LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Praktikum Mobile und Verteilte Systeme

Background Tasks and Storage Options

Prof. Dr. Claudia Linnhoff-Popien et al.

Sommersemester 2019

Background Tasks – Why?

- Main thread is in charge of handling
  - UI
  - User interactions
  - Receiving lifecycle events

- If there is too much work the app appears to hang or slow down

When do I need a Background Task?

- For long-running computations and operations
  - decoding a bitmap
  - accessing the disk
  - performing network requests
  - …
  - In general, anything that takes more than a few milliseconds

- Tasks may also run even when the user is not actively using the app
  - syncing periodically with a backend server
  - fetching new content

AsyncTasks for publishing to the UI thread

- The easiest solution for running tasks in the background are AsyncTasks
  - should only be used for short operations (a few seconds at the most.)
  - Examples:
    - Downloading small content on a button press
    - Calculations and other bigger operations on a button press
    - …

➔ This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask<Params, Progress, Result>

- An asynchronous task is defined by 3 generic types
  - Params
  - Progress
  - Result

- and 4 steps
  - onPreExecute
  - doInBackground
  - onProgressUpdate
  - onPostExecute.

Downloading an Image with AsyncTask

```
private class DownloadImageTask : AsyncTask<URL, Int, Long>() {

    override fun doInBackground(vararg urls: URL): Long? {
        val count = urls.size
        var totalSize: Long = 0
        for (i in 0 until count) {
            totalSize += Downloader.downloadImage(urls[i])
            publishProgress((i / count.toFloat() * 100).toInt())
        }
        return totalSize
    }

    protected override fun onProgressUpdate(vararg progress: Int) {
        setProgressPercent(progress[0])
    }

    override fun onPostExecute(result: Long?) {
        showDownloadedImages()
     }
}
```

Inline definition

```
fun downloadButtonClicked() {

    val myAnonymousAsyncTask = object : AsyncTask<ArrayList<URL>, Void, ArrayList<Bitmap>>() {
        override fun doInBackground(vararg params: ArrayList<URL>?): ArrayList<Bitmap> {
            return downloadAllThoseImages(params)
        }

        override fun onPostExecute(result: ArrayList<Bitmap>?) {
            showAllImages(result)
        }
    }

    myAnonymousAsyncTask.execute()
}
```

Bigger background tasks and their challenges

- Not all background tasks publish to the UI thread
- Some background tasks may take very long

- Background tasks consume a device's limited resources, like RAM and battery.

➔ This may result in a poor experience for the user if not handled correctly.

➔ To maximize battery and enforce good app behavior, Android restricts background work when the app (or a foreground service notification) is not visible to the user.

Android APIs for bigger background tasks

- Android offers different solutions to different tasks
  - DownloadManager
  - Foreground Service
  - WorkManager
  - AlarmManager

DownloadManager

- If your app is performing long-running HTTP downloads
- Clients may request that a URI be downloaded to a particular destination file that may be outside of the app process
- The download manager will conduct the download in the background, taking care of HTTP interactions and retrying downloads after failures or across connectivity changes and system reboots.

WorkManager

- Triggered by system conditions
- For work that is deferrable and expected to run even if your device or application restarts
- WorkManager is an Android library that runs background tasks when the conditions (like network availability and power) are satisfied.
- WorkManager offers a backwards compatible (API level 14+) API
- Example: You need to run a job every hour, but *not* at a specific time

AlarmManger

- If you need to run a job at a *precise* time
- Launches your app, if necessary, to do the job at the time you specify

➔ If your job does not need to run at a precise time, WorkManager is a better option

➔ If you need to run a job every hour you should use WorkManager to set up a recurring job

Android Services

- A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface.
- Another application component can start a service, and it continues to run in the background even if the user switches to another application.
- Additionally, a component can bind to a service to interact with it.
- Examples: Network transactions, play music, perform file I/O, or interact with a content provider

Types of Services

- These are the three different types of services:
  - Foreground
  - Background
  - Bound

Foreground Services

- For user-initiated work that need to run immediately and must execute to completion
- Using a foreground service tells the system that the app is doing something important and it shouldn’t be killed
- Foreground services are visible to users via a non-dismissible notification in the notification tray.

➔ Example: Continous Location Tracking

Background Services (deprecated?)

- 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.

- **Note:** If your app targets API level 26 or higher, the system imposes restrictions for background tasks
- In most cases like this, your app should use a scheduled job instead.

Bound Services

- A service is bound when an application component binds to it by calling bindService().
- Allows components to interact with the service
- 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.

Creating a Service

- IntentService as the Base class
- provides structure for running an operation on a background thread.

- An IntentService isn't affected by most lifecycle events
  → It continues to run in circumstances that would shut down an AsyncTask

- Limitations
  - Can't interact directly with your user interface
  - Can't be interrupted

Declaring your Service in the Manifest

```
<application
      android:icon="@drawable/icon"
      android:label="@string/app_name">
      ...
      <!--
         Because android:exported is set to "false",
         the service is only available to this app.
      -->
      <service
         android:name=".RSSPullService"
         android:exported="false"/>
      ...
   </application>
```

➔ You must declare all services in your application's manifest file, just as you do for activities and other components.

Implement your Service

- To create an IntentService component for your app, define a class that extends IntentService, and within it, define a method that overrides onHandleIntent().

```
class RSSPullService : IntentService(RSSPullService::class.simpleName)

    override fun onHandleIntent(workIntent: Intent) {
        // Gets data from the incoming Intent
        val dataString = workIntent.dataString
        ...
        // Do work here, based on the contents of dataString
        ...
    }
}
```

Interact with a Service through Broadcasts

Broadcasts

- Android apps can send or receive broadcast messages from the Android system and other Android apps
- Publish-subscribe design pattern
- The Android system sends broadcasts when system events occur
  - system boots up
  - device starts charging
- Apps can also send custom broadcasts
  - e.g. some new data has been downloaded
- Apps can register to receive specific broadcasts

Creating a BroadcastReceiver (1)

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

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

<receiver android:name=".MyCustomBroadcastReceiver"  android:exported="false">
   <intent-filter>
      <action android:name="de.lmu.ifi.mobile.MY_IMAGE_EVENT"/>
   </intent-filter>
</receiver>
```

Creating a BroadcastReceiver (1.1)

OR register a receiver with a context

- Receive broadcasts as long as their registering context is valid
- For an example, if you register within an Activity context, you receive broadcasts as long as the activity is not destroyed.
- If you register with the Application context, you receive broadcasts as long as the app is running.

```
val br: BroadcastReceiver = MyBroadcastReceiver()

val filter = IntentFilter("de.lmu.ifi.mobile.MY_IMAGE_EVENT")
registerReceiver(br, filter)
```

Creating a BroadcastReceiver (2)

Subclass BroadcastReceiver and implement onReceive(Context, Intent).

```
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()
            }
        }
    }
}
```

Sending a Broadcast

```
Intent().also { intent ->
    intent.setAction(“de.lmu.ifi.mobile.MY_IMAGE_EVENT")
    intent.putExtra("data", myImage)
    sendBroadcast(intent)
}
```

Interact with a Service by binding to it

- A bound service is the server in a client-server interface.
- It allows components (such as activities) to bind to the service, send requests, receive responses
- A bound service typically lives only while it serves another application component and does not run in the background indefinitely.

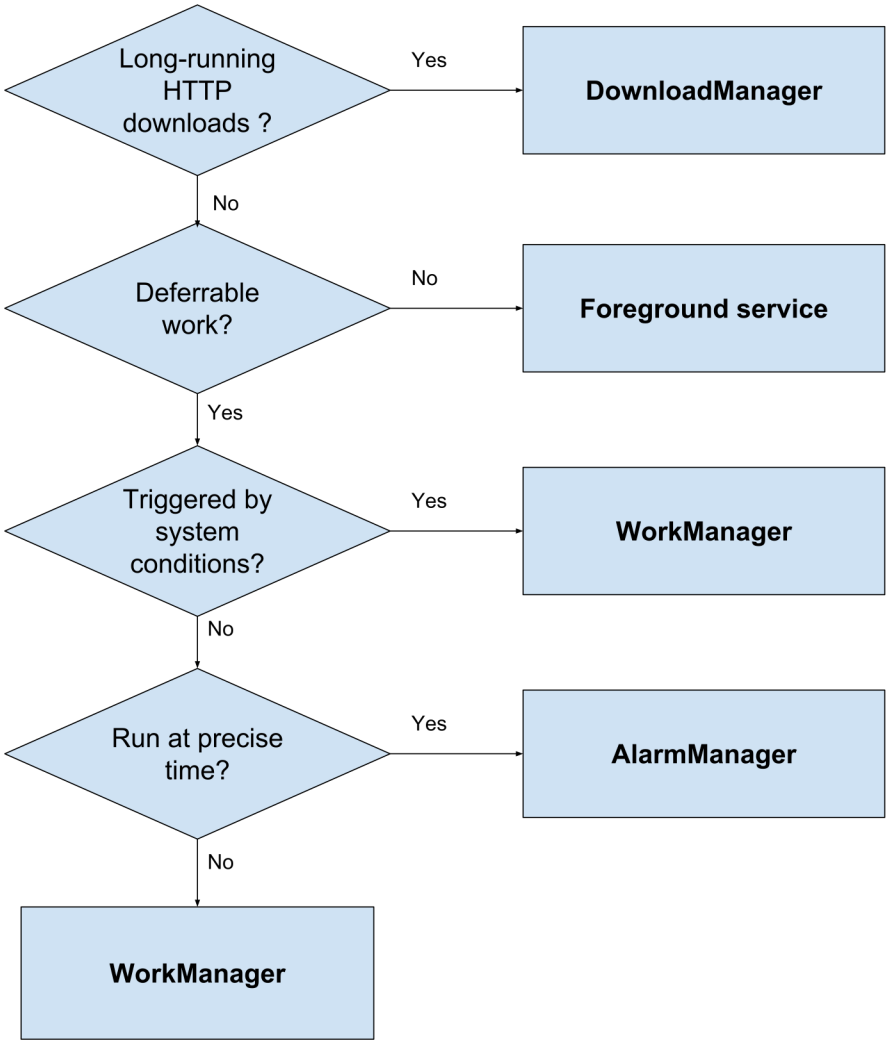
Creating a BoundService

```
class LocalService : Service() {
    // Binder given to clients
    private val binder = LocalBinder()

    ...

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    inner class LocalBinder : Binder() {
        // Return this instance of LocalService so clients can call public methods
        fun getService(): LocalService = this@LocalService
    }

    override fun onBind(intent: Intent): IBinder {
        return binder
    }
}
```

Bind to a Service

```
class BindingActivity : Activity() {
    private lateinit var mService: LocalService
    private var mBound: Boolean = false

    /** Defines callbacks for service binding, passed to bindService()  */
    private val connection = object : ServiceConnection {

        override fun onServiceConnected(className: ComponentName, service: IBinder) {
            // We've bound to LocalService, cast the IBinder and get LocalService instance
            val binder = service as LocalService.LocalBinder
            mService = binder.getService()
            mBound = true
        }

        override fun onServiceDisconnected(arg0: ComponentName) {
            mBound = false
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
    }

    override fun onStart() {
        super.onStart()
        // Bind to LocalService
        Intent(this, LocalService::class.java).also { intent ->
            bindService(intent, connection, Context.BIND_AUTO_CREATE)
        }
    }

    override fun onStop() {
        super.onStop()
        unbindService(connection)
        mBound = false
    }
}
```

Choosing the right solution (4)



Data and file storage

- Android provides several options for you to save your app data.

- The solution you choose depends on your specific needs
  - How much space your data requires
  - What kind of data you need to store
  - Whether the data should be private to your app

Storage Options

- Internal file storage
- External file storage
- Shared preferences
- Databases

Internal file storage

- Default: Files saved to the internal storage are private
  - other apps cannot access them
  - nor can the user without root

  ➔ Good for app data that the user doesn't need to directly access

- The system provides a private directory on the file system for each app where you can organize any files your app needs.

What happens on uninstall?

- When the user uninstalls your app, the files saved on the internal storage are removed.
- Because of this behavior, you should not use internal storage to save anything the user expects to persist independently of your app
- Exmample: Your app allows users to capture photos
  - the user would expect that they can access those photos even after they uninstall your app
  - You should instead use the MediaStore API

Save to the internal storage

```
val file = File(context.filesDir, filename)


val filename = "myfile"
val fileContents = "Hello world!"
context.openFileOutput(filename, Context.MODE_PRIVATE).use {
        it.write(fileContents.toByteArray())
}
```

External file storage

- Every Android device supports a shared "external storage"
- „External“ because it's not guaranteed to be accessible
- Users can mount it to a computer as an external storage device
- It might even be physically removable (such as an SD card)
- External storage is world-readable and can be modified by the user

Check for availability on external Storage

- Before you access a file in external storage check the availability of
  - external storage directories
  - the files you are trying to access
- Use external storage for data that should be accessible to other apps and saved even if the user uninstalls your app
- The system provides standard public directories for these kinds of files
- You can also save files to the external storage in an app-specific directory that the system deletes when the user uninstalls your app.
  - If you need more space
  - Still world-readable

Save to the external storage (1)

Requires permission WRITE_EXTERNAL_STORAGE

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

Check if the external storage is available

```
/* Checks if external storage is available for read and write */
fun isExternalStorageWritable(): Boolean {
   return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED
}

/* Checks if external storage is available to at least read */
fun isExternalStorageReadable(): Boolean {
    return Environment.getExternalStorageState() in
      setOf(Environment.MEDIA_MOUNTED, Environment.MEDIA_MOUNTED_READ_ONLY)
}
```

Save/read the external storage (2)

```
fun getPublicAlbumStorageDir(albumName: String): File? {
    // Get the directory for the user's public pictures directory.
    val file = File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName)
    if (!file?.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created")
    }
    return file
}
```

Shared Preferences

- “Shared preferences" is a bit misleading because it is not strictly for saving "user preferences" (such as what ringtone a user has chosen)
- You can save any kind of simple data (such as the user's high score)
- If you don't need to store a lot of data and it doesn't require structure, you should use SharedPreferences
- Read and write persistent key-value pairs of primitive data types: booleans, floats, ints, longs, and strings
- Key-value pairs are written to XML files that persist across user sessions

➔ However, if you do want to save user preferences:
Use the AndroidX Preference Library to build a settings screen and automatically persist the user's settings.

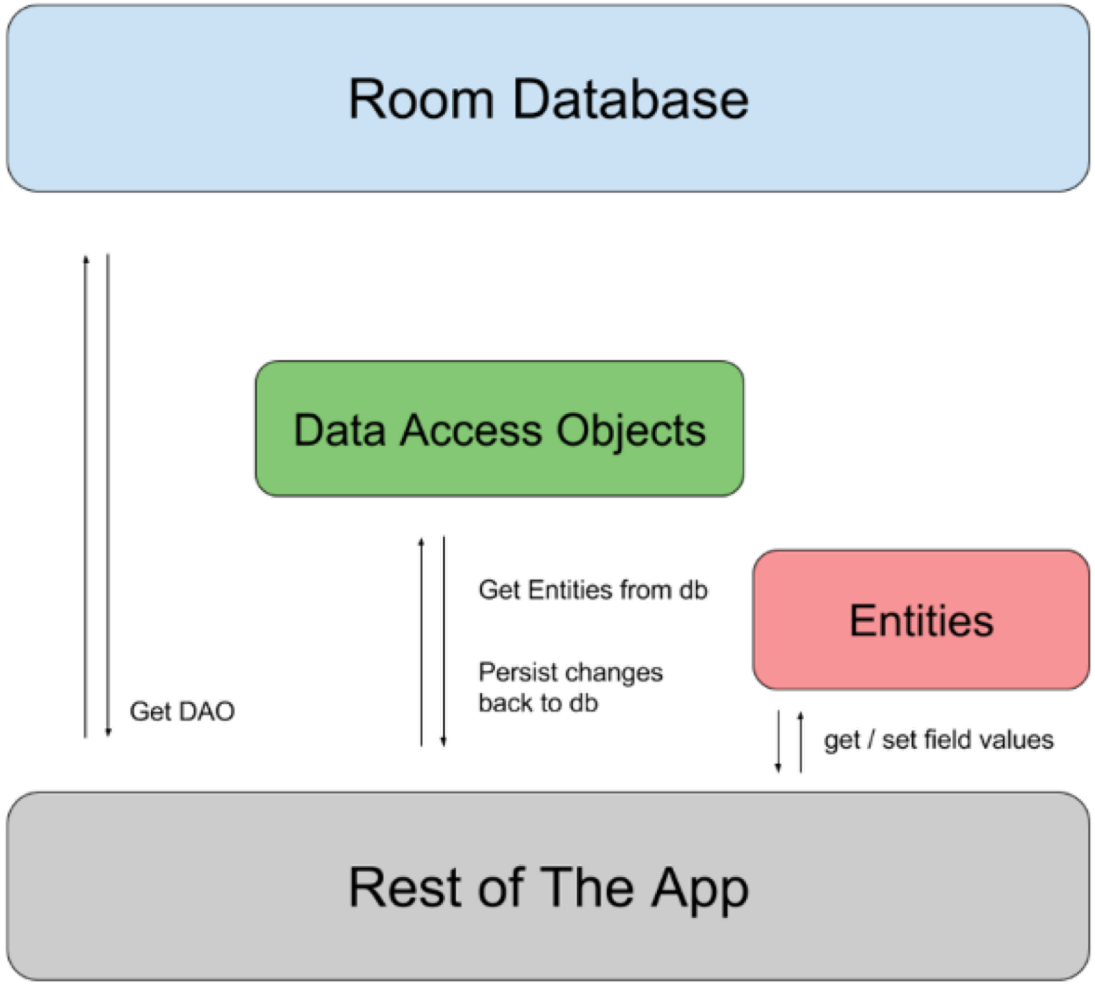
Read/write to SharedPreferences

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return

// Write
with (sharedPref.edit()) {
    putInt("My_Int_Key", newHighScore)
    commit()
}

// Read
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
val highScore = sharedPref.getInt("My_Int_Key", 20)
```

Databases

- Android provides full support for SQLite databases.
- Any database you create is accessible only by your app

- Recommended to use the Room persistence library
  - provides an object-mapping abstraction layer that allows fluent database access while harnessing the full power of SQLite.
  - Compile-time verification
  - Automatic scheme changes
  - No boilerplate code


Room Database
Data Access Objects
Get DAO
Get Entities from db
Persist changes back to db
Entities
get / set field values
Rest of The App

Entity

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Data Access Objects (DAO)

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Room Database

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

! Whenever you change the scheme of your database (e.g. the user gets the field „phone_number“) ➔ Increase the version

Get the database object

```
val db = Room.databaseBuilder(
        applicationContext,
        AppDatabase::class.java, "database-name"
    ).build()
```

Test your database

```
@RunWith(AndroidJUnit4::class)
class SimpleEntityReadWriteTest {
    private lateinit var userDao: UserDao
    private lateinit var db: TestDatabase

    @Before
    fun createDb() {
        val context = ApplicationProvider.getApplicationContext<Context>()
        db = Room.inMemoryDatabaseBuilder(
                context, TestDatabase::class.java).build()
        userDao = db.getUserDao()
    }

    ...

    @Test
    @Throws(Exception::class)
    fun writeUserAndReadInList() {
        val user: User = TestUtil.createUser(3).apply {
            setName("george")
        }
        userDao.insert(user)
        val byName = userDao.findUsersByName("george")
        assertThat(byName.get(0), equalTo(user))
    }
}
```

Useful links

- Async Task
  https://developer.android.com/reference/android/os/AsyncTask

- Background Tasks
  https://developer.android.com/guide/background/

- Data Storage
  https://developer.android.com/guide/topics/data/data-storage