


IMPROVING E-GOVERNMENT SERVICES FOR ADVANCED SEARCH

Goran P. Šimić

University of Defence in Belgrade, National Defense School, Department for Simulations and Distance Learning, Belgrade, Republic of Serbia, e-mail: goran.simic@va.mod.gov.rs, gshimic@gmail.com, ORCID iD:  <https://orcid.org/0000-0002-7563-699X>

DOI: 10.5937/vojtehg67-20356; <https://doi.org/10.5937/vojtehg67-20356>

FIELD: Computer Sciences, IT
ARTICLE TYPE: Original Scientific Paper
ARTICLE LANGUAGE: English

Abstract:

The E-government services depend on many archived documents mostly scanned and partially described to be machine searchable in order to be found fast and to offer appropriate responses to citizens and to the government personnel as well. In order to improve the existing state, the hybrid solution based on the previous research results is presented. This paper presents an in-depth view of the Web solution that combines different technologies on both the client and the server side thus improving regular search services and making them accessible to people with disabilities (e.g. blindness).

Key words: text search, text similarity, speech recognition, metadata exploitation.

Introduction

Contemporary document management systems provide different ways for storing and searching the archived content. However, there are many documents used in government affairs, archived in formats that are not appropriate for searching tasks. They are often in a paper format. To make such archives digitalized, the offered software solutions include scanning of documents, saving them mostly in PDF/A (portable document format for archiving) and storing them on some shareable repository, relational or non-normalized (NoSQL) database, as a core of some CMS (content management system), or some more advanced system (Asili & Tanriover, 2014, pp.57-67).

ACKNOWLEDGMENT: The author is grateful for the financial support from the Ministry of Education, Science and Technological Development of the Republic of Serbia (project code: III44007).

For making them searchable, one has to make these documents machine-readable first. Without converting scanned content into text, the describing process is inefficient. Fortunately, there are many OCR (Optical Character Recognition) solutions on the market designed for this purpose (e.g. ABBYY Fine Rider as a commercial one, or FreeOCR as a desktop solution, etc.). By using this software, the scanned content, previously readable only for humans, becomes machine-readable. This is a prerequisite for performing a text analysis and making a static footprint for each document. In contemporary CMS, this is usually the last phase in making their content searchable for further exploitation.

Sometimes this is not enough. Especially in e-government services where the users (other Web based applications and services as well as government staff and citizens) demand high reliability and accurate response. There should be additional information that describes the documents in a better and more efficient way. Adding this information needs engagement of extra resources – people who should better describe the meaning of a document than machines do. Unfortunately, the personnel responsible for it are not always competent enough. Moreover, introducing people into the process significantly slows it down. Consequently, a document will be found only if there is a hundred percent matching of the title, the key words, or some other specific property within the search criteria. If IT designers try to make such a system more flexible, another problem arises: flexibility causes too many hits (results) that are not useful for further processing.

Overcoming the described situation represents one of the basic motives for the project. It is not possible to perform advance search in order to obtain fast fact finding (and offering appropriate responses to the citizens and to the government personnel) based only on the document features presented in some standard format (e.g. Dublin Core, <http://dublincore.org/documents/dces/>). This paper presents the results, collected experiences and considerations on this matter.

Problem Description and Existing Solutions

The previously described process for preparing archived content for further search brings all types of documents to the same level of complexity. These are text documents presented in different formats (PDF, DOC, DOCX, ODT, etc.). Different formats include a lot of non-informational content intended just to keep the content structure and presentation. Practically, this part of a document does not contain information interesting for users and introduces information noise

(Watson, 2009). On the other hand, removing the format data converts a text from structured into a plain format, which can lead efforts in the wrong way.

There are already implemented technologies for separating useful document content from its formatting part. For instance, Apache Tika (Mattmann & Zitting, 2012) represents a software solution for text filtering. In other words, it extracts text and metadata from almost every standard document format. It separates these two and enables their use for further searching.

On the other hand, there are software solutions focused on grouping a huge number of distributed documents, based on their similarity. A good example for such a solution is a combination of Mohout (Owen et al, 2011) (analyzing and clustering tool) and Hadoop (Sammer, 2012) (top framework for large scale concurrent processing). Both are the Apache projects dedicated for improving advanced search capabilities. Besides open-source solutions, there are also commercial ones. Upon purchase, contractors deliver released packages as unchangeable black boxes (Asili & Tanriover, 2014, pp.57-67). Also on demand, they can customize delivering for specific purpose, which consequently results in extra costs of software products.

From the technology prospective, there are programming languages that offer support for advanced search. For instance, the Python libraries and especially the Natural Language Processing Toolkit (NLTK) (Bird et al, 2009) have built-in, high-level linguistic functions that provide powerful processing of linguistic data contained in different document formats such as XML (*xml.etree* library), MS Word (*pywin32* library), PDF (*pypdf* library), RSS feed (*feedparser* library), email (*imap* and *email* libraries). Moreover, there are libraries that represent interfaces for access to data stored in DBMS (e.g. *mysql-python* library), or large document collections (e.g. *pylucene* library). Different from other systems, the NLTK can analyze the content semantically. It offers numerous functions providing the rule-based inferring on textual content. On the other hand, there are grammars (knowledge bases stored in *fcfg* files) that hold definitions of rules. The NLTK recognizes the meaning of the content by trying to match the patterns in content's sentences with the patterns defined in the rules. Sequentially, each matching produces a true or a false result and finally, the system can find a meaning of the content analyzed. The NLTK supports the rule-based grammars formalized by the Propositional Logic as well as the First Order Logic.

The main disadvantage is that the Python libraries best fit the content written in English and there is a lot of room for contributions for

other languages. For instance, one can translate grammars and adapt them to a language other than English. For instance, there is Serbian *WordNet* – the lexical database of Serbian language (Serbian-dictionary.com/wordnet) [X], based on English *WordNet* designed by the Princeton University. Consisting of almost 2500 records named *synsets* (words in the basic form enriched with synonyms), it enables a semantical analysis and translation of the content written in Serbian. Alternatively, translating sentences into English and analyzing them afterwards, represents another solution.

Going deeper into the problem domain, there are pure mathematical solutions that can overcome the non-English content search problem and avoid complex solutions based on lexical analyses. It depends on similarity measures performed on statistically transformed text content and query strings. TFIDF (Yang & Chute, 1994) represents one of the most used statistical measures. This is a combination of term frequency (hereinafter TF) and inverse document frequency (hereinafter IDF). TF represents the number of term occurrences in the text modified in order to express term significance (Šimić, 2015). A term can be one word (any part of speech), or a collocation (phrase, or a few words frequently appearing together). IDF is another measure that expresses significance of a term regarding to the whole (usually huge) set of documents considered. This way, IDF acts as a corrective factor for each term considered. In the further processing, the advanced search system measures the documents' similarity based on TFIDF, clustering them based on their mutual similarity. Finally, the statistical model and indexes represent the documents that are clustered and ready for search. Then the system is ready for exploitation, which means it transforms the search criteria in the same way as documents in order to measure the similarity with them. The most similar documents represent the search result. The statistical transformation of the content includes the term normalization (part of speech should be put in single, neutral / infinitive form – lemmatized) and elimination of so-called *stop words* (articles, pronouns, propositions, conjunctions, and interjections). As it needs the existence of an appropriate language knowledge base, the conclusion is that there is no one pure mathematical solution which is language neutral.

Proposed Solution

Although many technologies support advanced document search, the results still depend on a search language. In other words, there

should be institutions responsible for forming textual and lexical resources at the national level. These resources should be accessible for advanced search services over the Internet.

On the other hand, if one tries to avoid language dependency by implementing pure mathematical functions and by using only quantitative values, without combining the semantical similarity in comparison with document content and search criteria, the results can be below expectations. Simplicity represents the biggest advantage of such systems.

There are several ways for obtaining the communication between software modules written in different programming languages. The oldest one is by using language native interfaces. The results can be below expectations as it can be very complex for implementation and inflexible in case of language version changes. A more flexible and easier solution is using already built modules (well known as *bridges*) that establish the black – boxed, but reliable communication channel, free of a lot of coding and cleaned from many implementation details. For instance, if one wants to couple Java and Python clients, there is a bridge named Py4J (<https://www.py4j.org/>) referenced from both clients as a gateway server application. Several solutions use this approach (Svyatkovsky et al, 2016) for advanced text processing. Moreover, many flexible solutions offer different ways for cross language communication. One of them is Apache ActiveMQ (<http://activemq.apache.org/>) that represents cross language support for information exchange between application clients written in different programming languages offering many different protocols for this purpose.

Considering the facts mentioned above, the proposed solution should be a modular one, a flexible hybrid system that can establish the connections with different resources useful for text processing and information retrieval.

Frontend (Client) application

The basic components of the system are the HTML5 based client application(s) on the frontend, RESTful based services as a façade (*Façade* design pattern) of the backend part, intermediate interfaces and modules that provide different stages of processing user inputs (speech to text transformation, text normalization), generating queries and making searching request sent to different resources. By using HTML5 based technologies (e.g. Angular and Typescript JS libraries and Bootstrap frameworks), the client applications can adapt the user interface for any kind of platform (smartphone, tablet, or laptop).

The server side delivers a client application on demand. On the client side, the Web browser hosts it and takes the responsibility for further data exchange. This way, the client application is platform independent. The *Fat* client application performs all necessary preparations on user input (query). The client application has a full multimedia support (particularly multimedia recording) and therefore, there are two possible scenarios of usage – voice and textual search. In the first case (Figure 1), the client obtains the voice search that can be especially useful for users on the move. The client application uses the Google Cloud Text – to – Speech service (hereinafter the GCTtS service, <https://cloud.google.com/text-to-speech/>) for this purpose. The client application uses Web Speech API for recording the voice query, sends it to the service and receives the textual query for further processing.

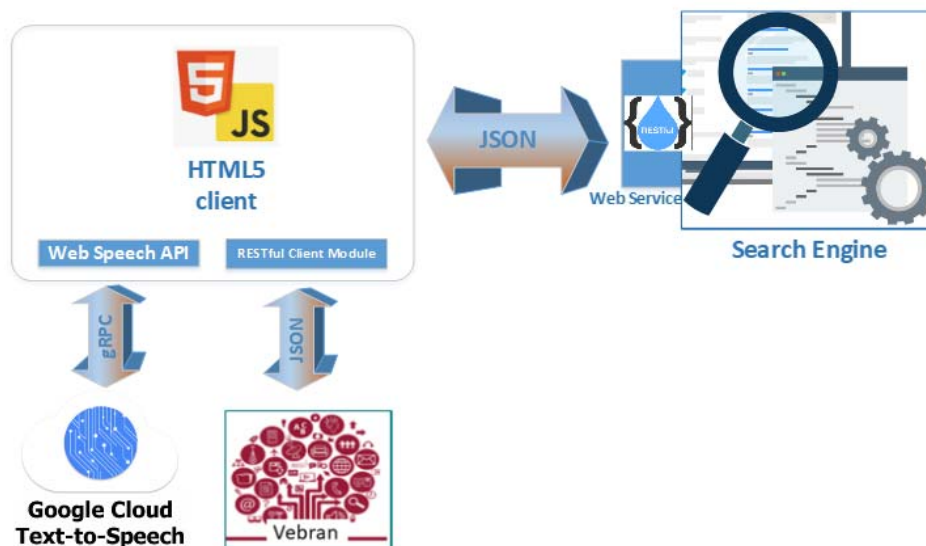


Figure 1 – Frontend – Backend communication with speech recognition
 Рус. 1 – Взаимодействие пользовательской и серверной частей системы
 Слика 1 – Комуникација између клијентског и серверског дела система

The GCTtS service supports almost 150 languages. Among others, there is a support for Serbian language. On the other hand, the client application has Serbian as a predefined language. As this is a parametrized value, it is changeable on demand. The client side application uses Web Speech API for establishing the communication channel to the GCTtS service, preparing it for Serbian speech recognition and for emitting the recorded voice query. On the other hand, the GCTtS

service is responsible for processing this request and for responding with the voice query transcript as a result. The gRPC (<https://grpc.io/>) protocol represents the flexible framework for this kind of communication. It enables sending audio files as well as establishing audio streams in the client-to-server direction. Transcriptions are the results in any case. In the same time, they represent the queries that client sends to the server side of the system.

Further, as the Serbian language has rich morphology, in both scenarios the client application continues with the preparation of a query. A particular module called 'normalizer' processes the query, preparing it for comparison with the content on the server side of the system. This 'preprocessing' includes several transformations: converting verbs into the infinitive form, nouns into the singular form, removing stop words, converting nouns into the 1st case (nominative). The client application uses the service named *Vebran* (hlt.rgf.bg.ac.rs/VeBran) for this purpose. This service provides all morphological forms for the term given as an input in both Cyrillic and Latin letters. As a term can be one or more words, the *Vebran* service expands the initial query with these forms. The client application uses only the basic forms of the words consisted in the initial query preparing the new one. Further, the client application sends the transformed query as a RESTful service request to the search engine on the backend side.

Backend architecture

Backend architecture supports three main functionalities of the system: extraction of useful content and metadata from original documents (accessible on the local repositories or over the network), document indexing and clustering and advanced search (Figure 2). Firstly, the system processes documents in order to clean all non-information content and to extract metadata useful for searching. Such content represents a searchable form of the original document and the system stores it in the local document storage. There is not heavy load for local storage as this is a pure textual representation.

Secondly, the system analyzes this searchable formatted content and performs its indexing. Through this process, the system creates the new data forms (files) that contain statistical information and indexes necessary for fast finding and advanced search ability.

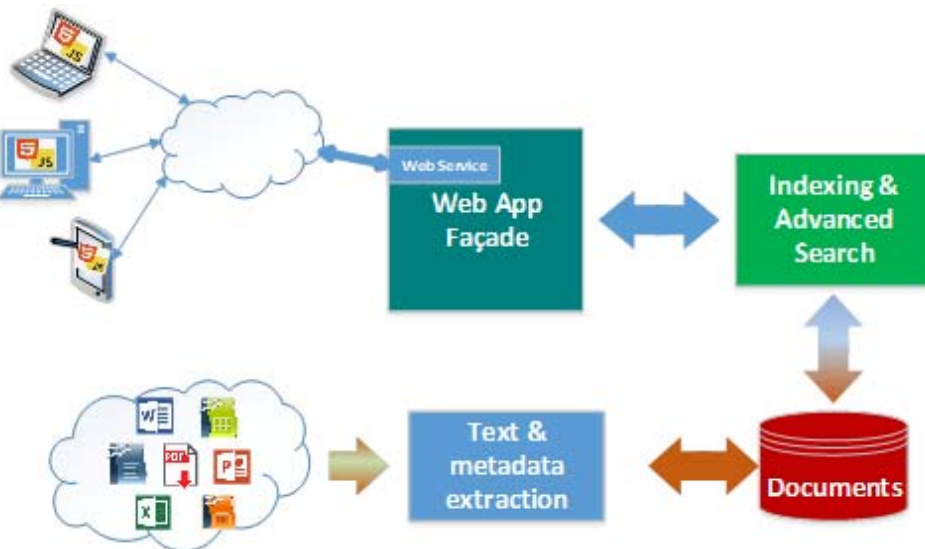


Figure 2 –Overall backend architecture
 Рус. 2 – Архитектура серверной части системы
 Слика 2 – Архитектура серверског дела система

The system is ready for advanced search only if it has previously performed both of the operations described above. It accepts clients' requests through the Web service forwarding them to the search engine in the middle of the system. Further, the system returns the search results in the form which consists of titles, short descriptions and links to the original documents.

Backend Preparations

As mentioned in the previous section, to be ready for exploitation, the proposed framework should have the content (documents) prepared for searching. There are several phases necessary for this purpose. The text filtering for both the information and the metadata found in different types of documents is the first one - 'Filtering' phase (Figure 3). The proposed solution uses the Apache Tika framework for this purpose. It performs the extraction of useful content from many different document formats (for instance, MS Word, Excel, PDF, various open document formats etc.). Further, the system creates a JSON formatted plain document representation from the extracted content and puts it in the appropriate document storage.

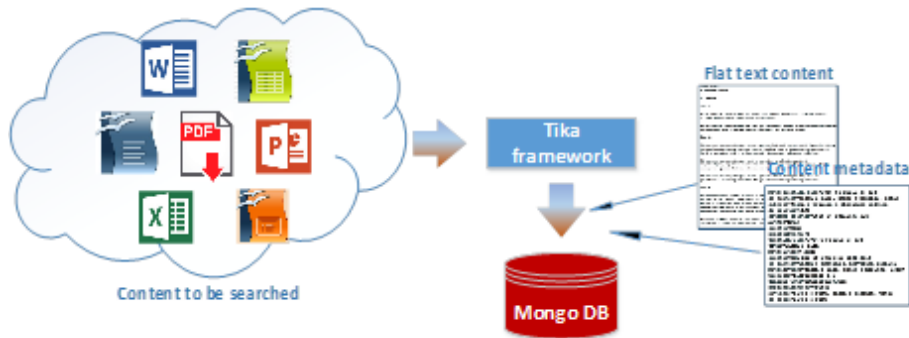


Figure 3 – Framework Backend 1st phase - 'Filtering'

Рис. 3 – Первый этап обработки документов на сервере - Очистка
 Слика 3 – Прва фаза обраде докумената на серверској страни – пречишћавање

In other words, the proposed solution generates two separate flat text documents for each document. As they differ in size and structure, the solution uses the MongoDB (NoSQL document database, docs.mongodb.com) as storage for non-normalized content. It enables easy document manipulation. Moreover, it provides indexing and making descriptive queries on documents. However, it does not have a support for the Serbian language (only 15 languages are supported) and for this reason, it is not appropriate for advanced search. Therefore, the solution uses the Apache Solr (<https://lucene.apache.org/solr>) indexing and search platform that can use the Python libraries for this purpose. It happens in the next phase named 'Indexing' (Figure 4).

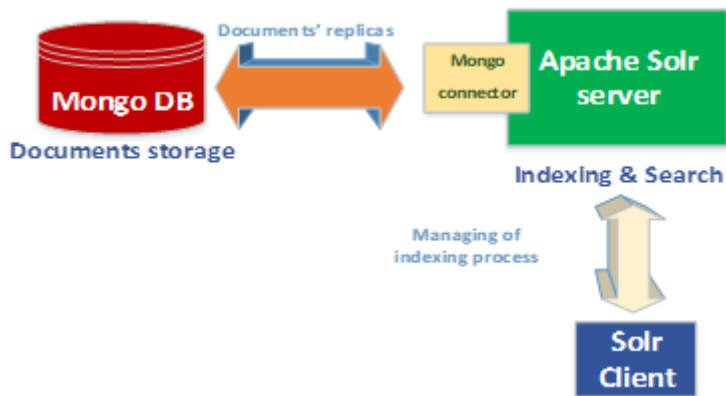


Figure 4 – Framework Backend 2nd phase - 'Indexing'

Рис. 4 – Второй этап обработки документов на сервере - Индексирование
 Слика 4 – Друга фаза обраде докумената на серверској страни – индексирање

There is a separate Solr *core* – the logical instance of the interface created for indexing remote content (stored on the MongoDB). On the other hand, the MongoDB setup enables the Solr indexing server to access the documents stored in the database. The data exchange happens over the Mongo Connector (created by Mongo Labs but community maintained by YouGov, Plc). For safety reasons, the Solr performs indexing on temporary created documents' replicas delivered through the connector instead the originals stored on the Mongo DB. The Solr supports the indexing of documents written in Serbian (in both Cyrillic and Latin letters). Moreover, the framework additionally improves the indexing by using the Python library named *SolrClient*. Before indexing, it enables full specification of particular fields that represent the documents, mapping them to the appropriate stop-word, synonyms and normalization filters. This way, it optimizes the indexing process and improves the searching results. Next is the code fragment that points to the important parts of the JSON request used for remotely setting up the Solr server in order to perform the indexing of documents written in Serbian (Figure 5).

```

3 {"add-field-type":{"name":"text_rs","class":"solr.TextField","positionIncrementGap":"100",
...
11     "filters":[{"class":"solr.StopFilterFactory","ignoreCase":"true","words":"stopwords_rs.txt"},
12               {"class":"solr.SynonymFilterFactory","synonyms":"index_synonyms.txt",
13                 "ignoreCase":"true","expand":"false"},
14               {"class":"solr.LowerCaseFilterFactory"},
15               {"class":"solr.SerbianNormalizationFilterFactory","haircut":"bald"}]},
16     "add-field" : {"name":"tekst","type":"text_rs","multiValued":"true",

```

Figure 5 – JSONized request that sets up the Solr server for indexing in Serbian

Рис. 5 – Настройка сербского языка на сервере Solr

Слика 5 – Подешавање Solr сервера за српски језик

The set up statement includes the document's field to be indexed (*text_rs* in the figure above), and links it with the appropriate content analyzer. Further, the analyzer's set up consists of different types of filters. In the example above, there are four different filters included. The first one is for finding and excluding the Serbian stop words (see *Problem Description*) from the indexing process. The second filter is for recognizing synonyms in the Serbian dictionary. The third one converts letters to the lower case. The last one preforms the text normalization. After the customization through the set up request, the Solr is ready for

indexing. This process includes the text analysis in the same order as the filters are enlisted in the set up statement.

The system performs indexing on the set of documents stored in the MongoDB by importing the fields specified for indexing (the Solr uses *solr-mongo-importer* and *solr-dataimporthandler* libraries for this purpose). In other words, the MongoDB consists of whole documents while the Solr stores only the fields, their statistical properties, important for search and references to the documents of origin. This way, the system keeps the data redundancy at the minimum level.

During the exploitation, if there is a new document stored in the MongoDB (after passing the Tika extraction), the system updates the indexes (simple *add* method call) immediately as the Solr is already set up. Only if there is another field of interest in the searching process, the Solr should explicitly reset before indexing.

Exploitation (Case Study)

When the user accesses the system URL, it delivers the reach client searching application implemented in HTML5 and JS (*Bootstrap*) technologies. This way, the client application has an adaptable layout regarding the concrete display dimensions. The next illustration shows the user interface adapted for a smartphone (Figure 6). The user interface contains a text field for typing and showing the search criteria, navigation buttons, the execution button and the voice search button.

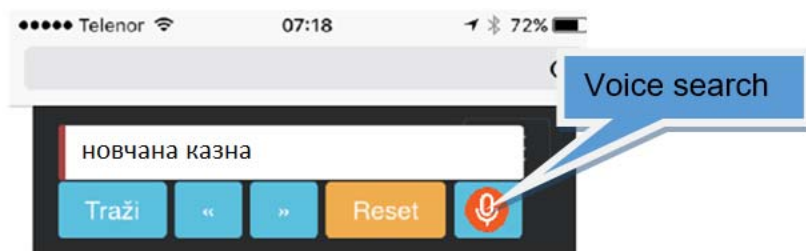


Figure 6 – Adaptable client applicaiton layout

Рис. 6 – Адаптивный пользовательский интерфейс

Слика 6 – Прилагодљиве интерфејс клијентске апликације

In the shown example, the user performed voice search. He can start and stop recording by toggling the voice search button. The client application uses simple mouse-over JavaScript to generate audio descriptions of each button. It helps people with disabilities (e.g.

blindness) to find the toggle button for start / stop recording. The next illustration shows the most important code fragments for the voice search implementation (Figure 6). As mentioned before, the client application uses WebSpeech API for this purpose. The application firstly creates the instance of the *SpeechRecognition* class. This class encapsulates speech recognition functionality hiding implementation details such as voice recording by microphone, sending the recorded voice as a byte array to the Google Cloud Speech-to-Text service and receiving the transcript responded from the service.

Recording starts by calling the *start* method on the *SpeechRecognition* object (Figure 7). The application will render the talk immediately after start, simultaneously sending it to the remote service. The service also responds immediately. The client catches the results by the *onresult* event handler. The service response is structured and it is accessible over the *event* object. It consists of an array named *results*. This array, among other properties, contains the transcribed text held in the *transcript* object.

```
<textarea id="recognizedText" rows=10 cols=80></textarea>
<button id="button" onclick="toggleStartStop()"></button>
<script type="text/javascript">
  var stat;
  var speechRecognition = new SpeechRecognition();
  ...
  speechRecognition.onresult = function (event) {
    for (var i = event.resultIndex; i < event.results.length; ++i) {
      if (event.results[i].isFinal) {
        recognizedText.value += event.results[i][0].transcript;
      }
    }
  }
  ...
</script>
```

Figure 7 – Fragment of the client application code for voice search
 Рис. 7 – Фрагмент пользовательского кода для голосового поиска
 Слика 7 – Фрагмент клијентског кода за гласовну претрагу

As described, the client application performs the normalization of the transcript forwarding it to the *Vebran* service encapsulated in the class named *NormalizationService* (Figure 8). The implementation of this functionality is in the method named *getNormalized*. This method sends the transcript previously changed into the query form to the service over the *http get* request, and returns the service response.

```

import { Observable } from 'rxjs/Observable';
...
@Injectable()
export class NormalizationService {
  private serviceUrl = 'http://...../vebran/..';
  ...
  getNormalized(transcript: SolrQuery): Observable<string> {
    let q = transcript.getQ();
    ...
    var response = this.http.get(this.serviceUrl + q)
      .map(response => response.json() as string)
      .catch(this.handleError);
    return response;
  }
  ...
}

```

Figure 8 – Fragment of the client application code for the normalization of the search query

Рис. 8 – Фрагмент пользовательского кода для нормализации запроса
Слика 8 – Фрагмент клијентског кода за нормализацију упита

The client application makes the services mutually synchronized by defining the service call methods (in classes that encapsulate them) *Observable*. The next code presents the important parts of the *performSearch* method (Figure 9). This method uses the *Vebran* service by calling the observable *getNormalized* method. Further, it synchronously calls the Solr service and updates the user interface with the search results.

```

performSearch(transcript: SolrQuery): void {
  ....
  let getDlfs = this.NormalizationService.getNormalized(transcript)
    .map(query => this.storeResults(query),
      ...
    ).mergeMap(response => this.SolrService.getResults(query)
    ).subscribe(
      response => this.updateSolrResults(response),
      ...
    );
  ...
}

```

Figure 9 – Using the Vebran service
Рис. 9 – Использование сервиса Vebran
Слика 9 – Коришћење Vebran сервиса

The *Vebran* response represents the plain string of all search criteria forms (singular, plural, in all cases) separated with a semicolon: *новчана казна;новчанум казнама;новчана казно;новчаном казном;новчану казну;novčana kazna;novčanim kaznama;novčana kazno;novčanom kaznom;novčanom kaznom*. Before calling the Solr service, the client application composes the search query by using the normal forms of the each word. Consequently, it calls the *getResult* service method passing the query and waiting for the service response. Finally, the returned result is rendered thorough the synchronized (by the *subscribe* function) *updateSolrResults* method. The server side receives the search queries in the pure text (JSON) format.

The server side RESTful controller service acts as a façade [X] of the backend system. It propagates the request to the Solr server for searching the indexed fields. The Solr returns the result records that contain references to the documents. Further, the server application uses these references to find the documents stored in the MongoDB. Such approach provides fast finding of the parts of documents large in size and with a complex structure. Finally, the service returns the name and the part of the text that matches the criteria for each resulting document (Figure 10).

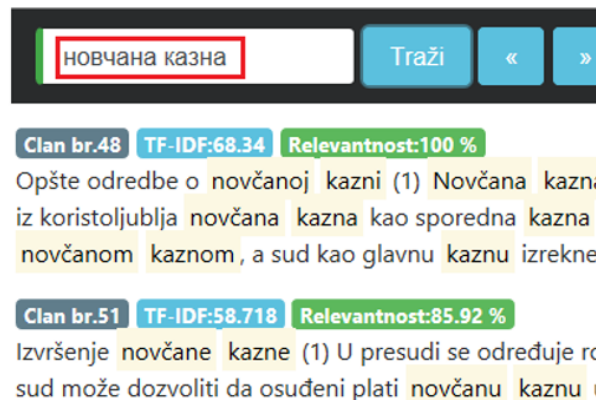


Figure 10 – Search result
 Рис. 10 – Результаты поиска
 Слика 10 – Приказ резултата претраге

The client application renders the JSON formatted returned result by using the CSS specification of each returned field.

For evaluation, there were 100 different searching tasks performed. The next chart (Figure 11) shows the results. There is a big difference

between server side processing and communication time. The server side processing includes the activities: accepting search request, finding the similar documents and preparing the responses. The measured processing time is less than $10e-1$ second (bottom part of the chart). The communication time is measured on the client side from sending the search request to receiving the search result from the server. The communication time is more than ten times longer (upper part of the chart).

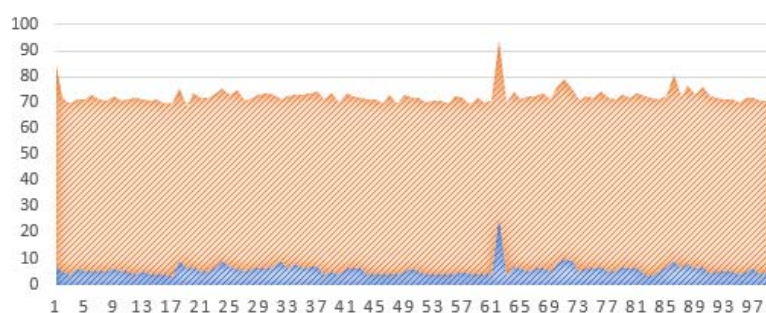


Figure 11 – Processing vs. communication time
 Рис. 11 – Время обработки данных и время обмена данными
 Слика 11 – Време обраде и време размене података

The other part of the evaluation includes the client side search query preparations (speech-to-text as well as lexical transformations). The measured values were similar to the client - server communication time. During experimentation, the overall responding time of the search service measured was 1.7 second in average.

Conclusion

The focus of the proposed solution is improving the Serbian e-government searching service in order to provide advanced search of huge documents corpuses in an efficient way as well as to enable this service for people with disabilities. Focusing on the Serbian language was a big challenge due to its grammar complexity and rich vocabulary of words, terms, synonyms and homonyms. We tried to overcome these difficulties on both sides – client and server applications. The client side application uses the *Vebran* service that performs an in-depth lexical analysis of the text responding with the all-possible forms of the sentence given as the search criteria. The client application uses the normalized one and sends it to the search engine on the server side.

The solution uses the *Google Cloud Speech-to-Text* service as an appropriate one to obtain searching by voice. As typing is complex for drivers or passengers and impossible for people with disabilities, this service is included in the solution. By setting it up in the proper way, this service presents a high level of accuracy and satisfactory responding time in both cases - during the evaluation and exploitation.

On the other hand, there are other languages of interest in Serbia (Albanian, Hungarian, Bulgarian, etc.). Owing to the flexibility, modularity and low coupling of components of the proposed solution, as well as to a lot of supporting libraries in the frameworks included, these requests are feasible on both the client and the server side application. The *Google Cloud Speech-to-Text* service supports 120 languages (including the above enlisted) while the Solr can provide indexing for 36 languages. Moreover, the Python libraries incorporate an advanced analysis and the processing features for more than 50 languages.

There are experimentations presented in the paper. Their results demonstrate respectable processing power of all the services used. The network speed still represents the main factor for slowing down the service response.

As document storage, the MongoDB is a flexible solution for holding non-normalized content. It fits well a great number of documents that differ in size and content structure. This way, it is appropriate for storing short content such as messages, comments and emails as well as books, laws, magazines and similar ones, much greater in size and more complex in structure. Also, a well-supported communication between the MongoDB and the Solr (*solr-mongoimporter* library) provides high performances in both indexing and searching processes.

The *Vebran* service authors suggest query expansion with the sentence returned by it service (Stanković et al, 2016, pp.112-123) in order to obtain results that better fit the initial query. On the other hand, the more terms and forms supported by the service, the better results returned. For instance, if the user changes the query above with one more word “*Дефиниција новчаних казни*” (*Definition of amercements*), it practically changes the term. If the service does not support a concrete collocation of the words in the criteria, the query will not have expected expansion and consequently, it will not produce the proper search result. Therefore, the client application uses the service firstly to find collocations (the terms that represent the ordered sequences of two words) and secondly, to find forms of the rest of the words contained in a query. Based on the previous example, it means that the word ‘definition’ will be treated separately from the term ‘criminal acts’. This way, the

client application prepared the query for searching the documents written in Serbian.

The proposed solution is scalable. As the MongoDB and the Solr hold different information of documents, the system is flexible for distribution. There can be more than one MongoDB and Solr instances. If these instances can hold the same documents translated into different languages, the solution can provide the same searching improvements for each one. Moreover, both MongoDB and Solr have support for cloud solutions. The MongoDB offers the Atlas (<https://www.mongodb.com/cloud>) commercial cloud solution while the SolrCloud () solution is free of charge. Nevertheless, migration on the cloud should depend on the size of the document base as well as on the number of requests (users). If these numbers arise rapidly, the service providers should start planning in time.

References

- Asili, H., & Tanriover, O.O. 2014. Comparison of Document Management Systems by Meta Modeling and Workforce Centric Tuning Measures. *International Journal of Computer Science, Engineering and Information Technology*, 4(1), pp.57-67. Available at: <https://doi.org/10.5121/ijcseit.2014.4106>.
- Bird, S., Klein, E., & Loper, E. 2009. *Natural Language Processing with Python*. O'Reilly Media.
- Mattmann, C., & Zitting, J. 2012. *Tika in Action*. Greenwich, USA: Manning Publications.
- Owen, S., Anil, R., Dunning, T., & Friedman, E. 2011. *Mahout in Action*. Greenwich, CT, USA: Manning Publications Co.
- Sammer, E. 2012. *Hadoop Operations*. O'Reilly Media.
- Stanković, R., Krstev, C., Vitas, V., Vulović, N., & Kitanović, O. 2016. Keyword-Based Search on Bilingual Digital Libraries. *LNCS*, 10151, pp.112-123.
- Svyatkovsky, A., Imai, K., Kroeger, M., & Shiraito, Y. 2016. Large Scale Text Processing Pipeline with Apache Spark. In *Big NLP Workshop, IEEE Big Data conference*.
- Šimić, G. 2015. E-Government Documents and Data Clustering. In Z. Mahmood, Ć. Dolićanin, E. Kajan, D. Randjelović, & B. Stojanović Eds., *Handbook of Research on Democratic Strategies and Citizen-Centered E-Government Services*. IGI Global, pp.164-191. Available at: <https://doi.org/10.4018/978-1-4666-7266-6.ch010>.
- Watson, M. 2009. *Scripting Intelligence: Web 3.0 Information Gathering and Processing*. Apress, pp.29-32.
- Yang, Y., & Chute, C.G. 1994. An example-based mapping method for text categorization and retrieval. *ACM Transactions on Information Systems*, 12(3), pp.252-277. Available at: <https://doi.org/10.1145/183422.183424>.

УЛУЧШЕНИЕ СЕРВИСА ЭЛЕКТРОННОГО ПРАВИТЕЛЬСТВА ДЛЯ РАСШИРЕННОГО ПОИСКА

Горан П. Шимич

Университет обороны в г. Белград, Школа национальной обороны, Отдел симуляции и дистанционного обучения, г. Белград, Республика Сербия

РУБРИКИ: 20.23.00 Информационный поиск;
20.23.25 Информационные системы с базами знаний

ВИД СТАТЬИ: оригинальная научная статья

ЯЗЫК СТАТЬИ: английский

Резюме:

Услуги электронного правительства зависят от архивирования документов, которые в основном сканируются и частично описываются с целью обеспечения машинного поиска и быстрого нахождения соответствующих ответов как для пользователей, так и для сотрудников электронного правительства. Для улучшения существующей ситуации было разработано гибридное решение, основанное на результатах предыдущих исследований. В данной работе представлено описание Веб-сервера, комбинирующего различные технологии, направленного на улучшение стандартных услуг поиска и обеспечения их доступности для людей с ограниченными возможностями.

Ключевые слова: текстовый поиск, схожесть текстов, распознавание речи, использование метаданных.

УНАПРЕЂЕЊЕ СЕРВИСА Е-ВЛАДЕ ЗА НАПРЕДНУ ПРЕТРАГУ

Горан П. Шимић

Универзитет одбране у Београду, Школа националне одбране, Одсек за симулације и учење на даљину, Београд, Република Србија

ОБЛАСТ: информатика
ВРСТА ЧЛАНКА: оригинални научни рад
ЈЕЗИК ЧЛАНКА: енглески

Сажетак:

Услуге е-управе зависе од архивских докумената који су углавном скенирани и делимично описани како би се могли машински претраживати и брзо пронаћи одговарајући одговори за грађане и службенике. Да би се побољшало постојеће стање, представљено је хибридно решење засновано на претходним резултатима истраживања. Овај рад представља опис веб софтверског решења које комбинује различите технологије како на страни клијента тако и на

страни сервера, побољшавајући редовне услуге претраживања и чинећи их приступачним за особе са инвалидитетом.

Кључне речи: претрага текста, сличност текстова, препознавање говора, експлоатација метаподатака.

Paper received on / Дата получения работы / Датум пријема чланка: 30.01.2019.

Manuscript corrections submitted on / Дата получения исправленной версии работы / Датум достављања исправки рукописа: 23.02.2019.

Paper accepted for publishing on / Дата окончательного согласования работы / Датум коначног прихватања чланка за објављивање: 25.02.2019.

© 2019 The Author. Published by Vojnotehnički glasnik / Military Technical Courier (www.vtg.mod.gov.rs, втг.мо.упр.срб). This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/3.0/rs/>).

© 2019 Автор. Опубликовано в «Военно-технический вестник / Vojnotehnički glasnik / Military Technical Courier» (www.vtg.mod.gov.rs, втг.мо.упр.срб). Данная статья в открытом доступе и распространяется в соответствии с лицензией «Creative Commons» (<http://creativecommons.org/licenses/by/3.0/rs/>).

© 2019 Аутор. Објавио Војнотехнички гласник / Vojnotehnički glasnik / Military Technical Courier (www.vtg.mod.gov.rs, втг.мо.упр.срб). Ово је чланак отвореног приступа и дистрибуира се у складу са Creative Commons лиценцом (<http://creativecommons.org/licenses/by/3.0/rs/>).

