Unified Search and Browse Interface for MusicBrainz

A tool for browsing music metadata

Ruchiranga Wickramasinghe
Department of Computer Science and Engineering
University of Moratuwa, Sri Lanka
ruchiranga.12@cse.mrt.ac.lk

***Abstract*—MusicBrainz provides an open music encyclopedia that collects music metadata and makes it available to the public for free. The musicbrainz.org [1] website currently supports an advanced query search to browse the music metadata database with queries that allow to specify descriptive search such as "The artist groups who started their career between 1992 and 2000". This search functionality is built using the Apache Lucene search library and one has to be familiar with the Lucene search syntax in order to type in a valid query that fulfils their need. If not, one will have to refer to a huge documentation that contains all the possible keywords for each entity in the database, which would consume a considerable amount of effort and time. This paper describes a solution to this issue developed by creating an interface that unifies the advanced query search with a simple indexed search while allowing the user to do any search that they could have done using the existing search functionality without needing any prior knowledge on Lucene search syntax or other documentation.**

Keywords—MusicBrainz; music; metadata; MetaBrainz; music encyclopedia; unifed search and browse.

I. Introduction

MusicBrainz is a project carried out by the MetaBrainz non-profit organization [2], which provides an open music encyclopedia that collects music metadata and makes it available to the public all over the world for free. It is one of the leading open source music metadata databases in the world and contains music metadata of over 1 million artists from all around the world. The musicbrainz.org web site [1] currently has a search facility that is built on top of this metadata database primarily with two options [3]. The first option is an indexed search functionality that allows a user to enter a search key and get a list of items that match the given search key. The other option is the indexed search with advanced query syntax. This search functionality is built using the Apache Lucene search library and one has to be familiar with the Lucene search syntax in order to type in a valid query that fulfils their searching or browsing needs. For someone who is not familiar with this syntax, he or she will have to refer to a lengthy documentation that contains all the possible keywords for each relevant entity in the database, which would be a cumbersome thing to do. The primary motivation behind this project is the difficulty in using the above mentioned advanced query syntax search that currently exists in the musicbrainz.org website. Furthermore, for someone visiting the site for the first time, the vastness of the database of metadata available with MusicBrainz is quite unseen simply due to the complexity of the procedure that one has to follow to get some advanced information retrieved from the database. The designed interface addresses all of these issues and provides a good exposure to the information made available via the MusicBrainz database to the general public. The difficulty in going through the lengthy documentation [4] to create the exact query that satisfy ones searching needs would have caused a significant amount of users to have given up on getting their search needs done with the MusicBrainz database. This in turn would have made a negative impact on the experience the people get in interacting with the musicbrainz.org website. With this unified search and browse interface, users are no longer needed to waste their time and effort, reading through the lengthy documentations finding the exact key terms to use in the query that they are trying to create.

II. Literature Review

The musicbrainz.org website, which is the official website of the MusicBrainz project at present, has its search functionality [3] implemented in three ways.

The first method is the indexed search where the search is carried out with the use of an indexing mechanism that is updated every three hours. This search allows the user to specify an entity, which the searching needs to be done with and a search term that is directly used to match with the items in the index and find the results. This is the most basic search feature implemented in the aim of providing a simple fast way of accessing the database with the cost of not returning the most recent data.

The second option is the indexed search with advanced query syntax where the search functionality provided is much more powerful than the previous method. This approach uses the Apache Lucene library which is a high-performance, full-featured text search engine library which can be used for any application that requires full-text search, especially cross-platform [5]. The advantage of using this library is that it allows making more abstract search queries like "Artists of type 'person' from Sri Lanka who were born before 2000". Even though this gives the user more power on retrieving very useful information from the database, the query syntax that specifies the above mentioned abstract sentence to the search system

happens to be quite complex. For someone who is not familiar with the Lucene search syntax, they will definitely have to go through the documentation provided in the web site itself for advanced query search [4]. Hence the exposure the contents in the database get via this search is quite limited.

The third method that the searches can be carried out is the direct database search, which does not use any indexes, but talks to the database itself directly. This option is made available to mitigate for the issue of indexed results not being up to date with the database contents real time due to the three hour update interval. But this search performs slower than the indexed searches and sometimes returns a less number of results due to the search timeout constraint imposed to make it responsive.

All these three existing search features by themselves fail to provide a fast, simple, feature rich search and browse functionality that provides better exposure and easy access to the contents in the database.

In addition, MusicBrainz server has a web service that is integrated to the server itself that accepts queries in the Lucene search syntax and returns results either in XML or JSON formats [6]. It is an interface to the MusicBrainz database that is aimed at developers of media players, CD rippers, taggers, and other applications requiring music metadata. The service's architecture follows the REST design principles. Interaction with the web service is done using HTTP and all content is served in either XML format or JSON format depending on what is specified in the query being sent to it [6].

III. System Models

A. System Requirement

The users need to be provided with the set of all categories that the database can be browsed with. The query to be sent to the Web Service needs to be generated depending on which categories the user has selected. The query that is currently being sent to the MB Web Service should be visible to the user as a hyperlink. That way the users are also able to get an idea on what the Web Service actually returns for that particular query even though it is shown in JSON format. Once the user has selected a category to browse the database through, its set of sub fields that the results can be narrowed with should appear. The sub fields should allow multiple selections and depending on the items selected at a particular time, the query shown to the user needs to be updated. At the same time on each update, an asynchronous request needs to be sent to the Web Service to retrieve the relevant results. The received results need to be parsed and shown in a tabular manner. The results retrieved after querying the MB Web Service are usually in large numbers and hence a pagination of results is needed. At every update made to the query, only the first 25 results of all that matched that particular query need to be shown avoiding an infinite length scroll. User should then be able to traverse through the next pages from a next button as well as to jump to a specific page from a button that has a label representing that specific page. In addition to all that the user should be able to optionally specify a search term to be incorporated to the query being generated so that the browsing and the searching will be unified. The primary functional requirements of the Unified Search and Browse Interface explained above is shown in Fig. 1, in the form of use cases.

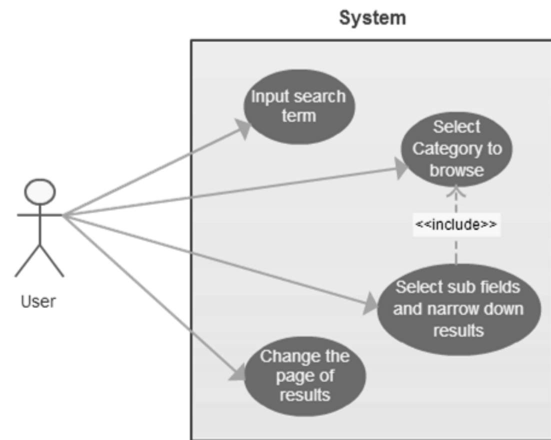


Fig. 1. Main use case diagram of the system.

B. System Design

As far as the functionality of the interface is concerned, the system can be looked at as an application that well adheres to the client server architecture. The client side includes the user interface which will handle all the user inputs and the actions that needs to be followed after receiving them including querying to the web service. It maintains a connection with the server side layer via HTTP requests and responses. The server side of the application includes the MB web service as the interface for accessing the database and the MB database as the back end source of information.

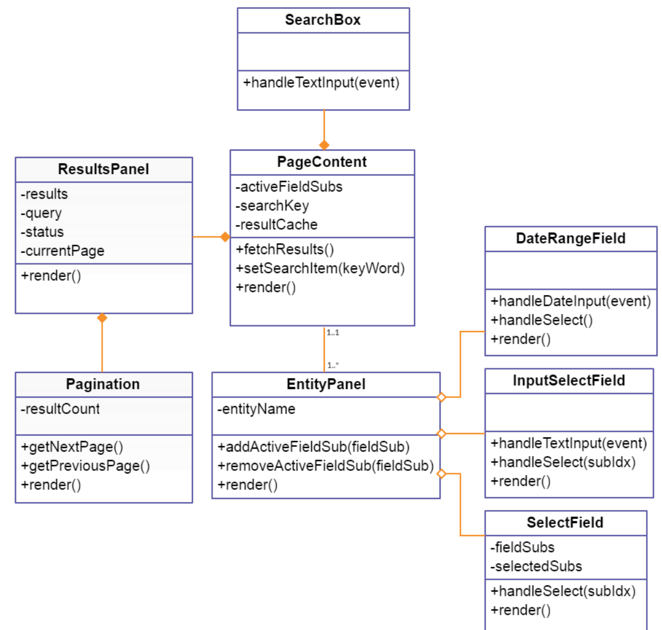


Fig. 2. Logical view of the system.

Logically the system is composed of two significant packages. The Interface Package contains the classes or components that are used in the development of the user interface. The Support Package contains the controllers and

other utility components that are being used by the interface to provide its functionality. Fig. 2 shows the logical view that corresponds to the contents of the Interface Package. The *PageContent* component is composed of the *ResultPanel* component and the *SearchBox* component and has one or more *EntityPanel* components associated with it. The *ResultPanel* component has the *Pagination* component. Each *EntityPanel* component is an aggregate of one or more of *DateRangeField*, *InputSelectField* and *SelectField* components. All these components are custom React UI components.

The design phase of the interface involved in a lot of UX research since the interface is meant to be used by the whole internet community and the interface needed to have the ability to provide a good user experience to each and every one of those users. Accordingly an initial prototype was created [7] and presented to the MusicBrainz community for collecting feedback and the feedback had to be associated carefully when the actual interface was built. The interface was made such that it could be used and navigated only using keyboard navigation as well. The interface had to be made as uncluttered as possible and a lot of effort was made to make sure the users feel engaged with the system while using it by making notifications appear, updating the user on what is happening on the background whenever it was needed.

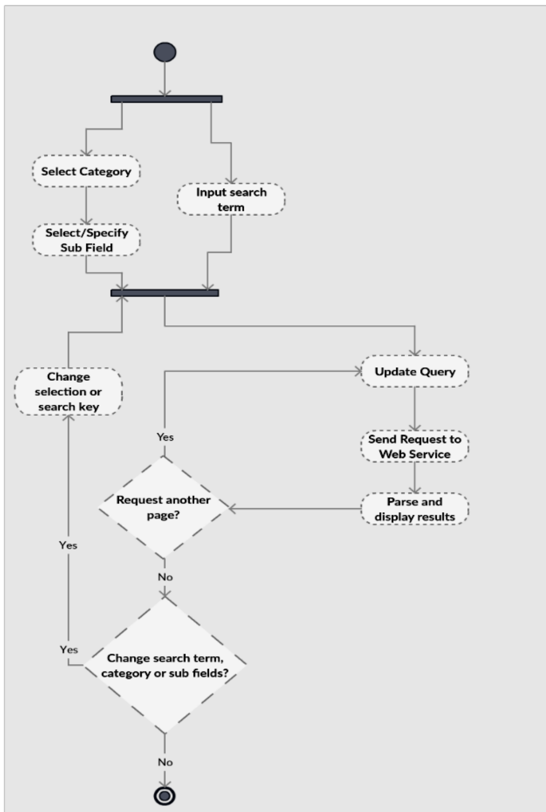


Fig. 3. Process view of the system.

Fig. 3 depicts the sequence of actions that a user is eligible to perform on the interface with the corresponding inputs as a diagram. The user can optionally enter a search term or select a certain category and further options will pop up guiding him or her to proceed to the next step. Then, selecting a single sub field or multiple sub fields will complete the query generation and the system will proceed to send an HTTP request to the web service and the response will be received in the JSON format. Upon successful reception of the results, the system will automatically parse the JSON string into an object and use the content of that object to display the results in a tabular format. At any time the user is able to change the category he has selected, the sub fields he has selected, the search key he has specified as well as the page of results.

IV. System Implementation

A. Implementation Procedure

The unified search and browse interface requires a good deal of interactions with the user and needs to be able to change it self dynamically depending on the inputs given and events fired by the users. Hence to meet this requirement of dynamic nature, a UI building library would be of good use. Two of the most prominent such libraries are KnockoutJS [8] and ReactJS [9]. The primary language used throughout the project was JavaScript and the prototype of this project [7] was built using the KnockoutJS JavaScript library [8]. The library was changed to ReactJS [9] during the actual development mainly because of the fact that ReactJS shows better performance than KnockoutJS in rendering the UI [10].

The most powerful effect of using react is that it allows designing the interface in a modular manner where each component would have its own state as well as its own mark up. The mark up for each component was written using JSX which is a JavaScript syntax extension with tags similar to those in XML that provides a concise and familiar syntax for defining tree structures with attributes [11]. The JSX allows specifying pointers to functions that needs to be called on certain event triggers. These event handling functions would alter the components state depending on the context and the input received and the changes in state are reflected in the mark up in turn. This concept helps building maintainable dynamic UI's of remarkable quality with very much less effort.

Since the interface is targeted for use by the whole world, all operations carried out by the interface needs to be cross browser compatible. For this purpose, for most of the computational necessities, the Lodash library [12] was used. Lodash is a JavaScript utility library that delivers consistency, modularity and performance with some other extra functionality as well [12]. Each and every entity, field and option shown in the unified search and browse interface are retrieved dynamically from the database as the page is compiled and generated in the MusicBrainz web server. This way any modification done to the database schema or values in the database would automatically be reflected in the interface contents avoiding the need of redundant work. This was achieved by calling the server side function utilities that act as an API over the MusicBrainz database.

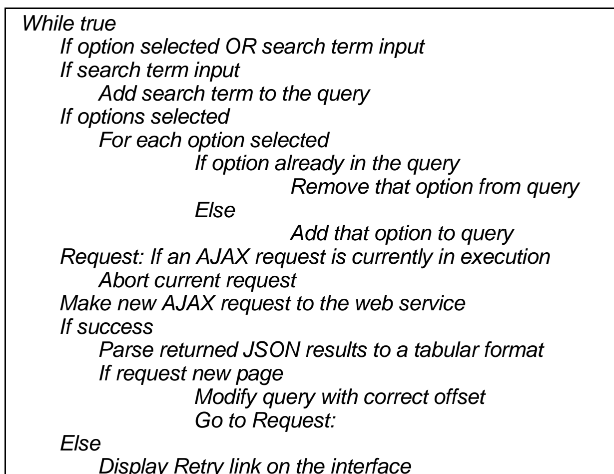
The styling of the interface was done using LESS which is a dynamic style sheet language that can be compiled into Cascading Style Sheets (CSS) and run either on the client-side or server-side [13]. LESS allows the use of variables as well as more structured style declarations. The MusicBrainz server has own colour palette and predefined sizes for fonts and other text.

The interface was styled using all those predefined constants to make it look consistent with the rest of the site.

All the development of the project was done using SublimeText [14] which is a sophisticated text editor for code and mark up. The necessary information in generating the query with valid syntax and keywords was taken from the documentation for the MusicBrainz XML web service available at the musicbrainz.org website [4], [6].

The primary purpose of the Unified Search and Browse Interface is to provide a way for the users to quickly and easily browse through music metadata. All the results displayed on this interface are solely retrieved from the MusicBrainz database. The application that acts as the interface for the database access is the MusicBrainz XML web service and the data are received by the client interface in JSON format. Furthermore, for maintainability purposes, all the entities, fields as well as options available for the user to interact with are also retrieved from the MusicBrainz database. This includes the list of all countries in the world and a list of all script languages in the world as stored in the database as well.

The query generated by the interface needs to have correct and valid Lucene key words for the corresponding fields and options selected by the user. The documentation of MusicBrainz xml web service [6] contains all the possible key words for each of the possible fields and options. This comprehensive document was used as a reference in coding the behavior of the interface that deals with generating the query. The primary pseudo code for the overview execution of the interface is given in Fig. 4.



```
While true
    If option selected OR search term input
    If search term input
        Add search term to the query
    If options selected
        For each option selected
                If option already in the query
                        Remove that option from query
                Else
                        Add that option to query
    Request: If an AJAX request is currently in execution
        Abort current request
    Make new AJAX request to the web service
    If success
        Parse returned JSON results to a tabular format
        If request new page
                Modify query with correct offset
                Go to Request:
    Else
        Display Retry link on the interface
```

Fig. 4. Main use case diagram of the system.

Once the interface is loaded, it waits for the user to select an entity for the browsing to be done with. Once an entity is selected, the user has the choice of selecting the options to browse the database with respect to the selected entity, enter a search term to be incorporated to the browsing or do both at once. If at least one of those actions is triggered, the algorithm would update the query in the state of the relevant component appropriately.

For performance tuning, bursts of requests being made to the MusicBrainz web service need to be avoided since it would waste a lot of time waiting for the responses of previous requests to come. Hence, the code was written in such a manner that at one moment at most one active request is being made to the web service. This was achieved by maintaining a variable that holds a reference to the current AJAX request in operation if there is any, and for each request that is being triggered, the existing non null request is aborted and the new request is stored in that variable. If the current sent request fails for some reason, a retry link along with the reason for failure would be displayed. Once the results are visible, the user has the option of traversing through the different pages of the results and if such a request is made, the query is updated with appropriate offset and a new request is made to the web service following the same procedure described earlier.

B. Main Interfaces

The web interface would contain a lot of dynamic content. But the basic layout of the interface will contain a pane on the left which will contain a list of categories the MB database can be browsed with. On top of that list would be a text input field that would allow the user to specify a search term to be included in the query to be sent to the MB Web Service. Clicking on a certain category would summon another panel under it that has further fields that can be selected to narrow down the results with respect to the selected category. Clicking on a category would show up all the results that match that category without any constraints on the right pane of the page. Clicking on one or more subfields would narrow down that browsing result to match all those selected fields.

Sub fields will be basically of five categories as per the requirement. First is the type that has predetermined set of values, which are fetched from the database directly. The second type is a dynamic set of values which include a text input at the bottom allowing the users to type in the text input and add what they have typed to the list. The third type is a field that allows input of range of dates. Next type is a field that allows entering a count of something that accepts number ranges as well as specific numbers. The final one is a list of items displayed as a drop down list. The results will be displayed in a tabular manner and pagination functionality is available on top of the results table. Some other information like the total number of results matched and the date and time the last request was made to the web service also is visible above the table containing the results. Another label would dynamically pop up notifying the user to wait in instances where the interface is doing some work in the background.

Initially, the interface does not support browsing through all the entities at once and *Search Box* is kept disabled. Once the user has selected an entity to browse the database with, the search box becomes enabled and a panel containing the options relevant to that entity comes up. It can be clearly seen in Fig. 5, that the user has selected options to browse the database requesting for artists of type person who are from Sri Lanka.

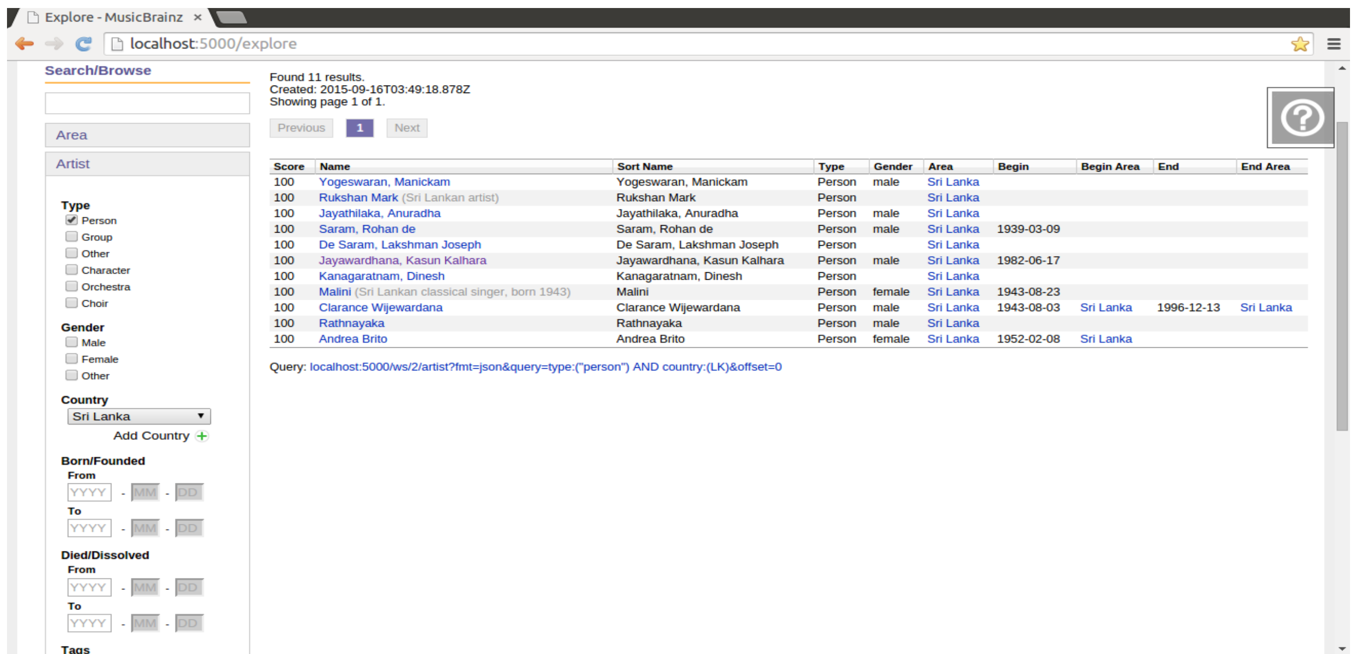


Fig. 5. The interface after a serving for a browse query.

The query that corresponds to the request in Lucene syntax is visible under the results table. Since the number of results returned for this query is eleven, a single page exists and hence the Next and Previous buttons of the pagination functionality are kept disabled and the current page 1, is highlighted.

V. System Testing and Analysis

The testing was done under four major techniques as Functional testing, UI testing, Performance testing, and Security testing. Functional testing of the unified search and browse interface was done targeting at the testing requirements that directly map to the use cases and business functions of the project. These tests verify proper data acceptance, processing, retrieval and the appropriate implementation of business rules. The tests were carried out by interacting with the UI of the interface and analyzing the changes happening in the mark-up of the interface. The strategy that was used for the above mentioned technique was as follows. The React Test Utilities provided by the React library allows simulation of events as if they were triggered by a user. This way it was possible to programmatically input data and trigger click events in the application interface and then using a test harness, it was possible to check whether the mark-up of the page is changed in response to the simulated event.

UI testing was used to verify a user's interaction with the interface ensuring that the UI provides the user with the appropriate access and navigation through the functions of the interface. Additionally, it involved in testing the interface for cross browser compatibility since the audience for this interface includes the whole public community and each of them might be accessing the interface from various different browsers and versions. The approach used for UI testing was to create tests to simulate the relevant events like key presses and mouse clicks on the target elements and check if the mark-up of the interface as well as the query changed appropriately. The strategy used to achieve that was to simulate events using the React Test Utils library [15] and programmatically input data and trigger events in the application interface. Then using a test harness, it was possible to check and see if the mark-up of the page changed in response to the simulated event as expected. The PhantomJS headless browser [16] was used as the environment for the mark-up to load and scripts to run and the test results were observed via its command line console.

For cross browser compatibility testing, the page was accessed using the BrowserStack online tool [17] manually and made sure the interface conformed to the expected standard in all the available browser environments. The tool allows selection of a browser with a particular version to navigate to the target-of-test and test the functionality of the interface in different browsers manually.

In performance profiling, the response times and other time-sensitive requirements were measured and evaluated there by verifying the performance requirements of the interface. The main approach used in stress performance testing was to simulate browsing requests for a certain entity in a burst and observe the time taken to respond for the last request and the number of requests sent to the web service while processing the burst of requests. The strategy that was used here was quite the same as with the prior techniques. Here the inputs were triggered in a burst. The test harness was used to observe if the response to the last one of the burst of requests is received within an expected period of time. Furthermore the number of requests sent could be determined by the number of times the results table mark-up shown in the interface changed itself. The interface also included a caching mechanism for all the pages retrieved for a specific query. This will improve the performance of a user moving back and forth in the pages in the paginated set of results since no communication with the web service is made for viewing an already loaded result page.

Apart from stress testing, Fig. 6 shows the variation of the average response time of the interface as the complexity of the browsing query changes with respect to the number of sub-fields selected under the field 'Type' for category Artists. Fig. 7 shows the same comparison with respect to the number of fields browsed through with only one sub-field selected for the same category.

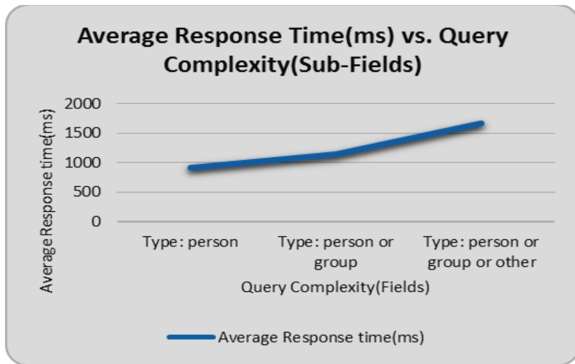


Fig. 6. The average response time variation of the interface as the browsing query complexity changes with respect to fields.

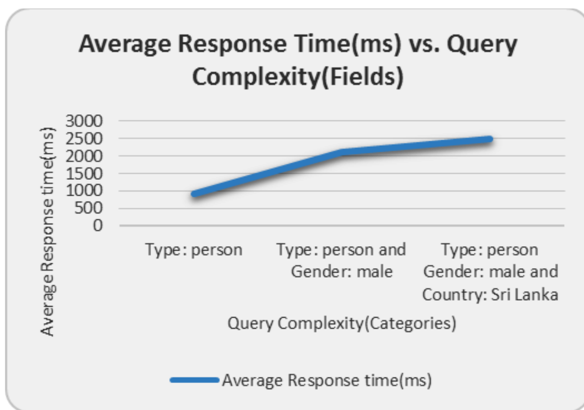


Fig. 7. The average response time variation of the interface as the browsing query complexity changes with respect to categories.

System security was tested by means of user inputs. The relevant test case uses React Test Utils to input strings to the interface that has characters which would invalidate the query if incorporated without escaping them. Once they are input, JQuery [18] is used to see if the updated query contains the exact same input but with the special characters escaped. This way the interface is made very much immune to possible code injection attacks [20]. Since the interface only deals with reading information from the web service, it is hardly possible for any other security threat to exist that would cause any considerable damage to the functionality of the system.

VI. Conclusion and Future Work

This paper presents a unified search and browse interface tool to overcome the search difficulties in existing MusicBrainz music encyclopedia web site. The tool is implemented using a user friendly wrapper through web services. The user feedback shows that the interface provides a satisfactory level of user experience and a user friendly search facilities. This tool enables the exposure of the metadata in the MusicBrainz database to many users and will be available in MusicBrainz beta site [19]. The tool has the strict requirement of selecting an entity first before starting a search task. As a future extension, the above restriction can be addressed using the MusicBrainz server side web services. Also, the caching mechanism can be improved by using a separate server to cache all the retrieved pages. Further, we suggest including a complete set of details for the recent database records to improve search facilities.

References

[1] Musicbrainz.org, 'MusicBrainz - The Open Music Encyclopedia', 2015. Available: https://musicbrainz.org/. [Accessed: 26- Oct- 2015].

[2] Metabrainz.org, 'MetaBrainz Foundation', 2015. Available: https://metabrainz.org/. [Accessed: 27- Oct- 2015].

[3] Musicbrainz.org, 'Search - MusicBrainz', 2015. Available: https://musicbrainz.org/search. [Accessed: 16- Sep- 2015].

[4] Musicbrainz.org, 'Indexed Search Syntax - MusicBrainz', 2015. Available: https://musicbrainz.org/doc/Indexed_Search_Syntax. [Accessed: 16- Sep- 2015].

[5] Lucene.apache.org, 'Apache Lucene - Welcome to Apache Lucene', 2015. Available: https://lucene.apache.org/. [Accessed: 16- Sep- 2015].

[6] Musicbrainz.org, 'Development / XML Web Service / Version 2 - MusicBrainz', 2015. Available: http://musicbrainz.org/doc/Development/XML_Web_Service/Version_2. [Accessed: 16- Sep- 2015].

[7] R. Wickramasinghe, 'Unified Search and Browse Interface Prototype', 2015. Available: http://musicbrainzexploremockup-ruchiranga.rhcloud.com/. [Accessed: 16- Sep- 2015].

[8] Knockoutjs.com, 'Knockout : Home', 2015. Available: http://knockoutjs.com/. [Accessed: 27- Oct- 2015].

[9] Facebook.github.io, 'A JavaScript library for building user interfaces | React', 2015. Available: https://facebook.github.io/react/. [Accessed: 27- Oct- 2015].

[10] Codementor.io, 'React vs AngularJS vs KnockoutJS: a Performance Comparison | Codementor', 2015. Available: https://www.codementor.io/reactjs/tutorial/reactjs-vs-angular-js-performance-comparison-knockout. [Accessed: 27- Oct- 2015].

[11] Jsx.github.io, 'JSX - a faster, safer, easier JavaScript', 2015. Available: https://jsx.github.io/. [Accessed: 27- Oct- 2015].Dfdf

[12] Lodash.com, 'lodash', 2015. Available: https://lodash.com/. [Accessed: 27- Oct- 2015].

[13] Wikipedia, 'Less (stylesheet language)', 2015. Available: https://en.wikipedia.org/wiki/Less_(stylesheet_language). [Accessed: 16- Sep- 2015].

[14] Sublimetext.com, 'Sublime Text: The text editor you'll fall in love with', 2015. Available: http://www.sublimetext.com/. [Accessed: 27- Oct- 2015].

[15] Facebook.github.io, 'Test Utilities | React', 2015. Available: https://facebook.github.io/react/docs/test-utils.html. [Accessed: 27- Oct- 2015].

[16] Phantomjs.org, 'PhantomJS | PhantomJS', 2015. Available: http://phantomjs.org/. [Accessed: 27- Oct- 2015].

[17] Browserstack.com, 'Cross Browser Testing Tool. 300+ Browsers, Mobile, Real IE.', 2015. Available: https://www.browserstack.com/. [Accessed: 27- Oct- 2015].

[18] j. jquery.org, 'jQuery', Jquery.com, 2015. Available: https://jquery.com/. [Accessed: 27- Oct- 2015].

[19] Beta.musicbrainz.org, 'MusicBrainz - The Open Music Encyclopedia', 2015. Available: https://beta.musicbrainz.org/. [Accessed: 27- Oct- 2015].

[20] Wikipedia, "Code injection", 2016. Available: https://en.wikipedia.org/wiki/Code_injection. [Accessed: 07- Feb- 2016].