**455-555 Week 7 Session A Notes**

Chapter 9 – Sqlite3 (SQL tools & CLI)

Chapter 10 More on Content Providers & Search (See Week 7 Content Provision demo)

Recycling Memory - Use of Memory & Recovery agents

Chapter 11 – Intro to Async Processing (See Week 7 JSON-ALERT demo)

[ Sqlite tools & CLI ]

* [Downloads - DB Browser for SQLite (sqlitebrowser.org)](https://sqlitebrowser.org/dl/) – all OSes
* [SQLite Download Page](https://www.sqlite.org/download.html) (Windows)
* Android Studio -> Built in [Database Inspector](https://developer.android.com/studio/inspect/database)!

[ **Content Providers** ]

Diagram

Description automatically generated

**Content URI(Uniform Resource Identifier)** is the key concept of Content providers. To access the data from a content provider, URI is used as a query string.

Purpose:

A content provider in Android is a component that manages access to a central repository of data. It serves as an interface to data saved in a structured format and provides mechanisms for defining data security. The main purpose of using a content provider is to manage access to data stored by an application or stored by other apps and provide a way to share data with other apps.

Content providers encapsulate the data and provide an abstraction that lets you make modifications to your application data storage implementation without affecting other applications that rely on access to your data. Here are some of the reasons why you might want to use a content provider:

* To share data between applications.
* To provide a way for users to copy complex data or files from your application to other applications.
* To provide custom search suggestions using the search framework.
* To expose your application data to widgets.
* To implement the AbstractThreadedSyncAdapter, CursorAdapter, or CursorLoader classes.

Content providers are primarily used by other applications, which access the provider using a provider client object. Together, providers and provider clients offer a consistent, standard interface to data that also handles interprocess communication and secure data access.

***Structure of a Content URI:****content://authority/optionalPath/optionalID*

**content:// –**Mandatory part of the URI as it represents that the given URI is a Content URI.

**authority –**Signifies the name of the content provider like contacts, browser, etc. This part must be unique for every content provider.

**optionalPath –**Specifies the type of data provided by the content provider. It is essential as this part helps content providers to support different types of data that are not related to each other like audio and video files.

**optionalID –**It is a numeric value that is used when there is a need to access a particular record.

**Operations used**: CRUD

Sample source:

public class MyContentProvider extends ContentProvider {

public static final Uri CONTENT\_URI   
 Uri.parse("content://com.example.myapp.provider");

private SQLiteDatabase db;

@Override

public boolean onCreate() {

DBHelper dbHelper = new DBHelper(getContext());

db = dbHelper.getWritableDatabase();

return (db != null);

}

@Override

public Cursor query(Uri uri, String[] projection, String selection, String[]

selectionArgs, String sortOrder) {

return db.query("mytable", projection, selection, selectionArgs, null, null, sortOrder);

}

@Override

public Uri insert(Uri uri, ContentValues values) {

db.insert("mytable", null, values);

return uri;

}

@Override

public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {

return db.update("mytable", values, selection, selectionArgs);

}

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

return db.delete("mytable", selection, selectionArgs);

}

@Override

public String getType(Uri uri) {

return "vnd.android.cursor.dir/vnd.example.myapp.provider.mytable";

}

}

The manifest inclusions:

To specify the permissions that client apps need in order to access the provider's data, you need to add the necessary permissions to your manifest file. Client applications specify the permissions they require in their manifest file using the

<uses-permission> element

A provider's application can specify permissions that other applications must have to access the provider's data. When you specify this element in your manifest, you are requesting this permission for your application. When users install your application, they implicitly grant this request.

The AndroidManifest file must contain the content provider name, authorities, and permissions which enable the content provider to be accessed.

Sample manifest:

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.myapp">

<application>

<provider

android:name=".MyContentProvider"

android:authorities="com.example.myapp.provider"

android:exported="true" />

<!-- Other components of the application -->

</application>

<uses-permission android:name="android.permission.READ\_CONTACTS" />

<uses-permission android:name="android.permission.WRITE\_CONTACTS" />

</manifest>

**[ Recycling memory concept ]**

**A screenshot of a cell phone

Description automatically generated**

Recycling

**Improving Performance with the** [**ViewHolder**](https://stackoverflow.com/questions/21501316/what-is-the-benefit-of-viewholder-pattern-in-android) **Pattern**

To improve performance, we should modify the custom adapter by applying the **ViewHolder** pattern which speeds up the population of the ListView considerably by caching view lookups for smoother, faster item loading (see bold code for the tweaks):

public class QuoteAdapter extends BaseAdapter {  
  
 *// View lookup cache* **private class ViewHolder {  
 ImageView thumbnail;  
 TextView quote;  
 }**  
 **private Context mContext;**

::   
 public QuoteAdapter(Context c) {  
 mContext = c;  
 mInflator = (LayoutInflater)  
 mContext.getSystemService(Context.*LAYOUT\_INFLATER\_SERVICE*);  
 mDataSource = new DataSource();  
 **this.context = context;**  
 }

::

**public View getView(int position, View convertView, ViewGroup parent) {  
  
 *// Get the data item for this position  
 // Check if an existing view is being reused, otherwise inflate the view* ViewHolder viewHolder; *// view lookup cache stored in tag* if (convertView == null) {  
 *// If there's no view to re-use, inflate a brand new view for row* viewHolder = new ViewHolder();  
 *//LayoutInflater inflater = LayoutInflater.from(context);* convertView = mInflator.inflate(R.layout.*list\_item\_layout*, parent,**

**false);  
 viewHolder.thumbnail = (ImageView)**

**convertView.findViewById(R.id.*thumb*);  
 viewHolder.quote = (TextView) convertView.findViewById(R.id.*text*);  
  
 *// Cache the viewHolder object inside the fresh view* convertView.setTag(viewHolder); *//MEMORY AGENT!!!!!* } else {  
 *// View is being recycled, retrieve the viewHolder object from tag* viewHolder = (ViewHolder) convertView.getTag(); *//RECOVERY AGENT!!!!!* }  
 *// Populate the data from the data object via the viewHolder object  
 // into the template view.* Log.*i*("Pos", ""+ position);  
 viewHolder.thumbnail.setImageResource(mDataSource.getmPhotoPool()  
 .get(position));  
 viewHolder.quote.setText(mDataSource.getmQuotePool().get(position));  
  
 *// Return the completed view to render on screen* return convertView;**}

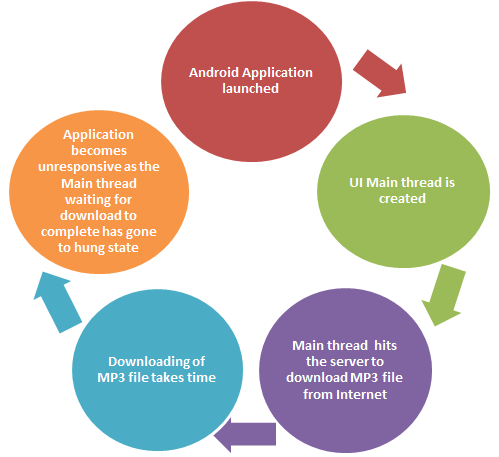
}

In this example we also have a private static class called **ViewHolder**. Making calls to findViewById() can be **slow** in practice, and if your adapter has to call it for each View in your row for every single row then you can often run into performance issues.

What the ViewHolder class *does* is ***cache*** the call to findViewById(). Once your ListView has reached the max amount of rows it can display on a screen, Android is smart enough to begin recycling those row Views. We check if a View is recycled with **if (convertView == null)**. If it is not null then we have a recycled View and can just change its values, otherwise we need to create a new row View. The magic behind this is the setTag() method which lets us attach an arbitrary object onto a View object, which is how we save the *already* inflated View for future reuse.

**[ Async Tasks ]**

AsyncTask is an *abstract* Android class which helps the Android applications to handle the Main UI thread in efficient way. AsyncTask class allows us to perform *minor* lasting tasks/**background operations** and **show** the result on the UI thread *without* affecting the **main** thread.



A screenshot of a social media post

Description automatically generated



Sample retrieval of data from the web using HTTP request (Check Demo module in BB)

package com.example.asyncjson;  
import android.app.AlertDialog;  
import android.content.DialogInterface;  
import android.os.AsyncTask;  
import android.os.Bundle;  
import androidx.appcompat.app.AppCompatActivity;  
import java.io.BufferedInputStream;  
import java.io.BufferedReader;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.net.HttpURLConnection;  
import java.net.URL;  
public class MainActivity extends AppCompatActivity {  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
 new AsynDataClass().execute();  
 }  
 public class AsynDataClass extends AsyncTask<String, Void, String> {  
 HttpURLConnection urlConnection;  
  
 @Override  
 protected String doInBackground(String... params) {  
  
 StringBuilder jsonResult = new StringBuilder();  
  
 try {  
 URL url = new  
 URL("http://www.papademas.net:81/cd\_catalog.json");  
 urlConnection = (HttpURLConnection)  
 url.openConnection();  
 InputStream in = new  
 BufferedInputStream(urlConnection.getInputStream());  
 BufferedReader reader = new BufferedReader(new  
 InputStreamReader(in));  
 String line;  
 while ((line = reader.readLine()) != null) {  
 jsonResult.append(line);  
 }  
 System.*out*.println("Returned Json url object " +  
 jsonResult.toString());  
 } catch (Exception e) {  
 System.*out*.println("Err: " + e);  
 } finally {  
 urlConnection.disconnect();  
 }  
 return jsonResult.toString();  
 }  
 @Override  
 protected void onPostExecute(String result) {  
  
 System.*out*.println("Result on post execute: " + result);  
 AlertDialog alertDialog = new

AlertDialog.Builder(MainActivity.this).create();  
 alertDialog.setTitle("Alert");  
 alertDialog.setMessage(result);  
 alertDialog.setButton(AlertDialog.*BUTTON\_NEUTRAL*, "OK",  
 new DialogInterface.OnClickListener() {  
 public void onClick(DialogInterface dialog, int which) {  
 dialog.dismiss();  
 }});  
 alertDialog.show();  
 }  
 }  
}