AUG
28

Android: Implement KSoap2 in an AsyncTask

When writing an application regardless of the platform it runs on, it will in most cases need to interact with other applications, devices, data sources, or all of the above. It is rare that an application of any value will not connect to anything at all.

So as I started writing my first Android application, I needed to connect to .NET web services, interact with a SQLite database, and work with bluetooth devices. Fortunately, in many ways, C# and Java have much in common and it wasn't too difficult to transition to Java. However, the Android platform and the Java framework are definitely different.

One of the primary rules of Android development is that you should never do operations on the user interface (UI) thread that could make the device appear to hang or otherwise become unresponsive. The generally recommended way of handling processes that would cause UI issues is to run them in an AsyncTask.

The AsyncTask is a very handy way to separate database interaction, web service interaction, or anything that could potentially cause any UI delays. By extending the AsyncTask class we can effectively execute our long running code in a separate thread and not cause issues with the UI.

In this blog, I will demonstrate how to access a Microsoft .NET web service using KSoap inside an AsyncTask.

When you extend the SQLiteOpenHelper you will need to be sure you have the constructor setup as shown below.

```
    public SQLDatabase(Context context, String name, CursorFactory factory,
            int version) {
        super(context, name, factory, version);
        // TODO Auto-generated constructor stub
        myContext = context;
    }
```

As you can see, there are 4 parameters that you will need to pass in through the constructor. First is your application context, followed by the name of the database, then a cursor to use or null for default. I most always pass null. Lastly, you need to pass in the version number. Version number must begin with 1.   This will be compared to the version of the database (if it already exists) and cause the onUpgrade() event to fire if there is a difference. Note: In the constructor I will get a context reference for use later (using a local Context class object named myContext).

Next step will be to override the onCreate() event: As shown below, I will modify the onCreate() event override so that when it fires, I will create my table which is defined locally as TBL_SAMPLE. I won't go into the details of SQL scripts, that will be a topic for a future blog. Suffice to say that the variable TBL_SAMPLE contains the appropriate script to create a "product" table.

```
    @Override
    public void onCreate(SQLiteDatabase db) {
        // TODO Auto-generated method stub
        String SQL = "";
        try{
            if(db.isOpen()){

                //create tables here
                db.execSQL(this.TBL_SAMPLE);
            }

        }
        catch(Exception e){
            Log.d("onCreateDB", e.getMessage());
            return;
        }
        finally{

        }

    }
```

Next we need to add some functionality to our new SQLite Helper object. Below is a sample Save function.

```
public boolean saveProduct(Product product){

        final String TABLE = "products";
        ContentValues myValues = new ContentValues();
        final String COL_ID = "id";
        final String COL_UPC = "upc";
        final String COL_SKU = "sku";
        final String COL_PRODUCTNAME = "productName";
        final String COL_MANUFACTURERNAME = "manufacturerName";

        SQLiteDatabase database = this.getWritableDatabase();
        int rowsAffected = 0;
        String[] Values = new String[1];
        String Where = "";


        try{
            database = this.getWritableDatabase();

            if (database.isOpen()){

                database.beginTransaction();

                myValues = new ContentValues();

                myValues.put(COL_ID, product.getProductID());
                myValues.put(COL_UPC, product.getUPC());
                myValues.put(COL_SKU, product.getSKU());
                myValues.put(COL_PRODUCTNAME, product.getName());
                myValues.put(COL_MANUFACTURERNAME, product.getMfgName());

                Values[0] = String.valueOf(product.getID());

                Where = COL_ID + "= ?";

                rowsAffected = database.update(TABLE, myValues, Where , Values);

                // Check to see if anything was really updated
                if (rowsAffected == 0){
                    //Insert row since it doesn't truly exist.
                    database.insert(TABLE, "", myValues);
                }

                database.setTransactionSuccessful();

                database.endTransaction();

                return true;
            }
            else{
                return true;
            }

        }
        catch(SQLiteException e){
            Log.e("SAMPLEDB", e.getMessage());
            return false;
        }
        finally{
            database.close();
        }
    }
```

It is important to note that I am instantiating a database connection, interacting with the database, and releasing the database inside this function. Calling the getWriteableDatabase() gives me an open database and in the finally() clause I am closing the database.

Now that we have a basic implementation. Here is how you would instantiate and use it from an AsyncTask. Inside the doInBackground override is where you will do all of your work. Below is an example of using the SQLiteOpenHelper in the AsyncTask.

```
@Override
    protected boolean[] doInBackground(Product... selectedProducts) {
        // TODO Auto-generated method stub
        Product product;
        SQLiteDatabase db;
        boolean[] results = new boolean[selectedProducts.length];

        try{

            db = new SQLDatabase(myContext, GlobalInformation.getDBFilePath(myContext),

            for(int p=0;p<selectedProducts.length;p++){
                product = selectedProducts[p];

                db.saveProduct(product);
            }

            return results;
        }
        catch(Exception e){

            Log.e(TAG, e.getMessage());
            return null;
        }
        finally{

            db.close();

        }
    }
```

Notice that I also close out the db instance in the finally() clause. I always strive to open, interact, and close the database in one function. Each call to interact with the database is responsible for opening/closing the database.

The full SQLiteOpenHelper is shown below:

```
package com.example.sample;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class SQLDatabase extends SQLiteOpenHelper {

    private Context myContext;
    private final String TBL_SAMPLE = "create table if not exists products(id integer primary
                                       "productName,  " +
                                       "upc,  " +
                                       "sku,  " +
                                       "manufacturerName)";



    public SQLDatabase(Context context, String name, CursorFactory factory,
            int version) {
        super(context, name, factory, version);
        myContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String SQL = "";
        try{

            if(db.isOpen()){

                //create tables here
                db.execSQL(this.TBL_SAMPLE);
            }




        }
        catch(Exception e){
            Log.d("onCreateDB", e.getMessage());
            return;
        }
        finally{

        }

    }

    @Override
    public void onUpgrade(SQLiteDatabase arg0, int arg1, int arg2) {
        // TODO Auto-generated method stub

    }
    public boolean saveProduct(Product product){

        final String TABLE = "products";
        ContentValues myValues = new ContentValues();
        final String COL_ID = "id";
        final String COL_UPC = "upc";
        final String COL_SKU = "sku";
        final String COL_PRODUCTNAME = "productName";
        final String COL_MANUFACTURERNAME = "manufacturerName";

        SQLiteDatabase database = this.getWritableDatabase();
        int rowsAffected = 0;
        String[] Values = new String[1];
        String Where = "";


        try{
            database = this.getWritableDatabase();

            if (database.isOpen()){

                database.beginTransaction();

                myValues = new ContentValues();

                myValues.put(COL_ID, product.getProductID());
                myValues.put(COL_UPC, product.getUPC());
                myValues.put(COL_SKU, product.getSKU());
                myValues.put(COL_PRODUCTNAME, product.getName());
                myValues.put(COL_MANUFACTURERNAME, product.getMfgName());

                Values[0] = String.valueOf(product.getID());

                Where = COL_ID + "= ?";

                rowsAffected = database.update(TABLE, myValues, Where , Values);

                // Check to see if anything was really updated
                if (rowsAffected == 0){
                    //Insert row since it doesn't truly exist.
                    database.insert(TABLE, "", myValues);
                }

                database.setTransactionSuccessful();

                database.endTransaction();

                return true;
            }
            else{
                return true;
            }

        }
        catch(SQLiteException e){
            Log.e("SAMPLEDB", e.getMessage());
            return false;
        }
        finally{
            database.close();
        }
    }
}
```

There you go. You can add more functions for other operations and include more table, view, index, constraint creation in the onCreate() override. This is good start.

One last note: The Product class is a custom class object to contain my product information.

Posted 28th August 2012 by 10KEyes

Labels: SQLite, Android, AsyncTask, SQLiteOpenHelper

+1 Tweet Like

AUG
28

Android: Implement KSoap2 in an AsyncTask

When writing an application regardless of the platform it runs on, it will in most cases need to interact with other applications, devices, data sources, or all of the above. It is rare that an application of any value will not connect to anything at all.

So as I started writing my first Android application, I needed to connect to .NET web services, interact with a SQLite database, and work with bluetooth devices. Fortunately, in many ways, C# and Java have much in common and it wasn't too difficult to transition to Java. However, the Android platform and the Java framework are definitely different.

One of the primary rules of Android development is that you should never do operations on the user interface (UI) thread that could make the device appear to hang or otherwise become unresponsive. The generally recommended way of handling processes that would cause UI issues is to run them in an AsyncTask.

The AsyncTask is a very handy way to separate database interaction, web service interaction, or anything that could potentially cause any UI delays. By extending the AsyncTask class we can effectively execute our long running code in a separate thread and not cause issues with the UI.

In this blog, I will demonstrate how to access a Microsoft .NET web service using KSoap inside an AsyncTask.

First you will need to create a class object extending the AsyncTask class as shown below.

```
import android.os.AsyncTask;
public class WebSearchTask extends AsyncTask<String, String, WebResponse> {
    @Override
    protected WebResponse doInBackground(String... params) {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    protected void onPostExecute(WebResponse result) {
        // TODO Auto-generated method stub
        super.onPostExecute(result);
    }
    @Override
    protected void onPreExecute() {
        // TODO Auto-generated method stub
        super.onPreExecute();
    }
    @Override
    protected void onProgressUpdate(String... values) {
        // TODO Auto-generated method stub
        super.onProgressUpdate(values);
    }
}
```

The first parameter (a String)

```
public class WebSearchTask extends AsyncTask<String, String, WebResponse> {
```

is passed to the doInBackground() function as an array.

```
protected WebResponse doInBackground(String... params) {
```

In this example we have used a String. Anything within the doInBackground function runs in another thread and not in the UI. The doInBackground method outputs the third parameter. In this case a custom object class I called WebResponse.

The second parameter is used fpr progress updates.

```
public class WebSearchTask extends AsyncTask<String, String, WebResponse> {
```

Here we will use a String. You can use other data types if you like.

The third parameter is for the results.

```
public class WebSearchTask extends AsyncTask<String, String, WebResponse> {
```

We are using a custom user class but it can be another data type, class object, array etc. However, the doInBackground() return type must match the third parameter data type.

Additionally, in most cases we will want to know when this task has been completed and possibly what are the results. One of the ways to achieve this is when we instantiate the AsyncTask we pass a reference to our activity so that we can later call a public function or set some value there. Below I have added my class constructor and can now pass a reference for my calling Activity.

```
public class WebSearchTask extends AsyncTask<String, String, WebResponse> {
    private ProductSearchActivity myActivity;
    public WebSearchTask(ProductSearchActivity activity){
        myActivity = activity;
    }
```

Next we may want to show the progress of our task on the screen. In this example we will use a

ProgressDialog. We will need to use the onPreExecute() method to set up our dialog as shown below.

```
@Override
    protected void onPreExecute() {
        // TODO Auto-generated method stub
        super.onPreExecute();
        try {
            progDialog = ProgressDialog.show(myActivity, myActivity.getString(R.string.product_search_dialog_title), myA
        }
        catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

To update the ProgressDialog (visible on the UI thread) we will call publishProgress() and pass a String (#2 parameter declared). As shown below:

```
publishProgress(myActivity.getString(R.string.search_matching_products));
```

Now we are ready to do something with this task. We will put our "Do Something" code in the doInBackground() method. In the example below, we will use KSoap and connect to a Microsoft .NET web service.

```
@Override
```

```
protected WebResponse doInBackground(String... params) {
    // TODO Auto-generated method stub
    WebResponse webResponse = new WebResponse();
    final String SOAP_ACTION = "http://test.webexample.net/netservices.asmx/SearchProducts";
    final String WEB_FUNCTION = "SearchProducts";
    final String SOAP_ADDRESS = "http://test.webexample.net/netservices.asmx";
    final String ERROR_MSG = "ErrorMessage";
    final String ERROR_SQL = "SqlErrorMessage";
    final String DATA_SEG = "Data";



    boolean hasError = false;
    SoapSerializationEnvelope envelope;
    HttpTransportSE httpTransport;
    SoapObject response = null;
    SoapObject request = new SoapObject(SOAP_ADDRESS, WEB_FUNCTION);
        SoapObject dataSeg;
        SoapObject soapProduct;
        Product product;
        PropertyInfo pi=new PropertyInfo();
        String Qty = params[3];

        try
        {
            pi.setName("deviceId");
            pi.setValue(params[0]);
            pi.setType(String.class);
            request.addProperty(pi);
```

```
pi = new PropertyInfo();
pi.setName("partialUPC");
pi.setValue(params[1]);
pi.setType(String.class);
request.addProperty(pi);

pi = new PropertyInfo();
pi.setName("partialSKU");
pi.setValue(params[2]);
pi.setType(String.class);
request.addProperty(pi);

envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
envelope.dotNet = true;

envelope.setOutputSoapObject(request);

httpTransport = new HttpTransportSE(SOAP_ADDRESS);

publishProgress(myActivity.getString(R.string.search_matching_products));

httpTransport.call(SOAP_ACTION, envelope);

response =(SoapObject) envelope.getResponse();

if (response.hasProperty(ERROR_MSG)){
    webResponse.setErrorMessage(response.getPrimitivePropertySafelyAsString(ERROR_MSG).toString());
```

```
}

if (response.hasProperty(ERROR_SQL)){
    webResponse.setSQLErrorMessage(response.getPrimitivePropertySafelyAsString(ERROR_SQL).toString
}

if (webResponse.getErrorMessage().length() > 1){
    webResponse.setErrorState(true);
}

if (webResponse.getSQLErrorMessage().length() > 1){
    webResponse.setErrorState(true);
}

if (webResponse.errorState()){
    return webResponse;
}

if (response.hasProperty(DATA_SEG)){

    dataSeg = (SoapObject) response.getProperty(DATA_SEG);

    for(int p=0;p<dataSeg.getPropertyCount();p++){

        if (this.isCancelled()) return null;

        soapProduct = (SoapObject) dataSeg.getProperty(p);
```

```
                product = new Product();

                product.setQuantity(Qty);

                product.setID(p);

                if (soapProduct.hasProperty("ID")){
                    product.setProductID(soapProduct.getPropertyAsString("ID").toString());
                }
                if (soapProduct.hasProperty("Name")){
                    product.setName(soapProduct.getPropertyAsString("Name").toString());
                }
                if (soapProduct.hasProperty("UPC")){
                    product.setUPC(soapProduct.getPropertyAsString("UPC").toString());
                }
                if (soapProduct.hasProperty("ManufacturerName")){
                    product.setMfgName(soapProduct.getPropertyAsString("ManufacturerName").toString());
                }
                if (soapProduct.hasProperty("SKU")){
                    product.setSKU(soapProduct.getPropertyAsString("SKU").toString());
                }

                webResponse.addData(product);

            }

        }
```

```
            if (hasError){

                publishProgress(myActivity.getString(R.string.search_product_failed));

                webResponse.setErrorState(true);
                return webResponse;
            }
            else{
                publishProgress(myActivity.getString(R.string.search_product_success));

                webResponse.setErrorState(false);
            }

            return webResponse;

        } catch (Exception e) {
            // TODO Auto-generated catch block
            publishProgress("Product search failed.");

            Log.d(SOAP_ACTION, e.getMessage());
            webResponse.setErrorState(true);
            webResponse.setErrorMessage(e.getMessage());

            return webResponse;
        }

    }
```

You will find that I have a couple of custom class objects in there and it really isn't important for this blog so I won't be going into detail about the Product class or the WebResponse class objects. The point of this blog entry is to demonstrate the use of the AsyncTask with KSoap2.

You will notice that I called the publishProgress() function a few times throughout the doInBackground task. This is to update the UI so that the user knows the device is doing something and not having problems. When you call publishProgress() the AsyncTask calls onProgressUpdate() and passes the parameter you specified. My implementation is below. *Note: You do not have to use the ProgressDialog at all if you do not wish to.*

```
@Override
protected void onProgressUpdate(String... values) {
    // TODO Auto-generated method stub
    super.onProgressUpdate(values);

    if(progDialog != null && progDialog.isShowing()){
        progDialog.setMessage(values[0]);
    }
}
```

Now that we have processed something, it is time to return to my Activity and drop off what the web search found. Using the onPostExecute() function will can do that.

```
@Override
protected void onPostExecute(WebResponse result) {
    // TODO Auto-generated method stub
    super.onPostExecute(result);

    if (progDialog != null || progDialog.isShowing()){
        progDialog.dismiss();
    }

    myActivity.onWebResult(result);
}
```

As you see above we are calling a public method on the activity we passed through the constructor. In this case, our Activity would have the public function listed below. In this case this is where we would process the results of our web service call.

To call this AsyncTask you will need to instantiate it from inside your Activity, it could be inside a button click or some other event and will look something like this...

```
if (doWebSearch && GlobalInformation.isOnline(this)){

   productWebSearch = new WebProductSearchTask (this);

   searchCriteria[0] = new GlobalInformation(this.getApplicationContext()).getDeviceID();
   searchCriteria[1] = editUPC.getText().toString();
```

```
    searchCriteria[2] = editSKU.getText().toString();

    productWebSearch.execute(searchCriteria);

 }
```

As you can see above, I do check to see if I have internet access using another custom class and it is not part of this blog, but I did feel it is worth mentioning to be sure to check the internet access beforehand. Now be sure to add a public function on your activity to be able to call back to from postExecute() as shown below. Again, I am passing a custom object. You can create your own class object to return from the postExecute() back to your Activity.

```
public void onWebResult(WebResponse result){

   //DO SOMETHING HERE

 }
```

It is also important to note that in the onPostExecute() function we need to be sure to close the ProgressDialog if we instantiated it. You will notice above, that I check to see if it is showing and if so get rid of it.

That pretty much sums up how to implement KSoap2 in the AsyncTask. There can be many variations to this. I will follow this up with how to implement SQLite in the AsyncTask in another blog.

I would also like to add that you should also override your Activity onDestroy() event, so that you cancel your AsyncTask if for some reason your Activity gets killed. It should look something like below.

```
@Override
protected void onDestroy() {
  // TODO Auto-generated method stub
   super.onDestroy();

   if(productWebSearch != null && productWebSearch.getStatus() != AsyncTask.Status.FINISHED){
     productWebSearch.cancel(true);
   }

 }
```

If anyone has any questions or comments let me know.

Posted 28th August 2012 by 10KEyes

Labels: .NET, KSoap2, Web Services, Android, AsyncTask

+1 Tweet Like

AUG
17

Ubuntu 11.04 Only Accepts Long Key Press On Login

ISSUE: I attempted to log into my Ubuntu laptop this morning only to find that the keyboard was not working. After the initial panic I discovered that only the long key press would make a character.

SOLUTION: At the log in screen go into the Accessibility Options and un-check the Slow Keys option.

Apparently, I had accidentally activated it. I believe I was plugging in my USB mouse receiver as the log in screen was coming up.

Posted 17th August 2011 by 10KEyes

Labels: login, password, Ubuntu 11.04, keyboard not working

+1 Tweet Like

AUG
12

E-Reader: Kindle vs Nook vs Tablet

The purpose of an e-reader is to read e-books and more recently other forms of electronic visual content such as newspapers and e-zines. E-books are electronic forms of the books you would find at the library. Books most often have few to no pictures and typically have black print on white pages. If the books have pictures they are generally simple gray scale pictures or line art. The same can be said for most newspapers. However, magazines generally have less print and much more color and pictures.

The first e-book readers were designed to emulate a book and printed paper. This means that the screen would look much like a page of a book. Amazon created the Kindle to do just that. It was designed to emulate a book and they used E Ink technology. Barnes & Noble and other e-reader manufacturers followed suit. As the e-reader software evolved and was put on desktop, laptop computers, tablets, and smart phones that had color screens everyone wanted it to handle color content.

The Amazon Kindle 3(not color) is an awesome device. It has an E Ink Electronic Paper screen. It is low power consumption with long battery life. Barnes & Noble has a similar E-Reader with similar technology called the Nook. Both Amazon and Barnes & Noble have color versions of their E-Reader. There are other E-Reader manufacturers out there but the two big ones are Amazon and Barnes & Noble.

I personally like the Amazon Kindle. That is the E-Reader I own. However, both the Kindle and Nook are good. Based on my observations the biggest difference in price for the E-Ink versions are about $5 after you figure in a comparable cover and any shipping if charged, with both coming in close to $200. I believe the last time I checked, both Amazon and Barnes & Noble offer FREE SHIPPING on their E-Readers.

Another point to take into consideration is purpose. I personally only want to read e-books on my Kindle. The Kindle was designed for reading e-books and it does that very well. Even the screen is well suited for reading and can be read from odd angles, even in bright sunlight, just like a book. It is a fact that reading a computer screen is worse for you than reading printed paper. So my opinion is that if my Kindle screen emulates the printed page then it must be better than a color computer screen for reading my books.

Yes I can go online with the built in web-browser, I can play MP3s, I can download games for it. Honestly, the user interface really doesn't lend itself to doing much more than reading a book. Seriously, how well could it possibly be a laptop when it was designed to be an E-Book reader? It was not really designed to play games, run apps, or browse the web. The same could be said for the Nook.

Often times people will want the E-Book reader to be their own little laptop. They get the color versions of whichever E-Reader and then try to multipurpose it. Yes, it may give them a portable computing device, however, it is not very well suited for replacing a laptop or desktop. I believe when this happens they get a device that may do a few different things, but none of them it can do perfectly. Additionally, these color screens are not very well suited to bright sunlight and outside reading. Anyone familiar with laptop computer screens realize that in bright sunlight they are not nearly as easy to read even if you are directly at it.

Then there are the tablets such as the iPad. You can download to these devices and run the freely available e-reader software from Amazon and Nook. Laptops, net-books, and tablets sure make reading books easier while on the go, but again they have the LCD type screens that are not very good for reading while outside in the bright sunshine. But they certainly do other things very well.

I personally think that if you want an e-reader, then get something that emulates a book. If you want a tablet or notebook, just get one. If you want to get something to read e-books then I would recommend the Amazon Kindle with E Ink Electronic Paper.

Posted 12th August 2011 by 10KEyes

Labels: Amazon Kindle Tablet, e-book, e-reader, E-Ink

+1 Tweet Like