**NÍCOLAS HENRIQUE VIEIRA TOLEDO**

**SEBASTIÃO BATISTA DE ANDRADE NETO**

**ESTATÍSTICAS DE REDES SOCIAIS**

**COM BIG DATA**

**UNIVERSIDADE DO VALE DO SAPUCAI**

**POUSO ALEGRE**

**2015**

# 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

A pesquisa tem como objetivo a obtenção de novos conhecimentos, tomando por base a utilização de procedimentos científicos. 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, foi desenvolvida uma aplicação web que tem como objetivo a mineração, análise e demonstração dos posts do *Twitter*, além do objetivo de demonstrar como foi feita integração das tecnologias utilizadas para análise de *Big Data*.

## Contexto de pesquisa

Atualmente, devido à facilidade de acesso à internet, dia após dia são geradas quantidades imensas de 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.

A esta pesquisa demonstra como soluções baseadas em *Big Data* tem o poder de influenciar e auxiliar determinadas áreas de conhecimento do ser humano, através 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 de redes sociais de forma rápida e clara, além da geração de gráficos para ser feita a análise de satisfação pública por região. 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 que 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 foram apresentados ao orientador os protótipos da 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 no 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

Para o desenvolvimento desta pesquisa, foi necessário configurar o 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.

O 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, habilitando, assim, seus comandos via terminal, conforme demonstrado na Figura 5.

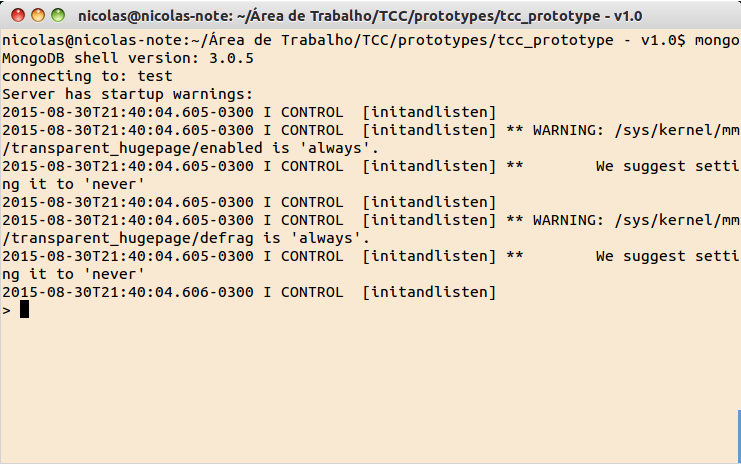
****

**Figura 5** - Demonstração da versão do Node.JS e NPM usadas, respectivamente. **Fonte:** Elaborado pelos autores.

Para se fazer real o armazenamento da massa de dados, 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, são mostrados na Figura 6.

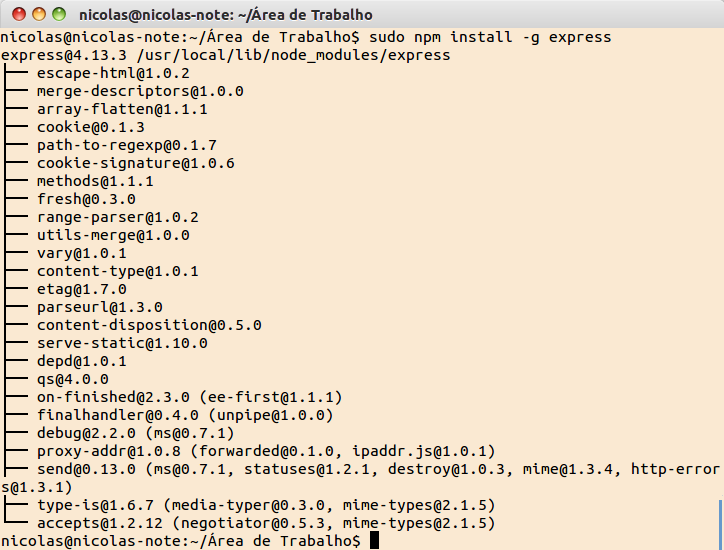
****

**Figura 6** - Terminal do MongoDB. **Fonte:** Elaborado pelos autores.

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.

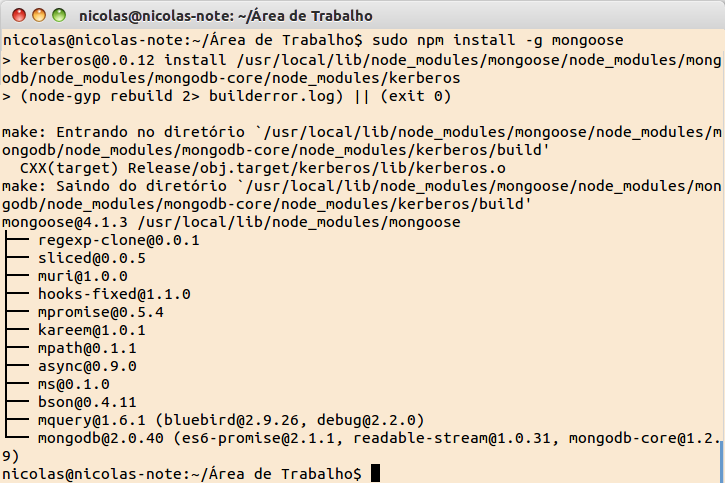
Posteriormente, foi executada a linha de comando “~/Documents/WebStorm-10.0.0/bin/WebStorm.sh”, que consistia na execução de um *script* Linux para a instalação e configuração da IDE.

Logo após a escolha e instalação da IDE de desenvolvimento, foi feita a instalação e configuração do *framework* *Express* através da linha de comando “npm install –g express” no terminal do Node.JS, instalando-o assim como um módulo global através do parâmetro “-g”, conforme mostrado na Figura 7.



**Figura 7** - 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 do comando “npm install –g mongoose”, também como módulo global, conforme demonstrado na Figura 8.



**Figura 8** - Demonstração da instalação do *Mongoose*. **Fonte:** Elaborado pelos autores.

Depois de todos os pré-requisitos, viu-se a necessidade de criar a arquitetura do sistema, visando melhor organização e respeito de padrões. 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*:** Arquivo 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*, *author*, informando respectivamente o nome, a versão, a descrição e os autores da aplicação. Também foi setado o atributo *private* 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 com o comando “npm”. A estrutura do arquivo é demonstrada na Figura 9.

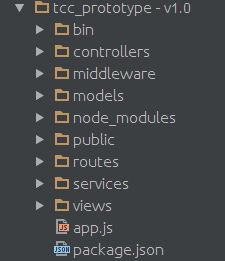


**Figura 9** - Demonstração do arquivo *package.json*. **Fonte:** Elaborado pelos autores.

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:

* ***Controller*:** Diretório responsável pelos arquivos controladores das *Views*;
* ***Model*:** Diretório responsável pelos arquivos de *persistence* dos modelos do banco de dados;
* ***Service*:** 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 10.

****

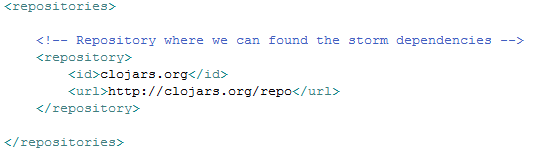
**Figura 10** - Demonstração da árvore de diretórios da aplicação. Fonte: Elaborado pelos autores.

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*.

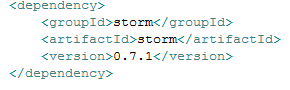
Após feito o *download*, extração e a movimentação da pasta do Eclipse para o diretório “/Documents”, foi necessária a instalação do plug-in de integração do *Maven* “m2e – *Maven Integration for Eclipse*”. Foi necessária também a instalação do *Maven2* através da linha de comando “sudo apt-get install maven2” no terminal do sistema. Desse modo, a instalação e configuração foram automáticas.

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 na Figura 11.



**Figura 11** - Adição da dependência *CloarJS.org* no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

A integração do *Apache Storm* também foi feita através do *Maven*. No arquivo “pom.xml” foram adicionadas as linhas de código demonstradas na Figura 12:



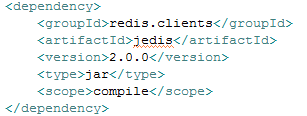
**Figura 12** - Adição da dependência do *Apache Storm* no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

Também através do *Maven*, foi feita a integração do *Commons-*HTTP. No arquivo “pom.xml” foram adicionadas as linhas de código demonstradas na Figura 13:



**Figura 13** - Adição da dependência *Commons-*HTTP no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

Do mesmo modo, foi feita integração do *Redis.clients*. No arquivo “pom.xml” foram adicionadas as linhas de código demonstradas na Figura 14:



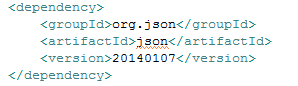
**Figura 14** - Adição da dependência *redis.clients* no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

A integração da biblioteca *Twitter4j* foi feita do mesmo modo. Também no arquivo “pom.xml”, adicionou-se as linhas de código demonstradas na Figura 15:



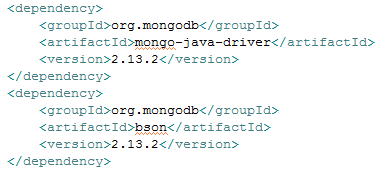
**Figura 15** - Adição da dependência da biblioteca *Twitter4j* no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

Do mesmo modo foi feita a integração do Org.JSON, demonstrada na Figura 16:



**Figura 16** - Adição da dependência Org.JSON no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

A integração do MongoDB e seu driver de acesso também foi feita também através do *Maven*. No arquivo “pom.xml” foram adicionadas as linhas de código demonstradas na Figura 17:



**Figura 17** - Adição das dependênncias banco de dados MongoDB no arquivo “pom.xml”. **Fonte:** Elaborado pelos autores.

Após serem feitas as configurações das dependências, executou-se o comando “maven install” no Eclipse 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*. 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.

Também inserido na página inicial da aplicação, está um mapa constantemente atualizado, que informa através de pontos ao usuário da aplicação quais são os locais do Brasil onde estão sendo realizados os *tweets*.

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.

O desenvolvimento dos protótipos do sistema foi uma etapa essencial para a elaboração do projeto. Através dela, foram adquiridos os conhecimentos necessários para desenvolver a versão final da aplicação, além de ser feita a comprovação que interação entre as tecnologias do projeto tornam possível a realização 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 *Apache Storm*. Os dois sistemas não têm relação entre si, 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*.

O *back-end* da aplicação é responsável por atender todas as requisições do usuário. Toda requisição feita pelo usuário no *front-end* chegam obrigatoriamente no *back-end*. Porém foram necessárias algumas configurações para que essa 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 na Figura 18.

****

**Figura 18** - Chamada dos módulos na *stack*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 1** - Módulo *Express* é carregado na variável express. Desse modo, os recursos módulo, 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;

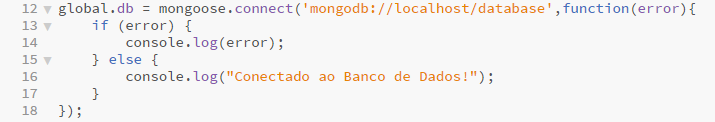
**Linha 7** - Carregamento do módulo *socket.io*, posteriormente chamando a sua função *listen* e passando como parâmetro o servidor HTTP recém-criado. O módulo *socket.io* foi adicionado à aplicação devido a necessidade de existência de uma conexão *socket* da mesma, para ser feito envio dos *tweets* para o *front-end* no tempo de sua ocorrência;

**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 na Figura 19.



**Figura 19** - Conexão com o Banco de dados. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 12** - Declaração da variável global *db*. Ela foi declarada como global para poder estar disponível em toda aplicação. Posteriormente, foi chamado o método *connect* da 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 na Figura 20.



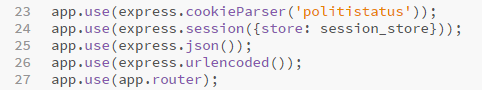
**Figura 20** - Configuração de diretórios. **Fonte:** Elaborado pelos autores.

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 a Figura 21.



**Figura 21** - Configurações da variável app. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

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

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

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

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

**Linha 27** - Setado 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 a Figura 22.



**Figura 22** - Configurações estáticas e de tratamento de erro da aplicação. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

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

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

**Linha 31** - Setado 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 na Figura 23.



**Figura 23 -** 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 na Figura 24.

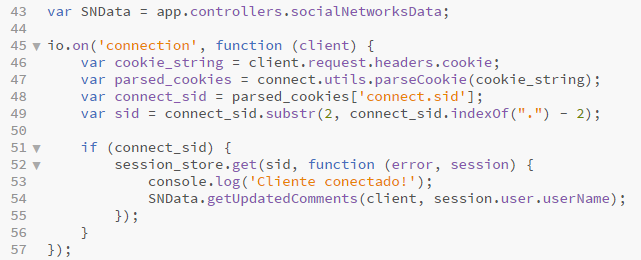


**Figura 24 -** Inicialização do servidor na porta 3000. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 39 a 41** - O servidor é iniciado na porta 3000, através do método listen(). Também 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 de uma conexão *socket* ativa para enviar os *tweets* minerados pelo *framework* *Apache Storm* para o *front-end* da aplicação web. Essa configuração foi feita após a inicialização do servidor, conforme mostra a Figura 25.



**Figura 25 -** Configuração da conexão *socket*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

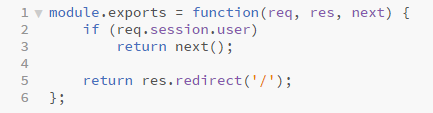
**Linha 43** - É criada a variável *SNData*, que recebe como valor o *Controller* de busca de *tweets* da aplicação;

**Linhas 46 a 49** - 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 cada usuário tem sua própria lista de palavras-chave;

**Linhas 51 a 56** - É feita uma verificação para validar se o usuário que requisitou *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 *getUpdatedComments* do *Controller* de busca dos *tweets*, sendo passados como parâmetro a conexão *socket*  e o nome do usuário da sessão.

Após todas estas configurações na *stack*, o servidor estava pronto para atender os requisitos da aplicação web.

A próxima ação a ser tomada logo após o desenvolvimento da *stack*, era 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*. A Figura 26 demonstra o *middleware* de autenticação de usuário.



**Figura 26 -** Autenticação de usuário via *middleware*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 1 até 6** - 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.

A Figura 27 demonstra o *middleware* de tratamento de páginas de erro.



**Figura 27 -** 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 era 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.

Na Figura 28 é demonstrado um exemplo de arquivo contendo suas respectivas rotas.



**Figura 28 -** Exemplo de um arquivo de rotas. **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** - 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 contêm respectivamente os dados da requisição e os dados para o retorno da chamada.

Com base no que foi mencionado anteriormente, a Figura 29 demonstra um trecho de código de um *Controller* da aplicação.



**Figura 29 -** Trecho de código exemplificando um *Controller*. **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 *Politics*, recebendo como valor o *Service* de *Politics*;

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

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

**Linha 11 -** Início do bloco *try*;

**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*;

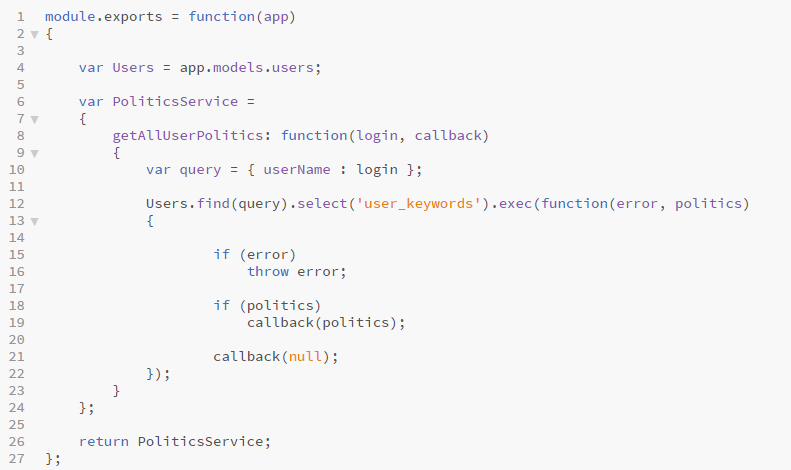
**Linha 30 -** O objeto *res* é chamado passando como parâmetro o objeto JSON previamente montado convertido em *String*. O objeto *res* irá se encarregar de fazer o retorno da requisição do *front-end*;

**Linha 35 -** Final do bloco *try* e início do bloco *catch*;

**Linhas 37 e 38 -** Caso a busca no *Service* retorne alguma *exception*, o erro é mostrado no terminal e posteriormente é mandado um *response* com 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 na Figura 30.



**Figura 30 -** Trecho de código exemplificando um *Service* e seu método de 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*;

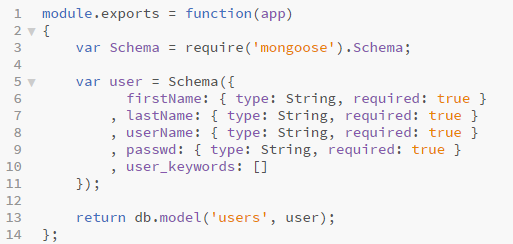
**Linha 10 -** Criada a variável *query*, a qual irá conter as condições de busca no banco de dados;

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

**Linhas 15 a 21 -** 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 26 -** 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 do banco de dados. Seguindo este contexto, a Figura 31 exemplifica como foi o padrão de desenvolvimento dos *Models* da aplicação.



**Figura 31 -** 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;

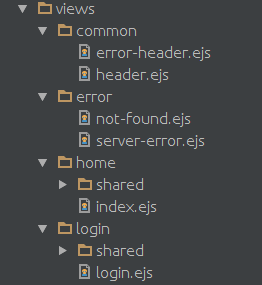
**Linhas 5 a 11 -** Geração de um modelo. É este objeto que será usado como modelo na persistência dos dados;

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

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

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 32 demonstra a árvore de diretórios as *Views*.

****

**Figura 32 -** Árvore de diretórios da *View*. **Fonte:** Elaborado pelos autores.

No diretório c*ommon* da estrutura, estão os arquivos comuns entre as *Views* da aplicação, os quais contém as chamadas dos *JavaScript’s* 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 na Figura 33.

****

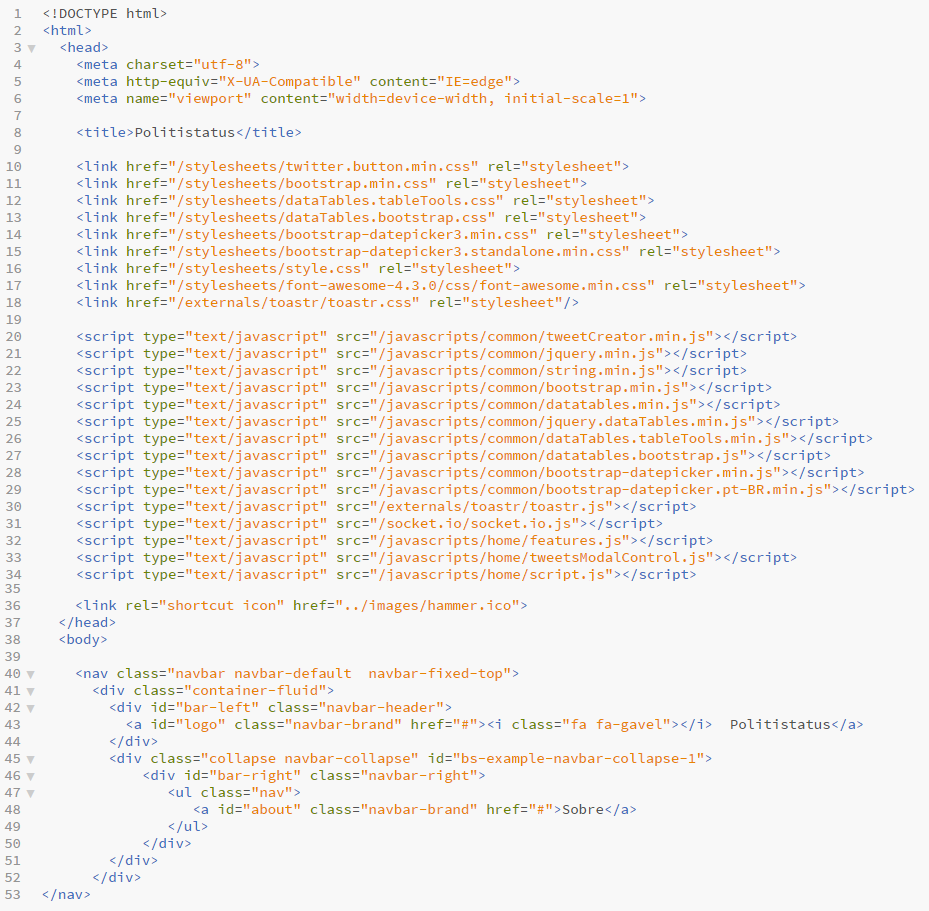
**Figura 33 -** *View* da página inicial. **Fonte:** Elaborado pelos autores.

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 41 -** Conteúdo da página HTML.

O *Header* da aplicação carrega os *JavaScripts* e *stylesheets* para a página HTML. A Figura 34 exemplifica a estrutura do *Header*.



**Figura 34 -** Exemplo de *Header*. **Fonte:** Elaborado pelos autores.

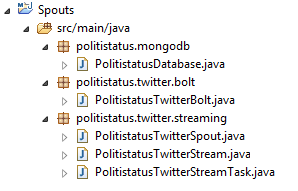
Também dentro dessa estrutura de diretórios, estão as subpastas *shared*. Dentro destas subpastas estão os arquivos referentes às *Partial Views* da aplicação. As *Partial Views* são as estruturas HTML que posteriormente serão convertidas nos modais da aplicação web, via *JavaScript*. A Figura 35 exemplifica a estrutura de uma *Partial View*.

****

**Figura 35 -** Exemplo de uma *Partial View*. **Fonte:** Elaborado pelos autores.

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 fazer a mineração dos dados, a qual é feita pelo *framework* *Apache Storm*.

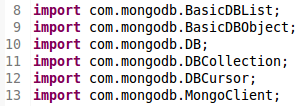
O *framework* de mineração de dados *Apache Storm*, constitui a parte responsável por buscar os dados referentes a postagens na rede social *Twitter*, e é dividido em quatro classes, conforme mostrado na Figura 36.



**Figura 36 -** Classes do *framework Apache Storm*. **Fonte:** Elaborado pelos autores.

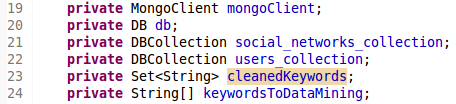
A primeira ação a ser tomada em relação ao desenvolvimento do *framework*, era o desenvolvimento de sua classe conexão com o MongoDB. Essa classe do *framework* é responsável por fazer as operações relacionadas ao banco de dados.

A Figura 37 demonstra os *imports* necessários para ser feita a comunicação com o MongoDB.



**Figura 37 -** *Imports* da classe *PolitistatusDatabase*. **Fonte:** Elaborado pelos autores.

A Figura 38 demonstra os atributos da classe *PolitistatusDatabase.*



**Figura 38 -** Atributos da classe *PolitistatusDatabase*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 19** - Cria o cliente para o MongoDB*;*

**Linha 20** - Cria um objeto *db* para receber a *collection* do banco de dados, permitindo acesso pela classe;

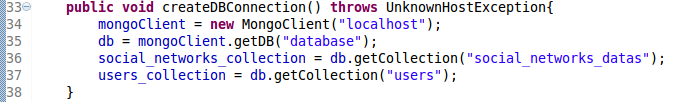
**Linha 21** - Cria uma *DBCollection* específica para salvar as informações mineradas das redes sociais;

**Linha 22** - Cria uma *DBCollection* específica para salvar as informações sobre os usuários;

**Linha 23** - Cria um *Array Set* do tipo *String* para posteriormente ser feito a exclusão das palavras-chave repetidas;

**Linha 24** - cria um vetor de *String* para receber diversas palavras-chaves a serem mineradas.

O método demonstrado na Figura 39 cria a conexão com o banco de dados.

****

**Figura 39 -** Criação da conexão com o MongoDB. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

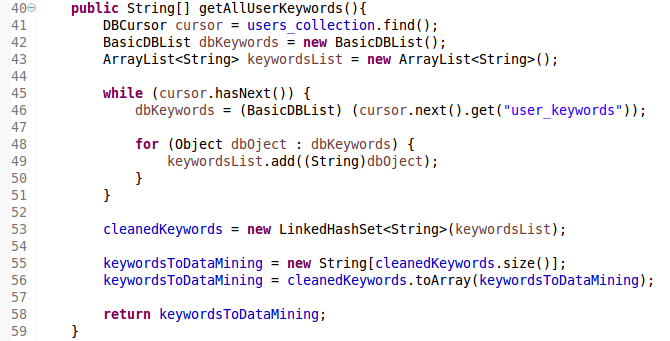
**Linha 34** - É informado o endereço da máquina onde o serviço de banco de dados está instalado, no caso a própria máquina é o servidor de banco de dados (*localhost)*;

**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* campos com nome *social\_networks\_datas*;

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

O método demonstrado na Figura 40, quando acionado, pega todas as palavras-chave dos usuários através da tabela *user\_keywords* do banco de dados.



**Figura 40 -** 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 de *users\_collection* as *keywords* cadastradas pelos usuários da aplicação web;

**Linha 42** - O objeto *dbKeywords* do tipo *BasicDBList* é criado;

**Linha 43** - O objeto *keywordsList* do tipo *ArrayList* de *String* é criado;

**Linhas 45 a 51** - Inicia-se um *while* para ser feita a consulta das palavras-chave no banco de dados. O objeto *dbKeywords*, posteriormente, recebe o retorno do mesmo. Posteriormente, é feito um *foreach* para ir adicionando as palavras-chave no *array* *keywordsList*;

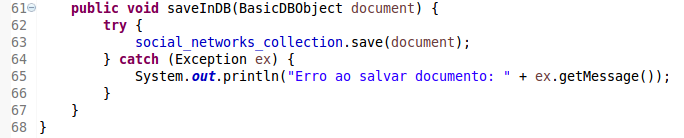
**Linha 53** - O objeto *cleanedKeywords* é instanciado, sendo passado como parâmetro o *array* *keywordsList*. Desse modo, o objeto *keywordsList* é mapeado para o objeto *cleanedKeywords*, ficando assim, somente as palavras-chave não repetidas;

**Linha 55** - O *array* de *string keywordsToDataMining* é instanciado recebendo como parâmetro o tamanho do objeto *cleanedKeywords;*

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

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

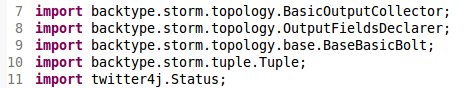
É demonstrado na Figura 41 o método responsável por salvar um objeto do tipo *BasicDBObject* no banco de dados. Caso o método *save* apresente um erro, é retornada uma *exception* que será posteriormente mostrada no terminal.



**Figura 41 -** Método responsável por para salvar um documento no banco de dados. **Fonte:** Elaborado pelos autores.

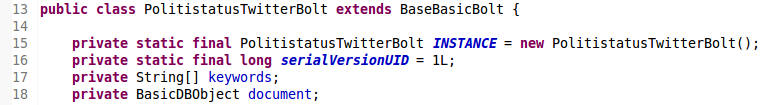
Logo após o desenvolvimento da classe responsável por fazer as operações com o banco de dados, era necessário construir classe *Bolt*, a qual é responsável por transformar uma tupla (linha de dados) recebida da classe *PolitistatusTwitterSpout* em outra tupla que contém dados processados para salvar no banco de dados. As Figuras 38 e 39 apresentam o conteúdo da classe.

As linhas da Figura 42 importam as classes necessárias para a o funcionamento da classe *PolitistatusTwitterBolt*.



**Figura 42 -** *Imports* da classe *PolitistatusTwitterBolt*. **Fonte:** Elaborado pelos autores.

As linhas da Figura 43 demonstram os atributos da classe.



**Figura 43 -** Atributos da classe *PolitistatusTwitterBolt*. **Fonte:** Elaborado pelos autores.

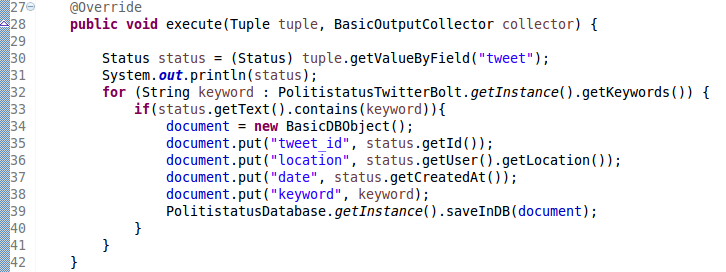
As funções de cada linha são detalhadas abaixo:

**Linha 15** - Instancia *PolitistatusTwitterBolt* como *final*;

**Linha 17** - Cria um vetor do tipo *string* com o nome de *keywords*;

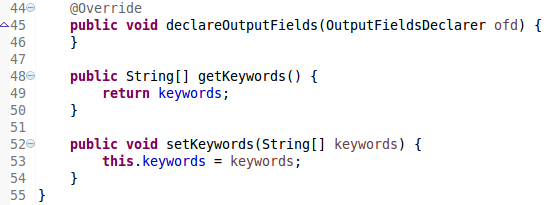
**Linha 18** - Cria novo objeto do tipo *BasicDBObject* com o nome *document*.

O método da Figura 44 executa a função primária da classe *PolitistatusTwitterBolt*, separando os dados em *tweet\_id*, *location*, *date* e *keyword*. Neste momento, é montado o objeto, posteriormente chamando o método *saveInDB* da classe *PolitistatusDatabase*.



**Figura 44 -** Método execute da classe *PolitistatusTwitterBolt*. **Fonte:** Elaborado pelos autores.

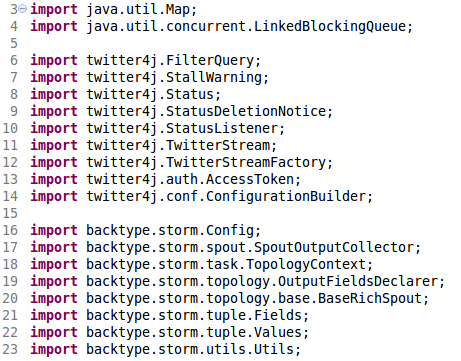
Os métodos *declareOutputFields* sobrescreve o método na classe mãe, os métodos *getKeywords setKeywords* são para tornar visíveis as variáveis privadas (*getters).* Ambos são demonstrados na Figura 45.



**Figura 45 -** Métodos *declareOutputFields*, *getKeywords* e *setKeywords*. **Fonte:** Elaborado pelos autores.

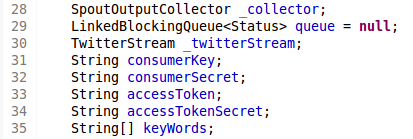
A próxima ação a ser tomada depois da criação do *Bolt*, era a criação do *Spout*. Basicamente, o *Spout* é esta classe a responsável pela a comunicação entre o emissor de mensagens e a parte interna da topologia.

As linhas da Figura 46 demonstram os *imports* necessários para a o funcionamento da classe *PolitistatusTwitterSpout*.



**Figura 46 -** Imports da classe *PolitistatusTwitterSpout*. **Fonte:** Elaborado pelos autores.

As Linhas de código da Figura 47 alocam variáveis necessárias para o funcionamento da classe.



**Figura 47 -** Atributos da classe *PolitistatusTwitterSpout*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 28** - Cria a variável *\_collector* do tipo *SpoutOutputCollector* que é responsável pela emissão de tuplas para envio ao bolt;

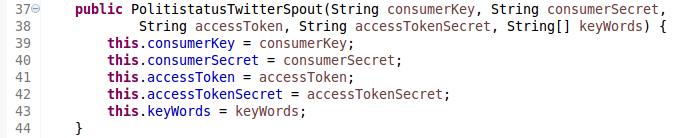
**Linha 29** - É criada a variável *queue* do tipo *LinkedBlockingQueue*<*Status*> que armazena o estado da fila de tuplas;

**Linha 30** - Cria a variável \_*twitterStream* do tipo *TwitterStream*  que é responsável pelo fluxo de dados entre o *Spout* e o emissor de mensagens (*Twitter Stream API*);

**Linhas 31 a 34** - São criadas as variáveis responsáveis por receber os campos dos *twetts* provenientes do emissor;

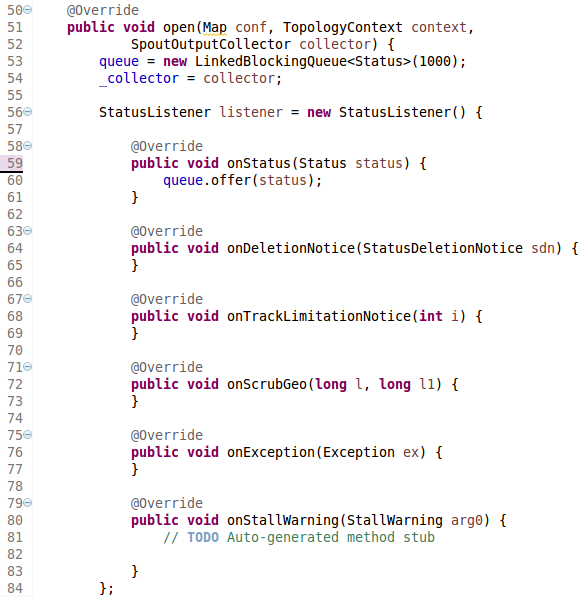
**A Linha 35** - É um vetor de *String* que armazena as palavras chave.

A Figura 48 demonstra o construtor da classe *PolitistatusTwitterSpout*.



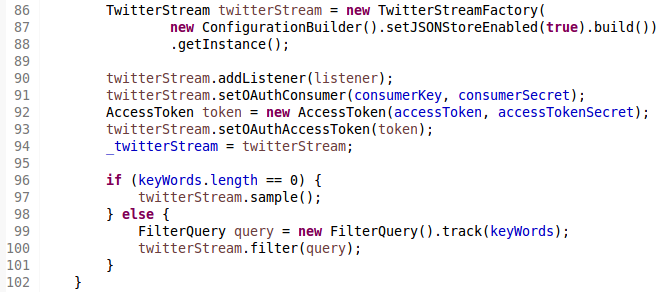
**Figura 48 -** Construtor da classe *PolitistatusTwitterSpout*. **Fonte:** Elaborado pelos autores.

A Figura 49 demonstram os *Overrides* provenientesda biblioteca *Twitter4j*,oferecendo personalização para funções específicas.



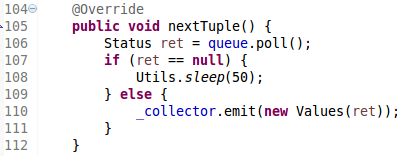
**Figura 49 -** *Overrides* da classe *PolitistatusTwitterSpout*. **Fonte:** Elaborado pelos autores.

A Figura 50 demonstra a criação do *twitterStream*, que é o próprio fluxo de dados que partem do emissor para o *Spout*.



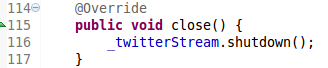
**Figura 50 -** Criação do *Twitter Stream*. **Fonte:** Elaborado pelos autores.

A Figura 51 demonstra o método *nextTuple*, que lê as tuplas uma de cada vez, para posteriormente envia-las ao *Bolt*.



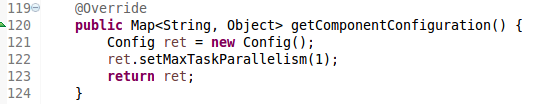
**Figura 51 -** Método de leitura das tuplas. **Fonte:** Elaborado pelos autores.

A Figura 52 demonstra o método *close*, que encerra a conexão com o emissor (*Twitter Stream API*).



**Figura 52 -**Método de encerramento de conexão. **Fonte:** Elaborado pelos autores.

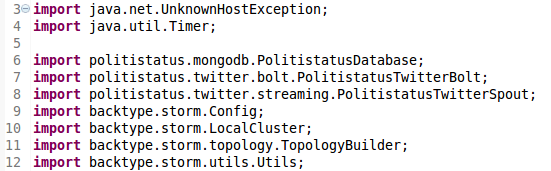
A Figura 53 demonstra o método *getComponentConfiguration*, que recebe as configurações de funcionamento e define o máximo de tarefas paralelas suportadas



**Figura 53 -**Método configuração de tarefas. **Fonte:** Elaborado pelos autores.

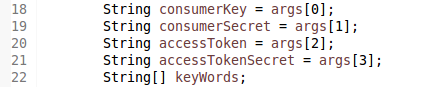
Após os *Spouts* estarem configurados, era necessário criar uma classe que passasse os parâmetros de requisição para o *Twitter Stream API*, além de executar o *framework* para minerar os dados. A partir destas necessidades, foi criada a classe *PolitistatusTwitterStream*.

A Figura 54 demonstra os *imports* da classe *PolitistatusTwitterStream.*



**Figura 54 -***Imports* da classe *PolitistatusTwitterStream*. **Fonte:** Elaborado pelos autores.

A Figura 55 demonstra os atributos que receberão as chaves de autorização da aplicação criada no *Twitter Developer*.



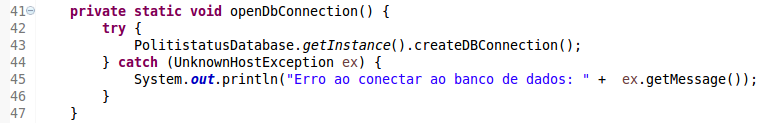
**Figura 55 -**Atributos que serão setados como chaves de acesso. **Fonte:** Elaborado pelos autores.

A Figura 56 demonstra respectivamente a abertura da conexão com o banco de dados e a busca das palavras chave através do método *getAllUserKeywords*, para posteriormente ser salva na variável *keyWords*.

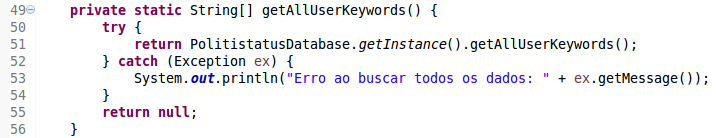


**Figura 56 -**Operações de banco de dados na classe de execução. **Fonte:** Elaborado pelos autores.

As figuras 57 e 58 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 e mais o código do erro. Já *getAllUsersKeywords* busca todas as palavras-chave do usuário e as procura no banco de dados.

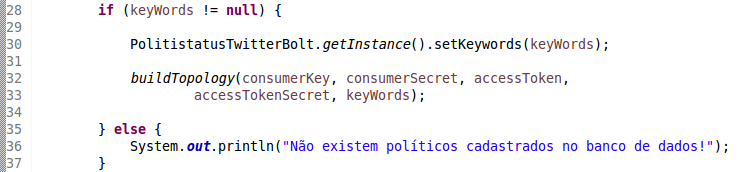


**Figura 57 -**Método *openDbConnection* . **Fonte:** Elaborado pelos autores.



**Figura 58 -**Método *openDbConnection* . **Fonte:** Elaborado pelos autores.

A Figura 59 demonstra a lógica da aplicação após serem executadas as linhas da Figura 56.



**Figura 59 -**Lógica da classe de execução. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linhas 28** **a 33** - Se a variável *keywords* não for nula, será enviado ao *bolt* o conteúdo da variável *keywords*, criando a topologia e enviando as variáveis *consumerKey*, *consumerSecret*, *accesToken*, *accesTokenSecret* e *keyWords*.

A Figura 60 demonstra o método *buildTopology*, que contém as configurações necessárias para a construção da topologia do *Storm*.



**Figura 60 -**Método de construção da topologia do *Storm*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

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

**Linha 65** - Setado o *Spout* que busca na internet as palavras chaves;

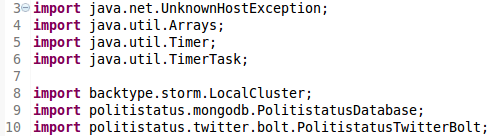
**Linha 67** - Setado o *Bolt* que vai processar as informações recebidas do *Spout*;

**Linha 75** - Criação da topologia do *Storm*.

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

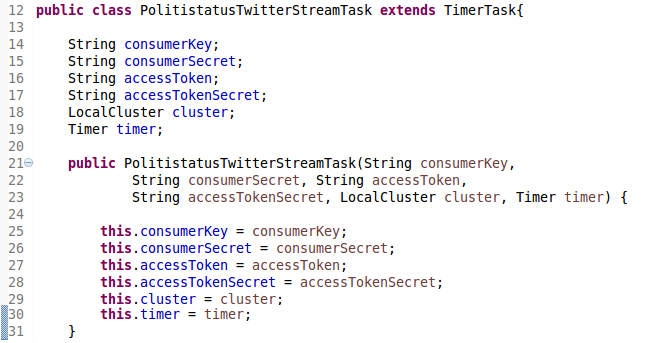
**Linhas 80 e 81** - Setada a classe responsável por manusear a *Task*, sendo passados como parâmetros as informações necessárias para a reconstrução da topologia do *Storm*. A *Task* será executada em um intervalo de tempo de 5 segundos.

A Figura 61 demonstra os *imports* da classe *PolitistatusTwitterStreamTask*. Essa classe é responsável por manusear a *Task* criada na topologia do *Storm*.



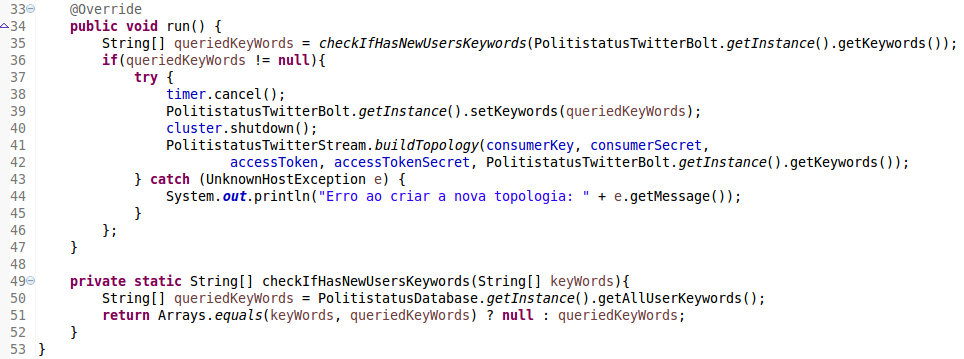
**Figura 61 -***Imports* da classe *PolitistatusTwitterStreamTask*. **Fonte:** Elaborado pelos autores.

A Figura 62 demonstra os atributos e o construtor da classe de *PolitistatusTwitterStreamTask*. Esses parâmetros foram passados para a classe depois da criação da topologia do *Storm*.



**Figura 62 -**Atributos e construtor da classe *PolitistatusTwitterStreamTask*. **Fonte:** Elaborado pelos autores.

Os métodos da Figura 63 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 busca.



**Figura 63 -**Métodos de reconstrução da topologia do *Storm*. **Fonte:** Elaborado pelos autores.

As funções de cada linha são detalhadas abaixo:

**Linha 34** - É criado um *array* de *String* que recebe o retorno da função *checkIfHasNewUsersKeywords*, sendo passado como parâmetro as *keywords* que estão sendo enviadas à *Twitter Stream* API;

**Linhas 35 a 42** - Se o objeto *queriedKeywords* não estiver nulo, é cancelada a *Task*, 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 passando como parâmetro as informações para a conexão com a *Twitter Stream* API juntamente com as novas palavras-chave;

**Linhas 49 a 52** - 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 da mineração dos dados, para a posterior discussão dos resultados.

# REFERÊNCIAS

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

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.

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