Introduction to Android App Development & Architecture

Presentation by:

Siddhartha Bhuyan

Sci/Eng – 'SD', NESAC

siddhartha.bhuyan1@nesac.gov.in

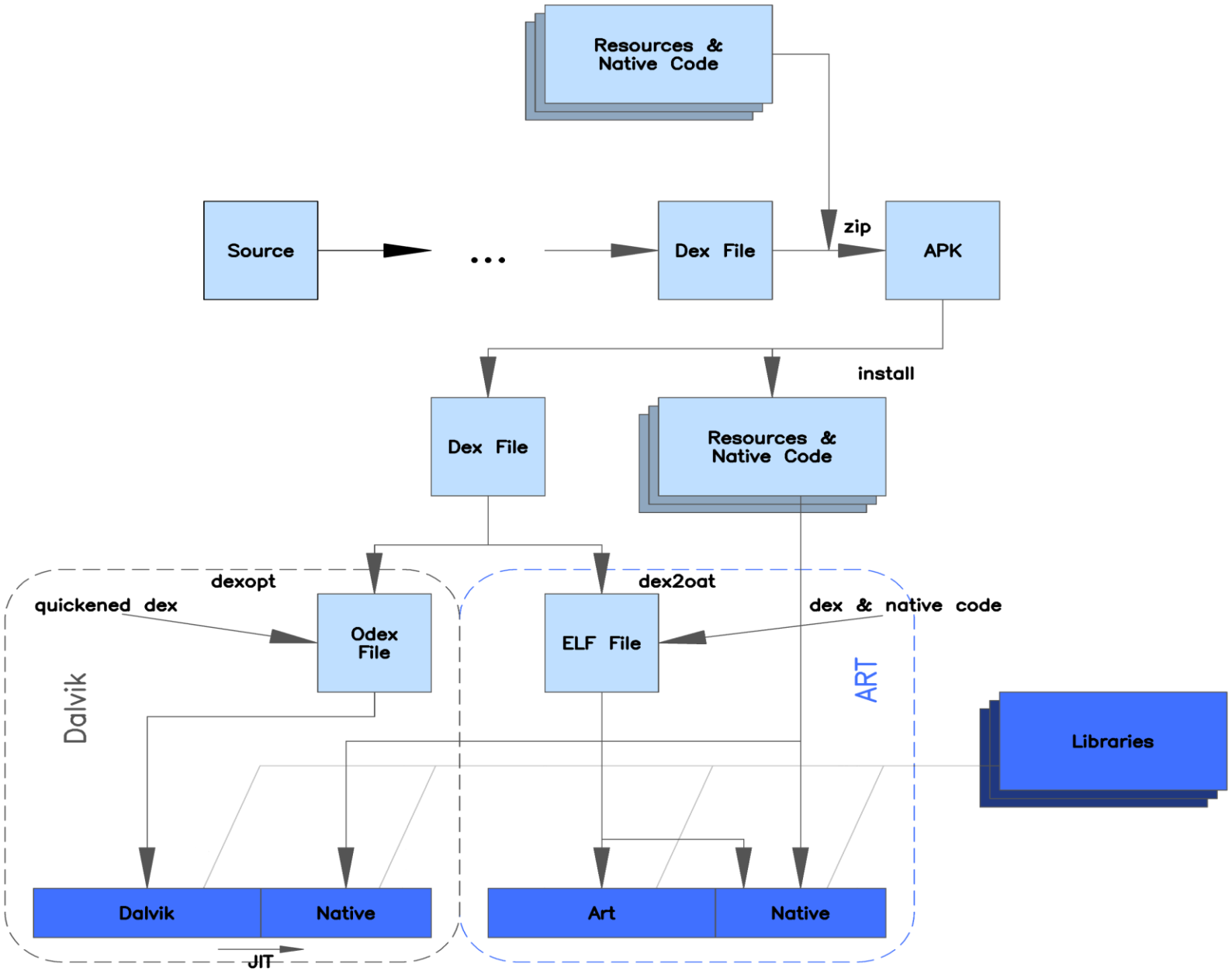
Mob: 9085995599

What is Android?

- Open source and Linux-based operating system for mobile devices
- Originally developed by Android Inc. (Andy Rubin, Rich Miner, Nick Sears, and Chris White), Oct 2003
- Bought by Google in 2005
- Written in Java(UI), C(core), C++
- Open Handset Alliance formed in 2007 with a goal to develop *"the first truly open and comprehensive platform for mobile devices"*
- Previously Dalvik virtual machine; now ART (Android RunTime)

DALVIK VIRTUAL MACHINE	*ANDROID RUN TIME*
Faster Booting time	Rebooting is significantly longer
Cache builds up overtime	The cache is built during the first boot
Occupies less space due to JIT	Consumes a lot of storage space internally due to AOT
Works best for small storage devices	Works best for Large storage devices
Stable and tested virtual machine	Experimental and new – not much app support comparatively
Longer app loading time	Extremely Faster and smoother Faster and app loading time and lower processor usage
Uses JIT compiler(JIT: Just-In-Time) Thereby resulting in lower storage space consumption	Uses AOT compiler(Ahead-Of-Time) thereby compiling apps when installed
Application lagging due to garbage collector pauses and JIT	Reduced application lagging and better user experience
App installation time is comparatively lower as the compilation is performed later	App installation time is longer as compilation is done during installation
DVM converts bytecode every time you launch a specific app.	ART converts it just once at the time of app installation. That makes CPU execution easier. Improved battery life due to faster execution.
It is slower than ART.	It is faster.
It does not provide optimized battery life as it consumes more power.	It provides optimized battery performance as it consumes less power.
While considering Booting, then this device is fast.	It lags in term of booting.

Dalvik vs ART



The Open Handset Alliance

A consortium of 84 companies as of today

Android Layers

User applications: Contacts, phone, browser, etc.

Application managers: windows, content, activities, telephony, location, notifications, etc.

Android Runtime: Java via Dalvik/ART VM

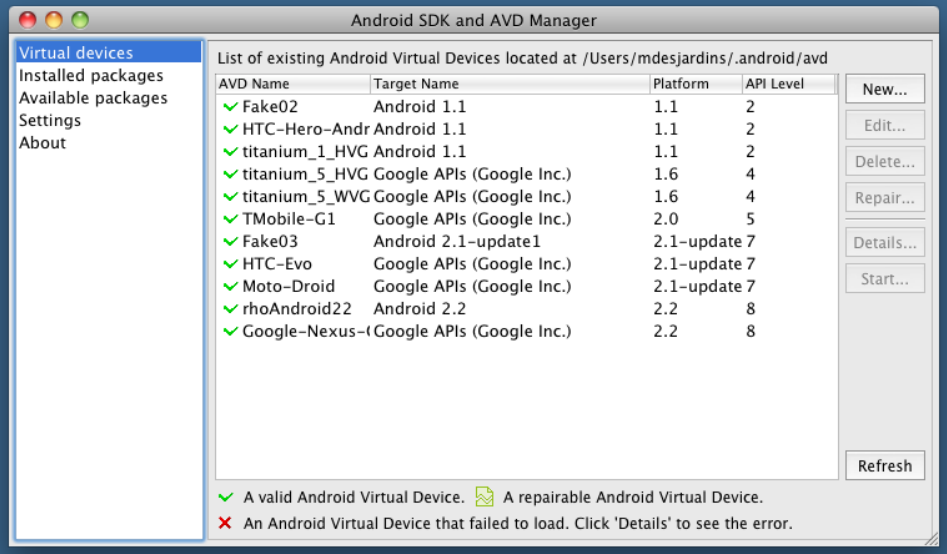
Libraries: graphics, media, database, communications, browser engine, etc.

Linux kernel, including device drivers

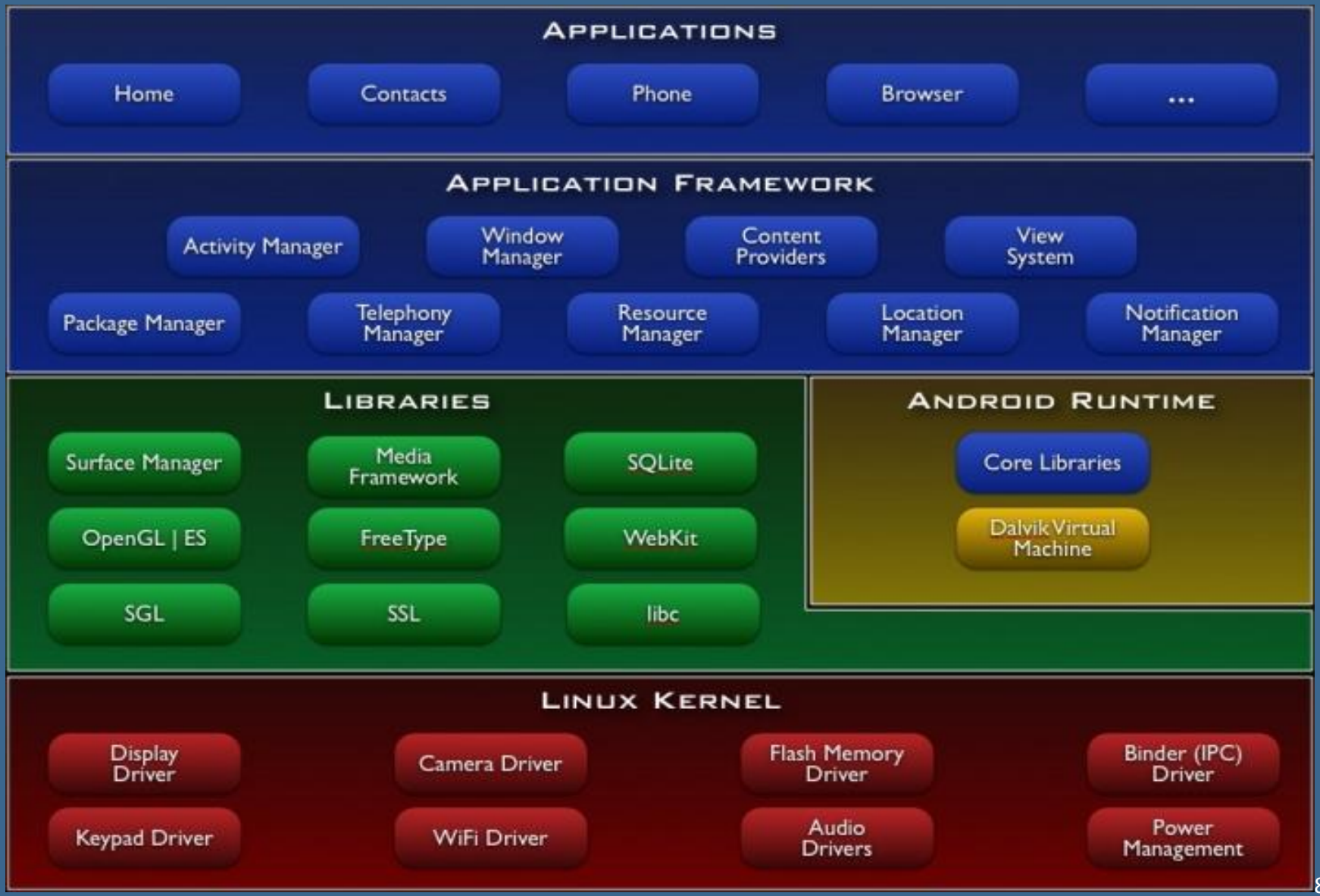
Hardware device with specific capabilities such as GPS, camera, Bluetooth, etc.

Development Environment

- Eclipse + ADT is the “default”; VSCode
- Android SDK comes with command line tools and emulator (Android Debug Bridge or adb)



Android Software Stack



Some Libraries Used by Android

System C library - implementation of the C library (libc)

Media Libraries - based on PacketVideo's OpenCORE

Surface Manager - composites 2D and 3D graphic layers

LibWebCore - a modern embeddable web view

SGL - the underlying 2D graphics engine

3D libraries - based on OpenGL ES 1.0 APIs; the libraries use hardware 3D acceleration

FreeType - bitmap and vector font rendering

SQLite - a powerful and lightweight relational database engine

Critical Android Components

Android Runtime

- Includes a set of core libraries of JAVA that provides most of the functionality
- Runs in its own process, with its own instance of the Dalvik/ART Virtual Machine

Linux Kernel

- Acts as an abstraction layer between the hardware and the rest of the software stack
- Relies on Linux for core system services such as security, memory management, process management, network stack, and driver model

Anatomy of an Android Application

There are some important building blocks for a typical Android application:

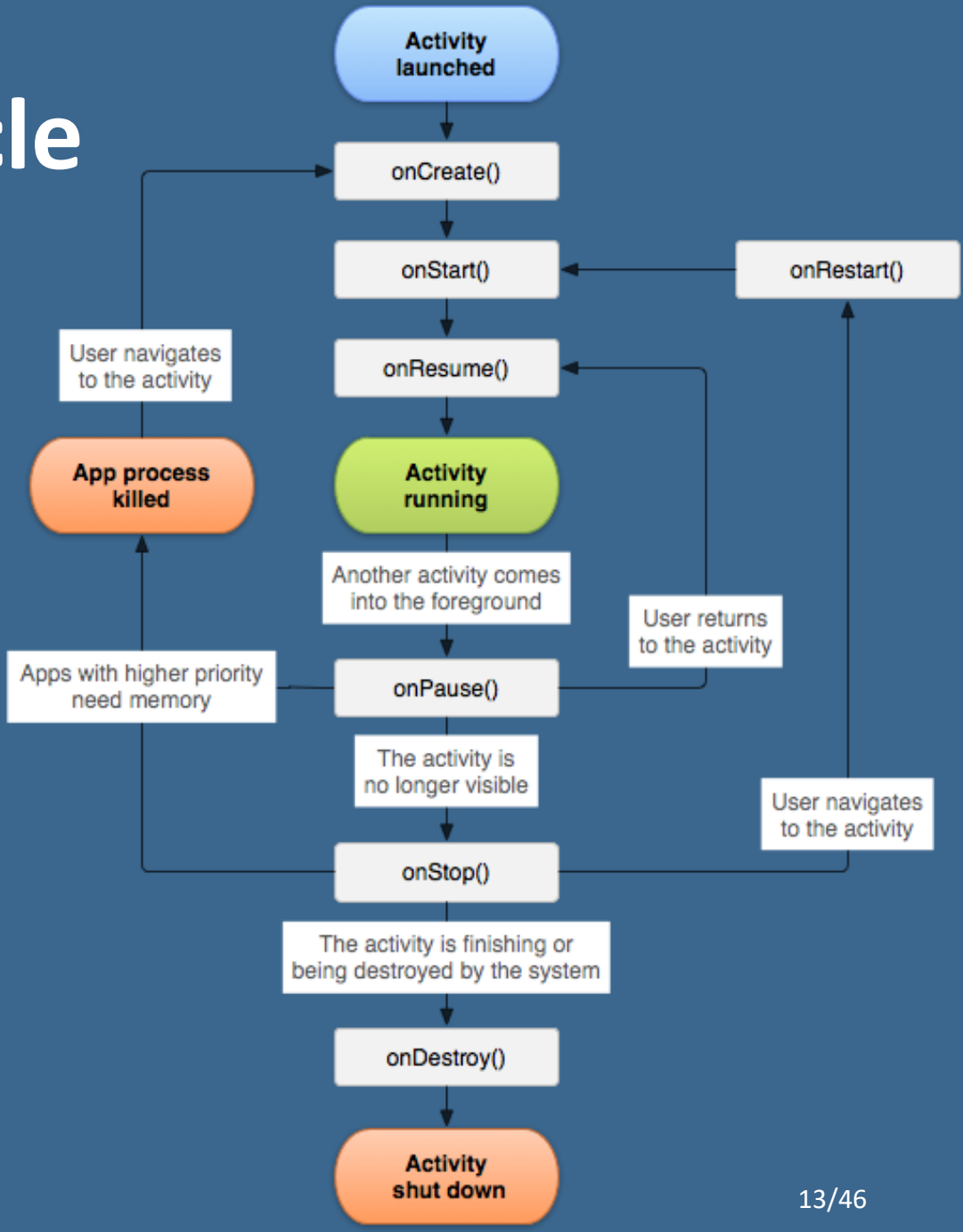
- Activity
  - a single screen
- View
  - all UI elements (through XML definitions)
- Intent Receiver
  - to execute in reaction to an external event(Phone Ring)
- Service
  - code that is long-lived and runs without a UI(Media Player)
- Content Providers
- Broadcast Receivers

Activities

- The basis of android applications
- A single Activity defines a single viewable screen
  - the actions, not the layout
- Can have multiple per application
- Each is a separate entity
- They have a structured life cycle
  - Different events in their life happen either via the user touching buttons or programmatically
- Class inheriting from android.app.Activity
- Started by “Intents” (more on those later)
- Callable from other applications if allowed

Activity Lifecycle

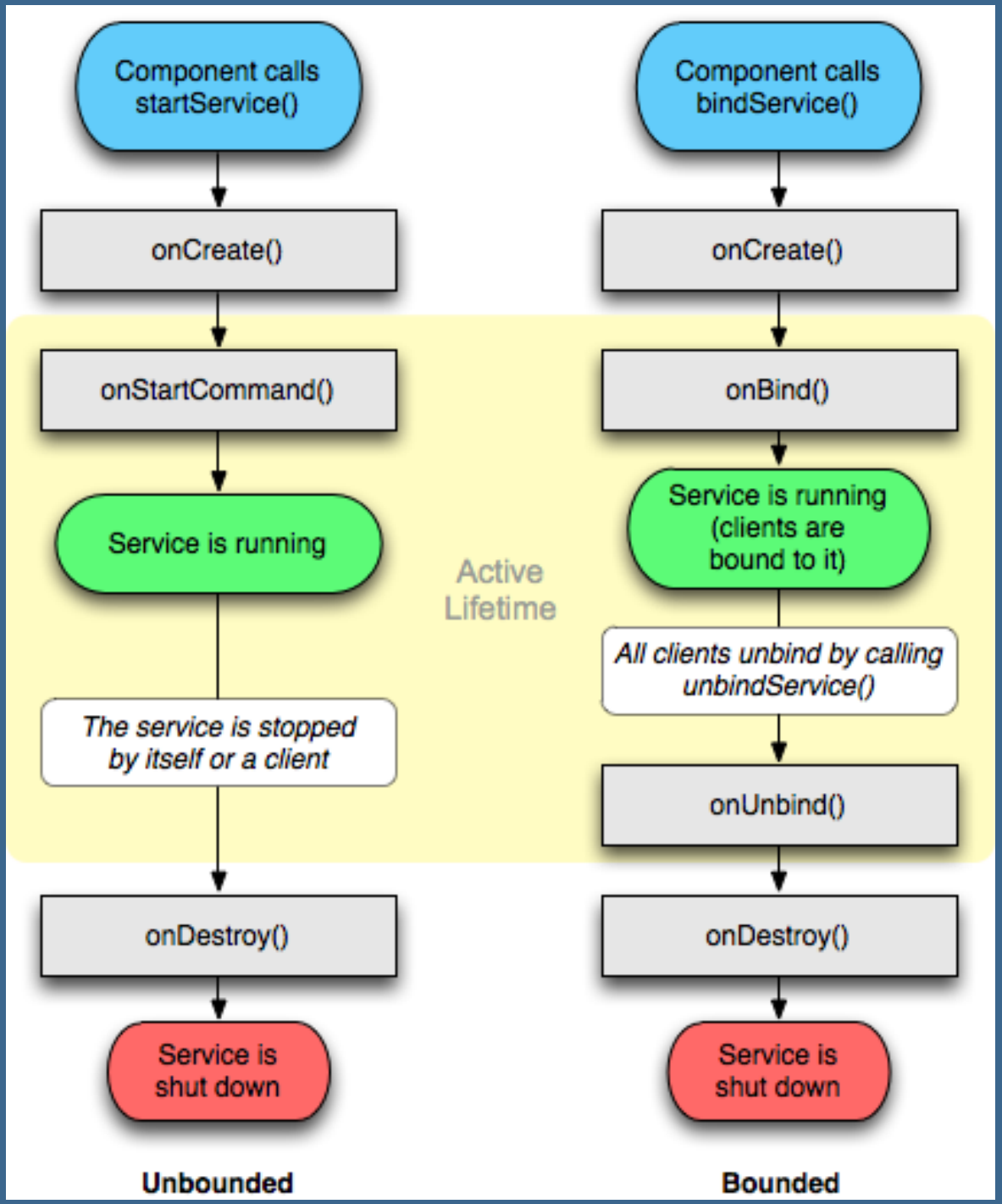
- **Entire Lifetime** - Between onCreate and onDestroy
- **Visible Lifetime** - Between onStart and onStop
- **Foreground Lifetime** - Between onResume and onPause



Services

- Run in the background, no user interface
  - Can continue even if Activity that started it dies
  - Should be used if something needs to be done while the user is not interacting with application
    - Otherwise, a thread is probably more applicable
  - Should create a new thread in the service to do work in, since the service runs in the main thread
- Can be bound to an application
  - In which case will terminate when all applications bound to it unbind
  - Allows multiple applications to communicate with it via a common interface
- Needs to be declared in manifest file
- Like Activities, has a structured life cycle

Service Lifecycle



Of Particular Interest – Location Services

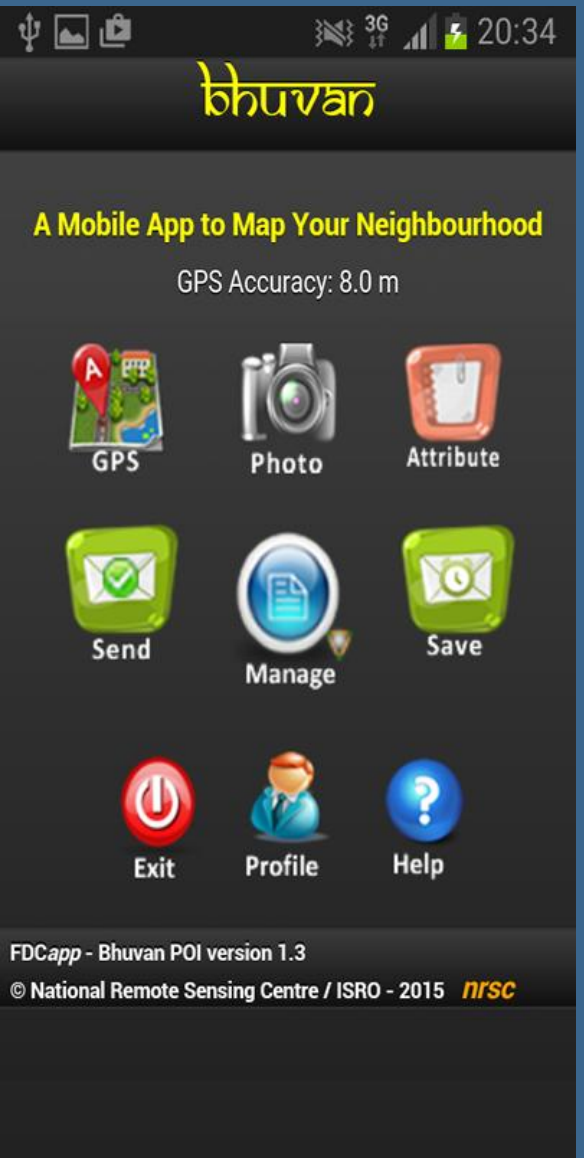
- Helps build location aware apps
- Knowing *where the user is* allows an app to be smarter and deliver better information to the user
- To acquire the user location you can use:
  - GPS
  - Android's Network Location Provider
- At its heart is the *Location* object which represents a geographic location (latitude, longitude, time stamp, bearing, altitude, velocity etc.)

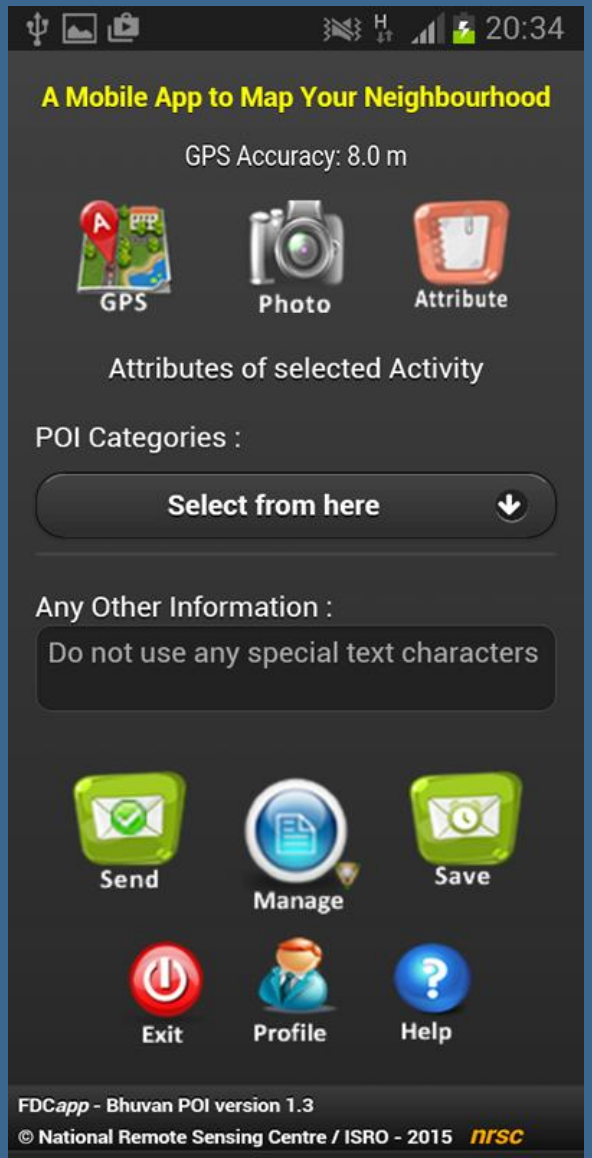
Location Services – Types

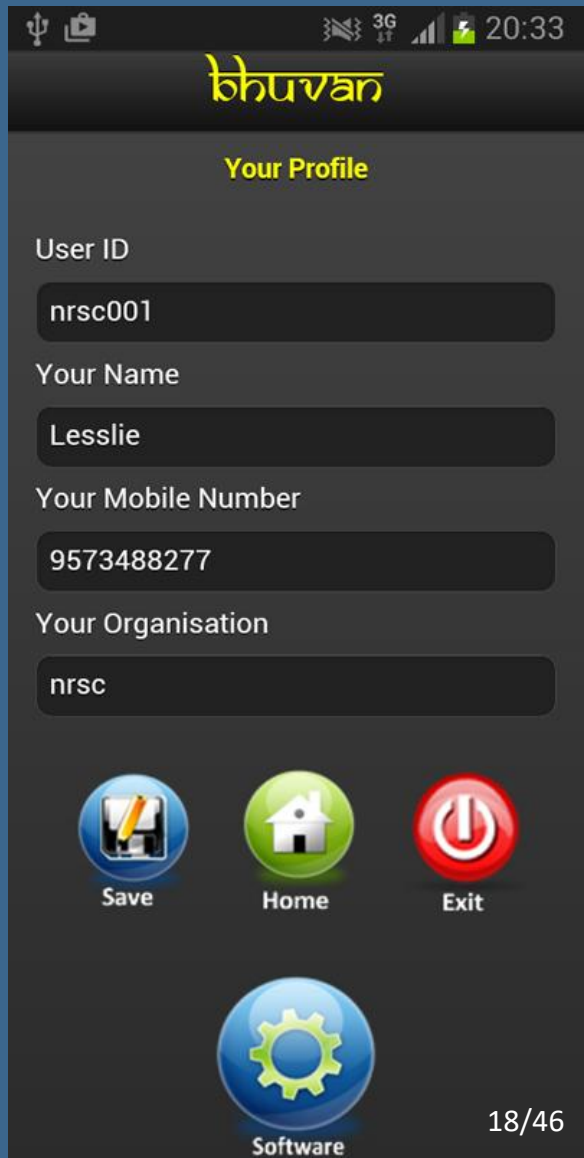
- GPS is most accurate but
  - It only works outdoors
  - It quickly consumes battery power
  - Doesn't return the location as quickly as users want

- Android's Network Location Provider determines user location using cell tower and Wi-Fi signals, providing location information in a way that
  - Works indoors and outdoors
  - Responds faster
  - Uses less battery power

Example: The *BhuvanPOI* App







Views

- Android allows you to use the Model-View-Controller architectural pattern
- Views contain UI elements displayed on the screen by an Activity (controller)
- Can be defined in XML or programatically
- Views are nested, topmost is a LayoutView
- setContentView(viewId) or setContentView(view)

Sample ‘Hello World’ View

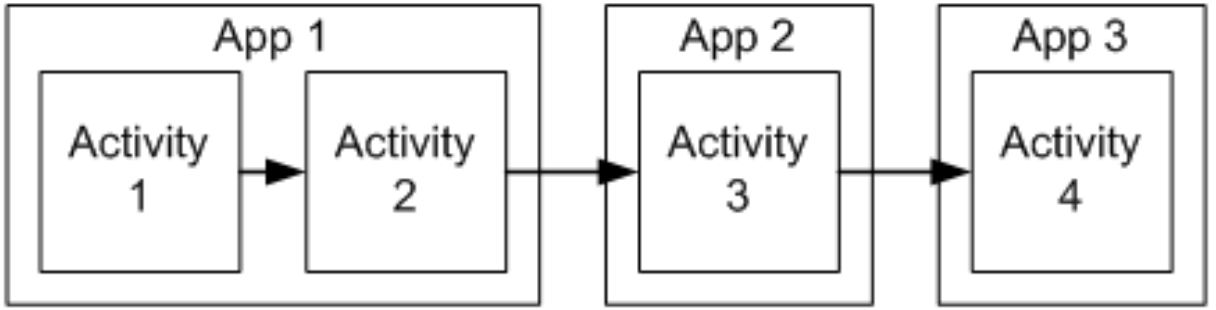
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
     android:orientation="vertical"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     >
<TextView
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="@string/hello"
     />
</LinearLayout>
```

Intents

- Allows an activity, service or broadcast receiver to link to another
  - Within or between apps
  - Allows apps to use components of others
- In some ways, Android resembles a Service-oriented Architecture (SOA)
- An Intent represents the user's desire to accomplish something
- An app can create an Intent without knowing or caring what Activity will receive it
- Applications create Intent Filters in their manifests, indicating Activities (or other things like *BroadcastReceivers*) that would like to be notified of Intents

Intent Resolution

App	Activity	User's next action
Messaging	View list of messages	User taps on a message in the list
Messaging	View a message	User taps Menu -> View Contact
Contacts	View a contact	User taps Call Mobile
Phone	Call the contact's mobile number	

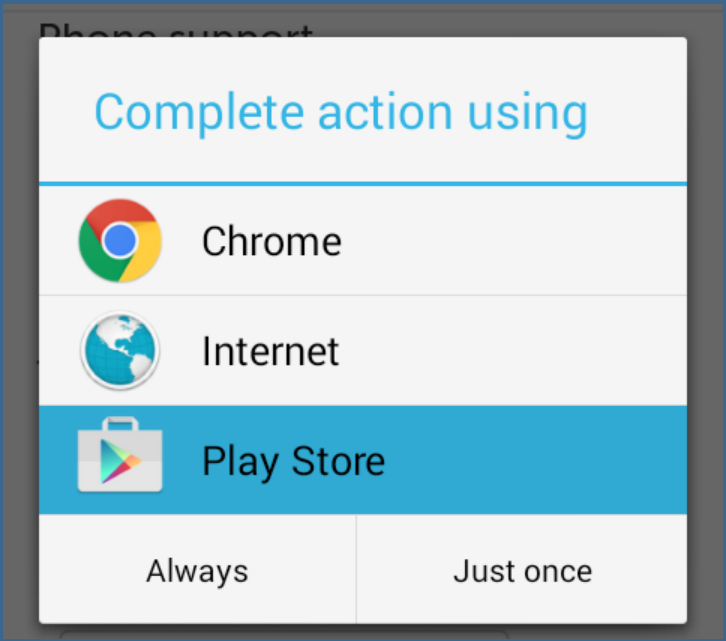


Two types of Intents:

- **Explicit** - identifies a Component Name (a specific Activity class) to be started
- **Implicit** - does not name a target Activity, instead identifies an action, a category, and data and/or data type
- Typical "Actions": ACTION_VIEW, ACTION_DIAL, ACTION_EDIT, etc.

What If More Than One Intent Works?

- If more than one Activity is registered to receive an intent, Android will ask the user which Activity they want to launch



- This is how people build things like alternate dialers or text messaging apps

Content Providers

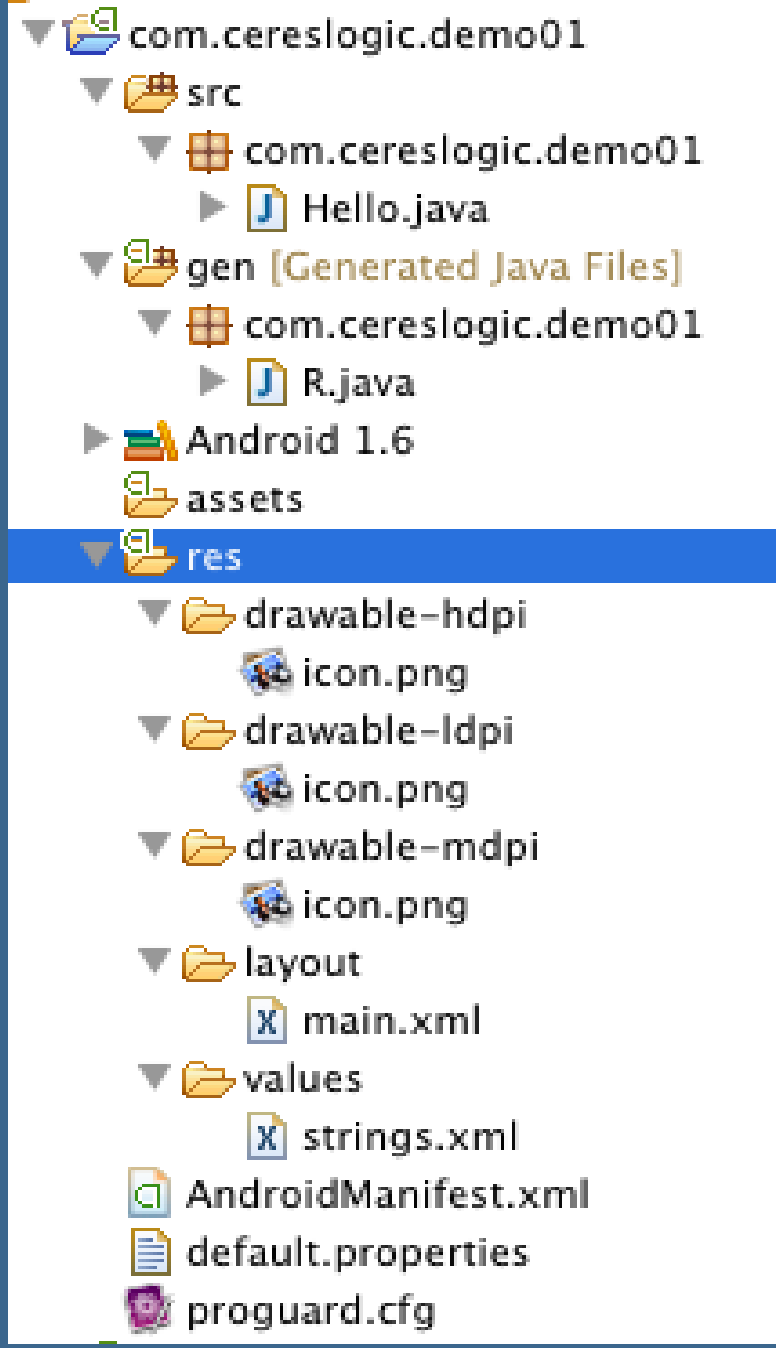
- Manages shared set of application data
- Data may be shared between apps or be private
- Performs data handling functions

Broadcast Receivers

- Listens for system wide (intent) broadcasts
- Intent filter limits which intents cared about
- Similar to an interrupt handler
- Redirects to appropriate activity or service
- Used for receiving events. Intended for short “trigger” style events, not long running activities (use Services for those) Examples: Incoming Phone Call, Text Message
- Registered using Intent Filters, just like Activities
- No View associated with them
- Can be registered statically in the manifest, or at runtime
- Instance goes away when you’re finished processing
- Ordered or (more typically) Unordered

Resources

- Resources go into the “res” directory
- Different Types: strings, layouts, drawables, arbitrary XML, styles, raw, etc.
- Referenced in XML using identifiers prefixed with an at sign (@) followed by type
- Handles / IDs end up in the “R” class



Referring to Resources

```
package com.cereslogic.demo01;

import android.app.Activity;
import android.os.Bundle;

public class Hello extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

The Class “R”

- Autogenerated by Android’s compile tools
- Scans res folder, creates R in your gen folder with the same package name as your app

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated
   by the
 * aapt tool from the resource data it
   found.  It
 * should not be modified by hand.
 */

package com.cereslogic.demo01;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int
    app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Activities Go Into the Manifest

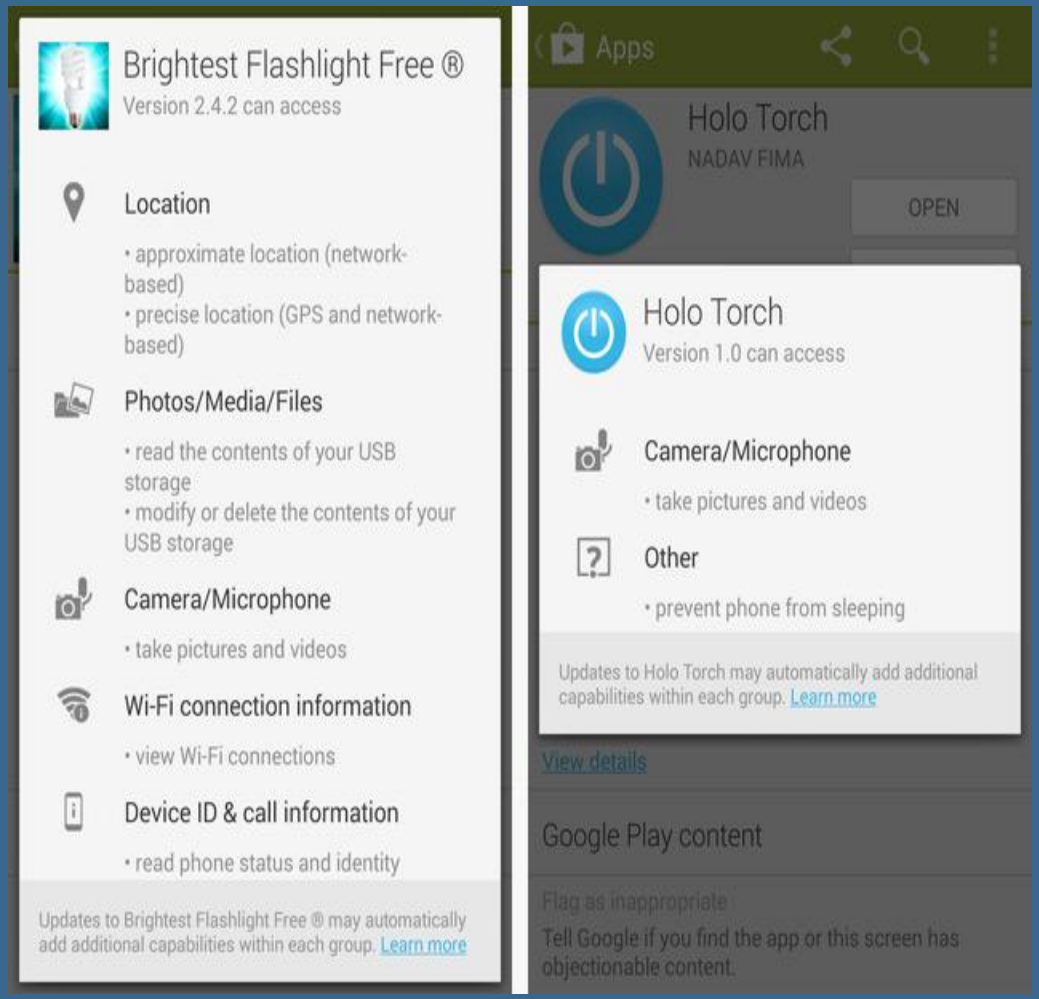
- AndroidManifest.xml exists in the root directory of the project.
- Describes activities in your project and any intents that should activate them
- Describes application permissions, version, name, required API version, among other things...

Sample Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.cereslogic.androidpreso"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
   android:label="@string/app_name">
        <activity android:name=".Hello"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
                <action android:name="android.intent.action.ACTION_SEND" />
                <data android:mimeType="image/png" />
            </intent-filter>
        </activity>
        <activity android:name=".MyTreat"
                  android:label="@string/treat_list">
        </activity>
    </application>
</manifest>
```

Permissions (*and Privacy Invasion?*)

- To use some of the facilities in the device, you need to declare that you will use them in the Manifest
- The user is informed of these permissions when he or she downloads your application
- Examples: Location (Fine and Coarse), WiFi state, Network state, Call, Camera, Internet, Send/Receive SMS, Read Contacts etc.



"*Rooting*" Android

- Processes in the Android system run under an unprivileged user account
- Default user can't access all of the filesystem, or some of the more interesting APIs (e.g., modifying mobile themes, setting CPU speed etc.)
- "Rooting" is finding back door shell access, and copying a program conceptually similar to *su* or *sudo* onto your phone
- Custom ROM (Read Only Memory) is something completely different

Application Architecture

An app architecture defines the boundaries between parts of the app and the responsibilities each part should have. The app architecture should follow a few specific principles:

- ***Separation of concerns*** – Don't write all your code in an Activity or a Fragment. These UI-based classes should only contain logic that handles UI and operating system interactions. Minimize your dependency on them as they can be killed by the OS

Application Architecture

- ***Drive UI from data models*** – Data models represent the data of an app and are independent from the UI elements and other components. Can be destroyed when the OS decides to remove the app's process from memory. Persistent models are ideal since:
  - Your users don't lose data if the Android OS destroys your app to free up resources.
  - Your app continues to work in cases when a network connection is flaky or not available.

  If you base your app architecture on data model classes, you make your app more testable and robust.

Application Architecture

- ***Single source of truth*** – Assign a Single Source of Truth (SSOT) to new data types. Only the SSOT can modify or mutate it. It exposes the data using an immutable type, and to modify the data, uses functions or receive events that other types can call.

  This pattern brings multiple benefits:

  - It centralizes all the changes to a particular type of data in one place.
  - It protects the data so that other types cannot tamper with it.
  - It makes changes to the data more traceable. Thus, bugs are easier to spot.

  In an offline-first application, the source of truth for application data is typically a database. In some other cases, the source of truth can be a ViewModel or even the UI.

Application Architecture

- ***Unidirectional Data Flow*** –In UDF, state flows in only one direction. The events that modify the data flow in the opposite direction.

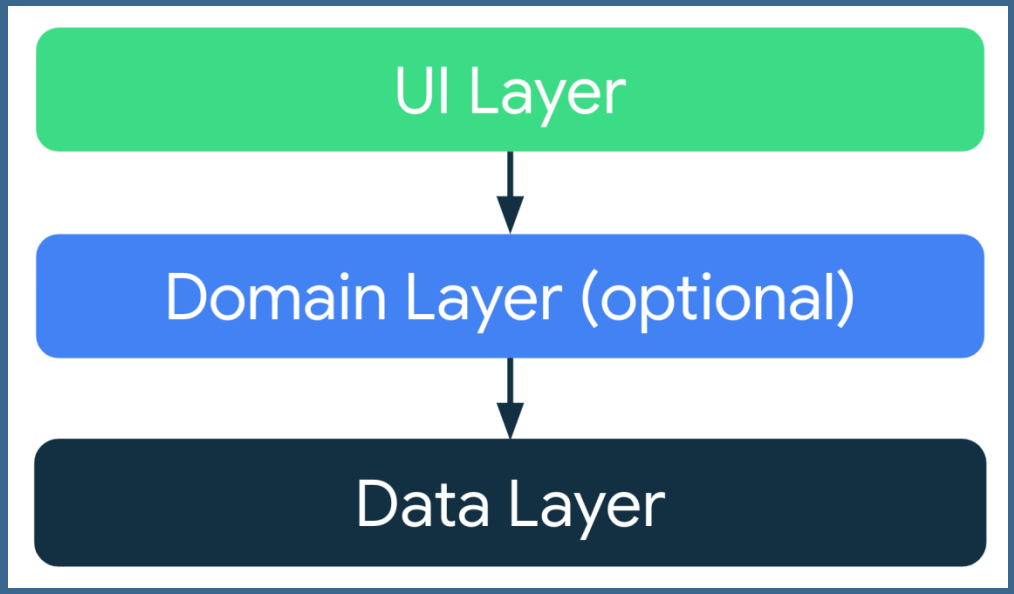
  In Android, state or data usually flow from the higher-scoped types to the lower-scoped ones. Events are usually triggered from the lower-scoped types until they reach the SSOT for the corresponding data type. For example, application data usually flows from data sources to the UI. User events such as button presses flow from the UI to the SSOT where data is modified.

  This guarantees data consistency, is less prone to errors, is easier to debug and brings all the benefits of the SSOT pattern.

Typical App Architecture

Each application should have at least two layers:

- The UI layer that displays application data on the screen.
- The data layer that contains the business logic of your app and exposes application data.
- You can add an additional layer called the domain layer to simplify and reuse the interactions between the UI and data layers.



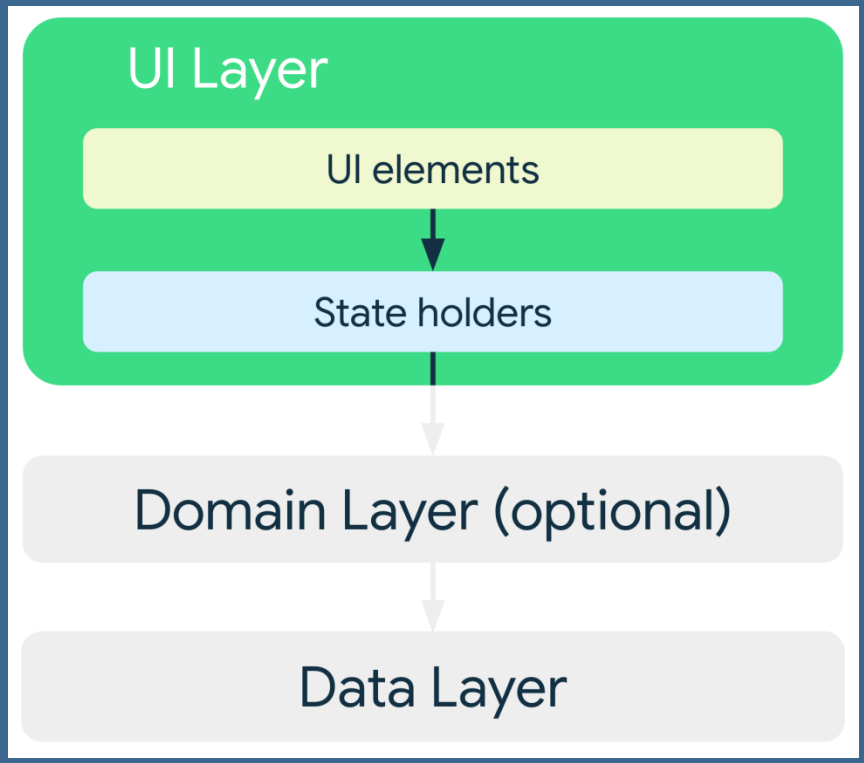
Modern App Architecture

Modern App Architecture encourages using the following techniques, among others:

- A reactive and layered architecture.
- Unidirectional Data Flow (UDF) in all layers of the app.
- A UI layer with state holders to manage the complexity of the UI.
- Coroutines and flows (simplify asynchronous programming)
- Dependency injection best practices (supplying dependent class instantiation as parameter instead of initializing and constructing it itself)

The User Interface (UI) Layer

The role of the UI layer (or presentation layer) is to display the application data on the screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI should update to reflect the changes.



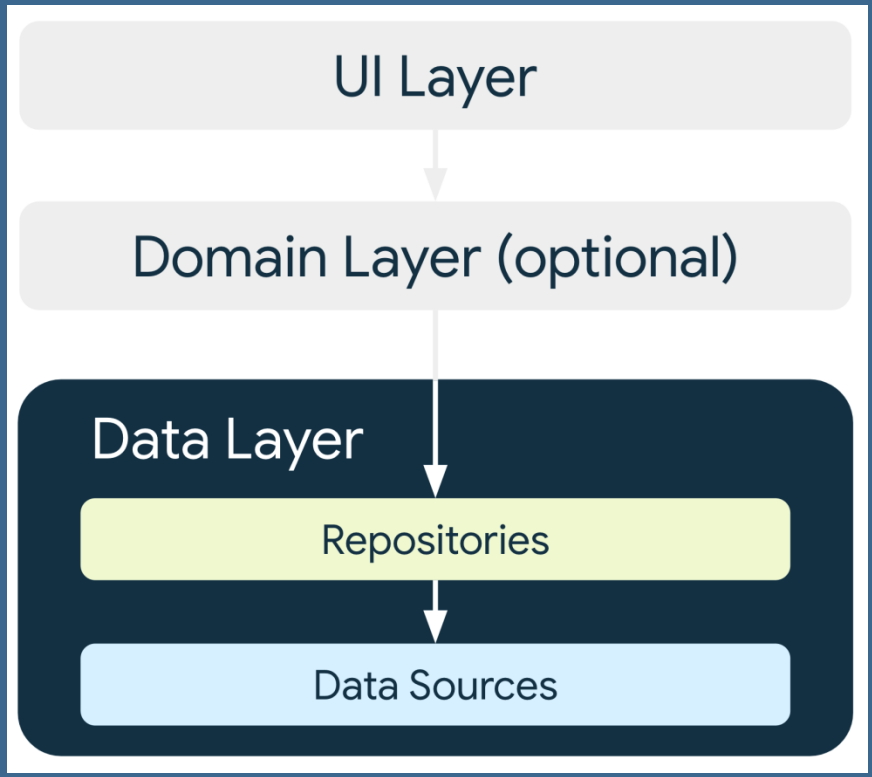
The UI layer is made up of two things:

- UI elements that render the data on the screen
- State holders (such as ViewModel classes) that hold data, expose it to the UI, and handle logic.

The Data Layer

The data layer of an app contains the business logic. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.

The data layer is made of repositories that each can contain zero to many data sources. Create a repository class for each different type of data you handle in your app. For example, you might create a MoviesRepository class for data related to movies, or a PaymentsRepository class for data related to payments.



The Data Layer

Repository classes are responsible for the following tasks:

- Exposing data to the rest of the app.
- Centralizing changes to the data.
- Resolving conflicts between multiple data sources.
- Abstracting sources of data from the rest of the app.
- Containing business logic.

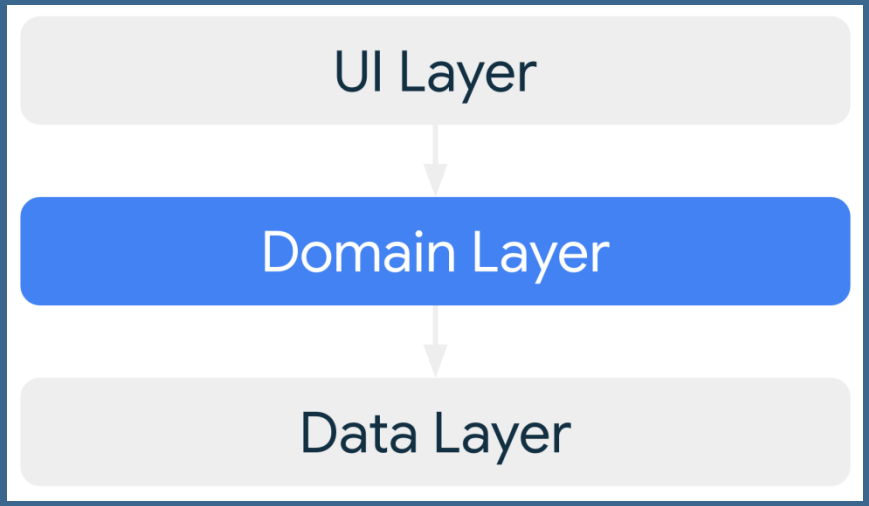
Each data source class should have the responsibility of working with only one source of data, which can be a file, a network source, or a local database. Data source classes are the bridge between the application and the system for data operations.

The Domain Layer

The domain layer is an optional layer that sits between the UI and data layers.

It is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. Use it only when needed—for example, to handle complexity or favor reusability.

Classes in this layer are commonly called use cases or interactors. Each use case should have responsibility over a single functionality.



Managing Dependency Between Components

Classes in your app depend on other classes in order to function properly. You can use either of the following design patterns to gather the dependencies of a particular class:

- Dependency injection (DI): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies.

- Service locator: The service locator pattern provides a registry where classes can obtain their dependencies instead of constructing them.

General Best Practices

Don't store data in app components. Avoid using app's entry points—such as activities, services, and broadcast receivers—as sources of data. Instead, they should only coordinate with other components to retrieve the subset of data that is relevant to that entry point.

Reduce dependencies on Android classes. Only app components should rely on Android framework SDK APIs such as Context, or Toast.

Create well-defined boundaries of responsibility between various modules in your app. For example, don't spread the code that loads data from the network across multiple classes or packages in your code base. Similarly, don't define multiple unrelated responsibilities—such as data caching and data binding—in the same class.

General Best Practices

Expose as little as possible from each module. Expose only what is required – nothing less, nothing more

Focus on the unique core of your app so it stands out from other apps. Avoid writing the same code again and again. Instead, focus on what makes your app unique, and let libraries handle the repetition.

Consider how to make each part of your app testable in isolation. For example, having a well-defined API for fetching data from the network makes it easier to test the module that persists that data in a local database. If you mix the logic from these two modules it becomes much more difficult to test effectively.

General Best Practices

Types are responsible for their concurrency policy. If a type is performing long-running blocking work, it should be responsible for moving that computation to the right thread. That particular type knows the type of computation that it is doing and in which thread it should be executed. Types should be main-safe, meaning they're safe to call from the main thread without blocking it.

Persist as much relevant and fresh data as possible. That way, users can enjoy your app's functionality even when their device is in offline mode. Remember that constant, high-speed connectivity is tough to come by.