ATLS 4320/5320: Advanced Mobile Application Development
Week 13: Android and Realm

Realm and Android

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

As Realm is cross-platform, it has all the same features on Android as it does on iOS. It 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.

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.

Model

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

You create a Realm model class by extending 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.

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 the onCreate() method in a subclass of your Application class.

```
Realm.init(context);
```

If you create your own application subclass, you must add it to the app's AndroidManifest.xml.

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 realmConfig = 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.

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:2.0.0 dependency in the application level Gradle file.

Queries
https://realm.io/docs/java/latest/#queries
You can query the database with 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.

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.
Write operations can be made using the following format:

```
realm.beginTransaction();
//... add or update objects here ...
realm.commitTransaction();
```

Or they can be made using transaction blocks that use the realm.executeTransaction() or realm.executeTransactionAsync() methods which we will use in our app. By default, write transactions block each other. It is recommended that you use asynchronous transactions on the UI thread that will run on a background thread and avoid blocking the UI. This is why we use executeTransactionAsync() as opposed to the other function. You can pass a callback function to executeTransactionAsync() that will get called when the transaction completes.

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)

You can view your data but also edit, delete, and add. This is a great tool for debugging.

To view the database on Windows, you can use Stetho Realm and there are also several Android libraries for easy viewing of the data on the device.

BookList
Create a new project called BookList.
Company name: a qualifier that will be appended to the package name
Package name: the fully qualified name for the project
Check Phone and Tablet, leave the rest unchecked
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Basic Activity template
Activity name: BookListActivity
Check Generate Layout File
Layout Name: activity_book_list
Title: Book List (for launcher activities this is the app title)
You can leave the hierarchical parent empty as we won't be implementing up navigation.

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:5.0.0"
}
```

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

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

3. Add the following dependency in the same application level build.gradle file. This library is required if your project is going to use Realm adapters. We're going to use the RealmBaseAdapter for our ListView.

```
dependencies {
   //for realm adapters
   compile 'io.realm:android-adapters:2.1.1'
}
```

Sync your gradle files.

While that's compiling, look at what the basic template created for us.

activity_book_list.xml sets the AppBarLayout which includes a Toolbar.
Includes the content_book_list xml layout which right now is just a textview.
Includes a floating action button.

BookListActivity.java sets up the floating action button as well as an onClickListener for it.
A menu has also been added and set up.

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.

Create a class named BookListApplication in your main Java package.
In the java folder select the booklist folder (not androidTest or test) and create a new Java class.

```
import io.realm.Realm;
import io.realm.RealmConfiguration;

@Override public void onCreate() {
   super.onCreate();
   //initialize Realm
   Realm.init(this);
   //define the configuration for realm
   RealmConfiguration realmConfig = new RealmConfiguration.Builder().build();
   //set the default realm 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=".BookListApplication"
…
```

Create a model class called Book.
In the java folder select the booklist folder (not androidTest or test)
File | New | Java class
Name: Book
Kind: Class

```
import io.realm.RealmObject;
import io.realm.annotations.PrimaryKey;
import io.realm.annotations.Required;
```

```
public class Book extends RealmObject{
    @Required
    @PrimaryKey
    private String id;
    private String book_name;
    private String author_name;
    private boolean read;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getBook_name() {
        return book_name;
    }

    public void setBook_name(String book) {
        this.book_name = book;
    }

    public String getAuthor_name() {
        return author_name;
    }

    public void setAuthor_name(String author) {
        this.author_name = author;
    }

    public boolean hasRead() {
        return read;
    }

    public void setRead(boolean done) {
        this.read = done;
    }
}
```

The id field is annotated with @PrimaryKey making it the primary key which means each instance must have a unique id.

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 BookListApplication 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.

Layout
Next open the content_task_list.xml layout file and replace the TextView with a ListView with the id book_list.

In the activity_book_list.xml file, change the icon on the FloatingActionButton from email to add
app:srcCompat="@android:drawable/ic_input_add"

Add a layout file named book_list_row.xml to the res/layout folder that will specify the format of each row of the ListView. We want to have TextViews for book name with the id bookTextView, author name with the id authorTextView, and a check box with the id checkBox to mark when a book's been read.

When layouts contain either focusable or clickable widgets, OnItemCLickListener won't be called if these widgets are focused. So for the checkbox you **MUST** add the following attribute so we'll be able to click on rows of the list to edit or delete them.
android:focusable="false"

Adapter
We'll use the RealmBaseAdapter to display the books in our listview.
To create an adapter, we need to create a new class and extend RealmBaseAdapter.

In the java folder select the booklist folder (not androidTest or test)
File | New | Java class
Name: BookAdapter
Kind: Class

```
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.CheckBox;
import android.widget.ListAdapter;
import android.widget.TextView;
import io.realm.OrderedRealmCollection;
import io.realm.RealmBaseAdapter;

public class BookAdapter extends RealmBaseAdapter<Book> implements ListAdapter{

   private BookListActivity activity;

   private static class ViewHolder {
      TextView bookName;
      TextView authorName;
      CheckBox hasRead;
   }
```

```
    BookAdapter(BookListActivity activity, OrderedRealmCollection<Book> data){
        super(data);
        this.activity = activity;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ViewHolder viewHolder;

        if (convertView == null) {
            convertView = LayoutInflater.from(parent.getContext())
                    .inflate(R.layout.book_list_row, parent, false);
            viewHolder = new ViewHolder();
            viewHolder.bookName = (TextView) convertView.findViewById(R.id.bookTextView);
            viewHolder.authorName = (TextView) convertView.findViewById(R.id.authorTextView);
            viewHolder.hasRead = (CheckBox) convertView.findViewById(R.id.checkBox);
            viewHolder.hasRead.setOnClickListener(listener);
            convertView.setTag(viewHolder);
        } else {
            viewHolder = (ViewHolder) convertView.getTag();
        }
        if (adapterData != null) {
            Book book = adapterData.get(position);
            viewHolder.bookName.setText(book.getBook_name());
            viewHolder.authorName.setText(book.getAuthor_name());
            viewHolder.hasRead.setChecked(book.hasRead());
            viewHolder.hasRead.setTag(position);
        }
        return convertView;
    }

    private View.OnClickListener listener = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            int position = (Integer) view.getTag();
            if (adapterData != null) {
                Book book = adapterData.get(position);
                activity.changeBookRead(book.getId());
            }
        }
    };
}
```

Creating and using the class ViewHolder optimizes performance by reusing existing views.
Every time the adapter goes to display a book it will check to see if the view exists. If it doesn't, it creates one, and uses setTag() so that view can be reused later.
Then we set the data for the view and return the view.
We also implement OnClickListener for later when we'll want to edit or delete the items.

changeBookRead() will give you an error until we implement it in BookListActivity so comment it out for now.

In BookListActivity create a variable for your realm database and then modify onCreate().

```
import android.content.DialogInterface;
import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.ListView;
import java.util.UUID;
import io.realm.Realm;
import io.realm.RealmResults;

private Realm realm;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_book_list);

    //get realm instance
    realm = Realm.getDefaultInstance();

    //get all saved Book objects
    RealmResults<Book> books = realm.where(Book.class).findAll();

    final BookAdapter adapter = new BookAdapter(this, books);

    ListView listView = (ListView) findViewById(R.id.book_list);

    //set our RealmBaseAdapter to the listview
    listView.setAdapter(adapter);

    listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
            //create a vertical linear layout to hold edit texts
            LinearLayout layout = new LinearLayout(BookListActivity.this);
            layout.setOrientation(LinearLayout.VERTICAL);
```

```
        final Book book = (Book) adapterView.getAdapter().getItem(i);

        //create edit texts and add to layout
        final EditText bookEditText = new EditText(BookListActivity.this);
        bookEditText.setText(book.getBook_name());
        layout.addView(bookEditText);
        final EditText authorEditText = new EditText(BookListActivity.this);
        authorEditText.setText(book.getAuthor_name());
        layout.addView(authorEditText);

        AlertDialog.Builder dialog = new AlertDialog.Builder(BookListActivity.this);
        dialog.setTitle("Edit Book");
        dialog.setView(layout);
        dialog.setPositiveButton("Save", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialogInterface, int i) {
                    //save edited book
                }
            });
        dialog.setNegativeButton("Delete", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialogInterface, int i) {
                    //delete book
                }
            });
        dialog.create();
        dialog.show();
    }
});

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //create a vertical linear layout to hold edit texts
        LinearLayout layout = new LinearLayout(BookListActivity.this);
        layout.setOrientation(LinearLayout.VERTICAL);

        //create edit texts and add to layout
        final EditText bookEditText = new EditText(BookListActivity.this);
        bookEditText.setHint("Book name");
        layout.addView(bookEditText);
        final EditText authorEditText = new EditText(BookListActivity.this);
        authorEditText.setHint("Author name");
```

```
        layout.addView(authorEditText);

        //create alert dialog
        AlertDialog.Builder dialog = new AlertDialog.Builder(BookListActivity.this);
        dialog.setTitle("Add Book");
        dialog.setView(layout);
        dialog.setPositiveButton("Save", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                //get book name entered
                final String newBookName = bookEditText.getText().toString();
                final String newAuthorName = authorEditText.getText().toString();

                if(!newBookName.isEmpty()) {
                    //start realm write transaction
                    realm.executeTransactionAsync(new Realm.Transaction() {
                        @Override
                        public void execute(Realm realm) {
                            //create a realm object
                            Book newbook = realm.createObject(Book.class, UUID.randomUUID().toString());
                            newbook.setBook_name(newBookName);
                            newbook.setAuthor_name(newAuthorName);
                        }
                    });
                }
            }
        });
        dialog.setNegativeButton("Cancel", null);
        dialog.show();
    }
  });
}
```

We first get a Realm instance which will be used in all interactions with the database.
After getting the instance, we query the database to get all Book objects saved and assign them to a RealmResults object.
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.
We then create an instance of a ListView, set its adapter and add an OnItemClickListener on it. When a list item is tapped, we display an AlertDialog that the user can use to either edit the task or delete it. We'll implement these later.
We then create an instance of a FloatingActionButton and set an OnClickListener on it which displays an AlertDialog that can be used to create a task.

Because we want our AlertDialog to have two EditTexts, if we create them and then use setView as we've done before, the second overwrites the first. So instead we create a LinearLayout with a vertical orientation and add the EditTexts to that. Then we set the AlertDialog view to that layout object.

All write operations to Realm (create, update and delete) must be wrapped in write transactions. The realm.executeTransactionAsync() method is used on the UI thread to avoid blocking the UI.

We create a Book object with realm.createObject(), set a random UUID value as its id. Then we set the book and author names.

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. For a UI thread the easiest way to close a Realm instance is in the onDestroy() method. Realm instances are reference counted, which means each call to getInstance() must have a corresponding call to close().

```
@Override
protected void onDestroy() {
   super.onDestroy();
   //close the Realm instance when the Activity exits
   realm.close();
}
```

Now we add the method that's called from the OnClickListener we created in the adapter. It gets an id of the Book whose checkbox was checked and updates its read field.

```
public void changeBookRead(final String bookId) {
   realm.executeTransactionAsync(new Realm.Transaction() {
      @Override
      public void execute(Realm realm) {
         Book book = realm.where(Book.class).equalTo("id", bookId).findFirst();
         book.setRead(!book.hasRead());
      }
   });
}
```

Run the app and you should be able to add books, mark them as read, exit the application and your data is persistent.

Edit
Add a method to BookListActivity to edit a book's name and/or author.
Again we wrap it in executeTransactionAsync() and we retrieve the book using its id.

```
private void changeBook(final String bookId, final String book_name, final String author_name) {
   realm.executeTransactionAsync(new Realm.Transaction() {
      @Override
      public void execute(Realm realm) {
         Book book = realm.where(Book.class).equalTo("id", bookId).findFirst();
         book.setBook_name(book_name);
         book.setAuthor_name(author_name);
      }
   });
}
```

Then we call this method with the updated book and author names from the AlertDialog positive button in onCreate() where we had *//save edited book*

```
//get updated book and author names
final String updatedBookName = bookEditText.getText().toString();
final String updatedAuthorName = authorEditText.getText().toString();
if(!updatedBookName.isEmpty()) {
   changeBook(book.getId(), updatedBookName, updatedAuthorName);
}
```

Delete

Add a method to BookListActivity to delete a book.
Again we wrap it in executeTransactionAsync() and we retrieve the book using its id. We can use findFirst() because ids are unique.

```
private void deleteBook(final String bookId) {
   realm.executeTransactionAsync(new Realm.Transaction() {
      @Override
      public void execute(Realm realm) {
         realm.where(Book.class).equalTo("id", bookId)
               .findFirst()
               .deleteFromRealm();
      }
   });
}
```

Then we call this method from the AlertDialog negative button in onCreate() where we had *//delete book*

```
deleteBook(book.getId());
```

You can also easily add a menu item to the menu to delete all read books. You would query realm where hasRead is equal to true and call deleteAllFromRealm().