Report for COM3504 Intelligent Web – Social Tracker

James Foster, James McIlveen, Thomas Miller

To execute the Node.js server, navigate to the /solution directory in the console and use the command:

node app.js

Introduction

For this project we created a web application in HTML and JavaScript which allows the user to access data streams from services such as Twitter and Foursquare by communicating queries over JSON to our Node.JS server and MySQL database. The project will also provide metadata about the users and topics of discussion, such as user location over time, information posted within a given time period or physical geographic area, or information posted about specific events or to specific hashtags.

Section 1: Querying the social Web

This section will discuss the communication between our client, server, and the social networks we interact with to retrieve the data we present to the user.

1.1: Issues

Our project allows the querying of user data and published posts on Twitter and Foursquare, which meant we needed to learn how to interact with the web services both social networks provide. This necessitated learning the APIs' various functions and procedures, what data to hand to the functions and what to expect back, and how to handle the data they returned. Part of the queries we requested was the geolocation of the tweets and check-ins – the boundaries for which presented a problem, as it was difficult to process the geolocation data correctly, and few libraries exist which could help.

1.2: Design Choices

We use the Twitter REST and Streaming APIs to allow searching and collection of data, filtered by location, time, user, keyword, or hashtag.

We also used the Foursquare REST API to retrieve venue information and user check-in data.

In addition, Google Maps is used to allow the user to search for and input location data in a simple, fast, and intuitive way. Using the interactive map interface, a user can find locations with the map, which are then translated into precise coordinates for use by our system, reducing computational and request load on our server. This also allows conversion between latitude/longitude coordinates and place names using Google's Mapping servers (it is not required that the user log into a Google account by design of the interface, as all queries are publically available – this means we do not need an authentication token when communicating with Google).

1.3: Requirements

We built a Node.JS server which allow the querying of Twitter and Foursquare via a web interface.
We were required to do the following:

1. Allow public discussion tracking via keyword and hashtag searches, outputting a list of messages which can be restricted by venue or by a region (latitude/longitude and radius).
2. Query specific users (accepting ID's or screen names as input) and show a list of common topics (keywords) mentioned in the tweets originating from the given user(s)
3. Search/show where a user has checked in (Foursquare) or tweeted from (Twitter) in the last X days, and points of interest close to these locations. User information is displayed alongside the search results including their name, id (@name on twitter), location if available, and profile picture (if available) – a link to their profile is also provided
4. Query a specific venue to find a list of users who have visited that venue in the last X days.

As such, we have met the requirements laid out in the specification to a high standard.

1.4: Limitations

The API's we have used contain request limits, meaning the number of requests we can make is a given time period is limited. We designed our server in such a way as to minimise the number of requests made to Twitter and Foursquare by ensuring requests are bundled together into one where possible. The API's also limit us in the types of query we can make, for example we cannot request tweets matching a keyword AND a location, and we cannot request the number of favourites a tweet has, or where a user lives. This means our application is unable to access or present this specific data. In addition, use of these API's requires being logged into a Twitter or Foursquare account, so use of our application is severely limited without a valid login.

Section 2: Querying the Web of Data

2.1: Issues

Our application allows a user-venue search, which pulls information about which users have visited which venues recently according to Foursquare. This section will discuss the queries we made to locate nearby points of interest to user-venue search results.

2.2: Design Choices

The 'User Venue' search shows the search results in a table, with a clickable link to pull up the nearby locations to each shown venue. The link will show all nearby points of interest pulled from DBPedia and FourSquare, including an image of the location, its name and description, its coordinates, the distance from the venue, a link to the location's webpage if one is available, and a column to indicate the source of information (usually DBPedia or Foursquare). In addition, it plots each of the nearby locations as pins on the Google Maps interface above the results table for a visual flourish and a clean user experience. This involved taking each result in the user-venue search and requesting the closest ten locations from foursquare and from DBPedia (where closest is defined as within 0.05 degrees of latitude and longitude) and displaying the returned information in a neat table.

2.3: Requirements

We were required to take the output of the user-venue search and locate nearby points of interest using queries to the Web of Data, and display the resulting information to the user in table form and as points on a map. We were also required to include additional information about each location such as a link to the location's webpage.

2.4: Limitations

Our application relies on the 'Web of Data' being accurate and robust, which is not something we can generally count on because the system is in its infancy with little few guarantees of data correctness or availability. This means the information we retrieve should be treated with caution as it could be inaccurate or misleading.

Due to the asynchronous nature of JavaScript's callback architecture, dealing with multiple simultaneous API requests was sometimes difficult. Concurrent returns from the APIs had no guarantee of any given order in which they returned and as such, sometimes made it difficult to match original queries to the returned data. The solution was to embed extra information in the API query parameters that could be obtained in the callback function, but in doing this, we may be exposing the inner workings of our project.

Section 3: Producing Data for the Web

3.1: Issues

This section will discuss the production of RDF data for the Web of Data.

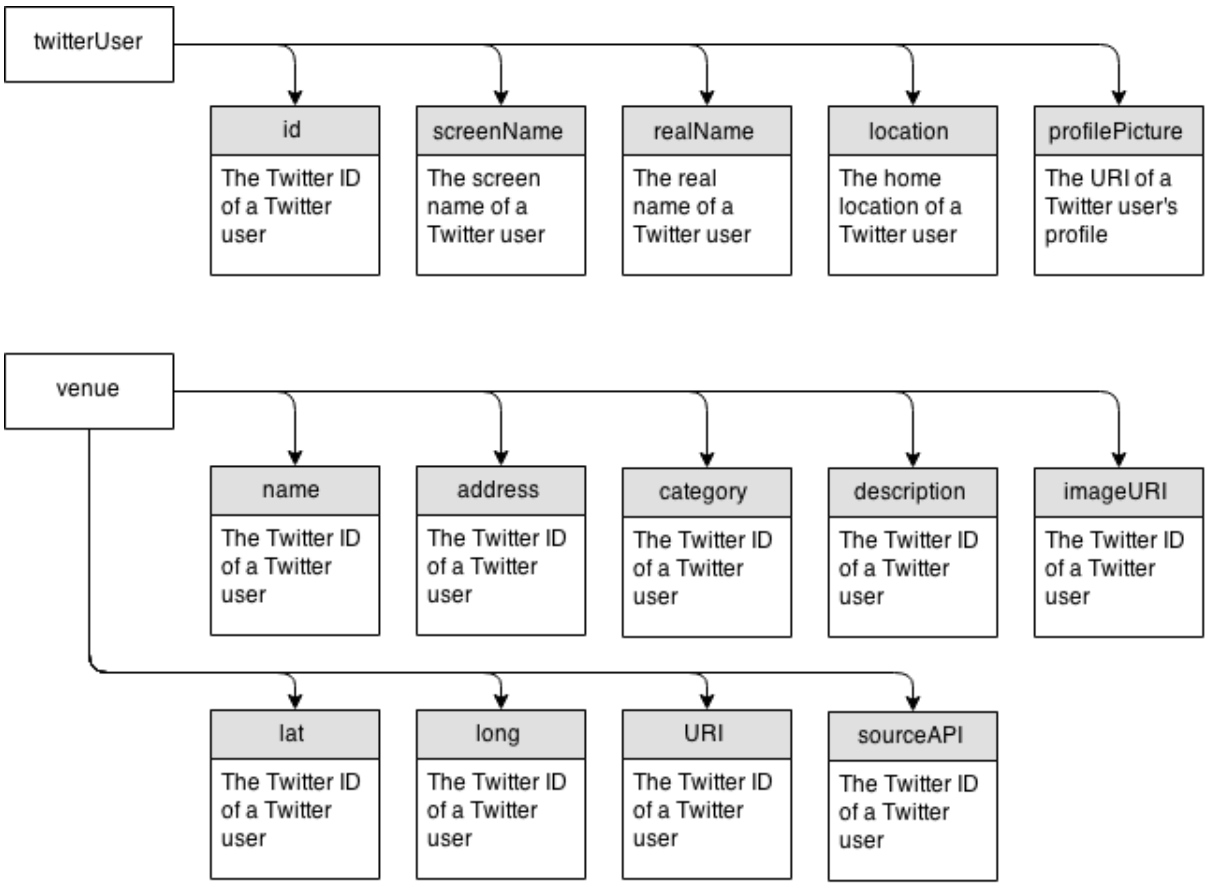
3.2: Design Choices

For the 'User Venue' search, we created a permalink to the search results, which we stored using HTML document with embedded RDFa rather than the creating the page dynamically using AJAX. In this way we

created a persistent, crawlable page. The ontology for the RDFa was stored on the server as an RDFS document.

The RDFS ontology contains a class for a Twitter User, which describes Twitter accounts in a simple way, and a class for a venue, which describes venues found on the search results. The ontology can be found at /public/rdf/rdf.xml.

The created pages have RDFa embedded in the HTML, with each venue entry including: an image of the location, a link to the foursquare or wikipedia page for that location, other information pulled from foursquare such as rating and likes, a lat/long coordinate and address, and finally the number of times that user has visited that location, and the date of their last visit. The RDFa also includes details about the Twitter user in which the check-in information was found from.



3.3: Requirements

We were required to create and RDFS ontology containing venue and user information which was displayed as the result of queries to Twitter and Foursquare. We were also required to add RDFa to the saved search results to make the page crawler-friendly.

3.4: Limitations

It is difficult to check that the RDFa and RDFA is valid when embedded in a page. RDF is a new technology and as such the tools are still underdeveloped, lacking features for debugging and online tools do not work as the content is hosted locally. The only way to test the features was to manually copy and paste the RDF into a validator.

Section 4: Storing Information

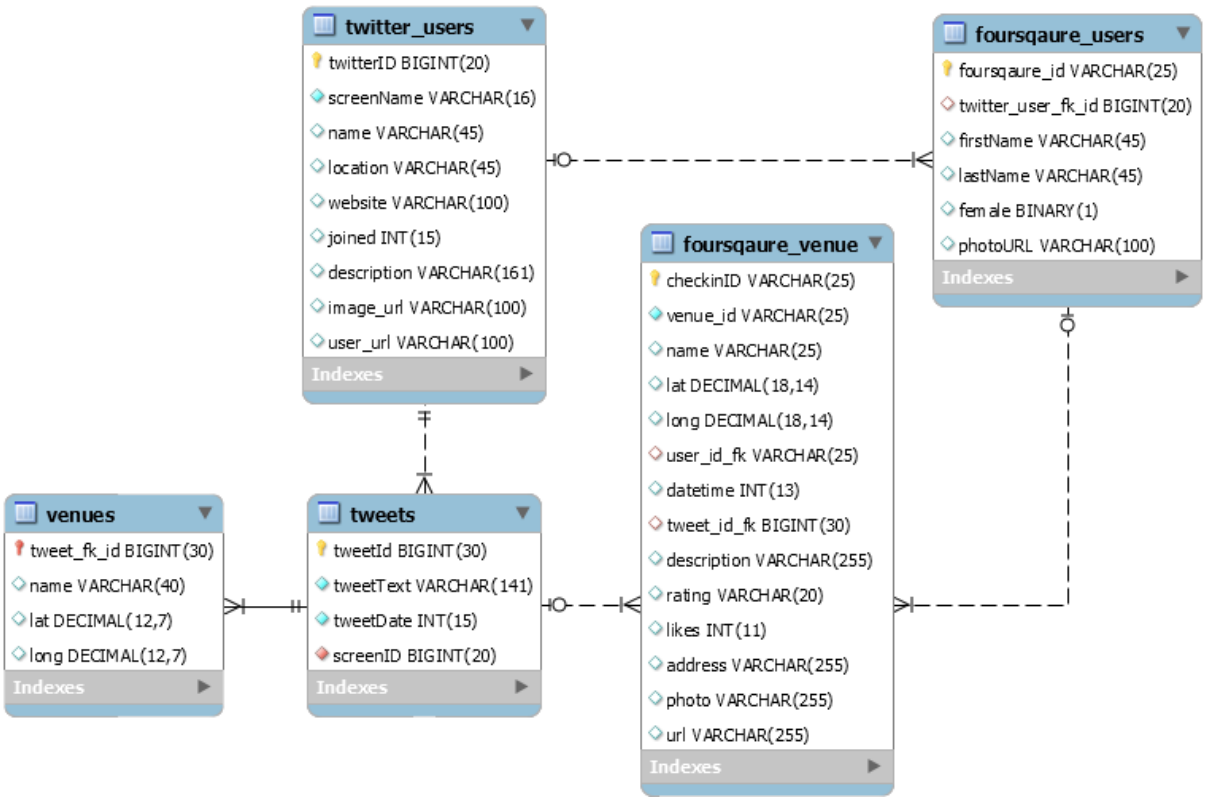
As a part of the assignment, we were required to store information related to tweets and user locations etc in a MySQL database. We store data relating to queries, users, and tweets to speed up future requests and minimise unnecessary traffic sent to twitter and foursquare (repeated queries and function calls).

4.1: Issues & Design

This section of the report will examine the design of the database used to store data for queries used in our application.

The database needed to be designed efficiently to minimise the amount of time taken to execute queries and retrieve information. Since some parts of our application can send hundreds of INSERT commands in a very short time, we found that it is far more efficient to bundle these commands and insert them all at once using a single SQL INSERT INTO statement, rather than using a separate command for each row to be inserted. This efficient database design mean that rows are inserted almost instantaneously rather than the 30-second wait we found when trying to insert many rows individually, each of which required Twitter and Foursquare API calls over the internet, which took a long time to return data.

The database was designed with five tables: FoursquareUser, FoursquareVenue, TwitterUser, Tweet, and TwitterVenue, which respectively store user information retrieved from foursquare, venue and location information retrieved from foursquare, twitter user information, individual tweets, and location information retrieved from Twitter. As a part of Twitter's TOS, the Twitter database is kept private. Foursquare's TOS dictates that data in FourSquare data must be kept fresh and up-to-date. The tables and relationships we chose to use are represented visually below.



4.2: Requirements

The specification required a web-accessible database which enables querying for a username or user ID, and returns the user's location, profile picture if available, home location if available, locations they have visited. This can be achieved on the User Database search tab by entering the username into the search box and viewing the results returned. As such, this section of the specification was fulfilled.

The specification also requires similar functionality for the venues - a web interface must be created to allow searching of venues stored from queries to Foursquare, and the ability to view which users have visited that venue. This is possible through the Venue Database search. For each result, the application shows the username, venue name, a picture of the venue if available, the coordinates of the venue, the number of times that user has visited the venue, and the date of their most recent visit.

4.3: Limitations

In addition, there is redundancy in the venue table as the same venues can appear in the table twice. The primary key of the venue table is the same as the primary key of the tweet, meaning each tweet has a unique and exclusive venue. This is a far simpler solution than matching like venues to one another and allowing venues to share a row in the venue table, but could impact database access speed for certain queries (where the query requests multiple venues which could be saved as the same row).

Section 5: Web Interface

5.1: Issues

In this section we will discuss the interface presented to the user when they request the page from the node.js server. We were required to implement a page in HTML/JavaScript which presented the ability to query social media (Twitter and FourSquare). The interface should allow searching of social networks by keyword, user, location, and time using the API's provided for public use.

5.2: Design Choices

We decided to use a single page for all of the project's functionality. This provides an intuitive and clean experience for the user, and negates the requirement to learn a potentially complex website. It also speeds up the experience of browsing; all functionality is within a click of the mouse, and only new data is downloaded for different searches without the need for re-downloading the entire page.

We implemented a tab interface within the page to allow the user to intuitively switch between the major models of functionality - this allows a tidy GUI with simple, fast, and easy navigation.

We implemented a graphical map using the Google Maps API, which provides a quick glance at the information entered into the form, represented in a graphical way which all users are familiar with.

5.3: Requirements

Our design meets the requirements of being implemented in HTML/JavaScript, allowing control of the application through a web browser interface. The page is implemented over a multitude of JavaScript files and .ejs files which are sent in HTML to the client side when requested. This interface is served by a Node.JS server using the Express framework. This setup fully meets the design requirements of the specification.

5.4: Limitations

Our design is non responsive, so our solution would not be viable for use on mobile devices or other small screens. This is a limitation which we could overcome by utilising CSS media queries or by serving a separate site for different devices based on the client's user-agent. This is an acceptable limitation for the page we've implemented as we would expect that our application will only run on a desktop device.

A second limitation is the requirement of JavaScript for our solution to run. Some browsers allow the user to block JavaScript from running – this would remove almost all functionality from our page and render it unusable. We have included a <noscript> tag to notify any users who attempt to run our page while their JavaScript is enabled, and encourage them to enable it for our page.

Our system requires the user to have a social media account on Twitter, and optionally Foursquare, to access the respective functionalities of Tweets or Check-Ins. This is a minor limitation because the pop-up window which appears when the user is asked to login also allows the user to create an account if they haven't already.

However, the popup window in itself could potentially be a limitation as some older browsers may attempt to block the popup. Newer browsers (tested on Chrome, Firefox, and Safari) will block all pop-ups unless they were initiated by a click event, which the Twitter Login window is, so the popup should be allowed.

Section 6: Quality of the Solution

6.1: Issues

In this section we will discuss the quality of the code we implemented.

6.2: Design Choices

We use Ajax in our JavaScript to minimise the server load by only loading new data rather than reloading the entire page on each request. The initial page request requires the entire page to be downloaded, but subsequent requests download smaller amounts of data using JSON rather than reloading the entire page. This reduces bandwidth requirements for the server and client, and creates a smoother user experience.

We used JSON to communicate requests between client and server rather than XML, because it adds readability, compactness, and involves sending a smaller amount of data (lower overhead), resulting in a faster transfer time than the more verbose XML. In addition, it has a simple syntax and is easy to use alongside JavaScript.

Finally, we decided to use the client's browser to validate all input – wherever text entry is required, the input is intercepted before it leaves the client machine and checked for various conditions (e.g. numeric input, format matching etc.) to reduce the number of invalid requests that would have to be handled by the server, hence reducing server load and bandwidth usage. This provides a much faster solution for the user over waiting for data to be sent to the server and back before finding out there was an input error.

6.3: Requirements

Our design meets and exceeds the requirements laid out in the specification: we use JavaScript to check on the client-side that the input is properly validated to reduce failed server requests, and a JSON-based exchange information between client and server. We also implemented Ajax to load only the elements required from the server and avoid refreshing the whole page.

Section 7: Additional Features

7.1: Issues

This section will discuss the additional features we implemented into the system to increase its functionality beyond the scope of the required features in the assignment specification.

7.2: Design Choices

Our application integrates with Google Maps to display Tweet and check-in locations in an intuitive and user-friendly manner. In addition, the Google Maps interface can be used to convert between place names and coordinates to make the input of locations easier and quicker.

We also optionally display live tweet results on the user discussion tab using the Twitter streaming API, allowing the user to conduct a search and view matching tweets immediately as they are posted for any amount of time after the query has been submitted. In addition to the live results, we show the location of tweets on the Google Maps interface.

On the Venues and User Venues tabs, there is the option to choose between Twitter Venues and Foursquare venues during the search to enable the user to choose the source of the location information.

7.3: Requirements

The requirements of additional features were vague and open to creativity depending on the implementation of other aspects of our individual project. The Google Maps integration, live Twitter results, venue selection from Foursquare, and Wikipedia links to nearby points of interest were all beneficial to our project, increasing the scope of the application's abilities and making the user experience faster and easier.

Work division

The team was each allocated individual tasks to complete during the project. The contributions for each individual are state below.

- James McIlveen
  - Design and implementation of the front-end UI
  - Client-server request structure on the front and back-end
  - Querying social web APIs (helped Thomas)
  - Querying web of data (part 2)
  - RDFa and RDFs (part 2)
  - Planning of the above sections
- Thomas Miller
  - Database implementation and backend queries
  - Querying the social web (helped by James McIlveen)
  - Backend database queries
  - Planning of the above sections
- James Foster
  - Documentation (organising meetings with James McIlveen and Thomas to gather implementation information)
  - Assisted James McIlveen with front-end form validation

Each team member has agreed to receive equal the following mark distribution for their contributions to the project: James Foster - 30%, Thomas Miller - 35%, James McIlveen - 35%.

References

Here is a list of links containing content we used to create our application.

NPM modules
https://www.npmjs.com/package/ntwitter
https://www.npmjs.com/package/keyword-extractor
https://www.npmjs.com/package/express
https://www.npmjs.com/package/mysql
https://www.npmjs.com/package/socket.io
https://www.npmjs.com/package/twit

Google maps
https://developers.google.com/maps/

Data sources
https://www.twitter.com/
https://foursquare.com/
http://dbpedia.com/