ANDROID PROGRAMMING

Course 14 -SQLite Databases (With ActiveAndroid Library)

- Almost every needs a place to save data for the long term, longer than savedInstanceState will keep it around. Android provides a place to do this for you : a local filesystem on your phone or tablet's flash memory storage.
- Each application on an Android device has a directory in its sandbox. Keeping files in the sandbox protects them from being accessed by other applications or even by prying eyes users (unless the device has been "rooted", in which case the user can get to whatever her or she likes)
- Each application's sandbox directory is a child of device's data/data directory named after the application package. For CriminalIntent, the full path to the sandbox directory is data/data/com.bignerdranch.android.com.criminalIntent
- However, most application data is not stored in plain old files. Here is why : say that you had a file with all of your Crimes written out. To change the title on a Crime at the beginning ıg the file, you would have to read in the entire file and write out a whole new version. With a lot of Crimes, that would take a long time.

- There is where SQLite comes in. SQLite is an open source relational database, like MySQL or Postgresql. Unlike other databases, through, SQLite stores its data in simple files, which you can read and write using SQLite library. Android includes this SQLite library in its standard library, along with other some additional Java Helper classes.

USING ACTIVEANDROID

- ActiveAndroid is an active record style ORM (object relational mapper). What does that mean exactly? Well, ActiveAndroid allows you to save and retrieve SQLite database records without ever writing a single SQL statement. Each database record is wrapped neatly into a class with methods like save() and delete().
- ActiveAndroid does so much more than this though. Accessing the database is a hassle, to say the least, in Android. ActiveAndroid takes care of all the setup and messy stuff, and all with just a few simple steps of configuration.

INSTALLING WITH GRADLE

- Modify your build.gradle to include:

```
repositories {
    mavenCentral()
    maven { url "https://oss.sonatype.org/content/repositories/snapshots/" }
}

compile 'com.michaelpardo:activeandroid:3.1.0-SNAPSHOT'
```

CONFIGURING YOUR PROJECT

- Now that you have ActiveAndroid added to you project, you can begin your two-step configuration process! The first thing we'll need to do is add some global settings. ActiveAndroid will look for these in the AndroidManifest.xml file. Open the AndroidManifest.xml file located at the root directory of your project. Let's add some configuration options.

- AA_DB_NAME (optional)
- AA_DB_VERSION (optional – defaults to 1)

- The configuration strings for my project look like this

```
<manifest ...>
    <application android:name="com.activeandroid.app.Application" ...>

        ...

        <meta-data android:name="AA_DB_NAME" android:value="Pickrand.db" />
        <meta-data android:name="AA_DB_VERSION" android:value="5" />

    </application>
</manifest>
```

Notice also that the application name points to the ActiveAndroid application class. This step is required for ActiveAndroid to work.

If you are using a custom Application class, just extend com.activeandroid.app.Application instead of android.app.Application

```
public class MyApplication extends com.activeandroid.app.Application { ...
```

- But what if you're already doing this to utilize another library? Simply initialize ActiveAndroid in the Application class.
- You may call ActiveAndroid.dispose(); if you want to reset the framework for debugging purposes. (Don’t forget to call initialize again.)

```
public class MyApplication extends SomeLibraryApplication {
    @Override
    public void onCreate() {
        super.onCreate();
        ActiveAndroid.initialize(this);
    }
}
```

- If you want to build a database dynamically.You can use the configuration class.

```
public class MyApplication extends SomeLibraryApplication {
    @Override
    public void onCreate() {
        super.onCreate();
        Configuration dbConfiguration = new Configuration.Builder(this).setDatabaseName("xx
        ActiveAndroid.initialize(dbConfiguration);
    }
}
```

CREATING DATABASE MODEL

```
@Table(name="Crimes")
public class Crime extends Model {

    @Column(name="mId")
    private UUID mId;

    @Column(name="mTitle")
    private String mTitle;

    @Column(name="mDate")
    private Date mDate;

    @Column(name="mSolved")
    private boolean mSolved;
```

INSERTING A RECORD

```
public void addCrime (Crime c)
{
      c.save();
}
```

GETTING A LIST OF CRIMES FROM DATABASE

```
public List<Crime> getCrimes() {
        return new Select()
                .from(Crime.class)
                .execute();
}
```

GETTING A SINGLE CRIME BY ID

```
public Crime getCrime(UUID id) {

    return new Select()
            .from(Crime.class)
            .where("mId = ?", id)
            .executeSingle();
}
```

CRIMEFRAGMENT UPDATE CRIME RECORD

```
@Override
public void onPause() {
    super.onPause();
    CrimeLab.get(getActivity()).updateCrime(mCrime);
}
```

BULK INSERT

- To insert multiple records at the same time you can use transactions. Calls wrapped in transactions can be speed up by a factor of around 100. Here's an example:

```
ActiveAndroid.beginTransaction();
try {
        for (int i = 0; i < 100; i++) {
            Item item = new Item();
            item.name = "Example " + i;
            item.save();
        }
        ActiveAndroid.setTransactionSuccessful();
}
finally {
        ActiveAndroid.endTransaction();
}
```

DELETING AN ITEM

What about deleting a record? Well, to delete a single record just call the delete() method. In the following example we'll load an Item object by Id and delete it.

```
Item item = Item.load(Item.class, 1);
item.delete();
```

Or you can delete it statically

```
Item.delete(Item.class, 1);
```

You can also use the query builder syntax

```
new Delete().from(Item.class).where("Id = ?", 1).execute();
```