Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

System Services

Federico Montori
federico.montori2@unibo.it

Table of Contents

System Services

System services are modular, focused components. Functionality exposed by Android framework API communicates with system services to access the underlying hardware.

They are exposed to apps in the form of singleton classes

- Often ***Manager**

```
val fakeManager: FakeManager =
    getSystemService(Context.FAKE_SERVICE) as FakeManager
```

System Services

AccessibilityManager
AccountManager
ActivityManager
AlarmManager
AppOpsManager
AudioManager
BatteryManager
BluetoothManager
ClipboardManager
ConnectivityManager
DevicePolicyManager
DisplayManager
DownloadManager
DropBoxManager
FingerprintManager
InputMethodManager
InputManager
JobScheduler
KeyguardManager
LauncherApps
LayoutInflater
LocationManager
MediaProjectionManager
MediaRouter
MediaSessionManager
MidiManager
NetworkStatsManager
NfcManager
NotificationManager
NsdManager
PowerManager
PrintManager
RestrictionsManager
SearchManager
SensorManager
StorageManager,
SubscriptionManager
TelecomManager
TelephonyManager
TextServicesManager
TvInputManager
UiModeManager
UsageStatsManager
UsbManager
UserManager
Vibrator
WallpaperService
WifiManager
WifiP2pManager
WindowManager

Schedule Jobs

There are several ways to do it:

- Alarm Service
  - The classic way, uses the system service plain and simple and can still force exact scheduling
- JobScheduler
  - Newer system service requiring GPlay for scheduling deferred jobs
- WorkManager
  - Uses the previous two seamlessly and has nice features for chaining and observation. Cannot schedule exact jobs though.

Alarm Service

Fires intents in the future.

```
val alarmManager = getSystemService(Context.ALARM_SERVICE) as AlarmManager
alarmManager.set(type: Int, triggerAt: Long, operation:PendingIntent)
```

Type is one of:

- ELAPSED_REALTIME
- ELAPSED_REALTIME_WAKEUP
- RTC
- RTC_WAKEUP

SystemClock.elapsedRealTime()

Elapsed since sys boot.
Better for time slices

System.currentTimeMillis()

UTC Clock
Better for time of the day

Alarm Service

Fires intents in the future.

```
val alarmManager = getSystemService(Context.ALARM_SERVICE) as AlarmManager
alarmManager.set(type: Int, triggerAt: Long, operation:PendingIntent)
```

Type is one of:

- ELAPSED_REALTIME
- ELAPSED_REALTIME_WAKEUP
- RTC
- RTC_WAKEUP

With **WAKEUP** the device would fire the intent even if the device is sleeping. This will force the device to wake up from sleep.

N.B. this will not wake up the device if turned off. This function cannot be implemented by user-level apps.

Alarm Service

Fire *alarmIntent* in **exactly** half an hour from now (no time tolerance).

```
alarmManager.setExact(
      AlarmManager.ELAPSED_REALTIME,
     SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
      alarmIntent )
```

Fire *alarmIntent* **every day** at 14:00 starting from today, waking up the device if sleeping and clustering the alarm with others if present (inexact).

```
val calendar = Calendar.getInstance()
calendar.timeInMillis = System.currentTimeMillis()
calendar.set(Calendar.HOUR_OF_DAY, 14)
alarmManager.setInexactRepeating(
      AlarmManager.RTC_WAKEUP,
      calendar.getTimeInMillis(), AlarmManager.INTERVAL_DAY, alarmIntent )
```

Alarm Service

It is possible to cancel a scheduled operation:

- cancel(PendingIntent operation)
  - Match with filterEquals(Intent anotherIntent)

BEST PRACTICE: Sometimes is useful to set the alarms again if the device has rebooted:

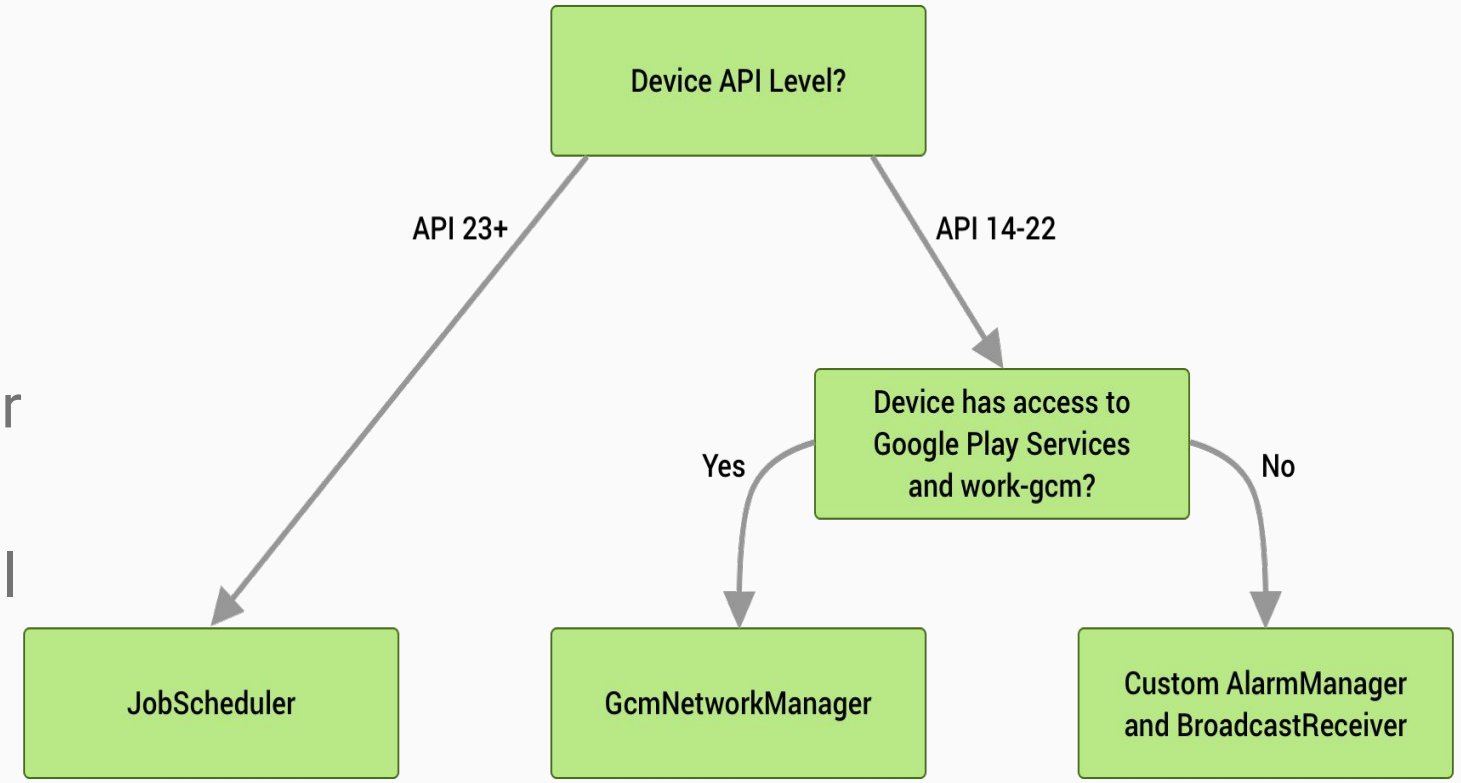
```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

    [...]
        <action android:name="android.intent.action.BOOT_COMPLETED"></action>
```

Work Manager

WorkManager is an API that makes it easy to schedule deferrable, asynchronous tasks that are expected to run even if the app exits or the device restarts.

- It uses a mix of **JobScheduler**, **AlarmManager** and **BroadcastReceiver**
- It is **NOT** a replacement for scheduling tasks at exact time, for that you must still use AlarmManager.



Work Manager

Import the necessary modules:

```
implementation("androidx.work:work-runtime-ktx:2.9.0")
```

Create a Worker class that defines the job to do:

- the body of the **doWork()** function runs in a background thread

```
class UploadWorker(context: Context, params: WorkerParameters):
    Worker(context, params) {
        override fun doWork(): Result {
            /* Do the work here – in this case, upload some images. */
            uploadImages()
            return Result.success()
        }
}
```

Work Manager

Then, we should instantiate the object by stating implicitly what kind of job is

```
val uploadWorkRequest = OneTimeWorkRequest.Builder(UploadWorker::class.java)
        .setInitialDelay(myDelayInSeconds.toLong(), TimeUnit.SECONDS)
        .build()
```

Then we need to get the reference to the **WorkManager** and submit the job

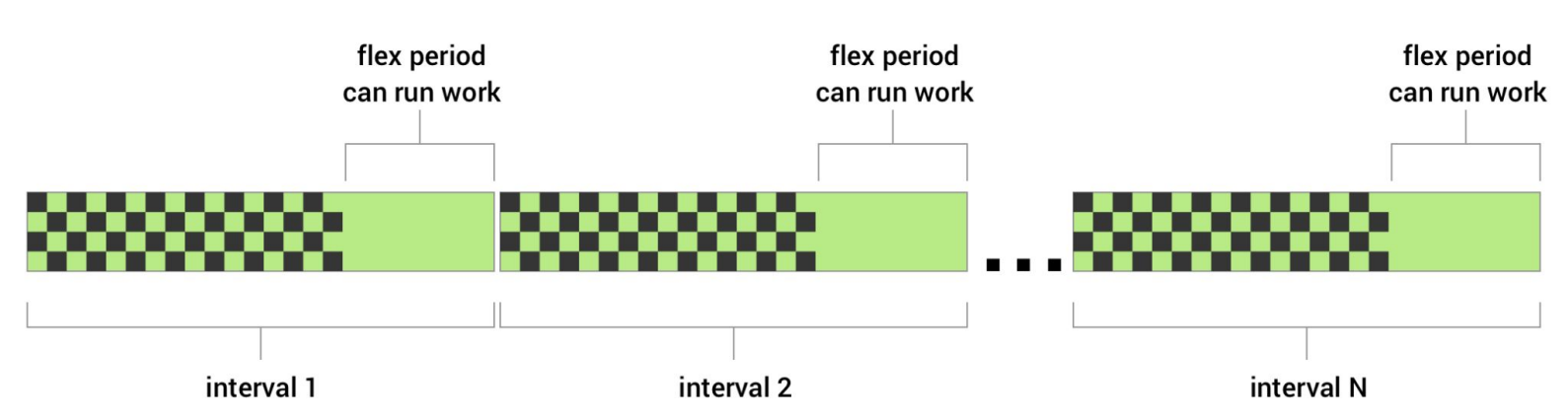
```
WorkManager.getInstance(this).enqueue(uploadWorkRequest)
```

From now on, the job will be executed on top of the constraints declared while building the WorkRequest.

- There are many parameters and constraints (e.g. retries, network types…)
  - https://developer.android.com/topic/libraries/architecture/workmanager/how-to/define-work

Work Manager

You can schedule **periodic jobs** pretty easily and **WorkManager** is powerful enough to set a flexible period.




```
/* In this example the job gets executed every hour with a 15-minutes tolerance */
val periodicUploadRequest = PeriodicWorkRequest.Builder(UploadWorker::class.java,
        1, TimeUnit.HOURS, 15, TimeUnit.MINUTES)
                .build()
```

Work Manager

You can observe changes on your job by using **LiveData**

```
workManager.getWorkInfoByIdLiveData(uploadWorkRequest.id).observe(this) {
    workInfo: WorkInfo ->
        if (workInfo.state != null && workInfo.state == WorkInfo.State.SUCCEEDED) {
            // Your Reaction Here
        }
```

You can also **chain jobs**

```
workManager          // begin with candidate work requests to run in parallel
    .beginWith(Arrays.asList(plantName1, plantName2, plantName3))
    // Dependent job (only runs after all previous jobs in chain)
    .then(cache)
    .enqueue()
```

Battery Manager

Android runs on limited capabilities devices

- It is crucial to use the battery wisely
- The battery service gives us information about the power of the system
- Get it with:

```
val batteryManager = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
```

However you don’t handle battery monitoring by calling directly its functions...

Battery Manager

The BatteryManager broadcasts a **sticky** intent (keeps the latest data in it, that's why the receiver is null) accessed by:

```
val batteryStatus: Intent? =
    registerReceiver(null, IntentFilter(Intent.ACTION_BATTERY_CHANGED))
```

A non null receiver will be updated anytime the battery status changes…

```
<intent-filter>
<action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
<action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
<action android:name="android.intent.action.BATTERY_LOW"/>
<action android:name="android.intent.action.BATTERY_OKAY"/>
</intent-filter>
```

Battery Manager

From the intent you can obtain a lot of information about the battery:

```
val status = batteryStatus?.getIntExtra(BatteryManager.EXTRA_STATUS, -1)

val isCharging = (
      status == BatteryManager.BATTERY_STATUS_CHARGING ||
      status == BatteryManager.BATTERY_STATUS_FULL
      )

if (batteryStatus != null)
      val batteryPercent =
            batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) * 100.0 /
            batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1)
```

Sensors

Any smartphone is equipped with a variety of sensors that can tell a lot about the *primary context*.

Get it with:

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

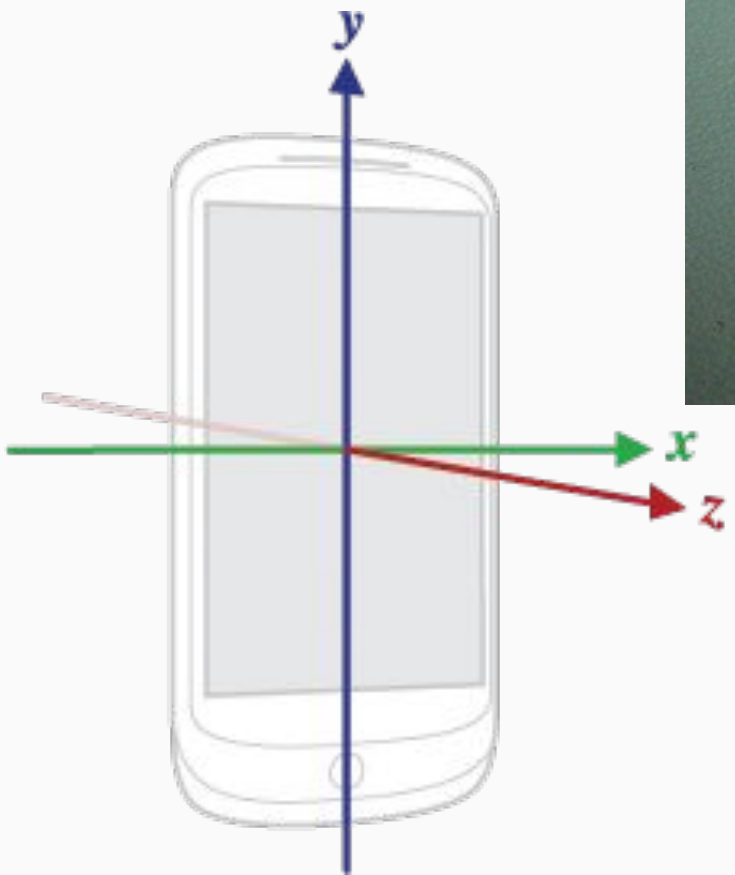
Various kinds of sensors

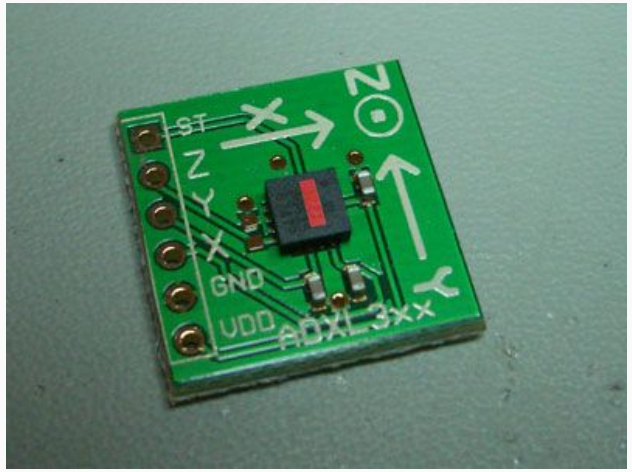
- Motion (accelerometer, gyroscope, ...)
- Environment (barometer, thermometer, photometer, ...)
- Position (compass, magnetometer, ...)

Sensors

Accelerometer

- To measure acceleration
- Given with 3-axes values
- Useful to inspect movements

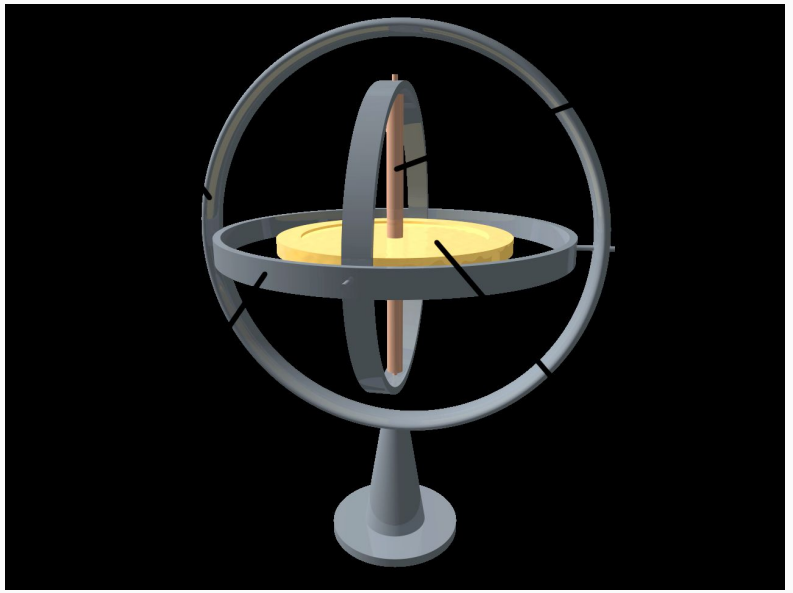




Sensors

Gyroscope

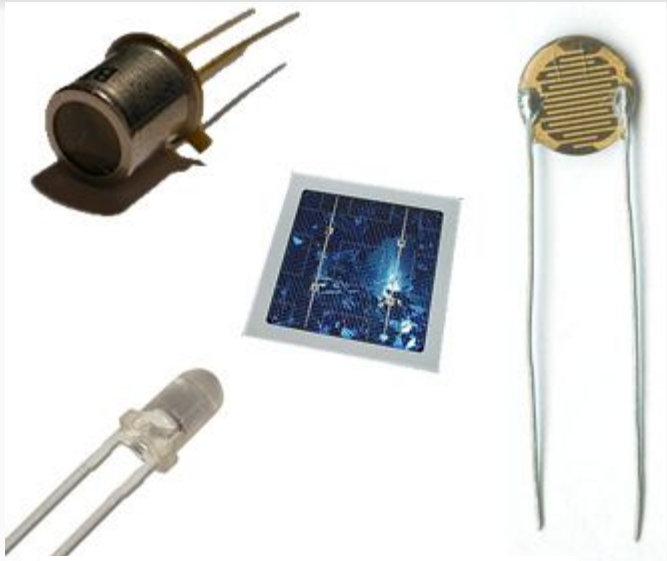
- To measure rotation
- Usually a spinning wheel or a spinning disk
- Gives angular speed

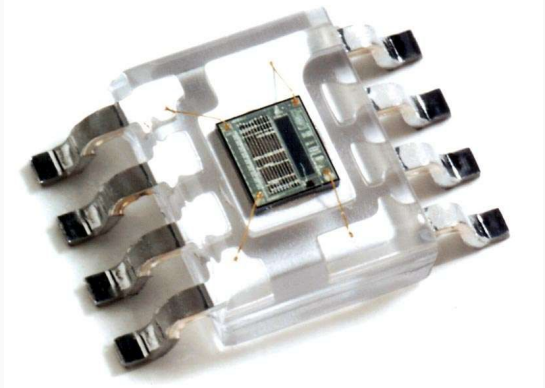


Sensors

Light Sensor

- Usually a photodiode or photoresistor
- When exposed to light, they let more current through
- More current, more light





Sensors

Proximity Sensor

- To measure distance from objects (a sonar or infrared)
- Useful to understand when the smartphone is in, for instance, a pocket
- Used to switch off screen during calls

Sensors

sensorManager.getSensorList(type: **Int**) (can be **Sensor.TYPE_ALL**)

Sensor	Type (Hardware/Software)	Used for
TYPE_ACCELEROMETER	Hardware	Acceleration along three axes (+ gravity)
TYPE_AMBIENT_TEMPERATURE	Hardware	Temperature
TYPE_GRAVITY	Can be both	Motion Detection
TYPE_GYROSCOPE	Hardware	Rotation
TYPE_LIGHT	Hardware	Ambient brightness
TYPE_LINEAR_ACCELERATION	Can be both	Acceleration along three axes (no gravity)
TYPE_MAGNETIC_FIELD	Hardware	Compass, indoor navigation
TYPE_ORIENTATION	Software	Obtaining device position
TYPE_PRESSURE	Hardware	Obtaining the height from sea level
TYPE_PROXIMITY	Hardware	Setting off the screen
TYPE_RELATIVE_HUMIDITY	Hardware	Humidity
TYPE_ROTATION_VECTOR	Can be both	Motion and Rotation detection

Sensors

- Not all smartphones are created equal
- Some carry a set of sensors some others don't
- Also different vendors offer different sensors with different capabilities…
  - getResolution()
  - getMaximumRange()
  - getPower()
  - getVendor()
  - getMinDelay()

Regardless, ***Sensors do not require permissions!!!***

Sensors

Each Sensor contains information about the vendor, type and others

- Implement SensorEventListener
  - onAccuracyChanged(sensor: Sensor?, accuracy: Int)
  - onSensorChanged(event: SensorEvent?)
- registerListener(listener: SensorEventListener, sensor: Sensor, rate: Int)
  [do this in the onResume (and the unregisterListener in the onPause)]
  rate is one of
  - SENSOR_DELAY_NORMAL
  - SENSOR_DELAY_FASTEST
  - SENSOR_DELAY_GAME
  - SENSOR_DELAY_UI

Sensors

Example for Light Sensor:

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val sensorLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
sensorManager.registerListener(this, sensorLight, SensorManager.SENSOR_DELAY_UI)

override fun onSensorChanged(event: SensorEvent?) {
      when (event?.sensor?.type) {
         Sensor.TYPE_LIGHT ->
            // React to light change…
         Sensor.TYPE_ACCELEROMETER ->
            // …
      }
   }
```

Sensors

In addition to the hardware sensors, there are a number of virtual sensors:

- Gravity
- Linear Acceleration
- Orientation
- Rotation

“Readings from hardware sensors are computed to offer aggregated data”

Sensors have their inherent challenges:

- **Bias/Drift**: Sensor reading is off by a constant value
- **Settling time**: Initial sensor readings may be inaccurate
- **Noise**: Data can’t report a reliable and steady value
- **Interference**: From the environment

Activity Recognition

Detecting the user activity is of paramount importance

- Start vehicle related apps while the user is driving
- Start tracking distances if the user is walking
- Activate fitness apps

How?

- Reading raw values and use machine learning models
  - Raw sensor usage… no permission!
- Exploit **Activity Recognition API**
  - Permission:

```
<uses-permission android:name="com.google.android.gms.permission.ACTIVITY_RECOGNITION"/>
```

Activity Recognition

You need the dependency first:

```
implementation("com.google.android.gms:play-services-location:21.2.0")
```

Make then a list with the preferred **ActivityTransitions** to monitor:

```
val transitions = ArrayList<ActivityTransition>()
transitions.add(
        ActivityTransition.Builder()
                .setActivityType(DetectedActivity.IN_VEHICLE)
                .setActivityTransition(ActivityTransition.ACTIVITY_TRANSITION_ENTER)
                .build()
)
transitions.add(
        …
)
```

Activity Recognition

Build the request:

```
val request = ActivityTransitionRequest(transitions)
```

Register the request:

```
ActivityRecognition.getClient(this) // This needs the context
    .requestActivityTransitionUpdates(request, myPendingIntent)
    .addOnSuccessListener{ /* Request inserted correctly */ }
    .addOnFailureListener{ /* Request not inserted */ }
```

We are passing in a pending intent which is the one that will be fired anytime one of the requested activity transitions occurs.

Much like Geofencing…

Activity Recognition

Get the events via, for instance, a broadcast receiver:

```
if (ActivityTransitionResult.hasResult(intent)) {
      val result = ActivityTransitionResult.extractResult(intent)
      val eventList = result.getTransitionEvents() // chronological sequence of events....
}
```

- Events are ordered…

Remember to de-register:

```
ActivityRecognition.getClient(this).removeActivityTransitionUpdates(myPendingIntent)
```

Other System Services

Audio Service

Able to

- select a stream and control sound
- adjust the volume
- change ring type
- play effects

Other System Services

Telephony Service

- Interacts with calls
- Get it with

```
getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager
```

- Ask the device about call information
  - getCallState()
  - getDataState()
  - getDataActivity()
  - getNetworkType()
  - isNetworkRoaming()

Other System Services

Connectivity Service

- Check device network state
- Get it with

```
getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
```

- Check WI-FI, GPRS, LTE
- Notify connection changes
- Needs
  - android.permission.ACCESS_NETWORK_STATE
  - android.permission.CHANGE_NETWORK_STATE

Other System Services

WiFi Service

- Get it with

```
getSystemService(Context.WIFI_SERVICE) as WifiManager
```

- Check Wi-Fi
  - getWifiState()
    - Returns WIFI_STATE_DISABLED, WIFI_STATE_DISABLING, WIFI_STATE_ENABLED, WIFI_STATE_ENABLING, WIFI_STATE_UNKNOWN
  - isWifiEnabled() / setWifiEnabled()
- Lists all the configured wifi connections
  - getConfiguredNetworks()

Other System Services

WiFi Service

- Check/edit wi-fi connection
  - addNetwork(config: WifiConfiguration)
  - updateNetwork(config: WifiConfiguration)
  - removeNetwork(netid: int)
- Scan for wi-fi networks
  - startScan()
- Be notified about wi-fi changes
  - Broadcast Intent: SCAN_RESULTS_AVAILABLE_ACTION
    - Call getScanResults()

Questions?

federico.montori2@unibo.it