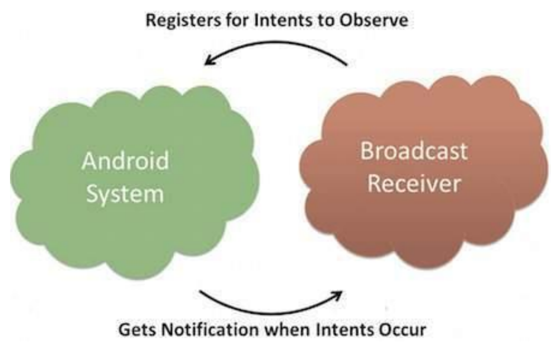
Broadcast Receiver

A broadcast receiver (receiver) is **an Android component which allows you to register for system or application events**.

- It is an Android component which allows you to register for system or application events
- Simply respond to broadcast events from other apps or from the Android OS . For example, events like phone booting, low battery, charger connected
- Many broadcasts originate from system
- Application can also originate broadcasts, by creating a status bar notification to alert user when a broadcast event occurs
- It is a gateway to other components and it is intended to do minimal amount of work
- An intent used to send broadcasts to other applications, called broadcast intents: it may be system events or application events

Broadcast Receiver's job is to **pass a notification to the user,** in case a specific event occurs. Each event creates a new Broadcast Receiver object and it runs on the *main* thread of the app, and after run, it is ready for garbage collection. Android mandates a Broadcast Receiver to complete its execution within 10s.


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

There are two ways to register Broadcast Receiver

- **Static:** Use <receiver> tag in your AndroidManifest.xml file
- **Dynamic:** Use Context.registerReceiver () method to dynamically register an instance

Following are some of the important system-wide generated intents

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

The two main things that we have to do in order to use the broadcast receiver,

Registering a BroadcastReceiver:

- ***Dynamic registration***
  - Dynamic Broadcast receivers run only when the app is running
  - It is implemented by extending the **BroadcastReceiver** class, and overriding its only callback method – **onReceive().** As soon as a Broadcast Receiver is triggered to respond to an event, the **onReceive()** executed

```
public class MyCustomBroadcastReceiver extends BroadcastReceiver
{
@override
  public void onReceive(Context context, Intent intent)
 {
   Toast.makeText(context, "The BR has been triggered", Toast.LENGTH_SHORT).show();
  }
 }
```

- ***Static registration***
  - Broadcasts work both when the app is active and even if the app is inactive or closed
  - Registration is done in the manifest file, using <register> tags

```
<receiver android:name="MyReceiver" >
<intent-filter> <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
</intent-filter>
 </receiver>
```

Sending broadcasts

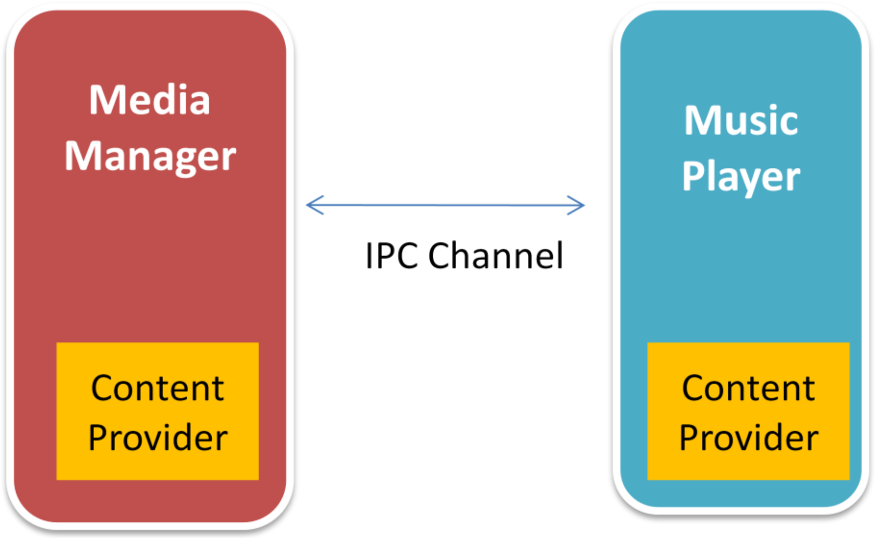
- We can send a broadcasts in apps using three different ways

Method	Description
sendOrderedBroadcast(Intent, String)	This method is used to send broadcasts to one receiver at a time.
sendBroadcast(Intent)	This method is used to send broadcasts to all receivers in an undefined order.

LoadBroadcastManager.sendBroadcast	This method is used to send broadcasts to receivers that are in the same app as the sender.

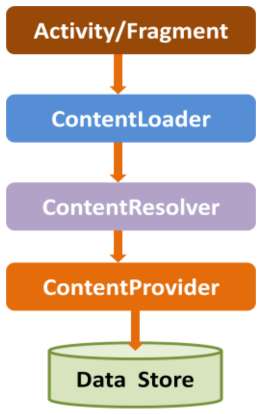
Content Provider

It acts like a central repository in which data of the applications are stored, and it facilitates other applications to securely access and modifies that data



- 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. With some restrictions, these providers are accessible by applications. It hides the implementation details of the data from other apps to provide an abstract and secure way of sharing data across apps
- We can carry out CRUD operations on data of other apps as a black box
- Data of in-built apps are made accessible using in-built content provider
- other apps can access our app's data using customer-built content provider

WORKING



- 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
- To access a provider, give some specific permission in manifest file

Content URI

- Content URI is the key concept used to access the data from a content provider, URI is used as a query string
- Structure of a Content URI: content://authority/optionalPath/optionalID
  - content:// – Mandatory part, 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. Content providers to support different types of data
  - 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

Built in Content Provider

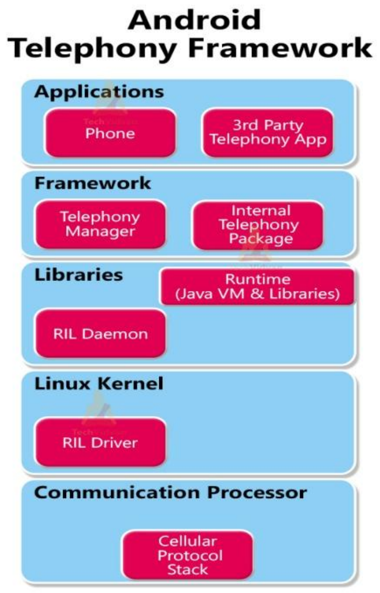
Provider	Remarks
Browser	Read/modify bookmarks, history or web searches
Calllog	View/update call history
Contacts	Store, retrieve or modify personal contact data
Medistore	Access media files
Settings	View/retrieve ring tone, blue tooth and other device settings

Telephony service

Android provides TelephonyManager API to implement telephony functionalities. It includes accessing network and device-type information, and retrieving information about phone state. TelephonyManager is accessed through a telephony service provided by Android, by calling getSystemService() method. The syntax is

TelephonyManager telephonyManager =(TelephonyManager)getSystemService (Context.TELEPHONY_SERVICE);

Access requires READ_PHONE_STATE permission to be included in manifest file. Applications can also register a listener to receive **notification** of telephony state changes



Communication processor

- Collect and distribute data from peripherals (remote)
- Designed to communicate with the data communication network

Radio Interface Layer(RIL) –

- interface through which hardware interact with framework. Two main components are
- RIL Daemon– It starts when the android system starts. It reads the system properties to find a library that is to be used for Vendor RIL
- Vendor RIL–Driver/library that is specific to each modem

Framework services – contains packages and assists Telephony manager

The following are list of telephony services in the android

- Initiating phone calls
- Reading phone, network, data connectivity and SIM states
- Monitoring changes to phone, network, data connectivity and SMS
- Using Intents to send SMS and MMS
- Using SMS Manager to send message
- Handling incoming message

SMS Manager

SmsManager APIs to implement SMS (Short Message Service) related functionalities like sending data, text and SMS message. Two main classes are SmsManager and SmsMessage. Get SmsManager object by calling static method SmsManager .getDefault()

SmsManager.getDefault().sendTextMessage(“phone number”, srcaddress, “msg”, PendingIndent sent , PendingIndent receive)

You can add permission by android.permission.SEND_SMS & android.permission.RECEIVE_SMS in manifest file

Native Data Handling

Scenarios where the app data may have to be stored permanently in order to be retrieve at later. Data can be saved either locally on the device or remotely on the servers. Data could be either primitive or complex in nature, and can be stored on the device in an unstructured or structured manner. Android framework offers several options for persistence:

- ***SharedPreferences***: store primitive private data on key-value pairs
- ***Internal Storage:*** store private data in the device memory
- ***External Storage***: store public data on the shared external storage
- ***SQLite Databases***: store structured data in a private database
- ***Network server*** : store data on the remote web server

Shared preferences

This class allows you to save and retrieve key / value pairs of primitive data type such as ringtone, app setting etc... We use same for saving the primitive data: booleans, floats, ints, longs, and strings Data will persist in the user session. Shared preferences stores data in an XML file in the internal memory of the device.

The creation, storage, and manipulation of the XML file are internally taken care by the SharedPreferences API
To create this object, we use ***getSharedPreferences (String name, int mode)***

To write values,

- Call the method edit () to get a SharedPreferences.Editor
- Add values methods such as putBoolean(), putInt(), putFloat() and putString()
- Persists the new values with commit()

To read values,

- use the methods as getBoolean () and getString ()

For example,

```
SharedPreferences preferences = getSharedPreferences(“SMSPreferences”,MODE_PRIVATE);
 btnSave.setOnClickListener(new OnClickListener() {
 @Override
 public void onClick(View arg0) {
 Editor editor=preferences.edit();
 editor.putBoolean(“SendSMS”, chkEnable.isChecked());
 editor.putString(“Message”, etMessage.getText().toString());
 editor.putString(“Signature”,
etSignature.getText().toString());
editor.commit();
 }
 });
```

To write values,

- Call the method edit () to get a SharedPreferences.Editor
- Add values methods such as putBoolean(), putInt(), putFloat() and putString()
- Persists the new values with commit()

For example,

```
SharedPreferences preferences = getSharedPreferences("SMSPreferences",MODE_PRIVATE);
 btnSave.setOnClickListener(new OnClickListener() {
 @Override
 public void onClick(View arg0) {
 Editor editor=preferences.edit();
 editor.putBoolean("SendSMS", chkEnable.isChecked());
 editor.putString("Message", etMessage.getText().toString());
 editor.putString("Signature",
etSignature.getText().toString());
editor.commit();
 }
 });
```

Internal Storage

Files saved to the internal storage are deprived of their application, allowing other applications can not access them. When the user uninstalls the app, these files are removed. To create and save a private file to the internal storage

- Call openFileOutput () with the file name and the operating mode (in case MODE_PRIVATE) which returns a FileOutputStream;
- Write on file with the write ()
- Close the stream with close ()

For example

```
String FILENAME = "myfile";
String string = "hello world !";
FileOutputStream fos = openFileOutput(FILENAME,  Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

External Storage

It may be removable storage media (such as an SD card) or an internal memory (not removable). Files saved to the external storage are reading for all and can be modified by the user when they allow USB mass storage to transfer files from a computer. It should always call

Environment.getExternalStorageState () to check that the media is available before doing any work with external storage.

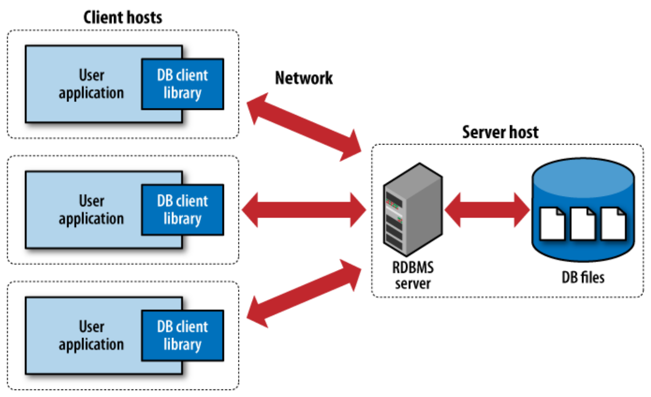
- Use getExternalFilesDir() to open a File representing the external storage directory
- Method requires a parameter that specifies the type of sub-directory you want, such as: Environment.DIRECTORY_MUSIC and Environment.DIRECTORY_RINGTONES (null to receive the root of your application directory)
- This method will create the appropriate directory, if necessary

SQLite Database

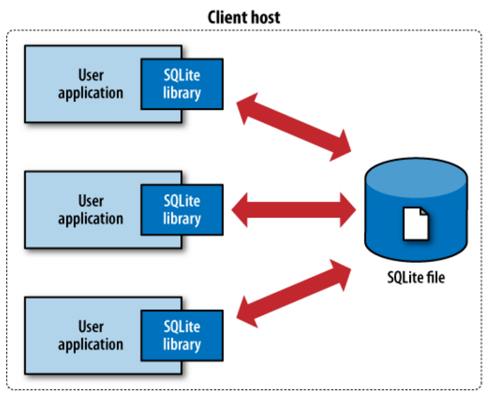
SQLite is a open source SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation. SQLite supports all the relational database features. In order to access this database, we don't need to establish any kind of connections for it like JDBC,ODBC e.t.c Database - Package

- Supports all the relational database features and available in android.database.sqlite
- Written in C, supports cross-mobile platform , configure it with less than 250 Kbs
- SQLite transactions are fully ACID(Atomicity, Consistency, Isolation, Durability)compliant
- Databases are stored in the /data/data/<package-name>/databases directory.
- Advantages
    - light weight database
    - Requires very little memory
    - Automatically managed database

The following figure shows how sqlite differs with normal database



(a) Traditional client-server architecture



(b) SQLite serverless architecture

The main package is android.database.sqlite that contains the classes to manage your own databases.

Database - Creation

- android.database.sqlite.SQLiteOpenHelper class is used to manage database creation.

constructor	
SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)	*public abstract void onCreate(SQLiteDatabase db)*
SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version, DatabaseErrorHandler errorHandler)	*public abstract void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)*
	public synchronized void close ()
	public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion)

SQLiteDatabase class is used to perform actions on database. It has the following methods

Methods
void execSQL(String sql)
long insert(String table, String nullColumnHack, ContentValues values)
int update(String table, ContentValues values, String whereClause, String[] whereArgs)
Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy
Int delete(String table, String whereClause, String[] whereArgs)
static boolean deleteDatabase(File file)
openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorhandler)

An alternative way of opening/creating a SQLITE database in your local Android's data space is given below:

We call this method openOrCreateDatabase with your database name and mode as a parameter. It returns an instance of SQLite database which you have to receive in your own object. Its syntax is given below

SQLiteDatabase mydatabase = openOrCreateDatabase("your database name",MODE_PRIVATE,null);

Apart from this, there are other functions available in the database package , that does this job. They are listed below

openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler)	This method only opens the existing database with the appropriate flag mode. The common flags mode could be OPEN_READWRITE OPEN_READONLY
openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)	2 It is similar to the above method as it also opens the existing database but it does not define any handler to handle the errors of databases
openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)	It not only opens but create the database if it not exists. This method is equivalent to openDatabase method
openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)	This method is similar to above method but it takes the File object as a path rather then a string. It is equivalent to file.getPath()

Database - Insertion

We can create table or insert data into table using execSQL method defined in SQLiteDatabase class. Its syntax is given below

mydatabase.execSQL("CREATE TABLE IF NOT EXISTS TutorialsPoint(Username VARCHAR,Password VARCHAR);");
mydatabase.execSQL("INSERT INTO TutorialsPoint VALUES('admin','admin');");

This will insert some values into our table in our database.

Another method that also does the same job but take some additional parameter is given below

execSQL(String sql, Object[] bindArgs)

This method not only insert data , but also used to update or modify already existing data in database using bind arguments

Database - Fetching

We can retrieve anything from database using an object of the Cursor class. We will call a method of this class called rawQuery and it will return a resultset with the cursor pointing to the table. We can move the cursor forward and retrieve the data.

Cursor resultSet = mydatbase.rawQuery("Select * from MRCET",null); resultSet.moveToFirst();

String username = resultSet.getString(0); String password = resultSet.getString(1);

getColumnCount()	This method return the total number of columns of the table
getColumnIndex(String columnName)	This method returns the index number of a column by specifying the name of the column
getColumnName(int columnIndex)	This method returns the name of the column by specifying the index of the column
getColumnNames()	This method returns the array of all the column names of the

	table
getCount()	5 This method returns the total number of rows in the cursor
getPosition()	This method returns the current position of the cursor in the table
isClosed()	returns true if the cursor is closed and return false otherwise

There are other functions available in the Cursor class that allows us to effectively retrieve the data.

Database - Helper class For managing all the operations related to the database , an helper class has been given and is called SQLiteOpenHelper. It automatically manages the creation and update of the database. Its syntax is given below

```
public class DBHelper extends SQLiteOpenHelper
 {
 public DBHelper()
{
 super(context,DATABASE_NAME,null,1);
}
 public void onCreate(SQLiteDatabase db)
 {}
public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion)
 {}
 }
```

Enterprise Data Access

It provides a layer of control for asset owners through a data management functionality to authenticate and authorize access to every business asset Enterprise Data Access refers a set of processes and activities focused on data accuracy, quality, security, availability, and good governance.

The enterprise systems expose specific functionalities, and in turn related underlying data, to serve the client apps. These functionalities are typically exposed using Web services; RESTful2 Web services are popular for mobile clients. Sheer simplicity, light-weight approach, and support for simple CRUD operations have resulted in the popularity of RESTful Web services.

The data between the mobile app and the enterprise app can be exchanged in several formats; JSON3 (JavaScript Object Notation) is a popular format for exchanging small chunks of data in key–value pairs.

Accessing data over the network requires an app to request android.permission.INTERNET permission. The app also needs to request android.permission.ACCESS_NETWORK_STATE permission to check network connectivity by accessing network state of the device. The following Snippet enlists the mechanism to check the network connectivity using a user-defined method checkNetworkAccess(). A ConnectivityManager instance is obtained by requesting the CONNECTIVITY_SERVICE (Line 2). Its getActiveNetworkInfo() method (Line 3) provides network information, using which network connectivity can be determined.

```
private boolean checkNetworkAccess()
 {
        ConnectivityManager connectivityManager =
            (ConnectivityManager)getSystemService(CONNECTIVITY_SERVICE);
        NetworkInfo info = connectivityManager.getActiveNetworkInfo();
        if (info != null && info.isConnected())
        {
            return true;
        } else {
            Toast.makeText(MainActivity.this, "No network access, network
            resource not accessible", Toast.LENGTH_SHORT).show();
            return false;
            }
}
```

Once the network connectivity is determined, the app needs to initiate an HTTP (Hypertext Transfer Protocol) request to exchange data with RESTful Web service. Android recommends an HttpURLConnectionAPI to initiate HTTP requests. The HttpURLConnection API facilitates CRUD operations using PUT, GET, POST, and DELETE, HTTP methods.

Snippet 6.20 demonstrates fetching expense data using the GET HTTP method from the RESTful Web service. An HttpURLConnection instance is created using the RESTful Web service URL (Lines 1–4), and its connection parameters are configured (Lines 5–7). To refer to the localhost from an Android emulator, IP address 10.0.2.2 is used (Line 3). The setRequestMethod() method sets the HTTP request method to GET (Line 7). The setReadTimeout() method is used to set the maximum time that a client can take to read the response from the Web service, and is set to 2s in this case (Line 5). The setConnectTimeout() method is used to set the maximum time within which a client has to establish the connection, and is set to 4s (Line 6). The connect() method is used to establish the connection, and make the HTTP request to fetch expense data (Line 8).

```
HttpURLConnection connection = null;
 try {
        URL url = new
URL("http://10.0.2.2:8080/ExpenseTrackerWebService/FetchExpensesServlet");
        connection = (HttpURLConnection) url.openConnection();
        connection.setReadTimeout(2000);
        connection.setConnectTimeout(4000);
        connection.setRequestMethod("GET");
        connection.connect();
        int responseCode = connection.getResponseCode();
        if (responseCode == 200) {
            InputStream inputStream = connection.getInputStream();
            BufferedReader bufferedReader = new BufferedReader(new
        InputStreamReader(inputStream));
            StringBuilder builder = new StringBuilder();
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                builder.append(line);
            }
            response = builder.toString();
        }else {
```

```
response = “Response was not successful”;
}
```

Once the connection is established, and response is fetched successfully (HTTP status code: 200), the response is retrieved using the getInputStream() method (Line 11). This input stream is read through, and converted into a String object.

readJsonStream() method of a user-defined class ExpenseParser is used to obtain the list of individual expense data items.