Group:

Ben Clegg (aca14bsc), Sam Dickinson (coa12sd), Rob Ede (aca14re)

Introduction

In this assignment we are using multiple modules to improve the quality of our application.
Twit is a Node.js library that allows us to query the Twitter API, and stream new tweets. This is used for all of the server's interactions with Twitter so is an integral part of this project.
MySQL is used as a database solution, to cache tweets and store searches.
Express was chosen, since it is an existing, well-documented web framework, which removes the need to manually configure a considerable amount of basic functionality.[3]
Moment.js allows for the formatting of date and time from the Twitter data.
Pug (Jade) is a templating engine, used as an alternative to writing pure HTML, as it allows for HTML page generation based upon data.[1]
Sockets.io allows data to be sent between the client and server, by sending and receiving particular events. This is used for all client-server interactions, allowing for dynamic pages.[3]
A "promises" library (any-promise) was used, which allows for subroutines to be executed without the rest of the program having to wait for a result, allowing for results to be provided when they're ready, without slowing down the system.
In order to provide access to DBPedia, the "sparql" module is used, allowing for SPARQL queries to be submitted to obtain player information.[4]
The application runs on Node versions 0.12.4 and above.

1.1 Querying the Social Web about Football Transfers

Solution

Queries are submitted by the user, entering comma-separated subterms for each of the terms (player, team, and author). A checkbox is used to select whether the terms should be combined as OR or AND.
This is stored in a database, as a "search", including each term, the OR/AND search mode, and a unique ID.
When showing tweets, these stored terms are retrieved from the relevant search, using its unique ID. This ID is provided to the system either when the search is created, or when the search is selected from a list of existing searches.
When clicking the "Get Remote Tweets" button in existing searches, these terms are used to construct two Twitter API queries, for the Search and Streaming APIs, respectively.
For the search query, each term is surrounded by brackets, and its subterms are combined with ORs in place of their input comma separations. The author subterms are also prepended with "from:" and surrounded by brackets. The terms are combined together with the AND/OR search mode boolean operators. The results of this query are retrieved from the API, and sent to the client via a socket event. This socket event adds the result tweets to the webpage, including their content, date, time, author's screenname, and links to the original tweet and author's account.
A similar approach is taken for the streaming API, except the query is constructed in a different manner (as shown in "Issues"), and a stream is started. When this stream receives a newly made tweet (as per the query), a socket event is sent, adding the tweet to the webpage.

Additionally, the dates of all the tweets are grouped, with the number of tweets for each day are counted and shown on the page dynamically. This information is also displayed on a graph.

Issues

Twitter Search API queries can sometimes be considered to be relatively inconsistent. For example, “(X) OR ()” is a valid query, returning tweets containing X, yet “(X) OR () OR (from:Y)” does not return any tweets, even if Y made a tweet containing X. This was resolved by experimenting with various different query forms, and finding which forms worked, alongside how to construct them, for example by checking for empty terms and ignoring them.
The Twitter stream queries are limited to only simple boolean operators (effectively OR/AND). Whilst OR mode queries are still trivial, AND term joining is more complex, requiring every combination of subterms to be joined together iteratively, as ORs of ANDs. Twitter’s 100 tweets limit per API call made it slightly tricky to return the required 300 tweets to the page. However, with a bit of careful promise chaining, we were able query twitter again and again, immediately after each request had returned.

Requirements

All requirements were fulfilled for this section.

Limitations

The access to the Twitter API will stop looking for more tweets after over 300 have been found (per the specification), to prevent the API limits from being reached. This does have the effect that some tweets will be missed, but this is preferable from being blocked from making further queries.
The search API will search for tweets made since seven days ago, however in cases where either too few tweets exist, or too many tweets exist (eg thousands of matching tweets in the past day), not all days will be fully populated. However, AND queries tend to be more appropriate in this regard.
A search can be added, even if an identical search exists. We attempted to implement a system to detect identical searches, and successfully used SQL REGEXPs and statements to find existing searches that contain all of the terms and subterms. However, searches which contained additional subterms also (incorrectly) matched, and this could not be overcome. This does not have a significant impact since the user can see existing searches already, and there is therefore no purpose in them making a new one.

1.2 Storing Information

Solution

Whenever a tweet is received, it is added to a “Tweets” table in our SQL database, with its content, author, tweet id, date and time, and the id of the search it belongs to.
The largest Twitter id of the retrieved tweets is stored in their search, as the “newestTweet” value.
When a search is selected, the tweets stored in the database which belong to it are automatically retrieved. To query Twitter, the user simply clicks the “Get Remote Tweets” button, unless the search is new, in which case this is performed automatically.
When Twitter is queried, the query is set with a “since_id” equal to that of the search’s newestTweet. This prevents unnecessary access to the API from being made, as only tweets

newer than the latest stored tweet are queried from Twitter, and therefore no tweets are repeated.

Issues

Tweet IDs are around 18 digits long which greatly exceeds JavaScript's 32-bit integer limit. The database schema defines Tweet IDs as BIGINTs. For a while we were using the "id" field returned from the API which would have been bitwise truncated to the nearest 32-bit representation and stored incorrectly. This was resolved by using the "id_str" field instead which encoded the ID as a string which could be stored correctly (as a BIGINT). One further problem we discovered later was that the database was not casting its BIGINTs to strings as seemed to be the case in the documentation. This was resolved by setting two flags when a connection to the database was created, "bigNumberStrings: true" and "supportBigNumbers: true".

Requirements

We believe that the requirements were fulfilled to the best of our ability.

Limitations

Since multiple "searches" are used, which may have an overlap in resulting tweets (for example, "rooney manutd" would match against "rooney" and "manutd"). Therefore, some tweets will be duplicated in the database, but not in the results.

1.3 Producing a Mobile App for Sports Enthusiasts

Solution

We created a Cordova hybrid mobile app, using a socket based architecture.[2][3] This was a good design choice because or webapp already communicated with the server using sockets so part of the functionality was already there. The solution used socket to connect to the server to retrieve the trackings and tweets and then to query twitter for more tweets. The mobile app does create a local database but the functionality for this was not fully completed so the system uses the server database solely. The app allows the user to create trackings and view existing trackings giving the same list of tweets the webapp would show. It does this using three separate tabs, one for all trackings, one for making a new tracking and one for the results. The results tab is hidden when the user is not on it.

Issues

For socket to connect to the locally hosted server the socket needed to connect to a specific address to connect outside of the emulator. The local SQL database was difficult to sync to the server database. Cordova only allows for one html page so a way to display all the information on the page in a clear manner is needed.

Requirements

The cordova app does not meet all the requirements. The app has the functionality to allow a user to interface with the query system and get results of tweets being able to view them all going into the server database. The local database was not fully implemented. The tables were

created successfully and could sync on startup but information could not be retrieved in time so the server database is solely used

Limitations

A limit of the solution is the fact all tweets for a query are displayed in a long list meaning large query results can be difficult to scroll down through. The frequency over the last week is not displayed as it not part of the requirements.

1.4 Working with the Web of Data

Solution

A table of Man. Utd. and Chelsea players' real names and Twitter screennames was manually created in the database.
The server checks if a search's player or author term contains any of these screennames when the search is selected. For every screenname found, the player's real name is used to query DBPedia using SPARQL[4], finding their team, date of birth, position, name, and an image.[6] The first result for each player is sent to the webapp via a socket, and added to a div at the top of the page accordingly.

Issues

We found it particularly challenging to find which properties and classes could be used. For example, the class for Footballer is "yago:FootballPlayer110101634", which is understandably difficult to find. The properties were found using Wayne Rooney's DBPedia entry.[5]

Requirements

The requirements were fulfilled.

Limitations

Some players cannot be retrieved since they do not have DBPedia entries. Similarly, some players could in theory match against another player with the same name.
Some results have outdated information (such as their team), due to DBPedia being outdated.
Queries have a wide range of response times, from near instantaneous, to several minutes, or sometimes not being received.
These limitations are out of our control, however.

1.5 Quality of the Solution

Solution

In our final solution we used socket.io for the client to server communication. This exchange is JSON based for simplicity.

Issues

An issue was we had to find a high quality way of communicating between the webapp/mobile app and server. This was achieved using socket.io with JSON.

Requirements

We feel our implementation fully completes the requirements outlined in 1.5 and our system communicates in a quality manner.

Limitations

The solution can be easily adapted for more requirements as more sockets can be written for new functionality.

Conclusions

We encountered many challenges but we feel we have overcome most and as a group we are happy with what has been produced.

Division of Work

We all worked well as a group together. This assignment involved lots of group programming sessions where we would do pair programming or generally bounce ideas to and from each other. Ben and Rob primarily focussed on the querying and storing information while Sam worked on the frequency task making a chart. Then Ben worked on displaying the DBPedia functionality. Sam worked on the Cordova implementation. And Rob worked on cleaning up the first few sections and worked on the style of the web app. We feel we were a good group and each contributed their fair share.

Extra Information

The project includes a README file, which repeats this information.
Our server includes the webapp, whilst the Cordova client is separate.
A custom database and Twitter client can be configured by editing the server's config.json accordingly, yet these are already configured with our own credentials.
The database schema is databaseschema/dbsetup.sql
To use the application, run 'node app.js', and open 'localhost:3000' in your web browser.
The server must be running for the Cordova app to function.
In the event that modules do not run correctly, please run 'npm install'

Bibliography

[1] Lecture slides 2, by Fabio Ciravegna for Module COM3504: The Intelligent Web
[2] Lecture slides 4, by Fabio Ciravegna for Module COM3504: The Intelligent Web
[3] Lecture slides 5, by Fabio Ciravegna for Module COM3504: The Intelligent Web
[4] Lecture slides 9, by Fabio Ciravegna for Module COM3504: The Intelligent Web
[5] http://dbpedia.org/page/Wayne_Rooney
[6] https://stackoverflow.com/a/31322930