



# KEEP THAT TWEET FINAL REPORT

**Higher Diploma in Science in**  
Software Development

Word Count: 6246

Eamon Myles

10510445@mydbs.ie

14/08/2020

Supervisor Rory O Donnell

## Abstract

My project was the development of an Android application built with the Java programming language. The project makes use of available twitter APIs to authenticate with a live twitter account and retrieve tweets from live twitter users. Selected tweets are then stored to a database for later viewing.

The application is intended to aid a user in retrieving tweets that they find interesting and that they would like to keep for future reference. How often does something of interest come up in conversation where you find yourself wishing you had remembered the users twitter handle or favourited the tweet to prove your point. My app will allow the user to save tweets with a search term like 'Comedy'. The user now only needs to remember their search term and all the Tweets tagged with keyword 'Comedy' will be returned. The user can then quickly select the required tweet.

## Acknowledgments

I would like to thank Rory O Donnell for his guidance through the project and Ehtisham Yasin who's module on android development formed the basis of a lot of the project.

## Contents

Abstract.....	1
Acknowledgments.....	1
Contents.....	1
1. Introduction .....	3
2. Background/Literature Review .....	4
Context .....	4
Users.....	4
Software Development methods used.....	4
3. Specification and Design .....	5
4. Implementation .....	7
Authentication (Signing a request);.....	8
Percentage encoding .....	9
Generate a viable String .....	10
Joining the returned values.....	10
Generating the Signature .....	11
Creating the Header .....	12
Sending the request .....	13

Successful response.....	14
Storing of tweets .....	15
Stetho .....	21
Flow diagram .....	24
Class diagram.....	24
5. Project testing and results: .....	26
6. Conclusion and Future Work.....	29
7. Bibliography .....	30

## 1. Introduction

The aim of my project is to develop an Android application that takes advantage of Twitter's public APIs to allow the user to authenticate with their Twitter account, search for another user's tweets and store tweets from that user for quick retrieval later.

The idea is that the user can store tweets of interest and save them with a keyword. When they want to view them later, rather than having to remember who posted or having to search through old posts. The user can simply enter their search term and have a list of related tweets returned to them associated with that keyword.

The scope of the project is to

- Authenticate with Twitter's APIs using OAuth 1.0.
- Allow the user to retrieve the top tweets from any known public Twitter handle.
- Tap on a selected tweet to bring the single tweet into view.
- Save that tweet with an additional Keyword or Tag to a Database.
- Allow the user to enter that Keyword or Tag to retrieve a list of all tweets saved with that Keyword or Tag.

I approached this project with some working knowledge of OAuth 1.0 and Twitter's APIs (Twitter, n.d.). Previously having worked with a testing software called RestAssured, which is used for "Testing and validating REST services in Java" (rest-assured, 2020). RestAssured made OAuth 1.0 seem like an easy choice to develop against.

However, as the project progressed I quickly realized that this approach was not good for my Android implementation as I didn't have access to all the OAuth libraries RestAssured provided. This meant that I had to create my own methods for working with OAuth 1.0. If I had researched other auth methods more thoroughly, I would have gone with OAuth 2.0 which is an easier method to work with. Twitter actually recommends not to use OAuth 1.0 if avoidable. "We don't recommend this unless you know what you are doing" (Authentication).

One of the interesting elements that I had to add to my project due to my use of OAuth 1.0, was the addition of Facebook's Stetho Tool. "Stetho is a sophisticated debug bridge for Android applications. When enabled, developers have access to the Chrome Developer Tools feature natively part of the Chrome desktop browser" (Facebook).

OAuth 1.0 requires a very specific set of authorization headers sent before each request can be validated. The headers require keys and various other attributes that must be assembled in a very specific manner to generate a valid auth signature. This signature must also be added and sent with the request. During development I needed a way to verify what I was sending out onto the network was what Twitter required. I had many issues where I could use a third party app like Postman (Postman, n.d.) which would generate a valid response from Twitter. However when I would send my request using my application with seemingly the same headers. The request would fail.

Stetho was the answer to this problem. One of its features is a Network Inspection tool. Once I had this implemented into my application. I was able to view all Network calls made from my application along with headers sent. Once I had this visibility it was very easy to compare what was being sent in the successful requests from Postman and the failing requests in my application. The issue was actually down to just a few spaces and commas but without the Network Visibility that Stetho afforded me, I would not have been able to progress the in-app authentication.

The rest of the features in the app I have studied in my DBS course.

With ROOM databases for android (Yasin, Moodle DBS) and Recycler views (Yasin, Moodle DBS) making up the bulk of the rest of the app. I was happy to progress with this as I was able to develop what I had already learnt in my previous module. Incorporating much larger Json responses from twitter than I had previously worked with in the module. These larger responses do require changes to my original learnings but have been interesting to work with and resolve.

The ROOM database stores just a very small selection of data returned from the Json response. These include

- Strid: Id of the tweet
- Text: Text of the tweet
- Screenname: Twitterhandle of the tweet
- ImageURL: URL to a profile image

I can actually combine a lot of this data in various ways to display the tweets in a WebView. For example. Twitters tweet URL's are made up of the following format <https://twitter.com/{Screenname}/status/{Strid}>

The above information is taken automatically from the Json response. I also have another element that is added manually to the database by the user to each stored tweet.

- Tag: Search term

The tag is a user entered search term. The user will add this to the database along with the above attributes. The user will later use this search term to retrieve their tweets.

## 2. Background/Literature Review

### Context

The application is intended to aid a user in retrieving tweets that they find interesting and that they would like to keep for future reference. How often does something of interest come up in conversation where you find yourself wishing you had remembered the users tweeter handle or favoured the tweet to prove your point. My app will allow the user to save tweets with a search term like 'Comedy. The user now only needs to remember their search term and all the 'Comedy' saved Tweets will be returned.

### Users

The application will be usable to anyone with a twitter account.

### Software Development methods used

I tried to break the project down into isolated Blocks of implementation and used an Agile approach to development. I didn't have any formal Stories created but in my original project plan I had broken the development down into Authentication, tweet retrieval, other users tweet retrieval.

### 3. Specification and Design

Oauth 1.0 – I had previously worked with restassured. “REST Assured is a Java library that provides a domain-specific language (DSL) for writing powerful, maintainable tests for RESTful APIs” (Dijkstra, 2020) . These libraries made working with Oauth very easy and required the user to enter only 4 variables retrieved from twitters developer app page. (Twitter, Twitter developer apps, n.d.). I hadn’t realized that implementing outside of these libraries would be such a challenge due to the ease of use the restassured libraries afforded me. Java had no such libraries available and even the twitter site discouraged the use of Oauth 1.0. “We don't recommended this unless you know what you are doing, or if you're using one of the tools mentioned below to make a request to an endpoint that requires OAuth 1.0a.” (Twitter, Authentication, n.d.)

However. The process of implementing Oauth 1.0, though challenging has been very interesting.

First step was to ensure I could call the required API and get a valid response

“Postman is a collaboration platform for API development. Postman's features simplify each step of building an API and streamline collaboration so you can create better APIs—faster.” (What is Postman?, n.d.)

I used POSTMAN initially to generate these required headers and would make the same network call with the same Oauth Keys in POSTMAN. I would then take the POSTMAN generated headers and manually copy and paste them to my Volley requests in the code to retrieve a Valid Json response from twitter using Volley requests.

“Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster.” (Google).

The more difficult part was to generate my own headers. Twitter provides an example on how to generate these headers and what attributes they need so first step was to implement that. (Twitter, n.d.)

The basic formula is, a Signature key is a required attribute of the header request and that signature key is generated by taking a base URL, timestamp, nonce, signature method, Oauth token, consumer key etc. These elements have to be joined together in a very specific order to create a Base String which must be further modified to percentage encoding standards. “Percent-encoding, also known as URL encoding, is a method to encode information in a Uniform Resource Identifier (URI) under certain circumstances.”, The characters allowed in a URI are either *reserved* or *unreserved* (or a percent character as part of a percent-encoding). *Reserved* characters are those characters that sometimes have special meaning. For example, forward slash characters are used to separate different parts of a URL (or more generally, a URI). *Unreserved* characters have no such meanings. Using percent-encoding, reserved characters are represented using special character sequences.” (Wikipedia, n.d.) .

This Base String is then passed into a base 64encoder with the users consumer Oauth Token and consumer secret key to generate a signature. “Base64 is a *binary-to-text* encoding scheme. It represents binary data in a printable ASCII string format by translating it into a radix-64 representation.

Base64 encoding is commonly used when there is a need to transmit binary data over media that do not correctly handle binary data and is designed to deal with textual data belonging to the 7-bit US-ASCII charset only.” (SINGH, n.d.)

Once the signature has been generated, it is then passed through the Percentage encoding method itself.

I was able to find a base64 and a percentage encoder method from the following website (Alagiya, How to Generate OAuth Signature for twitter in core JAVA, n.d.)

Using these methods I could prove that my percentage and base64 encoding methods worked as they would produce the same signature as the twitter example when using the keys, and strings provided in the twitter example. (Twitter, n.d.)

Next step was to implement the calls I wanted to authenticate with. Namely the “get-statuses-user” API. (Twitter, n.d.) with my own Keys, and BaseURL.

This was extremely difficult as for a long time I had no example to test my base string combinations against and had no way of verifying my generated signatures where valid.

Fortunately I found this example (Mackenzie, n.d.)

This example used exactly the Twitter API I required but using a bash script.

I inputted the tokens and secrets from this example and from there I was able to progress to the point where I was outputting the same signatures as the bash script was producing. Proving that my code was producing valid base strings and signatures.

At this stage I had a working base 64 encoder, a working percentage encoder and a valid Base String combination to add to the Volley request call.

Final problem.

I created a method called AuthHeader() which would take all the variables and create a String value that looked exactly the same as the working POSTMAN headers I was copying and pasting into my code. Each request would return a 401 Bad request response. Outside of the data I was entering I had no visibility of how the headers where displayed in the Network request.

I needed a way to verify my network request was actually being sent in a valid format. At this point I implemented Stetho.

“Stetho is a sophisticated debug bridge for Android applications. When enabled, developers have access to the Chrome Developer Tools feature natively part of the Chrome desktop browser.” (Facebook, Stetho)

It is developed by Facebook and allows app developers access to Database inspection, View hierarchy and most importantly Network Inspection!

It is quite easy to implement but works better with OKHTTP than it does with Volley.

For test purposes I created an OKHTTP client in the app and was able to use Stethos network inspector to view the requests and headers I was sending.

It was immediately clear from this that my requests were sending various commas and spaces that the postman requests must have been stripping out. Once I could see this it was

very easy to update my AuthHeader() method and send exactly what twitter was expecting and finally retrieving what I was expecting. A valid 200 and a JSON response.

The Json response:

One of my DBS modules was Android Development. In this we learnt how to take a small Json response Object and work with it to pass its key value pairs into a recycler view adapter and generate a UX element on screen. I used a similar method with the JSON response from twitter but had to deal with a much larger and more complex JSON array. As I progressed through this project I found myself returning to a few websites more than others. One is <https://stackoverflow.com>. The other is Codinginflow on YouTube. The videos on this channel are clear and well explained and here I was able to find help in dealing with JsonArrays and Volley requests (Flow, RecyclerView + JSON Parsing - Part 4 - PARSING JSON WITH VOLLEY - Android Studio Tutorial) .

This allowed me to progress to displaying information pulled from all levels of the JSON feed into my recycler views.

Saving the Tweets:

This section came together relatively quickly. I use a ROOM Database to store data from selected Tweets and allow the user to Add an additional item which is the Search term or Tag they wish to use to retrieve the tweets later. "Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite." (Google)

As mentioned earlier I save a Screenname, tweet\_Id and tweet\_Text and Image URL from the Json response returned, along with the additional user entered search term. These selected items are actually pretty powerful as I can use them to generate a Tweet URL and bring the user to twitters website to display the tweet directly. Twitters URLs are in the format of <https://twitter.com/SCREENNAME/status/TWEETID> . Both of which I pull from the JSON feed. Its is then quite easy to create the URL for different tweets by inserting the items into variables of the URL

Returning the tweets:

The tweets are returned to the user by making a request to the ROOM Database for ALL the stored data. I then user a searchArray() method which will add only the elements with a specified search tag to an ArrayObject(). Once that has been filled I can send it to my recycler Adapter to display the results to the user. The user can then tap on the returned tweet and will be brought to twitters website as described above.

## 4. Implementation

My application is designed to Authenticate with twitters website using an Oauth 1.0 authentication method.

“OAuth authentication is the process in which Users grant access to their Protected Resources without sharing their credentials with the Consumer. OAuth uses Tokens generated by the Service Provider instead of the User’s credentials in Protected Resources requests. The process uses two Token types:

Request Token:

Used by the Consumer to ask the User to authorize access to the Protected Resources. The User-authorized Request Token is exchanged for an Access Token, MUST only be used once, and MUST NOT be used for any other purpose. It is RECOMMENDED that Request Tokens have a limited lifetime.

Access Token:

Used by the Consumer to access the Protected Resources on behalf of the User. Access Tokens MAY limit access to certain Protected Resources, and MAY have a limited lifetime. Service Providers SHOULD allow Users to revoke Access Tokens. Only the Access Token SHALL be used to access the Protect Resources.

OAuth Authentication is done in three steps:

1. The Consumer obtains an unauthorized Request Token.
2. The User authorizes the Request Token.
3. The Consumer exchanges the Request Token for an Access Token.” (Workgroup)

I created these Keys using twitters App development page. (Twitter). It is necessary to register with twitter to register your application.

The keys are then combined in various ways to generate an auth signature that can be sent via Http request to retrieve data from Twitter in the form of a Json response.

The application will then take those Json responses and convert them into User friendly Items displayed on the application.

The user can then chose to save one of those items with a keyword or tag to a Database. For this application I used the ROOM DB for android (Android)

Finally the user can Retrieve the saved data by entering their keyword or tag and requesting the data back to a UX display on the application.

### Authentication (Signing a request);

In order to gain access to the Keys mentioned above, it is necessary to register your application with twitter. (Twitter, App)

Once you have obtained the 4 required Keys, namely Consumer Key, Consumer Secret, Access Token, Access Secret. It is necessary to understand how the keys are added and formatted with the require API to generate a String header for the request to Twitter.

### *“Signing a request with keys and tokens*

You have to sign each API request by passing several generated keys and tokens in an authorization header. To start, you can generate several keys and tokens in your **Twitter developer app’s** details page, including the following:

API key and secret:  
`oauth_consumer_key`  
`oauth_consumer_secret`

Think of these as the user name and password that represents your Twitter developer app when making API requests.

Access token and secret:  
`oauth_token`  
`oauth_token_secret`

An access token and access token secret are user-specific credentials used to authenticate OAuth 1.0a API requests. They specify the Twitter account the request is made on behalf of.

You can generate your own access token and token secret if you would like your app to make requests on behalf of the same Twitter account associated with your developer account on the **Twitter developer app's** details page. If you'd like to generate access tokens for a different user, see "Making requests on behalf of users" below.

“

(Twitter, Authentication)

“Once you have these keys and tokens, you can either create a signature from scratch. We don't recommended this unless you know what you are doing, or if you're using one of the tools mentioned below to make a request to an endpoint that requires OAuth 1.0a.”  
(Twitter, <https://developer.twitter.com>)

I created the signature myself using the following methods.

Percentage encoding

`percentEncode(url)`

The first element required to generate a signature is a baseUrl. The URL must be percentage encoded. “[Percent encode](#) every key and value that will be signed.” (Twitter, Authentication)

I found a sample code for doing the percentage encoding process at (Alagiya)

```

//used to % encode string values
@RequiresApi(api = Build.VERSION_CODES.O)
public String percentEncode(String value) {
    String encoded = "";
    try {
        encoded = URLEncoder.encode(value, enc: "UTF-8");
    } catch (Exception e) {
        e.printStackTrace();
    }
    String sb = "";
    char focus;
    for (int i = 0; i < encoded.length(); i++) {
        focus = encoded.charAt(i);
        if (focus == '*') {
            sb += "%2A";
        } else if (focus == '+') {
            sb += "%20";
        } else if (focus == '%' && i + 1 < encoded.length()
            && encoded.charAt(i + 1) == '7' && encoded.charAt(i + 2) == 'E') {
            sb += '~';
            i += 2;
        } else {
            sb += focus;
        }
    }
    Log.v( tag: "String Encoded:", sb);

    return sb;
}

```

### Generate a viable String

Next, the encoded URL needs to be concatenated with the HTTP method. In my case it was the GET method. I created a method called concatURL() to pass the encoded URL String with the GET method resulting in a String as such  
 GET&https%3A%2F%2Fapi.twitter.com%2F1.1%2Fstatuses%2Fuser\_timeline.json

```

}
public String concatURL(String url, String method) {

    //String contact = method + "&" + url + "&count%3D5%26";
    String contact = method + "&" + url + "&";
    return contact;
}

```

### Joining the returned values

#### StringConcat()

This method takes all the required values for a successful authentication (Consumer key, oauth nonce, signature method, timestamp etc) and combines them in a specific order into a String with required ampersands and quotes.

On returning the combined string it is percentage encoded with `percentencode()`;

```
@RequiresApi(api = Build.VERSION_CODES.O)
public String StringConcat() {

    String CONSUMER_API_KEY = "oauth_consumer_key=" + GLOBAL_CONSUMER_API_KEY;
    String OAUTH_NONCE = "oauth_nonce=" + GLOBAL_OAUTH_NONCE;
    String HMACSHA1 = "oauth_signature_method=" + GLOBAL_HMACSHA1;
    String OAUTH_TIMESTAMP = "oauth_timestamp=" + GLOBAL_OAUTH_TIMESTAMP;
    String ACCESS_TOKEN = "oauth_token=" + GLOBAL_ACCESS_TOKEN;
    String oauth_version = "oauth_version=" + GLOBAL_oauth_version;
    String screen_name = "screen_name=" + getTwitterhandle();

    String concat = CONSUMER_API_KEY + "&" + OAUTH_NONCE + "&" + HMACSHA1 + "&" + OAUTH_TIMESTAMP + "&" + ACCESS_TOKEN + "&" + oauth_version + "&" + screen_name;

    return concat;
}
```

Once both `concatURL()` and `StringConcat()` had returned their values they both were joined to output the Finished String to a string variable called `fin`.

```
//StringConcat creates the base string and sends it to get percentage encoded also
String fin = t1.concatURL(percentURL1, method: "GET") + t1.percentEncode(t1.StringConcat());
Log.v( tag: "FinishedString:", fin);
```

FinishedString:

```
GET&https%3A%2F%2Fapi.twitter.com%2F1.1%2Fstatuses%2Fuser_timeline.json&oauth_consumer_key%3DgdprVZGkHT7mOfAlr8ZRGMUfP%26oauth_nonce%3D8mX97zgqzrO%26oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%3D1595945997%26oauth_token%3D63139074-vNvsSSIMsMmosQwJ2EwVmHvGzAlgJn8VtGR040GD9%26oauth_version%3D1.0%26screen_name%3DrealDonaldTrump
```

## Generating the Signature

This finished string is then passed to the next method which will accept the string, the users Consumer API Secret, and their token secret.

`generateSignature()`

This code was also found at (Alagiya) and is used to combine the Consumer and token secret, and combine that with the String. Everything is then base64 encoded to produce the final valid signature and assign it to a global String variable 'signature'

```

@RequiresApi(api = Build.VERSION_CODES.O)
public String generateSignature(String signatureBaseStr, String oAuthConsumerSecret, String oAuthTokenSecret) {
    byte[] byteHMAC = null;
    try {
        Mac mac = Mac.getInstance("HmacSHA1");
        SecretKeySpec spec;
        if (null == oAuthTokenSecret) {
            String signingKey = percentEncode(oAuthConsumerSecret) + '&';
            spec = new SecretKeySpec(signingKey.getBytes(), algorithm: "HmacSHA1");
        } else {
            String signingKey = percentEncode(oAuthConsumerSecret) + '&' + percentEncode(oAuthTokenSecret);
            spec = new SecretKeySpec(signingKey.getBytes(), algorithm: "HmacSHA1");
        }
        mac.init(spec);
        byteHMAC = mac.doFinal(signatureBaseStr.getBytes());
    } catch (Exception e) {
        e.printStackTrace();
    }
    return twitter4j.BASE64Encoder.encode(byteHMAC);
}

```

## Creating the Header

### AuthHeader()

The AuthHeader() method then puts everything together to create a single string including the created signature. The output is assigned to a global string header. This will later be sent as an authorization header over https to twitter.

```

public String AuthHeader(String signature) {
    String content = "";

    content += "OAuth oauth_consumer_key=";
    content += "\"";
    content += GLOBAL_CONSUMER_API_KEY;
    content += "\"";

    content += "oauth_nonce=";
    content += "\"";
    content += GLOBAL_OAUTH_NONCE;
    content += "\"";

    content += "oauth_signature=";
    content += "\"";
    content += signature;
    content += "\"";

    content += "oauth_signature_method=";
    content += "\"";
    content += GLOBAL_HMACSHA1;
    content += "\"";

    content += "oauth_timestamp=";
    content += "\"";
    content += GLOBAL_OAUTH_TIMESTAMP;
    content += "\"";
}

```

```

        content += "oauth_token=";
        content += "\"";
        content += GLOBAL_ACCESS_TOKEN;
        content += "\",\"";

        content += "oauth_version=";
        content += "\"";
        content += GLOBAL_oauth_version;
        content += "\"";

        return content;
    }

```

Header produced:

OAuth

```

oauth_consumer_key="gdprVZGkHT7mOfAlr8ZRGMUfP",oauth_nonce="Y6paRo2kMe2",o
auth_signature="8k2sHqx%2Fib0N%2B8axnociALNw1Ns%3D",oauth_signature_method="HM
AC-SHA1",oauth_timestamp="1595952960",oauth_token="63139074-
vNvsSSIMsMmosQwJ2EWWmHvGzAlgJn8VtGR040GD9",oauth_version="1.0"

```

## Sending the request

The code then moves to the postRequest() method

The postRequest() method uses Volley.

Volley will send a request to a specified URL, with the Request method specified.

The addition of headers or other parameters can also be added. In the case of authorization with twitter. Auth headers are required and its here were we take all the information gathered previously and send the created header with the request to twitter

## Sending the request

```

private void postRequest() {
    // https://www.tutorialspoint.com/android/android_loading_spinner.htm
    //Initialize and display spinner
    spinner=(ProgressBar)findViewById(R.id.progressBar1);
    spinner.setVisibility(View.VISIBLE);

    RequestQueue queue = Volley.newRequestQueue( context: MainActivity.this);

    String urltest = "https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name="+twitterhandle+"";
    final StringRequest stringRequest = new StringRequest(Request.Method.GET, urltest,
        new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {

                Log.e( tag: "Http", msg: "success! response: " + response.toString());
                try {

                    post_reponse_text.setText("Twitter Feed");
                    //https://www.youtube.com/watch?v=bRvLq27EWp0

```

## Add header params to the request

```
//https://www.semicolonworld.com/question/45849/how-to-set-custom-header-in-volley-request

@Override
public Map<String, String> getHeaders() throws AuthFailureError {
    Map<String, String> params = new HashMap<>();
    String headers=headers;
    //passing headers https://www.youtube.com/watch?v=c1b2HehVL2M
    params.put("Authorization", headers);

    return params;
}

};

queue.add(stringRequest);
```

### Successful response.

Once a successful response is returned from the user the json file is then added to a JSONArray.

Each object in the JSON Array is then looped through and added to a JSON Object.

Individual items can be taken from these objects and added to a user created Object MyTweets. It is this that will be sent to the android recycler view to generate a graphical representation for the user on the android device.

```

JSONArray jsonArray = new JSONArray(response);

for (int i = 0; i < jsonArray.length(); i++) {
    JSONObject session = jsonArray.getJSONObject(i);

    JSONObject userdetails = session.getJSONObject("user");

    MyTweet tweet = new MyTweet( strid: "test", text: "test", screen_name: "test", profileurl: "test");
    tweet.text = session.getString( name: "text");
    tweet.strid = session.getString( name: "id_str");
    tweet.Screen_Name = userdetails.getString( name: "screen_name");
    tweet.Profileurl=userdetails.getString( name: "profile_image_url");

    tweets.add(tweet);
}

//Send to Adapter
RecyclerView recyclerView = findViewById(R.id.my_recycler_view);
RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getApplicationContext());
recyclerView.setLayoutManager(layoutManager);
Adapter = new MyTweetsAdapter(tweets);
recyclerView.setAdapter(Adapter);

```

## Storing of tweets

Once the user has tweets returned to the screen. He then has the option to select one of these tweets. Once he taps on the tweet, the selected media items are collected and passed onto a new activity for display.

These items are not called from the server. Rather they are pulled from the object already passed to the recycler view.

Items passed include

- Screenname
- Strid (tweet id)
- Text
- Image URL

```

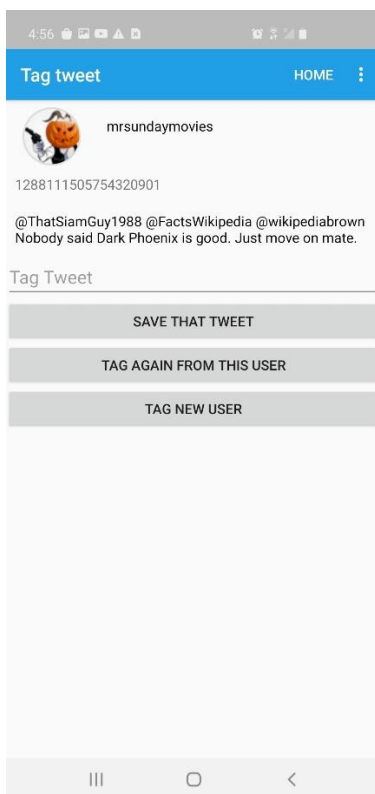
//enable the items to be click
v.setOnClickListener((v) -> {
    if (listener != null) {

        //int strid = Integer.parseInt(topText.getText().toString());
        String screen_name = (topText.getText().toString());
        String strid = (middleText.getText().toString());
        String text = (bottomText.getText().toString());

        int position = getAdapterPosition();
        Intent intent = new Intent(v.getContext(), Tagtweet.class);
        //intent.putExtra(EXTRA_NUMBER, strid);
        intent.putExtra(EXTRA_SCREENNAME, screen_name);
        intent.putExtra(EXTRA_STRID, strid);
        intent.putExtra(EXTRA_TEXT, text);
        intent.putExtra(EXTRA_IMAGEURL, imageUrl);
        v.getContext().startActivity(intent);
        if (position != RecyclerView.NO_POSITION) {
            listener.onItemClick(position);
        }
    }
});

```

These are then displayed to the user as a single View



In this case, as we are only dealing with single items of data. I created an xml layout similar to the existing recycler view but assigned the retrieved values directly to xml placeholders for display.

```
Intent intent = getIntent();
id = intent.getStringExtra(MyTweetsAdapter.EXTRA_STRID);
text = intent.getStringExtra(MyTweetsAdapter.EXTRA_TEXT);
sname = intent.getStringExtra(MyTweetsAdapter.EXTRA_SCREENNAME);
purl = intent.getStringExtra(MyTweetsAdapter.EXTRA_IMAGEURL);

screenname = findViewById(R.id.screenname);
screenname.setText("" + sname);
strid = findViewById(R.id.tweetid);
strid.setText("" + id);
tweettext = findViewById(R.id.tweettext);
tweettext.setText("" + text);
usertag = findViewById(R.id.tagTweet);

//Code to set the profile image from URL
profilePhoto = findViewById(R.id.tagtweetLogo);
StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(policy);
try {
    URL url = new URL(purl);
    Bitmap avatar = BitmapFactory.decodeStream(url.openConnection().getInputStream());
    profilePhoto.setImageBitmap(avatar);
} catch (Exception e) {
    e.printStackTrace();
}
```

It is on this page where we introduce the Database element of the application.

The retrieved information from the previous activity is used to create the UX element of the page. These are then added to the ROOM database with the addition of a user entered tag.

“The [Room](#) persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

The library helps you create a cache of your app's data on a device that's running your app. This cache, which serves as your app's single source of truth, allows users to view a consistent copy of key information within your app, regardless of whether users have an internet connection.” (Google, Room Persistence Library)

“When working with Room, we will work with three major components:

- Database: Represents an abstract database class, which provides one or more data access objects (DAO)
- DAO: It is an interface that lets us define how to get or change values stored within a DB
- Entity: Represents a value object in the DB” (Yasin, elearning.dbs.ie)

I called my database, AppDatabase

```

@Database(entities = {UserTweet.class}, version = 17, exportSchema = false)

public abstract class AppDatabase extends RoomDatabase {

    private static AppDatabase INSTANCE;

    public abstract UserTweetDAO userTweetDAO();

    public static synchronized AppDatabase getDatabase(Context context) {
        if (INSTANCE == null) {
            INSTANCE =
                Room.databaseBuilder(context, AppDatabase.class, name: "UserTweetDB")
                    .allowMainThreadQueries()
                    .fallbackToDestructiveMigration()
                    .build();
        }
        return INSTANCE;
    }

    public static void destroyInstance() { INSTANCE = null; }
}

```

Its DAO was called UserTweetDAO and it had the ability to get everything from the database using getAllUserTweets(), add an object to the database using add() and to remove all database information using removeAllUserTweets().

The application current makes use of only the first two of these methods.

```

@Dao
public interface UserTweetDAO {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void add(UserTweet usertweet);

    @Query("SELECT * FROM UserTweet")

    public List<UserTweet> getAllUserTweets();
    // @Query("SELECT * FROM UserTweet WHERE Tag = :Tag")
    @Query("SELECT * FROM UserTweet WHERE StrId = :StrId")
    public UserTweet getUser(long StrId);

    @Update(onConflict = OnConflictStrategy.REPLACE)
    void updateUserTweets(UserTweet usertweet);

    @Query("DELETE FROM usertweet")
    void removeAllUserTweets();

}

```

And finally My entity was UserTweet.

This would accept five string elements including

- StrId (or tweet id)
- Text (text from tweet)
- Screen\_Name
- Profileurl
- Tag (This was the only element that would be entered by the user)

```
@Entity
public class UserTweet {

    @PrimaryKey
    @NonNull
    public final String StrId;
    public String Text;
    public String Screen_Name;
    public String Profileurl;
    public final String Tag;

    public UserTweet(String StrId, String Text, String Tag, String Screen_Name, String Profileurl) {

        this.StrId = StrId;
        this.Text = Text;
        this.Screen_Name = Screen_Name;
        this.Tag = Tag;
        this.Profileurl = Profileurl;
    }
}
```

The next core element of the application is the retrieval of the data from the database.

I was not familiar with creating ROOM type SQL scripts to access the data outside of getAllUserTweets() as described in the DAO section of the database description.

My application only needs to return tweets from the database that match a given search criteria or tag, therefore returning All tweets is not helpful in this case.

I had issues injecting a variable into the DAO query to create an SQL query returning only items matching the users search tag. My final solution was to return ALL the information in the Database and store it in a list object. I then looped through the Object and populated a second object with information relating only to the tag. This meant I was able to create an object of the users desired search terms.

I would imagine this wouldn't be a very efficient method if the application was to scale up but for the amount of information I am working with it is more than sufficient and still very responsive.

```

if(tagged.equals("")){
    //if(tagged==""){
    Toast.makeText(getApplicationContext(), text: "Please enter a Tag", Toast.LENGTH_SHORT).show();
}
else {
    database = AppDatabase.getDatabase(getApplicationContext());
    List<UserTweet> tweetdb = database.userTweetDAO().getAllUserTweets();
    if (Utils.searchArray(tweetdb, tagged).size()==0) {
        Toast.makeText(getApplicationContext(), text: "No tweets tagged with "+tagged+"", Toast.LENGTH_LONG).show();
    }
    else
        adapterMethod(Utils.searchArray(tweetdb, tagged));
}
}

```

```

public static List<Object> searchArray(List<UserTweet> tweetdb, String userEnteredTag) {
    //pull strings from DB and convert them into List of hashmaps that can be consumed by the adapter
    //List<Movie> moviesdb = database.userDao().getAllMovies();
    List<Object> tweetDB = new ArrayList<>();

    HashMap mMap = new HashMap();

    for (int i = 0; i < tweetdb.size(); i++) {
        UserTweet tweet = (tweetdb.get(i));
        String Tag = tweet.Tag;
        if(Tag.equals(userEnteredTag)) {
            mMap.put("screen_name", tweet.Screen_Name);
            mMap.put("str_id", tweet.StrId);
            mMap.put("text", tweet.Text);
            mMap.put("Tag", tweet.Tag);
            mMap.put("Profileurl",tweet.Profileurl);

            tweetDB.add(mMap);
            mMap = new HashMap();
        }
    }
    return tweetDB;
}

```

The second list object returned is then passed into a recycler adapter and displayed to the user.

```

public void adapterMethod(List<Object> list) {
    RecyclerView recyclerView = (RecyclerView) findViewById(R.id.my_recycler_view2);
    RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getApplicationContext());
    recyclerView.setLayoutManager(layoutManager);
    returnTweetsAdapter Adapter = new returnTweetsAdapter(list);
    recyclerView.setAdapter(Adapter);
    Adapter.setOnItemClickListener(new returnTweetsAdapter.OnItemClickListener() {
        @Override
        public void onItemClick(int position) {

        }
    });
}
}

```

## Stetho

The final element I would like to discuss as a core element of the project was the introduction of Stetho into the project. “Stetho is a sophisticated debug bridge for Android applications. When enabled, developers have access to the Chrome Developer Tools feature natively part of the Chrome desktop browser.” (Facebook, Stetho)

As described earlier I needed this tool to help me understand what format my https requests were being sent out over the internet.

Stetho works best when using OKHTTP for networking. As stated earlier I was using Volley to send my requests but I created an activity to send the same requests out over OKHTTP in order to prove my concepts.

I later added a “Debug” element to the application to demonstrate this element of the application and also for easier use when I needed to access stetho.

The implementation of the feature was explained in the youtube videos “Debugging Android with Stetho” (Felker, n.d.). “intercept and debug http requests with Stetho” (Melardev, n.d.) and I also referenced a website Debugging Android Apps with Facebooks Stetho (Hathibelagal, n.d.)

Once the OkHttpClient and Stetho have been added to the project through Gradle, the OkHttpClient must be initialized and the StethoInterceptor added to the calls.

```

//https://www.youtube.com/watch?v=UYINRy6YPqo
OkHttpClient client = new OkHttpClient();
//Add Stetho interceptor
client.networkInterceptors().add(new StethoInterceptor());

```

```

com.squareup.okhttp.Request request = new com.squareup.okhttp.Request.Builder()
    .url("https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=eamonmyles&count=5")
    .method( method: "GET", body: null)
    .addHeader( name: "Authorization", headers)
    //.addHeader("Authorization", "OAuth oauth_consumer_key=\"gdprVZGkHT7m0fALr8ZRGMuFP\",oauth_token=\"6313
    .build();
client.newCall(request).enqueue(new Callback() {

```

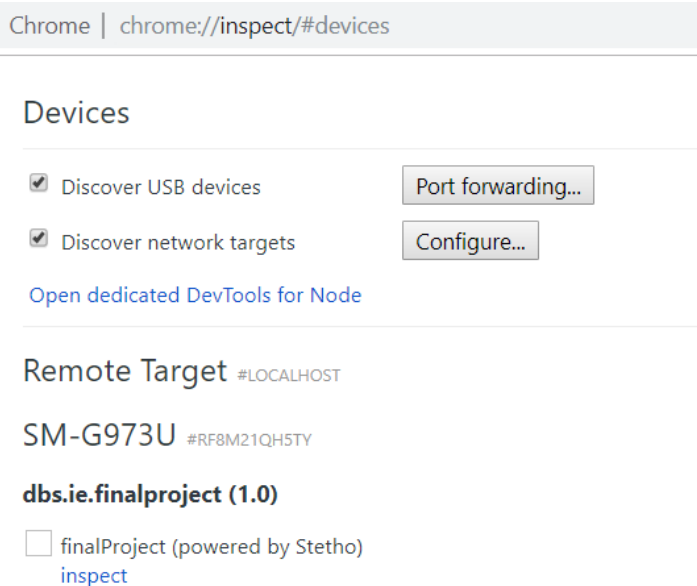
Once this has been done Stetho can intercept the network calls when they are sent over the internet and using the Chrome browser a dashboard can be used to view the calls and responses.

Simply connect the device to the computer,

Launch the application,

Open a Chrome browser and go to `chrome://inspect`

Your application will then be available for inspection.



Once 'inspect' is selected, a dev tools is launched. At this point you can select 'Network'. Return to your application and make the appropriate network call. This call is then intercepted and can be view and analysed in the browser.

Request URL: https://api.twitter.com/1.1/statuses/user\_timeline.json?screen\_name=eamonmyles  
 Request Method: GET  
 Status Code: 200

**Request Headers**

- Provisional headers are shown
- Accept-Encoding: gzip
- Authorization: OAuth oauth\_consumer\_key="gdrVZGkHT7mOfALr8ZRGmUFP", oauth\_nonce="S04vbErnMcg", oauth\_signature="EMlqXBVfaxChxdSuvMFPpFUSeogX3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1596124595", oauth\_token="63139074-vllvsSS1MsMmosQwJ2ENVmhVzGzA1gJn8VtGR0406D9", oauth\_version="1.0"
- Connection: Keep-Alive
- Host: api.twitter.com
- User-Agent: okhttp/2.7.5

**Query String Parameters**

- screen\_name: eamonmyles

**Response Headers**

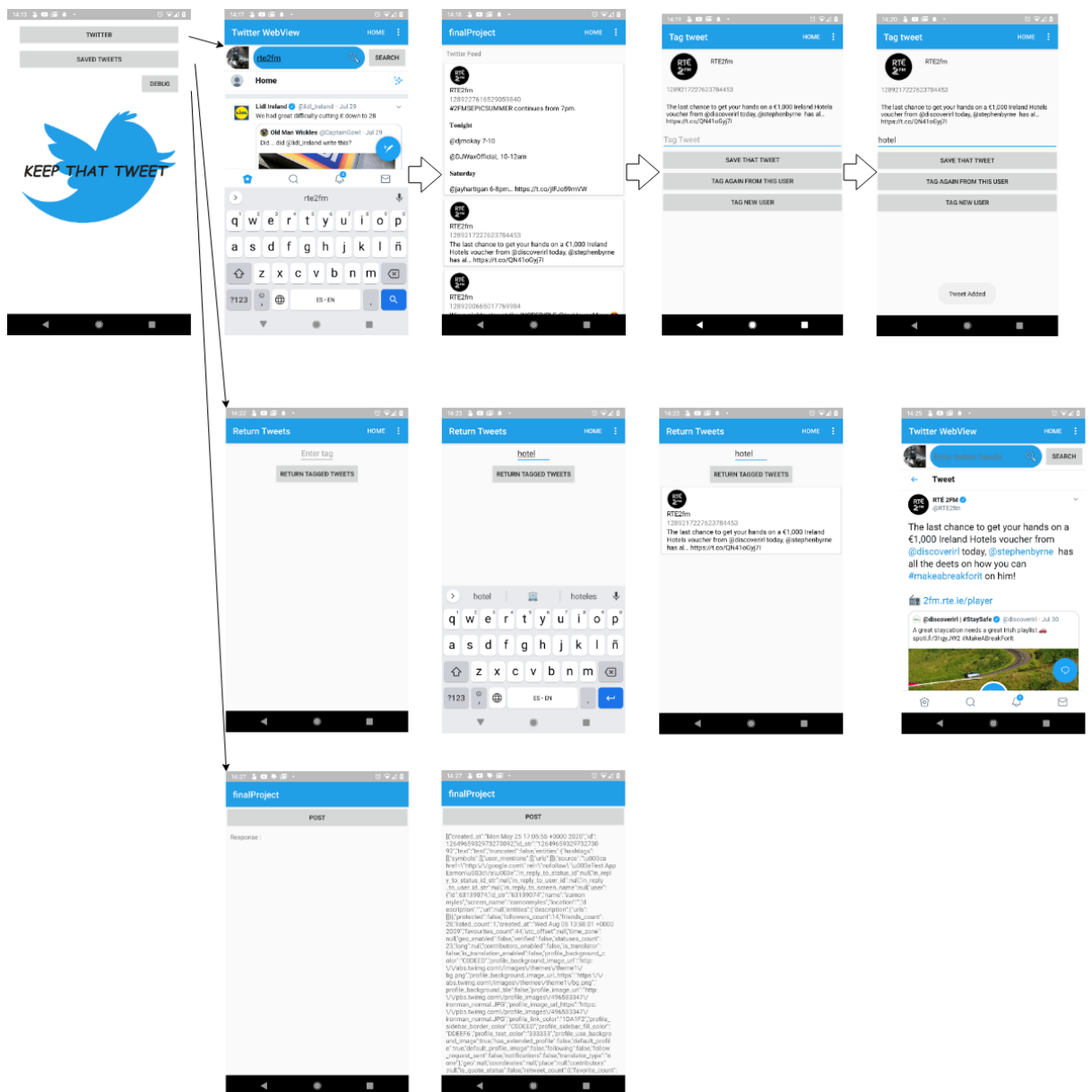
- cache-control: no-cache, no-store, must-revalidate, pre-check=0, post-check=0
- content-disposition: attachment; filename=json.json
- content-encoding: gzip
- content-length: 1863
- content-type: application/json;charset=utf-8
- date: Thu, 30 Jul 2020 15:56:34 GMT
- expires: Tue, 31 Mar 1981 05:00:00 GMT
- last-modified: Thu, 30 Jul 2020 15:56:34 GMT
- OkHttp-Received-Millis: 1596124596866
- OkHttp-Sent-Millis: 1596124596220
- pragma: no-cache

Stetho also has an available option from view local resources of the application.

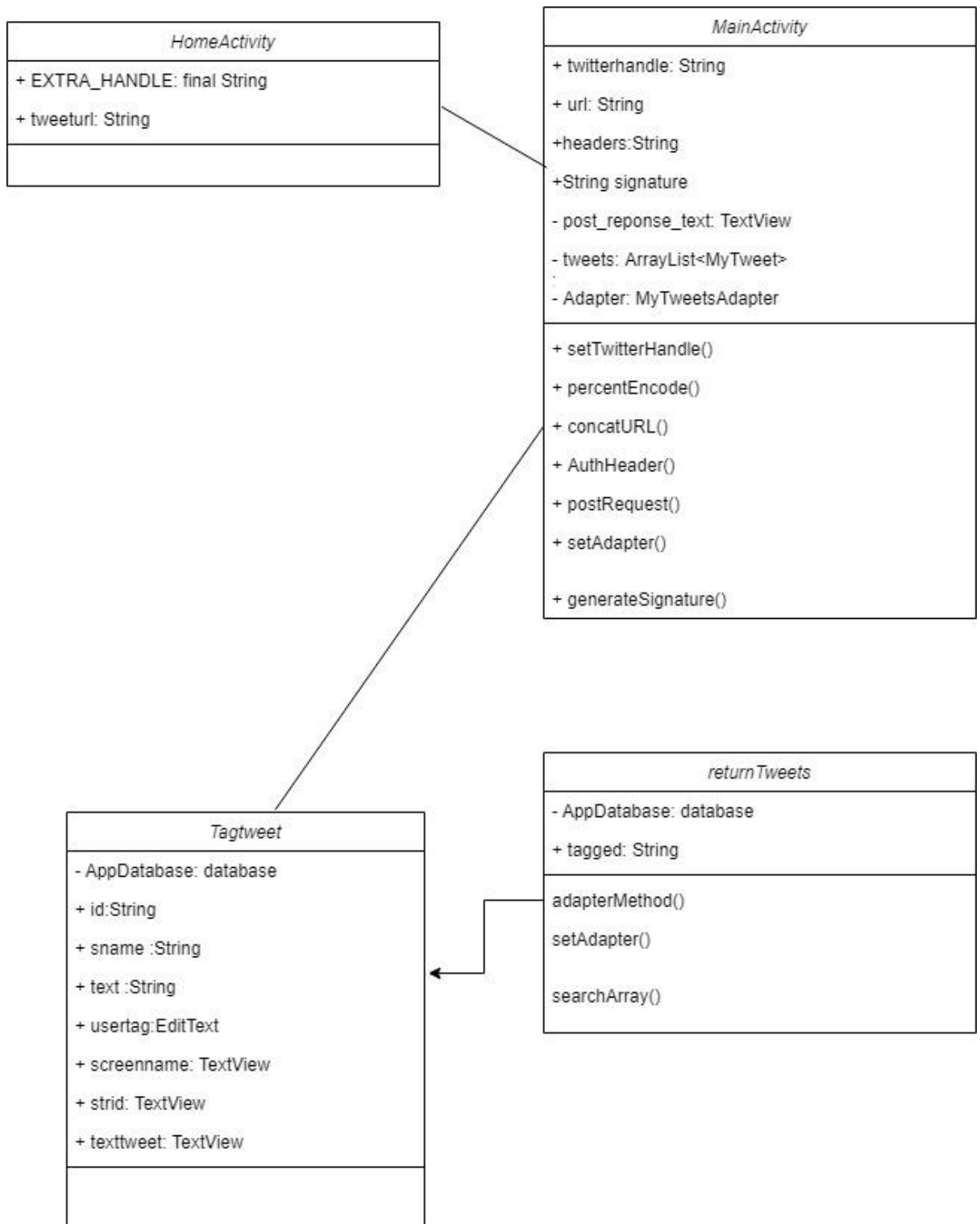
One of these resources is the application database, again this is a very helpful tool for debugging available data in the database.

rowid	Strid	Text	Screen_Name	Profileurl	Tag
5	12751...	@ctrl_alt_rees @BenRattigan77 Did you know the newer ARM ch...	jamesfmackenzie	http://pbs.twimg.com/profile_images/820687137695731713/qUDBSOK2_normal.jpg	test4
6	12822...	No, Radical Left anarchists, agitators, looters or protesters will no...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	test5
7	12822...	...rounds, no problem. When I play, Fake News CNN, and others. ...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	test6
8	12822...	RT @WhiteHouse: President @realDonaldTrump is a champion f...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	test8
9	12819...	I interview Evan Collins about his conversion from Buddhism to C...	mattfradd	http://pbs.twimg.com/profile_images/1278840507892785154/QHn1bgdf_normal.jpg	test1
10	12815...	RT @AustinSarabia: This is a fantastic interview with @mattfradd...	mattfradd	http://pbs.twimg.com/profile_images/1278840507892785154/QHn1bgdf_normal.jpg	test1
11	12808...	If you're into philosophy and the Catholic Faith, go follow my ma...	mattfradd	http://pbs.twimg.com/profile_images/1278840507892785154/QHn1bgdf_normal.jpg	test1
12	12822...	RT @greta: Watch ( DVR) Full Court Press tomorrow/ Sunday - @...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	test1
13	12822...	RT @WhiteHouse: President @realDonaldTrump took action this ...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	test1
14	12817...	@JezCorden MGS2! Social media echo chambers, fake news, dee...	jamesfmackenzie	http://pbs.twimg.com/profile_images/820687137695731713/qUDBSOK2_normal.jpg	test1
15	12751...	@ctrl_alt_rees @MarkFGrogan Great memories. I had an Atari ST ...	jamesfmackenzie	http://pbs.twimg.com/profile_images/820687137695731713/qUDBSOK2_normal.jpg	test1
16	12854...	RT @AndrewKirell: Between falling for a fake tweet that was deb...	justinbaragona	http://pbs.twimg.com/profile_images/1280589187645915138/nL_4ewHj_normal.jpg	apology
18	12854...	LIKE THIS CANT B REAL	bobseul	http://pbs.twimg.com/profile_images/1281439675241070593/1kNdOLGz_normal.jpg	apology
19	12854...	last time gowon had black hair was two years.... she's been blond...	bobseul	http://pbs.twimg.com/profile_images/1281439675241070593/1kNdOLGz_normal.jpg	apology
20	12853...	Big business news for the Lowcountry: @Walmart will build a ma...	jslovegrove	http://pbs.twimg.com/profile_images/1229491115813953538/FLZH-iYn_normal.jpg	apology
21	12853...	RT @FreeTimesSC: Lawrence Nathaniel is part of a push to start a...	jslovegrove	http://pbs.twimg.com/profile_images/1229491115813953538/FLZH-iYn_normal.jpg	apology
22	76523...	miley cyrus	DyFu	http://abs.twimg.com/sticky/default_profile_images/default_profile_normal.png	miley
23	12881...	....be able to produce what I have. So when you see those nasty a...	realDonaldTrump	http://pbs.twimg.com/profile_images/874276197357596672/kUuht00m_normal.jpg	trump

## Flow diagram



## Class diagram



## 5. Project testing and results:

As mentioned in Section 2 I have tried to incorporate an Agile methodology. This method requires break down of implementations and testing before moving onto the next section.

I think this has worked relatively well as I have completed all the tasks I had originally estimated. Before completing a section of work it's important to verify and test the implementation before progressing to the next. I have created a selection of testcases and ran through these at each completed stage. Increasing the cases as a new feature was added. I have verified my implementation once complete with the entire suite of tests.

Splash screen	Test name	Test description	Result	Validation
	Twitter button	Verify that the available button brings the user to the appropriate corresponding page.	User is brought to Homepage displaying Search bar and Twitter Webview	Pass
	Saved Tweets button	Verify that the available button brings the user to the appropriate corresponding page.	User is brought to the Return Tweets page.	Pass
	Debug button	Verify that the available button brings the user to the appropriate corresponding page.	User is brought to the Debug page.	Pass
Home page	Test name	Test description	Result	Validation
	Webview	Verify that the homepage loads in Twitters live webpage in the available Webview	Twitter webpage loads in	Pass
	Image load	Verify that the profile icon is loaded in on the top left of the search bar	Profile image loads in	Pass
	Valid Twitter handle Search	Verify user can enter a valid twitter handle and selects search	Tweets from specified user are returned	Pass

	Invalid Twitter handle Search	Verify invalid or none existing Twitter handle and selects search	Error returned	Pass
<b>Main page</b>	<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Validation</b>
	Tweets returned	Verify that the returned tweets display with , Screenname, users profile icon, tweet id, tweet text	All elements available	Pass
	Tap on tweet	Tapping on a single tweet will bring the user to a new page	The new page displays only the selected tweet with available options to add a tag, button to save the tag, tag again from this user or tag a new user	Pass
<b>Tag Tweet</b>	<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Validation</b>
	Add tag	User has an available text entry page where they can enter a tag to the tweet	User can tap on a text field to enter the text	Pass
	Save that Tweet	User has entered a tag term and selects Save that tweet	Tapping on save that tweet when text box is populated displays a toast 'Tweet added'. Tweet is saved to Database	Pass
	Save that Tweet – Empty value	User has not entered a tag term and selects Save that tweet	A toast is displayed with 'Please enter a tag'	Pass
	Tag again from this user	User can select tag again from this user to return the same users tweets again	The current users tweets are loading again for additional selection	Pass
	Tag new user	Selecting tag a new user will return the user to the Homepage	Tapping on Tag a new user returns user to home page	Pass
<b>Return Tweets</b>	<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Validation</b>

	Enter a valid tag	User can enter a valid tag to return matching tweets	A list of matching tweets are returned to the user display, Profile image, Screenname, Tweet id Tweet text	Pass
	Enter null tag	If a user does not enter a tag and selects Return tagged tweets	A toast is displayed asking the user to please enter a tag	Pass
	Enter invalid tag	If a user enters a tag that does not exist in the database and selects Return tagged tweets	A toast is displayed with "No tweets tagged with {search term}"	Pass
<b>Menu</b>	<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Validation</b>
	Menu not available on splash screen	The splash screen should not be displayed on the splash screen	Splashscreen does not contain a home menu Tweet text	Pass
	Menu not available on debug page	The Debug menu is for demo purpose only and does not require a menu. Select Debug button on the splash screen	Debug page does not contain a home menu Tweet text	Pass
	Twitter webview. Available menu	The Twitter Webview contains a HOME button and a overflow menu to Return tweets. Selecting Home brings user to the splash screen selecting Return Tweets brings the user to the Return Tweets page	Menus available and progress to appropriate pages	Pass
	Main page available menu	The mainpage contains a HOME button and an overflow menu to Return Tweets. Selecting Home brings user to the splash screen selecting Return	Menus available and progress to appropriate pages	Pass

		Tweets brings the user to the Return Tweets page		
	Tag tweet page available menu	The Tag tweet page contains a HOME button and an overflow menu to Return Tweets. Selecting Home brings user to the splash screen selecting Return Tweets brings the user to the Return Tweets page	Menus available and progress to appropriate pages	Pass
	Return tweets page available menu	The Tag tweet page contains a HOME button and an overflow menu to Return Tweets. Selecting Home brings user to the splash screen selecting Return Tweets displays a Toast	Menus available. Home button brings the user to the appropriate page. Selecting Return Tweets on the Return Tweets page displays a Toast to the user informing them they are already on the appropriate page	Pass
<b>Menu</b>	<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Validation</b>
	Debug button	Debug button on the splashscreen	User is brought to the debug page but pressing debug	Pass
	Post button	The post button returns raw json text to the screen	After selecting Post raw json text is displayed on screen	Pass

## 6. Conclusion and Future Work

I would consider my final submission to cover almost all of what I had originally envisioned for the project and overall, I am quite satisfied with the project. I felt I have learnt a lot about the authentication process within apps and have achieved a deep understanding of a lot of the features within the app.

I do think the application has more elements that could be added to it to make it a more user friendly applications and I hope to continue development of the application going forward.

These elements for further development include,

- Addition of multiple tagging with comma separation
- Returning tweets tagged with closest matches. e.g. Covid, returns items tagged with key words 'Covid 19', 'Covid virus', 'covid19'
- Displaying saved Keywords
- Deleting returned tweets
- Better error response reaction, e.g. There is a current issue where, If I search for a twitter handle that is private. I get a generic error message. I would like to improve that to display case appropriate responses to the user.
- Retrieve and display more tweet information, e.g. media(Photos and videos) added to the tweet.
- At the moment some image elements of the application are hardcoded links. Specifically the profile image on the homepage. I would like to pull this from the json feed. Currently, the image will fail if changed as the URL is not auto updated.
- Addition of further twitter APIs, example, post a tweet to twitter from the app.
- More graphically pleasing UI experience.
- I would like to implement some unit tests.
- Get the application submitted to Github

## 7. Bibliography

Alagiya, N. (n.d.). *How to Generate OAuth Signature for twitter in core JAVA*. Retrieved from optimumbrew: <http://optimumbrew.com/blog/2015/03/oauth/how-to-generate-oauth-signature-for-twitter-in-core-java>

Android. (n.d.). *Android developer guide*. Retrieved 2020, from <https://developer.android.com/guide>

Android. (n.d.). *Room Persistence Library*. Retrieved from Developers: <https://developer.android.com/topic/libraries/architecture/room>

*Authentication*. (n.d.). Retrieved from Twitter.com: <https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a>

Dijkstra, B. (2020, June 28). <https://techbeacon.com/app-dev-testing/how-perform-api-testing-rest-assured>. Retrieved from TechBeacon.

Facebook. (n.d.). *Stetho*. Retrieved from <http://facebook.github.io/stetho/>

Facebook. (n.d.). *Stetho*. Retrieved from <http://facebook.github.io>: <http://facebook.github.io/stetho/>

Felker, D. (n.d.). *Debugging Android with Stetho*. Retrieved from Youtube: <https://www.youtube.com/watch?v=iyXpdkqBsG8>

Flow, C. i. (n.d.). *Coding in Flow*. Retrieved May 2020, from [https://www.youtube.com/channel/UC\\_Fh8kvtkVPkeihBs42jGcA](https://www.youtube.com/channel/UC_Fh8kvtkVPkeihBs42jGcA)

Flow, C. i. (n.d.). *RecyclerView + JSON Parsing - Part 4 - PARSING JSON WITH VOLLEY - Android Studio Tutorial*. Retrieved from <https://www.youtube.com/watch?v=bRvLg27EWp0>

Github. (n.d.). *Github*. Retrieved May 2020, from <https://github.com/>

Google. (n.d.). *Room Persistence Library*. Retrieved from [/developer.android.com: https://developer.android.com/topic/libraries/architecture/room?gclid=Cj0KCQjwvIT5BRCqARIsAAwwD-TN59T2djxKhE0uotojg-zFxb-3lqYvyM5ka\\_buBT-oE-nm6H1geJsaAmCYEALw\\_wcB&gclsrc=aw.ds](https://developer.android.com/topic/libraries/architecture/room?gclid=Cj0KCQjwvIT5BRCqARIsAAwwD-TN59T2djxKhE0uotojg-zFxb-3lqYvyM5ka_buBT-oE-nm6H1geJsaAmCYEALw_wcB&gclsrc=aw.ds)

Google. (n.d.). *Save data in a local database using Room*. Retrieved from [developer.android.com: https://developer.android.com/training/data-storage/room](https://developer.android.com/training/data-storage/room)

Google. (n.d.). *Volley overview*. Retrieved from [developer.android.com: https://developer.android.com/training/volley](https://developer.android.com/training/volley)

Hathibelagal, A. (n.d.). *Debugging Android Apps with Facebook's Stetho*. Retrieved from [code.tutsplus.com: https://code.tutsplus.com/tutorials/debugging-android-apps-with-facebooks-stetho--cms-24205](https://code.tutsplus.com/tutorials/debugging-android-apps-with-facebooks-stetho--cms-24205)

Mackenzie, J. (n.d.). *UNDERSTANDING THE TWITTER API*. Retrieved from [https://www.jamesfmackenzie.com: https://www.jamesfmackenzie.com/2015/02/25/understanding-the-twitter-api/](https://www.jamesfmackenzie.com/https://www.jamesfmackenzie.com/2015/02/25/understanding-the-twitter-api/)

Melardev. (n.d.). *Intercept and debug http requests with Stetho*. Retrieved from [youtube: https://www.youtube.com/watch?v=UYINRy6YPqo](https://www.youtube.com/watch?v=UYINRy6YPqo)

Postman. (n.d.). *Postman*. Retrieved 2020, from <https://www.postman.com/>

*rest-assured*. (2020, 06). Retrieved from [rest-assured.io: http://rest-assured.io/](http://rest-assured.io/)

SINGH, R. (n.d.). *What is Base64 Encoding and How does it work?* Retrieved from [https://www.base64encoder.io: https://www.base64encoder.io/learn/](https://www.base64encoder.io/https://www.base64encoder.io/learn/)

*Stack Overflow*. (n.d.). Retrieved 2020, from <http://stackoverflow.com/>

Twitter. (n.d.). Retrieved from <https://developer.twitter.com/en/apps>

Twitter. (n.d.). *App*. Retrieved from <https://developer.twitter.com/en/apps>.

Twitter. (n.d.). *Authentication*. Retrieved from [https://developer.twitter.com: https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a](https://developer.twitter.com/https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a)

Twitter. (n.d.). *Authentication*. Retrieved from [Collecting parameters: https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a/creating-a-signature](https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a/creating-a-signature)

Twitter. (n.d.). *Authentication*. Retrieved from [developer.twitter.com: https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a](https://developer.twitter.com/https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a)

- Twitter. (n.d.). *Creating a signature*. Retrieved from <https://developer.twitter.com:https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a/creating-a-signature>
- Twitter. (n.d.). *Get Tweet timelines*. Retrieved 5 2020, from <https://developer.twitter.com/en/docs/tweets/timelines/overview>
- Twitter. (n.d.). *Get Tweet timelines*. Retrieved from [https://developer.twitter.com:https://developer.twitter.com/en/docs/tweets/timelines/api-reference/get-statuses-user\\_timeline](https://developer.twitter.com:https://developer.twitter.com/en/docs/tweets/timelines/api-reference/get-statuses-user_timeline)
- Twitter. (n.d.). *https://developer.twitter.com*. Retrieved from Authentication: <https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a>
- Twitter. (n.d.). *Twitter developer apps*. Retrieved from [developer.twitter.com:https://developer.twitter.com/en/docs/basics/apps/overview](https://developer.twitter.com:https://developer.twitter.com/en/docs/basics/apps/overview)
- Twitter. (n.d.). *Twitter Docs*. Retrieved May 2020, from <https://developer.twitter.com/en/docs>
- What is Postman?* (n.d.). Retrieved from [www.postman.com:https://www.postman.com/](http://www.postman.com:https://www.postman.com/)
- Wikipedia. (n.d.). *Percent-encoding*. Retrieved from <https://en.wikipedia.org:https://en.wikipedia.org/wiki/Percent-encoding>
- Workgroup, O. C. (n.d.). *Authenticating with OAuth*. Retrieved from <https://oauth.net/core/1.0/#anchor9:https://oauth.net/core/1.0/#anchor9>
- Yasin, E. (n.d.). Retrieved from [elearning.dbs.ie:https://elearning.dbs.ie/pluginfile.php/1148005/mod\\_resource/content/0/Tutorial%20April%202nd.pdf](https://elearning.dbs.ie:https://elearning.dbs.ie/pluginfile.php/1148005/mod_resource/content/0/Tutorial%20April%202nd.pdf)
- Yasin, E. (n.d.). *Moodle DBS*. Retrieved from [https://elearning.dbs.ie/course/view.php?id=10778:https://elearning.dbs.ie/pluginfile.php/1151561/mod\\_resource/content/0/Tutorial%207th%20April.pdf](https://elearning.dbs.ie/course/view.php?id=10778:https://elearning.dbs.ie/pluginfile.php/1151561/mod_resource/content/0/Tutorial%207th%20April.pdf)
- Yasin, E. (n.d.). *Moodle DBS*. Retrieved from [https://elearning.dbs.ie/mod/resource/view.php?id=864710:https://elearning.dbs.ie/pluginfile.php/1148005/mod\\_resource/content/0/Tutorial%20April%202nd.pdf](https://elearning.dbs.ie/mod/resource/view.php?id=864710:https://elearning.dbs.ie/pluginfile.php/1148005/mod_resource/content/0/Tutorial%20April%202nd.pdf)