Public Political Documents
A Technical Summary

Oscar Ivarsson and Trausti Dagsson

May 12, 2016

Abstract

This paper is a technical summary which covers the process of creating an application for presenting information about public political documents. It involves all parts of the pipeline, spanning from investigating the data source and setting up a tool for extracting and loading the data into a database, to server configuration and API development for presenting data to different clients.

1 Data

1.1 The Swedish Parliament's Open Data

All data used in this project is taken from the open datasets provided by the Swedish Parliament (http://data.riksdagen.se/). The site offers data in terms of calendar and member information, voting results, member speeches and multiple different documents covering for example laws, proposals and reports. The data is available through an API or via compressed datasets that can be downloaded in different formats.

The API that can be used to fetch documents is poorly made, which is why the compressed datasets has been used to retrieve data throughout this project. It lacks of well made documentation and has very restrictive forms for constructing search queries.

1.2 Motions and Speeches

In this project, two types of documents were selected for further analysis. Motions is the proposals given from the members of the Parliament to the Parliament. These are available from 1971 up to current date and consist of approximately 145000 documents. The other type of document selected is speeches in the Parliament, which can be found from 1993 up to current date and consist of approximately 295000 documents. The reason to restrict the selection to only include these two types of documents is mainly because there is a time frame, in which we want to produce something useful, and in focusing on a couple of different documents simplifies the process of storing and analyzing them. There is also a hardware limitation which affects the choice.

2 Server Configuration

During the project, a single server has been provided by IT services at the University of Gothenburg with the following specification:

OS: Red Hat Enterprise Linux Server 7.2

Processor: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

Memory: 19.45 GB

Disk Space: 99.50 GiB

2.1 Red Hat Enterprise Linux 7

RHEL 7 is a commercial Linux distribution based on community drive Fedora, another distribution which is free and open source. Both is owned by Red Hat and while Fedora focuses on quick releases with new features and functionality, RHEL has slower releases and comes with support. There is also another popular distribution named CentOS which is similar to RHEL but without the cost or support.

2.2 Systemd

The service manager used at the server is named Systemd and is provided by default in the RHEL 7 distribution. The purpose of the init system is a way of initializing components after kernel boot process and but also handling services at runtime. It comes with Systemctl, a central management tool for controlling the system, and works with unit files (suffix .service) for defining service tasks.

2.3 SELinux

Security-Enhanced Linux (SELinux) is a security module created by Red Hat that is put on top of the by default provided access control lists and file permissions. It has three basic modes of operation:

Enforcing: The default mode that will enable the SELinux security context on the system by denying access and logging actions.

Permissive: With this mode, SELinux is enabled but is only logging actions. Useful when troubleshooting permission problems.

Disabled: This option disables SELinux.

2.4 NGINX

NGINX is an open source web server that can act as a reverse proxy to other servers or serve static and dynamic content. It is used as a service in the server and multiple configuration files is created for running different virtual hosts, in NGINX called server blocks.

2.4.1 Apache HTTP Server

Another popular web server is the Apache HTTP Server which has a large community and widespread support. In comparison, NGINX includes less functionality and external modules but instead provides the core features for a web server without having the performance limitations an Apache

Server has. It is designed for high concurrency and therefore scales better by offloading network from the internal servers. It is also very common to use the two web servers together by deploying NGINX as a proxy server in front of an Apache-based application.

2.4.2 PHP-FPM

While Apache comes with built in support for several languages which could serve dynamic content, NGINX needs to proxy requests to other processes and one such protocol is FastCGI. One main use case with FastCGI is for PHP processing with PHP-FPM which is a separate processing manager made for handling PHP requests in NGINX.

2.4.3 httpd-tools

Basic authentication can be setup for the ability of enabling password protected entries within NGINX. This functionality can be provided by the package httpd-tools, which contains the command *htpasswd* that can be used to generate users and passwords.

3 Elasticsearch

Since a large part of the data consists of text documents, it is convenient to store, index and make them searchable using Elasticsearch. Elasticsearch is a scalable search server that is based on Lucene. It provides a schema-free document storage with JSON syntax together with a REST API as a communication interface, which makes it convenient to work with. Alternative choice to use as a full-text search engines is Apache Solr, which also is based on Lucene. Elasticsearch is originally created to address the flaws that exist in Solr and extends it with additional features.

The most popular use case for Elasticsearch is to build an analytics pipeline for log data. It is common to use the ELK stack, which in addition to Elasticsearch also includes Logstash and Kibana. Logstash is a data pipeline tool which is used for streaming and processing data. Kibana is a visualization tool and GUI for querying Elasticsearch, see section 3.2. In the case of this project, Elasticsearch is suitable for indexing documents which already is stored in JSON format. The texts and fields within the documents can be mapped in different ways so that one can utilize from the analytic possibilities within Elasticsearch. In this project, Elasticsearch version 2.2.0 is used.

3.1 Concepts

An instance of Elasticsearch is structured in clusters which are collections of nodes. A node can be defined as a single server where you store a part of your data, but participates in the indexing settings and search capabilities of the cluster. For increasing performance and availability of the data in a cluster, Elasticsearch also supports sharding and replication. In this project, Elasticsearch is deployed as a single node cluster with one shard and one replica.

A single cluster in Elasticsearch is structured in indexes. An index is a collection of documents that share some properties with each other and it can consist of one or multiple types. A type is logical partition of the data that usually have similar fields. The instance setup in this project includes two different indexes. The first relates to the political documents and is divided into two types. This is because two types of document, motions and speeches, are used in this project and each type defines a specific internal structure in the document. The other index includes user generated data, e.g. historical search queries, and for now it only contains one type related to documents of that specific functionality.

The documents are stored in JSON format and mapped when an index is created. The mapping is a process of defining how the documents, and the fields within it, will be stored, analyzed and indexed. In the setting of this project, every field is mapped to a specific type, for example date-fields are declared for the functionality of performing time-based queries.

The mapping also includes which fields that will be analyzed by Elasticsearch. Analyzing fields in Elasticsearch is the process of tokenization and normalization of text blocks before indexing. An analyzer can combine three different functions for performing this:

Character Filters: The job of these filters is to clean the strings before the tokenization step. It can for example be to strip out HTML tags or to convert specific characters to other.

Tokenizer: In this step the strings are separated into individual tokens, e.g. splitting a sentence into words.

Token Filters: The last type of filters will affect the different tokens that has been created. Common token filters is converting to lowercase or removing stopwords.

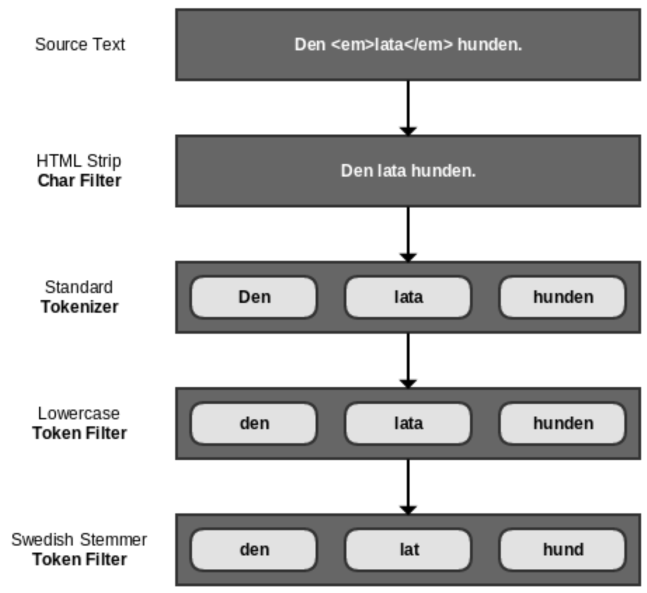

Figure 1: Custom analyzer for single words.

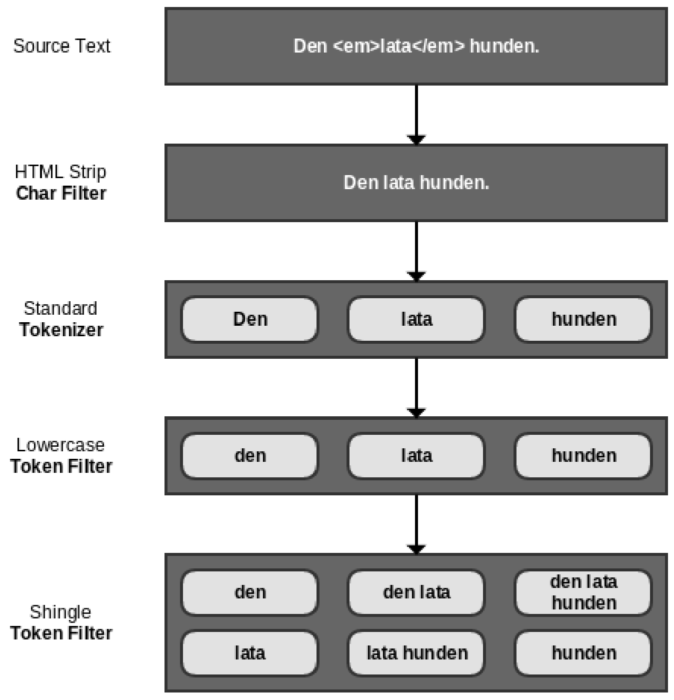

Figure 2: Custom analyzer for shingles.

A lot of the fields in the documents are not required to be analyzed and this is also stated the mapping. For those who are analyzed, two custom analyzer are used together with a standard analyzer. Both custom analyzers are applied to the field which contains the main text in motions for tokenizing and indexing that field in two different ways. The first is analyzing the full text into separate words, see fig. 1, and the other to shingles, see fig. 2. The shingle functionality creates word-nGrams which can be used for indexing phrases that later can be used for matching against. A downside is that this functionality generates a large set of indices and will thus consume a lot of memory.

Queries in Elasticsearch can be performed by using different language specific clients or the built in HTTP API. Data is submitted through JSON and a request for matching against all documents in all indices and types can e.g. be made by:

```
GET _search
{
  "query": {
    "match_all": {}
  }
}
```

Elasticsearch also has an aggregation framework which is widely used in this project. It provides aggregated data based on a search query and can be composed by multiple different building blocks to build complex summaries of data. A complex aggregation query performed by the server platform can look like:

```
GET ppd/type_motion/_search
{
  "size" : 0,
  "query" : {
    "filtered" : {
      "filter" : {
        "bool" : {
          "should" : [ {
            "match" : {
              "dokument.html" : "ekonomi"
            }
          } ]
        }
      }
    }
  },
  "aggs" : {
    "d6xvi9m8iy" : {
      "filter" : {
        "bool" : {
          "should" : [ {
            "nested" : {
              "path" : "dokintressent",
              "query" : {
                "bool" : {
                  "should" : [ {
                    "match" : {
```

```
                    "dokintressent.intressent.partibet" : "m"
                  }
                } ]
              }
            }
          }
        } ]
      }
    },
    "aggs" : {
      "term_agg" : {
        "terms" : {
          "min_doc_count" : 0,
          "field" : "dokument.html",
          "execution_hint" : "global_ordinals_hash",
          "collect_mode" : "breadth_first",
          "include" : "ekonomi",
          "size" : 10
        },
        "aggs" : {
          "date_agg" : {
            "date_histogram" : {
              "field" : "dokument.datum",
              "interval" : "year",
              "format" : "yyyy",
              "min_doc_count" : 0,
              "extended_bounds" : {
                "min" : 0,
                "max" : 1461313246544
              }
            }
          }
        }
      }
    }
  }
}
```

The first parameter **size** relates to the actual amount of documents that shall be returned. In this case we only want the aggregation data to be returned hence we set it to 0. The actual **query** segment is performed before the aggregation and in this segment, a lot of the documents are filtered out and the remaining ones are used in the aggregations step. The **aggs** segement includes information of how the aggregations should be performed. In this case, three types of aggregations are nested into a large one. The first is a filter aggregation which filters out documents based on some other field than the one that was stated in the query filter. The second aggregation is of the type term and collects documents into buckets where each bucket includes a key, a term, and a document count for that term. The last step is a date aggregation and divide the buckets further into date related ones.

3.2 Kibana

Kibana is an analytics and visualization platform which easily connects to Elasticsearch and can provide further insight into the data. It provides different plugins, e.g. Sense provides functionality for creating custom queries against Elasticsearch which is really convenient when developing applications. Besides external plugins it also offers an interface where one can look at the data in its default format and also a tool for creating basic visualizations and dashboards. In this project, version 4.4.1 of Kibana is used.

4 PPD-ETL

The PPD-ETL tool is developed for the purpose of extracting data from its source, transforming it to a proper format and load it into the database. It is created as a CLI tool and written in Python with support of different external libraries. Two different configuration files are served to the application, one of which includes which data to fetch from the data source and the other with the settings and mappings that will be used when creating the Elasticsearch instance. For example the custom analyzers used for the text fields in the documents are defined here. The tool communicates with the data source and the database through HTTP protocol.

The commands available when running the application is defined as:

Check Connection: Checks the connection to the Elasticsearch instance and that it is up and running.

Create Index: Creates an Elasticsearch index with the defined settings and mappings in the configuration file.

Fetch Data: Fetch the latest data from Riksdagens API and stores it in files.

Get Index Info: Get information about how many documents that are in the index and compare it with how many documents that are fetched to files.

Load Data: Load the file data into Elasticsearch.

Remove Index: Remove an Elasticsearch index together with all data entries.

Clean Query Index: Remove and recreate the index that includes search query history.

5 PPD-NGIN

PPD-NGIN is the central platform for connecting users and the GUI application with the database instance. At the front-end, there is communication through a simple API and at the back-end, the platform is sending queries to the Elasticsearch instance through the HTTP API. Different routes are defined as entry points and they all relates to specified function in the controller module. The controller module contains all logic within the platform. It extends the requests by asynchronous actions which for example can do any time consuming query to the Elasticsearch instance.

The controller module has multiple different sections for handling a request. Besides the actual entry points it includes one single method for parsing search queries that are included in the request data. Search queries are given in composed strings and need to be parsed so that useful information can be extracted from them. They consists of the actual search query together with different filters that are applied. Multiple search queries can also be made in the same request. Search can be written as three different types. *Terms* relates to single words, *phrases* to multiple words and *wildcards* to *phrases* that also include a wildcard.

Another section of the module creates Elasticsearch queries based on the parsed search queries and some logic. Especially the aggregations require some logic for producing correctly nested queries. There is also a section for handling the data retrieved from Elasticsearch and creating the response that will be sent back by the initial request.

5.1 Play Framework

The platform is developed in Scala and the Play framework. The reason for selecting a robust framework like Play is because will be easier to extend it in the future and it is also prepared for doing long running data processing tasks. This is because Play has a parallel distribution of its threads and uses non-blocking input/output. This means that a lot of idle threads waiting for responses can be avoided and instead used for other processes.

An alternative to using Play is Express.js (JavaScript) which is a web application framework for the Node.js platform. One benefit of using Express.js is that you can use the same programming language for both the client and server implementation. One disadvantage with using it is that it is very minimalistic framework so most of the time you need external plugins to do common tasks. In comparison Play is a much more full stack framework that provides everything by default, e.g. built-in authentication.

6 Interface design and development

The task was to build a visual interface for a website where data from the Swedish Parliament motions could be visualized and explored. The work was done in collaboration with Sverker Lundin project coordinator and Oscar Ivarsson whose work was to feed the data from Riksdagens önna data into an Elasticsearch database where an API provided an access to this data. Early in the process we found out that the first version of the system would take in a search query and display the results in two different views.

- A graph displaying usage frequencies of a given search query per year from 1971 to the current year (n-gram).
- A list of documents from the database which includes words or phrases the given search query. This will be referred to as the "hitlist".

The genaral concept of the interface is that the user can enter a search query, either a single word or a phrase or an array of words or phrases (search terms). A filter can then be applied to each term and in the first version the user can only filter by a political party. The search results will then appear on the graph visualizing usage frequencies per year and in the hitlist where documents will be listed. Furthermore, it is possible to define a range of years on the graph which will change the content of the hitlist showing only documents within that time range. The hitlist will include information such as the title of the motion, the date, which political parties are related to the motion and the possibility to view the full text of the motion.

6.1 Technology

The search interface was built mainly using the JavaScript libraries Backbone.js and D3.js. This interface will be embedded into a Wordpress theme.

6.1.1 Backbone.js

Backbone.js is a lightweight MVP (model-view-presenter) JavaScript framework built to work well with RESTful JSON data sources. The framework is modular and enables re-use of components created using it.

6.1.2 D3.js

D3.js is a framework for developing interactive and dynamic visualizations. D3.js outputs to a SVG (Scalable vector graphic) element in the Document Object Model in the browser and enables interactivity via the mouse to every visual elements that it creates, for example lines in a line chart.

6.1.3 Wordpress

Wordpress in an open-source content management system which is based on PHP and MySQL server technology. It is popular for maintaining blog sites but can also be used to structure more complex websites.

6.2 Interface components

The search interface is composed of four main components. Those are:

- A search query input form
- An n-gram graph viewer.

A slider for defining year range on the graph. A hitlist displaying documents corresponding to the given search query from the selected time range.

6.2.1 Search query input form

The search query input form is designed to take direct inputs in the form of a text string and search terms are divided by a comma. An example is “skatt, pengar” which will result in a graph showing usage frequency of both “skatt” and “pengar”. Party filters are applied to terms in the form of “skatt parti:(M), skatt parti:(S)”. This example will result in a graph showing comparasion of the usage frequency of the term “skatt” between the political parties Moderata samlingspartiet and Socialdemokraterna.

Each search term entered into the input form becomes an interactive button enabling the user to edit the search term by adding parti filters to it. This functionality is inspired by the recipient list when composing a new email in the Google Inbox web application. The query input form is built as a Backbone.js module.

6.2.2 N-gram graph viewer

The graph viewer makes requests to the API and displays the result as a line chart. The lines represent usage frequency per year, either showing absolute number of uses or as a percentage of total documents per year. The graph has visual control to switch between percentage view and absolute view. The graph component is built as a Backbone.js module but uses D3.js to build the actual graph.

7 Architecture

The server is structured into the multiple different modules stated in previous sections. The publicly available ones are served by NGINX through proxying requests to other internal web servers.

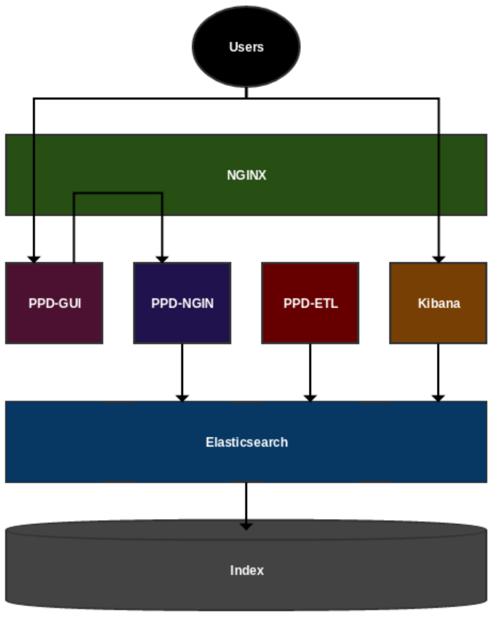

Figure 3: High-level design of the server structure.

Figure 3 is a simple design of the server structure and the arrows relates how communication flows between the modules. PPD-GUI is served through static content and could be reached by a browser. It also serves some dynamic content by proxying to a PHP-FPM process. The communication from PPD-GUI to the Elasticsearch instance occurs by making API request to the server platform PPD-NGIN which later sends the request further to Elasticsearch. PPD-ETL is a completely internal module, not reachable from outside. This is because it handles initialization and deletion of indices as well as modifying the data in the index. Kibana is also available by proxying requests through NGINX. It also includes basic authentication to be available.

8 Future Improvements

8.1 Automated Loading Scripts

In the current setting, if new data is wanted to be fetched and loaded into the database, this has to be performed manually by the PPD-ETL tool. This can of course be automated by making daily request to the data source and see if any new data is available.

8.2 Analyzing Relationships

The open data contains a lot of data regarding the members and people involved in different political parties. The relationship between people, how they have voted in different questions and the connections to different documents would be of high interest to further analyze. An idea here could be to utilize some sort of graph database, e.g. Neo4j, to create data models for finding interesting relationships within the data.