What is Database?

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).

Database Integration:

- Database integration is the process used to aggregate information from multiple sources like social media, sensor data from IoT, data warehouses, customer transactions, and more and share a current, clean version of it across an organization.
- In Mobile Application Development, databases for mobile applications are geared towards storing different data types, as well as offline and cloud-based data. Developing mobile applications also requires a database that offers information security and data mobility.

Which one is better for mobile application development?

- PostgreSQL. A unique relational database, PostgreSQL is the best database for Android and iOS apps. Developers can customize this database as they want; that's why it's the most preferred mobile app database.

Need of database:

- **Data Security –**
  Mobile app databases ensure ultimate data security from hacking and theft. DBMS arrives with various user logins, and every time the new users need permission to access the database.

- **Pointing Out Errors –**
  The information that app databases store is highly dependable as DBMS comes with a checking system that points out errors that need removal**.**

- **Easy & Fast Search –**
  Developers can quickly search through mobile app databases using Data Query Language (DQL).

- **Hassle-Free Updates –**
  DBMS includes Data Manipulation Languages (DML), which ensure timely and seamless updates of the app database.

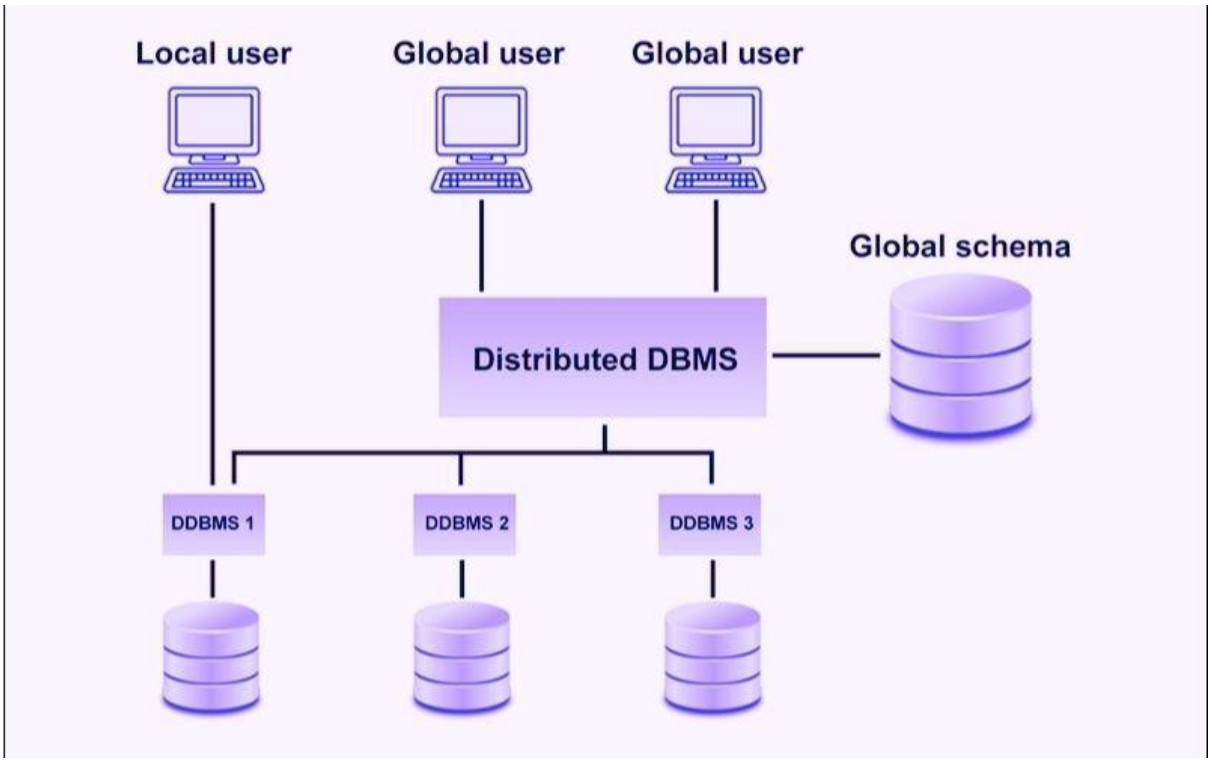
Types of Mobile App Databases:

Distributed Database:

When the data is stored across distinct physical locations in a database, it’s a distributed database. It can be located in various computers placed in the exact physical location or scattered over a network of interconnected computers. A distributed DBMS manages the distributed database in a manner so that it appears as one single database to users.

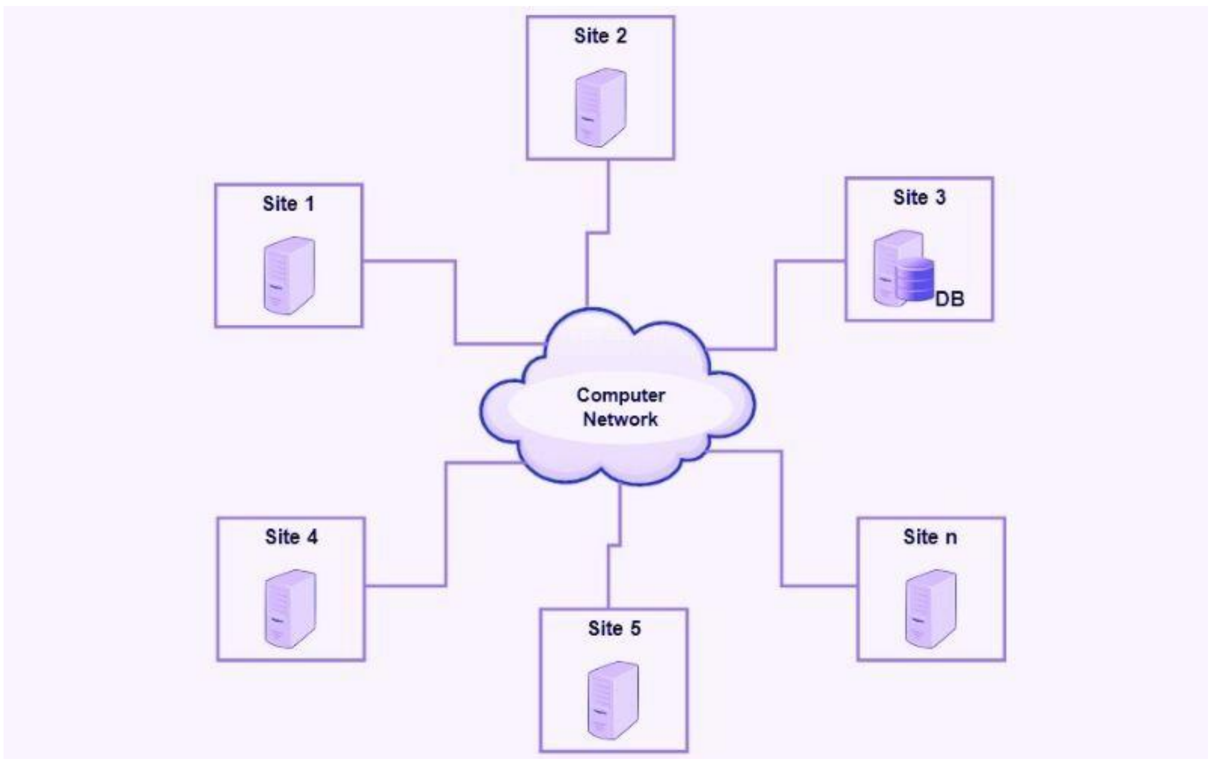
Features:

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.



Centralized Database

- A database where data is stored in a “center” spot, it’s a centralized database. Here the “center” location states the database is saved in a single place but can be accessed from any external resource.
- For instance, when you have the complete data on your work PC, you will have easy access to it from your personal PC if that database is a centralized one.
- Various kinds of authentication procedures are applied for the verification and validation of end users, likewise, a registration number is provided by the application procedures which keeps a track and record of data usage. The local area office handles this thing.

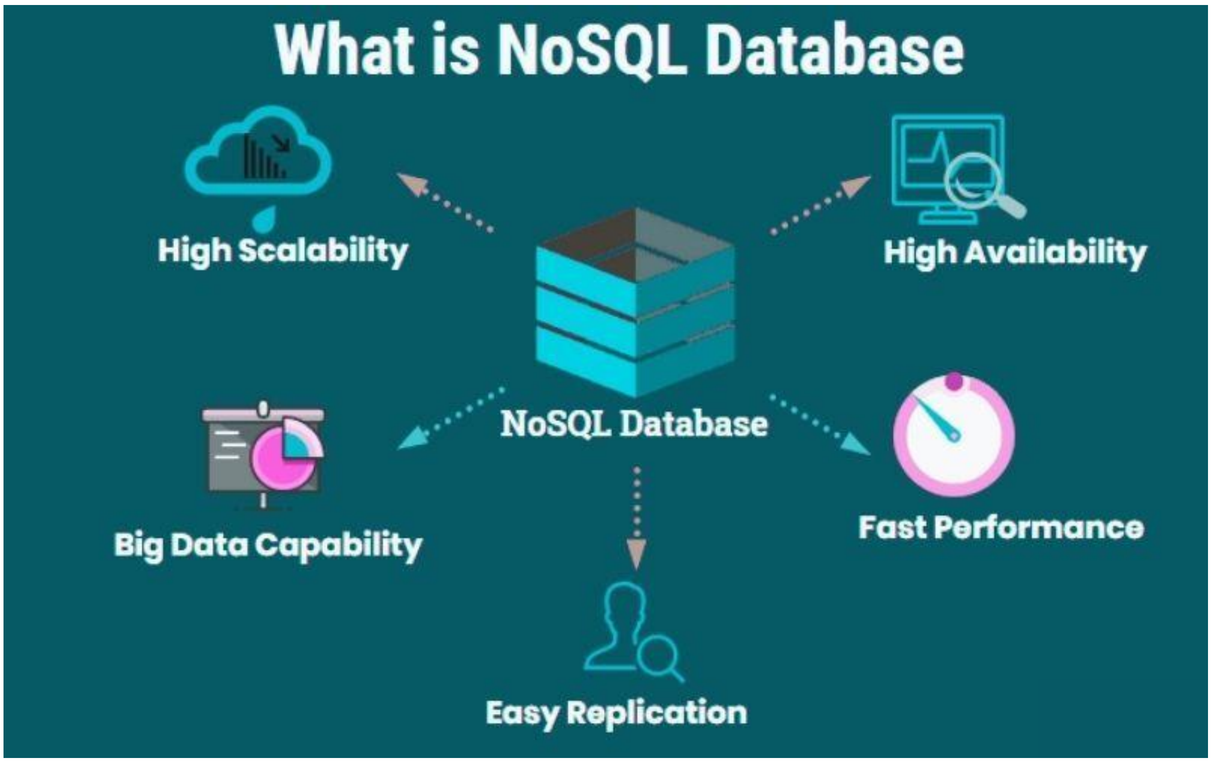


Cloud Database

Cloud-based database apps perform in the cloud. Such apps are a great innovation as they come with a large amount of space to house data. Cloud is the technology over the internet that functions as a centralized database as they are always available, and one can access it from anywhere.

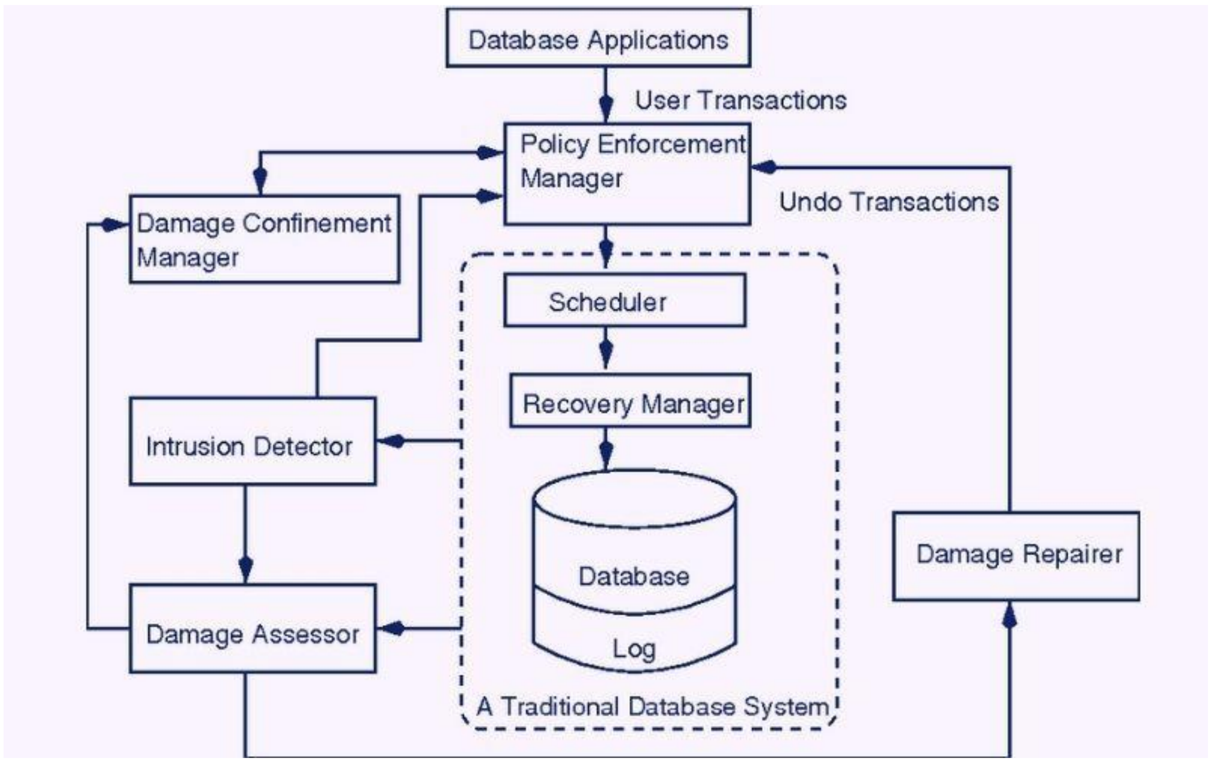
NoSQL Database

The only type of database that works uniquely from all the other application databases is NoSQL. General mobile app databases store data in rows and columns pattern, but NoSQL arrives with a flexible schema that permits a developer to house data in various shapes and sizes.



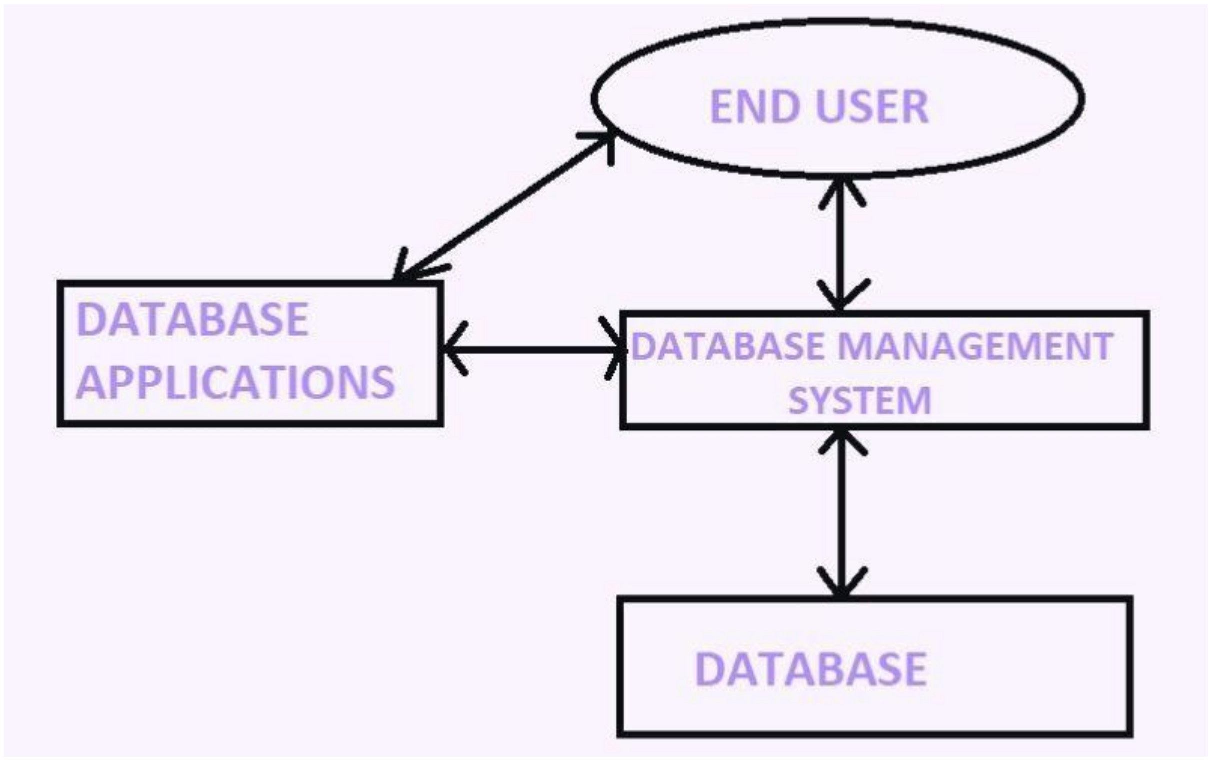
Commercial Database

- As organizations run on enterprise-based database apps, they need an extensive database to house employee information.
- Here arrive commercial app databases. These provide login controls and ample storage to ensure data security.
- Commercial Database are that which has been created for Commercial purpose only. 2. Examples: **MYSQL, PostgreSQL, MongoDB** etc. Examples: Oracle, DB2, Splunk etc.
- A system of commercial information and data of enterprises that are organized, arranged and updated, and are provided to tax authorities by business organizations as prescribed by law.



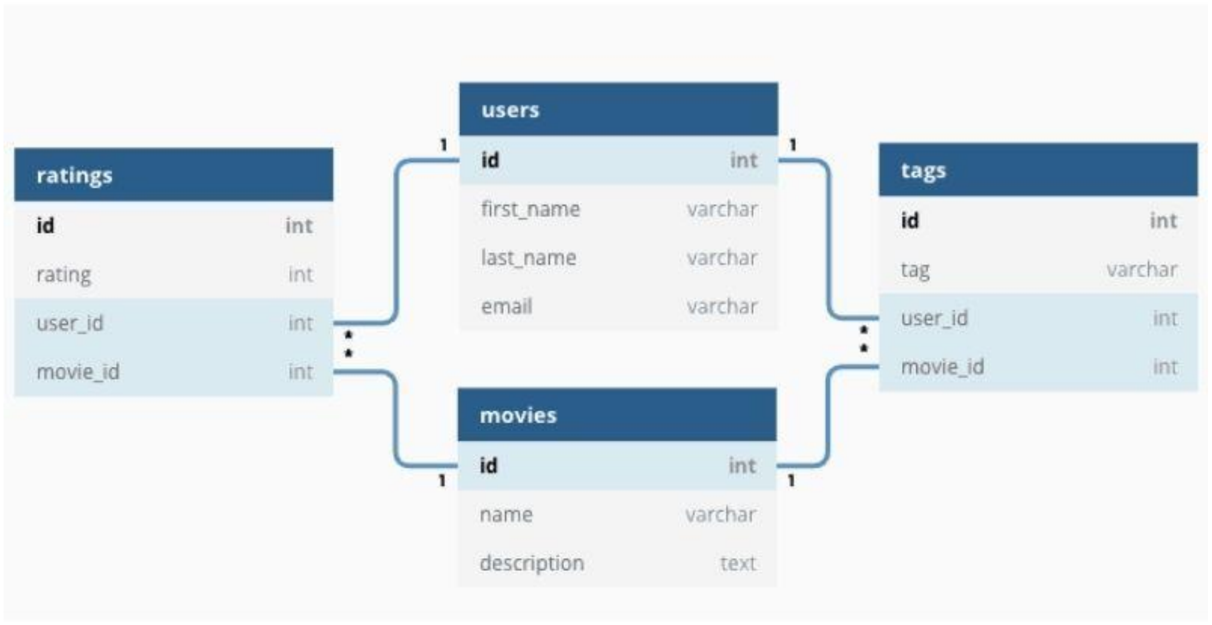
End-User Database

- When we enter our personal details on any social media page or online shopping site, that information is saved on the end-user database.
- Also, if you have seen the cookies that we accept while browsing websites, the databases store cookies that help personalize your online experience.
- The whole process takes place in the background and doesn't disturb any user information.



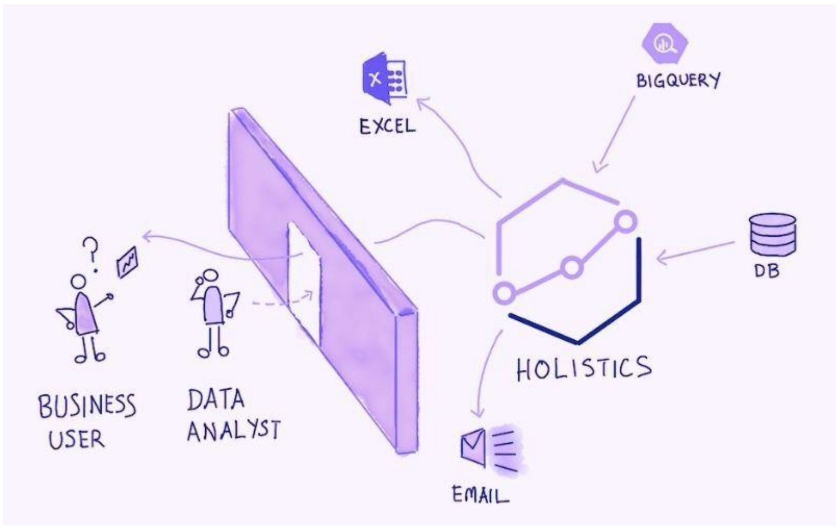
Relational Database

- The growing firms always try creating relationships between two or more app databases.
- This is known as relational data, and a relational database ease handling such sort of data.
- The data is stored in rows and columns in this app database, making it easy to build relationships between databases.



General criteria to choose the right databases for mobile apps
Various criteria assist in picking a suitable database for your mobile apps.

Structure of Your Data



- The structure points out the way you want to store and retrieve your data. Mobile apps deal with data in different formats.
- Offline applications store entire data on mobile devices, whereas Online applications rely on server access to let the store data function.

Size of Your Data

The data size is the data quantity that you want to store and retrieve as crucial app data. The amount of data may vary according to a combination of chosen data structure, the caliber of database to differentiate data across various file systems and servers.
So, you should pick a mobile database considering the entire volume of data generated by an app at any particular time and the data size you want to retrieve from the database.

Data Modeling

Before selecting a mobile app database, you need to perform data modeling as suggested by experts. It’s a representation of data structures that you want to store in the database and robust expression of the business needs.
Data modeling is best when your app holds features, such as reporting, search queries, location-based features, and more. Such mobile apps need various databases to handle distinct sorts of data.
For instance, Uber uses various databases, like MongoDB, MySQL, etc. such databases help it store a high amount of incoming data.

Speed & Scale

Scale and speed let us know the time incoming reads and writes to your app, demand to service. Some databases assist in optimizing read-heavy applications, while others are best to support write-heavy solutions.
Choosing a database with the caliber to handle an app’s I/O requirements leads to a scalable architecture.
For instance, MongoDB may be faster than MySQL to handle a large volume of unstructured data, but the latter is faster for structured data.

Data Security

While using decentralized and synchronized storage, it’s essential to securely transmit, access, and store data. For this, you would address authentication, data in motion, data at rest, and read/write access.
Authentication needs to be flexible and permit the use of public, standard, and custom authentication providers. For data at rest on the client and server, you need support for data-level encryption and file system encryption. Communication needs to carry over a secure channel for data in motion, such as TLS or SSL.

Data Synchronization Between Backend Server And Local Database

Many apps come with features that allow them to perform offline but demand an internet connection to store local data to the application's server.
For instance, Dropbox facilitates editing and crafting new files even in offline mode. And when you go online, the changes get synced to the cloud.
So, choose the app database that can allow automatic local database sync to the cloud server and vice-versa, like Couchbase.

For Highly Scalable Mobile Apps

While scaling your app, you think of appending more resources as servers that lead to an efficient database.
The database needs to be multi-threaded according to which it should hold the caliber to use the resources and manage the parallel processing.
Multithreading facilitates a database to schedule the parallel jobs on the current resources and reduce the workload on the server-side.
Furthermore, you should also have a distributed database to split the services on varied threads to decrease the workload of the main database. This leads to improved parallel processing of databases.

Resolve Data Conflicts Between Devices

A mobile app that changes the same data on various devices simultaneously may end up with conflicts. The database should always be supportive of conflict resolution mechanisms. It's essential and should permit resolution automatically in the cloud, on the device, by a human, or an external system.

Users With A Low Network Issue

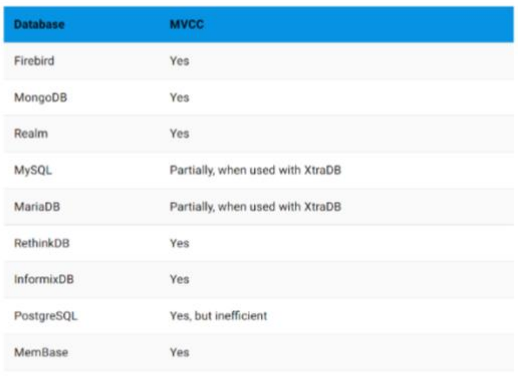
While transferring the required data, if the network connection of any SQL database disconnects from the client-side storage, it leads to an error message. And if it happens frequently, it may demand database re-configuration.
So picking a database that provides better reliability and doesn't lose connection frequently is better in this case.

Pushing Database Changes & New App Updates

If we want to update our mobile app, it will demand some modifications to the local database. So, developers should stay updated with old database versions.
The database you select should facilitate the addition of new tables and fields and handle old APIs and database structures for the users who have the latest app version.

Consider Databases That Follow Multiversion Concurrency Control (MVCC) Method.

Such a method permits simultaneous access with no interference in processes or threads. MVCC facilitates a reader to check data's snapshot before the writers make changes by permitting read and write operations parallelly.



Database	MVCC
Firebird	Yes
MongoDB	Yes
Realm	Yes
MySQL	Partially, when used with XtraDB
MariaDB	Partially, when used with XtraDB
RethinkDB	Yes
InformixDB	Yes
PostgreSQL	Yes, but inefficient
MemBase	Yes

Predictive Caching

It is best to improve the mobile app's performance by looking at when, how, and where users use your app. Based on their behavioral traits, you can identify the users' segment and serve them with specific information they always want.
You can cache and make data available locally before the users log in to your unique app ideas. You can also pick MongoDB as it offers predictive caching that permits developers to serve the users with predictive-cached data before they ask for it.

Database Caching To bring down the load

We should append a caching layer to decrease the servers' load. Moreover, we need to cache data on this caching layer to avoid a recurring request for the same data. This lowers the number of requests approaching the server and makes the server's performance more efficient.
For example, mobile apps like Amazon receive various recurring requests. One can decrease the server's load by appending the caching layer. It holds the caliber to manage millions of requests with the lowest latency. You can also use Memcached and Redic for such needs.

Low Latency Challenges

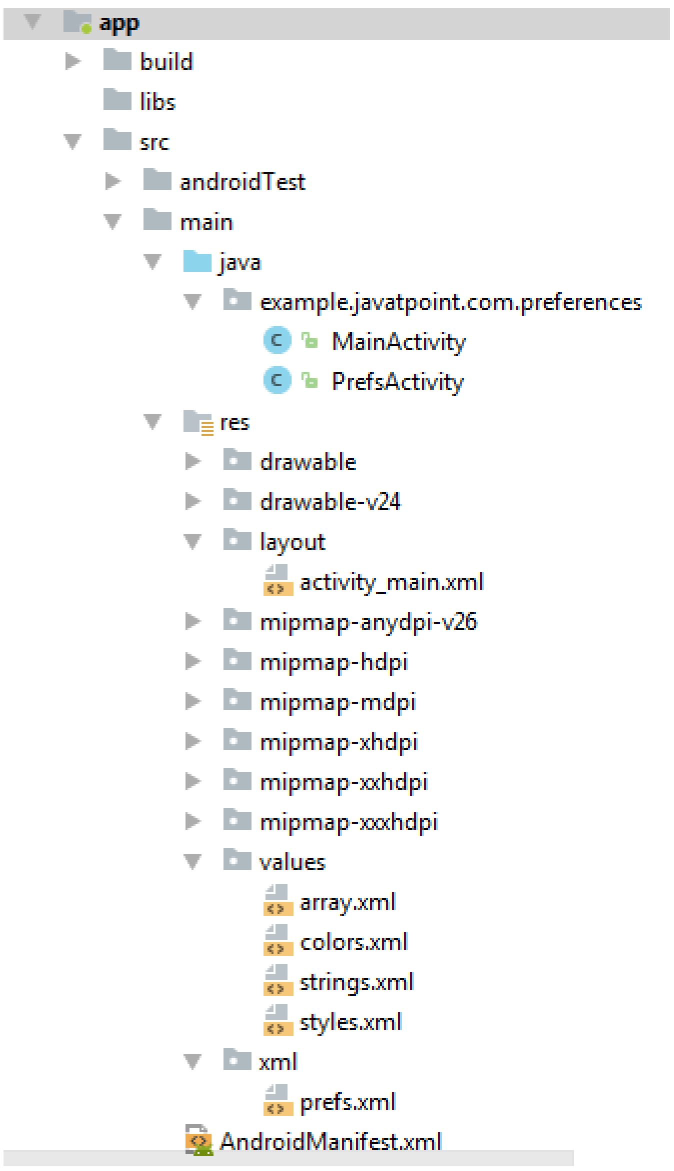
Low latency is vital for online gaming and real-time apps. High latency brings a wrong impression to the app users. Anything that moves down from 500ms is the high latency.
So, you can follow any method to reduce the replication and latency of the database.
Moreover, you can choose databases with, on an average lower latency:

- **PostgreSQL** – for transactional operations in real-time.
- **Pipeline DB** – SQL database best for streaming apps.
- **Rethink DB** – best for real-time apps.
- **MongoDB**

2. Android Preferences

- Android shared preference is used to store and retrieve primitive information. In android, string, integer, long, number etc. are considered as primitive data type.
- Android Shared preferences are used to store data in key and value pair so that we can retrieve the value on the basis of key.
- It is widely used to get information from user such as in settings.

Example:



```
app
  build
  libs
  src
    androidTest
    main
      java
        example.javatpoint.com.preferences
          MainActivity
          PrefsActivity
      res
        drawable
        drawable-v24
        layout
          activity_main.xml
        mipmap-anydpi-v26
        mipmap-hdpi
        mipmap-mdpi
        mipmap-xhdpi
        mipmap-xxhdpi
        mipmap-xxxhdpi
        values
          array.xml
          colors.xml
          strings.xml
          styles.xml
        xml
          prefs.xml
      AndroidManifest.xml
```

activity_main.xml

Drag one textview and two buttons from the pallete.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.preferences.MainActivity">
    <TextView
        android:id="@+id/txtPrefs"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:text="Data:" />

    <Button
        android:id="@+id/storeinformation"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/showinformation"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="18dp"
        android:text="Store Information" />

<Button
        android:id="@+id/showinformation"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="17dp"
        android:text="Show Information" />
</RelativeLayout>
```

array.xml

It is created inside res/values directory.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>
    <string-array name="listOptions">
        <item>English</item>
        <item>Hindi</item>
        <item>Other</item>
    </string-array>

    <string-array name="listValues">
        <item>English Language</item>
        <item>Hindi Language</item>
        <item>Other Language</item>
    </string-array>
</resources>
```

prefs.xml

It is created inside res/xml directory.
File: prefs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:summary="Username and password information"
        android:title="Login information" >
        <EditTextPreference
            android:key="username"
            android:summary="Please enter your login username"
            android:title="Username" />
        <EditTextPreference
            android:key="password"
            android:summary="Enter your password"
            android:title="Password" />
    </PreferenceCategory>

    <PreferenceCategory
        android:summary="Username and password information"
        android:title="Settings" >
        <CheckBoxPreference
            android:key="checkBox"
            android:summary="On/Off"
            android:title="Keep me logged in" />

        <ListPreference
            android:entries="@array/listOptions"
            android:entryValues="@array/listValues"
            android:key="listpref"
            android:summary="List preference example"
            android:title="List preference" />
    </PreferenceCategory>
</PreferenceScreen>
```

Main Activity Class

```
package example.javatpoint.com.preferences;
import android.content.Intent;
import android.content.SharedPreferences;
import android.preference.PreferenceManager;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button storeinformation = (Button) findViewById(R.id.storeinformation);
        Button showinformation = (Button) findViewById(R.id.showinformation);
        textView = (TextView) findViewById(R.id.txtPrefs);

        View.OnClickListener listener = new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                switch (v.getId()) {
                    case R.id.storeinformation:
```

```
                    Intent intent = new Intent(MainActivity.this,PrefsActivity.class);
                    startActivity(intent);
                    break;
                case R.id.showinformation:
                    displaySharedPreferences();
                    break;
                default:
                    break;
            }
        }
    };
    storeinformation.setOnClickListener(listener);
    showinformation.setOnClickListener(listener);
  }


  private void displaySharedPreferences() {
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(MainAc
tivity.this);
    String username = prefs.getString("username", "Default NickName");
    String passw = prefs.getString("password", "Default Password");
    boolean checkBox = prefs.getBoolean("checkBox", false);
    String listPrefs = prefs.getString("listpref", "Default list prefs");
    StringBuilder builder = new StringBuilder();
    builder.append("Username: " + username + "\n");
    builder.append("Password: " + passw + "\n");
    builder.append("Keep me logged in: " + String.valueOf(checkBox) + "\n");
    builder.append("List preference: " + listPrefs);
    textView.setText(builder.toString());
  }
   }
```

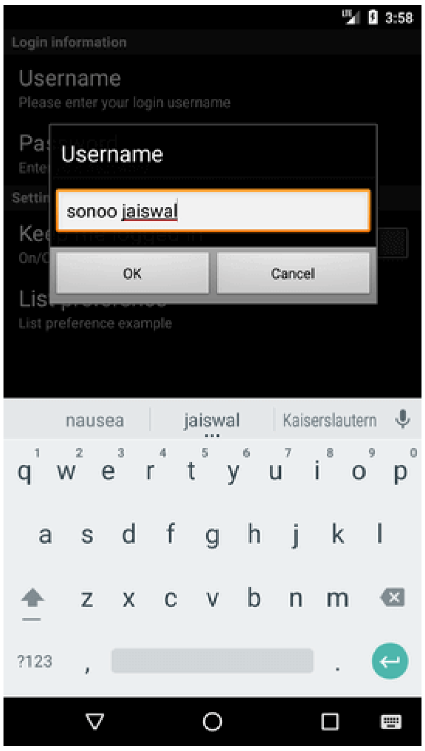
PrefsActivity class

```
  package example.javatpoint.com.preferences;
  import android.os.Bundle;
  import android.preference.PreferenceActivity;
public class PrefsActivity extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.prefs);
    }
     }
```

AndroidManifest.xml

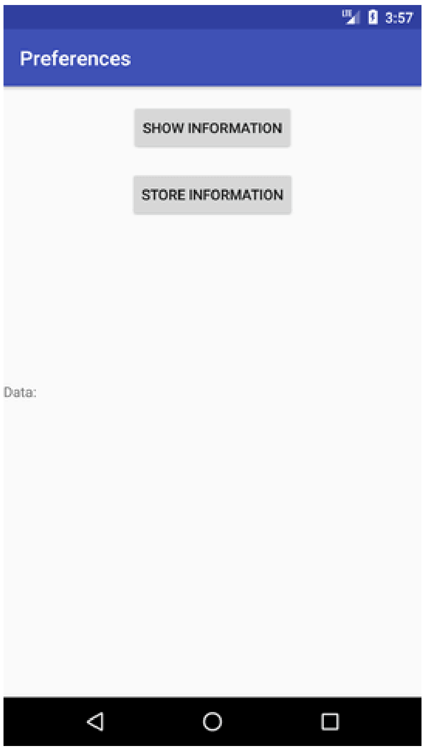
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="example.javatpoint.com.preferences">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity  android:name=".PrefsActivity"
            android:theme="@android:style/Theme.Black.NoTitleBar" >
        </activity>
    </application>
  </manifest>
```

Output:

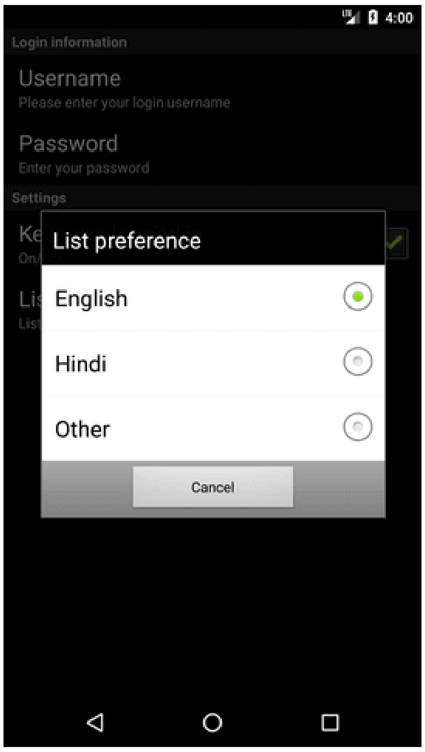


Username

sonoo jaiswal



Data:

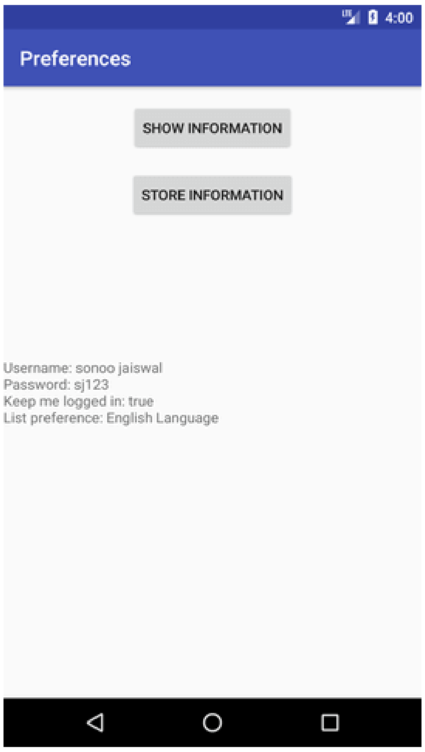


List preference

English [✓]

Hindi

Other



```
Username: sonoo jaiswal
Password: sj123
Keep me logged in: true
List preference: English Language
```

One of the most Interesting Data Storage options **Android** provides its users is **Shared Preferences**. **Shared Preferences** is the way in which one can store and retrieve small amounts of primitive data as key/value pairs to a file on the device storage such as String, int, float, Boolean that make up your preferences in an XML file inside the app on the device storage.

How is Shared Preferences different from Saved Instance State?

Shared Preferences	Saved Instance State
Persist Data across user sessions, even if the app is killed and restarted, or the device is rebooted	Preserves state data across activity instances in the same user session.
Data that should be remembered across sessions, such as the user’s preferred settings or their game score.	Data that should not be remembered across sessions, such as the currently selected tab or current state of activity.
A common use is to store user preferences	A common use is to recreate the state after the device has been rotated

Following are the methods of Shared Preferences

contains(String key): This method is used to check whether the preferences contain a preference.

edit(): This method is used to create a new Editor for these preferences, through which you can make modifications to the data in the preferences and atomically commit those changes back to the SharedPreferences object.

getAll(): This method is used to retrieve all values from the preferences.

getBoolean(String key, boolean defValue): This method is used to retrieve a boolean value from the preferences.

getFloat(String key, float defValue): This method is used to retrieve a float value from the preferences.

getInt(String key, int defValue): This method is used to retrieve an int value from the preferences.

getLong(String key, long defValue): This method is used to retrieve a long value from the preferences.

getString(String key, String defValue): This method is used to retrieve a String value from the preferences.

Nested classes of Shared Preferences

1. **SharedPreferences.Editor**: Interface used to write(edit) data in the SP file. Once editing has been done, one must **commit()** or **apply()** the changes made to the file.
2. **SharedPreferences.OnSharedPreferenceChangeListener()**: Called when a shared preference is changed, added, or removed. This may be called even if a

preference is set to its existing value. This callback will be run on your main thread.

3) Broadcast Receivers

What is Broadcast Receivers in mobile application development?

Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

Types:

There are mainly two types Broadcast Receivers,

- **Static Broadcast Receivers:** These types of Receivers are declared in the manifest file and works even if the app is closed.
- **Dynamic Broadcast Receivers:** These types of receivers work only if the app is active or minimized.

There are following two important steps to make Broadcast Receiver works for the system broadcasted intents −

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

There is one additional steps in case you are going to implement your custom intents then you will have to create and broadcast those intents.

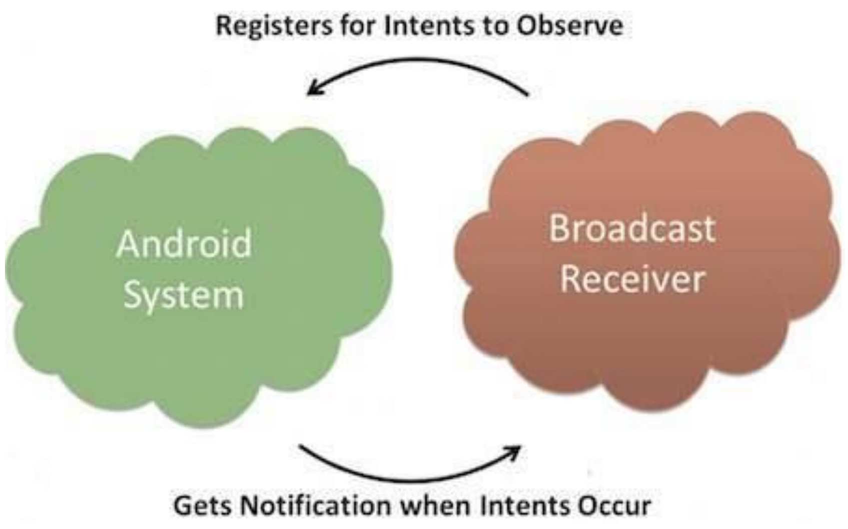
Creating the Broadcast Receiver

A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and overriding the onReceive() method where each message is received as a **Intent** object parameter.

```
public class MyReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
   }
}
```

Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.


Registers for Intents to Observe
Android System
Broadcast Receiver
Gets Notification when Intents Occur

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <receiver android:name="MyReceiver">

    <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED">
      </action>
    </intent-filter>

  </receiver>
</application>
```

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.

Example:
Step 1: Create a New Project
To create a new project in Android Studio

Step 2: Working with the activity_main.xml file
Go to the **activity_main.xml** file and refer to the following code. Below is the code for the **activity_main.xml** file.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 3: Working with the MainActivity file

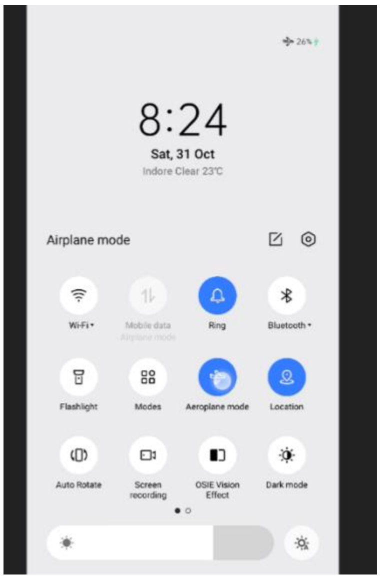
Go to the **MainActivity** file and refer to the following code. Below is the code for the **MainActivity** file. Comments are added inside the code to understand the code in more detail.

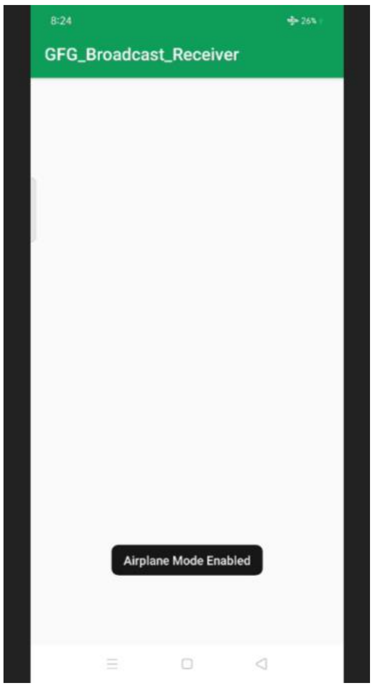
```
import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

	AirplaneModeChangeReceiver airplaneModeChangeReceiver = new
AirplaneModeChangeReceiver();

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_main);
	}

	@Override
	protected void onStart() {
		super.onStart();
		IntentFilter filter = new
IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
		registerReceiver(airplaneModeChangeReceiver, filter);
	}

	@Override
	protected void onStop() {
		super.onStop();
		unregisterReceiver(airplaneModeChangeReceiver);
	}
}
```

Step 4: Create a new class
Go to **app > java > your package name(in which the MainActicity is present) > right-click > New > Kotlin File/Class** and name the files as **AirplaneModeChangeReceiver**. Below is the code for the **AirplaneModeChangeReceiver** file. Comments are added inside the code to understand the code in more detail.

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.provider.Settings;
import android.widget.Toast;

public class AirplaneModeChangeReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {

                if (isAirplaneModeOn(context.getApplicationContext())) {
                        Toast.makeText(context, "AirPlane mode is on",
Toast.LENGTH_SHORT).show();
                } else {
                        Toast.makeText(context, "AirPlane mode is off",
Toast.LENGTH_SHORT).show();
                }
        }

        private static boolean isAirplaneModeOn(Context context) {
                return Settings.System.getInt(context.getContentResolver(),
Settings.Global.AIRPLANE_MODE_ON, 0) != 0;
        }
}
```

Output:





Airplane Mode Enabled

Since from API Level 26, most of the broadcast can only be caught by the dynamic receiver, so we have implemented dynamic receivers in our example. There are some static fields defined in the Intent class which can be used to broadcast different events. We have taken a change of airplane mode as a broadcast event, but there are many events for which broadcast register can be used. Following is some of the important system-wide generated intents: -

Intent	Description Of Event
android.intent.action.BATTERY_LOW :	Indicates low battery condition on the device.
android.intent.action.BOOT_COMPLETED	This is broadcast once after the system has finished booting
android.intent.action.CALL	To perform a call to someone specified by the data
android.intent.action.DATE_CHANGED	Indicates that the date has changed
android.intent.action.REBOOT	Indicates that the device has been a reboot
android.net.conn.CONNECTIVITY_CHANGE	The mobile network or wifi connection is changed(or reset)
android.intent.ACTION_AIRPLANE_MODE_CHANGED	This indicates that airplane mode has been switched on or off.

Restricting broadcasts

An unrestricted broadcast can pose a security threat, because any registered receiver can receive it. For example, if your app uses a normal broadcast to send an implicit Intent that includes sensitive information, an app that contains malware could receive that broadcast. Restricting your broadcast is strongly recommended.
Ways to restrict a broadcast:

- If possible, use a LocalBroadcastManager, which keeps the data inside your app, avoiding any security leaks. You can only use LocalBroadcastManager if you don't need interprocess communication or communication with other apps.
- Use the setPackage() method and pass in the package name. Your broadcast is restricted to apps that match the specified package name.
- Enforce access permissions on the sender side, on the receiver side, or both.

To enforce a permission when *sending* a broadcast:

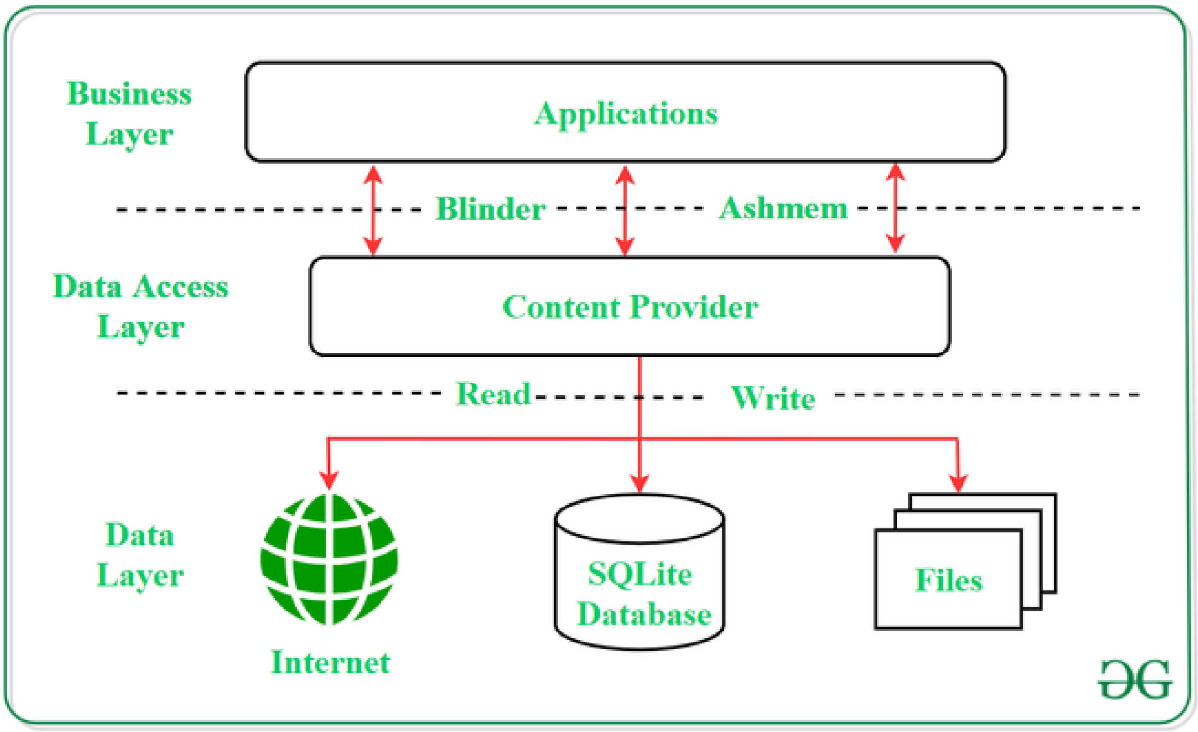
- Supply a non-null permission argument to sendBroadcast(). Only receivers that request this permission using the <uses-permission> tag in their AndroidManifest.xml file can receive the broadcast.

To enforce a permission when *receiving* a broadcast:

- If you register your receiver dynamically, supply a non-null permission to registerReceiver().
- If you register your receiver statically, use the android:permission attribute inside the <receiver> tag in your AndroidManifest.xml.

3). Content providers

In Android, Content Providers are a very important component that serves the purpose of a relational database to store the data of applications. The role of the content provider in the android system is like a central repository in which data of the applications are stored, and it facilitates other applications to securely access and modifies that data based on the user requirements. Android system allows the content provider to store the application data in several ways. Users can manage to store the application data like images, audio, videos, and personal contact information by storing them in SQLite Database, in files**,** or even on a network**.** In order to share the data, content providers have certain permissions that are used to grant or restrict the rights to other applications to interfere with the data.



Content URI
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.

Details of different parts of Content URI:

- **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.
-

If an ID is mentioned in a URI then it is an id-based URI otherwise a directory-based URI.

Operations in Content Provider

Four fundamental operations are possible in Content Provider namely **Create**, **Read**, **Update**, and **Delete**. These operations are often termed as **CRUD operations**.

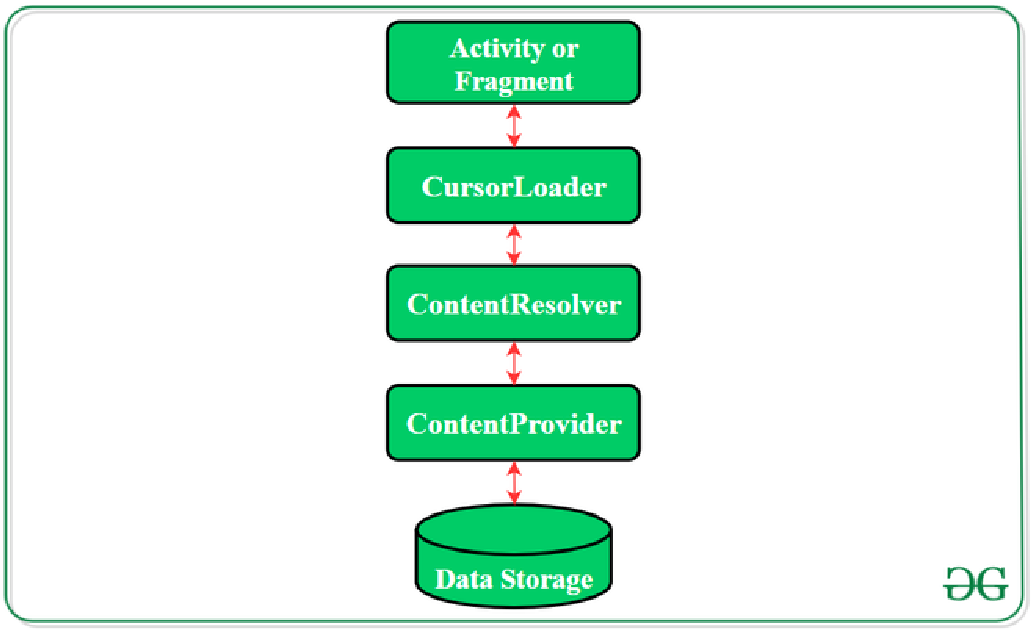
- **Create:** Operation to create data in a content provider.
- **Read:** Used to fetch data from a content provider.
- **Update:** To modify existing data.
- **Delete:** To remove existing data from the storage.

Examples of Content Providers

- Music Playlist.
- Calls Logs.
- Contact List.
- Message Threads.
- Gallery Application.
- File Manager, etc.

Working of the Content Provider

UI components of android applications like Activity and Fragments use an object **CursorLoader** to send query requests to **ContentResolver.** The ContentResolver object sends requests (like create, read, update, and delete) to the **ContentProvider** as a client. After receiving a request, ContentProvider process it and returns the desired result. Below is a diagram to represent these processes in pictorial form.



Creating a Content Provider
Following are the steps which are essential to follow in order to create a Content Provider:

- Create a class in the same directory where the that MainActivity file resides and this class must extend the ContentProvider base class.
- To access the content, define a content provider URI address.
- Create a database to store the application data.
- Implement the six abstract methods of ContentProvider class.
- Register the content provider in AndroidManifest.xml file using <provider> tag.

Following are the six abstract methods and their description which are essential to override as the part of ContenProvider class:

Abstract Method	Description
query()	A method that accepts arguments and fetches the data from the desired table. Data is retired as a cursor object.
insert()	To insert a new row in the database of the content provider. It returns the content URI of the inserted row.
update()	This method is used to update the fields of an existing row. It returns the number of rows updated.
delete()	This method is used to delete the existing rows. It returns the number of rows deleted.

getType()	This method returns the Multipurpose Internet Mail Extension(MIME)
onCreate()	As the content provider is created, the android system calls this method immediately to initialise the provider.

Example

The prime purpose of a content provider is to serve as a central repository of data where users can store and can fetch the data. The access of this repository is given to other applications also but in a safe manner in order to serve the different requirements of the user. The following are the steps involved in implementing a content provider. In this content provider, the user can store the name of persons and can fetch the stored data. Moreover, another application can also access the stored data and can display the data.

Creating a Content Provider:
Step 1: Create a new project

1. Click on File, then New => New Project.
2. Select language as Java/Kotlin.
3. Choose empty activity as a template
4. Select the minimum SDK as per your need.

Step 2: Modify the strings.xml file
All the strings used in the activity are stored here.

```
<resources>

        <string name="app_name">Content_Provider_In_Android</string>

        <string name="hintText">Enter User Name</string>

        <string name="heading">Content Provider In Android</string>

        <string name="insertButtontext">Insert Data</string>

        <string name="loadButtonText">Load Data</string>

</resources>
```

Step 3: Creating the Content Provider class

1. Click on File, then New => Other => ContentProvider.
2. Name the ContentProvider
3. Define authority (it can be anything for example **“com.demo.user.provider”**)
4. Select **Exported** and **Enabled** option
5. Choose the language as Java/Kotlin

This class extends the ContentProvider base class and override the six abstract methods. Below is the complete code to define a content provider.

```
package com.example.contentprovidersinandroid;

import android.content.ContentProvider;

import android.content.ContentUris;

import android.content.ContentValues;

import android.content.Context;

import android.content.UriMatcher;

import android.database.Cursor;

import android.database.sqlite.SQLiteDatabase;

import android.database.sqlite.SQLiteException;

import android.database.sqlite.SQLiteOpenHelper;

import android.database.sqlite.SQLiteQueryBuilder;

import android.net.Uri;

import java.util.HashMap;


public class MyContentProvider extends ContentProvider {

        public MyContentProvider() {

        }

        // defining authority so that other application can access it

        static final String PROVIDER_NAME = "com.demo.user.provider";

        // defining content URI

        static final String URL = "content://" + PROVIDER_NAME + "/users";

        // parsing the content URI

        static final Uri CONTENT_URI = Uri.parse(URL);
}

static final String id = "id";

static final String name = "name";

static final int uriCode = 1;

static final UriMatcher uriMatcher;

private static HashMap<String, String> values;

static {

        // to match the content URI

        // every time user access table under content provider

        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

        // to access whole table

        uriMatcher.addURI(PROVIDER_NAME, "users", uriCode);

        // to access a particular row

        // of the table

        uriMatcher.addURI(PROVIDER_NAME, "users/*", uriCode);

}

@Override

public String getType(Uri uri) {

        switch (uriMatcher.match(uri)) {

                case uriCode:

                        return "vnd.android.cursor.dir/users";

                default:

                throw new IllegalArgumentException("Unsupported URI: " + uri);

        }

}

// creating the database
```

```
@Override
public boolean onCreate() {
        Context context = getContext();
        DatabaseHelper dbHelper = new DatabaseHelper(context);
        db = dbHelper.getWritableDatabase();
        if (db != null) {
                return true;
        }
        return false;
}
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                                        String[] selectionArgs, String sortOrder) {
        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
        qb.setTables(TABLE_NAME);
        switch (uriMatcher.match(uri)) {
                case uriCode:
                        qb.setProjectionMap(values);
                        break;
                default:
                throw new IllegalArgumentException("Unknown URI " + uri);
        }
        if (sortOrder == null || sortOrder == "") {
                sortOrder = id;
        }
        Cursor c = qb.query(db, projection, selection, selectionArgs, null,
                null, sortOrder);
        c.setNotificationUri(getContext().getContentResolver(), uri);
        return c;
    }
    // adding data to the database
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        long rowID = db.insert(TABLE_NAME, "", values);
        if (rowID > 0) {
            Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
            getContext().getContentResolver().notifyChange(_uri, null);
            return _uri;
        }
        throw new SQLiteException("Failed to add a record into " + uri);
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,
                    String[] selectionArgs) {
        int count = 0;
        switch (uriMatcher.match(uri)) {
            case uriCode:
                count = db.update(TABLE_NAME, values, selection,
selectionArgs);
            break;
            default:
```

```
                        throw new IllegalArgumentException("Unknown URI " +
uri);
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return count;
        }
        @Override
        public int delete(Uri uri, String selection, String[] selectionArgs) {
                int count = 0;
                switch (uriMatcher.match(uri)) {
                        case uriCode:
                                count = db.delete(TABLE_NAME, selection, selectionArgs);
                                break;
                        default:
                                throw new IllegalArgumentException("Unknown URI " +
uri);
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return count;
        }
        // creating object of database
        // to perform query
        private SQLiteDatabase db;
        // declaring name of the database
        static final String DATABASE_NAME = "UserDB";
        // declaring table name of the database
        static final String TABLE_NAME = "Users";

        // declaring version of the database

        static final int DATABASE_VERSION = 1;

        // sql query to create the table

        static final String CREATE_DB_TABLE = " CREATE TABLE " +
TABLE_NAME

                        + " (id INTEGER PRIMARY KEY AUTOINCREMENT, "

                        + " name TEXT NOT NULL);";

        // creating a database

        private static class DatabaseHelper extends SQLiteOpenHelper {

                // defining a constructor

                DatabaseHelper(Context context) {

                        super(context, DATABASE_NAME, null, DATABASE_VERSION);

                }

                // creating a table in the database

                @Override

                public void onCreate(SQLiteDatabase db) {

                        db.execSQL(CREATE_DB_TABLE);

                }

                @Override

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

                        // sql query to drop a table

                        // having similar name

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

                        onCreate(db);
```

```
                    }

          }

}
```

Step 4: Design the activity_main.xml layout
One **Textview**, **EditText field, two Buttons**, and a **Textview to display the stored data** will be added in the activity using the below code.

```
<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout

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

          xmlns:app="http://schemas.android.com/apk/res-auto"

          xmlns:tools="http://schemas.android.com/tools"

          android:layout_width="match_parent"

          android:layout_height="match_parent"

          android:background="#168BC34A"

          tools:context=".MainActivity">

          <LinearLayout

                    android:id="@+id/linearLayout"

                    android:layout_width="match_parent"

                    android:layout_height="wrap_content"

                    android:layout_centerVertical="true"

                    android:orientation="vertical"

                    app:layout_constraintBottom_toTopOf="@+id/imageView"

                    app:layout_constraintEnd_toEndOf="parent"

                    app:layout_constraintStart_toStartOf="parent"

                    app:layout_constraintTop_toTopOf="parent"

                    app:layout_constraintVertical_bias="0.13"
```

```
        tools:ignore="MissingConstraints">
        <TextView
            android:id="@+id/textView1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="40dp"
            android:layout_marginBottom="70dp"
            android:fontFamily="@font/roboto"
            android:text="@string/heading"
            android:textAlignment="center"

    android:textAppearance="@style/TextAppearance.AppCompat.Large"
            android:textColor="@android:color/holo_green_dark"
            android:textSize="36sp"
            android:textStyle="bold" />
        <EditText
            android:id="@+id/textName"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="40dp"
            android:fontFamily="@font/roboto"
            android:hint="@string/hintText" />
        <Button
            android:id="@+id/insertButton"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_marginStart="20dp"
            android:layout_marginTop="10dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="20dp"
            android:background="#4CAF50"
            android:fontFamily="@font/roboto"
            android:onClick="onClickAddDetails"
            android:text="@string/insertButtontext"
            android:textAlignment="center"
android:textAppearance="@style/TextAppearance.AppCompat.Display1"
            android:textColor="#FFFFFF"
            android:textStyle="bold" />
<Button
            android:id="@+id/loadButton"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_marginStart="20dp"
            android:layout_marginTop="10dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="20dp"
            android:background="#4CAF50"
            android:fontFamily="@font/roboto"
            android:onClick="onClickShowDetails"
```

```
                    android:text="@string/loadButtonText"
                    android:textAlignment="center"

android:textAppearance="@style/TextAppearance.AppCompat.Display1"
                    android:textColor="#FFFFFF"
                    android:textStyle="bold" />
          <TextView
                    android:id="@+id/res"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_marginStart="20dp"
                    android:layout_marginEnd="20dp"
                    android:clickable="false"
                    android:ems="10"
                    android:fontFamily="@font/roboto"
                    android:textColor="@android:color/holo_green_dark"
                    android:textSize="18sp"
                    android:textStyle="bold" />
</LinearLayout>
<ImageView
          android:id="@+id/imageView"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          app:layout_constraintBottom_toBottomOf="parent"
          app:layout_constraintEnd_toEndOf="parent"
          app:layout_constraintStart_toStartOf="parent"
```

```
                app:srcCompat="@drawable/banner" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 5: Modify the MainActivity file
Button functionalities will be defined in this file. Moreover, the query to be performed while inserting and fetching the data is mentioned here. Below is the complete code.

```
package com.example.contentprovidersinandroid;

import androidx.appcompat.app.AppCompatActivity;

import android.content.ContentValues;

import android.content.Context;

import android.database.Cursor;

import android.net.Uri;

import android.os.Bundle;

import android.view.MotionEvent;

import android.view.View;

import android.view.inputmethod.InputMethodManager;

import android.widget.EditText;

import android.widget.TextView;

import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

        @Override

        protected void onCreate(Bundle savedInstanceState) {

                super.onCreate(savedInstanceState);

                setContentView(R.layout.activity_main);

        }

        @Override

        public boolean onTouchEvent(MotionEvent event) {
```

```
                    InputMethodManager imm =
(InputMethodManager)getSystemService(Context.INPUT_METHOD_SERVICE);

                    imm.hideSoftInputFromWindow(getCurrentFocus().getWindowToken(), 0);

                    return true;

          }

          public void onClickAddDetails(View view) {

                    // class to add values in the database

                    ContentValues values = new ContentValues();

                    // fetching text from user

                    values.put(MyContentProvider.name, ((EditText)
findViewById(R.id.textName)).getText().toString());

                    // inserting into database through content URI

                    getContentResolver().insert(MyContentProvider.CONTENT_URI, values);

                    // displaying a toast message

                    Toast.makeText(getBaseContext(), "New Record Inserted",
Toast.LENGTH_LONG).show();

          }

          public void onClickShowDetails(View view) {

                    // inserting complete table details in this text field

                    TextView resultView= (TextView) findViewById(R.id.res);

                    // creating a cursor object of the

                    // content URI

                    Cursor cursor =
getContentResolver().query(Uri.parse("content://com.demo.user.provider/users"), null,
null, null, null);

                    // iteration of the cursor

                    // to print whole table

                    if(cursor.moveToFirst()) {
```

```
                    StringBuilder strBuild=new StringBuilder();

                    while (!cursor.isAfterLast()) {

                strBuild.append("\n"+cursor.getString(cursor.getColumnIndex("id"))+ "-"+
cursor.getString(cursor.getColumnIndex("name")));

                        cursor.moveToNext();

                    }

                    resultView.setText(strBuild);

                }

                else {

                    resultView.setText("No Records Found");

                }

        }

}
```

Step 6: Modify the AndroidManifest file
The AndroidManifest file must contain the content provider name, authorities, and permissions which enables the content provider to be accessed by other applications.

```
<?xml version="1.0" encoding="utf-8"?>

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

        package="com.example.content_provider_in_android">

        <application

                android:allowBackup="true"

                android:icon="@mipmap/ic_launcher"

                android:label="@string/app_name"

                android:roundIcon="@mipmap/ic_launcher_round"

                android:supportsRtl="true"

                android:theme="@style/AppTheme">
        <provider
    android:name="com.example.contentprovidersinandroid.MyContentProvider"

            android:authorities="com.demo.user.provider"

            android:enabled="true"

            android:exported="true"></provider>

        <activity android:name=".MainActivity">

            <intent-filter>

                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />

            </intent-filter>

        </activity>

        <meta-data

            android:name="preloaded_fonts"

            android:resource="@array/preloaded_fonts" />

    </application>

</manifest>
```

Creating another application to access the Content Provider:

Step 1: Create a new Project

1. Click on File, then New => New Project.
2. Select language as Java/Kotlin.
3. Choose empty activity as a template
4. Select the minimum SDK as per your need.

Step 2: Modify strings.xml file

All the strings used in the activity are stored in this file.

```
<resources>

    <string name="app_name">Accessing_Content_Provider</string>

    <string name="heading">Accessing data of Content Provider</string>

    <string name="loadButtonText">Load Data</string>

</resources>
```

Step 3: Designing the ctivity_main.xml layout
Two TextView is added in the activity, one for heading and one to display the stored data in a content provider. One Button is also added to receive the command to display data. Below is the code to implement this design.

```
<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout

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

        xmlns:app="http://schemas.android.com/apk/res-auto"

        xmlns:tools="http://schemas.android.com/tools"

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:background="#168BC34A"

        tools:context=".MainActivity">

        <LinearLayout

                android:id="@+id/linearLayout"

                android:layout_width="match_parent"

                android:layout_height="wrap_content"

                android:layout_centerVertical="true"

                android:orientation="vertical"

                app:layout_constraintBottom_toTopOf="@+id/imageView"

                app:layout_constraintEnd_toEndOf="parent"

                app:layout_constraintStart_toStartOf="parent"

                app:layout_constraintTop_toTopOf="parent"

                app:layout_constraintVertical_bias="0.13"

                tools:ignore="MissingConstraints">

                <TextView

                        android:id="@+id/textView1"
```

```
                    android:layout_width="match_parent"

                    android:layout_height="wrap_content"

                    android:layout_marginTop="40dp"

                    android:layout_marginBottom="70dp"

                    android:fontFamily="@font/roboto"

                    android:text="@string/heading"

                    android:textAlignment="center"


android:textAppearance="@style/TextAppearance.AppCompat.Large"

                    android:textColor="@android:color/holo_green_dark"

                    android:textSize="36sp"

                    android:textStyle="bold" />

            <Button

                    android:id="@+id/loadButton"

                    android:layout_width="match_parent"

                    android:layout_height="match_parent"

                    android:layout_marginStart="20dp"

                    android:layout_marginTop="10dp"

                    android:layout_marginEnd="20dp"

                    android:layout_marginBottom="20dp"

                    android:background="#4CAF50"

                    android:fontFamily="@font/roboto"

                    android:onClick="onClickShowDetails"

                    android:text="@string/loadButtonText"

                    android:textAlignment="center"
```

```
android:textAppearance="@style/TextAppearance.AppCompat.Display1"
          android:textColor="#FFFFFF"
          android:textStyle="bold" />
     <TextView
          android:id="@+id/res"
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_marginStart="20dp"
          android:layout_marginEnd="20dp"
          android:clickable="false"
          android:ems="10"
          android:fontFamily="@font/roboto"
          android:textColor="@android:color/holo_green_dark"
          android:textSize="18sp"
          android:textStyle="bold" />
</LinearLayout>
<ImageView
     android:id="@+id/imageView"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:srcCompat="@drawable/banner" />
```

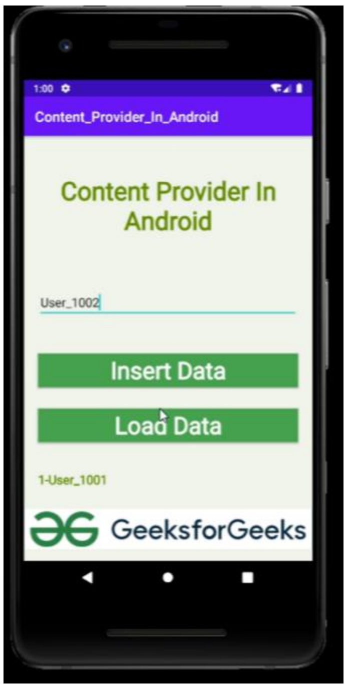
```
</androidx.constraintlayout.widget.ConstraintLayout>
```

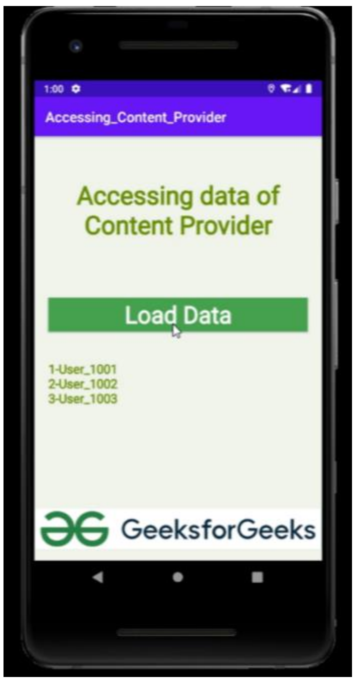
Step 4: Modify the MainActivity file
The ContentURI of the previous application is mentioned here and the same functions which were used in the previous app to display the records will also be used here. Below is the complete code:

```
package com.example.accessingcontentprovider;

import androidx.appcompat.app.AppCompatActivity;

import android.database.Cursor;

import android.net.Uri;

import android.os.Bundle;

import android.view.View;

import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

	Uri CONTENT_URI = Uri.parse("content://com.demo.user.provider/users");

	@Override

	protected void onCreate(Bundle savedInstanceState) {

		super.onCreate(savedInstanceState);

		setContentView(R.layout.activity_main);

	}

	public void onClickShowDetails(View view) {

		// inserting complete table details in this text field

		TextView resultView= (TextView) findViewById(R.id.res);

		// creating a cursor object of the

		// content URI

		Cursor cursor = 
getContentResolver().query(Uri.parse("content://com.demo.user.provider/users"), null, 
null, null, null);
```

```
                    // iteration of the cursor

                    // to print whole table

                    if(cursor.moveToFirst()) {

                              StringBuilder strBuild=new StringBuilder();

                              while (!cursor.isAfterLast()) {

                    strBuild.append("\n"+cursor.getString(cursor.getColumnIndex("id"))+ "-"+
cursor.getString(cursor.getColumnIndex("name")));

                                        cursor.moveToNext();

                              }

                              resultView.setText(strBuild);

                    }

                    else {

                              resultView.setText("No Records Found");

                    }

          }

}
```

Output: Run on Emulator




Accessing_Content_Provider
Accessing data of Content Provider
Load Data
1-User_1001
2-User_1002
3-User_1003
GeeksforGeeks

4) WHAT IS A MOBILE SENSOR?

- Mobile phone sensors can be defined as specific technologies that are able to capture changes in the environment and respond by summoning a specific app to perform a particular task.

- For example, a Near Field Communication (NFC) sensor reacts automatically when the smartphone is approaching a point-of-sale (POS) terminal that supports this technology. When a purchase is made, a payment app is opened and the payment is processed with the use of a fingerprint sensor in a mobile phone.

- Making instant payments is not the only use of sensors in a mobile phone though. There are several types of smartphone sensors, each of which can be used for specific or multiple tasks.

LOCATION-BASED AND MOTION MOBILE SENSORS

These are the sensors that use the smartphone's location and movement.

- **Gravity (accelerometer).** This is the sensor that is used to detect the orientation of the phone and its linear acceleration.
- **Gyroscope.** This is a sensor that helps measure angular velocity expressed in degrees.
- **Geomagnetic field (compass).** These are sensors that are able to detect cardinal points.
- **Proximity.** These are location- and GPS-based sensors that are able to detect whether a certain device is in close proximity to a certain location.

- **GPS.** This is one of the first sensors that was embedded into smartphones, and now it can be used across devices and industries for a lot of business tasks.
- **Phone distance sensor.** This is the sensor that allows for calculating the distance traveled by the user in a certain period of time.

USE CASES FOR LOCATION-BASED AND MOBILE MOTION SENSORS
Location-based and motion sensors on phones are the ones that modern smartphones can’t do without. Their potential, as well as the potential of other types of phone sensors, is better revealed when paired with corresponding applications.
For example:

- A gyroscope sensor in a phone can be used in photo and video apps for image stabilization. It can also be used by fitness tracking devices in combination with an accelerometer and optical sensor that measures pulse.
- A compass sensor in mobile phones can be used by apps for navigating the wilderness.
- Proximity sensors in combination with beacon technology can be used by retailers to send notifications to potential customers when they are in close proximity to their store.
- GPS sensors are widely used by smartphones, smartwatches, and other devices like fleet management solutions for tracking location, route planning, and optimization, and geofencing.

ENVIRONMENTAL MOBILE SENSORS
Environmental sensors are the ones that capture the smallest changes in the surrounding environment and when paired with a specific app they send a notification and update the user on these changes.

- **The ambient light** sensor is the technology that measures the level of surrounding ambient light and regulates the smartphone's screen brightness accordingly with the aim of saving battery charge or reducing the load on a user’s eyes.
- **Ambient temperature.** This is the sensor that can accurately measure the surrounding temperature.
- **The barometer** is the sensor that measures atmospheric pressure.

USE CASES FOR ENVIRONMENTAL MOBILE SENSORS
The ambient light sensor in smartphones can be used by applications for reading. This sensor can make the user experience more enjoyable by automatically adjusting the screen brightness depending on reading conditions.

Ambient temperature and barometer can be used by weather applications for cases where the users want to find out not only the weather in a certain location but also current weather conditions with a high level of accuracy. Also, these sensors can be used by mine workers to identify potentially dangerous changes within the mine.

BIOMETRIC MOBILE SENSORS
Biometric sensors are the ones that process a user's physical characteristics.

- **A fingerprint sensor** is a tool that allows for the scanning of the finger pattern of a user.

- **Facial recognition** is a biometric sensor that uses an image of the user's face and pairs it with camera sensors.

USE CASES FOR BIOMETRIC SENSORS
As a rule biometric sensors are used for user identification and authentication, for example, to unblock the phone, install or uninstall an app or make a payment. However, the scope of their usage goes beyond smartphones.

Biometric sensors can also be **embedded in security devices** and used for employee identification and access control. These devices use retina scanning as the main method for identification since this is currently the only parameter that's impossible to fake.

Also, face recognition is used in face swap and filtering apps. Face swapping allows for accurate facial recognition and exchanging of the face of the user with the face of another person. Filtering apps use the same technology to add filters and effects to the right place in the photo.

ACTIVITY AND HEALTH SENSORS
Below are the sensors that are used in conjunction with different health apps and wearable devices to track a user's activity and health changes.

- **Pedometer.** This is the sensor that is used by smartphones and **wearable devices** to measure the number of steps taken and miles walked per day.
- **Heart rate sensor.** This sensor allows for measuring the heart rate in real-time like in an Apple Watch.

USE CASES FOR ACTIVITY AND HEALTH SENSORS
There are a lot of use cases for activity and health sensors. For example:

- **A heart rate** sensor can be used in wearable devices and in specialized medical devices recommended for patients who need to monitor their heart rate.
- **Microphone sensors** can be used in pulmonary health monitoring, which is especially relevant with Covid-19. The microphone sensor can measure and read respiratory rhythms when paired with an application, inspiratory depth, and cough features, and then record this data for further analysis.
- **Camera sensors** can be used for skin health monitoring. With their help, analyzing the nature of red spots, rashes, and blemishes on the skin is possible. There are a lot of additional ways to use camera sensors beyond health and entertainment apps.

OTHER SENSORS
Barcode/QR code sensors. These are camera-powered sensors that are able to read barcode or QR code information. All modern smartphones already have these sensors built-in so using barcodes and QR codes in business makes a lot of sense. For example, a lot of restaurants use QR codes to prevent virus transmission from physical menus and to make changes to the menu easily. Also, barcode scanning with the help of the camera is often a feature of dieting and nutrition apps; it allows for quick scanning of the product code to add it to personal nutrition and calorie intake diary.

A microphone sensor is a tool that recognizes and listens to voice commands such as asking Apple’s Siri to do something. It may also be used for foreign speech recognition, translation, podcasting, and other tasks.

5). Mobile Application Services:

A Service is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform inter process communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Note: The WorkManager API offers a flexible way of scheduling tasks, and is able to run these jobs as foreground services if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

Note: If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground. In most situations, for example, you shouldn't access location information from the background. Instead, schedule tasks using WorkManager.

Bound

A service is *bound* when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses started and bound services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple of callback methods: onStartCommand() to allow components to start it and onBind() to allow binding.

Regardless of whether your service is started, bound, or both, any application component can use the service (even from a separate application) in the same way that any component can use an activity—by starting it with an Intent. However, you can declare the service as *private* in the manifest file and block access from other applications. This is discussed more in the section about Declaring the service in the manifest.

Choosing between a service and a thread

A service is simply a component that can run in the background, even when the user is not interacting with your application, so you should create a service only if that is what you need.

If you must perform work outside of your main thread, but only while the user is interacting with your application, you should instead create a new thread in the context of another application component. For example, if you want to play some music, but only while your activity is running, you might create a thread in onCreate(), start running it in onStart(), and stop it in onStop(). Also consider using thread pools and executors from the java.util.concurrent package or Kotlin coroutines instead of the traditional Thread class. See the Threading on Android document for more information about moving execution to background threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

The basics

To create a service, you must create a subclass of Service or use one of its existing subclasses. In your implementation, you must override some callback methods that handle key aspects of the service lifecycle and provide a mechanism that allows the components to bind to the service, if appropriate. These are the most important callback methods that you should override:

onStartCommand()

The system invokes this method by calling startService() when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling stopSelf() or stopService(). If you only want to provide binding, you don't need to implement this method.

onBind()

The system invokes this method by calling bindService() when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an IBinder. You must always implement this method; however, if you don't want to allow binding, you should return null.

onCreate()

The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either onStartCommand() or onBind()). If the service is already running, this method is not called.

onDestroy()

The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

If a component starts the service by calling startService() (which results in a call to onStartCommand()), the service continues to run until it stops itself with stopSelf() or another component stops it by calling stopService().

If a component calls bindService() to create the service and onStartCommand() is *not* called, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys it.

The Android system stops a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, it's less likely to be killed; if the service is declared to run in the foreground, it's rarely killed. If the service is started and is long-running, the system lowers its position in the list of background tasks over time, and the service becomes highly susceptible to killing—if your service is started, you must design it to gracefully handle restarts by the system. If the system kills your service, it restarts it as soon as resources become available, but this also depends on the value that you return from onStartCommand(). For more information about when the system might destroy a service, see the Processes and Threading document.

In the following sections, you'll see how you can create the startService() and bindService() service methods, as well as how to use them from other application components.

Declaring a service in the manifest

You must declare all services in your application's manifest file, just as you do for activities and other components.

To declare your service, add a <service> element as a child of the <application> element. Here is an example:

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

See the <service> element reference for more information about declaring your service in the manifest.

There are other attributes that you can include in the <service> element to define properties such as the permissions that are required to start the service and the process in which the service should run. The android:name attribute is the only required attribute—it specifies the class name of the service. After you publish your application, leave this name unchanged to avoid the risk of breaking code due to dependence on explicit intents to start or bind the service (read the blog post, Things That Cannot Change).

You can ensure that your service is available to only your app by including the android:exported attribute and setting it to false. This effectively stops other apps from starting your service, even when using an explicit intent.

Note: Users can see what services are running on their device. If they see a service that they don't recognize or trust, they can stop the service. In order to avoid having your service stopped accidentally by users, you need to add the android:description attribute to the <service> element in your app manifest. In the description, provide a short sentence explaining what the service does and what benefits it provides.

Creating a started service

A started service is one that another component starts by calling startService(), which results in a call to the service's onStartCommand() method.

When a service is started, it has a lifecycle that's independent of the component that started it. The service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is complete by calling stopSelf(), or another component can stop it by calling stopService().

An application component such as an activity can start the service by calling startService() and passing an Intent that specifies the service and includes any data for the service to use. The service receives this Intent in the onStartCommand() method.

For instance, suppose an activity needs to save some data to an online database. The activity can start a companion service and deliver it the data to save by passing an intent to startService(). The service receives the intent in onStartCommand(), connects to the Internet, and performs the database transaction. When the transaction is complete, the service stops itself and is destroyed.

The Service class is the base class for all services. When you extend this class, it's important to create a new thread in which the service can complete all of its work; the service uses your application's main thread by default, which can slow the performance of any activity that your application is running.

The Android framework also provides the IntentService subclass of Service that uses a worker thread to handle all of the start requests, one at a time. Using this class is **not recommended** for new apps as it will not work well starting with Android 8 Oreo, due to the introduction of Background execution limits. Moreover, it's deprecated starting with Android 11. You can use JobIntentService as a replacement for IntentService that is compatible with newer versions of Android.

The following sections describe how you can implement your own custom service, however you should strongly consider using WorkManager instead for most use cases. Consult the guide to background processing on Android to see if there is a solution that fits your needs.

Extending the Service class

You can extend the Service class to handle each incoming intent. Here's how a basic implementation might look:

```
class HelloService : Service() {

    private var serviceLooper: Looper? = null
    private var serviceHandler: ServiceHandler? = null

    // Handler that receives messages from the thread
    private inner class ServiceHandler(looper: Looper) : Handler(looper) {

        override fun handleMessage(msg: Message) {
            // Normally we would do some work here, like download a file.
            // For our sample, we just sleep for 5 seconds.
            try {
                Thread.sleep(5000)
            } catch (e: InterruptedException) {
                // Restore interrupt status.
                Thread.currentThread().interrupt()
            }

            // Stop the service using the startId, so that we don't stop
            // the service in the middle of handling another job
            stopSelf(msg.arg1)
        }
    }

    override fun onCreate() {
        // Start up the thread running the service.  Note that we create a
        // separate thread because the service normally runs in the process's
        // main thread, which we don't want to block.  We also make it
        // background priority so CPU-intensive work will not disrupt our UI.
        HandlerThread("ServiceStartArguments",
Process.THREAD_PRIORITY_BACKGROUND).apply {
            start()

            // Get the HandlerThread's Looper and use it for our Handler
            serviceLooper = looper
            serviceHandler = ServiceHandler(looper)
        }
    }

    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show()

        // For each start request, send a message to start a job and deliver the
        // start ID so we know which request we're stopping when we finish the job
        serviceHandler?.obtainMessage()?.also { msg ->
            msg.arg1 = startId
            serviceHandler?.sendMessage(msg)
        }

        // If we get killed, after returning from here, restart
        return START_STICKY
    }

    override fun onBind(intent: Intent): IBinder? {
        // We don't provide binding, so return null
        return null
    }

    override fun onDestroy() {
        Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show()
    }
}
```

The example code handles all incoming calls in onStartCommand() and posts the work to a Handler running on a background thread. It works just like an IntentService and processes all requests serially, one after another. You could change the code to run the work on a thread pool, for example, if you'd like to run multiple requests simultaneously.

Notice that the onStartCommand() method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it. The return value from onStartCommand() must be one of the following constants:

START_NOT_STICKY

> If the system kills the service after onStartCommand() returns, *do not* recreate the service unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

START_STICKY

If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand(), but *do not* redeliver the last intent. Instead, the system calls onStartCommand() with a null intent unless there are pending intents to start the service. In that case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands but are running indefinitely and waiting for a job.

START_REDELIVER_INTENT

If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand() with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

Starting a service

You can start a service from an activity or other application component by passing an Intent to startService() or startForegroundService(). The Android system calls the service's onStartCommand() method and passes it the Intent, which specifies which service to start.

Note: If your app targets API level 26 or higher, the system imposes restrictions on using or creating background services unless the app itself is in the foreground. If an app needs to create a foreground service, the app should call startForegroundService(). That method creates a background service, but the method signals to the system that the service will promote itself to the foreground. Once the service has been created, the service must call its startForeground() method within five seconds.

For example, an activity can start the example service in the previous section (HelloService) using an explicit intent with startService(), as shown here:

```
Intent(this, HelloService::class.java).also { intent ->
    startService(intent)
}
```

The startService() method returns immediately, and the Android system calls the service's onStartCommand() method. If the service isn't already running, the system first calls onCreate(), and then it calls onStartCommand().

If the service doesn't also provide binding, the intent that is delivered with startService() is the only mode of communication between the application component and the service. However, if you want the service to send a result back, the client that starts the service can create a PendingIntent for a broadcast (with getBroadcast()) and deliver it to the service in the Intent that starts the service. The service can then use the broadcast to deliver a result.

Multiple requests to start the service result in multiple corresponding calls to the service's onStartCommand(). However, only one request to stop the service (with stopSelf() or stopService()) is required to stop it.

Stopping a service

A started service must manage its own lifecycle. That is, the system doesn't stop or destroy the service unless it must recover system memory and the service continues to run after onStartCommand() returns. The service must stop itself by calling stopSelf(), or another component can stop it by calling stopService().

Once requested to stop with stopSelf() or stopService(), the system destroys the service as soon as possible.

If your service handles multiple requests to onStartCommand() concurrently, you shouldn't stop the service when you're done processing a start request, as you might have received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use stopSelf(int) to ensure that your request to stop the service is always based on the most recent start request. That is, when you call stopSelf(int), you pass the ID of the start request (the startId delivered to onStartCommand()) to which your stop request corresponds. Then, if the service receives a new start request before you are able to call stopSelf(int), the ID doesn't match and the service doesn't stop.

Creating a bound service

A bound service is one that allows application components to bind to it by calling bindService() to create a long-standing connection. It generally doesn't allow components to *start* it by calling startService().

Create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications through interprocess communication (IPC).

To create a bound service, implement the onBind() callback method to return an IBinder that defines the interface for communication with the service. Other application components can then call bindService() to retrieve the interface and begin calling methods on the service. The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it. You do *not* need to stop a bound service in the same way that you must when the service is started through onStartCommand().

To create a bound service, you must define the interface that specifies how a client can communicate with the service. This interface between the service and a client must be an implementation of IBinder and is what your service must return from the onBind() callback method. After the client receives the IBinder, it can begin interacting with the service through that interface.

Multiple clients can bind to the service simultaneously. When a client is done interacting with the service, it calls unbindService() to unbind. When there are no clients bound to the service, the system destroys the service.

There are multiple ways to implement a bound service, and the implementation is more complicated than a started service. For these reasons, the bound service discussion appears in a separate document about Bound Services.

Sending notifications to the user

When a service is running, it can notify the user of events using snackbar notifications or status bar notifications.

A snackbar notification is a message that appears on the surface of the current window for only a moment before disappearing. A status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).

Usually, a status bar notification is the best technique to use when background work such as a file download has completed, and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity.

Managing the lifecycle of a service

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow either of these two paths:

- A started service

  The service is created when another component calls startService(). The service then runs indefinitely and must stop itself by calling stopSelf(). Another component can also stop the service by calling stopService(). When the service is stopped, the system destroys it.

- A bound service

  The service is created when another component (a client) calls bindService(). The client then communicates with the service through an IBinder interface. The client can close the connection by calling unbindService(). Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. The service does *not* need to stop itself.

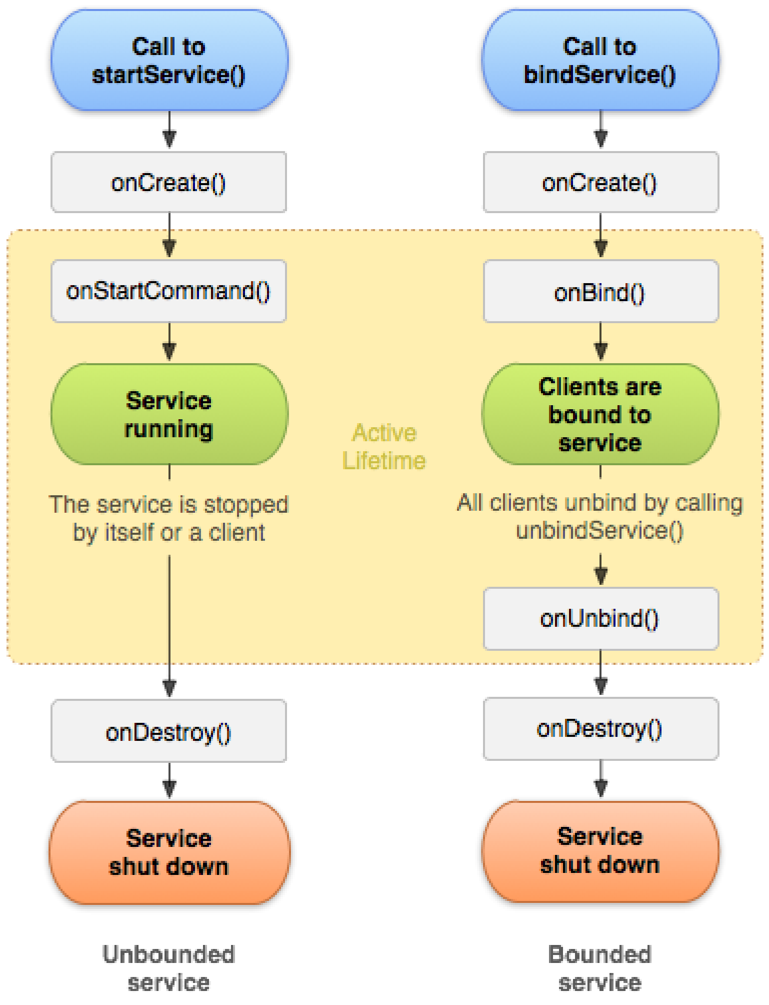
These two paths aren't entirely separate. You can bind to a service that is already started with startService(). For example, you can start a background music service by calling startService() with an Intent that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling bindService(). In cases such as this, stopService() or stopSelf() doesn't actually stop the service until all of the clients unbind.

Implementing the lifecycle callbacks

Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

```
public class ExampleService extends Service {
   int startMode;       // indicates how to behave if the service is killed
   IBinder binder;      // interface for clients that bind
   boolean allowRebind; // indicates whether onRebind should be used

   @Override
   public void onCreate() {
      // The service is being created
   }
   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      // The service is starting, due to a call to startService()
      return startMode;
   }
   @Override
   public IBinder onBind(Intent intent) {
      // A client is binding to the service with bindService()
      return binder;
   }
   @Override
   public boolean onUnbind(Intent intent) {
      // All clients have unbound with unbindService()
      return allowRebind;
   }
   @Override
   public void onRebind(Intent intent) {
      // A client is binding to the service with bindService(),
      // after onUnbind() has already been called
   }
   @Override
   public void onDestroy() {
      // The service is no longer used and is being destroyed
   }
}
```

Note: Unlike the activity lifecycle callback methods, you are not required to call the superclass implementation of these callback methods.



The service lifecycle. The diagram on the left shows the lifecycle when the service is created with startService() and the diagram on the right shows the lifecycle when the service is created with bindService().

Although the figure separates services that are created by startService() from those created by bindService(), keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. A service that was initially started with onStartCommand() (by a client calling startService()) can still receive a call to onBind() (when a client calls bindService()).

By implementing these methods, you can monitor these two nested loops of the service's lifecycle:

- The **entire lifetime** of a service occurs between the time that onCreate() is called and the time that onDestroy() returns. Like an activity, a service does its initial setup in onCreate() and releases all remaining resources in onDestroy(). For example, a music playback service can create the thread where the music is played in onCreate(), and then it can stop the thread in onDestroy().

Note: The onCreate() and onDestroy() methods are called for all services, whether they're created by startService() or bindService().

- The **active lifetime** of a service begins with a call to either onStartCommand() or onBind(). Each method is handed the Intent that was passed to either startService() or bindService().

  If the service is started, the active lifetime ends at the same time that the entire lifetime ends (the service is still active even after onStartCommand() returns). If the service is bound, the active lifetime ends when onUnbind() returns.

Note: Although a started service is stopped by a call to either stopSelf() or stopService(), there isn't a respective callback for the service (there's no onStop() callback). Unless the service is bound to a client, the system destroys it when the service is stopped—onDestroy() is the only callback received.

For more information about creating a service that provides binding, see the Bound Services document, which includes more information about the onRebind() callback method in the section about Managing the lifecycle of a bound service.

7). Intents and Intent Filters

An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

- **Starting an activity**

  An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to startActivity(). The Intent describes the activity to start and carries any necessary data.

  If you want to receive a result from the activity when it finishes, call startActivityForResult(). Your activity receives the result as a separate Intent object in your activity's onActivityResult() callback. For more information, see the Activities guide.

- **Starting a service**

  A Service is a component that performs operations in the background without a user interface. With Android 5.0 (API level 21) and later, you can start a service with JobScheduler. For more information about JobScheduler, see its API-reference documentation.

  For versions earlier than Android 5.0 (API level 21), you can start a service by using methods of the Service class. You can start a service to perform a one-time operation (such as downloading a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data.

  If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService(). For more information, see the Services guide.

- **Delivering a broadcast**

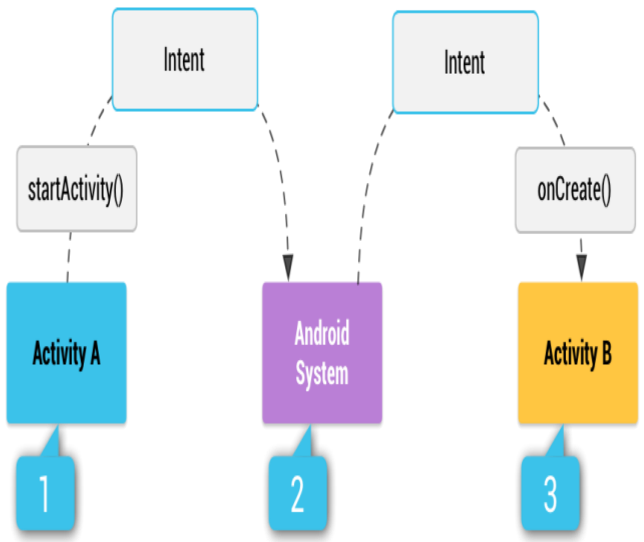
A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to sendBroadcast() or sendOrderedBroadcast().

The rest of this page explains how intents work and how to use them. For related information, see Interacting with Other Apps and Sharing Content.

Intent types

There are two types of intents:

- **Explicit intents** specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, you might start a new activity within your app in response to a user action, or start a service to download a file in the background.
- **Implicit intents** do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.



How an implicit intent is delivered through the system to start another activity: **[1]** Activity *A* creates an Intent with an action description and passes it to startActivity(). **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity by invoking its onCreate() method and passing it the Intent.

When you use an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the *intent filters* declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent.

Likewise, if you do *not* declare any intent filters for an activity, then it can be started only with an explicit intent.

Building an intent

An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).

The primary information contained in an Intent is the following:

Component name

The name of the component to start.

This is optional, but it's the critical piece of information that makes an intent *explicit*, meaning that the intent should be delivered only to the app component defined by the component name. Without a component name, the intent is *implicit* and the system decides which component should receive the intent based on the other intent information (such as the action, data, and category—described below). If you need to start a specific component in your app, you should specify the component name.

Note: When starting a **Service**, *always specify the component name*. Otherwise, you cannot be certain what service will respond to the intent, and the user cannot see which service starts.

This field of the Intent is a ComponentName object, which you can specify using a fully qualified class name of the target component, including the package name of the app, for example, com.example.ExampleActivity. You can set the component name with setComponent(), setClass(), setClassName(), or with the Intent constructor.

Action

A string that specifies the generic action to perform (such as *view* or *pick*).

In the case of a broadcast intent, this is the action that took place and is being reported. The action largely determines how the rest of the intent is structured—particularly the information that is contained in the data and extras.

You can specify your own actions for use by intents within your app (or for use by other apps to invoke components in your app), but you usually specify action constants defined by the Intent class or other framework classes. Here are some common actions for starting an activity:

ACTION_VIEW

Use this action in an intent with startActivity() when you have some information that an activity can show to the user, such as a photo to view in a gallery app, or an address to view in a map app.

ACTION_SEND

Also known as the *share* intent, you should use this in an intent with startActivity() when you have some data that the user can share through another app, such as an email app or social sharing app.

See the Intent class reference for more constants that define generic actions. Other actions are defined elsewhere in the Android framework, such as in Settings for actions that open specific screens in the system's Settings app.

You can specify the action for an intent with setAction() or with an Intent constructor.

If you define your own actions, be sure to include your app's package name as a prefix, as shown in the following example:

```
static final String ACTION_TIMETRAVEL = "com.example.action.TIMETRAVEL";
```

Data

The URI (a Uri object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied is generally dictated by the intent's action. For example, if the action is ACTION_EDIT, the data should contain the URI of the document to edit.

When creating an intent, it's often important to specify the type of data (its MIME type) in addition to its URI. For example, an activity that's able to display images probably won't be able to play an audio file, even though the URI formats could be similar. Specifying the MIME type of your data helps the Android system find the best component to receive your intent. However, the MIME type can sometimes be inferred from the URI—particularly when the data is a content: URI. A content: URI indicates the data is located on the device and controlled by a ContentProvider, which makes the data MIME type visible to the system.

To set only the data URI, call setData(). To set only the MIME type, call setType(). If necessary, you can set both explicitly with setDataAndType().

Caution: If you want to set both the URI and MIME type, *don't* call **setData()** and **setType()** because they each nullify the value of the other. Always use **setDataAndType()** to set both URI and MIME type.

Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an intent, but most intents do not require a category. Here are some common categories:

CATEGORY_BROWSABLE

The target activity allows itself to be started by a web browser to display data referenced by a link, such as an image or an e-mail message.

CATEGORY_LAUNCHER

The activity is the initial activity of a task and is listed in the system's application launcher.

See the Intent class description for the full list of categories.

You can specify a category with addCategory().

These properties listed above (component name, action, data, and category) represent the defining characteristics of an intent. By reading these properties, the Android system is able to resolve which app component it should start. However, an intent can carry additional information that does not affect how it is resolved to an app component. An intent can also supply the following information:

Extras

Key-value pairs that carry additional information required to accomplish the requested action. Just as some actions use particular kinds of data URIs, some actions also use particular extras.

You can add extra data with various putExtra() methods, each accepting two parameters: the key name and the value. You can also create a Bundle object with all the extra data, then insert the Bundle in the Intent with putExtras().

For example, when creating an intent to send an email with ACTION_SEND, you can specify the *to* recipient with the EXTRA_EMAIL key, and specify the *subject* with the EXTRA_SUBJECT key.

The Intent class specifies many EXTRA_* constants for standardized data types. If you need to declare your own extra keys (for intents that your app receives), be sure to include your app's package name as a prefix, as shown in the following example:

```
static final String EXTRA_GIGAWATTS = "com.example.EXTRA_GIGAWATTS";
```

Flags

Flags are defined in the Intent class that function as metadata for the intent. The flags may instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities).

For more information, see the setFlags() method.

Example explicit intent

An explicit intent is one that you use to launch a specific app component, such as a particular activity or service in your app. To create an explicit intent, define the component name for the Intent object—all other intent properties are optional.

For example, if you built a service in your app, named DownloadService, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context
// The fileUrl is a string URL, such as "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

The Intent(Context, Class) constructor supplies the app Context and the component a Class object. As such, this intent explicitly starts the DownloadService class in the app.

For more information about building and starting a service, see the Services guide.

Example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:orientation="vertical"
  tools:context=".MainActivity">
  <Button
    android:id="@+id/buton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="intent filter button" />
</LinearLayout>
```

Add the following code to **src/MainActivity.java**

```
package com.example.andy.myapplication;
import android.content.Intent;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.RadioButton;
public class MainActivity extends AppCompatActivity {
   RadioButton radioButton;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      final Button button = findViewById(R.id.buton);
      button.setOnClickListener(new View.OnClickListener() {
         @Override
public void onClick(View v) {
            Intent intent = new Intent(Intent.ACTION_SEND);
            intent.setType("message/rfc822");
            intent.putExtra(Intent.EXTRA_EMAIL, new
String[]{"contact@tutorialspoint.com"});
            intent.putExtra(Intent.EXTRA_SUBJECT, "Welcome to tutorialspoint.com");
            startActivity(Intent.createChooser(intent, "Choose default Mail App"));
         }
      });
}
}
```

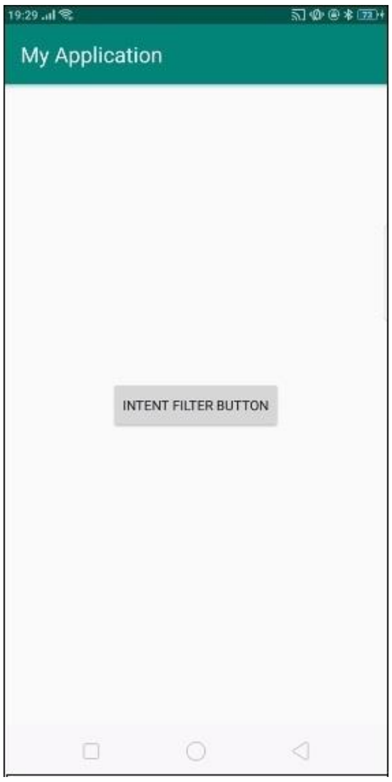
In the above when user click on button it will call intent using ACTION_SEND and will set type as message/rfc882. Now we passed out email id and subject message.

Add the following code to **manifest.xml**

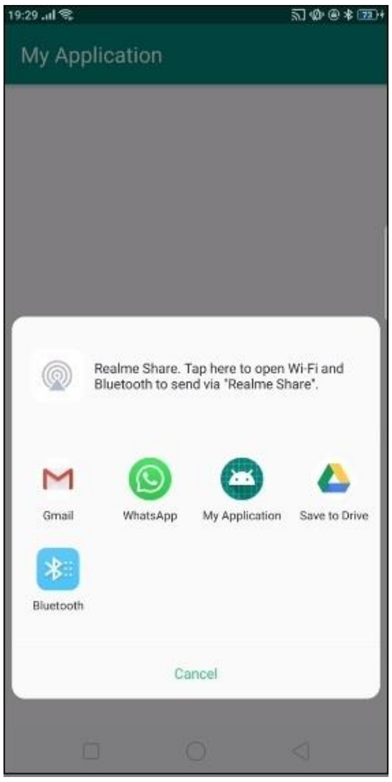
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.andy.myapplication">
   <uses-permission android:name="android.permission.INTERNET" />
   <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">
      <activity android:name=".MainActivity">
```

```
        <intent-filter>
                              <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
            <action android:name="android.intent.action.SEND" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="message/rfc822" />
        </intent-filter>
      </activity>
</application>
</manifest>
```

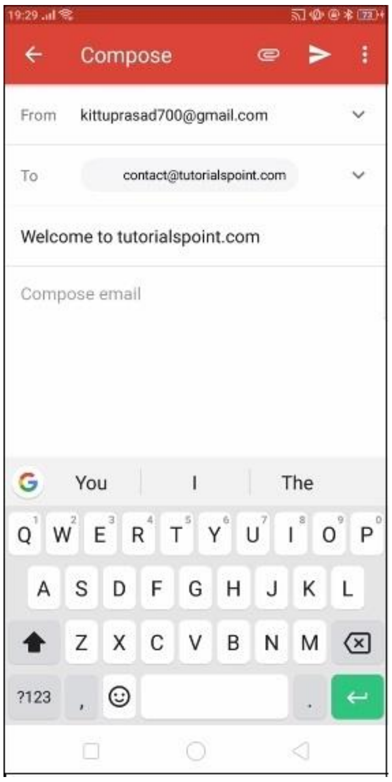
In the above we have declared action, category and data. Let's try to run your application. I assume you have connected your actual Android Mobile device with your computer. To run the app from android studio, open one of your project's activity files and click Run icon from the toolbar. Select your mobile device as an option and then check your mobile device which will display your default screen



Click on above button it will call intent chooser to choose application to send data from intent.


19:29
My Application
Realme Share. Tap here to open Wi-Fi and Bluetooth to send via "Realme Share".
Gmail
WhatsApp
My Application
Save to Drive
Bluetooth
Cancel

We have selected gmail application



From kittuprasad700@gmail.com

To contact@tutorialspoint.com

Welcome to tutorialspoint.com

Compose email

In the above result, it going to take the data from intent and append to gmail application.