Android Development with Kotlin

App Components and Background

App Components

Application components are the essential building blocks of an Android application. These components are loosely coupled by the application manifest file AndroidManifest.xml that describes each component of the application and how they interact. These components can be thought of multiple ways/outlets to access the contents of an application and launch the app for the user. An Android App consists of four App Components, namely:

1. **Activity:** An activity represents a single screen with a user interface. It is the only App Component that has a UI integrated with it. Hence, it is responsible for all the UI interactions within an Application. You have been working with Activities for a long time, In this lecture, the focus shifts towards other components like Services and Broadcast Receivers.
2. **Services:** A service is a component that runs in the background to perform long-running operations. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity.
3. **Broadcast Receivers:** Broadcast Receivers simply respond to broadcast messages from other applications or from the system. 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, then a broadcast receiver who will intercept this communication and will initiate appropriate action.
4. **Content Providers:** A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentResolver class. The data may be stored in the file system, the database or somewhere else entirely. A content provider is implemented as a subclass of ContentProvider class and must implement a standard set of APIs that enable other applications to perform transactions.

Service

A Service is an application component that can be used to perform a number of tasks for an application. A service can be of three types; background service, foreground services and bound services.

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. 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.

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***.

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.

Services can be added to your application using the following steps:

1. Declare a Service within manifest

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

2. Create a Service subclass

```
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()
    }
}
```

3. Start a Service within the Activity

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

Intent Service

IntentService is a base class for Services that handle asynchronous requests on demand. Clients send requests through startService(Intent) function calls. The service is then started as needed, maintains a queue of jobs and performs one job after the another. When it runs out of tasks, it stops by itself.

Intent Services should be used when the Service is expected to perform more than one task and you require the service to shut off after work completion. Ideally, most of the work performed using services should be done using the Intent Service.

To use the IntentService class, extend IntentService and implement the onHandleIntent. All requests are handled on a single worker thread - they may take as long as necessary (and will not block the application's main loop), but ***only one request will be processed at a time.***

The onStart() function was overridden in the JobIntentService until the API 30, but this function was deprecated in this API level. Now, the onStartCommand() function is used instead.

JobIntent Service

Job Intent Services are a subclass of intent service that acquire and release wakelocks on their before beginning its processes and completing their tasks respectively.

A wakelock is a powerful concept in Android that allows the developer to modify the default power state of their device. It allows the user to allow the device from sleeping when the background tasks are still running and is helpful in case of services. The danger of using a wakelock in an application is that it will reduce the battery life of a device.

Job Intent Services use the JobScheduler function on Android O, but runs as a background service to emulate JobScheduler on pre-O. The JobIntentService has the following characteristics:

- On Android N and below devices, the background service starts immediately and the loaded jobs are executed sequentially. This ensures backwards compatibility with the same code.
- In Android O and later, jobs loaded through JobScheduler are executed sequentially. Internally, it is scheduled by jobInfo.setOverrideDeadline(0).build(). If you want other conditions, JobIntentService doesn't fit it.
- You don't need WakefulBroadcastReceiver to use Wakelock when running on pre-O, because JobIntentService would manage Wakelock for you. This prevent writing wrong code such as battery drain due to not releasing Wakelocks.

A Job Intent Service should be used when the tasks/jobs completed by the service may extend the processing of the application. Such as data-backups. The backups should remain uninterrupted even when the application is closed.

To use JobIntentService within your application use the following steps:

1. Add the maven dependencies to the Gradle files.

```
allprojects {
    repositories {
        jcenter ()
        maven {
            url "https://maven.google.com"
         }
    }
}
dependencies {
   ...
   compile 'com.android.support:support-compat:26.0.0-beta2'
}
```

2. Extend the JobIntentService to create a subclass.

```
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.JobIntentService;
import android.util.Log;
import android.widget.Toast;
/**
 * Example implementation of a JobIntentService.
 */
public class UpdateJobIntentService extends JobIntentService {
    static final int JOB_ID = 1000;
    static final String WORK_DOWNLOAD_ARTWORK = ".DOWNLOAD_ARTWORK";
    ArtworkDownloader mDownloader;

    static void enqueueWork(Context context, Intent work) {
        enqueueWork(context, Update.class, JOB_ID, work);
    }

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

        mDownloader = ArtworkDownloader.getSequencialDownloader();
    }

    @Override
    protected void onHandleWork(Intent intent) {
      // enqueueWork()
      if (WORK_DOWNLOAD_ARTWORK.equals(intent.getAction())) {
          mDownloader.download(intent.getStringExtra("URL"))
      }
    }

    @Override
    public boolean onStopCurrentWork() {
      return !mDownloader.isFinished();
    }
}
```

In the above example, the "onHandleWork" function is called every time a job is assigned to the Service. The onHandleWork function should then call the enqueueWork function which isn't overridden. This function enqueues the work with a Job Id and a Work class that needs to be called to complete the task. The "enqueueWork" function is a function of the JobIntentService class, and a function with the same name and different parameters is created within the above example that calls the actual enqueueWork function.

Broadcast Receiver

Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. These broadcasts are sent when an event of interest occurs. For example, *the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging. Apps can also send custom broadcasts*, for example, to notify other apps of something that they might be interested in. Some use cases of the of the Broadcast Receiver are:

1. Communication between multiple apps
2. Create Bound Services within the Applications
3. Respond to system events that occur within the device

A BroadcastReceiver is an Android Class that receives these broadcasts and performs services accordingly. There are two types of receivers in Android, Manifest-declared and context-bound receivers.

Manifest-declared Receivers

1. Specify the <receiver> element in your app's manifest.

```
<receiver android:name=".MyBroadcastReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" /
>
    </intent-filter>
</receiver>
```

2. Create a subclass of the BroadcastReceiver

```
private const val TAG = "MyBroadcastReceiver"
class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        StringBuilder().apply {
            append("Action: ${intent.action}\n")
            append("URI: ${intent.toUri(Intent.URI_INTENT_SCHEME)}\n")
            toString().also { log ->
                Log.d(TAG, log)
                Toast.makeText(context, log, Toast.LENGTH_LONG).show()
            }
        }
    }
}
```

Context-Registered Receivers

1. Create an instance of the Receiver

```
val br: BroadcastReceiver = MyBroadcastReceiver()
```

2. Create and add intent-filters for the Receivers

```
val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION).apply {
    addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED)
}
registerReceiver(br, filter)
```

Sending a Broadcast

To send a broadcast, add the following code to your Activity

```
Intent().also { intent ->
    intent.setAction("com.example.broadcast.MY_NOTIFICATION")
    intent.putExtra("data", "Notice me senpai!")
    sendBroadcast(intent)
}
```

Work Manager

WorkManager is a library used to enqueue deferrable work that is guaranteed to execute sometime after its Constraints are met. WorkManager allows observation of work status and the ability to create complex chains of work.

WorkManager uses an underlying job dispatching service when available based on the following criteria:

- Uses JobScheduler for API 23+
- Uses a custom AlarmManager + BroadcastReceiver implementation for API 14-22

There are two types of Work Managers that can be added, one time work manager and a periodic work manager.

WorkManagers should be created within the Application class as the activity maybe created or destroyed a number of time during the run-time of an application, but an instance of the application class is created only once. A request for the WorkManager can be made using the following code:

```
val myWorkRequest = ...
WorkManager.getInstance(myContext).enqueue(myWorkRequest)
```

To schedule a one-time Work:

```
val uploadWorkRequest: WorkRequest =
   OneTimeWorkRequestBuilder<MyWork>()
       // Additional configuration
       .build()
```

To schedule a Periodic Work:

```
val saveRequest =
       PeriodicWorkRequestBuilder<SaveImageToFileWorker>(1, TimeUnit.HOURS)
    // Additional configuration
           .build()
```

Alarm Manager

This class provides access to the system alarm services. These allow you to schedule your application to be run at some point in the future. When an alarm goes off, the Intent that had been registered for it is broadcast by the system, automatically starting the target application if it is not already running. Registered alarms are retained while the device is asleep (and can optionally wake the device up if they go off during that time), but will be cleared if it is turned off and rebooted.

The Alarm Manager holds a CPU wake lock as long as the alarm receiver's onReceive() method is executing. This guarantees that the phone will not sleep until you have finished handling the broadcast. Once onReceive() returns, the Alarm Manager releases this wake lock. This means that the phone will in some cases sleep as soon as your onReceive() method completes. If your alarm receiver called Context.startService(), it is possible that the phone will sleep before the requested service is launched. To prevent this, your BroadcastReceiver and Service will need to implement a separate wake lock policy to ensure that the phone continues running until the service becomes available.

Some of the characteristics of the Alarms are:

1. Alarms can let one fire Intents at set times or in certain time intervals.
2. Alarms can be used together with the broadcast receiver to notifies the user.

3. Alarms can triggers even if the device falls asleep because they run outside the application.
4. Alarms can be scheduled to a specific sharp time thus reducing system resources by not relying on background services.

One time Alarm can be set as:

```
private var alarmMgr: AlarmManager? = null
private lateinit var alarmIntent: PendingIntent
...
alarmMgr = context.getSystemService(Context.ALARM_SERVICE) as AlarmManager
alarmIntent = Intent(context, AlarmReceiver::class.java).let { intent ->
    PendingIntent.getBroadcast(context, 0, intent, 0)
}
alarmMgr?.set(
        AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() + 60 * 1000,
        alarmIntent
)
```

Repeating Alarms can be set as:

```
alarmMgr?.setInexactRepeating(
        AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
        AlarmManager.INTERVAL_HALF_HOUR,
        alarmIntent
)
```