**Android Development**

1. **Methods used in opening a SQLite database:** Before using the methods, there are some steps to consider.
   1. **Import android.content.ContentValues:** The website developer.android.com explains in its article ContentValue that this class takes care of storing a set of values that the ContentResolver can process (ContentValue). In other words, this class allows using methods like put(String key, String value) to add values to keys.
   2. **Import android. database.Cursor:** The same website explains that "this interface provides random read-write access to the result set returned by a database query" (Cursor). This interface will help with reading and writing the information in the database.
   3. Import android. database.SQLException: developer.android.com explains that "[it is an] exception that indicates there was an error with SQL parsing or execution" (SQLException). This is important to know specifically what is causing an error.
   4. Import android. database.sqlite.\*: The website explains that this package "contains the SQLite database management classes that an application would use to manage its own private database" (android.database.sqlite). Developers need the methods included here to create the database.
   5. After Developers Include Everything Above: After all the above is included, they need a class that will extend the SQLiteOpenHelper class. In this class, developers need to incorporate private static final variables to store the name of the columns, the table's name, and the database version. Example:

package com.example.programfour;

import android.content.ContentValues;

import android.content.Context;

import android. database.Cursor;

import android. database.SQLException;

import android. database.sqlite.\*;

public class AddressDatabaseHelper extends SQLiteOpenHelper

{

public static final String TABLE\_ADDRESS = "address";

public static final String COLUMN\_ID = "\_id";

public static final String COLUMN\_FIRST\_NAME = "firstName";

public static final String COLUMN\_LAST\_NAME = "lastName";

public static final String COLUMN\_STREET = "street";

public static final String COLUMN\_CITY\_TOWN = "cityTown";

public static final String COLUMN\_STATE = "state";

public static final String COLUMN\_ZIP\_CODE = "zipCode";

private static final String DATABASE\_NAME = "address.db";

private static final int DATABASE\_VERSION = 1;

…..

}

After Developers Form the Constant Variables: After this, they need another constant String variable to create the SQL statements. This string follows the same rules as a normal java String variable. Example:

private static final String DATABASE\_CREATE\_SQL = "create table "

+ TABLE\_ADDRESS + "("

+ COLUMN\_ID + " integer primary key autoincrement, "

+ COLUMN\_FIRST\_NAME + " text not null, "

+ COLUMN\_LAST\_NAME + " text not null, "

+ COLUMN\_STREET + " text not null, "

+ COLUMN\_CITY\_TOWN + " text not null, "

+ COLUMN\_STATE + " text not null, "

+ COLUMN\_ZIP\_CODE + " text not null);";

How the Methods Work: To better explain this, first, I will explain how the app would create the database if it did not exist.

For this, developers must create a constructor to receive a Context object as its parameter. In this case, we will call it AddressDatabaseHelper. Once the object is created using the AddressDatabaseHelper constructor, developers can pass the Context object into the SQLiteOpenHelper. This object will also pass the database version and name into the SQLiteOpenHelper. The SQLiteOpenHelper will check first if the database exits or not. It will use the onCreate method to create the database if it does not. This method will receive a SQLiteDatabase object. This object is then used inside the onCreate() method to call another method named execSQL(). execSQL() receives the string we created before that contains the SQL statements we need to create the table. This method takes care of executing the statements. Example:

public AddressDatabaseHelper(Context context)

{ super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

@Override

public void onCreate(SQLiteDatabase database)

{ database.execSQL(DATABASE\_CREATE\_SQL);

}

Methods Used in Opening a SQLite Database: If the database already exists, Android will use the following methods:

a. onUpgrade() Method: This method belongs to the class SQLiteOpenHelper. If this class detects a new version of the database. It will call the onUpgrade() method. This method receives three parameters. First, a database object. Second, an int variable with the old database version, and last, another int variable with the new version number. Inside the method, developers need to use the execSQL() method again to execute a DROP TABLE IF EXISTS command. This will erase the old table. Once this is done, control will call the method onCreate(), creating the new table with all the updates. Example:

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)

{ // Check prior version to understand upgrade steps

// Export data

db.execSQL("DROP TABLE IF EXISTS " + TABLE\_ADDRESS);

onCreate(db);

// Import Data

}

Creating an array of strings with all the columns' names is also important. Example:

private String[] allColumns = { AddressDatabaseHelper.COLUMN\_ID,

AddressDatabaseHelper.COLUMN\_FIRST\_NAME, AddressDatabaseHelper.COLUMN\_LAST\_NAME,

AddressDatabaseHelper.COLUMN\_STREET, AddressDatabaseHelper.COLUMN\_CITY\_TOWN, AddressDatabaseHelper.COLUMN\_STATE,

AddressDatabaseHelper.COLUMN\_ZIP\_CODE};

open() throws SQLException Method: This method throws an exception if there is an error with SQL parsing or execution. This method instantiates an SQLiteDatabase object. In this object, we will store the information previously stored in the object AddressDatabaseHelper using the method getWritableDatabase(). developer.android.com says,

"[this method creates] and/or open a database that will be used for reading and writing. The first time this is called, the database will be opened and onCreate(SQLiteDatabase), onUpgrade(SQLiteDatabase, int, int) and/or onOpen(SQLiteDatabase) will be called" (SQLiteOpenHelper).

Example:

class AddressDataSource

{

//database fields

private SQLiteDatabase database;

private AddressDatabaseHelper dbHelper;

….

AddressDataSource(Context context) {

dbHelper = new AddressDatabaseHelper(context);

}

public void open() throws SQLException {

database = dbHelper.getWritableDatabase();

}

….

}

Close() Method: According to the video CIS240 Week 9 - Fall 2021 this method will close the database (01:16:33). Example:

public void close() {

dbHelper.close();

}

d. put() Method: This method belongs to the class ContentValues. In the following example, developers can use it to store a key per value of strings in the object ContentValues. This is the information control has collected so far from the AddressArrayAdapter class.

public long createAddress(AddressAttributeGroup address)

{ ContentValues values = new ContentValues();

values.put(AddressDatabaseHelper.COLUMN\_FIRST\_NAME, address.firstName);

values.put(AddressDatabaseHelper.COLUMN\_LAST\_NAME, address.lastName);

values.put(AddressDatabaseHelper.COLUMN\_STREET, address.street);

values.put(AddressDatabaseHelper.COLUMN\_CITY\_TOWN, address.cityTown);

values.put(AddressDatabaseHelper.COLUMN\_STATE, address.state);

values.put(AddressDatabaseHelper.COLUMN\_ZIP\_CODE, address.zipCode);

…….

}

e. insert() Method: This method belongs to the SQLiteDatabase class. developer.android.com mentions that "[this method is used] for inserting a row into the database" (SQLiteDatabase). In other words, this is what populates the table with information. It receives three parameters.

The table name.

A nullColumnHack string. Its value can be null. The article SQLiteDatabase says, "SQL doesn't allow inserting a completely empty row without naming at least one column name" (SQLiteDatabase). This means that empty rows cannot be inserted. If not set to null, the method will insert the word NULL.

A ContentValues object. The article SQLiteDatabase clarifies that "this map contains the initial column values for the row. The keys should be the column names and the values the column values" (SQLiteDatabase).

According to the same article, this method returns "the row ID of the newly inserted row, or -1 if an error occurred" (SQLiteDatabase). So, this method returns a long data type. Example:

public long createAddress(AddressAttributeGroup address)

{

ContentValues values = new ContentValues();

……

long insertId = database.insert(AddressDatabaseHelper.TABLE\_ADDRESS,

null,

values);

return insertId;

}

delete() Method: The article I have been mentioning says that it is a "[convenience] method for deleting rows in the database" (SQLiteDatabase). In other words, it is the opposite of the method insert(). This method accepts three arguments as well:

The table's name.

A whereClause string. This is optional, but if the null value is passed to it, it will erase all the rows from the table.

A whereArgs string. The article mentions that "[you] may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings" (SQLiteDatabase).

Example:

public void deleteAddress(AddressAttributeGroup address)

{ long id = address.id;

database.delete(AddressDatabaseHelper.TABLE\_ADDRESS,

AddressDatabaseHelper.COLUMN\_ID + " = " + id, null);

}

query() Method: The same article explains that "[it queries] the given URL, returning a cursor over the result set" (SQLiteDatabase). Therefore, this method receives several parameters. In the example below, the following arguments are used:

Table's name.

An array of strings, including the name of the columns.

A set of strings like selection, selectionArgs, groupBy, having, and orderBy. These strings will allow us to pass statements that will make our row selection more specific.

The information returned by the method will get stored in a Cursor object. Example:

public AddressCollectionDB getAllAddresses() throws Exception

{ AddressCollectionDB addresses = new AddressCollectionDB();

Cursor cursor = database.query(AddressDatabaseHelper.TABLE\_ADDRESS,

allColumns, selection: null, selectionArgs: null, groupBy: null, having: null, orderBy: null);

…

}

moveToFirst() method: It belongs to the interface cursor and according to developer.android.com "[it moves] the cursor to the first row. This method will return false if the cursor is empty" (Cursor). In other words, once all the rows get saved in the Cursor object, we can use this method to point to the first row.

isAfterLast() Method: The same webpage mentions that "[it returns] whether the cursor is pointing to the position after the last row" (Cursor). If it does, it returns true. We can use this method with a while loop to close the cursor later after we go through all the rows.

moveToNext() Method: If the isAfterLast() method does not return true, the method moveToNext() moves the cursor to the next row.

cursor.close() Method: If the isAfterLast() returns true, control will exit the while loop and close the cursor. Example:

public AddressCollectionDB getAllAddresses() throws Exception

{

….

cursor.moveToFirst();

while (!cursor.isAfterLast())

{ addresses.addAddress(cursorToAddressAttributeGroup(cursor));

cursor.moveToNext();

}

cursor.close();

return addresses;

}

cursor.getInt() Method: The article Cursor explains that "[it returns] the value of the requested column as an int" (Cursor). It requires the column index as a parameter.

cursor.getColumnIndex() Method: The same article says, "[it returns] the zero-based index for the given column name, or -1 if the column doesn't exist" (Cursor). It needs the name of the column as a string parameter.

cursor.getString() Method: It works similarly to the method getInt(). The difference is that the method getString() returns a string. Example:

private AddressAttributeGroup cursorToAddressAttributeGroup(Cursor cursor)

{ AddressAttributeGroup address = new AddressAttributeGroup(cursor.getInt(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_ID)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_FIRST\_NAME)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_LAST\_NAME)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_STREET)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_CITY\_TOWN)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_STATE)),

cursor.getString(cursor.getColumnIndex(AddressDatabaseHelper.COLUMN\_ZIP\_CODE)));

return address;

}

Comparison and Contrast of Threads and the Use of App Services.

Why and How They Are Different: First, developers need to understand that a tread is a task that runs within the app and finishes once done. According to developer.android.com,

"the Java Virtual Machine continues to execute threads until either of the following occurs:

The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.

All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method" (Thread).

The same website describes "[a service as] an application component that can perform long-running operations in the background…. Once started, a service might continue running for some time, even after the user switches to another application" (Services overview). So, threads end when all the tasks are performed, but a service does not, and it can continue going in the background.

b. Comparison and Contrast between Threads and App Services: We can use multithreading within an app to perform several tasks simultaneously. This is an excellent technique to improve app performance. developer.android.com clarifies that

"[by using the class ThreadPoolExecutor the process can become] easier. This class manages the creation of a group of threads, sets their priorities, and manages how work is distributed among those threads. As workload increases or decreases, the class spins up or destroys more threads to adjust to the workload" (Better performance through threading).

In other words, multithreading can still become a problem if we run many tasks simultaneously, but the class ThreadPoolExecutor can help with this problem.

One of the main differences between services and thread is that services have three different types:

Foreground: The Android developer website explains that "[a] foreground service performs some operation that is noticeable to the user" (Services overview). This means it continues working even when the user is not interacting with the app.

Background: This type of service is not noticeable to the user

Bound: The Android developer's website indicates that "[a] service is bound when an application component binds to it by calling bindService()…. [it] runs only as long as another application component is bound to it" (Services overview). Consequently, it is possible to bind several components, but the bound service will get destroyed once all of them are unbound.

So, when can we use a service, and when a thread? According to the webpage, we should only use a service if we intend to run a component in the background (Services overview). In other words, use threads in all the other cases.

What App Services Are and Usage of Different Types of Service Implementation:

a. App Services, Usages, and Different Types of Service Implementation: I already explained this in question two. However, here, I will go into more detail about some aspects. According to week eleven's explanation, "services can perform inter-process communications" (Services Overview). This means it can interact with contact providers, do network transactions, and more in the background.

There are two ways to use them, and the following image taken from the android developer's website, from one of its articles called Services overview, explains it well.

Differences between Bound and Started Services and Rules:

Differences:

A started service can run indefinitely in the background even when the component that started it gets destroyed. A bound service only runs until all its components unbind from the service.

Started services usually do not return a result. Conversely, bound services can be used to send requests, get results, and communicate with other processes.

Developers need to use the method startService() to start a started service. For bound ones, they need to use the method bindService().

Typically started services are good at performing operations. Bound services can offer more than that. They can provide a client-server interface and allow components to interact with it.

Rules: The article Service Basics from week eleven explains the rules well. There are three essential methods to consider. First is the startService(). When a component calls this method, the service will keep running until control gets to the stopSelf() method, which is the second vital method to mention. This method stops the service automatically. Another way to stop the service is when a component calls the method stopService(). According to developer.android.com,

"[the service will note stop] if a stopped service still has ServiceConnection objects bound to it with the BIND\_AUTO\_CREATE set[. The service] will not be destroyed until all of these bindings are removed" (Context).

If a component calls the bindService() method. The bound service will only run until the component unbinds. After this, it gets destroyed.

If memory is low, the android system will stop a service only in this case.

The service is less likely to be killed if a user has focus.

If the service runs in the foreground, it is even less likely to be killed.

The longer a service runs, the more susceptible it is to being killed by the Android system. This means that it will lose its hierarchy compared to other background services.

If a component starts a service, developers must design it so the Android system will restart smoothly.

If the system kills the service, it will restart once resources are available. This is based on the return value of the method onStartCommand().

How to Use the Android Device's Camera:

To use the Android device's camera, we need to follow the next steps:

First, developers must create an ImageView and Button widget in the layout following all the standard steps.

Next, they need to import the classes Android.widget.ImageView, Android.graphics.Bitmap, android.graphics.BitmapFactory, java.io.ByteArrayOutputStream, java.io.File, and java.io.FileOutputStream into the activity that will take care of accessing the ImageView and Button widgets by its ID.

In this class, the developers must create an int constant variable to help identify the image request.

In the same class, they need to include an ImageView, Button, and String variables.

In the onCreate() method, they need to use the findViewById() method to find the widgets ImageView and Button in the layout and store the view information in the variables that were created in part d.

The developer then needs to use the setOnClickListener() method for the button.

In the same method, they also need to store the image path in the string variable we created in part d.

At the end of the onCreate() method. They need to use a method that allows control to load the image. I will explain this method later. Example:

import android.graphics.Bitmap;

import android.graphics.BitmapFactory;

….

import android.widget.ImageView;

….

public class AddressEntry extends Activity implements View.OnClickListener

{

int CAMERA\_PIC\_REQUEST = 2;

….

ImageView imagePhoto;

….

String imagePath = "";

…..

@Override

public void onCreate(Bundle savedInstanceState)

{ super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_address\_entry);

….

imagePhoto = (ImageView) findViewById(R.id.imagePhoto);

….

cmdPhoto = (Button) findViewById(R.id.cmdPhoto);

cmdPhoto.setOnClickListener(this);

….

imagePath = stIntent.image;

….

loadImage();

}

i. The loadImage() method mentioned in part h first checks if there is a path for the picture. If there is one, this method creates a Bitmap variable and a BitmapFactory.Options object. According to developer.android.com, "[the BitmapFactory.Options object creates] a default Options object, which if left unchanged will give the same result from the decoder as if null were passed" (BitmapFactory.Options). In other words, we will use later this empty object to subsample the original image. The website also says that "if [the field called inSampleSize is] set to a value > 1, [it will] requests the decoder to subsample the original image, returning a smaller image to save memory" (BitmapFactory.Options). This means that it will return a smaller version of the original picture, both in size and pixels, that can be used to save in memory. Next, we use the decodeFile() method. developer.android.com says that this method "[decodes] a file path into a bitmap. If the specified file name is null, or cannot be decoded into a bitmap, the function returns null" (BitmapFactory.Options). Therefore, this method receives two parameters. The first one is a string that contains the image's path. For this, a method is used that I will explain in the next section. The second argument is a BitmapFactory.Options object with the image resized down to 1/3 in height and width. This image is later stored in the Bitmap variable we created initially. Finally, control goes into the method setImageBitmap(). For this method to work, the developer must pass the Bitmap variable with the resized image as a parameter. The android developer's webpage supports that "[the setImageBitmap() method] sets a Bitmap as the content of this ImageView" (ImageView). In other words, the ImageView object mentioned in part 4.a. This is what finally loads the image into the user's screen. Example:

void loadImage()

{ if (imagePath == null || imagePath.length() == 0)

return;

Bitmap imageBitmap;

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 3;

imageBitmap = BitmapFactory.decodeFile(getImagePathName(), options);

imagePhoto.setImageBitmap(imageBitmap);

}

j. As I mentioned in part 4.i the decodeFile() method needs the image path. To create this path, developers can use a method that returns the path as a string. In this case, the method is called getImagePathName(). This method has a string variable that control will access later. Next, developer need to use a File method called getDataDir(). The webpage says that "it returns the absolute path to the directory on the filesystem where all private files belonging to this app are stored" (ContextWrapper). In our case, it will return the image's absolute path. Control will concatenate the absolute path with the rest of the path stored in the string variable we created in part 4.d. This information will be stored in the string created at the beginning of the method we are describing. Example:

String getImagePathName()

{ String rcode="";

//rcode = this.getCacheDir()+"/"+imagePath;

//rcode = this.getExternalFilesDir("")+"/"+imagePath;

rcode = this.getDataDir()+"/"+imagePath;

return rcode;

}

k. When the user presses the button cmdPhoto, control creates an Intetn object. This object will receive the parameter "android.provider.MediaStore.ACTION\_IMAGE\_CAPTURE." developer.android.com explains that this parameter is a "[standard] Intent action that can be sent to have the camera application capture an image and return it" (MediaStore). As a result, the user can use the phone's camera. Next, control will go into the method System.currentTimeMillis(). The Android's webpage clarifies that "[it returns] the current time in milliseconds" (System). This information will be part of the image path that will be included in a string. This information gets stored next in the string variable we created in point 4.d. Next, developers must pass the Intent and the constant mentioned in part 4.c to the method startActivityForResult(). developer.android.com explains that this method is suitable when developers need to get a result back from an activity when it ends. The second parameter should be an integer identifying the call. The result will come back through the method onActivityResult(int, int, Intent) (Activity). This means that the information will be sent to the method onActivityResult(), where control will use it to perform more operations. Example:

public void onClick(View v)

{

….

if(cmdPhoto.getId() == v.getId())

{ Intent cameraIntent = new Intent(android.provider.MediaStore.ACTION\_IMAGE\_CAPTURE);

imagePath = "Address" + System.currentTimeMillis()+".png";

startActivityForResult(cameraIntent, CAMERA\_PIC\_REQUEST);

}

….

}

Once the method onActivityResult() receives the information, control will compare the int constant variable mentioned in part 4.c with the result from the method startActivityForResult(). If they are a match and the activity result is RESULT\_OK, control will extract the data from the intent and store it in a Bitmap variable. Subsequently, control will create a File object with the image path. The android developer's webpage says that the method createNewFile() will "atomically [create] a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist" (File). Thus, control will name the file, if it does not exist, with the image path developers used as a parameter for the File object.

The next step will be to create a ByteArrayOutputStream object. The same webpage says that "this class implements an output stream in which the data is written into a byte array" (ByteArrayOutputStream). Consequently, control will use this object to create an array with all the images. After this, control will enter the method compress(). developer.android.com clarifies that "[this method writes] a compressed version of the bitmap to the specified outputstream. If this returns true, the bitmap can be reconstructed by passing a corresponding inputstream to BitmapFactory.decodeStream()" (Bitmap). In other words, control will compress the image we extracted from the intent following the specifications passed to the method's parameters. After control has compressed the image, control will continue with the method toByteArray (). According to the Android's website, "[it will create] a newly allocated byte array. Its size is the current size of this output stream and the valid contents of the buffer have been copied into it" (ByteArrayOutputStream). This means this method will extract the compressed image or images from the object ByteArrayOutputStream and store the data in a new byte array.

Control then will create a new FileOutputStream object with the file it created using the method createNewFile(). Android's page explains that "[this class creates]…. an output stream for writing data to a File or to a FileDescriptor…. FileOutputStream is meant for writing streams of raw bytes such as image data" (FileOutputStream). So, control will use this object to write the image data we have so far. For this, it will use the method write(), which according to the android developer's page, "[it will write] b.length bytes from the specified byte array to this file output stream" (FileOutputStream). In this case, the compresses Bitmap data. Next, the method flush(), according to the webpage, "[flushes this stream by writing any buffered output to the underlying stream" (Flushable). This means that any previous data will be saved in its corresponding file. Contol then goes to the method close(). developer.android.com describes that "[it closes] this file output stream and releases any system resources associated with this stream. This file output stream may no longer be used for writing bytes" (FileOutputStream). As a result, the file output stream will close. After this, control will load the image using the method loadImage() already explained in point 4.i. If there is an exception or the int constant variable mentioned in part 4.c does not match the result from the method startActivityForResult(), control will prompt the user with the message "Picture not taken." Example:

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data)

{

super.onActivityResult(requestCode, resultCode, data);

if( requestCode == CAMERA\_PIC\_REQUEST)

{ if (resultCode == Activity.RESULT\_OK)

{ try

{ Bitmap imageBitmap= (Bitmap) data.getExtras().get("data");

File file = new File(getImagePathName());

file.createNewFile();

try

{ ByteArrayOutputStream bos = new ByteArrayOutputStream();

imageBitmap.compress(Bitmap.CompressFormat.PNG,100,bos);

byte[] bitmapdata = bos.toByteArray();

FileOutputStream fos = new FileOutputStream(file);

fos.write(bitmapdata);

fos.flush();

fos.close();

}

catch (IOException e)

{ e.printStackTrace();

}

loadImage();

}

catch (Exception e)

{ showInfo("Picture not taken");

e.printStackTrace();

imagePath = "";

}

}

}

else

{ showInfo("Picture not taken");

}

}

}

What Are Web Services and How They Could Enhance an Android App?

a. Web Services: They are functions that we expose on a web server that provides the logic or capabilities we are looking to perform. This indicates that the app will send a function to a web server, and the server will process the information. This information can travel through the network or a private network system owned by a company. It uses the HTTP protocol, but not all web services operate similarly. You can call a web service, and everything that the web service needs to satisfy that request should be provided in the request. This implies that developers do not need to create several functions to get a result from the server if the developer includes all the parameters correctly and the identifier. The same video says that web services support one or two formats to deliver their data. In other words, they use XML or JSON, although there are exceptions. In most cases, it is also essential to have a license to use a web service. Because of this, developers need to be aware of all the terms and conditions.

How Web Services Could Enhance an Android App:

They allow the app to access the cloud. This helps an app to integrate with other systems.

It allows companies to include logic in their server and share it with their apps using web services.

Web services improve the customer experience, businesses' systems, users' entertainment, and more.

Process to Ready and Release an Android App to the Marketplace.

The process to ready and release an Android app to the marketplace is as follows:

Create a release-ready package that users can install.

The release-ready package must contain the same components as the debug .apk file, compiled source code, resources, and manifest file. The package must be built using the same build tools.

The developers' apk files must be signed with their certificate, and they must optimize it with the zipalign tool. Using the Eclipse Export Wizard to compile, sign, and optimize an application is much easier.

Developers should consider the following four steps to ready their application:

Gather All materials and resources.

Cryptographic keys: Developers need to sign their application; this tells who designed the app. The cryptographic key needs to expire by 22 October 2033. Developers may need other private keys if they use services, for example.

Application Icon: Icons are necessary so users can quickly identify a developer's application on their device. Icons also need to follow specific guidelines. In this case, getting a graphic designer is a good idea.

End-user License Agreement: This part protects developers. It is a good idea to pay a qualified person to help with this.

Miscellaneous Materials: This part refers to marketing and publicity strategies.

ii. Configure and Build the Application for Release: A good package name is essential since it cannot be changed after the app is released. Deleting an app from Google Play erases all the users. Another important step before releasing the app is to disable login and debugging options. This also includes log files or static test files created during the project. Developers should also remove tracing calls added to the code, such as startMethodTracing() and stopMethodTracing() method calls. It is also important to clean the project directories. Other aspects to keep in mind are:

Compatibility Issues: It is crucial to test the app on multiple screen configurations, including tablets and old versions.

Update URLs for Servers and Services: Developers need to be sure they are not using a test URL or path.

Implement Licensing When Using Google Play: This is important if developers are planning to charge users. Google Play will take care that they receive their payment.

Build the Application for Release: In other words, build the app into a release-ready apk file that is signed and optimized. As mentioned before, it is better to use Eclipse Export Wizard.

iii. Prepare External Services: Developers must ensure that remote servers or services are secured, up-to-date, production-ready, and licensed.

iv. Test the Application: Developers need to be sure their app works well on actual devices by testing the user interface elements, the application's performance, and battery efficiency.

The Value of Releasing an Android App to Google Play.

The advantages of releasing an Android app to Google Play are:

Google Play is more secure when it comes to transactions. If developers want to monetize their apps, Google Play will take care that they are getting paid.

Google Play also provides excellent visibility for the developers' app.

Google Play allows developers to target a specific audience.

Works Cited

Activity, https://developer.android.com/reference/android/app/Activity#startActivityForResult(android.content.Intent,%20int). 25 Aug. 2022.

android. database.sqlite, https://developer.android.com/reference/android/database/sqlite/package-summary. 10 Feb. 2022

Better performance through threading, https://developer.android.com/topic/performance/threads. 10 Nov. 2021

Bitmap, https://developer.android.com/reference/android/graphics/Bitmap. 17 Mar. 2022

BitmapFactory.Options, https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html#public-constructors. 10 Feb 2022

ByteArrayOutputStream, https://developer.android.com/reference/java/io/ByteArrayOutputStream. 08 Jun. 2022

ContentValues, https://developer.android.com/reference/android/content/ContentValues. 10 Feb 2022

Context, https://developer.android.com/reference/android/content/Context#stopService(android.content.Intent). 11 Oct. 2022

ContextWrapper, https://developer.android.com/reference/android/content/ContextWrapper#getDataDir(). 13 Jul. 2022

Cursor, https://developer.android.com/reference/android/database/Cursor#moveToFirst(). 10 Feb. 2022

File, https://developer.android.com/reference/java/io/File. 18 Feb. 2021

FileOutputStream, https://developer.android.com/reference/java/io/FileOutputStream. 08 Jun.2022

Flushable, https://developer.android.com/reference/java/io/Flushable#flush(). 10 Feb. 2022.

ImageView, https://developer.android.com/reference/android/widget/ImageView#setImageBitmap(Android.graphics.Bitmap). 08 Jun. 2022.

MediaStore, https://developer.android.com/reference/android/provider/MediaStore#ACTION\_IMAGE\_CAPTURE. 08 Jun. 2022.

Services overview, https://developer.android.com/guide/components/services. 31 Aug. 2022

SQLException, https://developer.android.com/reference/android/database/SQLException. 10 Feb. 2022

SQLiteDatabase, https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase#insert(java.lang.String,%20java.lang.String,%20android.content.ContentValues) 08 Jun. 2022.

SQLiteOpenHelper, https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper#getWritableDatabase().10 Feb. 2022

System, https://developer.android.com/reference/java/lang/System#currentTimeMillis(). 08 Jun. 2022.

Thread, https://developer.android.com/reference/java/lang/Thread. 08 Jun. 2022