

CHAPTER – 7

PACKAGING AND MONETIZING

DATA MANAGEMENT:

Android provides several options to save persistent application data. The solution we choose depends on our specific needs, such as whether the data should be private to our application or accessible to other applications (and the user) and how much space our data requires.

Data storage options are the following:

1. **Shared Preferences:** Store private primitive data in key-value pairs.
2. **Internal Storage:** Store private data on the device memory.
3. **External Storage:** Store public data on the shared external storage.
4. **SQLite Databases:** Store structured data in a private database.
5. **Network Connection:** Store data on the web with our own network server.

Android provides a way for us to expose even our private data to other applications with a content provider. A content provider is an optional component that exposes read/write access to our application data, subject to whatever restrictions we want to impose.

1. Shared Preferences:

It is the simplest data storing option available in android. It stores data in key value pairs for example:

Key	Value
Username	Bishal
Password	Bishal_009
Location	Lakeside

Data are stored in XML file in the directory as Data/data/<package-name>/shared-prefs folder. Shared Preference support primitive data types such as Booleans, floats, int, longs and strings. This data will persist across user sessions (even if our application is killed)

To get a SharedPreferences object for our application, we have to use one of two methods:

1. **getSharedPreferences():** We use this if we need multiple preferences files identified by name, which we specify with the first parameter.
2. **getPreferences():** We use this if we need only one preferences file for our Activity. Because this will be the only preferences file for our Activity, we don't supply a name.

Operating Modes for Shared Preferences Are:

- ❖ **Use 0 or MODE_PRIVATE:** Only our application can access the file
- ❖ **MODE_WORLD_READBLE:** All apps can read the file

- ❖ **MODE_WORLD_WRITEABLE:** All apps can write the file
- ❖ **MODE_MULTI_PROCESS:** Multi process can modify the same shared-prefs file

Uses of Shared Preferences:

- ❖ Check whether the user is using our app
- ❖ Check when our app was last updated
- ❖ Remember user credentials
- ❖ Remember user settings
- ❖ Location caching

How to Use Shared Preference to Store Data?

1. **Get A Reference to A Shared Preference Object:**
 - For a single file, call `getPreferences(int mode)`
 - For several file, call `getSharedPreferences(String name, int mode)`
2. **Call the Editor:**
 - `SharedPreferences.Editor editor = sharedPreferences.edit()`
3. **Use The Editor to Add the Data with The Key:**
 - `editor.putString("name", "Bishal");`
 - `editor.putString("Password", "Bishal_009");`
4. **Commit Editor Changes:**
 - `editor.commit()`

How to Use Shared Preferences to Retrieve Data?

1. **Get A Reference to A Shared Preference Object:**
 - For a single file, call `getPreferences(int mode)`
 - For several file, call `getSharedPreferences(String name, int mode)`
2. **Use The Key Provided Earlier to Get Data:**
3. **Supply Default Value If Data Is Not Found:**
 - `String name=sharedPreferences.getString("name", "N/A");`
 - `String password=sharedPreferences.getString("name", "N/A");`

2. Internal Storage:

We can save files directly on the device's internal storage. By default, files saved to the internal storage are private to our application and other applications cannot access them (nor can the user). When the user uninstalls our application, these files are removed.

To Create and Write a Private File to The Internal Storage:

- ❖ Call `openFileOutput()` with the name of the file and the operating mode. This returns a `FileOutputStream`.
- ❖ Write to the file with `write()`.

- ❖ Close the stream with close().

To Read a File from Internal Storage:

- ❖ Call `openFileInput()` and pass it the name of the file to read. This returns a `FileInputStream`.
- ❖ Read bytes from the file with `read()`.
- ❖ Then close the stream with `close()`.

3. External Storage:

Every Android compatible device supports a shared "external storage" that we can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

In order to read or write files on the external storage, our app must acquire the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` system permissions.

For Example:

```
<manifest ...>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
...
</manifest>
```

If we need to both read and write files, then we need to request only the `WRITE_EXTERNAL_STORAGE` permission, because it implicitly requires read access as well.

4. SQLite Database:

SQLite is an open source embedded database. The original implementation was designed by D. Richard Hipp. Hipp was designing software used on board guided missile systems and thus had limited resources to work with. The resulting design goals of SQLite were to allow the program to be operated without a database installation or administration.

In android we use Sqlite database. Because it is inbuilt database in Android SDK more over it is lite weighted relation database suitable for Mobile Devices. We need not to load any drivers and we do not need to install the SQLite separately. The queries are also simple to understand and easy to implement.

Features:

- 1. Application File Format:** Transactions guarantee ACID [Atomicity, Consistency, Isolation, Durability] even after system crashes and power failures.
- 2. Temporary Data Analysis:** Command line client, import CSV files and use sql to analyze & generate reports.
- 3. Embedded Devices:** Applicable to small, reliable and portable like mobiles.
- 4. Portable:** Uses only ANSI-standard C and VFS, file format is cross platform (little vs. big endian, 32 vs. 64 bit)

5. **Reliable:** It has 100% test coverage, open source code and bug database, transactions are ACID even if power fails.
6. **Small:** It has only 300 kb library, runs in 16kb stack and 100kb heap.
7. **Single Database File:** An SQLite database is a single ordinary disk file that can be located anywhere in the directory hierarchy.
8. **Readable Source Code:** The source code to SQLite is designed to be readable and accessible to the average programmer.

Disadvantage:

1. **High Concurrency:** We have to handle reader/writer locks on the entire file.
2. **Huge Datasets:** Database file can't exceed file system limit or 2TB.
3. **Access Control:** We don't have any user interface to operate SQLite database objects as in MYSQL / SQL Server /Oracle. All the objects are virtual. However, there are few third party UI are available in the market.

How to Create a Database?

1. Create a new java class called **DataBaseHelper**:
2. Extend the class with **SQLiteOpenHelper**:
Public class **DataBaseHelper** extends **SQLiteOpenHelper** {}
3. Implement the methods:

```

public class DataBaseHelper extends SQLiteOpenHelper{
    @Override
    public void onCreate(SQLiteDatabase db){
    }
    @Override
    public void onUpgrade(SQLiteDatabase db,int oldVersion,int
    newVersion){
    }
}

```
4. Create the default constructor:

```

public class DataBaseHelper extends SQLiteOpenHelper {
    public DataBaseHelper (Context context, String name,
    SQLiteDatabase.CursorFactory, int version){
        super(context, name, factory, version);
        @Override
        public void onCreate(SQLiteDatabase db){
        }
        @Override
        public void onUpgrade(SQLiteDatabase db,int oldVersion,int
        newVersion){
        }
    }
}

```

5. Define name for Database and Table.

```
public static final String DATABASE_NAME="";
String TABLE_NAME="";
```
6. Define name for columns:

```
public static final String COL_1="ID";
public static final String COL_2="NAME";
```
7. Modify the constructor to take only Context.
Provide the database name and version to the superclass

```
public DataBaseHelper(Context context){
    super(context, DATABASE_NAME, null, 1);
}
```
8. In onCreate method write a Query to create a table

```
public void onCreate(SQLiteDatabase db){
    db.execSQL("CREATE"+DATABASE_NAME+ "(ID INTEGER PRIMARY KEY
    AUTO INCREMENT,NAME TEXT)
}
```
9. In onUpgrade method write a query to drop the table if exists.

```
public void onUpgrade(SQLiteDatabase db,int oldVersion,int newVersion){
    db.execSQL("DROP TABLE IF EXISTS"+TABLE_NAME);
}
```
10. To create database use getWritableDatabase() method in our constructor

```
public DataBaseHelper(Context context){
    super(context, DATABASE_NAME, null, 1);
    SQLiteDatabase db=this.getWritableDatabase();
}
```
11. Create an object of DataBaseHelper class in MainActivity and pass Context to it

```
DataBaseHelper myDb=new DataBaseHelper(this);
```

THE CONTENT PROVIDER CLASS:

Introduction:

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

When we want to access data in a content provider, we use the ContentResolver object in our application's Context to communicate with the provider as a client. The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider. The provider object receives data requests from clients, performs the requested action, and returns the results.

We don't need to develop our own provider if we don't intend to share our data with other applications. However, we need our own provider to provide custom search suggestions in our own application. We also need our own provider if we want to copy and paste complex data or files from our application to other applications.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. We can see some of them listed in the reference documentation for the `android.provider` package. With some restrictions, these providers are accessible to any Android application.

Content Provider Basics:

A content provider manages access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data. However, content providers are primarily intended to be 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 **Inter Process Communication** and secure data access.

Accessing a Provider:

An application accesses the data from a content provider with a `ContentResolver` client object. This object has methods that call identically-named methods in the provider object, an instance of one of the concrete subclasses of `ContentProvider`. The `ContentResolver` methods provide the basic "CRUD" (Create, Retrieve, Update, And Delete) functions of persistent storage.

The `ContentResolver` object in the client application's process and the `ContentProvider` object in the application that owns the provider automatically handle inter-process communication. `ContentProvider` also acts as an abstraction layer between its repository of data and the external appearance of data as tables.

Note: To access a provider, our application usually has to request specific permissions in its manifest file.

Requesting Read Access Permission:

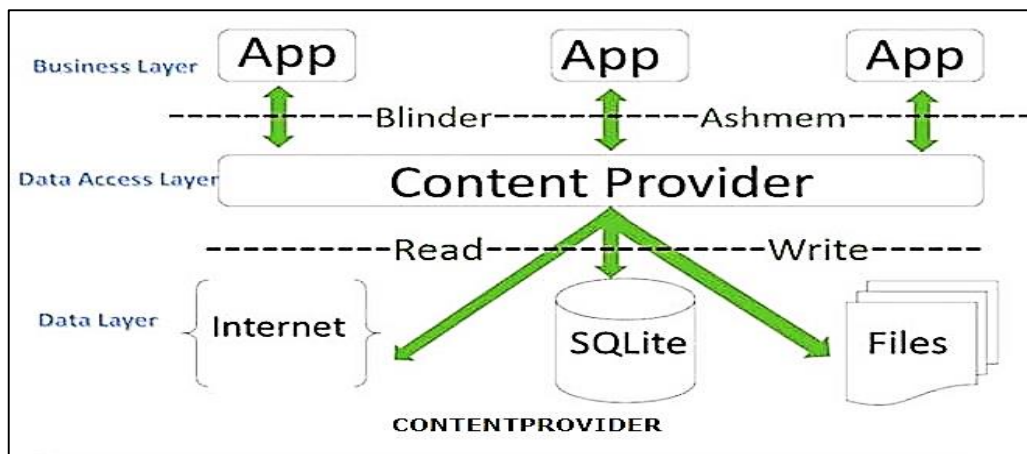
To retrieve data from a provider, our application needs "read access permission" for the provider. We can't request this permission at run-time; instead, we have to specify that we need this permission in our manifest, using the `<uses-permission>` element and the exact permission name defined by the provider.

When we specify this element in our manifest, we are in effect "requesting" this permission for our application. When users install our application, they implicitly grant this request.

Primary Methods:

- ❖ `onCreate()` which is called to initialize the provider
- ❖ `query(Uri, String[], Bundle, CancellationSignal)` which returns data to the caller
- ❖ `insert(Uri, ContentValues)` which inserts new data into the content provider

- ❖ `update(Uri, ContentValues, String, String[])` which updates existing data in the content provider
- ❖ `delete(Uri, String, String[])` which deletes data from the content provider
- ❖ `getType(Uri)` which returns the MIME type of data in the content provider



THE SERVICE CLASS:

Application Building Block:

Activity	<ul style="list-style-type: none"> • UI Component Typically Corresponding to one screen.
IntentReceiver	<ul style="list-style-type: none"> • Responds to notifications or status changes. Can wake up your process.
Service	<ul style="list-style-type: none"> • Faceless task that runs in the background.
ContentProvider	<ul style="list-style-type: none"> • Enable applications to share data.

Android Application Anatomy:

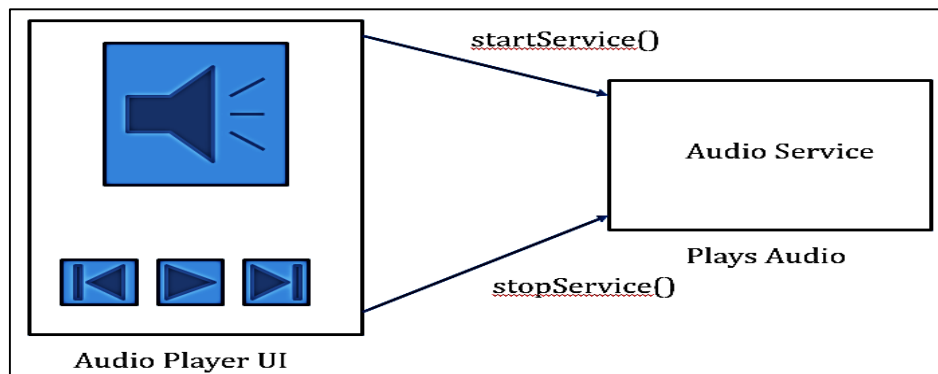


What is a Service?

Services are codes that run **in the background**. They can be **started** and **stopped**. Services **doesn't have UI**.

Features:

- ❖ to tell the system *about* something **it wants to be doing in the background** (even when the user is not directly interacting with the application)
- ❖ to calls to **Context.startService()**, which ask the system **to schedule work for the service**, to be run until the service or someone else explicitly stop it.



What a Service is not?

There are some **confusions**:

- ❖ A Service is **not a separate process**. The Service object itself does not imply it is running in its own process; unless otherwise specified, it runs in the same process as the application it is part of.
- ❖ A Service is **not a thread**. It is not a means itself to do work off of the main thread (**to avoid Application Not Responding errors**).

Service Example:

We'll create a simple **ServiceDemo** application which runs **in background** and shows notification in the upper **Notification Bar** with a **period of specified time**.

1. We Will Create a Project With Following:

Project Name: **ServiceDemo**

Build Target: **1.6**

Application name: **ServiceDemo**

Package name: **com.basistraining.servicedemo**

Create Activity: **ServiceDemoActivity**

Min SDK Version: **4**

New Android Project
Creates a new Android Project resource.

Project name:

Contents

☒ Create new project in workspace
☐ Create project from existing source
☒ Use default location

Location:

☐ Create project from existing sample

Samples:

Build Target

Target Name	Vendor	Platform	API ...
<input type="checkbox"/> Android 1.5	Android Open Source Project	1.5	3
<input type="checkbox"/> Google APIs	Google Inc.	1.5	3
<input checked="" type="checkbox"/> Android 1.6	Android Open Source Project	1.6	4

Standard Android platform 1.6

Properties

Application name:

Package name:

☒ Create Activity:

Min SDK Version:

2. We'll Add a New Class MyService That Extends Service. We Get the Following.

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {

        return null;
    }

}
```

3. We'll also need to add the Service in AndroidManifest.xml

```
<service android:name="MyService"></service>
```

4. Now we add other lifecycle methods of the MyService:

```
public class MyService extends Service {

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

}
```

```

public class MyService extends Service {

    @Override
    public void onCreate() {
        super.onCreate();
        Log.d("onCreate()", "Service Created");
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
        Log.d("onStart()", "Service Started");
    }

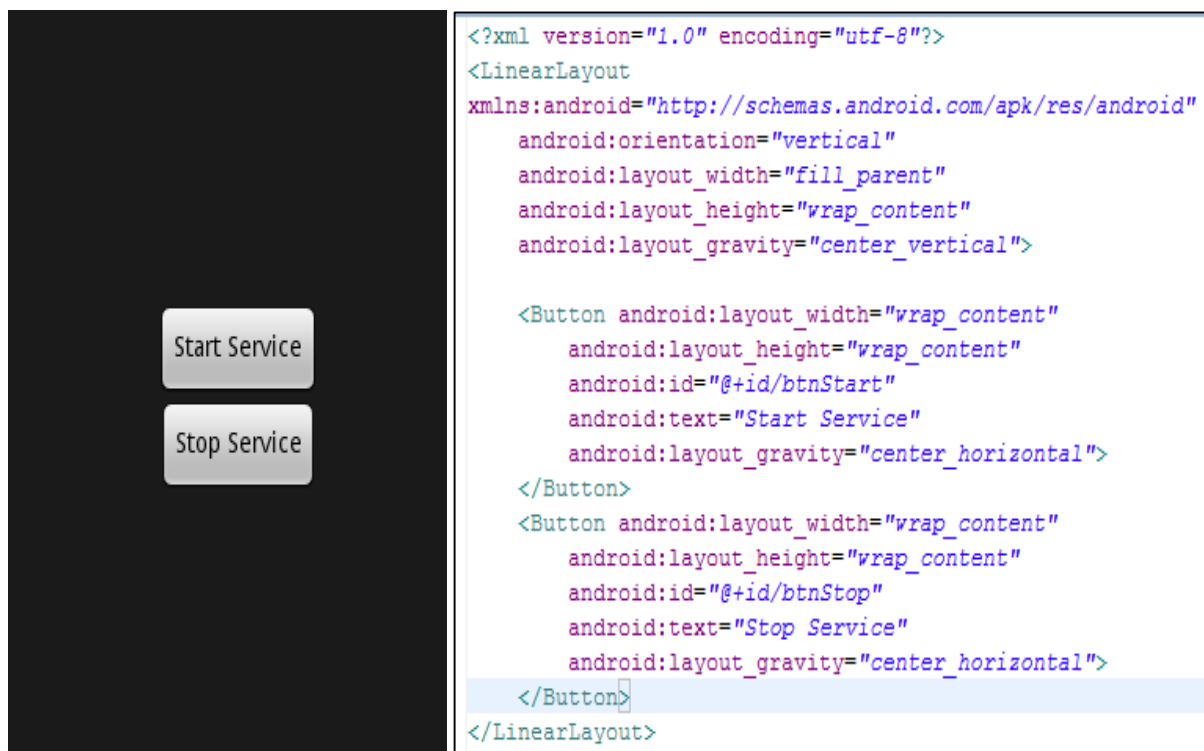
    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("onDestroy()", "Service Destroyed");
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

}

```

Now let's make the Layout *res/layout/main.xml* to have 2 buttons to start and stop the Service>



There are only 2 buttons with id `"@+id/btnStart"` and `"@+id/btnStop"`

Now we add action to our Buttons to **Start** or **Stop** the **MyService** and the Application in our **onCreate()** method of the Activity.

```

btnStart = (Button) findViewById(R.id.btnStart);
btnStop = (Button) findViewById(R.id.btnStop);

btnStart.setOnClickListener(new View.OnClickListener() {

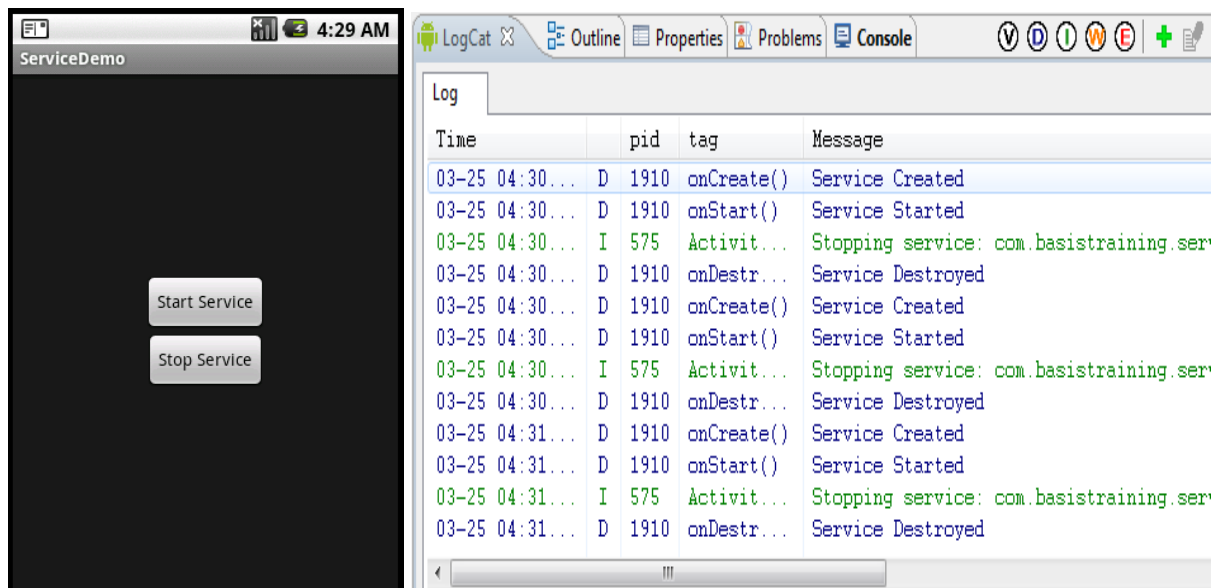
    @Override
    public void onClick(View v) {
        serviceIntent = new Intent(ServiceDemoActivity.this,
            MyService.class);
        startService(serviceIntent);
    }
});

btnStop.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        serviceIntent = new Intent(ServiceDemoActivity.this,
            MyService.class);
        stopService(serviceIntent);
    }
});

```

If we run the app and test, we'll see our buttons are **starting and stopping** the service in **LogCat**.



Now to do something on **Starting of our Service**, we do following:

```

TimerTask notifyTask;
Timer timer = new Timer();

@Override
public void onStart(Intent intent, int startId) {
    super.onStart(intent, startId);
    Log.d("onStart()", "Service Started");
    notifyTask = new TimerTask() {
        int i = 0;

        public void run() {
            i++;
            Log.d("Service Running", "Value of i=" + i);
        }
    };

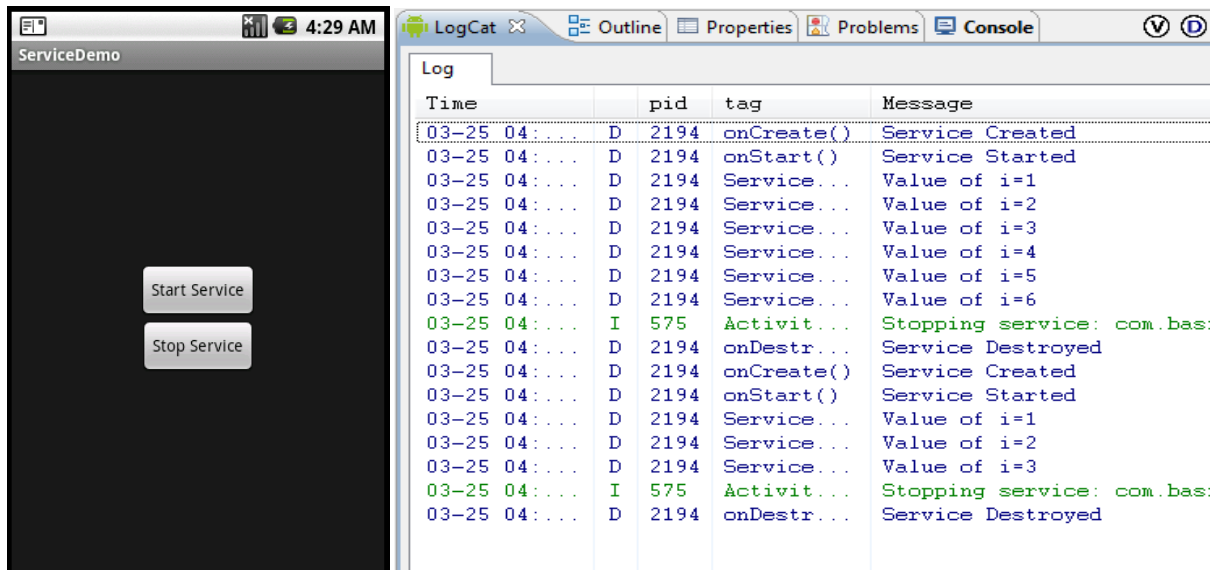
    timer.schedule(notifyTask, 1000, 1000);
}

```

Now to stop the timer, we do following:

```
@Override
public void onDestroy() {
    super.onDestroy();
    if (timer != null) {
        timer.cancel();
    }
    Log.d("onDestroy()", "Service Destroyed");
}
```

If we run the app and test we'll see our buttons are **starting and stopping** the service in **LogCat**



Now, let we want to show a notification in the **Notification Bar** instead of just **LogCat**

```
public void onStart(Intent intent, int startId) {
    super.onStart(intent, startId);
    Log.d("onStart()", "Service Started");

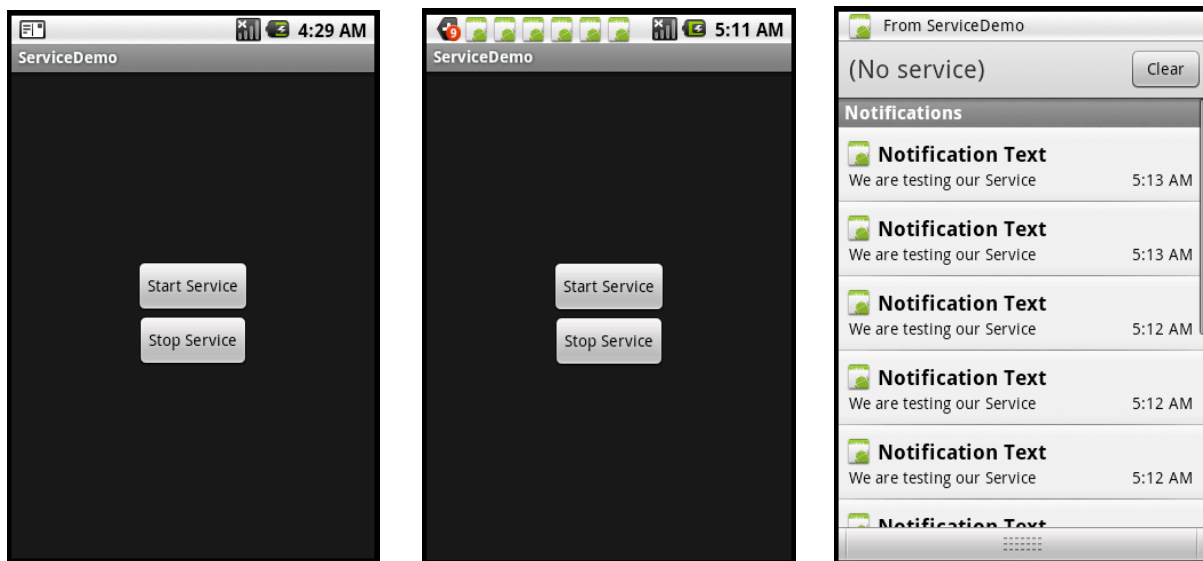
    final NotificationManager notificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    final PendingIntent contentIntent = PendingIntent.getService(this, 0,
        intent, 0);

    notifyTask = new TimerTask() {

        int i = 0;
        public void run() {
            Log.d("MyService", "Service is Running");

            long when = System.currentTimeMillis();
            Notification notification = new Notification(R.drawable.icon,
                "From ServiceDemo", when);
            notification.setLatestEventInfo(MyService.this, "ServiceDemo",
                "We are testing our Service", contentIntent);
            i++;
            notificationManager.notify(i++, notification);
        }
    };
    timer.schedule(notifyTask, 1000, 10000);
}
```

If we run the app and test, we'll see our buttons are **starting and stopping** the service in **Notification**



GOOGLE MOBILE ADS SDK:

Google AdMob is a mobile advertising platform that we can use to generate revenue from our app.

How does it work?

AdMob helps us monetize our mobile app through in-app advertising. Ads can be displayed as banner, interstitial, video, or native ads which are seamlessly added to platform native UI components.

Before we can display ads within our app, we'll need to create an AdMob account and activate one or more ad unit IDs. This is a unique identifier for the places in our app where ads are displayed.

AdMob uses the Google Mobile Ads SDK which helps app developers gain insights about their users and maximize ad revenue. To do so, the default integration of the Mobile Ads SDK collects information such as device information and publisher-provided location information.

SIGNING AND EXPORTING AN APP:

Signing an APP:

Android requires that all APKs be digitally signed with a certificate before they can be installed.

Certificates and Keystores:

A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata

identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.

When we sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as a "fingerprint" that uniquely associates the APK to us and our corresponding private key. This helps Android ensure that any future updates to our APK are authentic and come from the original author. The key used to create this certificate is called the *app signing key*.

A keystore is a binary file that contains one or more private keys. Every app must use the same certificate throughout its lifespan in order for users to be able to install new versions as updates to the app.

Signing in Debug Build:

When running or debugging our project from the IDE, Android Studio automatically signs our APK with a debug certificate generated by the Android SDK tools. The first time we run or debug our project in Android Studio, the IDE automatically creates the debug keystore and certificate in `$HOME/.android/debug.keystore`, and sets the keystore and key passwords. Because the debug certificate is created by the build tools and is insecure by design, most app stores (including the Google Play Store) will not accept an APK signed with a debug certificate for publishing.

Android Studio automatically stores our debug signing information in a signing configuration so we do not have to enter it every time we debug. A signing configuration is an object consisting of all of the necessary information to sign an APK, including the keystore location, keystore password, key name, and key password. We cannot directly edit the debug signing configuration, but we can configure how we sign our release build.

Expiry of The Debug Certificate:

The self-signed certificate used to sign our APK for debugging has an expiration date of 365 days from its creation date. When the certificate expires, we will get a build error. To fix this problem, simply delete the debug.keystore file. The file is stored in the following locations:

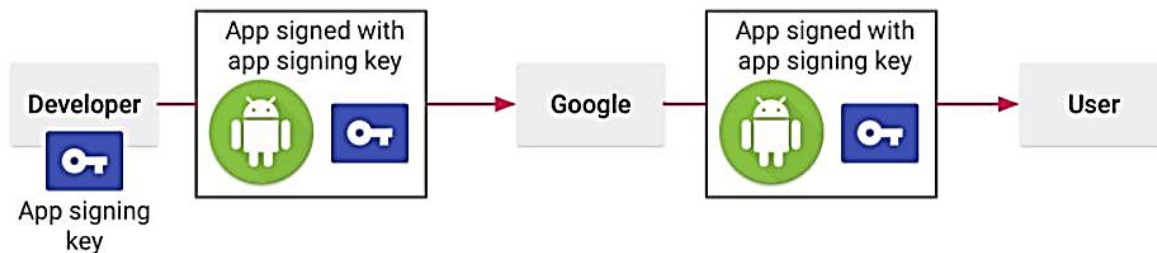
- ❖ `~/.android/` on OS X and Linux
- ❖ `C:\Documents and Settings\<user>\.android\` on Windows XP
- ❖ `C:\Users\<user>\.android\` on Windows Vista and Windows 7, 8, and 10

The next time we build and run the debug build type, the build tools will regenerate a new keystore and debug key.

Manage Our Own Key and Keystore:

Our app signing key is used to verify our identity as a developer and to ensure seamless and secure updates for our users, managing our key and keeping it secure are very important, both for us and for our users. We can choose either to opt into use Google Play App Signing to securely manage and store our app signing key using Google's infrastructure or to manage and secure our own keystore and app signing key.

Instead of using Google Play App Signing, we can choose to manage our own app signing key and keystore. If we choose to manage our own app signing key and keystore, we are responsible for securing the key and the keystore. We should choose a strong password for our keystore, and a separate strong password for each private key stored in the keystore. We must keep our keystore in a safe and secure place. If we lose access to our app signing key or our key is compromised, Google cannot retrieve the app signing key for us, and we will not be able to release new versions of our app to users as updates to the original app.



Sign an APK:

Regardless of how we choose to manage our key and keystore, we can use Android Studio to sign our APKs (with either the upload key or the app signing key), either manually, or by configuring our build process to automatically sign APKs.

If we choose to manage and secure our own app signing key and keystore, we will sign our APKs with our app signing key. If we choose to use Google Play App Signing to manage and secure our app signing key and keystore, we will sign our APKs with our upload key.

Generate A Key and Keystore:

We can generate an app signing or upload key using Android Studio, using the following steps:

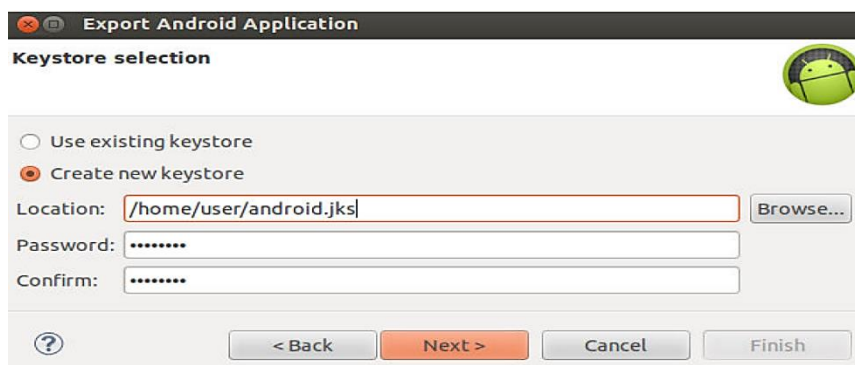
1. In the menu bar, click **Build > Generate Signed APK**.
2. Select a module from the drop down, and click **Next**.
3. Click **Create new** to create a new key and keystore.
4. On the **New Key Store** window, provide the following information for our keystore and key:
 - a. **Keystore:**
 - **Key store path:** Select the location where keystore should be created.
 - **Password:** Create and confirm a secure password for keystore.
 - b. **Key:**
 - **Alias:** Enter an identifying name for key.
 - **Password:** Create and confirm a secure password for key. This should be different from the password we chose for our keystore

- **Validity (years):** Set the length of time in years that key will be valid. Our key should be valid for at least 25 years, so we can sign app updates with the same key through the lifespan of our app.
- **Certificate:** Enter some information about ourself for our certificate. This information is not displayed in our app, but is included in our certificate as part of the APK.
- Once we complete the form, click **OK**.

Exporting an APP:

To sign our app for release with ADT while exporting, follow these steps:

1. Select the project in the Package Explorer and select **File > Export**.
2. On the *Export* window, select **Export Android Application** and click **Next**.
3. On the *Export Android Application* window, select the project we want to sign and click **Next**.
4. Enter the location to create a keystore and a keystore password. If we already have a keystore, select **Use existing keystore**, then enter keystore's location and password, and go to step 6.



Export Android Application

Keystore selection

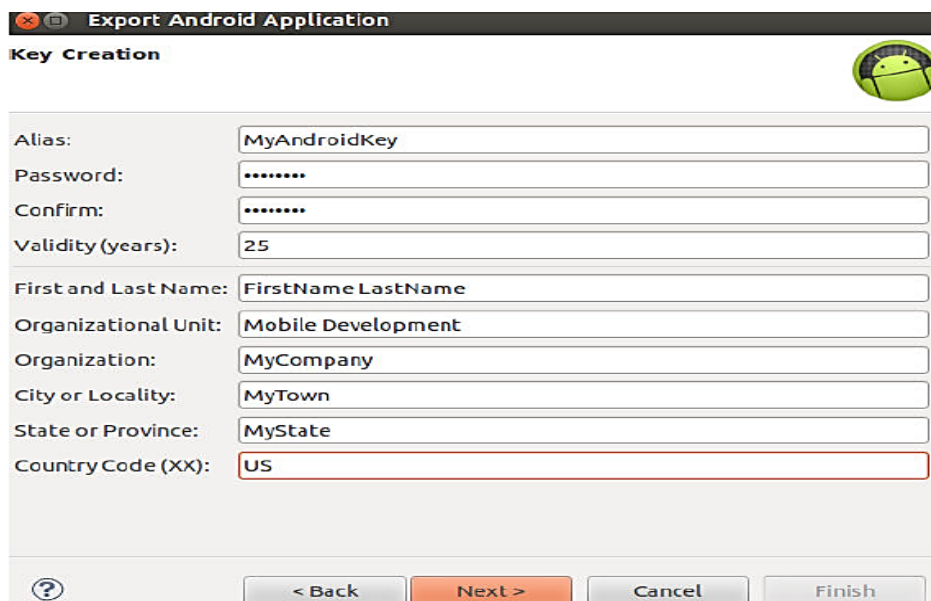
☐ Use existing keystore
☒ Create new keystore

Location:

Password:

Confirm:

5. Provide the required information as shown in figure below. Our key should be valid for at least 25 years, so we can sign app updates with the same key through the lifespan of our app.



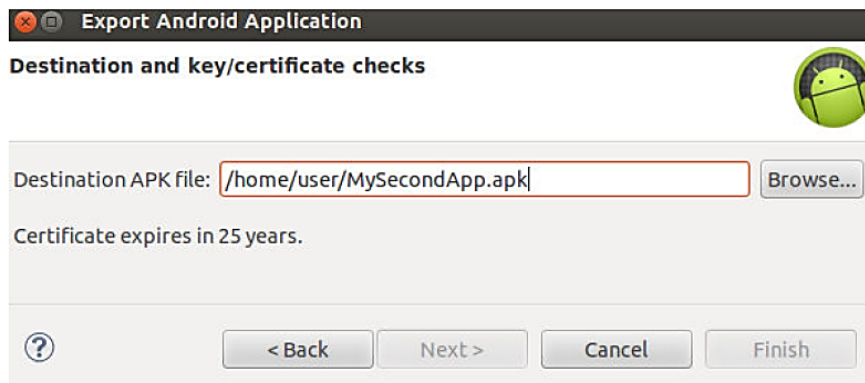
Export Android Application

Key Creation

Alias:
 Password:
 Confirm:
 Validity (years):

First and Last Name:
 Organizational Unit:
 Organization:
 City or Locality:
 State or Province:
 Country Code (XX):

6. Select the location to export the signed APK.



PUBLISHING AN APP TO THE PLAY STORE

There are basically three things to consider before publishing an android app:

1. Register for a Publisher Account:

- ❖ Visit the Google Play Developer Console.
- ❖ Enter basic information about **developer identity** such as name, email address, and so on. We can modify this information later.
- ❖ Read and accept the **Developer Distribution Agreement** for our country or region. Note that apps and store listings that we publish on Google Play must comply with the Developer Program Policies and US export law.
- ❖ Pay a **\$25 USD registration fee** using Google payments. If we don't have a Google payments account, we can quickly set one up during the process.
- ❖ When our registration is verified, we'll be notified at the email address we entered during registration.

2. Set Up a Google Payments Merchant Account:

If we want to sell priced apps, in-app products, or subscriptions, we'll need a Google payments merchant account. We can set one up at any time, but first review the list of merchant countries.

To set up a Google payments merchant account:

1. Sign in to Google Play Developer Console at <https://play.google.com/apps/publish/>.
2. Open Financial reports on the side navigation.
3. Click Setup a Merchant Account now.

This takes us to the Google payments site; we'll need information about our business to complete this step.



3. Explore the Developer Console:

When our registration is verified, we can sign in to our Developer Console, which is the home for our app publishing operations and tools on Google Play.

- All applications
- Financial reports
- Settings
- Announcements

ALL APPLICATIONS

[+ Add new application](#)

APP NAME	PRICE	ACTIVE / TOTAL INSTALLS	AVG. RATING / TOTAL #	CRASHES & ANRS	LAST UPDATE	STATUS
 Animal Translator 1.1	Free	12,078 / 185,410	★ 3.29 / 566		Apr 21, 2010	Published
 Earthquake! 3.8	Free	71,426 / 785,829	★ 4.15 / 6,212		Feb 1, 2013	Published

Page 1 of 1