About This Document

This document is a technical article containing information on how to use Facebook, Dropbox and Twitter APIs in Android apps. To understand the contents of this article, one should know the basics of Java and Android app development. Knowledge of the aforementioned APIs is not required. This article will guide you through the process of creating your first app, making use of Facebook, Dropbox and Twitter APIs.

1. Introduction

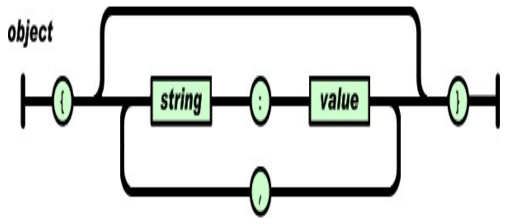
1.1 What do these APIs offer?

The subsequent sections of this article present how to use Facebook, Dropbox and Twitter services directly from your Android app. Such usage is a great way to enrich your app with a little effort. The integration with these services can be added to your app at every stage of the development, since it requires only a limited amount of changes in the source code and does not affect the app's architecture. There are native apps which enable users to interact with the Facebook, Dropbox and Twitter and can be invoked with the specific Intents from your app, but such a solution is not very convenient.

The utilization of the Facebook, Dropbox and Twitter APIs gives you the access to the great deal of useful functionalities. First of all, it is possible to add user authentication based on accounts from those services. Additionally, you could enable the user to interact with other people who have the same app using Facebook and Twitter direct messages. Moreover, it is possible to download various information from these services e.g. you could get the user's Facebook friends list or subscribe to Twitter status updates. Last but not least, you can store the app's files (e.g. information about game saves or any other information relevant to your app) on Dropbox and enable the user to access them from any device with Android. The detailed description on treating about adding mentioned functionalities to your Android app is provided further in this article.

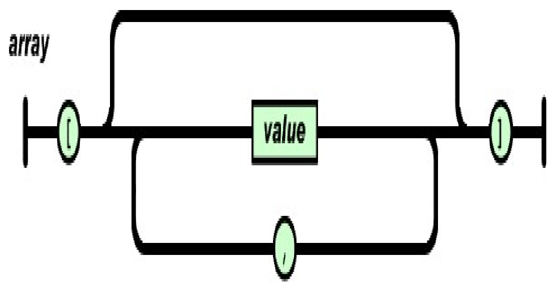
1.2 JSON

The APIs described in this article make use of JSON while communicating with servers. In fact, you don't have to deal with JSON directly, since the libraries used to communicate with Facebook, Dropbox and Twitter handle this by themselves. Nevertheless, it is worth saying a few words about JSON to outline how the API calls are structured. Simply put, JSON is a very popular text format designed specifically for data interchange. It contains two structures for holding data: object and array. Object is an unordered collection of name/value pairs. Its structure is self-explanatory and is presented in the image below.



[Image 1] Structure of JSON's object [Source: http://json.org]

The second element of JSON is the array, which is an ordered collection of values. The image presented below confirms its structure.



[Image 2] Structure of JSON's array [Source: http://json.org]

Having presented the two building blocks of all JSON formatted files, there is only one thing left to explain i.e. what value can be. It can be one of the following: string, number, object, array, true, false or null. As you may have noticed, objects and arrays can be nested in one another creating more complex structures.

2. Using the Facebook API

2.1 Getting started

The first API to be presented is the Facebook API. It can be accessed via Facebook SDK 3.0 for Android which is available for free on the Facebook's developer site under the following link: https://developers.facebook.com/resources/facebook-android-sdk-3.0.zip. The SDK comes packed with a jar library, a native Facebook app, the documentation and plenty of samples presenting common use cases. The Facebook API enables you to add almost all the functionalities of the native Facebook app to your own Android app. The most important ones are:

- •Logging in to a Facebook account
- •Publishing information from the app directly to Facebook (e.g. posting status updates or uploading photos)
- •Downloading information from Facebook
- •Communicating with other Facebook users via direct messages

It is worth mentioning that it is recommended for users of your application to have the native Facebook app installed (the SDK is partially dependent on the native app). That way you can benefit from the simplified authentication flow and ready UI elements like friend or place pickers. The following subsections provide a full guide on how to create an Android app which makes use of functionalities provided by the Facebook SDK.

2.2 Creating an app on Facebook

Before creating an Android app with the Facebook SDK, you have to create a Facebook app on your Facebook developer account (https://developers.facebook.com). The whole process is described in great detail on the aforementioned site. Each application created on Facebook has its unique, automatically generated App ID, which is attached to every request sent from the SDK to identify your app on the server, so it is important to store it somewhere in your Android app. This document will present this in the next subsection. The other thing you should be aware of is that you have to declare how your Android app integrates with the Facebook app. This can be performed in the developer's console by entering the Android app's package name, the class name of main activity and the key hash used by ADT to sign the apk file. If you're going to integrate Facebook with your existing app this is the time to enter the necessary information. You shouldn't be worried if your app isn't yet created. Information about the native Android app which integrates with Facebook can be updated at any time.

2.3 Creating an Android app using Facebook SDK

After creating the app in the Facebook developer console, you are ready to start developing your Android app integrated with Facebook. At this point you should create a new project or load an existing one and import the Facebook SDK into it. If you are using Eclipse, the easiest way to attach the SDK is to add it as an Android library by going to the project's properties.
The first thing that should be done is a little configuration in the app's manifest file. Below is a code snippet relevant to the Facebook sample.

```xml
<!-- Facebook -->
       <activity
            android:name="com.sprc.sample.FacebookSampleActivity"
            android:label="@string/facebook_sample" />

       <activity
            android:name="com.facebook.LoginActivity"
            android:theme="@android:style/Theme.Translucent.NoTitleBar" />

       <meta-data
            android:name="com.facebook.sdk.ApplicationId"
            android:value="@string/app_id"/>
```

[Code 1] Part of AndroidManifest.xml related to using Facebook SDK

Apart from declaring a FacebookSampleActivity which presents the usage of different API functions, there are two important things: the declaration of LoginActivity and ApplicationId meta-data. LoginActivity is a special activity provided by the Facebook SDK and is used for logging into a Facebook account. Its declaration is needed if you want to benefit from the simplified authentication flow. The Meta-data tag maps the App ID which is stored as the app_id string resource to a special element com.facebook.sdk.ApplicationId which is automatically recognized by the Facebook SDK. If you do not provide such a mapping, the SDK will not be able to know the ID of your app, and therefore it will not be possible to use Facebook's API. Having the manifest file configured for usage of the Facebook SDK, you can proceed to the next step, which is handling the authentication process.

At this point it is worth to say a little bit about how the sample activity looks and works. Its layout is defined in the activity_facebook.xml file whose content is presented below.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:facebook="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="20dp"
        android:orientation="vertical"
        android:gravity="center_horizontal" >

        <com.facebook.widget.LoginButton
```

```xml
                  android:id="@+id/fb_login_button"
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:layout_marginTop="5dp"
                  facebook:confirm_logout="false"
                  facebook:fetch_user_info="true" />

         <TextView
                  android:id="@+id/user_name"
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:layout_gravity="center"
                  android:textSize="18sp"
                  android:layout_margin="10dp"/>

         <Button
                  android:id="@+id/post_image"
                  android:layout_width="match_parent"
                  android:layout_height="wrap_content"
                  android:text="@string/post_image"
                  android:onClick="postImageClicked" />

         <Button
                  android:id="@+id/update_status"
                  android:layout_width="match_parent"
                  android:layout_height="wrap_content"
                  android:text="@string/update_status"
                  android:onClick="updateStatusClicked" />

         <Button
                  android:id="@+id/pick_place"
                  android:layout_width="match_parent"
                  android:layout_height="wrap_content"
                  android:text="@string/pick_place"
                  android:onClick="pickPlaceClicked" />

         <FrameLayout
                  android:id="@+id/fragment_container"
                  android:layout_width="fill_parent"
                  android:layout_height="fill_parent"/>

</LinearLayout>
```

[Code 2] Part of AndroidManifest.xml related to using Facebook SDK

As you can see the layout is composed of the following elements: LoginButton, TextView, three normal Buttons and a FrameLayout. The LoginButton is a UI component provided by the Facebook SDK. Interacting with it may have two results depending on the users login status. If the user is not logged in, clicking this button starts the authentication process. If the user is logged in, clicking the button results in logging out of the Facebook account. The other element which should be mentioned is the FrameLayout. It plays the role of the container for fragments. Its usage is presented in the last section of this part of the article. The rest is rather self-explanatory. Buttons fire different API calls and the TextView presents the user name after a successful authentication.

As authenticating with Facebook is the first step in every app which makes use of the Facebook API, it is time to say a few words on how it is implemented in this sample. There are two main classes provided by the Facebook SDK that deal with the authentication process: Session and UiLifecycleHelper. Session is used to authenticate a user and manage the user's session with Facebook. It may be in a few different states from which the most important one is opened. When a session is opened it means that it is active and can be successfully used to send Facebook API requests. You can listen for session state changes by creating an instance of the StatusCallback interface and passing it to the UiLifecycleHelper object. As its name suggests, UiLifecycleHelper is a helper class for managing sessions. If you decide to use it, you have to call all 6 methods it provides. All these methods are related to the following Activity's lifecycle methods: onCreate(Bundle b), onResume(), onPause(), onDestroy(), onActivityResult(int requestCode, int resultCode, Intent data), onSaveInstanceState(Bundle b). We therefore need to override these 6 methods and invoke the corresponding helper methods in each of them. Part of the FacebookSampleActivity which is related to handling authentication with Facebook is presented below. All the imports, fields and methods irrelevant at this point were removed for code clarity.

```java
public class FacebookSampleActivity extends FragmentActivity {

    private LoginButton facebookLoginButton;
    private UiLifecycleHelper helper;
    private static final List<String> PERMISSIONS = Arrays.asList("publish_actions");

    private Session.StatusCallback callback = new Session.StatusCallback() {
        @Override
        public void call(Session session, SessionState state, Exception exception) {
            if(state.isOpened()) {
                Log.d("FacebookSampleActivity", "Facebook session opened");
            } else if(state.isClosed()) {
                Log.d("FacebookSampleActivity", "Facebook session closed");
            }
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
```

```
        helper = new UiLifecycleHelper(this, callback);
        helper.onCreate(savedInstanceState);
        setContentView(R.layout.activity_facebook);

        userName = (TextView) findViewById(R.id.user_name);
        facebookLoginButton = (LoginButton) findViewById(R.id.fb_login_button);
        facebookLoginButton.setUserInfoChangedCallback(new UserInfoChangedCallback() {
             @Override
             public void onUserInfoFetched(GraphUser user) {
                  if(user!=null) {
                       userName.setText("Logged as: " + user.getName());
                  }
                  else {
                       userName.setText("User not logged");
                  }
             }
        });
    }

    public void requestPermissions() {
         Session s = Session.getActiveSession();
         if (s!=null)
              s.requestNewPublishPermissions(new Session.NewPermissionsRequest(this,
PERMISSIONS));
    }

    @Override
    public void onResume() {
         super.onResume();
         helper.onResume();
    }

    @Override
    public void onPause() {
         super.onPause();
         helper.onPause();
    }
}
```

[Code 3] Excerpt from FacebookSampleActivity.java

As you may see using UiLifecycleHelper simplifies managing the authentication flow a great deal, since you almost don't have to directly interact with the Session object. The helper instance is created in the onCreate method by passing two parameters: the current activity and StatusCallback which is stored in a callback field. The provided callback does nothing more than print log messages indicating the session's state. You are of course free to implement more complicated logic in the callback's call method. The presented snippet contains only the onPause() and onResume() methods, but the source code attached to this article implements all the necessary methods. In the onCreate method you will find another callback - serInfoChangedCallback - in use. This is added to the LoginButton instance with the setUserInfoChangedCallback method. This callback provides only the onUserInfoFetched method which is invoked with the GraphUser parameter when the authentication process succeeds. In case of this sample application, the object of GraphUser type is used to access the user's name and set it in the TextView field. The last thing from the code snippet which needs to be described is the requestPermissions() method. The Facebook API has many different permission levels which define which actions we are able to perform. If we would like to post information to Facebook on the user's behalf, we need to ask for appropriate permissions. In this case we need to use requestNewPublishPermissions provided by the Session object. Our sample app asks for the publish_actions permission but there are many more and their description can be found on the Facebook developer site. FacebookSampleActivity has also a checkPermissions() method which simply checks if the user allowed us to perform certain actions. Having taken care of the proper authentication flow, we are ready to start using functionalities provided by the Facebook API. The subsequent sections show how to use the Facebook SDK from within our Android app.

2.4 Posting status updates

The first functionality to be presented is posting status updates. The whole code needed to implement it is contained in one function - updateStatusClicked - which is invoked when the user clicks one of the buttons that is part of the FacebookSampleActivity. This function's code is presented below.

```
public void updateStatusClicked(View view) {
        if(checkPermissions()) {
            Request request = Request.newStatusUpdateRequest(Session.getActiveSession(),
sampleMessage, new Request.Callback() {
                @Override
                public void onCompleted(Response response) {
                    if(response.getError()==null)
                        Toast.makeText(FacebookSampleActivity.this, "Status updated
successfully", Toast.LENGTH_LONG).show();
                }
            });
            request.executeAsync();
        }
        else {
            requestPermissions();
        }
    }
```

[Code 4] Method updateStatusClicked() from FacebookSampleActivity.java

The first thing this function does is check if the user granted our app the necessary permissions. If not, it invokes the requestPermissions() method which, as a result, shows a dialog asking for the necessary permission. It then creates a Request object which is executed by invoking the executeAsync() method. The Request class provides many different methods which return a specific Request to be sent as part of the Facebook API call. In this case the newStatusUpdateRequest(…) method is called. It has 3 parameters: an active session object, the message to be sent as status update and a callback which will be invoked on action completion. The callback has an onCompleted(Response response) method, which can be used to check if our request succeeded. As you may see, the use of the Facebook API through the provided SDK is a fairly easy and convenient method.

2.5 Uploading images

The second thing to be discussed is uploading images to Facebook. Due to the fact that it is very similar to posting status updates we will start with a code snippet.

```
public void postImageClicked(View view) {
      if(checkPermissions()) {
           Bitmap img = BitmapFactory.decodeResource(getResources(),
R.drawable.ic_launcher);
           Request uploadRequest = Request.newUploadPhotoRequest(Session.getActiveSession(),
img, new Request.Callback() {
                @Override
                public void onCompleted(Response response) {
                    Toast.makeText(FacebookSampleActivity.this, "Photo uploaded
successfully", Toast.LENGTH_LONG).show();
                }
           });
           uploadRequest.executeAsync();
      } else {
           requestPermissions();
      }
  }
```

[Code 5] Method postImageClicked() from FacebookSampleActivity.java

As you may have noticed it looks almost the same as previously. There are two differences: first, the Bitmap object was created from a drawable resource. It represents the image to be uploaded. You could start the camera at this point and capture a photo to upload to Facebook. The second difference is the method used to create the Request object. In this case the newUploadPhotoRequest method is invoked. Its second parameter is the image to be uploaded. Similarly to the previous code snippet, Request is invoked within the executeAsync() method. When the response comes from the Facebook server it is passed to the callback function we registered.

2.6 Picking places with PlacePickerFragment

The last thing to be described is the PlacePickerFragment. It is one of a few ready UI components provided by the Facebook SDK. The other ones are e.g. FriendPickerFragment, ProfilePictureView and UserSettingsFragment. The code snippet from FacebookSampleActivity which shows how to make use of this fragment is presented below. It is followed by the description of the implementation's details.

```
public class FacebookSampleActivity extends FragmentActivity {
```

```
private static final Location SAMPLE_LOCATION = new Location("") {
    {
        setLatitude(25.2697);
        setLongitude(55.3095);
    }
};

public void pickPlaceClicked(View view) {
    final PlacePickerFragment frag = new PlacePickerFragment();
    frag.setLocation(SAMPLE_LOCATION);
    frag.setTitleText("Pick nearby place");

    frag.setOnDoneButtonClickedListener(new OnDoneButtonClickedListener() {
        @Override
        public void onDoneButtonClicked(PickerFragment<?> fragment) {
            placePicked(frag);
        }});

    frag.setOnSelectionChangedListener(new OnSelectionChangedListener() {
        @Override
        public void onSelectionChanged(PickerFragment<?> fragment) {
            if(frag.getSelection() != null) {
                placePicked(frag);
            }
        }});

    showPlacePickerFragment(frag);
}

private void placePicked(PlacePickerFragment fragment) {
    FragmentManager fm = getSupportFragmentManager();
    fm.popBackStack();

    String msg = "";

    GraphPlace selectedPlace = ((PlacePickerFragment) fragment).getSelection();
    if (selectedPlace != null) {
        msg = "Selected: " + selectedPlace.getName();
    }
    else {
        msg = "No place was selected";
    }
```

```
        Toast.makeText(FacebookSampleActivity.this, msg, Toast.LENGTH_LONG).show();


    }
 }
    private void showPlacePickerFragment(PickerFragment<?> fragment) {
        fragment.setOnErrorListener(new PickerFragment.OnErrorListener() {
            @Override
            public void onError(PickerFragment<?> pickerFragment, FacebookException error)
{
                Toast.makeText(FacebookSampleActivity.this, "Error",
Toast.LENGTH_LONG).show();
            }
        });

        FragmentManager fm = getSupportFragmentManager();
        fm.beginTransaction()
                .replace(R.id.fragment_container, fragment)
                .addToBackStack(null)
                .commit();

        fm.executePendingTransactions();
        fragment.loadData(false);
    }


}
```

[Code 6] Snippet from FacebookSampleActivity.java showing how to use PlacePickerFragment

The sample app has the following 3 methods that deal with handling PlacePickerFragment: pickPlaceClicked(View view), placePicked(PlacePickerFragment fragment) and showPlacePickerFragment(PickerFragment fragment). The pickPlaceClicked method is invoked each time the user clicks on a button. Its responsibility is to create an instance of PlacePickerFragment. It sets the geographic location which is used to show nearby places. In case of this app, the location is defined by the SAMPLE_LOCATION variable and points to Seoul. Additionally it defines two listeners: OnDoneButtonClickedListener() and OnSelectionChangedListener(). They are connected to the picker fragment's behavior and all they do is forward the selected item from the picker's list to the placePicked method. At the end of the placePicked method there is a call to the showPlacePickerFragment method, to which the newly created fragment is passed. The showPlacePickerFragment creates a new fragment transaction and adds a picker fragment to the FrameLayout from the activity's layout file. The last function to be described is placePicked. What this does is get the selected place from the picker fragment and present its name using a Toast.

3. Using the Dropbox API

3.1 Getting started

The Dropbox API, similarly to the Facebook one, can be accessed from Android using the official SDK. The SDK for Dropbox is available from the following link: https://www.dropbox.com/static/developers/dropbox-android-sdk-1.5.3.zip. It contains a JAR library (also dependent libraries) with source code, the documentation and some sample code. Making use of this API is a good way to enrich your app with cloud-based file storage. The Dropbox SDK gives you a way to share files with a Dropbox account.

3.2 Creating app on Dropbox

Before we delve into how to use the Dropbox SDK in your Android app, there is one thing left. Similarly to Facebook, you have to create a Dropbox app on the Dropbox developer page (https://www.dropbox.com/developers/apps). In case of Dropbox you don't have to enter any information about your Android app. You need to provide an app name and the app's folder name. It's worth remembering that each Dropbox app has access only to its own folder. During the creation of a new Dropbox app two strings are generated: App key and App secret which we will be using while communicating with the Dropbox server from the Android app.

3.3 Adding Dropbox SDK to Android app

Now it is time to create a new Android app which integrates with the Dropbox API. The first thing which should be done is copying all the jars that come with the SDK to the libs folder of your project. Having done that, a little bit of configuration in the app's manifest file is needed. The part which is relevant to the Dropbox sample is presented below.

```xml
<activity
        android:name="com.dropbox.client2.android.AuthActivity"
        android:launchMode="singleTask"
        android:configChanges="orientation|keyboard">
        <intent-filter>
                <!-- Change this to be db- followed by your app key -->
                <data android:scheme="db-YOUR-APP-KEY" />
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.BROWSABLE"/>
                <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</activity>
```

[Code 7] Part of AndroidManifest.xml related to using Dropbox SDK

As you can see, AuthActivity needs to be declared in the manifest file. This activity comes from the Dropbox SDK and manages the authentication process. It is important not to forget to declare an intent-filter which is used to capture the intent invoked by the SDK. One of the filter's elements has to be the data tag which contains the android:schema attribute with the following value db-APP_KEY ,where instead of APP_KEY the key generated in the developer's console is placed. Our app is now ready to perform the Dropbox authentication process. A code snippet from DropboxSampleActivity which deals with authentication is visible below. Any irrelevant details like imports have been omitted.

```java
public class DropboxSampleActivity extends Activity {
    DropboxAPI<AndroidAuthSession> dropbox;

    private final static String FILE_DIR = "/MyFiles/";
    private final static String DROPBOX_NAME = "dropbox_prefs";
    //Replace APP_KEY with key from Dropbox dev account
    private final static String ACCESS_KEY = "APP_KEY";
    //Replace APP_SECRET with secret from Dropbox dev account
    private final static String ACCESS_SECRET = "APP_SECRET";
    private boolean isLoggedIn;
    private Button logIn;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dropbox);

        logIn = (Button) findViewById(R.id.dropbox_login);
        loggedIn(false);

        AndroidAuthSession session;
        AppKeyPair pair = new AppKeyPair(ACCESS_KEY, ACCESS_SECRET);

        SharedPreferences prefs = getSharedPreferences(DROPBOX_NAME, 0);
        String key = prefs.getString(ACCESS_KEY, null);
        String secret = prefs.getString(ACCESS_SECRET, null);

        if (key != null && secret != null) {
            AccessTokenPair token = new AccessTokenPair(key, secret);
            session = new AndroidAuthSession(pair, AccessType.APP_FOLDER, token);
        } else {
            session = new AndroidAuthSession(pair, AccessType.APP_FOLDER);
        }

        dropbox = new DropboxAPI<AndroidAuthSession>(session);
    }

    @Override
    protected void onResume() {
        super.onResume();

        AndroidAuthSession session = dropbox.getSession();
        if(session.authenticationSuccessful()) {
            try {
                session.finishAuthentication();

                TokenPair tokens = session.getAccessTokenPair();
                SharedPreferences prefs = getSharedPreferences(DROPBOX_NAME, 0);
                Editor editor = prefs.edit();
                editor.putString(ACCESS_KEY, tokens.key);
                editor.putString(ACCESS_SECRET, tokens.secret);
                editor.commit();
                loggedIn(true);
            } catch (IllegalStateException e) {
                Toast.makeText(this, "Error during Dropbox authentication",
Toast.LENGTH_SHORT).show();
            }
        }
    }

    public void loggedIn(boolean isLogged) {
        isLoggedIn = isLogged;
        logIn.setText(isLogged ? "Log out" : "Log in");
    }

    public void onLogInClicked(View v) {
        if (isLoggedIn) {
            dropbox.getSession().unlink();
            loggedIn(false);
        } else {
            dropbox.getSession().startAuthentication(DropboxSampleActivity.this);
        }
    }
    public void onUploadClicked(View v) {
        UploadFileToDropbox upload = new UploadFileToDropbox(this, dropbox, FILE_DIR);
        upload.execute();
    }

    public void onListFilesClicked(View v) {
        ListFiles list = new ListFiles(dropbox, FILE_DIR);
        list.execute();
    }
}
```

[Code 8] Excerpt from DropboxSampleActivity.java

In comparison to the Facebook sample, the process of the user’s authentication with the Dropbox API is a little more complicated. You have to directly deal with access tokens and the session. The first thing we need to concentrate on is the onCreate method which performs the necessary configuration. AppKeyPair which is a container for App key and App secret that are stored respectively as ACCESS_KEY and ACCESS_SECRET is created at the beginning. We then try to obtain the token keys from app preferences. These keys are used together with app keys while sending requests to Dropbox’s server. If tokens were stored in preferences we create AccessTokenPair which will be used to create a session. You can store access tokens in preferences and then read them in onCreate to simplify the authentication process. The next thing is creating an AndroidAuthSession object using the appropriate constructor. We usually pass two parameters: AppKeyPair and AccessType objects. If we know the access tokens, we pass AccessTokenPair as the third parameter. The last thing that you have to take care of in onCreate is creating a DropboxAPI instance by passing the session object as the parameter and storing it in the dropbox attribute.

The second method needing clarification is onLogInClicked. It is invoked every time the user clicks the log in button. Its function depends on the isLoggedIn field which indicates whether the user is logged in. If that field is true we call the unlink() method from AndroidAuthSession. That method logs out the user from Dropbox using the API. In case the user is already logged out we invoke startAuthentication(Activity activity) on the session object. As the method's name suggests it begins the authentication process by opening AuthActivity. If authentication is handled by AuthActivity the app flow comes back to our activity and we need to perform some finishing actions. First we get the session object from the DropboxAPI instance with the getSession() method. It is then necessary to check with the help of authenticationSuccessful() if the authentication process has succeeded. If the returned value is positive we have to implicitly end the authentication process calling finishAuthentication() on the session object. At this point the session object contains a valid access token pair which is used to authenticate our app on the Dropbox's server. In case of the presented sample app the access token pair is stored in app preferences. Here ends the authentication process and we are ready to perform calls to the Dropbox API. The next section shows how to upload files to a Dropbox account and download the file listing of a specified directory. The onUploadClicked(View v) and onListFilesClicked(View v) methods are used to handle button clicks and execute AsyncTasks which perform described actions.

3.4 Sample usage of the Dropbox API from an Android app

The sample app prepared for the purpose of this article presents two useful functionalities provided by the Dropbox SDK. The first of them is uploading an image directly from the Android app to the specific folder of the Dropbox app. The AsyncTask which performs this action is presented below.

```java
public class UploadFileToDropbox extends AsyncTask<Void, Void, Boolean> {

    private DropboxAPI<?> dropbox;
    private String path;
    private Context context;
    private static int fileNo=0;

    public UploadFileToDropbox(Context context, DropboxAPI<?> dropbox, String path) {
         this.context = context.getApplicationContext();
         this.dropbox=dropbox;
         this.path=path;
    }

    @Override
    protected Boolean doInBackground(Void... params) {
         File tempDir = context.getCacheDir();
         File tempFile;
```

```
        FileWriter fr;
        try {
            fileNo++;
            tempFile = File.createTempFile("file", ".txt", tempDir);
            fr = new FileWriter(tempFile);
            fr.write("Sample file content "+fileNo);
            fr.close();

            FileInputStream fis = new FileInputStream(tempFile);
            dropbox.putFile(path+"file"+fileNo+".txt", fis, tempFile.length(), null,
null);
            tempFile.delete();
            return true;
        } catch (IOException e) {
            e.printStackTrace();
        }
         catch (DropboxException e) {
             e.printStackTrace();
        }

        return false;
    }

    @Override
    protected void onPostExecute(Boolean result) {
        if(result) {
            Toast.makeText(context, "File successfully uploaded",
Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(context, "File not uploaded", Toast.LENGTH_LONG).show();
        }
    }
}
```

[Code 9] Content of UploadFileToDropbox task

As you can see the UploadFileToDropbox's constructor has three parameters: Context context, DropboxAPI dropbox and String path. Context is the activity this task is created in. Dropbox is the instance of DropboxAPI initialized with the active session. It is used to perform all Dropbox API calls. The last parameter – path – is the path of the folder on Dropbox where the file is going to be uploaded. The real job is performed in the doInBackground method. At the beginning there are I/O operations which result in the creation of a temporary file with sample content. Having done that, we are ready to finally use one of the methods provided by the DropboxAPI class - putFile(String fileName, InputStream is, long size, String parentRev, ProgressListener progress). In this sample app we only use the first three of them, passing null as 4th and 5th parameter since we are not really interested in monitoring the progress of the file’s update. If you would like to present the user some graphical indication of the upload progress you are free to create a new ProgressListener and perform the necessary action inside its callback method. When the file has been successfully uploaded, a Toast with appropriate info is presented. Additionally, the temporary file which was uploaded to Dropbox is deleted from the device’s storage. If a problem occurs while communicating with the Dropbox server, a DropboxException which you should handle by yourself is thrown.

The second functionality presented by the Dropbox sample app is listing all the files from the specified directory. It is implemented in a separate AsyncTask – ListFiles. Its content is presented below.

```
public class ListFiles extends AsyncTask<Void, Void, List<String>>{

    private DropboxAPI<?> dropbox;
    private String path;

    public ListFiles(DropboxAPI<?> dropbox, String path) {
        this.dropbox=dropbox;
        this.path=path;
    }

    @Override
    protected List<String> doInBackground(Void... params) {
        List<String> files = new ArrayList<String>();
        try {
            Entry directory = dropbox.metadata(path, 1000, null, true, null);
            for(Entry entry : directory.contents) {
                files.add(entry.fileName());
            }
        } catch (DropboxException e) {
            e.printStackTrace();
        }

        return files;
    }

   @Override
   protected void onPostExecute(List<String> result) {
       for(String fileName : result) {
           Log.i("ListFiles", fileName);
       }
   }

}
```

[Code 10] Content of the ListFiles task

Since the constructor of the ListFiles task is very similar to the constructor of the previously described task (the only difference is that there is no Context needed) we can go directly to the doInBackground method. The goal of this function is to return a list of all files which are placed in a specified Dropbox folder. The first thing performed inside this function is calling the metadata(…) method on a dropbox object. It has five parameters: file/directory path, limit of entries to be returned, hash of previous metadata, boolean param indicating if we want to get info about the entry's children and a revision identifier. Our invocation of this method uses only the 1st, 2nd and 4th parameters. The Entry object is returned as a result of this method. It contains all the necessary info about the specified directory. One such information is the public contents attribute which contains a list of all child entries. Knowing that attribute we can easily iterate through that list and get each entry's name using the fileName() method. The rest of the presented code is pretty straightforward. Each file name is stored in a list which is returned from the doInBackground method. This list is then passed to the onPostExecute method. This method simply iterates through all the returned files and prints their names in a separate log entry on the console. You can check the names of all the files in the specified folder by looking at the LogCat's output.

4. Using the Twitter API

4.1 Getting started

The last API to be presented in this article is Twitter's API. Unfortunately Twitter does not provide an official SDK for Android. In this situation we have to use the Twitter4J library which is an unofficial Java library for the Twitter API. It is available under the following link: http://twitter4j.org/en/index.html. The biggest drawback of Twitter4J from our point of view is that it is not Android specific and does not contain any ready UI elements, which complicates the authentication process.

4.2 Creating an app on Twitter

At the beginning of our journey of using Twitter's API from within our Android app we need to create Twitter's app. The whole process looks very similar to the previously described Facebook and Dropbox processes. You need to provide the app's name, description, website and callback URL. The consumer key and consumer secret and access token key and access token secret are generated. The first pair of keys is used to identify your app while communicating with the server while the second pair is used to define what permissions your app has. After creating Twitter's app it is time to move to the integration of Twitter4J with your Android app.

4.3 Creating an Android app with Twitter4J

You are now ready to integrate Twitter into a new or existing Android app (for demo apps check the Samsung Developers site). You should copy the Twitter4J library JARs to the libs folder of your project. At the time of writing this article the library comes with 4 JARs: twitter4j-async, twitter4j-core, twitter4j-media and twitter4j-stream. The sample project attached to the article makes use of basic Twitter's functionalities, therefore only the twitter4j-core jar can be found in the libs folder.

4.4 Using Twitter4J to communicate with Twitter

When the library has been added and your project is configured to use Twitter4J, you are ready to create your first app which uses Twitter's API. At the beginning you have to add the intent-filter tag to the activity which will be responsible for communicating with Twitter. The code snippet provided below shows how you should do this.

```
<!--  Twitter -->
       <activity
            android:name="com.sprc.sample.TwitterSampleActivity"
            android:label="@string/twitter_sample" >
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:scheme="oauth" android:host="sample" />
            </intent-filter>
       </activity>
```

[Code 11] Part of AndroidManifest.xml related to using Twitter4J

The most important part of this code snippet is the data tag contained in the intent-filter. The attributes defined in this tag should be the same as the callback URL used for requesting the OAuth token. In case of the provided Twitter sample app the callback URL is oauth://sample.

The next step is performing the authentication process. A snippet from TwitterSampleActivity highlighting the code relevant to Twitter's authentication process is presented below. It is followed by a description of core concepts.

```
public class TwitterSampleActivity extends Activity  {

    private static final String CONSUMER_KEY = "3nlBEpJhGMVeQCDWL9EQ";
    private static final String CONSUMER_SECRET =
"NbtW9PUmweaqwIc2cuG1muTaDifEsHgMoiCmC0ZiREo";
    private static final String CALLBACK = "oauth://sample";
    private String OAuthToken;
    private String OAuthSecret;
    private boolean isLogged;
    private static Twitter twitter;
    private static RequestToken requestToken;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_twitter);

        Uri uri = getIntent().getData();
        if(uri!=null && uri.toString().startsWith(CALLBACK)) {
            String verifier = uri.getQueryParameter("oauth_verifier");

            try {
                AccessToken token = twitter.getOAuthAccessToken(requestToken, verifier);
                OAuthToken = token.getToken();
                OAuthSecret = token.getTokenSecret();
                isLogged = true;

                User user = twitter.showUser(token.getUserId());
                String username = user.getName();

                userName.setText("Logged as: " + username);
            } catch (Exception e) {
                Log.e("Login error: ", e.getMessage());
            }
        }
    }

    public void onLogInClicked(View v) {
        if(!isLogged) {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.setOAuthConsumerKey(CONSUMER_KEY);
            builder.setOAuthConsumerSecret(CONSUMER_SECRET);
            Configuration config = builder.build();
            TwitterFactory factory = new TwitterFactory(config);
            twitter = factory.getInstance();

            try {
                requestToken = twitter.getOAuthRequestToken(CALLBACK);
                this.startActivity(new Intent(Intent.ACTION_VIEW,
Uri.parse(requestToken.getAuthenticationURL())));
            } catch (TwitterException e) {
                e.printStackTrace();
            }
        } else {
            Toast.makeText(this, "Already logged", Toast.LENGTH_LONG).show();
        }
    }

     public void onSendClicked(View v) {
        String text = status.getText().toString();
        if(text.length()>0) {
           new SendTwitterStatusUpdate(this, CONSUMER_KEY, CONSUMER_SECRET, OAuthToken,
OAuthSecret).execute(text);
        } else {
            Toast.makeText(this, "Enter some text" , Toast.LENGTH_SHORT).show();
        }
    }


}
```

[Code 12] Excerpt from TwitterSampleActivity.java

The first thing which should be explained is the onLogInClicked method, which is invoked when the user clicks the log in button. Twitter's authentication process is initialized by this method, which begins with creating a Configuration object using the provided builder. It is important to set the consumer key and consumer secret using the appropriate builder's methods. Having the Configuration object ready we create TwitterFactory, whose main responsibility is creating instances of the Twitter class. Twitter is the main class which gives you access to Twitter's API methods. We obtain the new instance using TwitterFactory's getInstance() method. Then we create an OAuth request token with a specific callback URL and start a new activity by passing the authentication URL. The result of this code is that a web browser with Twitter's authentication page is opened. When the user successfully logs to Twitter, the web browser is redirected to a callback URL. If the intent-filter was properly defined, our activity should capture this redirection and the flow control should go back to the onCreate method. Therefore the rest of the authentication process is handled in the activity's onCreate method. All you have to do is to obtain the verifier which is passed as the intent's query parameter and use it to get the OAuth access token from the Twitter instance. At this point you are able to obtain information about the user by calling Twitter's showUser(long id) with the id returned by the AccessToken's getUserId() method. The last thing from the provided snippet left to be described is the onSendClicked(View v) method. It is invoked when the user clicks the send button. The goal of this method is to create a SendTwitterStatusUpdate task for sending Twitter status updates and passing all the necessary information to it (consumer keys pair, access token and message to be posted). The content of this task is provided below.

```
public class SendTwitterStatusUpdate extends AsyncTask<String, Void, Boolean>{

    private Context context;
    private String consumerKey;
    private String consumerSecret;
    private String accessToken;
    private String accessTokenSecret;

    public SendTwitterStatusUpdate(Context context, String consumerKey, String
consumerSecret, String accessToken,String accessTokenSecret) {
        this.context=context;
        this.consumerKey=consumerKey;
        this.consumerSecret=consumerSecret;
        this.accessToken=accessToken;
        this.accessTokenSecret=accessTokenSecret;
    }

    @Override
    protected Boolean doInBackground(String... params) {
        String status = params[0];

        if(status!=null) {
            try {
                ConfigurationBuilder builder = new ConfigurationBuilder();
                builder.setOAuthConsumerKey(consumerKey);
                builder.setOAuthConsumerSecret(consumerSecret);

                AccessToken token = new AccessToken(accessToken, accessTokenSecret);
                Twitter twitter = new TwitterFactory(builder.build()).getInstance(token);

                twitter.updateStatus(status);
                return true;
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return false;
    }

    @Override
    public void onPostExecute(Boolean result) {
        if(result) {
            Toast.makeText(context, "Status successfully updated",
Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(context, "Status not updated", Toast.LENGTH_LONG).show();
        }
    }

}
```

[Code 13] Content of SendTwitterStatusUpdate task

The task's constructor is pretty self-explanatory. All it does is store parameters as attributes to make them accessible from other methods. The most interesting part of this class is the doInBackground method since it presents the usage of Twitter's API. All you need to do is get the Twitter instance and then invoke the appropriate API method. In this case updateStatus() is called. The whole process looks similar to the code from TwitterSampleActivity, with the main difference being that we invoke TwitterFactory's getInstance method with one parameter – the token being an instance of the AccessToken class. The access token is needed to confirm that the user permitted our app to communicate with the Twitter server on his/her behalf. If everything goes without a problem the user's Twitter status is updated. This sample app makes use of one of Twitter4J's simplest functionalities i.e. getting the user's info and posting a status update on his behalf. It is worth mentioning that this library gives you much more possibilities e.g. you can search for different Tweets or create a live stream of Tweets with the Streaming API.

We hope that you find this guide on integrating Facebook, Twitter and Dropbox into your applications useful. If you have any issues, please let us know via the Samsung Developer forums.

Ref:http://developer.samsung.com/android/technical-docs/Using-Facebook-Dropbox-and-Twitter-APIs