto ou diagramas diretamente relevantes.

Como estava tudo muito recente, era necessário ter uma implementação relativamente simples. Seguindo esse conceito, Berners-Lee desenvolveu o protocolo HTTP. Desse modo, o formato de texto escrito para o HTTP foi nomeado de HTML. Entre 1993 e 1997, o HTML iria passar pelas versões HTML+, HTML 2.0, HTML 3.0, HTML 4.0 e HTML 4.0.1, todas abrangendo diversas propostas para enriquecer as possibilidades da linguagem. Em 2009, foi lançado o HTML5, sua atual versão (W3C, 2015).

Ainda de acordo com W3C (2015), o HTML5 tem como uma de suas principais funções a facilidade de manipulação do elemento. Com isso, o desenvolvedor pode modificar as características dos objetos de forma não intrusiva e transparente para o usuário final. Contrariamente as suas versões anteriores, o HTML5 fornece um número de ferramentas para o CSS[[39]](#footnote-39) e o JavaScript[[40]](#footnote-40) executarem suas funções de maneira mais eficiente possível.

Continuando com W3C (2015), HTML é uma abreviação de *Hypertext* *Markup* *Language* - Linguagem de Marcação de Hypertexto. A função do HTML é a publicação de conteúdo (texto, imagem, vídeo, áudio e etc) na web.

### CSS3

Segundo Pereira (2009), o CSS nasceu das complicações de aparência e da diferença de compatibilidade dos navegadores das páginas HTML. A principal função do CSS é separar o conteúdo e formato de um documento de sua apresentação, incluindo elementos como cores, formatos de fontes e layout, além de ser responsável por definir animações e transições. Com isso, foi proporcionada uma maior flexibilidade e controle na especificação de como as características serão exibidas, além de permitir um compartilhamento de formato e reduzir a repetição no conteúdo estrutural de uma página.

Para Mozilla Developer Network (2015), o CSS3 é a mais recente evolução do CSS. Dentre suas novidades, estão cantos arredondados, sombras, gradientes, transições ou animações, bem como novos *layouts* como multiplas colunas, caixa flexível ou *layouts* de grade. As partes experimentais devem ser usadas com cautela, pois tanto sua sintaxe quanto sua semântica podem mudar no futuro.

### JavaScript

De acordo com Powers (2009), a linguagem JavaScript foi inicialmente destinada a ser uma interface *script* entre uma página web carregada no cliente do navegador (na época, o Netscape) e a aplicação no servidor. Desde a sua introdução em 1995, JavaScript se tornou um componente chave do desenvolvimento web, além da descoberta seu uso em outros lugares.

Segundo Microsoft (2015), a linguagem JavaScript é uma linguagem de *script* interpretada com base em objetos. Sua sintaxe é semelhante a sintaxe da linguagem C.

Continuando com Powers (2009), JavaScript é uma linguagem fracamente tipada, ou seja, os tipos de dados das variáveis não são declarados de forma explícita. Na maioria das vezes, o JavaScript executa conversões de tipo das varáveis automaticamente.

## Bootstrap

De acordo com Magno (2012), o Bootstrap é um *framework* *front-end*[[41]](#footnote-41) intuitivo criado para facilitar o desenvolvimento dos elementos de interface em páginas web. Ele pode ser também utilizado como um guia para reproduzir de forma consistente os padrões de desenvolvimento consolidados pelo Twitter. O Bootstrap também pode ser utilizado para facilitar na padronização e nas melhores práticas de desenvolvimento HTML/CSS e JavaScript, tanto para iniciantes, quanto para desenvolvedores com habilidades mais avançadas que desejam dar um passo além em interações mais complexas.

## Google Maps API

De acordo com Svennerberg (2010), O Google Maps foi inicialmente introduzido em um *post* de um *blog* em 2005, onde, até então, as soluções de mapa da época eram caras, necessitando servidores especiais exclusivos para os sistemas de mapas. A partir de então, ocorreu uma revolução do modo como os mapas em páginas web funcionavam, trazendo um maior nível de interatividade e menor custo. Desse modo, em junho de 2005 foi lançada a primeira versão pública da API do Google Maps. Atualmente, o Google MapsAPI já se encontra em sua terceira versão, lançada em Maio de 2010.

Continuando com Svennerberg (2010), o funcionamento do Google Maps API é uma interação entre HTML, CSS e JavaScript. O mapa é composto por imagens que são carregadas em segundo plano por meio de chamadas AJAX. Posteriormente, o conteúdo é inserido em uma *div* na página HTML.

Conforme é feita a navegação no mapa, a API envia informações sobre novas coordenadas e intensidade de zoom no mapa, renderizando assim novas imagens.

Ainda de acordo com Svennerberg (2010), a API consiste em arquivos JavaScript que contém classes com os métodos e propriedades que definem o comportamento do mapa, fazendo assim com que o mesmo tenha um comportamento dinâmico.

## Highcharts

Para Kuan (2012), o Highcharts é uma biblioteca JavaScript de geração de gráficos em duas dimensões desenvolvida em 2009 por uma empresa norueguesa chamada Highsoft *Solutions* AS. Embora o Highcharts apresente somente gráficos em 2D, sua interface é atraente e profissional, com o uso de cores sem conflito de contrastes, auxiliadas pela sombra sutil e o uso de efeitos de borda. Sua implementação não necessita inicialmente de nenhum *framework* em particular. Isso permite aos desenvolvedores decidirem com mais liberdade qual *framework* será o mais adequado para seu projeto, ou até mesmo, fazer a implementação do Highcharts sem comprometer o que já foi previamente desenvolvido.

# QUADRO METODOLÓGICO

Neste quadro metodológico são apresentadas as informações e procedimentos definidos para realização e conclusão do projeto. São apresentados o tipo de pesquisa, contexto, instrumentos, procedimentos e resultados da pesquisa.

## Tipo de Pesquisa

Uma pesquisa tem como objetivo a obtenção de novos conhecimentos, tomando por base a utilização de procedimentos científicos. Ela contribui para a solução dos problemas e processos eventuais nas mais diversas atividades humanas, em ações comunitárias, no processo de formação e outros. Dessa forma, o conhecimento se torna uma ferramenta para o desenvolvimento do ser humano e a pesquisa uma consolidação da ciência (SILVA, 2008).

Para Oliveira (2002, p. 62), “A pesquisa, tanto para efeito científico como profissional, envolve a abertura de horizontes e a apresentação de diretrizes fundamentais, que podem contribuir para o desenvolvimento do conhecimento”.

Abrangendo o contexto de pesquisa, este projeto tomou por base a metodologia de pesquisa aplicada, que é utilizada quando o resultado final é um produto real, o qual pode ser aplicado em um determinado contexto.

De acordo com Marconi e Lakatos (2009, p. 6), “Pesquisa aplicada caracteriza-se por seu interesse prático, isto é, que os resultados sejam aplicados ou utilizados, imediatamente, na solução de problemas que ocorrem na realidade”.

Após a análise e aplicação os conceitos de pesquisa, foram desenvolvidos um sistema de mineração e armazenamento dos *tweets* de ênfase política do Twitter, e uma aplicação web, que tem como objetivo a demonstração destes *tweets* e suas informações, tanto em tempo real quanto em busca por data. Além da mineração e demonstração dos dados, tem-se o objetivo de demonstrar como foi feita integração das tecnologias utilizadas para análise de *Big Data*.

## Contexto de pesquisa

Atualmente, devido a facilidade de acesso à internet, dia após dia são geradas grandes quantidades e dados. Esses dados contém uma variedade imensa de informações. Eles podem ser gerados por bancos, operadoras de telefonia, redes varejistas ou até mesmo uma pessoa comum. Mas de nada serve uma quantidade inimaginável de dados se ela não for analisada. Para suprir essa necessidade, foi criado o conceito de *Big Data*.

Não é difícil imaginar o cenário em que este conceito se aplica. Por exemplo, podem ser citados os milhares de e-mails trocados por dia, bem como o número de transações bancárias e de *posts* de uma rede social.

Esta pesquisa demonstra como soluções baseadas em *Big Data* tem o poder de influenciar e auxiliar determinadas áreas de conhecimento do ser humano, por meio da análise, gerenciamento e demonstração dos dados.

As soluções de *Big Data* possibilitam aos analistas um melhor entendimento de determinado produto, serviço ou situação. Desse modo, torna-se capaz de ser feita uma melhoria ou reestruturação para evitar desperdício de recursos, tal como melhorar uma produção em quantidade, qualidade e tempo. Esses fatores podem ser decisivos para o futuro de uma empresa.

Desse modo, o objetivo desta pesquisa é demonstrar como uma solução baseada em *Big Data* se torna eficiente e decisória no âmbito político, apresentando as opiniões públicas dos usuários do Twitter de forma rápida e clara, além da geração de um gráfico para ser feita a análise de quantidade de menções por político cadastrado. Esta pesquisa, por envolver diversas tecnologias relacionadas à *Big Data*, servirá de base para a formação do conhecimento de estudantes da área de tecnologia, bem como o aprimoramento do conhecimento de profissionais da área.

## Instrumentos

Para ser feita a escolha do tipo de instrumento, se faz necessária a observação do que será estudado.

Na realização de uma pesquisa, segundo Oliveira (2002, p. 66):

Depois de definidas as fontes de dados e o tipo de pesquisa, que pode ser de campo ou de laboratório, devemos levantar as técnicas a serem utilizadas para a coleta de dados, destacando-se: questionários, entrevistas, observação, formulários e discussão em grupo.

A coleta de dados foi realizada por meio de livros e materiais relacionados com o tema, trabalhos e artigos acadêmicos, páginas na internet e pesquisas bibliográficas.

Para ter uma visão profissional do tema, foi realizada uma entrevista com Lucas Vilela, especialista em sistemas no Inatel *Competence Center* - ICC. Lucas tem experiência profissional com tecnologias relacionadas à *Big Data*.

A entrevista foi realizada pessoalmente em 9 de julho de 2015, no campus do Inatel. Lucas compartilhou experiências acerca dos conceitos e funcionamento das tecnologias envolvidas, além dos padrões de desenvolvimento que poderiam ser aplicados ao projeto.

A entrevista foi relevante para o desenvolvimento do projeto, pois por meio dela foi possível compreender e analisar como e com quais informações a aplicação deveria ser construída.

Também foram feitas reuniões semanais com o orientador da pesquisa, com a finalidade de demonstrar os protótipos da aplicação, bem como identificar possíveis falhas e melhorias dos mesmos. Nessas reuniões também era apresentada ao orientador a pesquisa teórica, com a finalidade de identificar correções e melhorias da mesma.

## Procedimentos e Resultados

Nesta pesquisa foi desenvolvida uma aplicação web que demonstra como soluções baseadas no conceito de *Big Data* podem impactar nas diferentes áreas do conhecimento, mais especificamente na área política. Foram utilizadas tecnologias de mineração, armazenamento e demonstração de grandes quantidades de dados.

Para organizar e facilitar o desenvolvimento do projeto, foi previamente definido um cronograma estimando a quantidade de tempo que cada uma das atividades durariam até serem finalizadas. As atividades progrediam conforme era feito o estudo, implementação e análise dos resultados de cada tecnologia empregada no projeto.

### Configuração do ambiente

O primeiro passo tomado para a criação do sistema desta pesquisa foi configuração do ambiente de desenvolvimento. O sistema operacional usado para o desenvolvimento foi o Ubuntu 14.04 LTS, um sistema operacional de código aberto baseado no Linux. O sistema operacional já se encontrava previamente instalado na máquina de desenvolvimento.

Como o sistema operacional já estava devidamente instalado, o próximo passo a ser tomado era a instalação dos pré-requisitos de ambiente do sistema. Os pré-requisitos de ambiente do sistema são demonstrados e detalhados no Apêndice I.

Para o desenvolvimento do *front-end* e *back-end* da aplicação, foi feito o *download* da IDE WebStorm, através de seu próprio site. Após feito o *download*, o arquivo foi descompactado em uma pasta. Logo em seguida, a pasta foi movida para o diretório “/Documents”, na intenção de facilitar sua localização.

Posteriormente, foi executada a linha de comando “~/Documents/WebStorm-10.0.0/bin/WebStorm.sh”, que consistia na execução de um *script* Linux para a instalação e configuração da IDE.

Como os pré-requisitos, bem como a IDE de desenvolvimento da aplicação web, estavam instalados, viu-se a necessidade de criar a arquitetura do sistema, visando uma melhor organização da aplicação. Ao executar a linha de comando “express politistatus -ejs”, foi criada a árvore de diretórios do projeto, no padrão MVC. O parâmetro “-ejs” da linha de comando ativou o sistema para o uso do EJS *template engine.* Inicialmente, a estrutura de diretórios gerada pelo Express ficou da seguinte forma:

* ***views*:** diretório responsávelpelos arquivos de tela da aplicação;
* ***public*:** diretório responsável pelos arquivos estáticos da aplicação web, como JavaScripts, *stylesheets*, imagens, dentre outros;
* ***routes*:** diretório responsável pelos arquivos de rota da aplicação;
* ***app.js*:** arquivo JavaScript que contém a *stack* de configurações responsável pela inicialização da aplicação, através do comando “npm start”;
* ***package.json*: a**rquivo JSON contendo as informações da aplicação, tais como: versão do projeto, controle de módulos, dependências entre outros. É por meio deste arquivo que se torna possível a atualização dos módulos a aplicação, através do comando “npm install”.

Para deixar o arquivo package.json com as informações corretas sobre o sistema, foram configurados os atributos *name*, *version*, *description* e *author*, informando respectivamente o nome, a versão, a descrição e os autores da aplicação. O atributo *private* foi atribuído como *false*, informando que a aplicação desenvolvida é *open-souce*. Dentre todas as configurações de atributos, a mais importante do arquivo *package.json* é a versão, pois sem ela é impossível instalar e/ou atualizar os módulos por meio do comando “npm”. A estrutura deste arquivo é demonstrada na Figura 8.



Figura 8 - Arquivo package.json. Fonte: Elaborado pelos autores.

Após todas estas etapas, o ambiente básico para o desenvolvimento da aplicação estava criado. Porém, devido à necessidade da pesquisa, foram adicionados ao sistema os seguintes diretórios:

* ***controllers*:** diretório responsável pelos arquivos controladores das *Views*;
* ***models*:** diretório responsável pelos arquivos de *persistence* dos modelos do banco de dados;
* ***services*:** diretório responsável pelos arquivos de conexão com o banco de dados, possibilitando assim as operações de *create*, *retrieve*, *update* e *delete*.

A estrutura de diretórios é demonstrada na Figura 9.

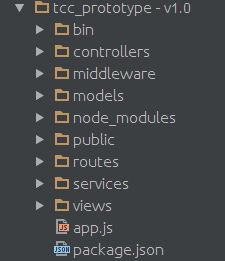
****

Figura 9 - Árvore de diretórios da aplicação web. Fonte: Elaborado pelos autores.

Na árvore de diretórios, encontram-se também as pastas *bin* e *node\_modules*, as quais contém respectivamente o arquivo de inicialização do servidor web e os módulos Node.JS necessários para a execução do sistema.

Ao final das configurações da aplicação web, se viu necessário o *download* e configuração da IDE de desenvolvimento Eclipse Luna. Essa IDE é específica para desenvolvimento de projetos na linguagem Java. Ela foi empregada no desenvolvimento do *framework* de mineração de dados Apache Storm.

Como os pré-requisitos de ambiente do Storm já se encontravam instalados, viu-se a necessidade de configurar o Apache Maven, para poder ser feito o gerenciamento das dependências do Storm. Para o funcionamento correto do Maven, foi primeiramente necessário configurar o repositório do clojars.org, por este não ser um repositório padrão do Maven. No arquivo pom.xml, foram adicionadas as linhas de código demonstradas no Código 1.

1. <repositories>
2. <!-- Repository where we can found the storm dependencies -->
3. <repository>
4. <id>clojars.org</id>
5. <url>http://clojars.org/repo</url>
6. </repository>
7. </repositories>

Código 1 - Adição do repositório CloarJS.org no arquivo pom.xml. Fonte: Elaborado pelos autores

A integração das dependências também foi feita através do Maven. Além da dependência do próprio Apache Storm, foram adicionadas também as dependências do Commons-HTTP, do Redis.clients, da biblioteca Twitter4j, do Org.JSON, e do MongoDB. O

Código 2 demonstra a adição dessas dependências no arquivo pom.xml.

1. <dependencies>
2. <dependency>
3. <groupId>storm</groupId>
4. <artifactId>storm</artifactId>
5. <version>0.7.1</version>
6. </dependency>
7. <dependency>
8. <groupId>commons-http</groupId>
9. <artifactId>commons-http</artifactId>
10. <version>1.1</version>
11. <type>jar</type>
12. <scope>compile</scope>
13. </dependency>
14. <dependency>
15. <groupId>redis.clients</groupId>
16. <artifactId>jedis</artifactId>
17. <version>2.0.0</version>
18. <type>jar</type>
19. <scope>compile</scope>
20. </dependency>
21. <dependency>
22. <groupId>org.twitter4j</groupId>
23. <artifactId>twitter4j-stream</artifactId>
24. <version>3.0.3</version>
25. </dependency>
26. <dependency>
27. <groupId>org.json</groupId>
28. <artifactId>json</artifactId>
29. <version>20140107</version>
30. </dependency>
31. <dependency>
32. <groupId>org.mongodb</groupId>
33. <artifactId>mongo-java-driver</artifactId>
34. <version>2.13.2</version>
35. </dependency>
36. <dependency>
37. <groupId>org.mongodb</groupId>
38. <artifactId>bson</artifactId>
39. <version>2.13.2</version>
40. </dependency>
41. </dependencies>

Código 2 - Dependências adicionadas no arquivo pom.xml. Fonte: Elaborado pelos autores.

Após serem feitas as configurações das dependências, executou-se o comando “maven install” para ser feito o *download* das bibliotecas necessárias para o desenvolvimento do *framework* de mineração de dados do Twitter.

Ao final de todas essas configurações, a estrutura-base e o ambiente de desenvolvimento estavam prontos para se iniciar a fase de protótipos do projeto.

### Desenvolvimento dos protótipos

O desenvolvimento dos protótipos da aplicação deu-se através dos graduais estudos sobre as tecnologias envolvidas. O desenvolvimento dos protótipos serviu de base prática para a geração de conhecimentos relacionados ao projeto, bem como testes de funcionamento e integração das tecnologias.

Gradualmente, ao serem desenvolvidos os protótipos da aplicação web, foi se consolidando que a mesma teria a necessidade de ser *single-page*, onde todas as informações principais do sistema estariam disponíveis em sua página inicial. Do mesmo modo, foi definido que o sistema deveria ser multiusuário, devido a aplicação estar disponível na internet.

Depois de feitas essas definições, começou-se a desenvolver versões finais dos protótipos das telas, usando as tecnologias HTML5, CSS3, JavaScript e o *framework* Bootstrap, além da *template engine* EJS. As telas desenvolvidas foram:

* **Tela de *login*:** desenvolvida para sanar a necessidade multiusuário da aplicação;
* ***Partial View* de cadastro de usuário:** desenvolvida, juntamente com a tela de *login*, para ser feito o cadastro dos usuários do sistema;
* **Página incial da aplicação:** é a página principal da aplicação, onde estão todas as suas informações importantes;
* ***Partial View* de cadastro de palavra-chave:** desenvolvida para ser feito o cadastro dos políticos, os quais serão buscados como palavras-chave no nos comentários do Twitter;
* ***Partial View* de edição de palavra-chave:** desenvolvida para ser feita a edição das palavras-chave (nome dos políticos);
* ***Partial View* de exclusão de palavra-chave:** desenvolvida para ser feita a exclusão das palavras-chave (nome dos políticos);
* ***Partial View* de *Stream* e filtragem dos *tweets* minerados:** desenvolvida em conjunto com a página inicial, essa modal tem como finalidade listar os *tweets* retornados pelo Apache Storm, tanto em tempo de ocorrência dos mesmos, quanto filtrados por data*.* Esse último lista os *tweets* mais antigos de acordo com a data inicial e final informadas pelo usuário.
* **Página “Sobre”:** página onde existem as informações relacionadas ao projeto, como autores, contato com os desenvolvedores e o *link* para ser feito o acesso ao controlador de versão do projeto.

Na página inicial da aplicação foi inserida uma *grid*, que corresponde à listagem de todas as palavras-chave cadastradas pelo usuário. Também foi inserida na página inicial algumas informações do usuário logado, como nome e sobrenome previamente cadastrados.

Paralelamente ao desenvolvimento dos protótipos de *front-end*, foram desenvolvidos os protótipos de *back-end* Node.JS. À medida que o as telas precisavam das informações a serem mostradas, eram criadas rotinas no *server-side* da aplicação.

Em relação ao Apache Storm, a fase de prototipação consistiu inicialmente na implementação e adequação de necessidade das classes Java disponibilizadas pela biblioteca Twitter4j, onde foram realizados testes de conexão com o Twitter *Sream* API e visualização de resultados retornados pelo mesmo. Posteriormente, à medida que foi aprimorada a aprendizagem da tecnologia, foi feita a conexão do *framework* com o banco de dados MongoDB, para ser realizada a busca dos dados cadastrados pela aplicação web. Desse modo, os dados retornados são salvos no banco e posteriormente consumidos pela aplicação web.

A fase de desenvolvimento dos protótipos do sistema foi uma etapa essencial para a elaboração do trabalho. Através dela, foram adquiridos os conhecimentos necessários para desenvolver a versão final da aplicação, além de ser feita a comprovação que a integração entre as tecnologias do projeto tornam possível a demonstração do conceito de *Big Data*.

### Desenvolvimento da aplicação

A aplicação final consiste em dois sistemas com finalidades distintas: A aplicação web, que é dividida em *front-end* e *back-end*, e o *framework* de mineração de dados. Os dois sistemas são desacoplados, porém eles têm acesso aos mesmos registros do banco de dados. Desse modo, o banco de dados serve como um “elo” entre os dois sistemas.

O *front-end* da aplicação web é uma interface entre o usuário e a aplicação. Sua função é basicamente fornecer ao usuário todas as informações que ele necessita através de consultas no *back-end*. Todas as informações que o usuário requisita são geradas e demonstradas através do *front-end*.

Também fazendo parte do *front-end*, está um mapa constantemente atualizado, que informa ao usuário da aplicação quais são os locais do Brasil onde estão sendo realizados os *tweets*. Essa informação é demonstrada para os usuários por meio de marcadores que indicam as localizações no mapa.

Juntamente com o mapa, existem dois botões com finalidades distintas. Na parte superior do mapa, está um botão que, quando clicado, demonstra para o usuário um gráfico de pizza com a quantidade de menções no Twitter por cada palavra-chave cadastrada. Já na parte inferior, está um botão que, quando clicado, efetua a limpeza dos marcadores do mapa.

O *back-end* da aplicação é responsável por atender todas as requisições do usuário. Toda requisição feita pelo usuário no *front-end* chega obrigatoriamente no *back-end*. O *back-end* da aplicação web foi inteiramente desenvolvido na linguagem de programação JavaScript, possibilitando assim, ser feita a execução do mesmo por meio do servidor Node.JS.

O *framework* de mineração de dados é composto pela junção entre o Apache Storm, a biblioteca Twitter4j, o Apache Maven e o Twitter *Stream* API. Sua função é conectar no Twitter, por meio da conexão realizada entre a implementação da biblioteca Twitter4j e o Twitter *Stream* API, e minerar os *tweets* em tempo real, com base em palavras-chave cadastradas pelos usuários da aplicação web. Desse modo, se o corpo *tweet* contiver alguma palavra-chave específica, ele é enviado para a topologia de processamento de dados do Storm, para que posteriormente ele possa ser salvo no banco de dados. Desse modo, é gerada a base *Big Data* do sistema, a qual estará disponível para consulta na aplicação web.

Desse modo, a Figura 10 demonstra a arquitetura completa do sistema, composta pelas tecnologias de mineração e processamento de dados, a tecnologia de armazenamento, as tecnologias de *back-end*, as tecnologias de *front-end* e pela tecnologia de armazenamento na nuvem.



Figura 10 - Arquitetura do sistema. Fonte: Elaborado pelos autores.

Os tópicos seguintes têm a finalidade de fazer o detalhamento do desenvolvimento e hospedagem do sistema, para que assim, possam ser obtidos os devidos conhecimentos sobre esta pesquisa.

### *Back-end*

O *Back-end* da aplicação web consiste em um servidor web capaz de atender as demandas feitas pelo usuário. Para que isso ocorra, foi desenvolvido um sistema de rotas, para que cada requisição feita chame o seu respectivo *controller*. Porém, antes que a requisição execute a lógica necessária, é feita uma validação de sessão, onde o usuário logado é validado perante o servidor.

Caso o usuário esteja logado corretamente, o fluxo segue normalmente, podendo ser feitas as chamadas nos *controllers*, *services* e *models*. Porém, se existir algum problema de autenticação, o usuário é automaticamente redirecionado para a página de *login* da aplicação web. Desse modo, a Figura 11 demonstra a modelagem do *Back-end* da aplicação web.

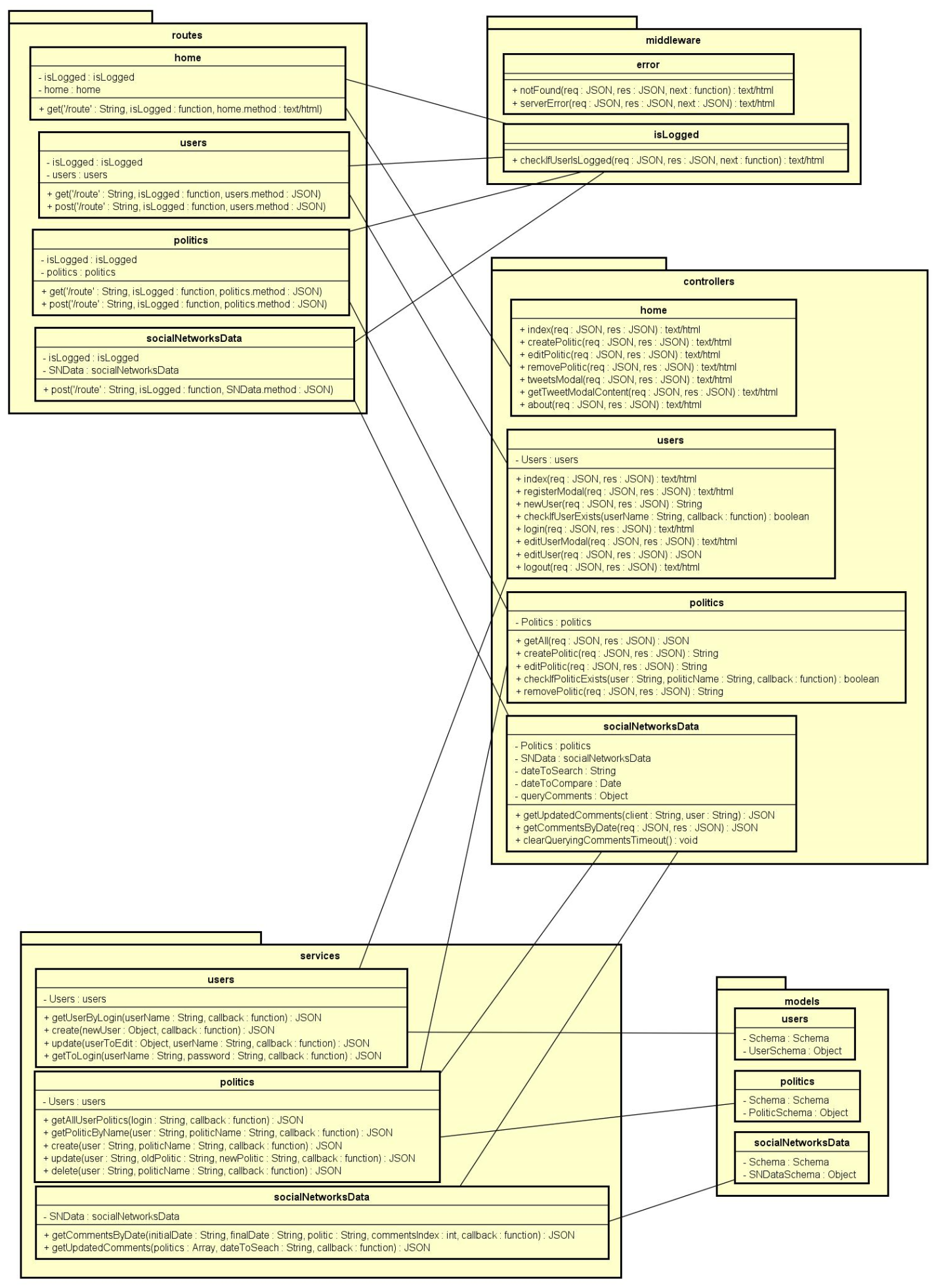


Figura 11 - Modelagem do *Back-end*. Fonte: Elaborado pelos autores.

Porém, para fazer com que o *back-end* atenda a todas as requisições feitas pelo usuário, foi necessário fazer algumas configurações para que a sua lógica funcionasse de maneira correta. A primeira delas é a configuração da *stack*.

A *stack* é correspondente ao arquivo app.js, arquivo este que concentra todas as configurações de inicialização do sistema, tais como conexão com o banco de dados, configuração de diretórios, inicialização do servidor, dentre outras. As figuras a seguir formarão a sequência de código desenvolvida para a aplicação web.

As primeiras configurações foram as chamadas dos módulos necessários da *stack*, conforme demonstrado no Código 3.

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)
8. , MemoryStore = require('connect/middleware/session/memory')
9. , session\_store = new MemoryStore()
10. , connect = require('connect');

Código 3 - Chamada dos módulos na *stack*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1** - O Módulo Express é carregado na variável express. Desse modo, seus recursos, tal como a criação do servidor, já podem ser usados através da variável;

**Linha 2** - A função express() é executada e posteriormente atribuída à variável app, possibilitando assim a criação das configurações básicas para criação de um servidor Node.JS;

**Linha 3** - Carregado o módulo express-load, o qual permite ser feito o carregamento dos diretórios da aplicação;

**Linha 4** - Carregado o módulo middleware, responsável pelas permissões da aplicação e tratamentos de erros HTTP;

**Linha 5** - Carregado o módulo Mongoose, atribuindo-o à variável mongoose. Desse modo, através da variável mongoose pode ser feita a criação de *schemas* para a persistência de dados no banco;

**Linha 6** - Carregamento do módulo HTTP, posteriormente chamando a sua função createServer e passando como parâmetro a variável de configurações do Express, criando assim o servidor HTTP;

**Linha 7** - Carregamento do módulo socket.io, posteriormente chamando a sua função listen e passando como parâmetro o servidor HTTP recém-criado. O módulo socket.iofoi adicionado à aplicação devido a necessidade de existência de uma conexão *socket* da mesma, para ser feito envio dos *tweets* para o *front-end* em tempo real;

**Linha 8** - Carregado o módulo MemoryStore, responsável por salvar a sessão do usuário no servidor;

**Linha 9** - Instanciado o módulo MemoryStore através da variável session\_store. Desse modo, fica disponível para que ao ser efetuado um *login* na aplicação, o objeto session\_store guarde as credenciais de acesso;

**Linha 10** - Carregado o módulo connect, responsável por fazer a manipulação de todas as solicitações da aplicação.

Posteriormente, foi feita a conexão com o banco de dados, demonstrada no Código 4.

1. global.db = mongoose.connect('mongodb://localhost/database',function(error){
2. if (error) {
3. console.log(error);
4. } else {
5. console.log("Conectado ao Banco de Dados!");
6. }
7. });

Código 4 - Conexão com o Banco de dados. Fonte: Elaborado pelos autores.

Detalhamento de cada linha:

**Linha 12** - Declaração da variável global db. Ela foi declarada como global para poder estar disponível em toda aplicação. Posteriormente, foi chamado o método connectda variável mongoose, passando como parâmetro a URL de conexão com o MongoDB;

**Linhas 13 à 17** - Verificação da *callback* de conexão. Se a conexão for estabelecida, uma mensagem de sucesso é exibida no console. Caso contrário, será exibida uma mensagem de erro.

A configuração seguinte foi a de diretórios, mostrada no Código 5.

1. app.set('views', \_\_dirname + '/views');
2. app.set('view engine', 'ejs');

Código 5 - Configuração de diretórios. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 20** - Setado o *path* padrão das *views* da aplicação. A constante \_\_dirname indica o nome do diretório base, e a *string* ‘/views’ indica o diretório das telas;

**Linha 21** - Setado o *template engine* EJS, para que a *view* seja montada por completo antes de ser retornada ao navegador.

Após as configurações de diretório, foi necessário fazer as configurações básicas para criação do servidor, conforme o Código 6.

1. app.use(express.cookieParser('politistatus'));
2. app.use(express.session({store: session\_store}));
3. app.use(express.json());
4. app.use(express.urlencoded());
5. app.use(app.router);

Código 6 - Configurações da variável app. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 23** - Atribuído o uso dos *cookies* com o nome de ‘politistatus’;

**Linha 24** - Atribuído o recurso de sessão de usuário, permitindo assim o controle de acesso da aplicação;

**Linha 25** - Atribuído o tráfego JSON da aplicação;

**Linha 26** - Atribuído o recurso *urlencoded*, o qual faz a conversão de caracteres não alfa-numéricos em ‘%’ seguido de 2 dígitos na base hexadecimal;

**Linha 27** - Atribuído o recurso que define como padrão o redirecionamento de todas as requisições do navegador ao serviço de rotas.

Posteriormente, foi necessário criar o conteúdo estático da aplicação, bem como suas páginas de erro, conforme o Código 7.

1. app.use(express.static(\_\_dirname + '/public'));
2. app.use(error.notFound);
3. app.use(error.serverError);

Código 7 - Configurações estáticas e de tratamento de erro da aplicação. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 29** - Atribuído o diretório estático da aplicação, onde estão os JavaScripts, *stylesheets* e imagens da aplicação (arquivos do *front-end*);

**Linha 30** - Atribuído o tratamento do erro de página não encontrada, o qual consiste na apresentação de uma página amigável para o usuário;

**Linha 31** - Atribuído o tratamento do erro de falha do servidor, o qual consiste na apresentação de uma página amigável para o usuário.

A próxima configuração necessária era a de carregamento dos diretórios da aplicação em si: *routes*, *controllers*, *services*, e *models*, conforme demonstrado no Código 8.

1. load('models')
2. .then('services')
3. .then('controllers')
4. .then('routes')
5. .into(app);

Código 8 - *Load* de diretórios da aplicação. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 33 a 37** - Chamada do método load() do Express, passando como parâmetro os diretórios da aplicação. Desse modo, as funções contidas em cada diretório ficam disponíveis na variável app (linha 37).

Dentre as configurações mais importantes da *stack*, está a inicialização do servidor, demonstrado no Código 9.

1. server.listen(3000, function(){
2. console.log('Politistatus Online!');
3. });

Código 9 - Inicialização do servidor na porta 3000. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 39 a 41** - O servidor é iniciado na porta 3000, através do método listen(). Também é passada como parâmetro uma *callback*, a qual mostra no terminal quando a aplicação está disponível para conexão.

Depois de feitas todas as configurações anteriores, o servidor estava pronto para funcionar. Porém o sistema necessitava também de uma configuração completa de conexão *socket* para realizar a interação entre os *tweets* minerados pelo *framework* Apache Storm e o usuário da aplicação web. Essa configuração foi feita após a inicialização do servidor, conforme mostra o Código 10.

1. var SNData = app.controllers.socialNetworksData;
2. io.on('connection', function (client) {
3. var cookie\_string = client.request.headers.cookie;
4. var parsed\_cookies = connect.utils.parseCookie(cookie\_string);
5. var connect\_sid = parsed\_cookies['connect.sid'];
6. var sid = connect\_sid.substr(2, connect\_sid.indexOf(".") - 2);
7. var sessionUserName;
8. if (connect\_sid) {
9. session\_store.get(sid, function (error, session) {
10. console.log('O Cliente de foi conectado!');
11. sessionUserName = session.user.userName;
12. SNData.getUpdatedComments(client, sessionUserName);
13. });
14. }
15. client.on('pause', function() {
16. SNData.pauseCommentsQuery();
17. });
18. client.on('resume', function() {
19. SNData.resumeCommentsQuery(client, sessionUserName);
20. });
21. client.on('disconnect', function() {
22. SNData.pauseCommentsQuery();
23. console.log('O Cliente foi desconectado!');
24. });
25. });

Código 10 - Configuração da conexão *socket*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 43** - É criada a variável SNData, que recebe como valor o *Controller* de busca de *tweets* da aplicação;

**Linhas 46 a 50** - Nestas linhas são feitos os *parsers* necessários para se obter as informações de *login* do usuário. Essas informações são de extrema importância para a busca dos *tweets*, já que a mesma é feita com base no usuário que está logado e as palavras-chave que ele cadastrou;

**Linhas 52 a 58** - É feita uma verificação para validar se o usuário que requisitou o *socket* corresponde ao usuário da sessão do servidor. Caso o usuário esteja validado, é mostrada uma mensagem de sucesso no terminal e posteriormente é chamado o método getUpdatedCommentsdo *Controller* de busca dos *tweets*, sendo passados como parâmetro a conexão *socket*  e o nome do usuário da sessão;

**Linhas 60 a 62** - Caso a conexão socket receba um evento de *pause*, a função pauseCommentsQuery será chamada. Essa função pausa o envio de *tweets* em tempo real para o usuário;

**Linhas 64 a 66** - Caso a conexão socket receba um evento de *resume*, a função resumeCommentsQuery será chamada. Essa função retoma o envio de *tweets* em tempo real para o usuário, caso o mesmo esteja pausado;

**Linhas 68 a 71** - Caso a conexão socket receba um evento de *disconnect*, a função pauseCommentsQuery será chamada. A conexão *socket* só ira receber um evento de *disconnect* caso o usuário faça *logoff* da aplicação web.

Após todas estas configurações na *stack*, o servidor está pronto para atender os requisitos da aplicação web. A próxima ação a ser tomada logo após o desenvolvimento da *stack*, foi o desenvolvimento do *middleware*. O *middleware* é o responsável pela autenticação do usuário e pelo tratamento de erros da aplicação. Toda requisição recém-chegada a uma rota, obrigatoriamente passa pela validação de usuário do *middleware*. O Código 11 demonstra o *middleware* de autenticação de usuário.

1. module.exports = function(req, res, next) {
2. if (req.session.user)
3. return next();
4. return res.redirect('/');
5. };

Código 11 - Autenticação de usuário via *middleware*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 1 até 6** - É criada uma função de uso global que valida se o usuário está com sessão aberta. Caso esteja com a sessão aberta, dá-se continuidade ao fluxo da aplicação, caso contrário a página de *login* (*default*) da aplicação é retornada.

O Código 12 demonstra o *middleware* de tratamento de páginas de erro.

1. exports.notFound = function(req, res, next) {
2. res.status(404);
3. res.render('error/not-found');
4. };
5. exports.serverError = function(error, req, res, next) {
6. res.status(500);
7. res.render('error/server-error', {error: error});
8. };

Código 12 - Tratamento de erros via *middleware*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 1 até 4** - Criada uma função de uso global renderiza a *view* de página não encontrada caso o *status* da requisição HTTP seja 404;

**Linhas 6 até 9** - Criada uma função de uso global renderiza a *view* de erro do servidor caso o *status* da requisição HTTP seja 500, passando como parâmetro a mensagem de erro do servidor.

Após criadas as validações de requisição via *middleware*, a próxima ação a ser tomada foi o desenvolvimento das rotas. As rotas têm a função de fazer o redirecionamento das requisições ao servidor, enviando suas informações para o seu respectivo *Controller*.

O sistema de rotas consiste em métodos que recebem como argumento a variável app, a qual contém o sistema de rotas ativado previamente configurado na *stack*. Nestes métodos são feitos os tratamentos e redirecionamento de chamadas de cada requisição.

No Código 13 é demonstrado um exemplo de arquivo contendo suas respectivas rotas.

1. module.exports = function(app)
2. {
3. var isLogged = require('./../middleware/isLogged')
4. , politics = app.controllers.politics;
5. app.get('/getAll', isLogged, politics.getAll);
6. app.post('/createPolitic', isLogged, politics.createPolitic);
7. app.post('/editPolitic', isLogged, politics.editPolitic);
8. app.post('/removePolitic', isLogged, politics.removePolitic);
9. };

Código 13 - Exemplo de um arquivo de rotas. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1 -** É criado o módulo recebendo como parâmetro a variável app;

**Linha 3** - Requisição do *middleware* para a autenticação de usuário logado;

**Linha 4 -** Carregamento do *Controller* na variável politics;

**Linhas 6 até 9** - Rotas que inicialmente passam pelo *middleware* para fazer a autenticação de usuário, e posteriormente fazem o redirecionamento na chamada de sua rota para o método do seu respectivo *Controller*.

Com as rotas da aplicação criadas, o próximo passo era a criação dos *Controllers*. Os *Controllers* são responsáveis por dar o suporte às requisições do *front-end*. Nos *Controllers* são feitas as chamadas para os métodos dos *Services*, os quais retornam a informação desejada através da persistência de dados.

Logo após ser chamado pela sua respectiva rota, o método do *Controller* recebe como parâmetros os objetos req e res, que contém respectivamente os dados da requisição e os dados para o retorno da chamada.

Com base no que foi mencionado anteriormente, o Código 14 demonstra um trecho de código de um *Controller* da aplicação.

1. module.exports = function(app)
2. {
3. var Politics = app.services.politics;
4. var PoliticsController =
5. {
6. getAll: function(req, res)
7. {
8. try
9. {
10. var user = req.session.user.userName;
11. Politics.getAllUserPolitics(user, function (politics)
12. {
13. if (politics != null) {
14. var array = {
15. data : []
16. };
17. politics.forEach(function (item) {
18. item.user\_keywords.forEach(function (keyword){
19. array.data.push({
20. "politic\_name" : keyword
21. });
22. });
23. });
24. res.end(JSON.stringify(array));
25. }
26. });
27. }
28. catch(ex)
29. {
30. console.log(ex.message);
31. res.status(500).send("Problemas ao buscar todos os políticos!");
32. }
33. }
34. };
35. return PoliticsController;
36. };

Código 14 - Trecho de código exemplificando um *Controller*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1 -** É criado o módulo recebendo como parâmetro a variável app;

**Linha 3 -** Declarada a variável Politics, recebendo como valor, o *Service* de Politics;

**Linha 5 -** Declarada a classe PoliticsController;

**Linha 8 -** Criação do método responsável por buscar todas as palavras-chave (nome dos políticos) cadastradas pelo usuário;

**Linha 12 -** Declarada a variável user, a qual recebe o *login* do usuário como valor;

**Linha 14 -** Chamada do método getAllUserPolitics do *Service*, passando como parâmetro o *login* do usuário cadastrado na sessão. Esse método irá retornar a lista de palavras-chave cadastradas pelo usuário;

**Linhas 16 à 28 -** Operações lógicas para a montagem de um objeto JSON para ser mandado para o *front-end*. Essas operações foram necessárias para que o objeto a ser retornado seguisse um padrão específico, pois ele seria consumindo no *front-end* por uma DataTables;

**Linha 30 -** O objeto res é chamado passando como parâmetro o objeto JSON previamente montado convertido em *String*. O objeto res irá se encarregar de fazer o retorno da requisição do *front-end*;

**Linhas 37 e 38 -** Caso a busca no *Service* lance alguma *exception*, o erro é mostrado no terminal e posteriormente é mandado um responsecom *status* HTTP 500, juntamente com uma mensagem amigável, para o *front-end*;

**Linha 44 -** Retorno da classe PoliticsController.

Após concluída a implementação dos *Controllers*, era necessário ser feita a implementação dos *Services*. Os *Services* são responsáveis por realizar as operações relacionadas ao banco de dados, tais como *create*, *retrieve*, *update* e *delete*. Logo após a execução dessas operações, os *Services* retornam as informações aos *Controllers*, conforme exemplificado no Código 15.

1. module.exports = function(app)
2. {
3. var Users = app.models.users;
4. var PoliticsService =
5. {
6. getAllUserPolitics: function(login, callback)
7. {
8. var query = { userName : login };
9. Users.find(query)
10. .select('user\_keywords').exec(function(error, politics)
11. {
12. if (error)
13. throw error;
14. if (politics)
15. callback(politics);
16. callback(null);
17. });
18. }
19. };
20. return PoliticsService;
21. };

Código 15 - Trecho de código exemplificando um *Service* e sua consulta no banco. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1 -** Criado o módulo recebendo como parâmetro a variável app;

**Linha 4 -** Declarada a variável Users, recebendo como valor o *Model* de Users;

**Linha 6 -** Declarada a classe PoliticsService;

**Linha 8 -** Criação do método que irá ser chamado no *Controller* para efetuar a busca dos dados. São recebidos como parâmetro respectivamente o *login* do usuário e uma *callback*;

**Linha 10 -** Criada a variável query, a qual irá conter as condições de busca no banco de dados;

**Linhas 12 a 14 -** Consulta no banco de dados através do método find() do *Model*, sendo passado como parâmetro a variável query. O método select() é uma restrição, informando que somente deverá ser retornado do banco de dados o campo users\_keywords;

**Linhas 16 a 22 -** Verificações da *callback* da consulta. Existem 2 possibilidades de retorno ao *Controller*, sendo uma *exception* (caso ocorra um erro na consulta), os dados da consulta (caso a mesma tenha sido realizada com sucesso) e null(caso a consulta não retorne valores) ;

**Linha 27 -** Retorno da classe PoliticsService.

Com as etapas de criação de Rotas, *Controllers*, e *Services* prontas, era necessário desenvolver os *Models*. Os *Models* são arquivos onde são feitos os *Schemas*, necessários para persistência dos objetos no banco de dados. Seguindo este contexto, o Código 16 exemplifica como foi o padrão de desenvolvimento dos *Models* da aplicação.

1. module.exports = function(app)
2. {
3. var Schema = require('mongoose').Schema;
4. var user = Schema({
5. firstName: { type: String, required: true }
6. , lastName: { type: String, required: true }
7. , userName: { type: String, required: true }
8. , passwd: { type: String, required: true }
9. , user\_keywords: []
10. });
11. return db.model('users', user);
12. };

Código 16 - Trecho de código exemplificando um *Schema* de um *Model*. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1 -** Criado o módulo recebendo como parâmetro a variável app;

**Linha 3 -** Declarada a variável Schema, que recebe o objeto *Schema* do Mongoose. Desse modo, essa variável fica responsável por gerar os modelos das operações de CRUD;

**Linhas 5 a 11 -** Geração de um modelo. É este objeto que será usado como modelo na persistência dos dados;

**Linha 13 -** Retorno dos dados da *collection* users do banco através da variável user.

Após feitas todas as etapas de desenvolvimento descritas, toda requisição irá chegar no servidor, sendo posteriormente validada, tratada, buscada e terá o seu retorno. As operações de *Create*, *Retrieve*, *Update* e *Delete*, bem como a conexão *Socket* estarão completas.

### *Front-end*

Paralelamente ao desenvolvimento do servidor, o *front-end* da aplicação web estava sendo desenvolvido. Era necessário existir o *client-side* da aplicação para serem feitas as requisições no servidor, posteriormente demonstrar os dados retornados, tais como mensagens de sucesso e erro e os próprios dados do banco. O *front-end* corresponde à parte de telas da aplicação, as quais estão no diretório *Views*. No desenvolvimento das *Views* foram utilizadas as tecnologias HTML5 em conjunto com o *template engine* EJS, CSS3 e JavaScript.

Para evitar a reescrita de código, foram criados os *templates* padrão para serem usados em telas que têm cabeçalho e rodapé iguais. Esses *templates* são divididos em *Header* e *Footer*, e são importados por telas em que seu uso é comum. A Figura 12 demonstra a árvore de diretórios das *Views*.

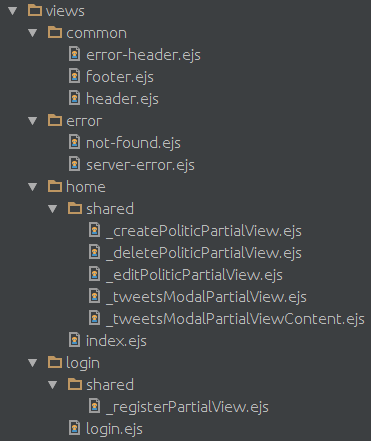


Figura 12 - Árvore de diretórios das *Views*. Fonte: Elaborado pelos autores.

No diretório common da estrutura, estão os arquivos comuns entre as *Views* da aplicação, os quais contém as chamadas dos JavaScripts e dos *Stylesheets*.

O diretório error contém todas as páginas de erro da aplicação. Elas são chamadas através do *middleware* do servidor.

Os demais diretórios são referentes todos os aos módulos das *Views* da aplicação web. Essas *Views* primeiramente importam o seu cabeçalho. Posteriormente é feito seu conteúdo exclusivo. Por último, importa-se o rodapé. Esse padrão é demonstrado no Código 17, o qual teve seu conteúdo reduzido devido ao tamanho do arquivo.

1. <% include ../common/header %>
2. <div class="presentation">
3. <div id="userPanel">
4. Bem vindo <label><%= firstName %> <%= lastName %>!</label>
5. <div class="row">

.

.

.

1. </div>
2. </body>
3. <% include ../common/footer %>
4. </html>

Código 17 - Código parcial da *View* da página inicial. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1 -** Importado o *Header*, através do include. O comando include é disponível graças à EJS *tamplate engine*;

**Linhas 2 a 97 -** Conteúdo da página HTML.

**Linha 98 -** Importado o *Footer*, do mesmo modo que o *Header*;

A Figura 13 demonstra como é, visualmente, a *View* que corresponde ao Código 17.



Figura 13 - Página inicial. Fonte: Elaborado pelos autores.

O *Header* da aplicação carrega os JavaScripts e *stylesheets* para a página HTML. O Código 18 exemplifica a estrutura do *Header*. O Código 18 teve seu conteúdo reduzido devido ao tamanho do arquivo.

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <meta charset="utf-8">
5. <meta http-equiv="X-UA-Compatible" content="IE=edge">
6. <meta name="viewport" content="width=device-width, initial-scale=1">
7. <title>Politistatus</title>
8. <link href="/stylesheets/twitter.button.min.css" rel="stylesheet">

.

.

.

1. <link href="/externals/toastr/toastr.css" rel="stylesheet"/>
2. <script type="text/javascript" src="common/socket.io.js"></script>

.

.

.

1. <script type="text/javascript" src="home/script.js"></script>
2. <link rel="shortcut icon" href="../images/hammer.ico">
3. </head>
4. <body>
5. <nav class="navbar navbar-default navbar-fixed-top">
6. <div class="container-fluid">
7. <div class="navbar-header"><a></i>Politistatus</a></div>
8. <div class="navbar-collapse" id="nav-collapse-1">
9. <div id="bar-right" class="navbar-right">
10. <ul class="nav">
11. <a class="navbar-brand" href="/about">Sobre</a>
12. </ul>
13. </div>
14. </div>
15. </div>
16. </nav>

Código 18 - Código parcial exemplificando um *Header*. Fonte: Elaborado pelos autores.

Também dentro dessa estrutura de diretórios, estão as subpastas shared. Dentro destas subpastas estão os arquivos referentes às *Partial Views* da aplicação. As *Partial Views* são as estruturas HTML que posteriormente serão convertidas nos modais da aplicação web, via JavaScript. O Código 19 exemplifica a estrutura de uma *Partial View*.

1. <div id="registerModal" class="modal fade">
2. <div class="modal-dialog">
3. <div class="modal-content">
4. <div class="modal-header">
5. <button class="close" data-dismiss="modal">times;</button>
6. <h4 class="modal-title">Cadastro</h4>
7. </div>
8. <div class="modal-body">
9. <form id="newUserForm">
10. <div class="form-group">
11. <label>Nome:</label>
12. <input type="text" id="firstName class="form-control">
13. </div>
14. <div class="form-group">
15. <label>Sobrenome:</label>
16. <input type="text" id="lastName" class="form-control">
17. </div>
18. <div class="form-group">
19. <label>Usuário:</label>
20. <input type="text" id="userName" class="form-control">
21. </div>
22. <div class="form-group">
23. <label>Senha:</label>
24. <input type="password" id="passwd" class="form-control">
25. </div>
26. <div class="form-group">
27. <label>Digite a senha novemente:</label>
28. <input type="password" id="retype" class="form-control">
29. </div>
30. <br />
31. <div class="modal-footer">
32. <div class="form-group">
33. <button type="submit" class="btn">Salvar</button>
34. <a class="btn" data-dismiss="modal">Cancelar</a>
35. </div>
36. </div>
37. </form>
38. </div>
39. </div>
40. </div>
41. </div>
42. <script type="text/javascript" src="/javascripts/login/register.js"></script>

Código 19 - Exemplo de uma *Partial View*. Fonte: Elaborado pelos autores.

A Figura 14 demonstra como é, visualmente, a *Partial View* exemplificada no Código 19.



Figura 14 - *Partial View* de cadastro de usuário. Fonte: Elaborado pelos autores.

Na *View* principal da aplicação, ou seja, na página inicial, está o mapa que demonstra os locais do Brasil de onde estão ocorrendo os *tweets*. Inicialmente, é feito o *import* do JavaScript do Google Maps API. Desse modo, fica disponível para a aplicação o consumo dos serviços oferecidos pela API. O Código 20 ilustra o *import* necessário.

<script async defer src="https://maps.googleapis.com/maps/api/js?key=chave\_acesso &callback=initMap"></script>

Código 20 - Import do JavaScript do Google Maps API. Fonte: Elaborado pelos autores.

Logo após o *import*, fica a critério do desenvolvedor manipular o mapa de acordo com sua necessidade. O controlador do mapa é demonstrado em 5 partes, por meio dos Códigos 21, 22, 23, 24 e 25, demonstrando assim as configurações e a lógica do mapa na página inicial da aplicação.

1. var MapScript = (function () {
2. var map;
3. var marker;
4. var markers = [];
5. initMap = function () {
6. var resolution = $(window).width();
7. if (resolution >= 952){
8. $("#mapDiv").width(700);
9. map = new google.maps.Map($('#mapDiv')[0], {
10. center: {lat: -14.718, lng: -56.449},
11. zoom: 4
12. });
13. }else{
14. $("#mapDiv").width(500);
15. map = new google.maps.Map($('#mapDiv')[0], {
16. center: {lat: -15.000, lng: -54.449},
17. zoom: 4
18. });
19. }
20. };

Código 21 - JavaScript controlador do mapa da aplicação (Parte 1). Fonte: Elaborado pelos autores.

1. getLocationByAddress = function (address, keyword) {
2. if (address != "") {
3. var data = { address: address };
4. $.ajax({
5. url: 'https://maps.googleapis.com/maps/api/geocode/json',
6. data: data,
7. success: function (result) {
8. setMaker(result, keyword);
9. }
10. });
11. }
12. };

Código 22 - JavaScript controlador do mapa da aplicação (Parte 2). Fonte: Elaborado pelos autores.

1. setMaker = function (placeObject, keyword) {
2. if (placeObject.results.length > 0) {
3. var latitude = placeObject.results[0].geometry.location.lat;
4. var longitude = placeObject.results[0].geometry.location.lng;
5. var location = placeObject.results[0].address\_components[0].long\_name;
6. marker = new google.maps.Marker({
7. position: {lat: latitude, lng: longitude},
8. map: map,
9. title: location + ". Palavra-chave: " + keyword,
10. animation: google.maps.Animation.DROP
11. });
12. markers.push(marker);
13. }
14. };
15. clearMarkers = function () {
16. for (var i = 0; i < markers.length; i++) {
17. markers[i].setMap(null);
18. }
19. };

Código 23 - JavaScript controlador do mapa da aplicação (Parte 3). Fonte: Elaborado pelos autores.

1. handleMapAccordingScreenResolution = function () {
2. var resolution = $(window).width();
3. if (resolution >= 952){
4. $("#mapDiv").width(700);
5. var newLatLng = new google.maps.LatLng(-14.718, -56.449);
6. map.setCenter(newLatLng);
7. }else{
8. $("#mapDiv").width(500);
9. var newLatLng = new google.maps.LatLng(-15.000, -54.449);
10. map.setCenter(newLatLng);
11. }
12. };

Código 24 - JavaScript controlador do mapa da aplicação (Parte 4). Fonte: Elaborado pelos autores.

1. setMapEvents = function () {
2. $(window).resize(function () {
3. handleMapAccordingScreenResolution();
4. });
5. $("#clearMakers").attr("onclick", "clearMarkers()");
6. };
7. //Public Methods
8. return {
9. initMap: initMap,
10. setMapEvents: setMapEvents
11. };
12. })();
13. $(document).ready(function () {
14. MapScript.setMapEvents();
15. });

Código 25 - JavaScript controlador do mapa da aplicação (Parte 5). Fonte: Elaborado pelos autores.

Neste trabalho, o mapa tem a finalidade de demonstrar a localização em que estão sendo feitos os *tweets* através de um marcador. Também deverá ser informado para o usuário o nome da cidade e o político que foi mencionado naquele local.

Com base neste objetivo, a função initMap fica responsável por iniciar o mapa com a latitude/longitude e zoom corretos para ser feita a demonstração do território brasileiro. Essa função é chamada quando é carregada a página inicial da aplicação web, pois o mapa tem que ficar acessível ao usuário quando o mesmo faz o *login*.

A função getLocationByAddress é chamada quando um novo *tweet* é recebido do *back-end* via *socket*. Essa função fica responsável por enviar a cidade cadastrada no *tweet* para a API do Google Maps. Se a cidade existir, é retornado um objeto JSON contendo as informações da cidade correspondente, tais como nome oficial, latitude e longitude. Seguindo o fluxo, esse objeto, juntamente com o nome do político, é enviado para a função setMaker, onde enfim ocorrerá a marcação no mapa através da latitude, longitude, nome da cidade e o político mencionado. Desse modo, o usuário consegue distinguir no mapa de quais cidades estão vindo os *tweets*, bem como o político que foi mencionado naquela cidade.

As funções clearMakers e handleMapAccordingScreenResolution têm, respectivamente, as finalidades de limpar os marcadores do mapa, e o tornar responsivo. A função handleMapAccordingScreenResolution foi feita pelo fato do mesmo poder estar acessível a qualquer tipo de resolução, já que ele não é responsivo por *default*.

Juntamente com o mapa da aplicação, está um gráfico onde é demonstrada a quantidade de menções feitas no Twitter por palavra-chave (político) cadastrada. Desse modo, torna-se possível para o usuário a visualização de qual político está sendo mais mencionado, em tempo real. O gráfico será visualizado se o usuário clicar no botão “Quantidade de Menções”, que está localizado na parte superior do mapa.

O desenvolvimento do gráfico foi feito por meio da biblioteca Highcharts, uma biblioteca JavaScript específica para a criação de gráficos para aplicações web. Como o arquivo controlador é significativamente extenso, sua demonstração foi dividida em quatro partes. Desse modo, os Códigos 26, 27, 28 e 29 demonstram as funções responsáveis pela criação e manipulação do gráfico de menções da aplicação web.

1. setChartConfigurations = function () {
2. chart = new Highcharts.Chart({
3. chart: {
4. renderTo: 'politicChart',
5. plotBackgroundColor: null,
6. plotBorderWidth: null,
7. plotShadow: false,
8. type: 'pie'
9. },
10. title: {
11. text: 'Quantidade de menções no Twitter por Político - ' + getDate()
12. },
13. plotOptions: {
14. series: {
15. dataLabels: {
16. enabled: true,
17. format: '{point.name}: {point.y}'
18. },
19. showInLegend: true
20. },
21. },
22. tooltip: {
23. headerFormat: '<span>{series.name}</span><br>',
24. pointFormat: '<span>{point.name}</span>: ' +
25. '<b>{point.y}</b> do total<br/>'
26. },
27. series: [{
28. name: "Qtd. Menções",
29. colorByPoint: true,
30. }]
31. });
32. };

Código 26 - Funções de manipulação do gráfico (Parte 1). Fonte: Elaborado pelos autores.

1. getDate = function () {
2. var date = new Date();
3. var day = date.getDate();
4. if (day.toString().length == 1)
5. day = "0" + day;
6. var month = date.getMonth() + 1;
7. if (month.toString().length == 1)
8. month = "0" + month;
9. var year = date.getFullYear();
10. return day + "/" + month + "/" + year
11. };
12. getPoliticsMentionCountObject = function(callback){
13. callback(politicsMentionCountObject.data);
14. };

Código 27 - Funções de manipulação do gráfico (Parte 2). Fonte: Elaborado pelos autores.

1. updatePoliticsCounter = function (politic) {
2. if (politicsMentionCountObject.data.length > 0) {
3. if (checkIfPoliticExistsInObject(politicsMentionCountObject.data, politic)) {
4. $(politicsMentionCountObject.data).each(function () {
5. if (this.name == politic) {
6. this.y = this.y + 1;
7. }
8. });
9. } else {
10. politicsMentionCountObject.data.push({name: politic, y: 1});
11. }
12. } else {
13. politicsMentionCountObject.data.push({name: politic, y: 1});
14. }
15. };

Código 28 - Funções de manipulação do gráfico (Parte 3). Fonte: Elaborado pelos autores.

1. checkIfPoliticExistsInObject = function (politicsObject, politic) {
2. for (var i = 0; i < politicsObject.length; i++) {
3. if (politicsObject[i].name == politic)
4. return true;
5. }
6. };
7. setChartValues = function () {
8. if (politicsMentionCountObject.data.length > 0) {
9. chart.series[0].setData(politicsMentionCountObject.data, true);
10. }
11. };

Código 29 - Funções de manipulação do gráfico (Parte 4). Fonte: Elaborado pelos autores.

O detalhamento de cada função é feito abaixo:

A função setChartConfigurations fica responsável por criar o gráfico, bem como setar os seus atributos. São setadas informações como: título, tipo, opções de plotagem e *tooltips*.

A função getDate retorna a data formatada para ser feita a demonstração do dia em questão. Isso se torna relevante para a aplicação web, já que a implementação da biblioteca Highcharts possibilita que o mesmo possa ser impresso.

A função getPoliticsMentionCountObject é executada somente quando o botão “Quantidade de Menções” for pressionado. O controle de menções por palavra-chave é feito por meio do objeto politicsMentionCountObject. Quando o botão é pressionado, esta função retorna este objeto para o JavaScript controlador da página inicial da aplicação web. Se o objeto estiver vazio, uma mensagem informa para o usuário que não existem informações no gráfico. Caso contrário, a modal Bootstrap responsável pela demonstração do gráfico é chamada.

A função updatePoliticsCounter é chamada quando um novo *tweet* é recebido via *socket*. Ela recebe como parâmetro a palavra-chave do respectivo *tweet*, ou seja, o político cadastrado pelo usuário. Ela é responsável por checar se a palavra-chave já está sendo demonstrada no gráfico.

Para tal checagem, é feita uma chamada na função checkIfPoliticExistsInObject, que irá verificar se o objeto politicsMentionCountObject contém a palavra-chave em questão. Se ele contiver, o respectivo contador daquela palavra-chave é incrementado, atualizando assim a sua quantidade de menções. Caso contrário, a palavra-chave é adicionada no objeto, e seu contador recebe o valor 1, indicando assim que esta palavra-chave foi mencionada pela primeira vez.

Já a função setChartValues é chamada quando a modal de demonstração do gráfico é chamada. Ela tem a finalidade colocar os valores do objeto politicsMentionCountObject no gráfico, para que assim, finalmente, ser feita a demonstração das estatísticas de menção por político para o usuário da aplicação web.

### *Framework* de *Data Mining* e *Big Data*

Após serem desenvolvidas todas as estruturas, tanto de *front-end* quanto de *back-end*, a aplicação web estava pronta para suprir as necessidades do usuário. O próximo passo era desenvolver o *framework* Apache Storm, que é responsável por fazer a mineração dos dados e gerar o *Big Data*.

O *framework* Apache Storm constitui a parte responsável por buscar e armazenar os dados referentes às postagens na rede social Twitter, e é dividido em cinco classes, conforme mostrado na Figura 15.

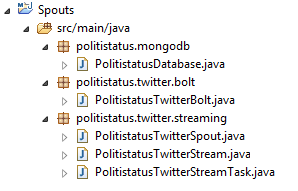


Figura 15 - Classes do *framework* Apache Storm. Fonte: Elaborado pelos autores.

A Figura 16 demonstra o diagrama de classes completo do sistema de mineração e armazenamento de dados desta pesquisa.

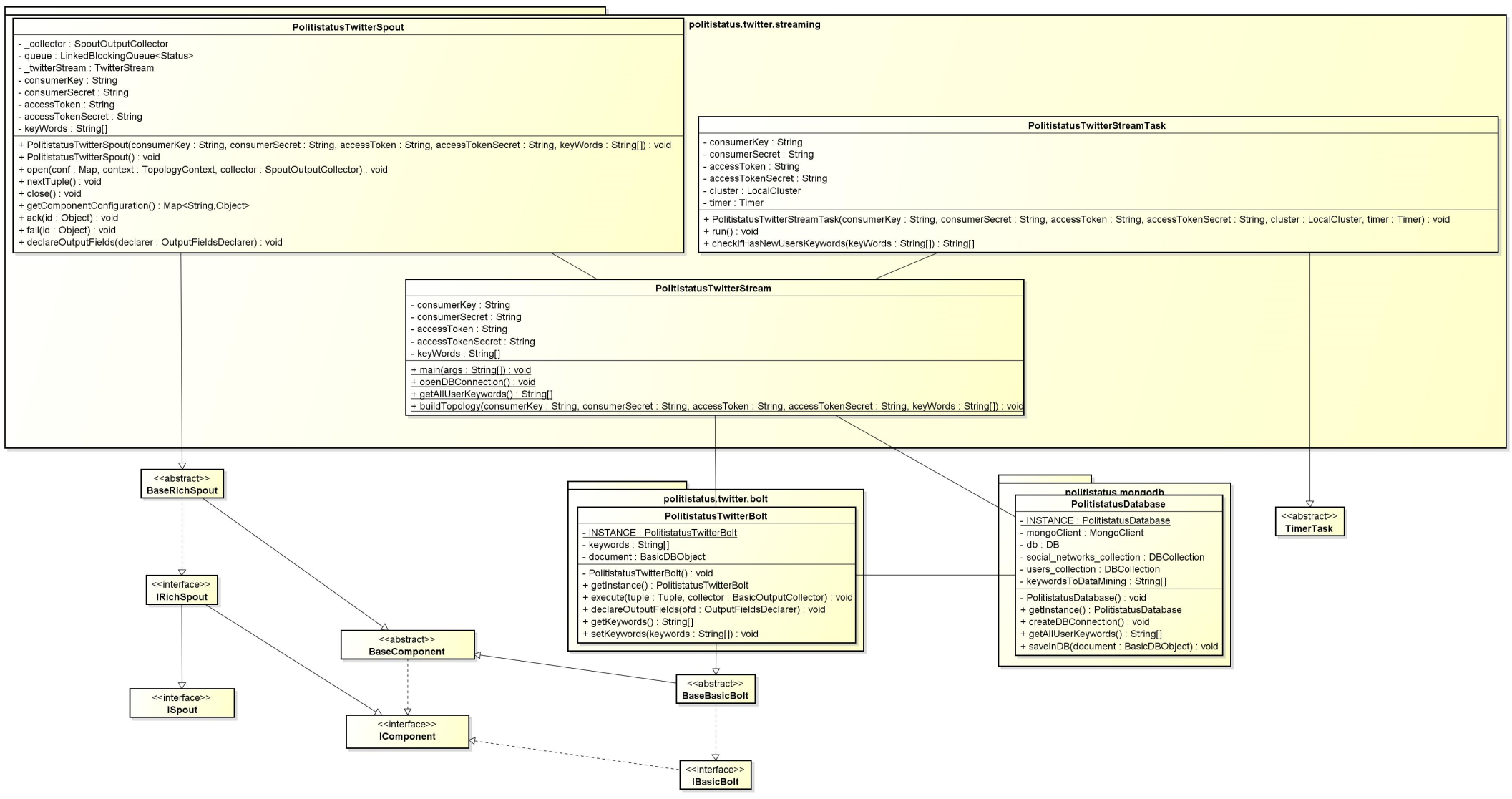


Figura 16 - Diagrama de classes do *framework* Apache Storm. Fonte: Elaborado pelos autores.

A primeira ação a ser tomada em relação ao desenvolvimento do *framework*, era o desenvolvimento de sua classe de conexão com o MongoDB. Essa classe do *framework* é responsável por fazer as operações relacionadas ao banco de dados, dentre as quais pode ser citada a operação de armazenamento dos dados, que posteriormente irá gerar o *Big Data* desta pesquisa. O método demonstrado no Código 30 demonstra a criação da conexão com o banco de dados.

1. public void createDBConnection() throws UnknownHostException{
2. mongoClient = new MongoClient("localhost");
3. db = mongoClient.getDB("database");
4. social\_networks\_collection = db.getCollection("social\_networks\_datas");
5. users\_collection = db.getCollection("users");
6. }

Código 30 - Criação da conexão com o MongoDB. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 34** - É informado o endereço da máquina onde o serviço de banco de dados está instalado, no caso a própria máquina é o servidor de banco de dados (*localhost)*;

**Linha 35** - O objeto db recebe do mongoClient o banco de dados com nome database;

**Linha 36** - O objeto social\_networks\_collection recebe da coleção db os campos com nome social\_networks\_datas;

**Linha 37** - O objeto users\_collection recebe da coleção db os campos com nome users.

O método demonstrado no Código 31, quando acionado, obtém todas as palavras-chave dos usuários através da *collection* users do banco de dados. Os documentos desta *collection* contém um campo do tipo *array* denominado users\_keywords, o qual contém a lista de palavras-chave de cada usuário da aplicação web.

1. public String[] getAllUserKeywords(){
2. DBCursor cursor = users\_collection.find();
3. BasicDBList dbKeywords = new BasicDBList();
4. ArrayList<String> keywordsList = new ArrayList<String>();
5. while (cursor.hasNext()) {
6. dbKeywords = (BasicDBList) (cursor.next().get("user\_keywords"));
7. for (Object dbOject : dbKeywords) {
8. keywordsList.add((String)dbOject);
9. }
10. }
11. cleanedKeywords = new LinkedHashSet<String>(keywordsList);
12. keywordsToDataMining = new String[cleanedKeywords.size()];
13. keywordsToDataMining = cleanedKeywords.toArray(keywordsToDataMining);
14. return keywordsToDataMining;
15. }

Código 31- Busca de todas as palavras-chave cadastradas no MongoDB. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 41** - O objeto cursor recebe as *keywords* de users\_collection cadastradas pelos usuários da aplicação web;

**Linhas 45 a 51** - Inicia-se um *while* para ser feita a consulta das palavras-chave no objeto cursor. Logo após, o objeto dbKeywords recebe as palavras-chave do objeto cursor. E, posteriormente, é feito um *foreach* para ir adicionando as palavras-chave no *array* keywordsList;

**Linha 56** - O objeto keywordsToDataMiningrecebe uma conversão em formato de *array* do objeto cleanedKeywords;

**Linha 57** - São retornadas as palavras-chave por meio do objeto keywordsToDataMining;

É demonstrado, no Código 32, o método responsável por salvar um novo *tweet* no banco de dados, em formato de documento. Este método é o responsável por gerar o *Big Data* desta pesquisa. Caso o método save apresente um erro, é lançada uma *exception* que será posteriormente mostrada no terminal.

1. public void saveInDB(BasicDBObject document) {
2. try {
3. social\_networks\_collection.save(document);
4. } catch (Exception ex) {
5. System.out.println("Erro ao salvar documento: " + ex.getMessage());
6. }
7. }

Código 32 - Método responsável por para salvar um documento no banco de dados. Fonte: Elaborado pelos autores.

Logo após o desenvolvimento da classe responsável por fazer as operações com o banco de dados, era necessário construir classe *Bolt*, a qual é responsável por transformar uma tupla (linha de dados) recebida da classe PolitistatusTwitterSpout em um objeto que contém os dados processados para salvar no banco de dados. A tupla é o objeto que contém o *tweet* recém-minerado.

O método demonstrado no Código 33 faz parte do corpo da classe PolitistatusTwitterBolt. Sua função é separar a tupla, a qual contém o *tweet*, em *tweet\_id*, *location* e *date*. Porém, antes de fazer o armazenamento do *tweet*, deve ser feita uma validação, que consiste na checagem do comentário do *tweet*. Se ele contiver alguma palavra-chave do *array*, o objeto é montado, juntamente com a palavra-chave validadora. Posteriormente, é chamando o método saveInDB da classe PolitistatusDatabase, onde enfim é feito o armazenamento dos dados.

1. @Override
2. public void execute(Tuple tuple, BasicOutputCollector collector) {
3. Status status = (Status) tuple.getValueByField("tweet");
4. System.out.println(status);
5. for (String keyword :
6. PolitistatusTwitterBolt.getInstance().getKeywords()) {
7. if(status.getText().contains(keyword)){
8. document = new BasicDBObject();
9. document.put("tweet\_id", status.getId());
10. document.put("location", status.getUser().getLocation());
11. document.put("date", status.getCreatedAt());
12. document.put("keyword", keyword);
13. PolitistatusDatabase.getInstance().saveInDB(document);
14. }
15. }
16. }

Código 33 - Método execute da classe PolitistatusTwitterBolt. Fonte: Elaborado pelos autores.

Quando existem múltiplas a serem processadas pelo *bolt*, torna-se possível declarar um nome específico para cada uma através da implementação do método declareOutputFields. Como nesta pesquisa o evento relacionado à tupla é único (os *tweets* são recebidos um de cada vez), não foi necessária a implementação da lógica deste método, pois, a tupla, já vem nomeada genericamente como “*tweet*”, na implementação do *Spout*, a qual será vista posteriormente.

Os métodosgetKeywordse setKeywordsservem para, respectivamente, armazenar e retornar o *array* de palavras-chave que estão sendo usados na mineração dos dados. Todos os métodos descritos acima também pertencem à classe PolitistatusTwitterBolt, e são demonstrados no Código 34.

1. @Override
2. public void declareOutputFields(OutputFieldsDeclarer ofd) {
3. }
4. public String[] getKeywords() {
5. return keywords;
6. }
7. public void setKeywords(String[] keywords) {
8. this.keywords = keywords;
9. }

Código 34 - Métodos declareOutputFields, getKeywords e setKeywords. Fonte: Elaborado pelos autores.

A próxima ação a ser tomada depois da criação do *Bolt*, era a criação do *Spout*. Basicamente, o *Spout* é a classe responsável pela a comunicação entre o emissor de mensagens e a parte interna da topologia de mineração de dados, a qual é previamente criada na classe *main* do Storm.

A classe *Spout* desenvolvida foi a PolitistatusTwitterSpout. O Código 35 demonstra seu construtor.

1. public PolitistatusTwitterSpout(String consumerKey, String
2. consumerSecret, String accessToken, String accessTokenSecret, String[]
3. keyWords) {
4. this.consumerKey = consumerKey;
5. this.consumerSecret = consumerSecret;
6. this.accessToken = accessToken;
7. this.accessTokenSecret = accessTokenSecret;
8. this.keyWords = keyWords;
9. }

Código 35 - Construtor da classe PolitistatusTwitterSpout. Fonte: Elaborado pelos autores.

Os parâmetros recebidos no construtor do *Spout* são provenientes da classe de construção da topologia de mineração dos dados. Os parâmetros consumerKey, cosumerSecret, accessToken e accessTokenSecret contém as credenciais de conexão com a Twitter *Stream* API, enquanto o parâmetro keyWords contém as palavras-chave cadastradas pelos usuários da aplicação web.

O Código 36 demonstra as implementações provenientesda biblioteca Twitter4j,as quais oferecem a oportunidade de tratamento dos eventos relacionados aos *tweets* que estão sendo minerados.

1. @Override
2. public void open(Map conf, TopologyContext context,
3. SpoutOutputCollector collector) {
4. queue = new LinkedBlockingQueue<Status>(1000);
5. \_collector = collector;
6. StatusListener listener = new StatusListener() {
7. @Override
8. public void onStatus(Status status) {
9. queue.offer(status);
10. }
11. @Override
12. public void onDeletionNotice(StatusDeletionNotice sdn) {
13. }
14. @Override
15. public void onTrackLimitationNotice(int i) {
16. }
17. @Override
18. public void onScrubGeo(long l, long l1) {
19. }
20. @Override
21. public void onException(Exception ex) {
22. }
23. @Override
24. public void onStallWarning(StallWarning arg0) {
25. // TODO Auto-generated method stub
26. }
27. };

Código 36 - Implementações da biblioteca Twitter4j na classe PolitistatusTwitterSpout. Fonte: Elaborado pelos autores.

As funções de cada método são detalhadas abaixo:

***onStatus*** - Este método é chamado quando um *tweet* relacionado com alguma palavra-chave é criado. Na implementação desta pesquisa, o *tweet* é colocado em uma fila para posteriormente ser enviado ao *bolt* do Storm, para enfim ser feito o seu processamento;

***onDeletionNotice*** - Este método é chamado por meio de um evento que é lançando quando um *tweet*, relacionado a alguma palavra-chave, é deletado. Desse modo é possível realizar um tratamento dos *tweets* que foram deletados. O tratamento dos *tweets* deletados não é abordado neste trabalho, então, a lógica deste método não foi implementada;

***onTrackLimitationNotice*** - Este método é chamado quando a palavra-chave cadastrada retorna uma quantidade elevada de *tweets* constantemente, indicando assim que a mesma é muito ampla. Esta pesquisa tem como embasamento teórico a política, não sendo necessário assim criar uma lógica de tratamento de palavras-chave genéricas;

***onScrubGeo*** - Este método é chamado por meio de um evento que é lançando quando um *tweet*, relacionado a alguma busca que envolve localização, é deletado. Do mesmo modo que o método onDeletionNotice, torna-se possível realizar o tratamento destes *tweets* que foram deletados. Esta pesquisa não realiza filtragem de *tweets* por localização, então não foi necessária a implementação da lógica deste método;

***onException*** - Este método é chamado quando é lançada uma exceção relacionada à conexão com a Twitter *Stream* API, possibilitando assim ser feito o tratamento de erros de conexão. Devido a conexão com o Twitter *Stream* API ser feita por meio de uma conta única, e as credenciais de conexão serem previamente cadastradas, os autores deste trabalho optaram por não implementar a lógica deste método;

***onStallWarning*** - Este método é chamado quando ocorre a perda de algum *tweet*, ou seja, o *tweet* em questão foi detectado, porém ele não foi enviado para a topologia de processamento Storm. O tratamento de perda de *tweets* não é abordado neste trabalho, então, a lógica deste método não foi implementada;

Continuando com o método open a classe PolitistatusTwitterSpout, depois de ser configurada a fila de recebimento dos *tweets*, foi necessário criar a conexão com a Twitter *Stream* API, através da manipulação dos objetos provenientes da biblioteca Twitter4j. Desse modo, o Código 37 demonstra a criação e atribuição dos eventos do objeto twitterStream, o qual fica responsável pelo conexão e controle do fluxo de dados do Twitter, que posteriormente são encaminhados para o *Spout* do Storm.

1. TwitterStream twitterStream = new TwitterStreamFactory(
2. new ConfigurationBuilder().setJSONStoreEnabled(true).build())
3. .getInstance();
4. twitterStream.addListener(listener);
5. twitterStream.setOAuthConsumer(consumerKey, consumerSecret);
6. AccessToken token = new AccessToken(accessToken,
7. accessTokenSecret);
8. twitterStream.setOAuthAccessToken(token);
9. \_twitterStream = twitterStream;
10. if (keyWords.length == 0) {
11. twitterStream.sample();
12. } else {
13. FilterQuery query = new FilterQuery().track(keyWords);
14. twitterStream.filter(query);
15. }
16. }

Código 37 - Criação do Twitter *Stream*. Fonte: Elaborado pelos autores.

Criados a conexão e o controle do fluxo de dados, foi necessário criar o emissor de *tweets* para o *bolt* do Storm. Desse modo, o Código 38 demonstra o método nextTuple, o qual lê as tuplas, que estão contidas na fila, uma de cada vez, posteriormente enviando-as ao *bolt*.

1. @Override
2. public void nextTuple() {
3. Status ret = queue.poll();
4. if (ret == null) {
5. Utils.sleep(50);
6. } else {
7. \_collector.emit(new Values(ret));
8. }
9. }

Código 38 - Método de leitura das tuplas. Fonte: Elaborado pelos autores.

Caso fosse necessário, foi implementado o método que encerra a conexão com o Twitter *Stream API*, demonstrado no Código 39.

1. @Override
2. public void close() {
3. \_twitterStream.shutdown();
4. }

Código 39 - Método de encerramento de conexão com o Twitter Stream API. Fonte: Elaborado pelos autores.

O Código 40 demonstra os métodos de controle do Storm. O método getComponentConfiguration é responsável por controlar o numero máximo de *threads* executoras de cada tarefa. Os métodos ack e fail são responsáveis por manipular as tuplas da fila após feita a sua execução. Já o método declareOutputFields tem a função de nomear genericamente cada tupla como “*tweet*”.

1. @Override
2. public Map<String, Object> getComponentConfiguration() {
3. Config ret = new Config();
4. ret.setMaxTaskParallelism(1);
5. return ret;
6. }
7. @Override
8. public void ack(Object id) {
9. }
10. @Override
11. public void fail(Object id) {
12. }
13. @Override
14. public void declareOutputFields(OutputFieldsDeclarer declarer) {
15. declarer.declare(new Fields("tweet"));
16. }

Código 40 - Método configuração de tarefas. Fonte: Elaborado pelos autores.

Normalmente, a implementação do método ack retira a tupla da fila, impedindo-a de ser novamente processada, enquanto o método fail coloca-a novamente na fila, para ser novamente processada. Essa lógica é *default* do Storm, não sendo necessária nenhuma modificação em sua implementação.

Após os *Spouts* estarem configurados, foi necessário criar uma classe que passe os parâmetros de conexão com o Twitter para o *Spout*, além de executar o *framework* para realizar a mineração dos dados (classe *main*). A partir destas necessidades, foi criada a classe PolitistatusTwitterStream.

Os atributos consumerKey, consumerSecret, accessToken e accessTokenSecret da classe PolitistatusTwitterStream têm valor fixo, pois eles contém as credencias de acesso com o Twitter *Stream* API. Por outro lado, o atributo keyWords é variável, pois os usuários da aplicação web podem alterar as palavras-chaves do banco de dados a qualquer momento, sendo necessário realizar uma nova consulta para atualizar a lista de palavras-chave.

Do mesmo modo, para ser feita a criação da topologia de mineração de dados, é necessário realizar uma consulta no banco de dados, para realizar filtragem dos *tweets* com base nas palavras-chave que estão cadastradas no banco de dados. Desse modo, o Código 41 demonstra respectivamente a abertura da conexão com o banco de dados e a busca das palavras chave por meio do método getAllUserKeywords. Posteriormente então, as palavras-chave são salvas no atributo keyWords.

1. openDbConnection();
2. keyWords = getAllUserKeywords();

Código 41 - Operações de banco de dados na classe de execução. Fonte: Elaborado pelos autores.

Os Códigos 42 e 43 mostram, respectivamente, os métodos openDbConnection e getAllUserKeywords. O método openDbConnection inicia a comunicação com o banco de dados, em caso de falha exibe mensagem de erro juntamente com o código do erro. Já o método getAllUsersKeywords busca todas as palavras-chave cadastradas pelos usuários da aplicação web e as retorna.

1. private static void openDbConnection() {
2. try {
3. PolitistatusDatabase.getInstance().createDBConnection();
4. } catch (UnknownHostException ex) {
5. System.out.println("Erro ao conectar ao banco de dados: "
6. + ex.getMessage());
7. }
8. }

Código 42 - Método openDbConnection. Fonte: Elaborado pelos autores.

1. private static String[] getAllUserKeywords() {
2. try {
3. return PolitistatusDatabase.getInstance().getAllUserKeywords();
4. } catch (Exception ex) {
5. System.out.println("Erro ao buscar todos os dados: " +
6. ex.getMessage());
7. }
8. return null;
9. }

Código 43 - Método getAllUserKeywords. Fonte: Elaborado pelos autores.

O Código 44 demonstra a continuação da lógica do método main da classe PolitistatusTwitterStream aplicação após as linhas do Código 41.

1. if (keyWords != null) {
2. PolitistatusTwitterBolt.getInstance().setKeywords(keyWords);
3. buildTopology(consumerKey, consumerSecret, accessToken,
4. accessTokenSecret, keyWords);
5. } else {
6. System.out.println("Não existem políticos cadastrados!");
7. }

Código 44 - Lógica da classe de execução. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 28** **a 33** - Se o atributo keyWords não for nulo, seu conteúdo será enviado ao *bolt*. Também após a validação, é chamado o método buildTopology, passando como parâmetros os atributos consumerKey, consumerSecret, accesToken, accesTokenSecret e keyWords.

O Código 45 demonstra o método buildTopology, que contém as configurações necessárias para a construção da topologia de mineração de dados do Storm.

1. static void buildTopology(String consumerKey,
2. String consumerSecret, String accessToken,
3. String accessTokenSecret, String[] keyWords)
4. throws UnknownHostException {
5. TopologyBuilder builder = new TopologyBuilder();
6. builder.setSpout("twitter", new PolitistatusTwitterSpout(consumerKey,
7. consumerSecret, accessToken, accessTokenSecret, keyWords));
8. builder.setBolt("print", new PolitistatusTwitterBolt())
9. .shuffleGrouping("twitter");
10. Config conf = new Config();
11. final LocalCluster cluster = new LocalCluster();
12. cluster.submitTopology("Politistatus-topology", conf,
13. builder.createTopology());
14. Utils.sleep(10000);
15. Timer timer = new Timer();
16. timer.scheduleAtFixedRate(new PolitistatusTwitterStreamTask(
17. consumerKey,consumerSecret, accessToken,accessTokenSecret,
18. cluster, timer) { }, 0, 5000);
19. }

Código 45 - Método de construção da topologia do Storm. Fonte: Elaborado pelos autores.

Detalhamento de cada linha:

**Linha 63** - Instanciado o objeto do tipo TopologyBuilder, para ser feita a criação da *build* do Storm;

**Linha 65** - Setado o *Spout* que irá minerar os *tweets* relacionados com as palavras-chave cadastradas pelos usuários;

**Linha 67** - Setado o *Bolt* que vai processar as informações recebidas do *Spout*;

**Linha 75** - Criação da topologia de mineração de dados do Storm.

**Linha 79** - Instanciado o objeto timer, para ser feita a criação da *Task* para busca de palavras-chave atualizadas.

**Linhas 80 a 82** - Setada a classe responsável por manusear a *Task* de verificação de palavras-chave, sendo passados como parâmetros as informações necessárias para a reconstrução da topologia do Storm.

Como os usuários da aplicação web tem a liberdade de alterar as suas palavras-chave a qualquer momento, foi necessário criar uma lógica de atualização das mesmas. Para tal, foi criada uma *Task*, que é executada em um intervalo de tempo de 5 segundos.

Os métodos do Código 46 são os responsáveis pela lógica da *Task* em si. Primeiramente, as palavras-chave são novamente buscadas no banco de dados. Se houve uma diferença nas palavras-chave, destrói-se a topologia do Storm e cria-se outra, reiniciando assim o processo de mineração de dados.

1. @Override
2. public void run() {
3. String[] queriedKeyWords =
4. checkIfHasNewUsersKeywords(PolitistatusTwitterBolt.getInstance()
5. .getKeywords());
6. if(queriedKeyWords != null){
7. try {
8. timer.cancel();
9. PolitistatusTwitterBolt.getInstance().setKeywords(queriedKeyWords);
10. cluster.shutdown();
11. PolitistatusTwitterStream.buildTopology(consumerKey,
12. consumerSecret,accessToken, accessTokenSecret,
13. PolitistatusTwitterBolt.getInstance().getKeywords());
14. } catch (UnknownHostException e) {
15. System.out.println("Erro ao criar a nova topologia: " +
16. e.getMessage());
17. }
18. };
19. }
20. private static String[] checkIfHasNewUsersKeywords(String[] keyWords){
21. String[] queriedKeyWords =
22. PolitistatusDatabase.getInstance().getAllUserKeywords();
23. return Arrays.equals(keyWords, queriedKeyWords) ? null :
24. queriedKeyWords;
25. }

Código 46 - Métodos de reconstrução da topologia do Storm. Fonte: Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 35** - É criado um *vetor* de *Strings* que recebe o retorno da função checkIfHasNewUsersKeywords, sendo passado como parâmetro as *keywords* que estão sendo enviadas à Twitter *Stream* API;

**Linhas 38 a 45** - Se o objeto queriedKeywords não estiver nulo, a *Task* é cancelada, e é setado no *bolt* as novas palavras-chave a serem buscadas e destruída a topologia do Storm. Logo após essas etapas, é criada uma nova topologia sendo passadas como parâmetro as informações para a conexão com a Twitter *Stream* API juntamente com as novas palavras-chave;

**Linhas 53 a 58** - Método checkIfHasNewUsersKeywords. Este método é responsável por fazer uma nova consulta de palavras-chave no banco de dados. Se as palavras-chave foram alteradas, o método irá retorná-las, senão, o retorno será nulo.

Com a aplicação web e o *framework* Apache Storm devidamente configurados e operantes, iniciou-se a fase de *deploy*, mineração dos dados e geração do *Big Data*.

### Hospedagem na *Amazon Web Services*

Com o sistema devidamente desenvolvido, o próximo passo foi fazer a hospedagem do sistema na Amazon *Web Services*, para que, finalmente, os usuários tenham acesso a todas as suas funcionalidades. A criação e configuração da instância na Amazon *Web Services* são detalhadas no Apêndice II.

### Integração das tecnologias - Mineração dos dados, geração do *Big Data* e demonstração dos dados políticos

Com o sistema devidamente hospedado na Amazon *Web Services*, o próximo passo era minerar e armazenar os *tweets*, bem como demonstrá-los para o usuário da aplicação web. Para ser feita a mineração, o usuário da aplicação web deve cadastrar pelo menos uma palavra-chave na busca.

Logo após o cadastro, o *framework* Apache Storm se encarrega de buscar a palavra-chave no banco de dados, conforme demonstrado no Código 31. Após ser obtida a palavra-chave, o Storm cria a sua topologia de mineração de dados, conforme anteriormente demonstrado no Código 45.

Para ser feita a criação da topologia de mineração de dados do Twitter, o Storm precisa implementar os métodos da biblioteca *Twitter4j*. Assim, torna-se possível criar a conexão com o Twitter *Stream* API, e consequentemente, com o Twitter. Durante a criação da topologia de mineração de dados, é passada a palavra-chave do usuário como parâmetro para ser feita a busca dos *tweets*, conforme anteriormente demonstrado no Código 37.

Quando um novo *tweet*, contendo a palavra-chave, é feito, o Storm encaminha-o para o seu *bolt*, que irá fazer a validação do *tweet*. Se o *tweet* estiver válido, suas informações serão salvas no MongoDB. Esta lógica é demonstrada no código no Código 33. A Figura 17 exemplifica os *tweets* já salvos no MongoDB.

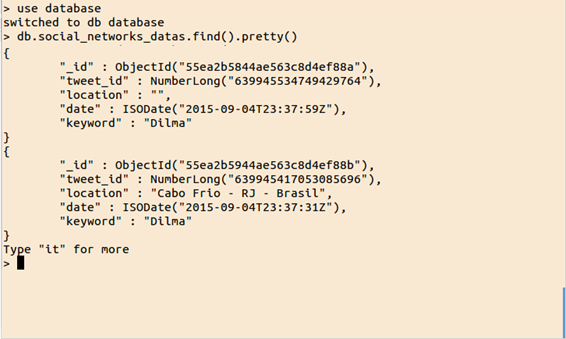


Figura 17 - *Tweets* no MongoDB, no formato de documentos. Fonte: Elaborado pelos autores.

Paralelamente a toda esta lógica, foi implementada uma *Task* que tem a finalidade de verificar se as palavras-chave informadas pelo usuário foram alteradas. Ela é criada após a criação da topologia de mineração de dados do Storm, e executa em um intervalo de tempo de 5 segundos, continuamente.

A lógica da *Task* consiste em realizar uma busca no banco de dados, retornando uma lista contendo todas as palavras-chave cadastradas pelos usuários da aplicação web. Caso essa lista de palavras-chave tenha sido alterada, destrói-se a topologia de mineração antiga, criando uma nova topologia para a mineração dos dados atualizados, iniciando novamente o ciclo de vida do Storm. Toda a lógica da *Task* é demonstrada no Código 46.

Com todas essas implementações do Storm, o crescimento do banco de dados torna-se exponencial. À medida que o tempo passa mais *tweets* são criados, e, consequentemente, mais *tweets* são armazenados no banco de dados, criando assim, o *Big Data* desta pesquisa.

O fluxo do servidor Node.JS é paralelo a toda essa atividade de mineração e armazenamento de dados. No servidor, além de conter toda a lógica de *back-end* da aplicação web, contém também um mecanismo que tem a finalidade de buscar os *tweets* mais recentes do banco de dados. Esse mecanismo, com o funcionamento semelhante à *Task* criada no Storm, é executado em um intervalo de tempo de 1 segundo. O código correspondente a este mecanismo é ilustrado no Código 47.

1. getUpdatedComments: function (client, user) {
2. try {
3. isQuerying = true;
4. this.queryComments = setTimeout(function () {
5. Politics.getAllUserPolitics(user, function (politics) {
6. if (politics != null) {
7. var politicsArray = [];
8. politics.forEach(function (item) {
9. item.user\_keywords.forEach(function (keyword) {
10. politicsArray.push(keyword);
11. });
12. });
13. if (politicsArray.length > 0) {
14. SNData.getUpdatedComments(politicsArray,dateToSearch,
15. function (snData) {
16. if (snData != null) {
17. clearTimeout(queryComments);
18. if (snData.length > 0) {
19. snData.forEach(function (item) {
20. if (item.date >= dateToCompare) {
21. console.log(item);
22. dateToCompare = item.date;
23. dateToSearch = item.date.toISOString();
24. client.emit('data', item);
25. } else {
26. console.log("Registro com data menor!");
27. }
28. });
29. }
30. }
31. });
32. } else {
33. console.log("Sem nenhum político cadastrado na busca!");
34. }
35. }
36. });
37. SNDataController.getUpdatedComments(client, user);
38. }, 1000);
39. }
40. catch (ex)
41. {
42. console.log(ex.message);
43. res.status(500).send("Problemas ao buscar os comentários atualizados!");
44. }
45. }

Código 47 - Método da busca atualizada e envio dos tweets. Fonte: Elaborado pelos autores.

Funcionamento do código:

Inicialmente, quando este método é chamado, são recebidos como parâmetro o cliente da conexão *socket* e o *login* do usuário que está logado na aplicação web. Posteriormente, é feita uma consulta no banco de dados para obter as palavras-chave com base no usuário que está logado. Se essa consulta retornar dados, eles são iterados e salvos em um objeto do tipo *array*. Posteriormente, uma nova consulta é executada, porém, dessa vez, é feita no documento de socialNetworksData, o qual é o *Big Data* propriamente dito, onde estão armazenados todos os *tweets* minerados pelo Storm.

Para ser feita a consulta dos *tweets* mais recentes, foi necessário passar como parâmetros: o *array* de palavras-chave e a data do último *tweet* que foi buscado, para que possam ser consultados os *tweets* mais recentes do banco de dados. Se ainda não foi buscado nenhum *tweet*, a data *default* é a do sistema.

Quando a consulta no banco de dados retorna um ou mais *tweets* recentes, as datas de consulta no banco de dados são atualizadas para a data do *tweet* mais recente, e, posteriormente, e os *tweets* são enviados via *socket* para o *client-side* da aplicação web. Essa lógica foi possível graças a implementação do *framework* Socket.IO no servidor. Desse modo, existe uma conexão *socket* sempre aberta para o envio dos *tweets* atualizados para o usuário.

Os *tweets* recebidos no *client-side*, tanto por meio da conexão *socket* quanto pelo filtro de data, estão no formato JSON. Porém, eles precisam ser gerados e mostrados para o usuário. Para tal, foi criada uma modal do Bootstrap responsável por mostrar esses *tweets*. A geração dos *tweets* é feita através do arquivo JavaScript controlador da modal dos *tweets*. Este JavaScript faz uma chamada em outro arquivo JavaScript, do próprio Twitter, o qual recebe o *id* do *tweet* como parâmetro. Posteriormente, o *tweet* é carregado e retornado para o JavaScript controlador da modal, que que finalmente se encarrega de mostrá-lo.

Paralelamente à demonstração dos *tweets* para o usuário, é feita a marcação dos locais no mapa. Primeiramente, é setado no mapa, a latitude, a longitude e o zoom para que a área do território brasileiro fique devidamente centralizada na tela. Estas informações foram obtidas através do próprio site do Google Maps. Quando os *tweets* são recebidos via *socket*, é chamada a função getLocationByAddress do JavaScript controlador do mapa, sendo passado como parâmetro a cidade do usuário responsável pelo *tweet*. Essa função fica responsável por passar essa cidade para uma URL do GoogleMaps API, a qual irá retornar um objeto do tipo JSON contendo a latitude e a longitude da cidade.

Após o retorno das informações de latitude e longitude, é chamada a função setMaker do controlador do mapa, a qual fica responsável por criar um marcador no mapa indicando qual a cidade e qual palavra-chave que gerou aquele *tweet*. Toda a lógica do mapa foi previamente demonstrada nos Códigos 21, 22, 23, 24 e 25.

Juntamente com o mapa, existe um botão que, quando clicado, demonstra para o usuário um gráfico do tipo pizza com a quantidade de menções no Twitter por cada palavra-chave cadastrada. O gráfico é feito por meio do uso da biblioteca Highcharts. O controlador do gráfico foi previamente demonstrado nos Códigos 26, 27, 28 e 29.

Inicialmente, é feita a chamada na função setChartConfigurations, a qual é responsável por fazer a criação do gráfico e setar seus atributos, como tais como: título, tipo (pizza), *tooltips* e opções de plotagem.

Do mesmo modo que o mapa, as informações do gráfico são populadas através dos *tweets* recebidos via *socket*. Quando um novo *tweet* é recebido, a função updatePoliticsCounter é chamada, sendo passado como parâmetro a palavra-chave daquele *tweet*. Essa função verifica se o gráfico já está demonstrando dados desta palavra-chave.

Se o gráfico já estiver mostrando a quantidade de menções relacionadas àquela palavra-chave, seu contador é incrementado, atualizando assim sua quantidade de menções no gráfico. Porém, se o gráfico ainda não estiver demonstrando os dados desta palavra-chave, ela é imediatamente adicionada no gráfico, e seu contador com quantidade de menções é setado para 1, indicando que esta palavra-chave foi mencionada pela primeira vez.

A integração das tecnologias, descrita anteriormente, foi desenvolvida de forma que os dados políticos estejam sempre à disposição do usuário da aplicação. Os dados que estão presentes no *Big Data* estão disponíveis para o usuário final a qualquer momento via consulta por data, e, os dados minerados em tempo real, são demonstrados quase instantaneamente. Foi considerada também, na fase de desenvolvimento, a demonstração de informações tão relevantes quanto os próprios *tweets*, que são os locais de criação e a quantidade de menções dos mesmos. Desse modo, o sistema se torna uma ferramenta poderosa para a demonstração do atual cenário político brasileiro.

# DISCUSSÃO DE RESULTADOS

Neste capítulo são apresentados e discutidos os resultados obtidos pela pesquisa por meio do desenvolvimento das tecnologias empregadas no trabalho. Esta discussão foi realizada sob diferentes perspectivas, com o intuito de fazer a demonstração dos pontos fortes e fracos de cada uma delas.

Inicialmente, antes mesmo de ser feita a escolha do tema, o objetivo dos participantes era desenvolver um sistema de acesso público que abrangesse grande parte da população brasileira. Além disso, era necessário fazer a escolha das ferramentas e tecnologias que, preferencialmente, não fossem abordadas pela grade do curso de Sistemas de Informação.

Após serem feitas as pesquisas sobre as áreas de grande influência no Brasil, decidiu-se que o sistema deveria abranger o cenário político. A ideia inicial era ajudar a população brasileira a fazer uma análise dos papéis desempenhados pelos políticos brasileiros, por meio da demonstração da opinião pública dos mesmos. Para que isso fosse realizado, foi necessário pesquisar sobre as tecnologias a serem empregadas.

Inicialmente, foi constatado que um sistema web atenderia a necessidade do acesso público do sistema. O sistema web também tem a vantagem de poder atingir uma proporção maior da população brasileira. Seguindo esta direção, chegou-se a conclusão que as tecnologias relacionadas ao *Big Data* sanariam as necessidades do projeto, já que o volume de dados seria alto e as informações deveriam estar disponíveis para os usuários em tempo integral.

Além disso, foi necessário definir de onde os dados seriam minerados. Depois de algumas análises, foi decidido que o melhor caminho seria a mineração de dados de redes sociais. Dentre todas, a rede social Twitter foi a escolhida, pois por meio dela foi possível ter acesso às opiniões públicas de maneira mais direta e clara, além dos usuários estarem em constante expressão política em seus *tweets*. Para isso, foi utilizado o *framework* Apache Storm, o qual faz a mineração dos *tweets* através da conexão com o Twitter Stream API, realizada por meio da implementação da biblioteca Twitter4j.

Para ser feito o armazenamento das informações, foi necessária a escolha de um banco de dados especificamente designado para grandes volumes de dados, não podendo assim, ser um banco de dados relacional. Para tal atividade, foi utilizado o banco MongoDB, pois ele é um banco de dados NoSQL que tem um ótimo desempenho relacionado a volume de dados, além de ser designado à aplicações web.

Assim que foram decididas as tecnologias de mineração e armazenamento de dados, foi preciso decidir quais tecnologias seriam utilizadas para ser feito o desenvolvimento da aplicação web, definida como um dos objetivos específicos do projeto. Desse modo, foi utilizado o Node.JS juntamente com o *framework* Express, os quais auxiliaram na criação da infraestrutura da aplicação, rotinas *server-side* e persistência dos dados.

Para facilitar a comunicação, foi utilizada a linguagem JavaScript no *front-end*, devido ao servidor Node.JS ser inteiramente construído na linguagem JavaScript. Desse modo, foi também utilizada a linguagem de marcação HTML5 juntamente com a folha de estilos CSS3 e o *framework* Bootstrap para a apresentação das informações para o usuário da aplicação, além do EJS *tamplate engine*.

Após estas etapas, os objetivos específicos e as ferramentas da pesquisa estavam definidos, e os prazos estipulados. As tarefas foram posteriormente divididas entre os autores do projeto, sendo periodicamente realizadas reuniões para o acerto dos detalhes.

O resultado de todas estas etapas foi o sistema de estatísticas políticas Politistatus. O sistema foi desenvolvido para ser acessado por qualquer cidadão brasileiro com acesso à internet que deseja saber como está a aprovação popular de seu representante na rede social Twitter. O sistema foi desenvolvido para ser executado no servidor Node.JS, totalmente escrito na linguagem de programação JavaScript, em contraste com o *framework* de mineração de dados Apache Storm, inteiramente escrito na linguagem Java.

Confirmando o que foi previamente dito por Leibiusky, Eisbruch e Simonassi (2012), a implementação do Apache Storm foi simples de ser feita, resultando em um sistema confiável para o processamento do grande fluxo de dados.

Para que o Storm fosse capaz de minerar os dados do Twitter, foi necessária a implementação da biblioteca Twitter4j. Essa biblioteca é a responsável pela conexão do ApacheStorm com o Twitter *Stream* API, que, conforme mencionado por Twitter Developers (2015), é responsável por retornar os *tweets* e seus respectivos eventos.

Devido ao Storm ser inteiramente desenvolvido na linguagem Java, a implementação da biblioteca Twitter4j facilitou o processo de comunicação com o Twitter *Stream* API, confirmando assim o que foi previamente dito por Twitter4j (2015).

Para que o Storm crie a sua topologia de mineração de dados, é necessário existir alguma palavra-chave no banco de dados, já que o fluxo principal do sistema se inicia ao fazer o seu cadastro. Para tal, basta o usuário clicar no botão “Novo”, na seção de “Políticos a serem buscados”, que está no começo da página inicial. O usuário tem total controle de suas palavras-chave, pois além de poder ser cadastrada, a mesma pode ser editada ou deletada.

Todas as palavras-chave do usuário são listadas em uma *grid*, que também tem respectivamente os botões de edição e exclusão. As operações de um usuário nunca irão afetar outro usuário, desse modo, cada usuário tem sua lista de palavras-chave de forma independente.

Além de todas essas funcionalidades principais, o usuário tem também as opções de listagem, filtragem e ocultação da *grid*. Desse modo, a Figura 18 ilustra a seção de manipulação das palavras-chave do usuário, que comporta todas as funcionalidades descritas acima.



Figura 18 - Seção de manipulação das palavras-chave. Fonte: Elaborado pelos autores.

Após ser feito o cadastro de uma palavra-chave, o Storm automaticamente se encarrega de detectá-la e inseri-la em sua topologia de mineração de dados. Caso a palavra seja editada ou deletada, o Storm irá detectar a alteração e reconstruir a sua topologia de mineração. Este processo de constante verificação de palavras-chave se deu graças à *Task* que foi criada em sua implementação.

Ao criar uma nova topologia de mineração de dados, o Storm começa a minerar os *tweets* que contenham as palavras-chave que foram cadastradas pelos usuários. Estes *tweets* irão passar pelo processamento necessário, até que finalmente serão salvos no MongoDB.

Confirmando o que foi mencionado por Chodorow (2013), o banco de dados MongoDB foi flexível o bastante para armazenar a massa de dados gerada pelo Storm. Por ser um banco de dados orientado a documentos, foi de extrema facilidade realizar consultas mais complexas, sanando assim uma das necessidades do sistema.

Após feito o *deploy* do sistema, começou-se a notar que com o decorrer do tempo, a base de dados começou a crescer exponencialmente. Desse modo, notou-se que a política está sendo a motivação para a criação de uma grande quantidade de *tweets* pelo Brasil. Assim, foi percebido que o embasamento político foi a escolha certa para a mineração dos *tweets*, pois por meio deles que se deu a geração do *Big Data* desta pesquisa.

A mineração dos dados políticos, já no ambiente da Amazon *Web Services*, foi iniciada no dia 28/10/2015. Inicialmente, a instância da Amazon contava com 30% de uso de armazenamento, já que a instância com conta *free* tem 8GB de capacidade. Essa porcentagem era ocupada quase que integralmente pelo sistema operacional. Apenas uma pequena parte dessa porcentagem era ocupada pelos pré-requisitos do sistema, que estavam instalados, e os arquivos do *deploy*.

O processo do ApacheStorm, responsável pela mineração e armazenamento dos dados, ficou ativo em tempo integral na instância até o dia 31/10/2015. Neste dia, foi checado que o uso do armazenamento já se encontrava em 56%. Desse modo, foi notado que a porcentagem de uso de armazenamento subia em média 13% ao dia, por conta da grande quantidade de *tweets* que eram armazenados. Se o processo continuasse ativo em tempo integral, em cerca de 5 dias a instância estaria com 100% de armazenamento em uso.

Porém, como o sistema desta pesquisa tem um apelo demonstrativo em relação às tecnologias relacionadas à *Big Data*, para resolver o problema, os autores decidiram deixar o processo do *Storm* ativo até ser completada uma carga de no máximo 2% de uso de armazenamento por dia. Convertendo essa porcentagem em valores, o processo do Storm fica ativo na instância até serem baixados por volta de 164 MB de dados por dia.

Seguindo esta linha de contagem, a última verificação da porcentagem uso de armazenamento da instância foi no dia 08/11/2015, a qual estava atingindo a quantidade de 72%de uso de armazenamento. Considerando que, desta porcentagem 42% era relativo à mineração de dados, convertendo em valores, a quantidade de dados armazenados era de aproximadamente 3.4 GB. Essa quantia equivale a 545.264 *tweets* armazenados. Essa informação é ilustrada na Figura 19, a qual demonstra o console do MongoDB da instância da Amazon.

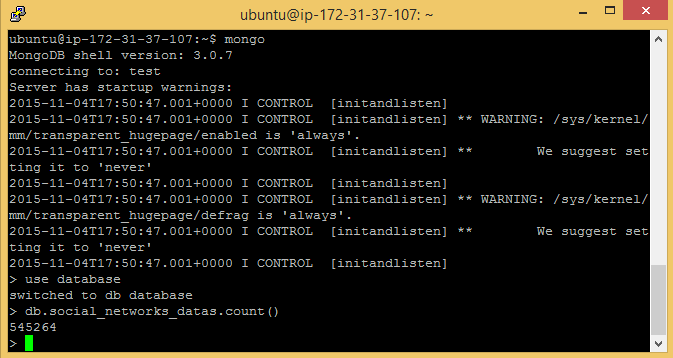
****

Figura 19 - Quantidade de tweets minerados até o dia 08/11/2015. Fonte: Elaborado pelos autores.

Voltando ao fluxo do sistema, após um novo *tweet* ser armazenado no banco de dados, o fica a cargo *back-end* da aplicação web buscá-lo e enviá-lo em tempo hábil para o *front-end*, via Socket.IO. Isso só foi possível graças a escolha e uso do Node.JS como servidor web do sistema. Sua implementação proporcionou à aplicação uma excelente performance e baixo consumo de memória, graças a seu modelo não-bloqueante, confirmando assim, o que foi previamente mencionado por Pereira (2013).

O *tweet*, já no *front-end*, é demonstrado na modal de visualização de *tweets*. A modal demonstra para o usuário todas as palavras-chave cadastradas em forma de abas, organizando assim, a visualização das informações.

Na modal de demonstração dos *tweets*, existe a seção de *Live Stream*. Essa seção demonstra somente os *tweets* que estão acontecendo em tempo real, ou seja, quando um usuário do *Twitter* cria um novo *tweet* que contenha alguma das palavras-chave cadastradas, ele será demonstrado na seção de *Live Stream*. Nessa seção, existem dois botões: “Pausar” e “Retomar”. O botão “Pausar” tem o intuito de pausar temporariamente o serviço de *Live Stream*, ou seja, pausar a busca em tempo real dos *tweets*, visando facilitar a leitura das informações por parte do usuário. Já o botão “Retomar” reinicia o serviço de *Live Stream*, para que serviço de busca de *tweets* em tempo real seja novamente retomado.

Também na modal de visualização de *tweets*, existe a seção de consulta dos *tweets* mais antigos, sendo aplicado um filtro por data. Nesta seção, o usuário tem a liberdade buscar no banco de dados todos os *tweets* que foram feitos em um intervalo de tempo específico.

Porém, devido à grande quantidade de dados, esta consulta no banco teve que ser desenvolvida de uma forma otimizada, sendo usados três tipos de filtros:

* Filtro por palavra-chave: São buscados somente os *tweets* que contenham a palavra-chave da aba que está ativa no momento;
* Filtro por data: São buscados os *tweets* que estão em um intervalo de tempo entre duas datas;
* Filtro de quantidade: São buscados somente 50 *tweets* por vez. Se o usuário desejar ver mais *tweets*, no final da seção estará um botão de “Ver Mais”, que quando clicado, buscará os próximos 50 *tweets*, e assim, sucessivamente.

A disposição dos elementos da modal de exibição é idêntica para cada palavra-chave. O que segrega a exibição das informações são as abas, as quais contém as palavras-chave cadastradas. Assim, as informações que são apresentadas em uma aba nunca irão interferir nas apresentadas em outra aba, bem como os *tweets*, que sempre serão encaminhados para a aba com a sua respectiva palavra-chave. Para a demonstração, a Figura 20 ilustra a exibição da aba com a palavra-chave “Dilma”, a Figura 21 ilustra a exibição da aba com a palavra-chave “Lula”, enquanto a Figura 22 ilustra a exibição da aba com a palavra-chave “Aécio Neves”.

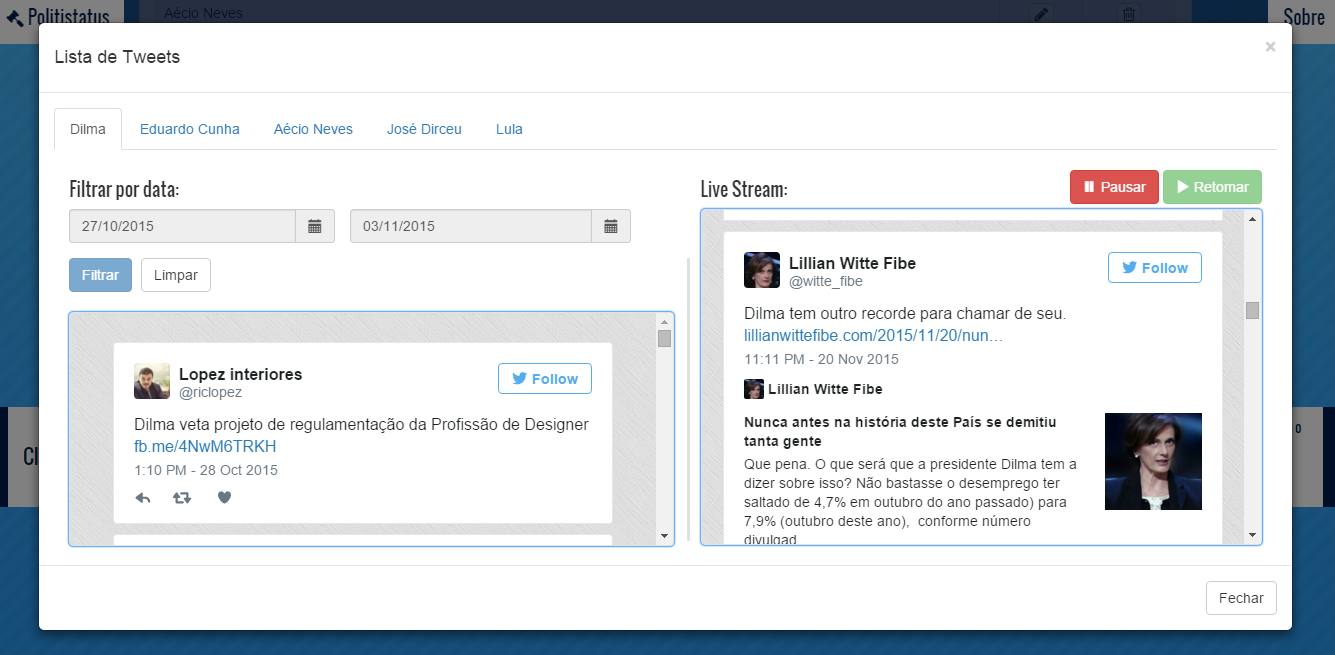


Figura 20 - *Tweets* da palavra-chave “Dilma”. Fonte: Elaborado pelos autores.

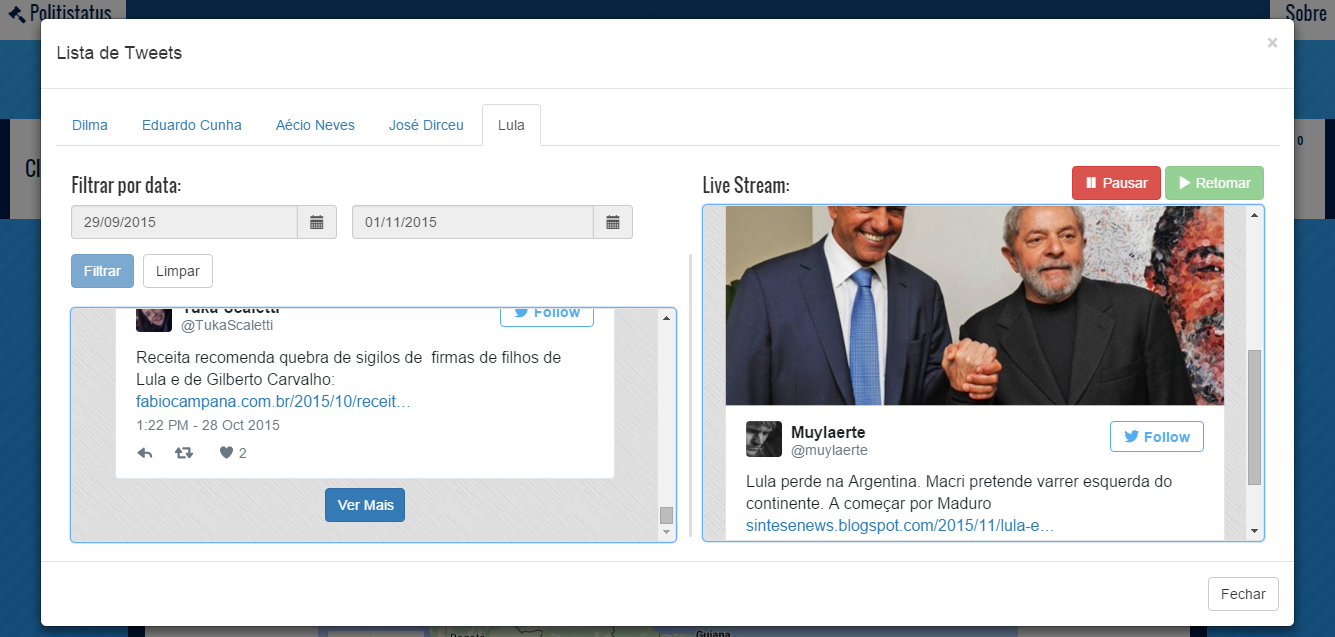


Figura 21 - *Tweets* da palavra-chave “Lula”. Fonte: Elaborado pelos autores.

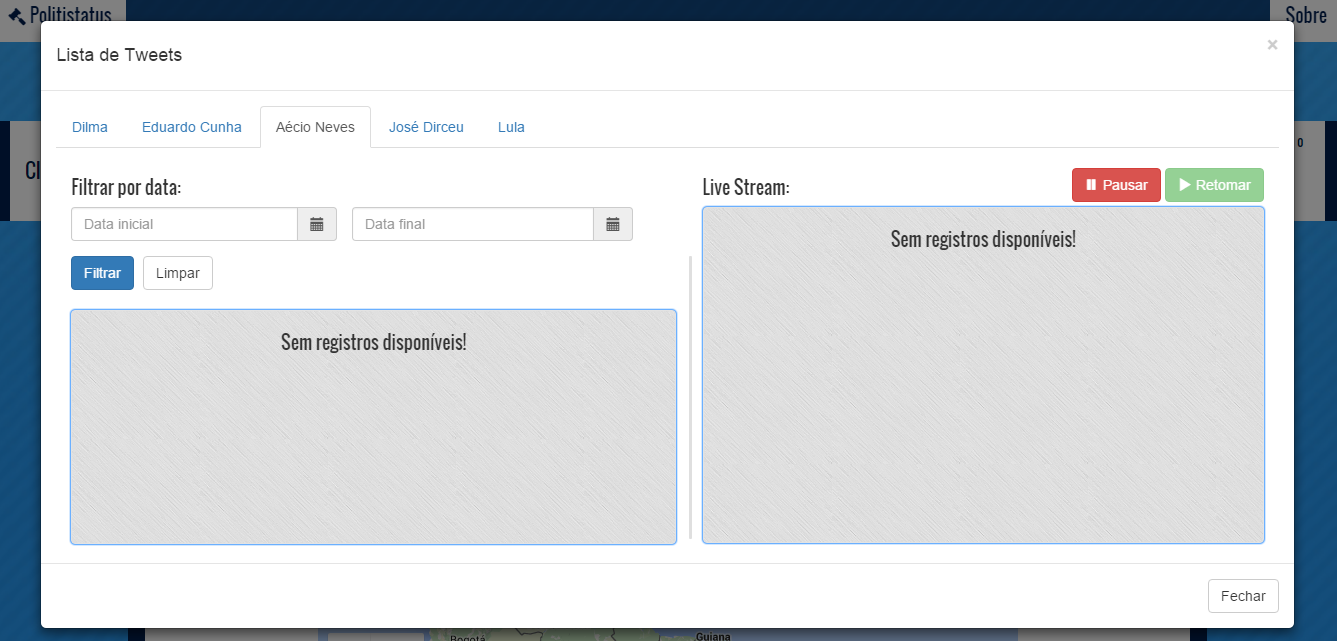


Figura 22 - Aba da palavra-chave “Aécio Neves”, sem nenhum *tweet* recebido. Fonte: Elaborado pelos autores.

Paralelamente à demonstração dos *tweets* em tempo real, é feita a demonstração dos seus respectivos locais de criação em um mapa, também em tempo real. Isso foi possível graças à implementação do Google Maps API, que além de possibilitar o mapa em si, possibilita também a criação de marcadores, confirmando assim, o comportamento dinâmico do mapa, previamente mencionado por Svennerberg (2010).

Os marcadores em si não têm muitas opções de personalização, no entanto, foi possível demonstrar as informações necessárias por meio do título do marcador, que informa ao usuário a cidade de criação do *tweet* e sua respectiva palavra-chave. Para visualizar essas informações, basta que o usuário passe o *mouse* em cima do marcador.

As Figuras 23, 24 e 25 demonstram o preenchimento do mapa com os marcadores. A Figura 23 demonstra o mapa em seu estado inicial, sem nenhum marcador. A Figura 24 demonstra o mapa após 10 segundos de uso da aplicação, e a Figura 25 demonstra o mapa após 20 segundos de uso da aplicação.



Figura 23 - Mapa da aplicação, em seu estado inicial. Fonte: Elaborado pelos autores.



Figura 24 - Mapa da aplicação, após 10 segundos de uso. Fonte: Elaborado pelos autores.



Figura 25 - Mapa da aplicação, após 20 segundos de uso. Fonte: Elaborado pelos autores.

Perceptivelmente, ao decorrer do tempo de uso da aplicação, o mapa irá ficar com diversos marcadores, dificultando assim a interpretação dos dados. Para resolver este problema, na parte inferior do mapa foi adicionado o botão “Limpar Marcadores”. Ao clicar neste botão, todos os marcadores do mapa são excluídos, deixando-o novamente em seu estado inicial, conforme demonstrado na Figura 23. Este botão é demonstrado na Figura 26.



Figura 26 - Botão de limpeza dos marcadores. Fonte: Elaborado pelos autores.

Os resultados previamente descritos são frutos de um sistema desenvolvido para proporcionar um maior nível de interação do usuário com os *tweets*. A modal de demonstração supre as necessidades de demonstração e interação para/com o usuário, e o mapa o deixa informado de quais são as palavras-chave que estão sendo mais comentadas em cada região e cidade. Porém, com o decorrer desta pesquisa, viu-se a necessidade de criar mais uma funcionalidade para o usuário. Essa funcionalidade teria que trazer um tipo de informação mais mensurável para o usuário, e que, ao mesmo tempo, demonstrasse alguma informação numérica, mais “palpável”.

Seguindo essa linha de raciocínio, os autores desta pesquisa decidiram fazer a implementação de um gráfico que demonstrasse a quantidade de menções por palavra-chave cadastrada. Para tal, foi adicionada a biblioteca Highcharts à aplicação web.

A implementação do gráfico Highcharts, mesmo sendo relativamente “tardia”, se comparada à implementação das outras tecnologias empregadas no sistema, não gerou nenhum conflito com nada do que já estava previamente desenvolvido, confirmando assim, o que foi mencionado por Kuan (2012).

Do mesmo modo que a demonstração dos *tweets* e a criação dos marcadores no mapa, as informações do gráfico são populadas por meio dos *tweets* que são recebidos via Socket.IO. O gráfico, em forma de pizza, informa ao usuário qual palavra-chave que ele cadastrou que está sendo mais mencionada no Twitter. Quando os *tweets* são recebidos no *front-end*, eles são segregados por palavra-chave, computados e demonstrados no gráfico.

Além de todas as funcionalidades anteriormente descritas, o gráfico também conta com as funcionalidades nativas do Highcharts, que incluem a impressão do gráfico e exportação para os formatos .PNG, .JPEG, .PDF e .SVG.

Para que o usuário tenha acesso ao gráfico, basta que ele clique no botão “Quantidade de Menções”, que está localizado na parte superior do gráfico da aplicação web. A Figura 27 ilustra o gráfico que foi desenvolvido, o qual está demonstrando a quantidade de menções no Twitter por palavra-chave cadastrada.

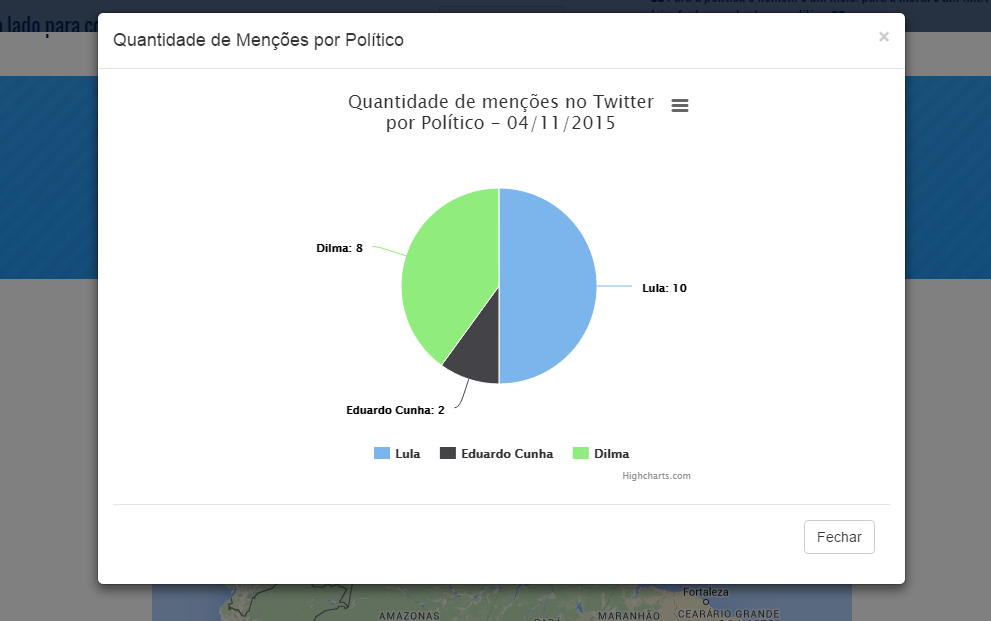


Figura 27 - Gráfico de menções no Twitter por palavra-chave. Fonte: Elaborado pelos autores.

Após a implementação de todas as funcionalidades previamente descritas, os autores desta pesquisa entraram no consenso que o sistema já apresentava uma gama de informações suficientes para o usuário. Assim, após a realização dos testes pós-fase de desenvolvimento, verificou-se que a aplicação já se encontrava apta a demonstrar de forma eficiente e concisa a opinião política dos usuários do Twitter. Desse modo, comprovou-se que o objetivo geral da pesquisa foi alcançado.

# CONCLUSÃO

Nesta pesquisa, foi demonstrado e desenvolvido um sistema para ser feita a demonstração de como a as tecnologias relacionadas à *Big Data* podem influenciar em uma determinada área de conhecimento humano. Além do sistema propriamente dito, também foram apresentadas as características e vantagens das tecnologias que o compõem, bem como os resultados obtidos por meio da integração entre elas.

Deve-se destacar também a importância da mineração de dados para a geração do *Big Data* desta pesquisa. Para que se gere o *Big Data*, a fonte deve ser confiável e ter um alto fluxo de dados por dia. Portanto, a rede social Twitter foi a escolha correta para esta pesquisa, já que milhões de *tweets* são gerados diariamente no Brasil. No entanto, foi necessário escolher um tema para o embasamento da mineração dos *tweets*. Tema este, que deveria ser responsável pela criação de uma grande quantidade de *tweets* pelo Brasil. Após análises, pode-se perceber que o tema político foi a escolha perfeita, pois o atual cenário político brasileiro favorece o ambiente de protesto em redes sociais, consequentemente, gerando grandes quantidades de *tweets* por dia.

Para ser feita a mineração de dados no Twitter, foi utilizado o *framework* Apache Storm integrado com a biblioteca Twitter4j, que, juntos, oferecem a conexão e processamento do grande volume de dados gerados pelo Twitter. Para o armazenamento do *Big Data*, foi utilizado o banco de dados MongoDB, que ofereceu ao sistema desta pesquisa uma ótima performance e escalabilidade, suportando o grande volume de dados e retornando consultas sem perda de velocidade.

A integração das tecnologias foi fundamental para que os objetivos desta pesquisa fossem atingidos, já que cada uma teve um papel específico para que o funcionamento do sistema atendesse às necessidades que o conceito de *Big Data* exige. *Big Data* não é somente armazenamento de quantidades maciças de dados, é também a apresentação das informações em tempo hábil para que possa ser feita a análise e tomada de decisão. Destacam-se aqui as tecnologias Node.JS, Express e Socket.IO, que, juntas, foram as responsáveis por buscar as informações no *Big Data*, tanto em tempo real, quanto por consultas em registros mais antigos.

A integração dessas tecnologias, se realizada da mesma maneira que nesta pesquisa, seria de grande valia para objetivos organizacionais. Dados que antes eram desperdiçados e ignorados, podem agora, por meio dessa integração de tecnologias, ser armazenados e analisados para se obter vantagem competitiva, além “abrir um leque” para uma série de outras vantagens: análise de tendências, análise de padrões, análise de aprovação e/ou reprovação pública, prevenção de erros, etc.

No que diz respeito a contribuição e relevância, esta pesquisa deixa como legado uma ferramenta com a capacidade de geração de informações analíticas, abrindo assim a possibilidade da tomada de decisões visando a vantagem competitiva.

No âmbito político, o sistema Politistatus possibilita a população brasileira, por meio da exposição dos *tweets*, a demonstração do desempenho de seus representantes perante a sociedade, abrindo a possibilidade de análise das informações e posterior reinvidicação de direitos ou decisão de voto. Em contrapartida, o sistema também possibilita ao próprio político saber como a população o está avaliando em seu cargo. Outra vantagem para o político em si é a demonstração das áreas de maior carência da população, auxiliando-o assim a atender a população com maior eficiência.

Aos acadêmicos do curso de Sistemas de Informação, esta pesquisa, além apresentar o conceito de *Big Data*, apresenta também os meios e as tecnologias empregados a fim de obtê-lo. Aos futuros trabalhos de pesquisa acadêmica, fica disponível a exploração do código-fonte e dos tutoriais criados, a fim de aprendizagem ou agregação de novas funcionalidades ao sistema.

Esta pesquisa demonstrou que o conceito de *Big Data* abre a possibilidade para diversas análises visando a tomada de decisão, nas mais diversas áreas do conhecimento humano. Assim, espera-se que tanto o documento quanto o sistema desenvolvido sejam de grande serventia aos que os utilizarem como referencial para futuros projetos.

# REFERÊNCIAS

ALECRIM, E. **O que é Big Data?**. Disponível em: <http://www.infowester.com/big-data.php>. Acessado em: 31/05/2015.

ANDRIOLE, S. **Unstructured Data: The Other Side of Analytics**. Disponível em: <http://www.forbes.com/sites/steveandriole/2015/03/05/the-other-side-of-analytics/2/>. Acessado em 07/05/2015.

APACHE. **Apache Storm**. Disponível em: <https://storm.apache.org/documentation/Concepts.html>. Acessado em 06/03/2015.

APACHE MAVEN PROJECT. **Maven**. Disponível em: <https://maven.apache.org/what-is-maven.html>. Acessado em 25/07/2015.

BROWN, E. **Web** **Development with Node & Express**. Sebastopol: O’Reilly, 2014.

CHORODOW, K. **MongoDB The Definitive Guide**. 2. Ed. Sebastopol: O’Reilly, 2013.

DEVMEDIA, **O que é NoSQL?**. Disponível em: <http://www.devmedia.com.br/o-que-e-nosql-java-magazine-86/18777>. Acessado em 03/03/2015.

FRANÇA, T. C.; FARIA, F. F de; RANGEL, F. M.; FARIAS, C. M. de; OLIVEIRA, J. **Big Social Data: Princípios sobre Coleta, Tratamento e Análise de Dados Sociais**. Disponível em: <http://www.inf.ufpr.br/sbbd-sbsc2014/sbbd/proceedings/artigos/pdfs/127.pdf>. Acessado em 05/05/2015.

HAN, J; KAMBER, M; PEI, J. **Data Mining Concepts and Techniques**. 3. Ed. Waltham: Morgan Kaufmann, 2012.

KUAN, J. **Learning Highcharts**. Birmingham: Packt Publishing Ltd, 2012.

LAVELLE, S.; LESSER, E.; SHOCKLEY, R.; HOPKINS, M. S.; KRUSCHWITZ, N. **Big Data, Analytics and the Path From Insights to Value**. Disponível em: <http://sloanreview.mit.edu/article/big-data-analytics-and-the-path-from-insights-to-value/>. Acessado em 05/05/2015.

LEIBIUSKY, J.; EISBRUCH, G.; SIMONASSI, D. **Getting Started Whith Storm.** Sebastopol: O’Reilly, 2012.

MAGNO, A. **Globo Bootstrap**. Disponível em:

<http://blog.alexandremagno.net/2012/08/globo-boostrap/>. Acessado em 19/04/2015.

MARAS, J. **Secrets of the JavaScriptNinja – The JavaScript ecosystem**. <http://java.dzone.com/articles/secrets-javascript-ninja>. Acessado em 13/05/2015.

MARCONI, M. A; LAKATOS, E. M. **Fundamentos de metodologia científica**. 6. ed. São Paulo: Atlas, 2009.

MARINESCU, D. C. **Cloud Computing Theory And Pratice.** Waltham: Elsevier Inc., 2013.

MASSÉ, M. **REST API:** Design Rulebook**.** Sebastopol: O’Reilly, 2011.

MCAFEE, A.; BRYNJOLFSSON, E. **Big Data: The Management Revolution**. Disponível em: <http://www.researchgate.net/profile/Erik\_Brynjolfsson2/publication/232279314\_Big\_data\_the\_management\_revolution/links/53ecf40e0cf23733e804e561.pdf>. Acessado em 04/05/2015.

MICROSOFT, **Java Script princípios básicos**. Disponível em: <https://msdn.microsoft.com/pt-br/library/ie/6974wx4d%28v=vs.94%29.aspx>. Acessado em 21/02/2015.

MOZILLA DEVELOPER NETWORK. **CSS3**. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3>. Acessado em: 05/06/2015.

MURTY, J. **Programming Amazon Web Services:** S3, EC2, SQS, FPS, and SimpleDB**.** Sebastopol: O’Reilly, 2008.

NAGARAJAN, M.; SHETH A.; VELMURUGAN, S. **Citizen Sensor Data Mining, Social Media Analytics and Development Centric Web Applications**. Disponível em: <http://knoesis.wright.edu/library/download/tr27-sheth1.pdf >. Acessado em 05/05/2015.

NODEBR. **Primeiros passos com Express e Node.js**. Disponível em: <http://nodebr.com/primeiros-passos-com-express-em-node-js/>. Acessado em: 24/05/2015.

OLIVEIRA, S. L. de. **Metodologia científica aplicada ao direito**. São Paulo: Pioneira Thomson Learning, 2002.

OLIVEIRA, S. L. de. **Tratado de metodologia científica**: projetos de pesquisa, TGI, TCC, monografia, dissertação e teses. 2. ed., quarta reimpressão. São Paulo: Pioneira Thomson Learning, 2002.

PEREIRA, A. P. **O que é CSS?**.Disponível em: <http://www.tecmundo.com.br/programacao/2705-o-que-e-css-.htm>. Acessado em 07/03/2015.

PEREIRA, C. R. **Aplicações web real-time com Node.js.** São Paulo: Casa do Código, 2013.

POWERS, S. **Learning JavaScript**. 2. Ed. Sebastopol: O’Reilly, 2009.

PRESS, G. **A Very Short History Of Big Data**. Disponível em: <http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/>. Acessado em 04/02/2015.

RUSSEL, M. A. **Mining the Social Web**. 2. Ed. Sebastopol: O’Reilly, 2013.

SADALAGE, P. J; FOWLER, M. **NoSQL Distilled:** A Brief Guide to the Emerging World of Polyglot Persistence**.** Crawfordsville: Pearson Education Inc, 2013.

SCUP IDEAS. **O voto conectado**. Disponível em:

< http://scupideas.scup.com/ideas/eleicoes-2014/>. Acessado em 06/05/2015.

SILVA, L. T. da; FERREIRA JUNIOR, A. B. **Marketing político e sua importância através das mídias sociais**. Disponível em: <http://periodicos.ufpb.br/ojs/index.php/tematica/article/viewFile/21645/11972>. Acessado em 06/05/2015.

SILVA, R. **Apostila de metodologia científica**. Brusque: ASSEVIM- Associação Educacional do Vale do Itajaí-Mirim, fev. 2008.

SOUBRA, D. **The 3Vs that define Big Data**.Disponível em: <http://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data>. Acessado em: 14/05/2015.

SVENNERBERG, G. **Beginning Google Maps API 3**. New York: Apress, 2010.

TWITTER4J. **Twitter4j - A Java library for the Twitter API**. Disponível em: <http://twitter4j.org/en/index.html>. Acessado em 30/05/2015.

TWITTER DEVELOPERS. **The Stream APIs Overview**. Disponível em: <https://dev.twitter.com/streaming/overview>. Acessado em 29/08/2015.

W3C. **Visão Geral do HTML5**. Disponível em: <http://www.w3c.br/cursos/html5/conteudo/capitulo1.html>. Acessado em 21/02/2015.

W3C. **A vocabulary and associated APIs for HTML and XHTML**. Disponível em: <http://www.w3.org/TR/html/introduction.html#introduction>. Acessado em 19/04/2015.

W3C. **A history of HTML**. Disponível em: <http://www.w3.org/People/Raggett/book4/ch02.html>. Acessado em 26/04/2015.

W3C. **Open Web Plataform**. Disponível em: <http://www.w3.org/wiki/Open\_Web\_Platform>. Acessado em 26/05/2015.

# ANEXO

**Anexo I: Folha de revisão de Língua Portuguesa**



**TERMO DE REALIZAÇÃO DE REVISÃO DE LÍNGUA PORTUGUESA**

Eu, MARIA AGDA MENDES ANDRADE professora de LÍNGUA PORTUGUESA, graduada pela FACULDADE DE FILOSOFIA CIÊNCIAS E LETRAS PROF. JOSÉ AUGUSTO VIEIRA (FAFIMA) informo ter realizado a revisão de Língua Portuguesa do Trabalho de Conclusão de Curso dos alunos Nícolas Henrique Vieira Toledo e Sebastião Batista de Andrade Neto do curso de Sistemas de Informação da UNIVÁS-Fafiep. Informo, ainda, que o trabalho encontra-se em conformidade com a norma padrão da Língua Portuguesa.

Pouso Alegre, \_\_\_\_\_/\_\_\_\_\_\_/\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Assinatura do(a) revisor(a)

# APÊNDICES

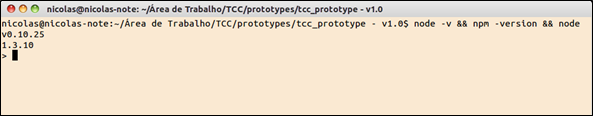
**Apêndice I: Pré-requisitos ambiente**

# INSTALAÇÃO DOS PRÉ-REQUISITOS DE AMBIENTE DO SISTEMA

O sistema desta pesquisa foi desenvolvido por meio da integração de diversas tecnologias. Para que ele funcionasse corretamente, foi necessário fazer todas as configurações de ambiente mínimas, bem como a instalação de todos os seus pré-requisitos.

A primeira ação tomada foi a instalação do Node.JS e de seu gerenciador de pacotes, o NPM, obtidos e instalados através da linha de comando “sudo apt-get install nodejs npm”. Após feita a instalação, automaticamente foram configurados o compilador e gerenciador de pacotes NPM. Desse modo foi possível observar a versão do Node.JS e do NPM que foram usadas no desenvolvimento do sistema, as quais consistem respectivamente na 0.10.25 e na 1.3.10. A Figura 1 demonstra, respectivamente, os comandos para:

* Visualizar a versão do Node.JS;
* Visualizar a versão do NPM;
* Iniciar o console do Node.JS.

****

**Figura 1** - Demonstração das versões do Node.JS e NPM, respectivamente. **Fonte:** Elaborado pelos autores.

Para se fazer real o armazenamento do *Big Data*, foi feito o *download* e instalação do banco de dados MongoDB, através das seguintes linhas de comando:

* **Importação da chave pública usada no sistema de gerenciamento de pacotes:**

sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10;

* **Criação arquivo de listas para o MongoDB:**

sudo touch /etc/apt/sources.list.d/mongodb-org-3.0.list && echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb\_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list;

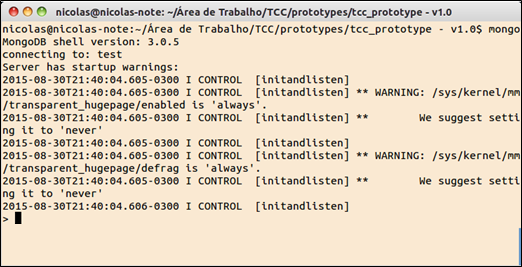
* **Atualização do repositório do sistema operacional:**

sudo apt-get update;

* **Instalação do MongoDB:**

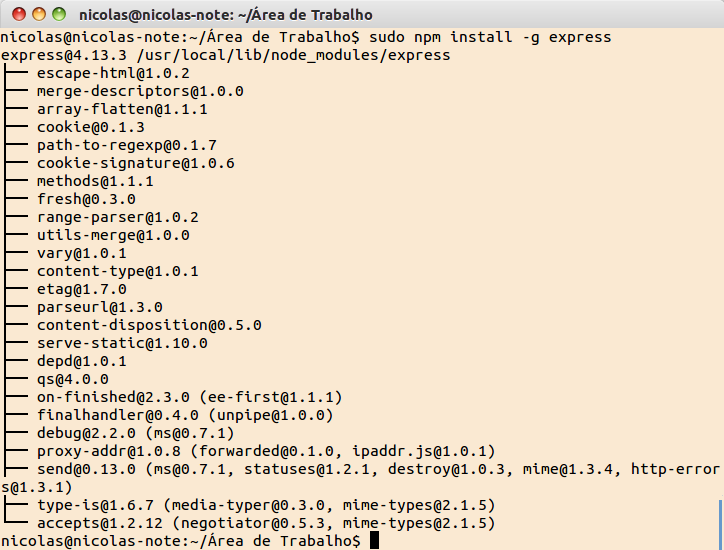
sudo apt-get install -y mongodb-org.

Automaticamente, seu executável foi adicionado na inicialização do sistema operacional, tornando-se assim disponível para a aplicação a qualquer momento. Logo após sua instalação, por meio de seu próprio terminal, foram criados respectivamente o banco de dados da aplicação e suas *collections*, para a posterior consulta, inserção e atualização dos dados. Sua inicialização, bem como seu terminal, é mostrada na Figura 2.

****

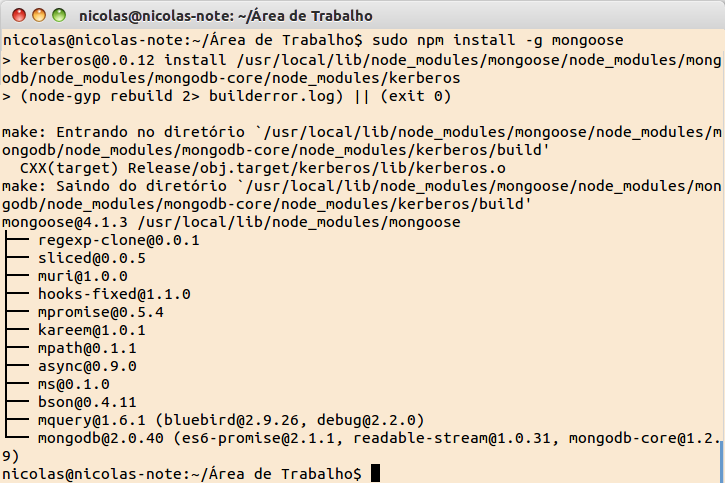
**Figura 2** - Terminal do MongoDB. **Fonte:** Elaborado pelos autores.

Com o banco de dados e o servidor Node.JS instalados, o próximo passo foi instalação e configuração do *framework* *Express* através da linha de comando “npm install –g express”, instalando-o assim como um módulo global através do parâmetro “-g”, conforme mostrado na Figura 3.



**Figura 3** - Demonstração da instalação do *framework Express*. **Fonte:** Elaborado pelos autores.

Depois do *framework* *Express*, foi necessário instalar o Mongoose, módulo responsável pela comunicação com o MongoDB. Ele foi instalado através da linha de comando “npm install –g mongoose”, também como módulo global, conforme ilustrado na Figura 4.



**Figura 4** - Demonstração da instalação do Mongoose. **Fonte:** Elaborado pelos autores.

Após todos os procedimentos descritos acima, os pré-requisitos de ambiente da aplicação web já estavam instalados. O posterior passo a ser tomado, foi a instalação dos pré-requisitos de ambiente do *framework* Apache Storm. O Storm é desenvolvido na linguagem Java, logo, o Java *Runtime Enviroment* e o Java *Development Kit* tem que estar instalados. Outro pré-requisito do Storm é o ApacheMaven, já que ele o usa como gerenciador de dependências. Para ser feita a instalação dos pré-requisitos do Storm, bastou executar as linhas abaixo no console do sistema operacional:

* **Instalação do JRE e JDK da Oracle:**

sudo apt-get install python-software-properties

sudo add-apt-repository ppa:webupd8team/java

sudo apt-get update

sudo apt-get install oracle-java7-installer

* **Instalação do *Apache Maven*:**

sudo apt-get install maven2

Após a execução de todas as configurações de ambiente descritas nesta seção, os pré-requisitos de ambiente do sistema dessa pesquisa se encontram devidamente instalados e configurados.

**Apêndice II: Hospedagem na Amazon *Web Services***

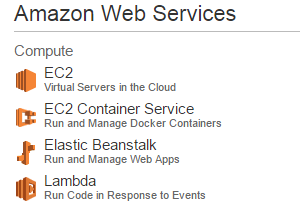
# HOSPEDAGEM

Um dos objetivos desta pesquisa é tornar o sistema disponível na web para que o público interessado possa desfrutar de suas funcionalidades. Para tal, foi criada uma máquina para ser feita a hospedagem no servidor da Amazon *Web Services*. Desse modo, o sistema tem a possibilidade de ficar disponível para acesso a qualquer momento.

## Criação da instância

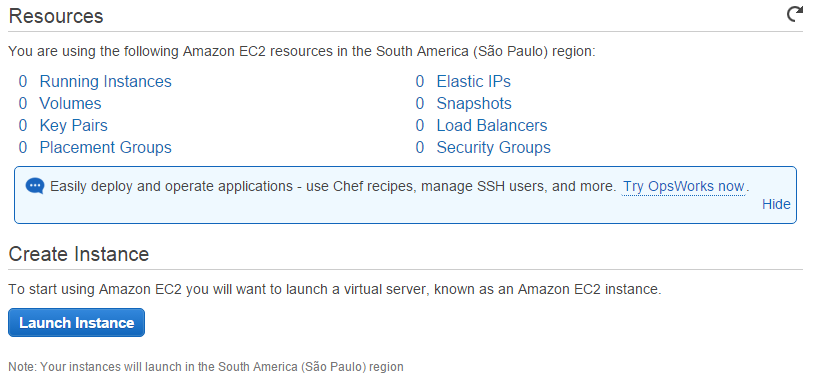
A primeira ação tomada foi o cadastro da conta no próprio site da Amazon *Web Services*. Parte do processo de cadastro consiste em receber uma chamada telefônica e informar um número de PIN por meio do teclado do telefone.

Depois de feito o cadastro da conta, o próximo passo foi criar uma instância no servidor. Ao fazer o *login*, é mostrada uma *dashboard* com todos os serviços disponibilizados pela Amazon *Web Services*. O serviço utilizado nesta pesquisa foi o EC2 (*Elastic Compute Cloud*). Esse serviço é específico para criar uma máquina em um servidor Amazon na nuvem. A Figura 1 ilustra parcialmente os serviços presentes na *dashboard* da Amazon *Web Services*, demonstrando somente a parte de serviços computacionais.



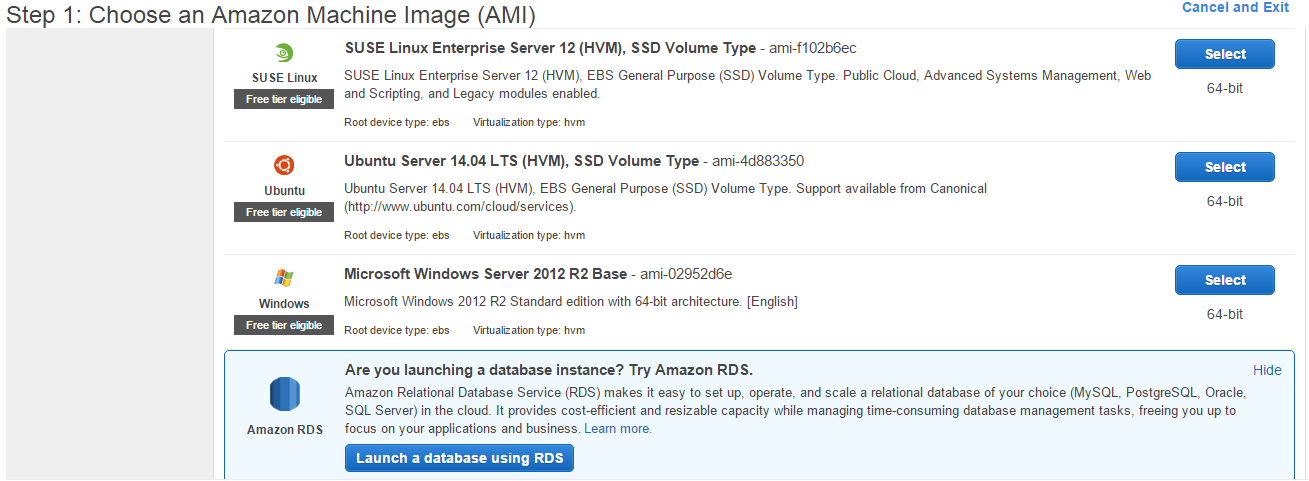
**Figura 1 -**Ilustração parcial da *dashboard* da *Amazon Web Services*. **Fonte:** Elaborado pelos autores.

Ao escolher o serviço EC2, é feito o redirecionamento para a página para a de criação de uma nova instância. A instância pode ser criada clicando no botão *Launch Instance*. A Figura 2 a página responsável por começar a criação de uma nova instância.



**Figura 2 -**Tela para começar a criar uma instância. **Fonte:** Elaborado pelos autores.

Após clicar em *Launch Instance*, é feito o redirecionamento para a página de escolha do sistema operacional da instância. Como o sistema operacional utilizado no ambiente de desenvolvimento desta pesquisa foi o Ubuntu 14.04 LTS, a escolha mais viável para a criação da instância era o Ubuntu Server 14.04 LTS, que é a versão específica para servidores deste sistema operacional. Desse modo, é garantido que a preparação do ambiente de hospedagem fique semelhante ao de desenvolvimento, minimizando assim o impacto de compatibilidade do sistema desenvolvido nesta pesquisa. A Figura 3 demonstra o Ubuntu Server 14.04 LTS na listagem de sistemas operacionais disponíveis.



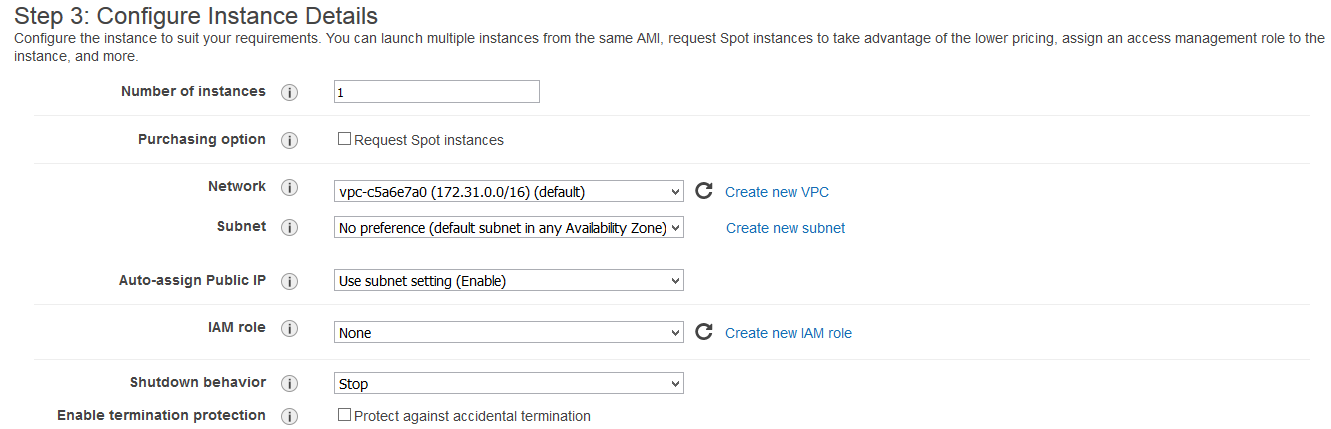
**Figura 3 -**Listagem de sistemas operacionais para a criação da instância. **Fonte:** Elaborado pelos autores.

Após feita a escolha do sistema operacional, é preciso fazer a escolha do tipo de instância que será utilizada. As configurações da máquina variam de acordo com a instância escolhida. Nesta pesquisa, o tipo escolhido foi o t2.micro, pois suas configurações são suficientes para hospedar o sistema desenvolvido nesta pesquisa, além de que ela tem o uso gratuito por 1 ano. A Figura 4 demonstra a escolha deste tipo de instância.



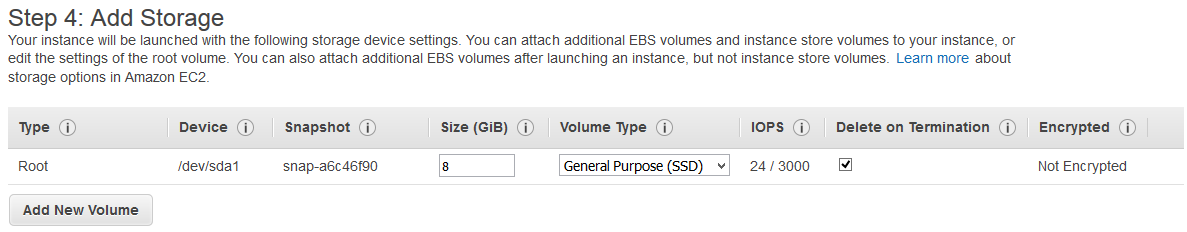
**Figura 4 -**Escolha do tipo da instância. **Fonte:** Elaborado pelos autores.

A próxima tela que é chamada foi a de configuração da instância. Como as configurações *default* atendem as necessidades desta pesquisa, nenhuma alteração foi feita. A Figura 5 ilustra esta tela.



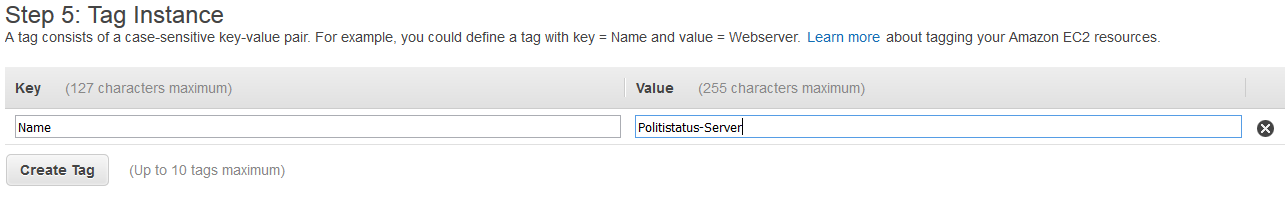
**Figura 5 -**Tela de configuração da instância. **Fonte:** Elaborado pelos autores.

A próxima tela chamada é a de configuração de armazenamento. O tipo de instância t2.micro oferece 8GB de armazenamento. Para os autores desta pesquisa, essa quantidade de armazenamento foi considerada como suficiente para a demonstração do sistema desenvolvido. Desse modo, nenhuma configuração adicional foi feita nesta tela. A Figura 6 ilustra a tela das configurações de armazenamento da instância.



**Figura 6 -**Tela de configuração de armazenamento da instância. **Fonte:** Elaborado pelos autores.

Depois das configurações de armazenamento, o próximo passo é escolher um nome para a instância. Este nome, que é conhecido como *tag*, irá aparecer no console da instância, quando a mesma for iniciada. Nesta pesquisa, a instância foi nomeada de Politistatus-Server. A Figura 7 ilustra a tela de configuração do nome da instância.



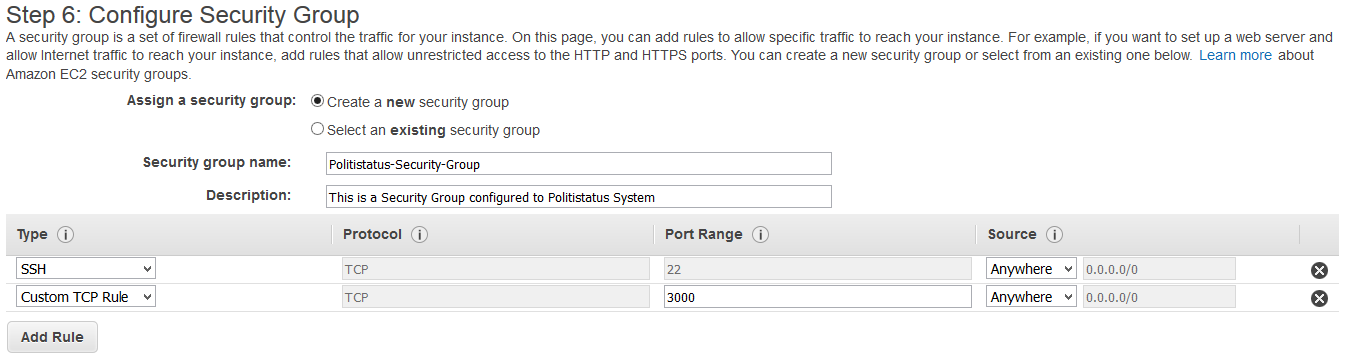
**Figura 7 -**Tela de configuração de armazenamento da instância. **Fonte:** Elaborado pelos autores.

O próximo passo, após a configuração do nome da instância, é definir um S*ecurityGroup*. O S*ecurityGroup* irá definir as regras de *firewall* da instância. Por *default*, a porta SSH vem liberada, para ser feita a conexão remota com a instância. Porém, nesta pesquisa, foi necessário também fazer a liberação da porta 3000, já que o servidor Node.JS Express é iniciado por *default* nessa porta. Para tal, foi necessário clicar no botão *Add Rule*, que irá inserir na *grid* de regras uma nova linha, para ser feita a configuração de uma nova regra.

As configurações de cada coluna ficaram da seguinte forma:

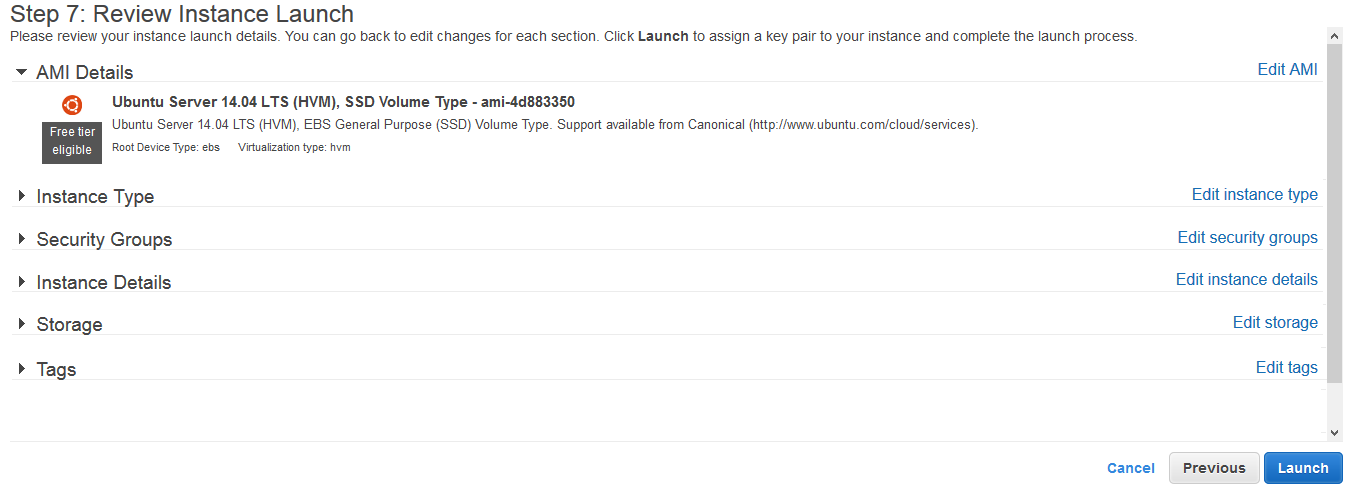
* *Type*: *Custom TCP Rule*;
* *Protocol*: TCP (valor *default* quando a coluna *Type* foi setada como *Custom TCP Rule*);
* *Port Range*: 3000;
* *Source*: *Anywhere* e 0.0.0.0/0 (Este último é *default*, mas somente quando é selecionado *Anywhere*);

Após feitas estas configurações, a conexão está liberada para o acesso de qualquer pessoa que queira desfrutar do sistema desta pesquisa. Também foram adicionados um nome e uma descrição para o S*ecurityGroup* criado. A Figura 8 ilustra como foram feitas as configurações posteriormente mencionadas.



**Figura 8 -**Tela de configuração do *SecurityGroup*. **Fonte:** Elaborado pelos autores.

O último passo é a revisão das informações da instância. Tudo o que foi previamente configurado pode ser revisto, e, caso necessite de alguma alteração, qualquer passo da criação pode ser refeito. Caso as informações da instância estejam em ordem, basta clicar no botão *Launch*, que a instância será criada. Desse modo, a Figura 9 demonstra a tela de revisão de configurações.

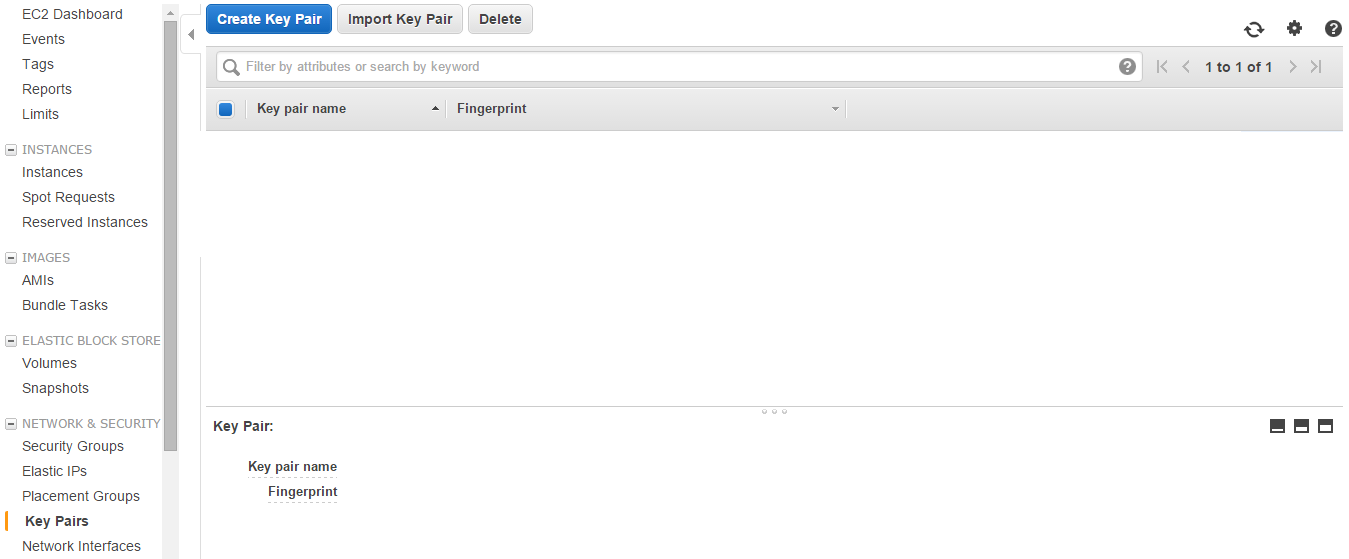


**Figura 9 -**Tela de revisão da instância. **Fonte:** Elaborado pelos autores.

Com a instância devidamente criada, o próximo passo é instalar os requisitos de ambiente necessários para fazer a hospedagem do sistema desta pesquisa.

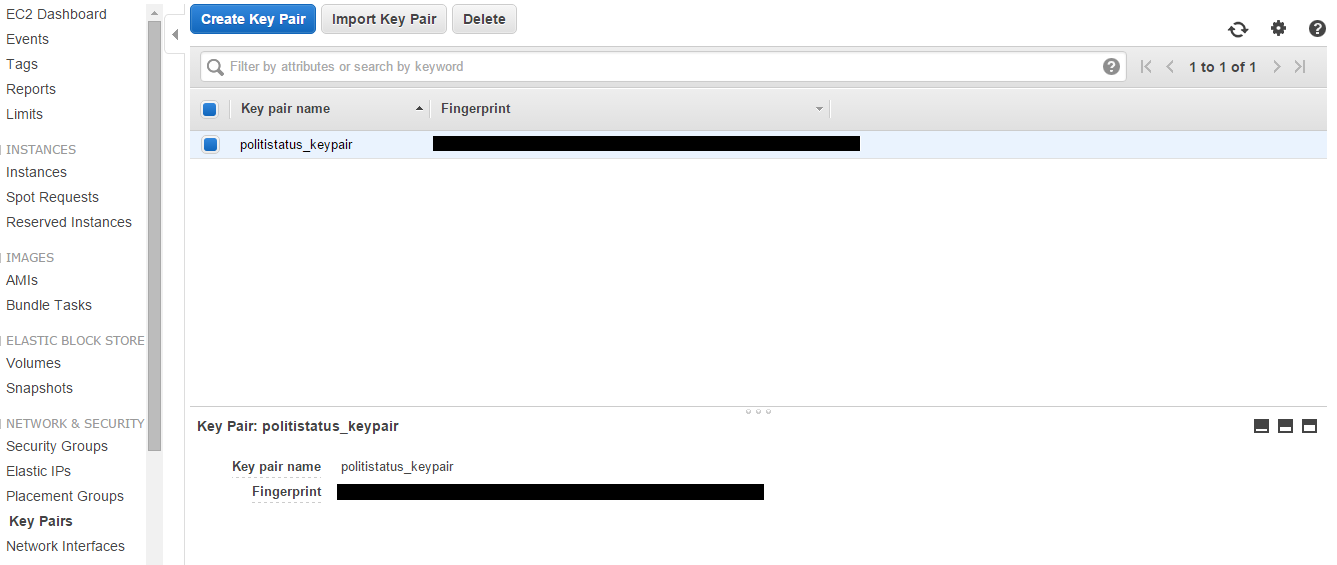
## Preparação do ambiente para o *deploy*

Para hospedar o sistema desenvolvido na instância da Amazon, é necessário instalar todos os pré-requisitos de ambiente, do mesmo modo que foi feito no ambiente de desenvolvimento. Porém, para que isso seja feito, é necessário fazer a conexão remonta com a instância. Para fazer esta conexão, a Amazon *Web Services* disponibiliza a criação de uma *Key Pair*, que basicamente é um arquivo que funciona como uma chave de acesso para a instância criada. Para criá-la, basta ir à *dashboard* do serviço EC2 e clicar em *Key Pairs*. Essa tela irá conter botão *Create Key Pair*, que por sua vez, irá fazer a criação da *Key Pair*. A Figura 10 ilustra a *dashboard* do serviço EC2, bem como a tela de criação de uma nova *Key Pair*.



**Figura 10 -**Tela de criação da *Key Pair*. **Fonte:** Elaborado pelos autores.

Ao clicar no botão *Create Key Pair*, é aberta uma pequena modal de somente 1 campo para preenchimento, sendo necessário informar somente o nome desejado para a *Key Pair*. Neste projeto, o nome utilizado foi politistatus\_keypair. Logo após a sua criação, é feito o *download* automático da *Key Pair*, por meio de um arquivo com a extensão “.pem”. Este arquivo contém a chave de acesso da instância, e será usado para realizar a conexão remota. A Figura 11 ilustra a *Key Pair* já criada.



**Figura 11 -***Key Pair* criada. **Fonte:** Elaborado pelos autores.

Com a chave de acesso já criada, o próximo passo foi fazer a conexão remota com a instância e preparar o ambiente para o *deploy*. Nesta pesquisa, a conexão remota foi feita por meio do software PuTTY, um cliente SSH *free* para Windows. Nele, foi feita a importação da *Key Pair* para a conexão com a instância.

Desse modo, por meio da conexão remota, foi possível preparar o ambiente para *deploy*. A preparação do ambiente de *deploy* é idêntica a preparação do ambiente de desenvolvimento, já que os pré-requisitos são os mesmos. Desse modo, a instalação e configuração dos pré-requisitos de sistema são demonstradas e detalhadas no Apêndice I.

Após os pré-requisitos de sistema estarem prontos, a instância estava pronta para hospedar o sistema desta pesquisa. Porém, viu-se a necessidade de criar um domínio público para facilitar o acesso para os usuários. Para tal, foi criado o domínio “www.politistatus.tk” no site DotTK. Todos os domínios criados no Dot TK são gratuitos até 1 ano de uso.

Ao acessar o domínio “www.politistatus.tk” no navegador, é feito o redirecionamento automático para IP da instância na Amazon, mais especificamente na porta 3000, que foi previamente liberada na *Security Group*. Desse modo, o domínio sempre irá apontar para a página de *login* da aplicação web. Essas configurações de redirecionamento foram feitas nas próprias configurações do domínio Dot TK.

Com todos os pré-requisitos devidamente configurados, o último passo da hospedagem foi o *deploy* da aplicação.

## *Deploy*

O *deploy* da aplicação é a última fase da hospedagem do sistema. A intenção é tornar o sistema público, para que os usuários desfrutem de suas funcionalidades.

O primeiro passo para fazer o *deploy* foi instalar o WinSCP, um cliente FTP para Windows. Após instalado, o WinSCP importou as configurações de acesso do PuTTY, facilitando assim a conexão com a instância. Ao iniciar o WinSCP, foi mostrado a árvore de diretórios da instância. Para facilitar o acesso aos arquivos do sistema, foi criada a pasta “politistatus” no diretório “/home/ubuntu”, onde posteriormente, foi feito o *deploy* do sistema desta pesquisa.

Dentro do diretório “/home/ubuntu/politistatus” existem duas pastas, a “Big Data Framework”, onde está o JAR gerado do *Apache Storm*, e a “Web Application”, com os arquivos da aplicação web.

Com os arquivos já deployados na instância, o último passo a ser tomado era executar o servidor web e o *framework* de mineração dos dados, que, posteriormente, varia a gerar o *Big Data* da pesquisa. Para tal, foi criado um *script* de inicialização Linux, colocado no diretório “/etc/init.d/”. Seu conteúdo é o seguinte:

* #!/bin/bash
* sudo npm start ~/politistatus/Web\ Application/ &
* sudo java -jar ~/politistatus/Big\ Data\ Framework/Storm.jar Consumer Key Consumer\_Secret Access\_Token Access\_Token\_Secret &
* **Observação:** As credenciais de acesso *Consumer Key*, *Consumer Secret*, *Access Token* e *Access Token Secret* são únicas e de uso exclusivo, então, foram colocadas neste trabalho de forma representativa.

O *script* permite que, ao iniciar a instância, sejam também iniciados o servidor web e o *framework* Apache Storm. Ambos são colocados em *background*, devido ao “&” inserido ao final da linha de cada comando.

Para finalizar, foram dadas as permissões de execução para o *script*, por meio do comando “sudo chmod +x /etc/init.d/initScript.sh”, e ele foi colocado na inicialização do sistema da instância, por meio do comando “sudo update-rc.d /etc/init.d/initScript.sh defaults”. Após estes passos, foi reiniciada a instância, que já iniciou o servidor web, bem como o sistema de mineração de dados Apache Storm. A Figura 12 ilustra a página de *login* da aplicação web, já acessada por meio do domínio público “www.politistatus.tk”.



**Figura 12 -**Página de *login* da aplicação web, acessada pelo domínio politistatus.tk. **Fonte:** Elaborado pelos autores.

Após todos os passos descritos nas seções anteriores, o sistema já se encontrava *online* e disponível para o acesso na internet. Para iniciar o acesso à aplicação, basta que um usuário se cadastre e faça o *login*. Para minerar os *tweets*, o usuário terá que cadastrar as palavras-chave desejadas, fazendo assim com que o Apache Storm crie sua topologia de busca com base nas palavras-chave cadastradas. Este fluxo é demonstrado com detalhes na seção 3.4.8 do Quadro Metodológico.

1. A escola e o futuro da biblioteca de pesquisa. [↑](#footnote-ref-1)
2. Os usuários podem realmente absorver dados nas taxas de hoje? Amanhã? [↑](#footnote-ref-2)
3. Paginação por demanda controlada por aplicativo para visualização externa. [↑](#footnote-ref-3)
4. Explorar visualmente conjuntos de dados gigabyte em tempo real.

   Exabyte - Unidadede medida computacional. [↑](#footnote-ref-4)
5. Quanta informação? [↑](#footnote-ref-5)
6. Modelos de fator dinâmico de Big Data para medição e previsão macroeconômica. [↑](#footnote-ref-6)
7. Gestão de dados 3D: controlar o volume de dados, velocidade e variedade. [↑](#footnote-ref-7)
8. A expansão do universo digital: uma previsão de crescimento da informação Worldwide até 2010. [↑](#footnote-ref-8)
9. Computação Big-Data: Criando avanços revolucionários no comércio, ciência e sociedade. [↑](#footnote-ref-9)
10. Dados, dados em toda parte. [↑](#footnote-ref-10)
11. Big data: a próxima fronteira para a inovação, concorrência e produtividade. [↑](#footnote-ref-11)
12. Termo usado para definir o conceito de armazenamento dos dados relativos às atividades de uma organização. [↑](#footnote-ref-12)
13. Questões críticas para Big Data em Informação, Comunicação e Sociedade. [↑](#footnote-ref-13)
14. Estrutura de dois ou mais computadores para que estes trabalhem de maneira conjunta no intuito de processar uma tarefa. [↑](#footnote-ref-14)
15. *Java Virtual Machine* - Máquina Virtual Java. [↑](#footnote-ref-15)
16. Termo inglês usado para designar o pensamento ou gosto atual da maioria da população. [↑](#footnote-ref-16)
17. Conjunto de instruções e padrões de programação para acesso a um aplicativo baseado na web. [↑](#footnote-ref-17)
18. Solução de integração de sistemas, bem como na comunicação entre aplicações diferentes. [↑](#footnote-ref-18)
19. Tecnologia de processamento de requisições HTTP. [↑](#footnote-ref-19)
20. Publicação de um usuário do *Twitter*. [↑](#footnote-ref-20)
21. Arquivo compactado usado para distribuir um conjunto de classes ou um aplicativo Java, entre outros. [↑](#footnote-ref-21)
22. Protocolo aberto que permite uma autorização segura com um método simples e padronizado para aplicações web, *mobile* e *desktop*. [↑](#footnote-ref-22)
23. Abreviação de GNU zip, um Software Livre de compressão sem perda de dados. [↑](#footnote-ref-23)
24. *Extensible Markup Linguage* - Linguagem de Marcação Extensiva. [↑](#footnote-ref-24)
25. *JavaScript Object Notation* - Notação de Objetos *JavaScript*. [↑](#footnote-ref-25)
26. Banco de dados criado pelo Google. [↑](#footnote-ref-26)
27. Serviço de banco de dados NoSQL. [↑](#footnote-ref-27)
28. *Structured Query Language* - Linguagem de consulta estruturada, usada em bancos de dados. [↑](#footnote-ref-28)
29. Linguagem baseada em *scripts*. [↑](#footnote-ref-29)
30. Sistema operacional. [↑](#footnote-ref-30)
31. *American Standard Code for Information Interchange* - Codificação usada para representar textos em computadores, equipamentos de comunicação, entre outros dispositivos que trabalham com texto. [↑](#footnote-ref-31)
32. Termo da computação que significa uma lista que armazena uma coleção de elementos. [↑](#footnote-ref-32)
33. Input/Output – Entrada/Saída. [↑](#footnote-ref-33)
34. Gargalos. [↑](#footnote-ref-34)
35. Abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. [↑](#footnote-ref-35)
36. Empresa multinacional de comércio eletrônico. [↑](#footnote-ref-36)
37. Organização europeia de pesquisas nucleares. [↑](#footnote-ref-37)
38. Ligações, referências. [↑](#footnote-ref-38)
39. *Cascading Style Sheets*, formata a informação entregue pelo HTML [↑](#footnote-ref-39)
40. Linguagem de programação interpretada pelo navegador. [↑](#footnote-ref-40)
41. Toda parte incluída na apresentação visual de um site. [↑](#footnote-ref-41)