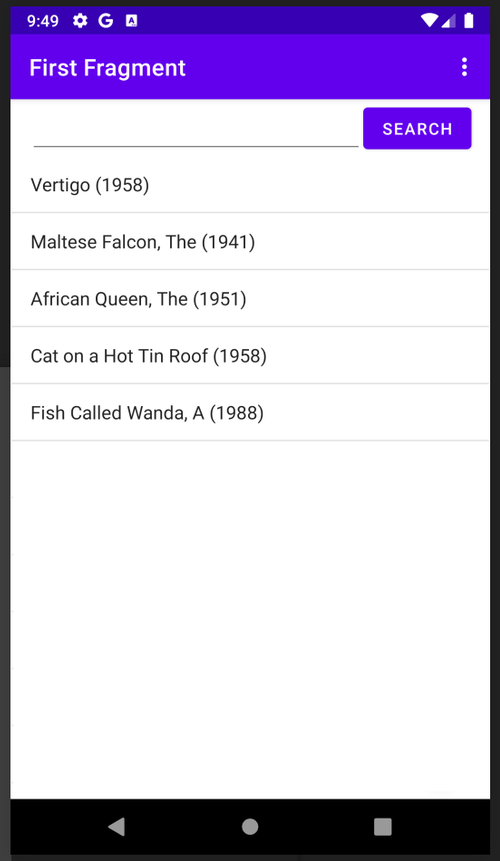
RECAP: EXERCISE DB APP I

Create a new Project DBApp containing a ListView, an EditText and a Button:

```
//The movie class only has one attribute name
val movies = mutableListOf(Movie("Vertigo (1958)"),
             Movie("Maltese Falcon, The (1941)"),
             Movie("African Queen, The (1951)"),
             Movie("Cat on a Hot Tin Roof (1958)"),
             Movie("Fish Called Wanda, A (1988)"))

//it is possible to use your own class (instead of
//String) in an ArrayAdapter
adapter = ArrayAdapter<Movie>(
  requireContext(),
  android.R.layout.simple_list_item_1,
  android.R.id.text1, movies)

//but keep in mind that the Movie class
//should implement toString!
```

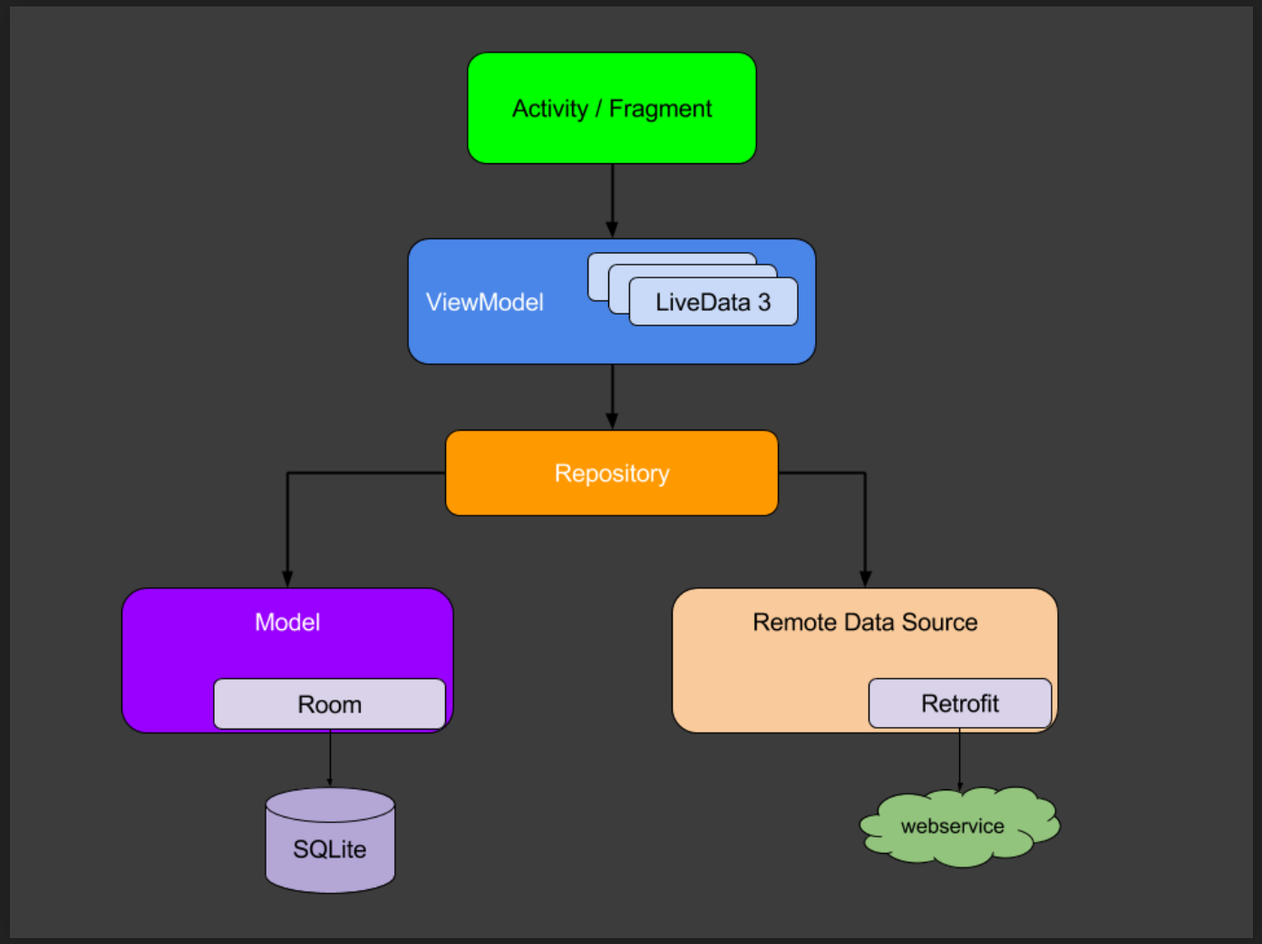


ANDROID VIEWMODEL

VIEWMODEL INTRODUCTION

- Until now, all data has been stored in the Fragment / Activity. That's ok for UI centered data, but the model should be stored in a model class.
- Android supports the MVVM pattern: A ViewModel connects the View with the model.
- The ViewModel can restore and save state as well as react on lifecycle events.
- Make sure that you never reference UI components in a (View)Model.

RECOMMENDED ARCHITECTURE



Source: developer.android.com

VIEWMODEL EXAMPLE I

The following example will be refactored to ModelView

```
//the main activity shows a TextView with the name of the user
//right now, the name is stored as a member variable:
class AFragment : Fragment() {
 {
  val name : String
  ...
}
```

VIEWMODEL EXAMPLE II

```
//Create a ViewModel for storing the name
class UserDataViewModel() : ViewModel() {
    var name = MutableLiveData<String>()

    init {
        name.value = "Peter Muster"
    }
}
```

VIEWMODEL EXAMPLE III

```
//Use the ViewModel in a Fragment
val model: UserDataViewModel by activityViewModels()
model.name.observe(viewLifecycleOwner, Observer<String>{newVal ->
  // update UI
  binding.textviewFirst.text = newVal
})
//whenever the name changes, the UI will be updated
//automatically
model.name.value = "Paul Muster" //will update the UI
```

VIEW MODEL INSTANCES I

- The view model instance is stored in the activity and can be retrieved using "activityViewModels()" (or viewModels() from the Activity).
- Note that "val model" in the example is a local variable! However, the instance is stored as an activity's member variable using the keyword by (Delegate, see here).
- You can access the model everywhere within the Activity / Fragment, ie:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
return when (item.itemId) {
R.id.action_settings -> {
 val model: UserDataViewModel by viewModels()
 Log.i("TAG", "${model.name}") //"Peter Muster"
 true }}}
```

VIEW MODEL DELEGATE I

- A Delegate (the class after "by") handles the request of an assigned object (in the next example: "p"), see here.
- It is a class that needs to define the methods getValue and setValue.
- It is automatically called whenever p is accessed (read or write access).

VIEW MODEL DELEGATE II

```
class Example {
    var p1: String by Delegate()} //a delegate example
class Delegate {
  operator fun getValue(thisRef: Any?,
   property: KProperty<*>): String {
    return "${thisRef} ${property.name}"}
  operator fun setValue(thisRef: Any?,
   property: KProperty<*>, value: String) {
    println("$value, assigned to '${property.name}' in $thisRef")}}
@Test
fun delegateTest() {
    val e = Example()
    e.p1 = "Hallo Welt" //will print Hallo Welt, assigned to 'p1' in
                        //ExampleUnitTest$Example@5a42bbf4
    print(e.p1) } // will print ExampleUnitTest$Example@5a42bbf4 p1
```

VIEW MODEL INSTANCE I

```
//the following definition
val model: UserDataViewModel by viewModels()

//will in fact call the method "viewModels":
inline fun <reified VM : ViewModel> ComponentActivity.viewModels(
    noinline factoryProducer: (() -> Factory)? = null
): Lazy<VM> {
 //create a default factory if the parameter is null
 val factoryPromise = factoryProducer ?: {
   defaultViewModelProviderFactory
 }
 //return the ViewModel (Note the viewModelStore reference. This
 //returns a store that's created as an activity's member variable:
 //private ViewModelStore mViewModelStore)
 return ViewModelLazy(VM::class, { viewModelStore }, factoryPromise)
}
```

LIVEDATA I

- LiveData propagates changes from the model to the view. LiveData provides the "observe-interface". By registering an Observer instance a view can be informed when the data change:

```
model.name.observe(this, Observer<String>{
  // update UI
  nameTextView.text = model.name
})
```

- It is lifecycle-aware, it doesn’t push changes if the observer is not in an active state (called from a component (ie. fragment) that is not active (ie. "onPause") will not trigger a change).

LIVEDATA II

- The subscription will automatically be removed when the component is destroyed. There is no need to unregister.
- Due to the lifecycle-aware feature, a recreated component (ie after rotation) will automatically receive the newest data.
- It is possible to extend LiveData, ie. for observing state changes of a service (see here).

LIVEDATA III

When using LiveData with a ListView adapter, observe is:

- *not* called when the content of a collection changes, ie an element is added to the list!
- called when a new List is assigned to the model value.

```
//init the adapter with an empty list
adapter = ArrayAdapter<Person>(..., mutableListOf<Person>())
...
model.persons.observe(viewLifecycleOwner,
  //note that the observer sends the new value as parameter
  Observer<MutableList<Person>>{ newVal ->
  adapter?.clear()
  adapter?.addAll(newVal) //addAll will call notifyDatasetChanged
})

... //this changing will call the observer
persons.value = mutableListOf<Person>()
```

EXERCISE DB APP II

Add a MovieViewModel to your app. Use the trick from the previous slide to update the ListView automatically when the model changes.

LAZY INITIALIZATION I

- In the example, the observed model was statically initialized with a literal:

```
var name = MutableLiveData<String>()
init {
    name.value = "Peter Muster"
}
```

LAZY INITIALIZATION II

- In practice, it is often needed to move the initialization to a later point in time. This can be accomplished by an optional value:

```
var name : MutableLiveData<String>? = null
```

- However, this will result in an optional variable that needs to be checked for null whenever used. Another way is to use the lazy method from Kotlin that is available for vals:

```
val name : MutableLiveData<String> by lazy {
loadName() //will be called when name is first accessed
}
```

LAZY INITIALIZATION III

Lazy initialization has a number of advantages:

- No optional attribute
- No initialization until the data is really needed
- No initialization if the data is not needed at all

Note that lazy only works with vals! There is a lateinit keyword for variables that basically does the same for variables (but without some important details like threadsafe).

EXERCISE DB APP III

Implement the search and filter the list of movies. Hints:

- Add a list "allMovies" to your ViewModel.
- filter the allMovies list and set the result to the movies LiveData model.

```
movies.value = allMovies.filter {movie ->
 movie.name.contains(searchText)
}?.toMutableList()
```

PERSISTENCE

SHARED PREFERENCES I

The easiest way to store data locally is by using SharedPreferences:

```
val settings = context.getSharedPreferences("prefsfile",
  Context.MODE_PRIVATE)

//it is important that you get an editor reference!
val editor = settings.edit()

//save strings, booleans, floats, ints, longs, stringsets
editor.putString("USER_NAME", "john.ford@hollywood.com")

//and commit your changes
editor.commit()
```

SHARED PREFERENCES II

Reading preferences is a two liner:

```
val settings = getSharedPreferences("prefsfile",
    Context.MODE_PRIVATE)

val defaultUser = settings.getString("USER_NAME",
  "no-user@no-domain.com")
```

SQLITE

SQLite is an SQL database developed for smaller devices/scenarios:

- It only needs one file to store the data
- It is self-contained and serverless
- It supports transactions

ROOM

- Google recommends to use the library Room for accessing SQLite.
- Insert the following entry to your build.gradle:

```
plugins {
 ...
  id 'kotlin-kapt'}
...
implementation "androidx.room:room-runtime:2.3.0"
implementation "androidx.room:room-ktx:2.3.0"
    kapt "androidx.room:room-compiler:2.3.0"
```

ROOM: ENTITY

```
@Entity
class Person(
    @PrimaryKey(autoGenerate = true) val uid: Int = 0,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

ROOM: DAO I

```
//DAO: Database Access Object
@Dao
interface PersonDao {
    @Query("SELECT * FROM person")
    fun getAll(): List<Person>

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

    @Query("SELECT * FROM person WHERE first_name IN (:names)")
    fun loadAllByFirstName(names: List<String>): List<Person>
```

ROOM: DAO II

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

  @Insert
  fun insertAll(vararg persons: Person)

  @Delete
  fun delete(person: Person)
}
```

ROOM: APPDATABASE

```
@Database(entities = arrayOf(Person::class),
          version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun personDao(): PersonDao
}
```

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

COROUTINES

- It is not allowed to access the database in the main thread (same goes for internet connections).
- AsyncTask is the old way of dealing with this problem. However, AsyncTask is cumbersome. Coroutines solve this task in a more elegant way.

COROUTINES: INSTALLATION

Add the following line to build.gradle:

```
implementation "androidx.lifecycle:lifecycle-runtime-ktx:2.3.1"
```

COROUTINES: METHOD DEFINITION

Use keyword suspend:

```
suspend fun insertUsers() {
  db?.personDao()?.insertAll(
    Person(firstName="Peter", lastName="Muster"),
    Person(firstName="Paul", lastName="Muster"))
}
suspend fun readUsers() {
  //we are in a background thread and need to call postValue
  persons.postValue(db?.personDao()?.getAll()?.toMutableList())
}
```

COROUTINES: CALL ASYNC

Call this method from our Activity/Fragment:

```
//lifecycleScope is an attribute from the Activity/Fragment
//Note that you need to define the default dispatcher explicitly
//otherwise the method will be called on the main thread
lifecycleScope.launchWhenStarted {
  withContext(Dispatchers.Default) {
    model.insertUsers()
  }
}
//readData is now executed in a different thread and won't block
//the main ui thread
```

EXERCISE DB APP IV

Implement a DB to the DB App:

- Define a DAO where movies can be retrieved from and written to the database (getAll, InsertAll)
- Implement the AppDatabase class
- Add three methods in the ViewModel: initDB, readMovies and writeMovies

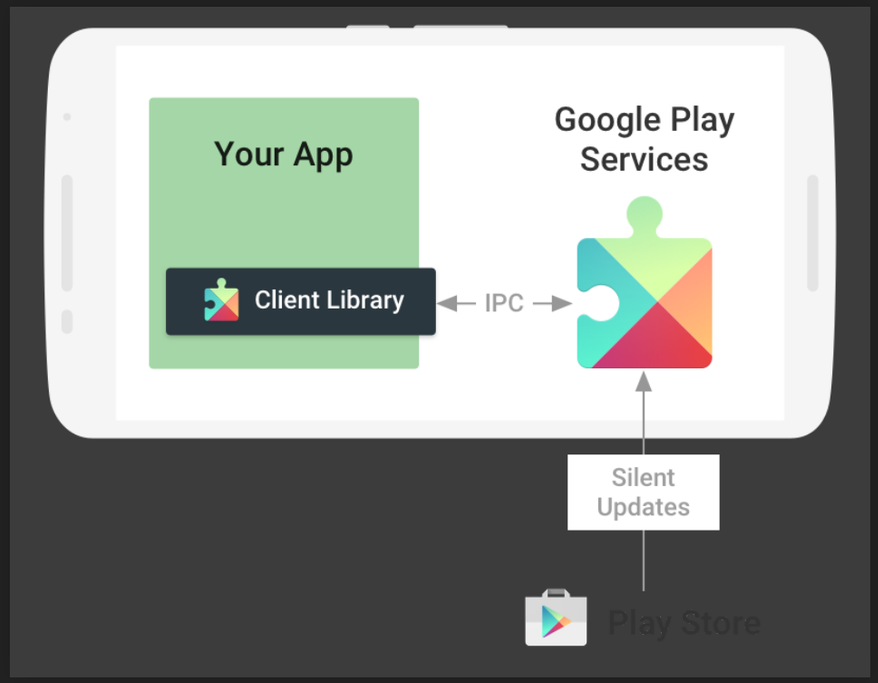
```
//writing all movies from a list can be done with one line:
db?.movieDao()?.insertAll(*movies.toTypedArray())

//after you have read the data, you need to post them (because you)
//are in your own thread (movies is a LiveData object)
movies.postValue(allMovies)
```

EXERCISE DB APP V

Finally, dowload the movies.json file from moodle, read it with Klaxon and write all movies to the database. Note that it will take a minute or two for this process.

ANDROID SERVICES

SERVICE ARCHITECTURE I



Source: Google Android Play Service Documentation

SERVICE ARCHITECTURE II

- The Play service is executed as background task.
- New service versions are delivered by the Play Store. This ensures that the newest version is available (Google controls updates).
- Lately, Play Services have been criticized for sending personal data to Google servers regularly (see here)

AVAILABLE SERVICES

Google+, Google Account Login, Google Actions, Base Client Library, Google Sign In, Google Analytics, Google Awareness, Google Cast, Google Cloud Messaging, Google Drive, Google Fit, Google Location and Activity Recognition, Google Maps, Google Mobile Ads, Google Places, Mobile Vision, Google Nearby, Google Panorama Viewer, Google Play Game services, SafetyNet, Android Pay, Android Wear

SETUP

- Add the desired service to your build.gradle, ie:

```
implementation 'com.google.android.gms:play-services-vision:11.8.0
```

- Check if the service is available using

```
val googleAPI = GoogleApiAvailability.getInstance()
val isAvailable =
 googleAPI.isGooglePlayServicesAvailable(applicationContext)
if (isAvailable != ConnectionResult.SUCCESS) {
 //show an error message if it is not available
}
```

- Use the Service Client Api (some Apis require authentication)

BARCODE SCANNER I

```
//CAMERA permission is needed!
val detector = BarcodeDetector.Builder(activity.applicationContext)
  .setBarcodeFormats(Barcode.EAN_13)
  //Barcode.DATA_MATRIX | Barcode.QR_CODE)
  .build()
val cameraSource =
    CameraSource.Builder(activity.applicationContext, detector)
  .setFacing(CameraSource.CAMERA_FACING_BACK)
  .setRequestedFps(15.0f)
  .setAutoFocusEnabled(true)
  .build()
if (!detector.isOperational) {
  return "Could not set up the detector!"
}
```

BARCODE SCANNER II

```
detector.setProcessor(object : Detector.Processor<Barcode> {
  override fun release() {}
  override fun receiveDetections(detections: Detections<Barcode>) {
      if (detections.detectedItems.size() > 0) {
          val barcode = detections.detectedItems.valueAt(0)
          val value = barcode.rawValue
          Log.i("MainActivity", "found barcode: $value")
          activity.runOnUiThread { //stop the camera
              cameraSource.stop()
              Log.i("MainActivity", "camera stopped")
          }
      } else {
          Log.i("MainActivity", "scanning...")
      }
  }
})
```

ANDROID ACTIVITY RECOGNITION

- Recognizes the user's activity changes, ie. driving, walking or biking.
- Uses device sensors to detect the activity. The data is processed using machine learning models.
- Provides two different APIs: The Activity Recognition Transition API and the Activity Recognition Sampling API. Transition is the prefered way, Sampling can be used to finegrain the frequency of the recognition (used in the example on the next slides simply because it is a better demo).

SETUP I

- Add the service to your build.gradle, ie:

```
implementation 'com.google.android.gms:play-services-location:1
```

- Add the needed permission:

```
<= API 28:
com.google.android.gms.permission.ACTIVITY_RECOGNITION
> API 28:
android.permission.ACTIVITY_RECOGNITION
```

- Check if the service is available:

```
val googleAPI = GoogleApiAvailability.getInstance()
val isAvailable =
 googleAPI.isGooglePlayServicesAvailable(applicationContext)
if (isAvailable != ConnectionResult.SUCCESS) {
 //show an error message if it is not available
}
```

SETUP II

```
//store the client as member variable so that it won't be deleted
activityRecognitionClient = ActivityRecognition.getClient(context)
//create a request by specifying all transitions (next slide)
val request = ActivityTransitionRequest(createAcitivtyTransitions())
//create an Intent and set a unique action
//string (RECOGNITION_ACTION)
val intent = Intent("RECOGNITION_ACTION")
//start the recognition
val task = activityRecognitionClient?.requestActivityUpdates(0,
PendingIntent.getBroadcast(context, 0, intent, 0))
```

SETUP III

```
fun createAcitivtyTransitions() : MutableList<ActivityTransition> {
  val transitions = mutableListOf<ActivityTransition>()
  transitions +=
      ActivityTransition.Builder()
          .setActivityType(DetectedActivity.IN_VEHICLE)
          .setActivityTransition(
             ActivityTransition.ACTIVITY_TRANSITION_ENTER)
          .build()
  transitions +=
      ActivityTransition.Builder()
          .setActivityType(DetectedActivity.IN_VEHICLE)
          .setActivityTransition(
             ActivityTransition.ACTIVITY_TRANSITION_EXIT)
          .build()
  return transitions
}
```

RETRIEVE RECOGNITIONS I

- The service will send an intent to the app. Therefore, a BroadcastReceiver is needed to retrieve the intent.
- The BroadcastReceiver has to implement the onReceive method:

```
class MyBroadcastReceiver() : BroadcastReceiver() {

override fun onReceive(context: Context, intent: Intent) {
if (intent != null &&
   intent.getAction() == "RECOGNITION_ACTION") {
    if (ActivityRecognitionResult.hasResult(intent)) {
      val intentResult = ActivityRecognitionResult
          .extractResult(intent);

      val detectedActivity = intentResult?.mostProbableActivity
      //do something
}}}
```

RETRIEVE RECOGNITIONS II

```
//finally, register the broadcast receiver
val intentFilter = IntentFilter("RECOGNITION_ACTION")
broadcastReceiver = MyBroadcastReceiver(logger)
context.registerReceiver(broadcastReceiver, intentFilter)
```