CMU 04-801: Advanced Mobile Application Development
Week 6: Android and Realm

Realm and Android

https://realm.io/docs/java/latest/
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://realm.io/docs/java/latest/#installation
To add Realm to an Android app you need to install Realm as a Gradle plug-in:

1. Add the Realm plug-in to the project gradle file as a dependency.
2. Apply the realm-android plug-in to the top of the application gradle file.

Initialization

https://realm.io/docs/java/latest/#initializing-realm
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(context);
```

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

```
Realm 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

https://realm.io/docs/java/latest/#realms
To control how Realms are created, use a RealmConfiguration object. The minimal configuration usable by Realm is:

```
RealmConfiguration config = new 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://realm.io/docs/java/latest/#models
Realm model classes are very similar to POJO 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.

Adapters

https://realm.io/docs/java/latest/#adapters

Realm makes two adapters available that can be used to bind its data to UI widgets.

- RealmBaseAdapter for working with ListViews
- RealmRecyclerViewAdapter for working with RecyclerViews

To use any one of these adapters, you have to add the io.realm:android-adapters:2.1.1 dependency in the application level Gradle file.

Transactions

https://realm.io/docs/java/latest/#writes

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

https://realm.io/docs/java/latest/#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. The RealmBaseAdapter also automatically keeps track of changes to its data model and updates when a change is detected.

Realm Studio
https://realm.io/products/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)

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
- Need to use LiveData or listeners to update UI

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

Realm Setup
Install Realm as a Gradle plugin.
1. Add the class path dependency to the project level build.gradle file.
```
dependencies {
    classpath "io.realm:realm-gradle-plugin:6.0.2"
}
```

2. Apply the realm-android plugin to the top of the application level build.gradle file.

```
apply plugin: 'realm-android'
```

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 class called ListApplication.
Name: ListApplication
Kind: Class
Superclass: android.app.Application (just start typing Application)

```
public class ListApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();

        //initialize realm
        Realm.init(this);
        //define realm configuration
        RealmConfiguration realmConfig = new
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.

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

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

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 being persistent.

Model

We already have an Item model class so I just modified it to use as my Realm model. I added the @RealmClass annotation and extend the RealmModel base class.
Realm doesn't support auto generating values for the primary key so I changed id to a string and then assign random UUIDs to it. I also added a getter method for it.
Realm also requires an empty constructor method so I added that as well.

```
@RealmClass
public class Item implements RealmModel {

    @PrimaryKey public String id = UUID.randomUUID().toString();
    private String name;

    public Item(){}

    public Item(String name) {
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

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. 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 class in the data package called RealmLiveData.

```
public class RealmLiveData<T extends RealmModel> extends
LiveData<RealmResults<T>> {
    private RealmResults<T> results;
    private final RealmChangeListener<RealmResults<T>> listener =
            new RealmChangeListener<RealmResults<T>>() {
                @Override
```

```
                public void onChange(RealmResults<T> results) {
                    setValue(results);
                }
            };

    public RealmLiveData(RealmResults<T> realmResults) {
        results = realmResults;
    }

    @Override
    protected void onActive() {
        results.addChangeListener(listener);
    }

    @Override
    protected void onInactive() {
        results.removeChangeListener(listener);
    }
}
```

DAO

Since we already have a DAO class that contains the queries which are made via our ViewModel we'll just update it to hold our realm queries.

In the constructor for the ItemDAO class we get the default realm instance which will be used in all interactions with the database.

In getAllItems() 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.

```
public class ItemDAO {
    private Realm realmDb;

    public ItemDAO(){
        this.realmDb = Realm.getDefaultInstance();
    }

    public RealmLiveData<Item> getAllItems(){
        return new RealmLiveData(realmDb.where(Item.class).findAllAsync());
    }

    public void insertItem(final String name){
        final Item newItem = new Item(name);
```

```
        //start realm write transaction
        realmDb.executeTransactionAsync(new Realm.Transaction() {
            @Override
            public void execute(Realm realm) {
                //create realm object
                realm.copyToRealm(newItem);
            }
        });
    }

    public void deleteItem(final String id){
        realmDb.executeTransactionAsync(new Realm.Transaction() {
            @Override
            public void execute(Realm realm) {
                //finds and deletes the realm object
                realm.where(Item.class).equalTo("id",
id).findAll().deleteAllFromRealm();
            }
        });
    }

    public void deleteAll(){
        //start realm write transaction
        realmDb.executeTransactionAsync(new Realm.Transaction() {
            @Override
            public void execute(Realm realm) {
                realm.deleteAll();
            }
        });
    }

    public void close(){
        realmDb.close();
    }
}
```

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

ViewModel

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

It only has a few changes from our Room version. itemList is now of type RealmResults instead of List. 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.

It no longer needs to be passed the application context because we subclassed Application to create our Realm database. So we can switch back from AndroidViewModel to ViewModel which also changes the constructor. We also don't need a data member for context or the app repository, but we do add a data member for an instance of the ItemDAO class. We also update our methods to call the corresponding methods in the ItemDAO class. And lastly we add the onCleared() 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.

```
public class ItemViewModel extends ViewModel {

    private LiveData<RealmResults<Item>> itemList;
    private ItemDAO itemDAO;

    public ItemViewModel() {
        itemDAO = new ItemDAO();
        itemList = itemDAO.getAllItems();
    }

    public LiveData<RealmResults<Item>> getItemList() {
        return itemList;
    }

    public void insertItem(String name){
        itemDAO.insertItem(name);
    }

    public void deleteItem(Item item){
        itemDAO.deleteItem(item.getId());
    }

    public void deleteAll(){
        itemDAO.deleteAll();
    }

    @Override
    protected void onCleared() {
        super.onCleared();
        itemDAO.close();
    }
}
```

MainActivity

In MainActivity we had two changes.
Because ItemViewModel extends ViewModel now (and not AndroidViewModel) we need to change how it's instantiated.

```
itemViewModel = new ViewModelProvider(this).get(ItemViewModel.class);
```

Most importantly because our view model's itemList is of type RealmResults, it is automatically kept up to date. Which means in MainActivity the observer will fire whenever there's a change in the list of items. So when a new item is added we can simply add it to our realm database and when the observer fires, the viewmodel, main activity, and recyclerview, will all be updated.

```
protected void onCreate(Bundle savedInstanceState) {
...
fab.setOnClickListener(new View.OnClickListener() {
...
```

```
public void onClick(final View view)
...
dialog.setPositiveButton("Add", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int whichButton) {
        //get item entered
        String newName = edittext.getText().toString();
        if (!newName.isEmpty()) {
            //call insert method
            itemViewModel.insertItem(newName);
        }
        Snackbar.make(view, "Item added", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
});
```

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.