Advanced Mobile Application Development
Week 14: Android and Realm

Realm and Android

https://www.mongodb.com/docs/realm/sdk/java/
As Realm is an open-source, cross-platform object database that provides local data persistence easily and efficiently. Data in Realm is auto-updating so your data is always up to date and never needs to be refreshed.
Note that Realm does not support Java outside of Android.

Getting Started/Installation

https://www.mongodb.com/docs/realm/sdk/java/install/
To add Realm to an Android app you need to install Realm as a Gradle plug-in:

1. Add the realm-gradle-plugin dependency to a buildscript.repositories block in the project gradle file
2. Add the kotlin-kapt plugin and apply the realm-android plug-in in the application gradle file
3. Add needed dependencies

Initialization

https://www.mongodb.com/docs/realm/sdk/java/quick-start-local/
You must initialize Realm before you can use it in your app. This often is done in a subclass of your Application class. If you create your own application subclass, you must add it to the app's AndroidManifest.xml.

Initialize Realm:

```
Realm.init(this)
```

Then in the class that will interface with the Realm database you get the Realm instance:

```
val realmDB:Realm = Realm.getDefaultInstance()
```

When we're finished with a Realm instance, it is important that you close it with a call to close() to deallocate memory and release any other used resource.

Configuration

To control how Realms are created, use a RealmConfiguration object. The minimal configuration usable by Realm is:

```
val realmConfig = RealmConfiguration.Builder().build()
```

That configuration—with no options—uses the Realm file default.realm located in Context.getFilesDir.
Setting a default configuration in your Application subclass makes it available in the rest of your code.
You can also call the initializer that lets you name the database.
You can have multiple RealmConfiguration objects, so you can control the version, schema and location of each Realm independently.
Realm objects are live, auto-updating views into the underlying data; you never have to refresh objects.

Models

https://www.mongodb.com/docs/realm/sdk/java/fundamentals/object-models-and-schemas/
Realm model classes are very similar to Kotlin classes but they must extend the RealmObject base class.
Realm supports the following field types: boolean, byte, short, int, long, float, double, String, Date and

byte[] as well as the boxed types Boolean, Byte, Short, Integer, Long, Float and Double. Subclasses of RealmObject and RealmList are used to model relationships.

The @Required annotation tells Realm to enforce checks on these fields and disallow null values. Only Boolean, Byte, Short, Integer, Long, Float, Double, String, byte[] and Date can be annotated with Required.

The @PrimaryKey annotation tells Realm that the field is the primary key which means that field must be unique for each instance. Supported field types can be either string (String) or integer (byte, short, int, or long) and its boxed variants (Byte, Short, Integer, and Long). Using a string field as a primary key implies that the field is indexed (i.e. it will implicitly be marked with the annotation @Index). Indexing a field makes querying it faster, but it slows down the creation and updating of the object, so you should be careful about the number of fields in your object that you @Index.

You can also define relationships between RealmObject classes.

Realm Objects
RealmObjects are live, auto-updating views into the underlying data so you don't need worry about refreshing or re-fetching realm objects before updating the UI.

Transactions
All write operations to Realm (create, update and delete) must be wrapped in transactions. A write transaction can either be committed or cancelled. Committing a transaction writes all changes to disk. If you cancel a write transaction, all the changes are discarded. Transactions are "all or nothing": either all the writes within a transaction succeed, or none of them take effect. This helps guarantee data consistency, as well as providing thread safety.

Transaction blocks automatically handles begin/commit, and cancel if an error happens. Since transactions are blocked by other transactions, you should perform write operations on a background thread to avoid blocking the UI thread. By using the asynchronous method realm.executeTransactionAsync() the transaction will be performed on a background thread and avoid blocking the UI thread.

Queries
You can query the database using realm.where(*Class*.class).findAll() to get all *Class* objects saved and assign them to a RealmResults object.
Realm also includes many filters such as equalTo(), logical operators such as AND and OR, and sorting. RealmResults (and RealmObject) are live objects that are automatically kept up to date when changes happen to their underlying data.

Realm vs SQLite/Room
Realm

- Un-structured object database
- Simple API
- Uses a simple ORM (object relational mapping) query interface, no SQL knowledge needed
- Supports Android and iOS
- Reactive - data is automatically updated

SQLite/Room

- Structured SQL database
- Requires knowledge of SQL
- Complex, many steps; simplified with Room

List

I'm going to update my List app to use Realm for data persistence.

Realm Setup

1. Add a buildscript block to the top of your project level build.gradle file.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "io.realm:realm-gradle-plugin:10.10.1"
    }
}
```

Install Realm as a Gradle plugin by adding this to the plugins section.

```
plugins {
    ...
    id "org.jetbrains.kotlin.kapt" version "1.6.20" apply false
}
```

2. Then, make the following changes to your application-level Gradle build file to use the `kotlin-kapt` plugin.

```
plugins {
    ...
    id 'org.jetbrains.kotlin.kapt'
}
```

Beneath the `plugins` block, apply the realm-android plugin.

```
apply plugin: "realm-android"
```

Make sure you have, or add, these dependencies:

```
implementation "androidx.core:core-ktx:1.7.0"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.4.1"
```

Sync your gradle files.

Application

To use Realm in your app, you must initialize a Realm instance. Realms are the equivalent of a database. They map to one file on disk and contain different kinds of objects. Initializing a Realm is done once in the app's lifecycle. A good place to do this is in an Application subclass.

In your app's java package create a new Kotlin class called ListApplication.

Name: ListApplication

Kind: Class

You'll want this class to extend the Application class, android.app.Application and then we'll initialize and configure our Realm instance.

```
class ListApplication: android.app.Application() {
    override fun onCreate() {
        super.onCreate()

        //initialize realm
        Realm.init(this)

        //define realm configuration
        val realmConfig  = RealmConfiguration.Builder().build()
        //for debugging or if you change the db structure and don't want to
migrate this will clear out the database
        //Realm.deleteRealm(realmConfig)

        //set default real configuration
        Realm.setDefaultConfiguration(realmConfig )
    }
}
```

First we initialize the Realm and then configure it with a RealmConfiguration object. The RealmConfiguration controls all aspects of how a Realm is created. Since we're using the default configuration our realm will be called default.realm.

Note: If you change your class structure after you've run your app and created the Realm database you'll get an error about needing to migrate your database. If you don't care about the data, you can just delete it before you set the configuration. So you can add this line to ListApplication before setDefaultConfiguration.

```
Realm.deleteRealm(realmConfig)
```

Then run it once and it will delete and then create a new database. But then **remove** deleteRealm or it will keep doing this and you won't know why your data isn't persisting.

In the AndroidManifest file, set this class as the name of the application.

```
<application
    android:name=".ListApplication"
…
```

Model

We already have an Item model class so I just modified it to use as my Realm model.

The Item class needs to be declared as open because Kotlin declares all classes as final by default and Realm needs to be able to extend the class. I removed the data modifier.
The Item class also has to extend the RealmObject class.
I made the id a primary key and made sure my properties were declared as a 'var' since class members MUST be variables.
Realm also requires an empty constructor method so I added a secondary empty constructor as well.

```
open class Item(
    @PrimaryKey var id: String,
    var name: String): RealmObject()
    {
        constructor(): this("", "")
    }
```

Errors

If you have any issues with your gradle files or your Realm definitions, you might see this error. Try to see if you can see a more specific error because it could be anything in your Realm definition (such as using val instead of var in your class).

```
Execution failed for task ':app:kaptDebugKotlin'.
> A failure occurred while executing
org.jetbrains.kotlin.gradle.internal.KaptExecution
   > java.lang.reflect.InvocationTargetException (no error message)
```

Realm and LiveData

Realm is reactive, so normally you would add a change listener for every realm query to update our UI when the query is resolved. But with LiveData we can use the built-in LiveData observer to automatically update the UI. Realm queries return RealmResults so for our Realm objects to work with LiveData we need to create a wrapper for RealmResults to expose them as lifecycle aware LiveData properties. We'll use LiveData lifecycle methods to add and remove the listener. This wrapper will be a class named RealmLiveData.

It's good practice to have all database operations including the realm wrapper in a single package so create a new package called util. Then create a new class in the util package called RealmLiveData.

```
class RealmLiveData<T: RealmModel>(val realmResults: RealmResults<T>):
LiveData<RealmResults<T>>() {

    private val listener = RealmChangeListener<RealmResults<T>>(){results->
        value = results
    }

    //called when the number of active observers change from 0 to 1
    override fun onActive() {
        realmResults.addChangeListener(listener)
    }

    //called when the number of active observers change from 1 to 0.
    override fun onInactive() {
        realmResults.removeChangeListener(listener)
    }
}
```

DAO

To access your app's data in a database it's common to use *data access objects*, or DAOs. Each DAO class contains the methods used for accessing the database. By accessing a database using a DAO class

instead of query builders or direct queries, you can separate different components of your database architecture.

In the util package create a class called ItemDAO which will contains the queries to our database.

In getItems() we query the database to get all Item objects. We use our RealmLiveData wrapper since findAllAsync() returns the type RealmResults and we want to use them in our ViewModel as LiveData. RealmResults (and RealmObject) are live objects that are automatically kept up to date when changes happen to their underlying data.

All of our write operations are wrapped in a transaction block so Realm automatically performs the operation on a background thread to avoid blocking the UI thread.

We add a close() method we can call when the app exists and we're finished with the Realm instance so we can close it to deallocate memory and release any other used resource.

```
open class ItemDAO() {
    private val realmDB:Realm = Realm.getDefaultInstance()

    fun getItems(): RealmLiveData<Item> {
        return RealmLiveData(realmDB.where(Item::class.java).findAllAsync())
    }

    fun addItem(item:Item){
        realmDB.executeTransactionAsync{transactionRealm ->
            transactionRealm.insert(item)
        }
    }

    fun deleteItem(id:String){
        realmDB.executeTransactionAsync{transactionRealm ->
            transactionRealm.where(Item::class.java).equalTo("id",
id).findAll().deleteAllFromRealm()
        }
    }

    fun deleteAll(){
        realmDB.executeTransactionAsync{transactionRealm ->
            transactionRealm.deleteAll()
        }
    }

    fun close(){
        realmDB.close()
    }
}
```

The key difference between `insert()` and `copyToRealm()` is whether a proxy is returned or not.

- insert() saves an unmanaged object into the Realm without creating a managed proxy object as return value.

    - much faster as not returning the object makes it possible to optimize it more
    - inserting *many* items is much more efficient by re-using a single unmanaged object and calling `insert()` on it with the right parameters.
- copyToRealm() saves an unmanaged object into the Realm, with returning a proxy to the created managed object.

We don't need an extra repository class when working with Realm as we did with Room/SQLite.

ViewModel

The ViewModel is still the single datasource for the app so the lifecycle changes are handled and its scope is the lifetime of the app.

I had to make some updates to my ItemViewModel to work with Realm.

1. Made the class open
2. itemList is now of type RealmResults instead of List (or ArrayList). Since RealmResults are live objects they are automatically kept up to date when changes happen to their underlying data. This will ensure that our viewModel is always up to date and in sync with our database.
3. Made itemList a LiveData object so it's still observable in MainActivity.
4. Created an instance of the ItemDAO class so we can call the methods in that class.
5. Updated our methods to call the corresponding methods in the ItemDAO class.
6. Added the onCleared() ViewModel lifecycle method which is called when the app is destroyed and the ViewModel is no longer needed and is destroyed so we can also deallocate the memory used by our Realm instance.

```
open class ItemViewModel: ViewModel() {
    private val itemDAO = ItemDAO()
    val itemList: LiveData<RealmResults<Item>> = itemDAO.getItems()

    fun add(item: Item){
        itemDAO.addItem(item)
    }

    fun delete(item: Item){
        itemDAO.deleteItem(item.id)
    }

    fun deleteAll(){
        itemDAO.deleteAll()
    }

    override fun onCleared() {
        super.onCleared()
        itemDAO.close()
    }
}
```

MainActivity

No changes were needed to my MainActivity class. This is great because it shows we did a good job separating the controller from the model and viewmodel.

RecyclerView Adapter
No changes were needed to my adapter class as the interface with the ViewModel stayed the same. Since we're using LiveData and not RealmResults directly there's no need to use the RealmRecyclerViewAdapter base class.

Delete all
Since I have a method to delete all the items from the database I created an options menu with one item that will clear the list by calling out deleteAll() method.

In strings.xml add a string.
```
<string name="action_settings">Clear List</string>
```

In resources create a new directory called menu.
In menu create a new menu resource file.
Name: options_menu
Source set: main
Directory name: menu

In options_menu.xml add one item.
```
<item
    android:id="@+id/action_deleteAll"
    android:orderInCategory="100"
    android:title="@string/action_settings"
    app:showAsAction="never" />
```

MainActivity
In MainActivity add the onCreateOptionsMenu() method so the menu is inflated/added to the action bar.

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.options_menu, menu)
    return true
}
```

Then add the onOptionsItemSelected() method which is called when an item in the options menu is clicked.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_deleteAll -> {
            viewModel.deleteAll()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Realm Studio
https://github.com/realm/realm-studio

Realm Studio is a developer tool that let's you manage the Realm Database and Realm Platform. You can open and edit local and synced Realms and administer any Realm Object Server instance. It supports Mac, Windows and Linux.

Realm Browser
Realm Browser allows you read and edit Realm databases from your computer (available on the App store for Mac only). It's really useful while developing as the Realm database format is proprietary and not easily human-readable.
Download the Realm Browser from the Mac app store https://itunes.apple.com/app/realm-browser/id1007457278

To help you find where your Realm database is you can get the path using Realm.getPath.
Navigate there and double click on default.realm and it will open in Realm Browser.
The easiest way to go to the database location is to open Finder, press Cmd-Shift-G and paste in the path. Leave off the file:/// and the file name
(Users/aileen/Library/Developer/CoreSimulator/Devices/45F751D5-389F-4EED-AE12-73A28081DEBD/data/Containers/Data/Application/2D9CEC13-8089-4F77-B474-E83578F98179/Documents)