University of Sheffield

COM3504

Intelligent Web

Report

Team: Northern Lights

Petar Barzev
Blagoslav Mihaylov
Mikhail Molotkov

23rd May 2017

Table of Contents

1 INTRODUCTION

This report gives a thorough description of the Northern Lights team solution. The full requirements of the project have been implemented. This includes a client-side web-based interface, a server with a database, and a mobile app. The front-end is Single Page Application built using React.js and the AntDesign UI. The server-side runs on NodeJS 6.9.5 using Express.js as a framework with SQLite as the database engine. Additionally, several different packages are used to provide the functionality, including a package for handling Twitter API calls and one for SPARQL queries. The mobile app was built using Cordova and implements much of the functionality of the web app. In designing the solution, we have aimed for good capabilities, high performance and user friendliness. Hopefully, we have achieved this goal.

2 QUERYING THE SOCIAL WEB ABOUT FOOTBALL TRANSFERS

Solution - To make it possible to query social media, we designed a system that interfaces with Twitter through their API to provide information about various football-related topics. This is done by issuing a **query.** The queries are issued by the **client** and are processed by the **server.** This communication is achieved using Socket.io and passes queries and results back and forth, as well as any possible errors which might arise.

There are three main sources of information that the user can query: the Twitter REST API, the Streaming API and a Database, which is maintained on our server (read Chapter 3). If querying the REST API, the server calls Twitter to receive tweets going back in time, up to a maximum of 7 days. The tweets are received in batches of a 100, up to a maximum total of 600. The streaming API sends back one tweet at a time, while the Database returns as many as are found to match the query.

The same query structure can be used on all three of these sources. The query must contain several keywords separated by spaces. Additionally, logical operators can be used, such as AND/OR. A space is usually understood as an AND, while a comma is understood as an OR. Additionally it is possible to target only tweets posted by a specific user. This is done using BY. An example query would be “#manutd OR #chelsea BY @WayneRooney”.

The front-end has been built in React and the AntDesign UI library. The design of the UI is clean and simple, allowing the user to easily query and read the returned data. The Single Page app is split into two sections that represent two Routes, namely Feed and Database. The first section allows the user to search, stream and track statistics of transfer rumours. The second one allows the user to query the database. A common element across these two routes is the Search bar that allows the user to enter a query as per the requirements. A hint element next to the Search bar reveals a sample query in case the user is dubious.

The reason we used React + ES6 is because of the highly-organised and high-performance nature of the framework. Code is easier to maintain and the application is thus scalable. In terms of UI, AntDesign provides a clean and professional look that has been tested extensively by the React community.

Issues - The main issue with the project was the tendency of Twitter to rate-limit the app. In order to protect their servers, they have a system where apps are blocked for a certain period of time if they make too many requests. Finding ways around this was the biggest challenge of the project.

Also, the Twitter API is not very extensive. They usually return only a subset of all of the tweets. The search function could only retrieve a maximum of 100 tweets, so it had to be called more than once per query. Also, where tweets were streamed too fast, it made the UI laggy and unresponsive. For this reason, the number of tweets that can be streamed at each second was limited by the server.

Requirements - This implementation satisfies the requirements defined for the project. It is capable of responding to queries, both through the REST-ful API and the Streaming API. By typing the correct query, the user is able to search using multiple keywords separated by logical operators, as well as search by the author of the tweets.

Limitations - the main limitations are all due to the limitations of the Twitter API itself. Firstly, tweets beyond 7 days ago are not available through their API. The ones that are available, are only a subset of all the tweets that correspond to the query. For the streaming API, a similar limitation is present. For queries where many tweets are coming in per second, it is not possible to display all of them, as the amount of information is too great.

2.1 Output: Tweets

Solution - To display individual tweets, a TwitterCard component was designed. It presents the data in a user-friendly and structured way. The card contains all necessary fields as specified by the assignment sheet. When the Twitter Search API is used, a maximum of 600 tweet cards are returned. Because this is a very high volume of data and UI components, a Pagination mechanism had to be implemented. Thus, each page only displays 10 cards, which results in a clear and swift UX. In the case of the Streaming section, the server throttles the output to 1 tweet per second. It is then sent to the client and displayed to the user, up to a maximum of 50 cards on the screen. When data is loading, a 'pulsating' loading card component is presented to the user - to let them know that the server is currently preparing their results and they will arrive shortly.

Issues - Several issues and challenges were encountered. First, creating a visually-pleasing, user-friendly and self-explanatory component to represent a tweet. Second, presenting 600 or more (in the case of the database) tweets to the user needed to be done in a way that does not slow performance or make app hard to use (long scroll, freezing, etc.). We solved that by utilising a Pagination. Third, when streaming data on popular key word (such as 'football') data had to be throttled to 1 tweet per second and the stream cards are limited to 50 at once. When that limit is reached, a card is removed from the bottom of the stack every time a new card comes in. Fourth, when a tweet contains a link we needed a way to make it clickable. We solved that by parsing the data via Linkify to generate anchor tags whenever a link is detected.

Requirements - Our design fully complies and meets all requirements specified in the assignment sheet. Presented are the author, text, time/date, link to original message, as well as a link to the author's page. Data is presented to the user in a structured and visually-appealing way.

Limitations - An exceptional situation where our solution might be limited is if Twitter change their twitter object structure and fields. We have not utilised an object analysing mechanism to feed to detect object fields and structure changes. Other than that, we believe our solution is extensible in terms of UI design, as the same card UI can be used for other sources of data (e.g. Facebook posts). However, it is important to note that a different a more general and advanced object property parsing mechanism would need to be developed in order to handle the intricacies of every source API.

2.2 Frequency

Solution - The app displays a frequency graph regarding a given query. The frequency is calculated by the server which counts the number of tweets issued at each date. It then creates a hashmap which it sends back to the client. The client then uses it to display a graph, which is implemented in a ChartJS 2 React component, namely 'react-chartjs2'.

Issues - The main issue with implementing the frequency graph was finding enough information, which spans several days, so that a meaningful graph can be produced. This can be difficult as the Twitter API only returns 100 tweets at a time and only from the past 7 days. Getting information that goes further back is not possible without using data from the database. However the data from the Database would not be very reliable for showing the distribution, as it is not an extensive list of results for a given query.

Requirements - The requirements are evidently satisfied as the app is able to track the frequency of tweets for a given query and display them in a graph.

Limitations - the main limitation is that depending on the volume of the tweets, a meaningful graph might not be available. For example, if the last 600 results were all posted in the same day, the graph would only show a dot in the middle. A solution to this might have been to get more than 600 tweets but due to the limitations of the Twitter API that approach was decided to be infeasible.

3 STORING INFORMATION

Solution - A database was implemented using SQLite. Whenever tweets are retrieved using the Twitter API, they are automatically saved in the database. This creates a record of all of the tweets ever obtained by the app and also removes the need to query Twitter every time a duplicate query is received. This helps us avoid getting rate-limited. All queries ever run are stored in the Database along with a timestamp of the last time they were received. If the server receives a query which was recently (in the last 5 minutes) run, it returns the results from the database. If more than 5 minutes have passed, it uses the Twitter API. Additionally, the user can explicitly query the Database. The client specifies this by changing a flag in the JSON. This forces the server to return tweets only from the database.

Issues - Storing information was not incredibly difficult. It was important to avoid storing duplicates of tweets, which was done by looking if a tweet with a given ID is already present in the Database. Also, the asynchronous nature of Javascript made it difficult to run SQL queries but by adopting proper serialization techniques, these problems were avoided.

Requirements - the requirements were satisfied as the app is able to return tweets from the database on demand as well as return cached tweets.

Limitations - the obvious limitations are that the database cannot possibly store a representative set of all tweets for a given query. It only stores the tweets that were ever retrieved by a user query.

4 PRODUCING A MOBILE APP FOR SPORTS ENTHUSIASTS

Solution - The flow of the application is straightforward: when a user types in the search query to either Search or Streaming, the request is sent to the server, where this query is processed. The server then issues the tweets to the mobile application. If Streaming was selected, the server opens a stream channel, so that the mobile application would receive requested tweets.

Moreover, the mobile application has its own database, which gets updated every time the user sends a query. When the data is received, it is sent to the mobile database. It may then be retrieved by the user later on when an Internet or Wi-Fi connection is unavailable. This ensures that the mobile application will not use a lot of data (to save the user's money and allowance) and ensure that all heavy computations and sorting are done on the server side, rather than on the client device. The mobile application is only responsible for sending queries to server, storing the data and displaying tweets to the user.

Issues - The most difficult part was to create simple and good-looking UI, so that the user could have an outstanding experience using the mobile application. Other than that, the core functionality was implemented without any problems. Therefore, the mobile application at its current state, is fully functional and ready to use.

Requirements - Initial requirements, that were listed in the assignment specifications are presented below:

1. A human computer interface that allows querying Twitter Search and Streaming API's
2. A local database to store the data locally so to avoid repeating the queries

Both above features were implemented and tested on real device - Nexus 5X, Android 7.1.2 and emulator - Nexus 7, Android 5.0.
Additional functionality was developed to enhance user experience: when the user sends a query to about a player, ie "@WayneRooney", the application would display information about the player, including a photo. This function works in a similar manner as the one in the web application. Moreover, it was decided to implement geolocation suggestions, ie if the user is currently in Sheffield, the application would suggest to search for '@swfc' or '@SUFC_tweets'.

Limitations - Present solution has few limitations as the application architecture and implementation are simple and straightforward. However there are some limitations caused by framework. Cordova does not allow access to the native UI components and uses WebViews. This brings performance limitations, that could be avoided if mobile application is to be re-written in pure Java/Kotlin or different framework, such as React Native. For example, it is currently very difficult to implement 'lazy load' which means that the application will experience performance drops when processing large volumes of data. In pure Java a solution would be simple and elegant, but since Cordova does not provide an access to native Android components, it is not possible to implement it in such a way.

5 WORKING WITH THE WEB OF DATA

Solution - Every query that the server receives is carefully analyzed in order to find names of football players. If the server finds a match, it sends a SPARQL query to DBPedia to retrieve information about the player. This is then sent to the client, processed and supplied to a PlayerCard component. This component presents the data in a visually-appealing and user-friendly way. The design follows the colouring scheme implemented by

the app. Extracted and presented is the player name, date of birth, current club and field position. Also, a short paragraph made up of usually 3 (maximum 4) sentences that gives some information to the user.

Issues - The main challenge here was organising the player data in the database against which queries are matched and thus SPARQL queries are created. Also, since DBPedia usually takes long to return a data, a loading card is displayed to let the user know that data is coming. If DBPedia cannot produce the information required by the server, an error is generated and pushed to the clients. The web client handles this by notifying the user.

Requirements - All spec-sheet requirements are met. Data for Manchester United FC and Chelsea FC has been added to the database. Keywords are matched against user queries, a SPARQL query is generated to query DBPedia, data is retrieved by the server, pushed to the client and presented to the user in a visually-appealing and structured way.

Limitations - A perfect solution would be to return player information for every player in the world. This might be achieved by parsing a third-party player data to generate keywords to match the user queries against. Also, since the player images are of different height, the current solution crops images from the bottom by limiting the player card height to 200 pixels. This means that on screens larger than 25-27", sometimes player images are cropped way too high and thus crops out most of the image. This can potentially be solved by an advanced image-scaling algorithm, to make sure all images are converted to a specific height first.

6 CONCLUSIONS

We are proud of our final assignment solution, but we have learned some important lessons:

- A clear and detailed plan on how a project such as this will be tackled is a MUST.
- Separation of roles, responsibilities and expectations is CRUCIAL.
- Separating work on front, server and mobile side sections works great for large projects such as this one.
- Frequent meetings, use of collaboration channels (e.g. Slack, GitHub) is crucial to enable a productive and effective workflow.

7 DIVISION OF WORK

We have shared the workload in a balanced way. Thus, we have agreed to have an equal mark allocation. Following is an overview of work done by each member:

Petar Barzev:

- Full web app implementation. Set up the React architecture, AntDesign UI and Sockets.io.
- Created a search interface for sending queries to the server.
- Implemented a TweetCard display for showing individual tweets.
- Used a Pagination to control the amount of tweets shown at a time.
- Added a chart for displaying the frequency of the tweets.
- Created functionality for displaying player information using a PlayerCard.
- Added loading animations for both Player and Tweet cards.
- Optimized the performance of the webapp.

- Created the initial implementation of the server side.

Blagoslav Mihaylov:

- Implemented most of the server-side.
- Created all of the database functionality.
- Added the possibility of getting data from DBPedia using SPARQL queries.
- Implemented most of the Twitter API functionality.
- Added a frequency counter for the tweets.
- Added error handling and introduced error objects in client-server communication.
- Various tweaks and fixes.

Mikhail Molotkov:

- Full mobile application, including:
  - UI design
  - Error handling
  - Setting up connection between mobile app and the server
  - Creating local DB and strategy to save tweets to that database
  - Adopting 'Player Card' functionality from Web App
  - Geolocation suggestions
- Initial project template and server code

8 EXTRA INFORMATION

Following is information how to run the solution:

8.1 Web client and server:

a. Navigate to './solution/client' and execute 'npm install'
b. Navigate to './solution/server' and execute 'npm install'
c. Execute 'npm start' in server folder

8.2 Mobile:

1. Start the server, by going to 'solution/server' and running 'npm start'
2. Open 'localhost:3333' to the Internet (you may use ngrok for that purpose). After you do that, insert your host_name (server address) into placeholders in 'solution/mobile/www/js/index.js' (top of the file) and 'solution/mobile/www/index.html' (bottom of the file, look for <script> element under 'Socket dependencies' comment), so mobile application can connect to the server.
3. If you do not have an Android device in your possession, then ensure that you have Android SDK, AVD and emulator is set, in order to run emulator on your computer (If you do not have environment set, please refer to https://cordova.apache.org/docs/en/latest/guide/platforms/android/).
4. Once you choose your option to run application , go to 'solution/mobile' folder and run 'cordova prepare' & 'cordova run android'