**ejdelrosarioFramework**

Android Studio Library / Framework

written by: Ernest John Del Rosario

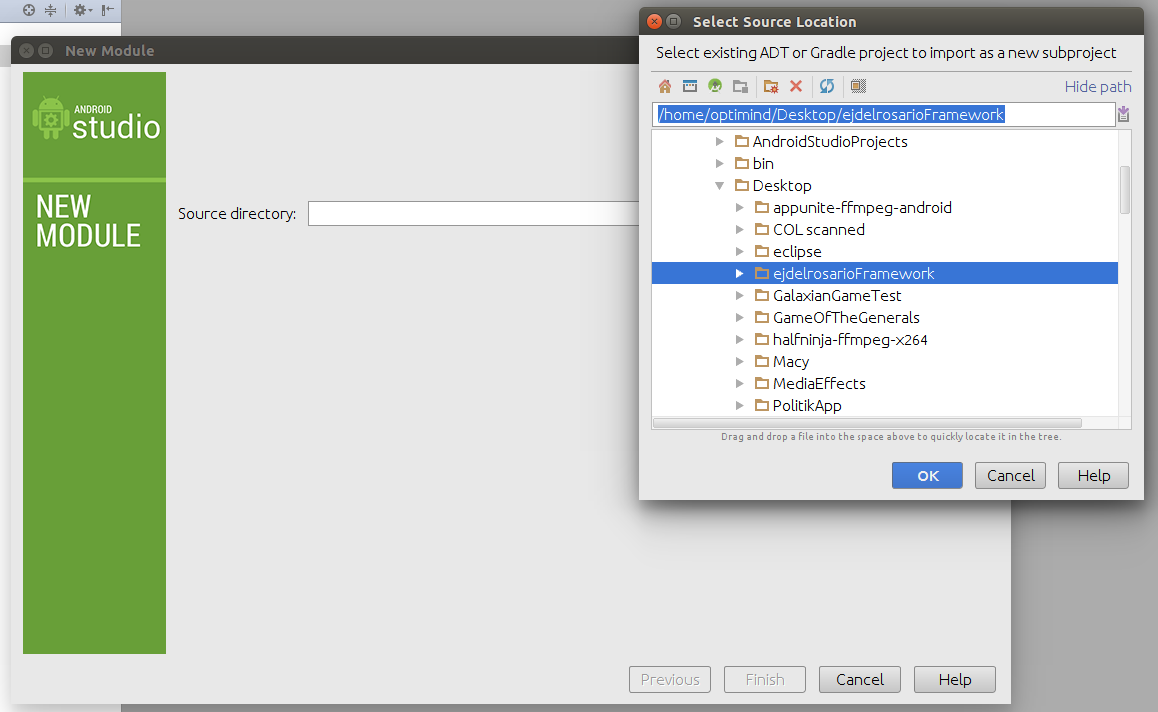
This library contains some of the common modules or features being used across all android applications such as database and webservice connections, based on an existing framework I've personally used before. This document will walk you through on how to include and use this library into your android application project aside from the java doc of this library. This walk-through assumes that you understand some of the OOP concepts in JAVA.

**1. Importing**

While your project is open in Android Studio, select File > New > Import module...

A dialog should appear asking for the Source Directory, click the open directory icon ( … ) then select the library.

Figure 1.1



Go to your Project Properties. Right-click “app” module (your main module in project) > Open Module Settings... Then go to Dependencies tab click the plus ( + ) sign on the right-hand part of the window and select Module Dependency and select the library then hit OK.

Figure 1.2

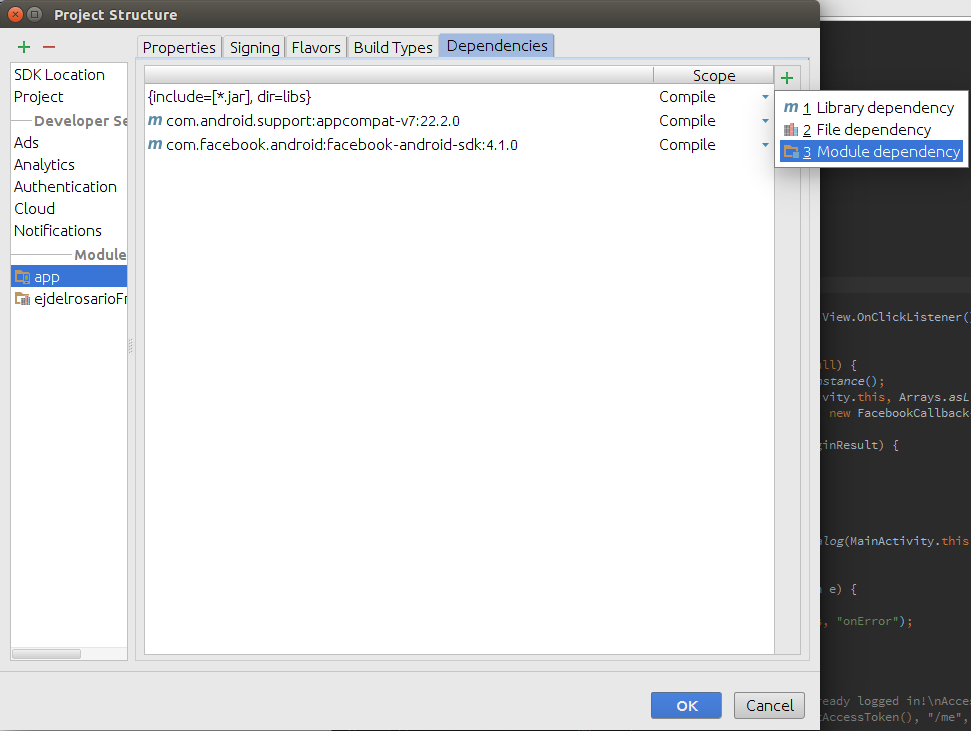
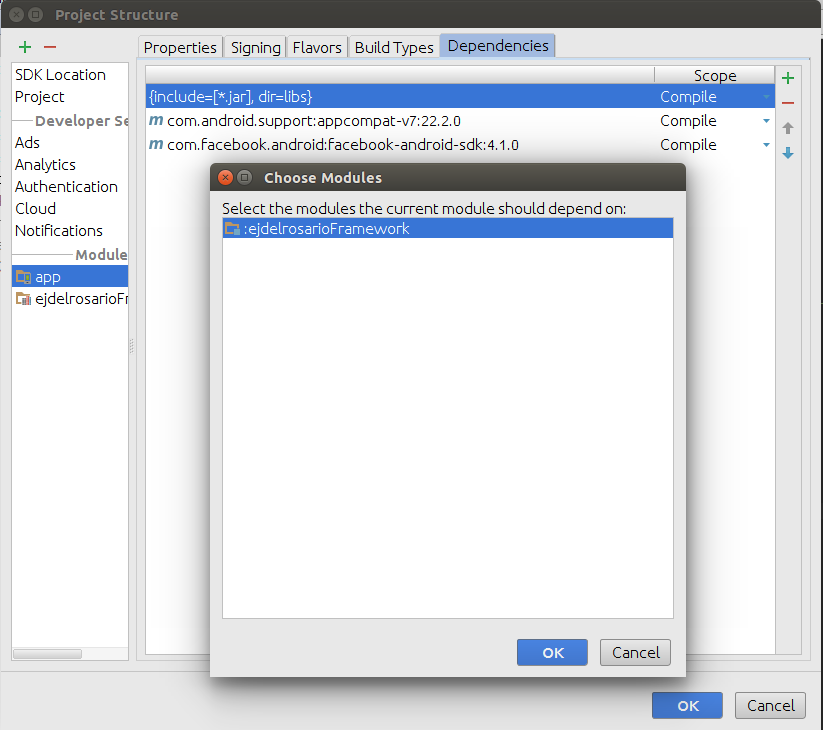


Figure 1.3



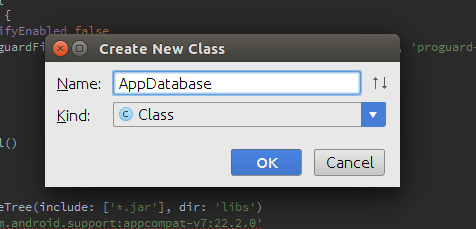
Note that after hitting ok on the Project Structure window, Android Studio will automatically update your gradle file to add such dependency to your project and will sync gradle right after you click OK.

You are now ready to use this library in your project. Let's start with the database.

**2. Database Helper, Engine Database and Tables**

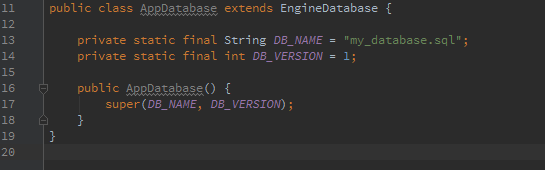
Create first your database class. For this expample, let's use AppDatabase as the name.

Figure 2.1



Extend it as Engine Database (ejdelrosario.framework.database.EngineDatabase), declare your String database name and int version then create your constructor supplying the constructor of Engine Database via “super”.

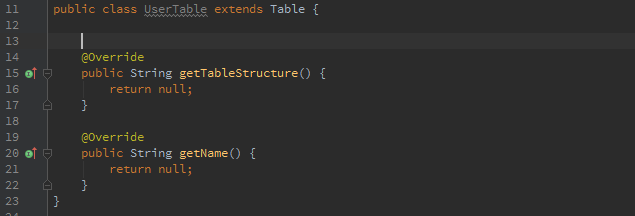
Figure 2.2



Now that we have our Database class ready, let's leave it there first and proceed to Tables.

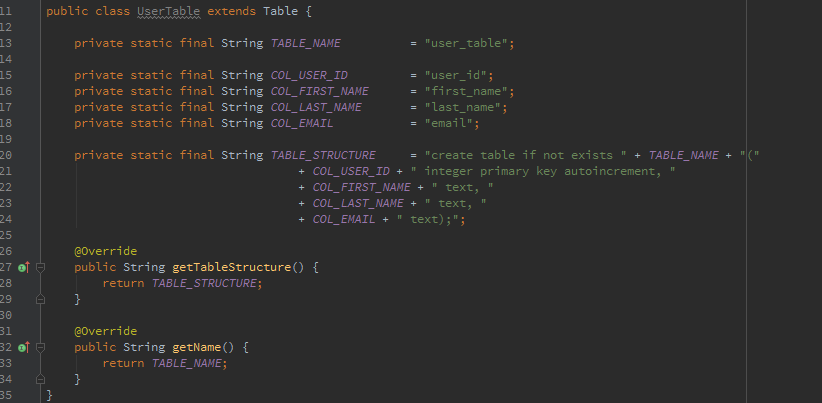
Create a New Class and for this example, let's use UserTable as the name and extend it as Table (ejdelrosario.framework.databse.Table) and implement abstract methods.

Figure 2.3



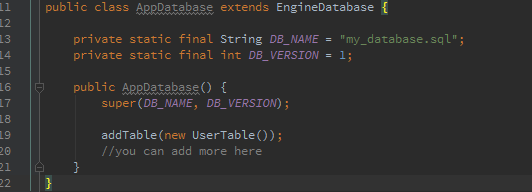
Now declare your Table name, Columns of the table and the Structure or the create query for this table then supply it to their corresponding method returns.

Figure 2.4



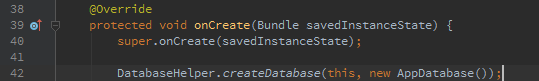
Let's just leave this table first and get back to our AppDatabase. Since we already have a structured Table, we can now add the Table to our Database. Just call the inherited method addTable( Table table ) and pass a new instance of your table in your Database's constructor.

Figure 2.5



Now that we have our Database and included Tables set up, we have to actually start creating the database instance for it to be usable accross the application. We can do that by calling our DatabaseHelper class and calling the createDatabase(Context, EngineDatabase) method statically.

Figure 2.6

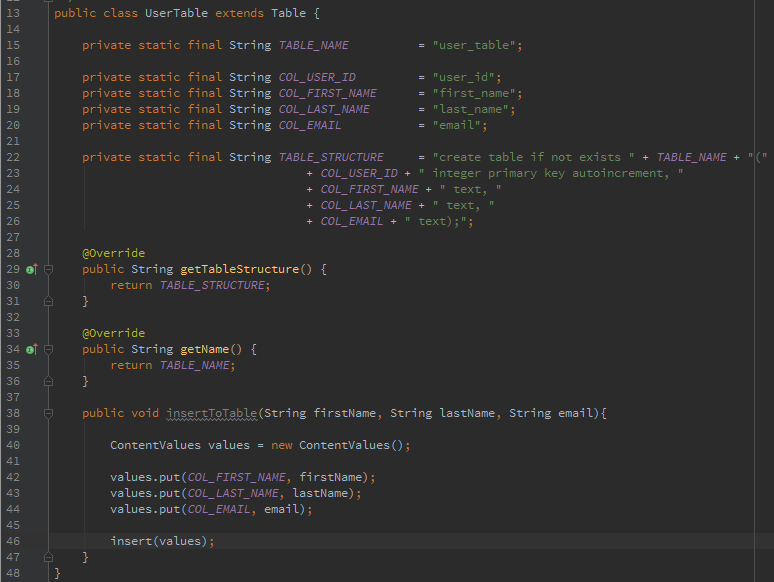


Creating the database instance will be done statically so you can just create the instance once for your entire application's life cycle, you don't have to create the database's instance everytime you're going to do database activities. So create the instance preferrably on the first Activity when your Application has started (commonly on Splash screen). But note that sometimes, Android automatically shuts down an application's life cycle when left in background long enough or more space in the memory is needed for the currently running application so watch out for some NullPointerExceptions when your application has been in background for a long time and then restored to foreground.

Going back to our table, since we can have separate classes for our different tables, we can now have a much cleaner approach when it comes to handling data for each table unlike the traditional one-class-for-all-tables approach like in every tutorial we can find on the internet when we we're first studying databases in android.

We can implement a separate method for inserting, updating, and removing data for each table classes dedicated solely for all the elements inside that specific table. For this example, we will create an insert method for our UserTable.

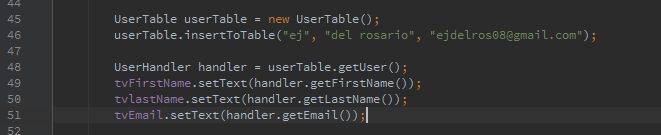
Figure 2.7



Just explore the Table class to see all the available public methods.

You can then access the table's methods by just creating a new instance anytime like this:

Figure 2.8

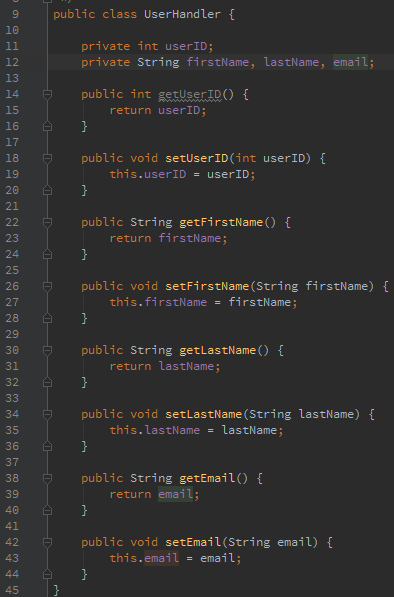


Note that you can always create a new instance of a table anytime and anywhere when you need it.

Just to add some tips for this Table approach, you can also implement your method for queries with your desired return like I did for the example on Figure 2.8 to populate the UI with the user's information.

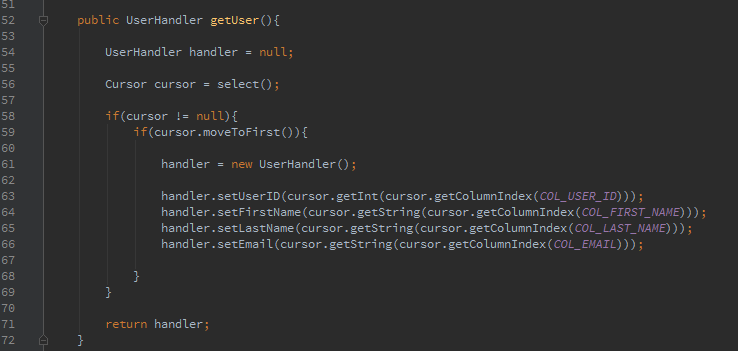
I did that by first creating a setter and getter class to handle grouped data when queried from the table. (Encapsulation)

Figure 2.9



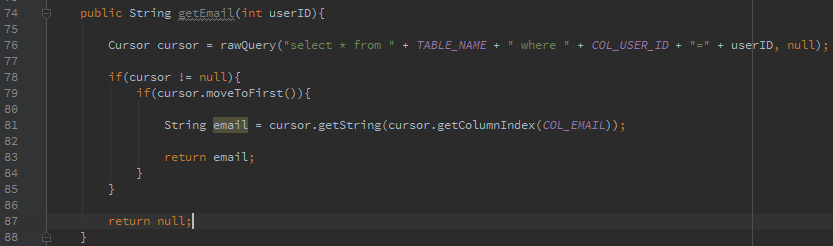
And then using it in the query method inside the Table. This is equivalent to select all query.

Figure 2.10



You can also have rawQueries

Figure 2.11



again, just explore the Table class to know more about the available public methods.

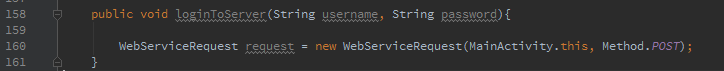
Now, let's proceed to webservice class.

**3. Webservice Request**

Another common feature among Applications is connecting through your own API or webservice and the WebserviceRequest class is designed for that task. Note that this class extends AsyncTask so you don't have to worry about threading, this behaves asyncronously.

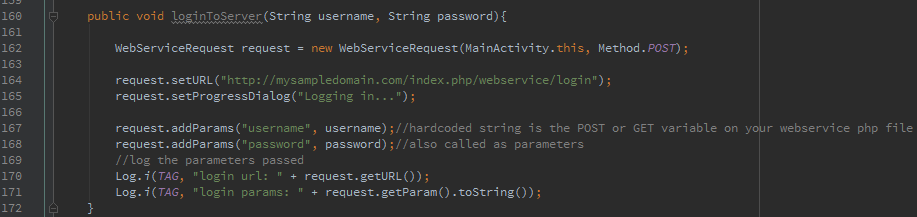
First, create an instance of WebserviceRequest class supplying the required paramters on its constructor, Context and Method (enum). Note that we only have two (2) Methods: GET and POST.

Figure 3.1



Next, set the URL, the progress message(optional), and the web parameters; you can also get the arguments you've passed for logging to track or debug.

Figure 3.2



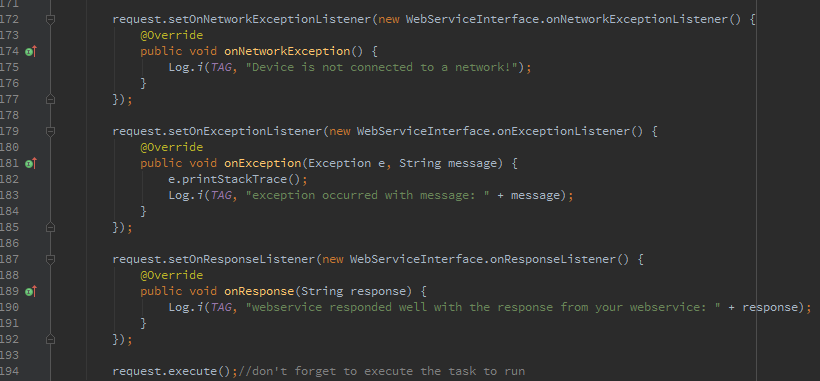
Note that if you don't supply the setProgressDialog method, there will be no progressDialog to be displayed while the webservice task is requesting. And it is always important to let the users know what's happening on most of the cases.

Next, set the three (3) listeners to handle various events on this process. First one is the NetworkExceptionListener: called when the device is not connected to a network.

ExceptionListener: called when the connection has timed out, when parameters passed contains unsupported encoding, when client (httpClient) exception has occurred, Unknown Host exception occurred and when a general IOException occurred; each with corresponding messages.

ResponseListener: called when the webservice request responded successfully and returns the response of the webservice called.

Figure 3.3



The process will be just the same when you set the Method to GET, you don't have to manually embed the arguments on the url string, just use addParam and the class will embed it to the url for you.

Don't forget to add the required permissions to perform this process in your app's manifest.

Figure 3.4



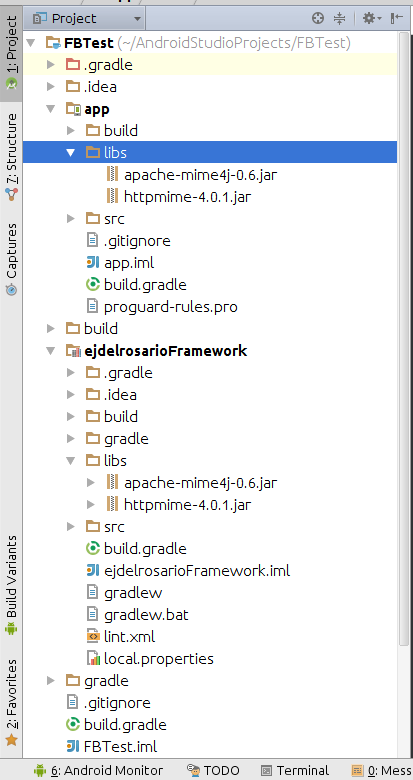
**4. Webservice Upload**

WebserviceUpload class has similar approach with Webservice Request. The only difference is the setting of arguments or parameters and the arguments for the constructor.

First off, add the dependencies required for this process to your project. The apache-mime4j-0.6.jar and httpmime-4.0.1.jar located at the library's libs directory.

Copy those jar files from the ejdelrosarioFramework/libs and paste it into your app's libs folder.

Figure 4.1



Open again the Project Properties / Structure window of your app, go to Dependecies tab and hit the plus ( + ) button on the right-hand part of the window > Fil dependency and select the 2 jar files mentioned from your app's libs folder.

Figure 4.2

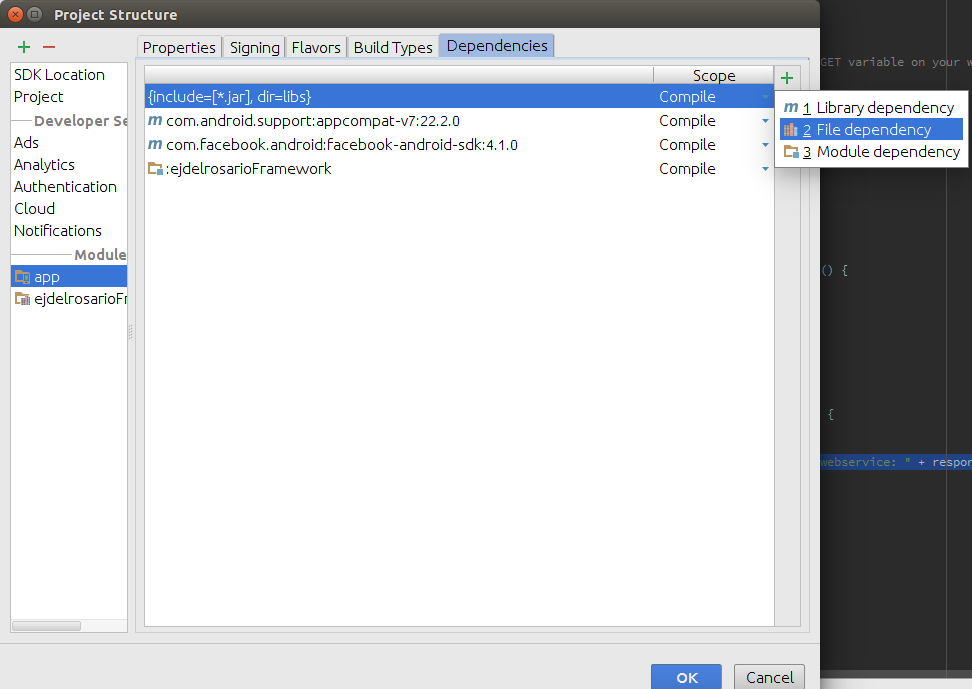
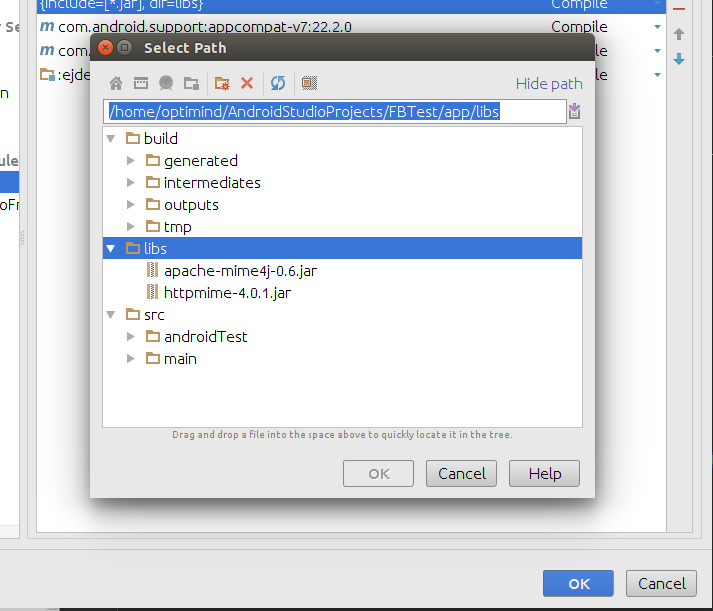


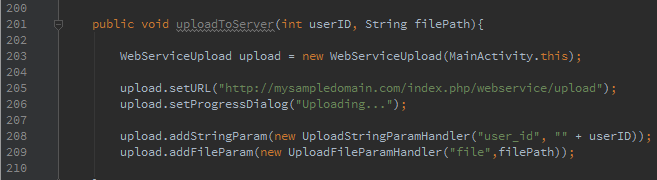
Figure 4.3



Now that all has been set up for Webservice Upload, let's start implementing this function.

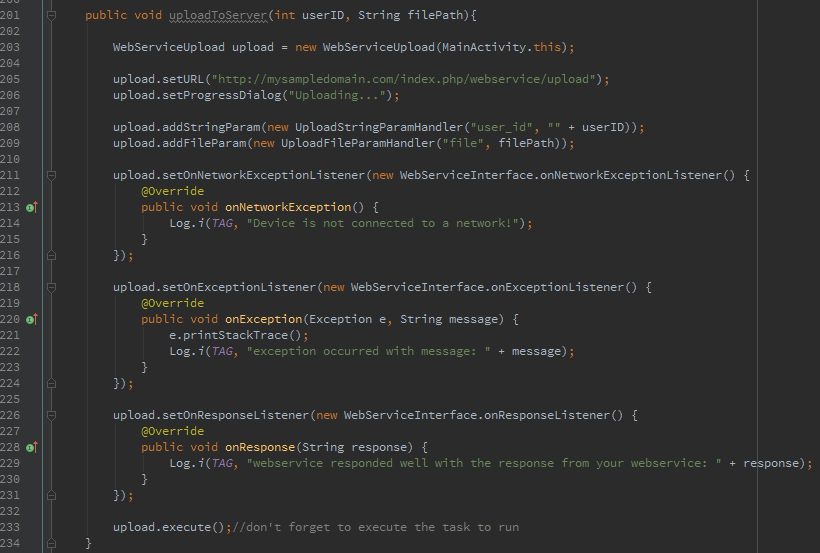
First, get a new instance of WebserviceUpload class passing a context to its constructor then set the url and progressDialog message just like in WebserviceRequest. To add a file argument in the parameter for the webservice, use the setter and getter class UploadFileParamHandler class supplying the key POST paramter and the full file path. If you want to add a string parameter in addition to the file parameter, use the UploadStringParamHandler class.

Figure 4.4



And then add the three listeners just like in the WebserviceRequest.

Figure 4.5



For now, the upload progress (in percent) will be shown along with the progressDialog message. So if you don't set the progressDialog message, the progress of the upload will not be shown. A progress listener will be best to implement for you to handle the callback but it's not yet available at this time.

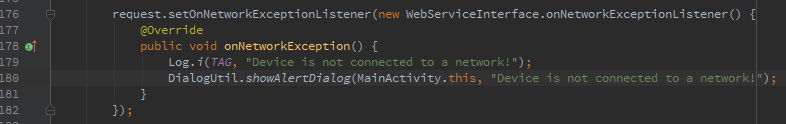
**5 Utilities**

**Dialog Util**

The most common element being used in Android is the AlertDialog. DialogUtil class helps you to easily build and show AlertDialogs in a static manner, so you can access this anytime and anywhere easily provided that you have the required parameters for the methods.

Just call the DialogUtil class and pick the method you want to use. Let's use our WebserviceRequest as the example.

Figure 5.1

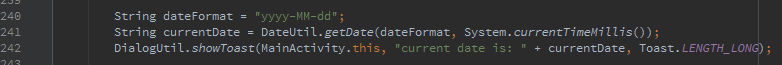


As simple as that. Just explore the DialogUtil class to know all the available public methods.

**Date Util**

DateUtil class provides date parsing methods, date difference calculation (in days) and getting date on a given format. Just like DialogUtil, all the methods in this class can be accessed statically.

Figure 5.2



Like I've said on the DialogUtil, explore the class to know more about the public methods available.