./



|  |  |  |
| --- | --- | --- |
| **Name** | **PS Number** | **Email ID** |
| R PAVAN RAJ | 99002460 | rpavan.raj@ltts.com |

**Table of Contents**

[**Day -1**](#_3fcldiwkg9k) **5**

[**Learnings:**](#_askrlegxcpl1) **5**

[Overview of Android Architecture:](#_sx9uznv2pdnh) 5

[Linux kernel](#_co6sieau5xs2) 6

[Libraries](#_hdfga59gl0r8) 6

[Android Libraries](#_puoq5gg311yy) 6

[Android Runtime](#_kwpbumz3q0qd) 7

[Application Framework](#_430f0amqgw0y) 7

[Applications](#_wqb38h3oz336) 7

[**Summary:**](#_2dk84d32qv68) **8**

[**Day - 2 :**](#_5un8a1nvbunt) **8**

[**Learnings:**](#_j9hfiepd9g3d) **9**

[Android Services:](#_yktlooa8bt1) 9

[Case studies for Services Apps:](#_wjj7l1p8vq1x) 12

[Steps for implementing services:](#_vsbfm4md8o64) 13

[Examples of services by AOSP:](#_6howqpxuooyc) 13

[Services LifeCycle Operations:](#_y82b3gvj9ch7) 14

[**Summary:**](#_1t403ekojg85) **19**

[**Day - 3 :**](#_fandzq24vz6l) **19**

[**Android Local Inter Process Communication(IPC):**](#_5z8f4jhp9c0a) **20**

[Communicating from Services to Activities:](#_kgo3u4gkf2y7) 21

[Android Handlers:](#_pk21ehagal72) 21

[Android Messengers:](#_ce15dipb31ve) 23

[Android Messengers on Bound Services:](#_gvfnrhpovit8) 24

[**Content Providers:**](#_oioz94sz37yz) **27**

[**Content Resolvers:**](#_dg6wih2v9z10) **28**

[**Content URIs:**](#_f59i1nmksh3j) **31**

[**Android SQLite:**](#_nr5v6jihh12a) **31**

[**SQLite-related Classes**](#_outs0q72ftrc) **32**

[**Developing Content Provider**](#_qxyyl6bd85i) **34**

[**Summary:**](#_pc4fj3vutkpy) **36**

# Day -1

**DAY -1**

# Learnings:

## Overview of Android Architecture:

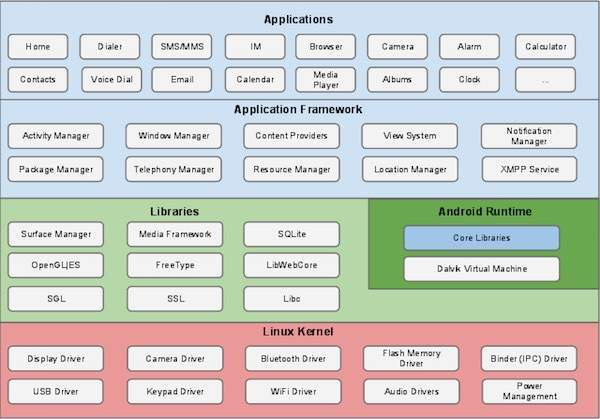


Fig 1 : Android Architecture

Android framework is a pile of programming segments which is generally partitioned into five areas and four principal layers as appeared above in the figure.

## 

## 

## Linux kernel

At the lower part of the layers is Linux - Linux 3.6 with around 115 patches. This gives a degree of reflection between the device drivers and it contains all the basic equipment drivers like camera, keypad, show and so on Additionally, the kernel handles all the things that Linux is great at, for example, organizing and a huge range of drivers, which remove the agony from interfacing to fringe equipment.

## Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

## Android Libraries

This class envelops those Java-based libraries that are explicit to Android improvement. Instances of libraries in this classification incorporate the application system libraries notwithstanding those that encourage UI building, illustrations drawing and information base access. A rundown of some key center Android libraries accessible to the Android engineer is as per the following −

* app − Provides access to the application model and is the cornerstone of all Android applications.
* content − Facilitates content access, publishing and messaging between applications and application components.
* database − Used to access data published by content providers and includes SQLite database management classes.
* opengl − A Java interface to the OpenGL ES 3D graphics rendering API.
* os − Provides applications with access to standard operating system services including messages, system services and inter-process communication.
* text − Used to render and manipulate text on a device display.
* view − The fundamental building blocks of application user interfaces.
* widget − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
* webkit − A set of classes intended to allow web-browsing capabilities to be built into applications.

## Android Runtime

This is the third part of the engineering and accessible on the second layer from the base. This part gives a key segment called Dalvik Virtual Machine which is a sort of Java Virtual Machine uncommonly planned and improved for Android.

The Dalvik VM utilizes Linux center highlights like memory, the board and multi-stringing, which is inborn in the Java language. The Dalvik VM empowers each Android application to run in its own cycle, with its own occasion of the Dalvik virtual machine.

The Android runtime additionally gives a lot of center libraries which empower Android application engineers to compose Android applications utilizing standard Java programming language

## Application Framework

The Application Framework layer gives numerous more significant level administrations to applications as Java classes. Application engineers are permitted to utilize these administrations in their applications.

The Android system incorporates the accompanying key administrations −

Activity Manager− Controls all parts of the application lifecycle and action stack.

Content Providers − Allows applications to distribute and impart information to different applications.

Resource Manager − Provides admittance to non-code installed assets, for example, strings, shading settings and UI designs.

Notification Manager − Allows applications to show alarms and notices to the client.

View System − An extensible arrangement of perspectives used to make application UIs.

## Applications

All the Android applications are at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

# Summary:

The overviewed learning of android architecture is covered in week -1 contents of the coursera. The Linux kernel , middleware and android services were learnt. Theoretical part of the course is completed. Week-2 contents have hands-on coding questions.

# Day - 2 :

**DAY - 2**

# Learnings:

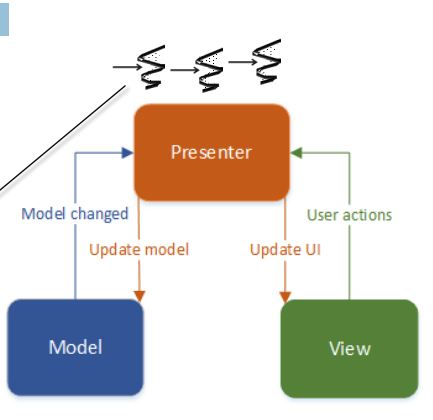
## Android Services:

Limitations of activities.

* Activities are short lived. They are used to handle runtime changes.
* Activities are recreated when later when the back button is pressed.
* Activity is paused when new activity is started.

MVP frameworks:

* The Model-View-Presenter (MVP) pattern alleviates some—but not all—of these limitations.
* Android MVP frameworks separate background threads from activities to make them independent of most lifecycle-related events.



Android needs other components for background process other the MVP Frameworks as it is not available in

the android framework.

Examples of services:

1. Music player in the background.

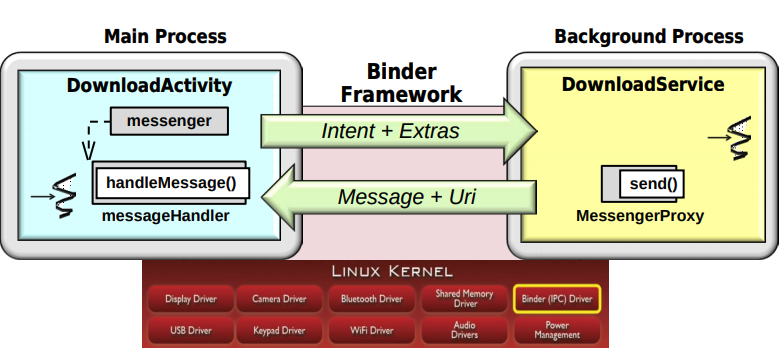
2. Sync of contents from DataBase in background like email, calendar etc

3. Download and store files in the background like .apk files from playstore.

* Unlike activities, services only interact with the user in very limited ways.They can’t use the UI directly.
* Eg. Status Bar notifications allow background components to interact with the user.
* Services must interact with the user through activities connected via local communication mechanisms provided by the **Binder framework.**
* Services are typically launched by **activities using intents &** Services can run in the **background.**

**Types of Android services:**

1. Started services
2. Bound services
3. Scheduled services.



**1. Started services:** are launched via startService() method and initialized by “virtual constructor” i.e, onCreate().

* Parameters are passed through intents to start a service.
* Parameters can be passes via the intent used to start the service, and to pass into a method, or on start command which is called every time a client component called startService.
* For example, a URL to download can be passed as data to the intent, or list of email addresses can be passed as extras to the intent.
* Often, a started service is used to perform a single operation. For example, it could be used to download one image or play one song, etc.
* A started service need not return a result to the client. Although it's possible to return a result from a started service In which case, this result is typically returned via some kind of messenger, or broadcast intent.
* Started services stops when its done with the client requests is finished.
* Started services can be explicitly stopped using the stopService() method.

**2. Bound Service:**

* A bound service is launched by the bindService method, and it has several capabilities that are somewhat different than a started service.
* For example, the service implementor has to implement the on bind hook method, which is used to perform a handshake between the client and the service that sets up a so called connection for exchanging information.
* This connection can be used to exchange in messages via so called IPC channels.
* Exchanging these messages can be used to allow extended conversations to take place between clients, and the service where a client would send a request, get a result back. That client, or another client could send another request, get a result back from the service, and so on.
* These requests and results can be interactive via two ways. Synchronous, or asynchronous messages, or remote method indications, all of which are supported by the Android binder IPC framework, which you can learn more about at the link at the bottom of this slide.
* One of the key differences between a started service and a bound service is that bound services are automatically destroyed when all clients that are bound to them, call unbind service.
* In other words, bind service calls to a service, are reference counted. And when that reference count drops to zero, the Android activity manager framework goes ahead and cleans up the service by calling it's on destroy hook method. Note, however, that a so called hybrid started and bound service is not automatically destroyed.

**3. Hybrid Service:**

* A hybrid service that combines the characteristics of both started and bound services. And the way this works is by having the bound service implement the onStartCommand.
* A started service implements the onStartCommand to receive the intent from a client. But a bound service typically does not implement the onStartCommand, and instead, implements the on bind hook method instead.

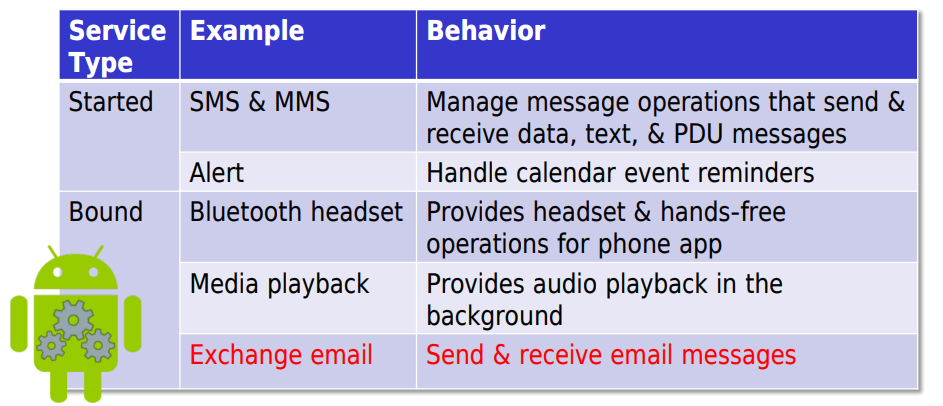
## Case studies for Services Apps:

1. **Music player app:** The MusicPlayer app prompts the user for a song URL, and then uses an intent and a started service to start or stop playing the song. You can take a loop at the source code for this app at the link at the bottom of the slide.
2. **Ping pong receiver app:** This app has a main activity that prompts the user for a count, and then plays ping pong by passing intents between a dynamically registered broadcast receiver and an intentservice.

## Steps for implementing services:

1. Extend the services class
2. Overriding the lifecycle hook methods: onCreate, onStartCommand, onBind,onUnbind, onDestroy. Key difference between a started and bound service is the started services don't have explicit connection whereas bound services have.
3. Define other methods and nested classes needed to implement the service.
4. Update the AndroidManifest.xml file to include the service so that android knows about it. Service declaration is important for this.

## Examples of services by AOSP:

****

* Started Services :

1. SMS & MMS
2. Alert

* Bound services:
  1. Bluetooth headset
  2. Media Playback
  3. Exchange email.

## Services LifeCycle Operations:

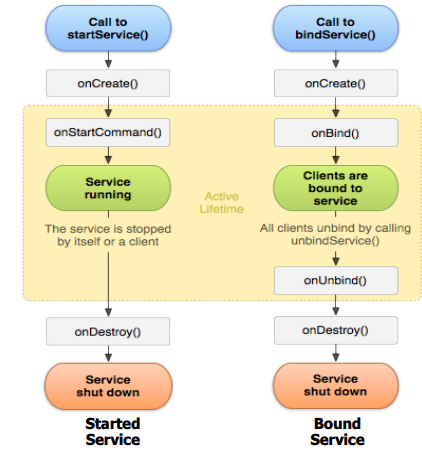
* Started Services lifecycle states:

1. Service starting
2. Service running: processing intents
3. Service shutdown: voluntarily finished, involuntarily shutdown or killed by android for memory management

* Bound Service lifecycle states:

1. Service starting: onCreate and onBind().
2. Running with client or clients.
3. Service shutdown.

Operations on Started and Bound services:



1. **onCreate()** method is used in order to perform initialization. It's essentially a virtual constructor that's called after the service is first launched. It's implemented by both started and bound services.

It's typically used to initialize the service. For example, it might be used to allocate binder objects that

are used for local communication. It may also allocate handlers to process various requests

concurrently, depending on what concurrency model you may choose. This is particularly important

for starting services. And it might be a no-op if there's really nothing to initialize. For example, if the

fields themselves that are defined in the service implementation perform the initialization, then

onCreate is often really not doing anything at all.

1. **OnStartCommand()** is the next method that's called back. Unlike onCreate(), which is only called once every time a service is launched, onStartCommand() is called every time a startService is sent an intent via startService. And as you can see here, onStartCommand is something that's only relevant for started services, or for hybrid started and bound services. But we won't consider that case any further. OnStartCommand() receives the intent that was passed when the client called startService(). And, this intent is often then used in conjunction with the service's concurrency model, in order to be able to carry out the processing on the intent. OnStartCommand() returns a flag that indicates the semantics to use if the service crashes or is killed. And this flag is actually passed back to the Activity Manager service. It's not passed back to the client, it's passed back to the Activity Manager Service. That interprets the flag and uses it to indicate how to manage that service henceforth.
2. **onBind()** hook method. This is what's called back during the initialization phase, when we're in the starting state. And essentially, onBind is the factory method that's called when the client connects via bindService. It receives the intent past by the client call to bind service. Interestingly enough, any extras that are included in the intent are actually not seen here. They are stripped away by Android.In contrast, a started service can pass extras with the intent that's past to start service. And the reason the way it works like this is because on bind is typically used as part of a bounce service to have a conversation between the client component or the activity, typically, and the Bound Service. And they use other means to pass information back and forth.OnBind returns a reference to a binder object that the client will use later to communicate back to the Bound Service. And this is typically a binder object that's either a messenger or an AIDL object that's used for local inter-process communication between processes in Android. OnBind() must return null if a service is a started service. That's how the Android's activity manager framework knows that something is a started service, if onBind() returns null.
3. **OnUnbind()** is another hook method that gets called back automatically by the activity manager service. This method is called when all the clients have disconnected. So when all the clients have finally called onBind service, the onUnbind hook method gets called back. And by default, this returns the value of false. So if you just want it to act like a normal bound service just keep the default the way it is. You can return true however, if you want to make a hybrid bound and started service that allows new clients to bind later to the service via the rebound method. So this is kind of the unusual hybrid use case we were talking about earlier.
4. **onDestroy()** hook method is really the virtual destructor. It's a disposal method that's called when a service is shut down for various reasons. We'll see in a second It's typically used to clean up resources that were allocated by the service such as shutting down the thread pool in the case of a started service or deallocating some kind of handler, if that's necessary.

**Steps for starting a service and able to name the methods used to start services and**

**integrate a service into an app:**

* A service can be launched on-demand via an intent, just like an activity.
* For example, the MusicPlayer app we described can be used to play a song at a given URL. The user is prompted to enter the song URL and then they can click on a floating action button.
* Next, MusicActivity creates an intent.This intent is an explicit intent that indicates the song URL as data, and the MusicService class as the target of the intent.
* The client then calls startService, which passes this intent and the data to the Android Activity Manager Service. The Activity Manager Service in turn passes the explicit intent to the component that's registered to handle it.
* In this particular case, assuming the MusicService wasn't already up and running, the Activity Manager Service will create and start a process containing the MusicService class component implementation.
* Creation of a new process to handle the service is controlled by configuration information in the AndroidManifest.xml file.
* At this point, control then passes to the Music Services onCreate hook method and the process carries from there.
* An intent can be passed using one of two methods.

The startService method can be used to launch a new started service,whereas the bindService method can

be used to connect to a bound service.Which of these methods to choose depend largely on whether there's

a need to have an extended conversation with the service.If there's a need for a conversation,

in other words an exchange of messages back and forth for some period of time,then the bindService

method and the bound service is probably what you want to use.

1. The startService is an asynchronous one-way operation.In other words, it returns immediately and the

caller continues executing its next operation while the service is started and the intent is delivered.By

default, a started service returns no result.If you want to return results via started services, of course you can

use various IPC mechanisms, such as messengers or broadcast receivers to get the results.

2.The bindService method can be used to connect to a bound service. The bindService method is a little

different from the startService method in the sense that it's actually a two-way asynchronous method.What

that means is again it doesn't really block the However, there's going to be a result coming back from the

server,from the bound service, via something called a ServiceConnection callback.This ServiceConnection

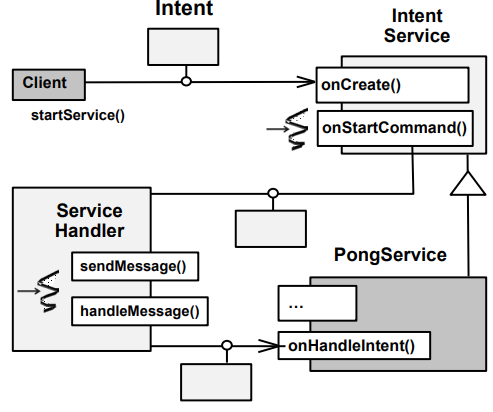
callback that comes back from the unbind method can be used to have the conversation between the client

and the bound service. Note that bound services can return the results via either messengers or objects that

implement AIDL interfaces.

**Intent Service Framework structure and functionality**

A multi-threaded assistance must be modified as opposed to designed. • e.g., start a Handler Thread, execute an undertaking in the Java Executor structure, and so on Growing impromptu simultaneous administrations can be repetitive and mistake inclined. We accordingly need some approach to robotize and streamline the programming of simultaneous began administrations. The IntentService system is a simple method to make a simultaneous help.



Clients send intents via calls to startService(). Data can be passed with the intent. e.g., via setData() or by putting “extras” into intent. IntentService is started on-demand by the Activator pattern. An IntentService subclass implements the onHandleIntent() hook method. It processes the intent in a Handler Thread. This thread is different from the main thread

HaMeR framework dispatches onHandleIntent() hook method. The PongService app overrides onHandleIntent() to do two things. 1. Update the status bar 2. Send intent to PingReceiver

**IntentService Usage Considerations**

Improves simultaneous treatment of orders as intents. Simply supersede onHandleIntent() to deal with a purpose simultaneously. No compelling reason to deal with runtime setup changes. android:process can be utilized to provide assistance to run in another cycle. In any case, utilizing a different cycle might be pointless excess.

A key IntentService downside is that just a single goal can be handled simultaneously. This cutoff points versatility on multi-center stages. Administrations that utilization string pools are more adaptable, yet harder to close down.

**Android Service Deployment Models**

A deployment model guides the physical organization of administrations to measures. Begun and bound administrations can run in the equivalent or various cycles as their customer parts. This decision is resolved by means of an AndroidManifest.xml document setting. The default conduct is to run a help in a similar cycle.

An application with help arranged to run in a different cycle is essentially simultaneously. Each cycle has a different string of control as a matter of course. There are a few explanations behind running an assistance in its own cycle, for example An assistance that is shared by more than one application needs to run in a different cycle. Running an assistance in its own location can make applications more strong. Trash assortment in a different help measure doesn't influence different cycles in an application.

# Summary:

1. Understood the need for android services
2. Types of Android services.
3. Examples of services.
4. Brief Introduction to Started, Bounded and Hybrid Services.
5. Service Life Cycle Operations on Started and Bounded Services.

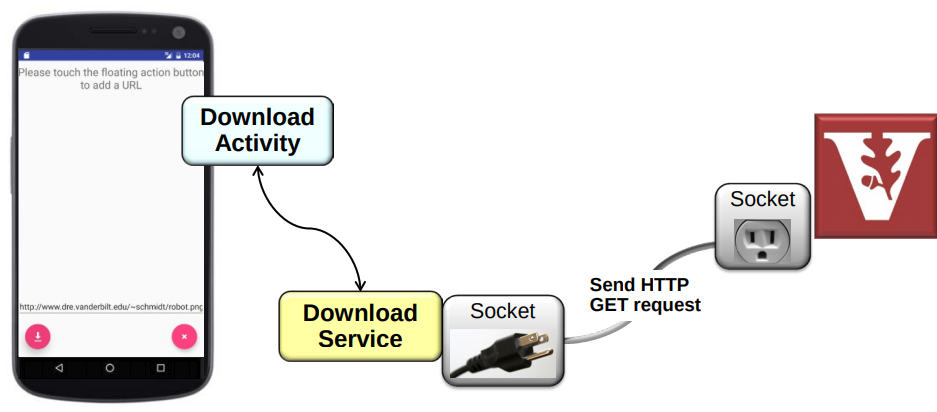
# Day - 3 :

Day 3

# Android Local Inter Process Communication(IPC):

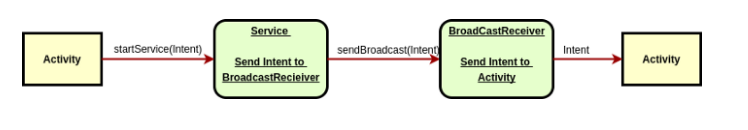
* Inter-process communication (IPC) mechanisms enable clients & servers to exchange data across address spaces.
* IPC mechanisms are needed because apps running in separate processes can’t directly access each other’s data
* The Android Linux kernel supports a range of IPC mechanisms.
  + e.g., Internet & Unix-domain sockets & the Binder driver.
* Binder IPC mechanisms Android provides to communicate between components.

Example for Applying Local IPC Mechanisms:



**Communicating from Activities to Services:**

* Activities can select several IPC mechanisms to interact with services, e.g.
  + Send an intent to a started service via startService()
  + Bind to a bound service via bindService()
  + Call send() on a reference to a messenger to pass a message .
  + A messenger encapsulates a handler implemented within a service.
  + Enables passing messages to a handler across process boundaries.
  + Invoke method calls on proxies generated by the AIDL compiler.
* Internally, Android’s Binder IPC framework is used to communicate messages (& methods encapsulated within messages) across processes.



## Communicating from Services to Activities:

* Services use IPC mechanisms to reply to activities that communicate with them, e.g.
* Use a messenger to send messages back to activity
* The messenger is created by the activity & a reference to it is passed to the service
* Use an AIDL object reference passed from activity to service
* Use a broadcast intent reply back to activity

## 

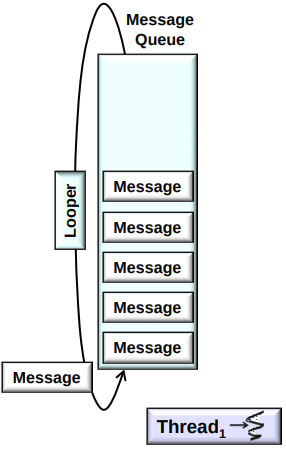
## 

## 

## 

## 

## Android Handlers:

****

A handler can be used to send & process messages & runnables in one or more threads in a process.

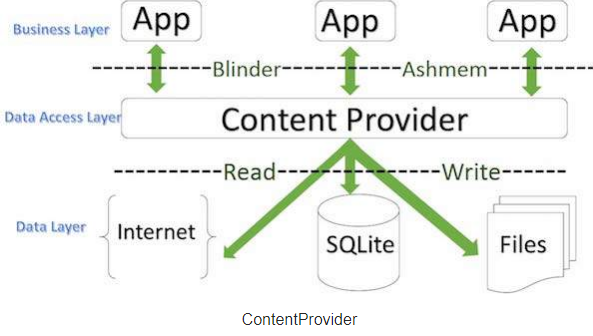
* A handler is a key part of Android’s “Handler, Messages, & Runnables” (HaMeR) framework.This framework contains a looper associated w/a message queue. The looper processes messages & runnables placed on queue via handlers by one or more threads
* A handler does two main things

1. Sends messages and/or posts runnables to thread’s looper :

A message contains meta-data & fields passed between threads. A runnable is a command that can be executed in another context. The looper’s message queue enqueues & schedules them for subsequent execution.

2. Collaborates with looper to serialize processing of messages in a thread:

Each message/runnable tracks its associated target handler. Calling sendMessage() on a handler triggers the handleMessage() hook method. Calling post() on a handler triggers the run() hook method on the runnable.



• Android handlers often eliminate the need for apps to use synchronizers, as long as design rules are followed , e.g., always use the HaMeR or AsyncTask frameworks

• Objects in different threads within a process can interact via their handlers , e.g., exchange messages & runnables via handlers that are associated with each other's loopers

Limitations of Android Handlers:

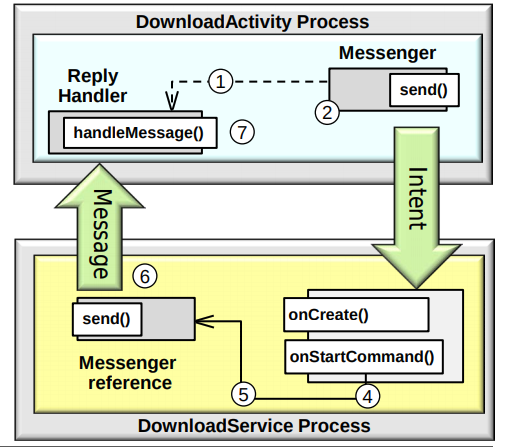
* Handlers don’t implement the Parcelable interface.
* They thus can’t be passed as data in a message or as an “extra” in an intent
* To overcome limitations with handler, Android provides the Messenger class

## Android Messengers:

* A messenger encapsulates a handler in a 1st component , e.g., an activity
* A messenger reference can be passed to a 2nd component in another process ,e.g., a service.
* 2nd component gets messenger reference from the intent
* 2nd component use reference to send messages to handler

**Android Messengers on Started Services:**

Started services follow a common architecture when performing IPC with an activity via a messenger.



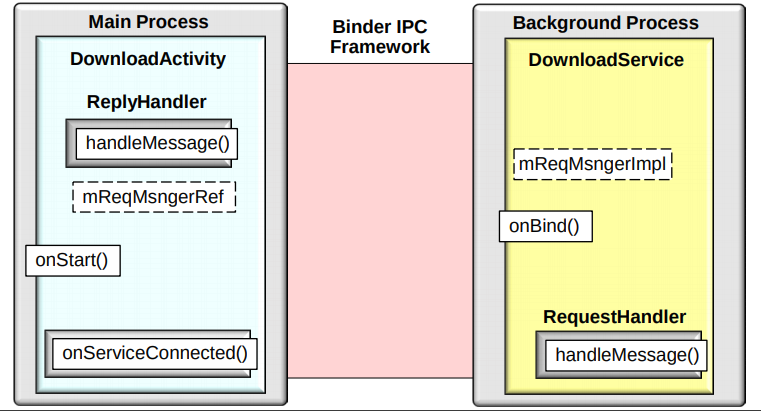
A started service often does three things when it receives an intent.

1. Obtains reference to messenger from the intent, e.g., extracts it from the “extra” placed in the intent by the activity.

2. Performs some processing, e.g., retrieve an image from a remote server

3. Returns the results back to the sender process via a message, e.g., service uses reference to the messenger to send image pathname back to the activity.

## Android Messengers on Bound Services:

****

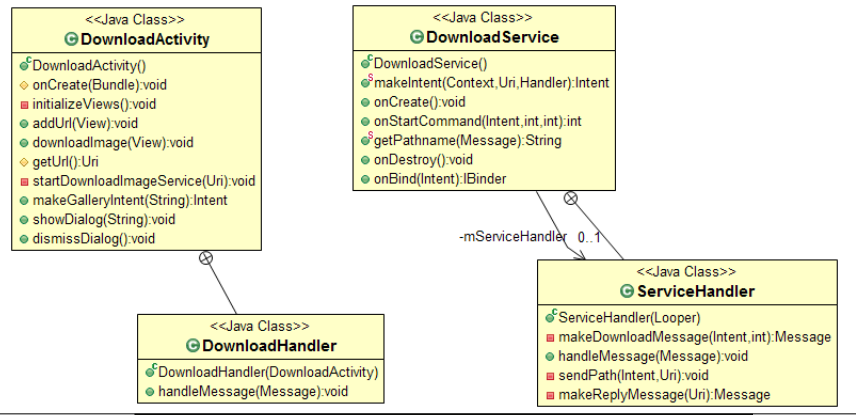
A bound service can contain a request messenger that encapsulates a handler. Likewise, an activity can contain a reply messenger that encapsulates a handler. DownloadService creates a request messenger & request handler. DownloadActivity initiates a connection to the DownloadService

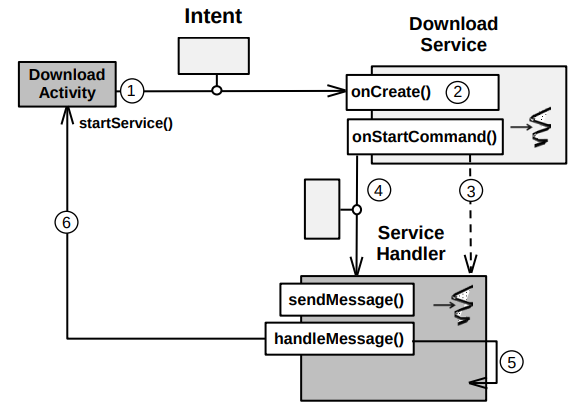
DownloadService obtains & returns a reference to the request messenger. DownloadActivity receives the messenger reference via a callback hook method.DownloadActivity stores the messenger reference locally for future use.

**Programming Started Services with Intents & Messengers:**

Description: This app uses a messenger to communicate from a started service back to an activity.The download activity enables a user to download and display a bitmap image via the DownloadService. The DownloadService in turn, downloads and stores a bitmap image on behalf of the DownloadActivity.It has a nested class called server handler, that processes messages received from the download activity that indicate which image, or images to download, store, and return.A DownloadHandler is a nested class inside of DownloadActivity. It inherits from Handler, and displays the images retrieved and stored by the DownloadService.

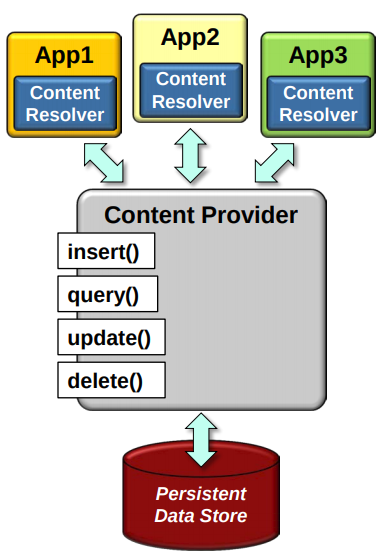
Functionality:



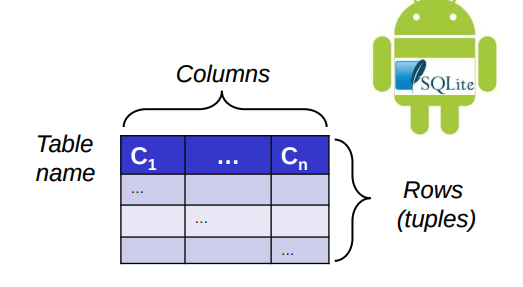


# Content Providers:

* Content Providers manage access to a central repository of structured data.
* They support database “CRUD” (Create, Read, Update, Delete) operations
* Apps always access Content Providers via Content Resolvers
* A Content Resolver is a proxy that shields apps from details of how Content Providers are accessed.

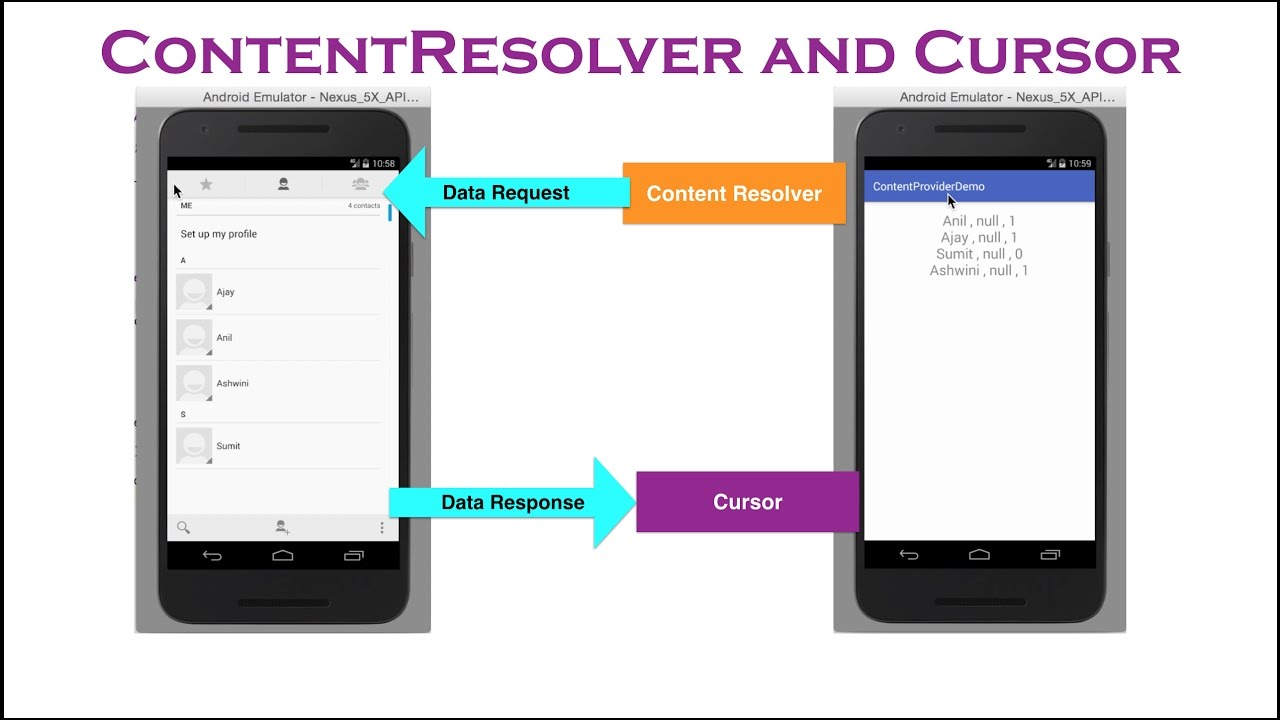


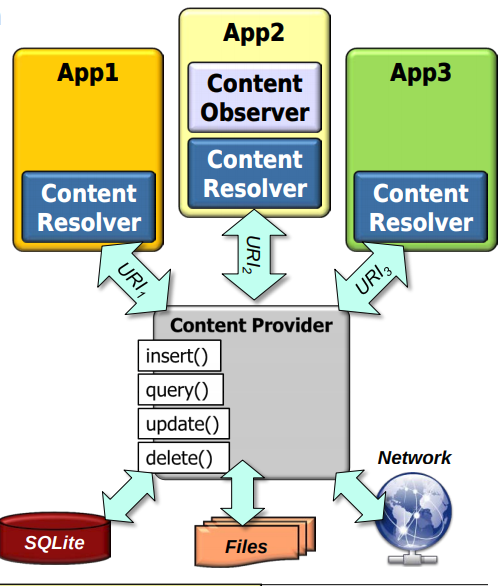
* E.g., a Content Provider could reside in another process or in the same process as an app
* Content Providers/Resolvers can make an app’s data available to other apps
* Together they encapsulate data & provide mechanisms for defining data security
* A content provider typically manages data that’s stored persistently
* E.g., relational database tables implemented via SQLite.



* Android includes many Content Providers,
* E.g. Browser – bookmarks, history , Call log – telephone usage , Contacts – contact data , MMS/SMS – Stores messages sent & received , Maps – previous searches , YouTube – previous searches & many more.
* Many Android features depend on Content Providers
* E.g., CursorLoaders, SyncAdapters, & notifying the apps of changes in a database.

# Content Resolvers:

* A Content Provider is accessed via a Content Resolver proxy
* A Content Provider isn’t created until a Content Resolver tries to access it.
* Uses the Activator pattern
* Content Resolvers manage & support Content Providers
* Enables Content Providers to be shared by multiple Apps
* Provides additional services, such as change notification using the Observer pattern
* Abstracts away from the details of how the data is stored
* e.g., data can be stored in an SQLite database, flat files, on a server accessed over the Internet, etc.



• Use ContentResolver.insert() to insert a single data item Uri insert(Uri uri, ContentValues values)

• Inserts a row into a table at the given URI

• If the content provider supports transactions the insertion will be atomic

• Implementation should call notifyChange() after inserting Parameters

• uri – uri of the table to insert into

• values – initial values for the newly inserted row, where the key is the column name for the field (passing an empty ContentValues will create an empty row)

Returns : The URI of the newly created row

# Content URIs:

* Any URI that begins with content:// scheme represents a resource managed by a Content Provider.
* e.g., content://authority/path/id
  + content - data is managed by a ContentProvider
  + authority – id for the content provider
  + path – 0 or more segments indicating the type of data to access
  + id – specific record being requested

# 

# Android SQLite:

*SQLite* is an Open Source database. SQLite supports standard relational database features like SQL syntax, transactions and prepared statements. The database requires limited memory at runtime (approx. 250 KByte).

SQLite supports the following data types:

* TEXT(similar to String in Java)
* INTEGER(similar to long in Java)
* REAL (similar to double in Java). All other types must be converted into one of these fields before getting saved in the database. SQLite itself does not validate if the types written to the columns are actually of the defined type. This means you can write an integer into a string column and vice versa.
* Access to an SQLite database involves accessing the file system. This can be slow. Therefore it is recommended to perform database operations asynchronously.

Android supports SQLite, which provides a relational database for mobile devices.i.e., it contains tables (consisting of rows & columns), indexes, etc. that form a “schema”.

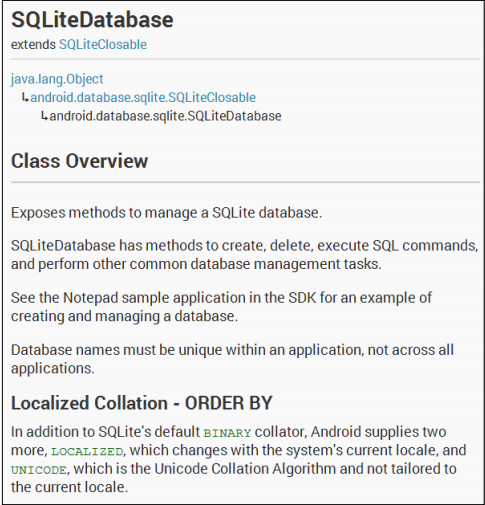
It operates in a small footprint (~350kB) within a single cross-platform disk file, Every Android device has SQLite. You only need to define SQL statements for creating & updating the database.

* Implements most of SQL92 & supports so-called “ACID” transactions • “ACID” == “Atomic, Consistent, Isolated, & Durable.
* Accessing an SQLite database typically involves accessing Android’s filesystem, Often used concurrently and/or asynchronously since filesystem accesses may be slow.

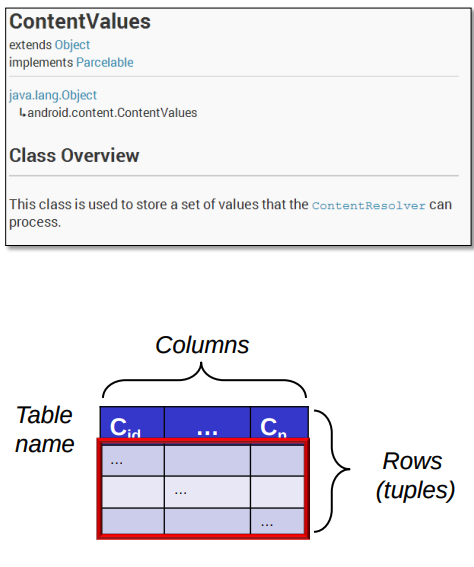
# SQLite-related Classes

# 

* SQLiteDatabase is the base class for using a SQLite database.



* It provides the insert(), update(), & delete() methods.
* It also provides the execSQL() method that can execute an SQL statement directly
* Queries can be created several ways : The rawQuery() & query() methods



* The SQLiteQueryBuilder class :

ContentValues defines key/values used for inserts & updates of database row(s), “Key” represents table column ID , “Value” represents the content for table record in this column.

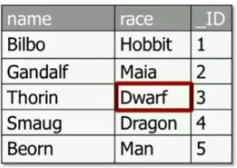
# Developing Content Provider

Steps for creating content provider

1. Analyze requirements to see if the content provider is necessary.

Consideration for creating Content Provider are:

* Want to use complex files or data
* Want to decouple common data access API from the data storage mechanism
* Want custom search suggestion
* Want to synchronize local data with remote data
* Want to notify activities automatically when data changes



2. Design the data model

The data model will contain data in the form of tables. In that table each column represents type of

data and each row provides instance of some type of data and each column in row represents single data

piece.

3. Define contrast class

\*Contract class is public final class containing constant definitions. This class establishes contract

between the provider and other apps to ensure the provider can be correctly accessed even if actual

values change.

\*Column names will be identical to SQL database column names which can be given as public static

string constants. Also should define the ID of integer type.

\*Define MIME types for items and directories.

ContentProvider.getType () returns a string in MIME format.

4. Determine the format of content URI to access managed data

* The content URI identifies data in a provider. There are several parts in Content URIs some of

them are:

* The symbolic name of the complete provider. It is basically giving authority to users.
* Name should be created to point to a table or file. It is providing path
* An optional ID can be provided to point to individual rows of tables.
* Defines the unique data members that represents each content URI part
* The UriMatcher class maps content URI “Pattern” to integer values using wildcard

characters.

5. Implement a provider as one or more classes, along with <provider> element in manifest file

* The subclass ContentProvider to define the interface between provider and other logic. Also
* implement the storage system for data for example files.
* OnCreate () initializes a provider (it is called immediately after creating provider)
* Insert () selects the table &amp; column values to use to insert new column or row
* Query() selects table to query, rows &amp; columns to return via cursor or sort order of result
* Update() selects table or rows to update
* Delete() selects table and row to delete
* getType() returns MIME type corresponding to content URI

6. Implement Change Notification

Content provider should notify registered observers when data for given content URI has

changed. The content resolver implements the observer pattern to avoid requiring Apps to poll

for updates to data storage.

7. Declare Provider in AndroidManifest.xml

Declare ContentProvider with <provider> in AndroidManifest.xml

Authorities attribute omits the path of content://URI

# 

# 

# Summary:

1. The motivations for Android's local inter-process communication, or IPC, mechanisms.
2. Recognize how local IPC mechanisms can be applied in practice via image downloader case study apps
3. Android IPC mechanisms & frameworks that activities & services use to communicate between threads in separate processes.
4. Role of content providers in android architecture.
5. Significance of content resolver.
6. Key operators of content resolver.
7. Why android support SQLite and related android classes.
8. Steps involved in developing content provider.
9. Understanding the steps, design and implementation of content provider app.