Mobile Application Development

Data Persistence

DATA PERSISTENCE WITH SHARED PREFERENCES

Shared Preferences

- If you have a relatively small collection of key-values that you'd like to save, you should use the **SharedPreferences APIs**.
- A SharedPreferences object points to a file containing **key-value pairs** and provides simple methods to read and write them.
- Each SharedPreferences file is managed by the framework and can be **private or shared**.

Shared Preferences

- The SharedPreferences APIs are only for reading and writing key-value pairs and you should not confuse them with the **Preference APIs**, which help you **build a user interface for your app settings** (although they use SharedPreferences as their implementation to save the app settings).

Shared Preferences

- You can create a new shared preference file or access an existing one by calling one of two methods:
  - **getSharedPreferences()** — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.
  - **getPreferences()** — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, **you don't need to supply a name**.

Shared Preferences

Saving Shared Preferences:

```
fn="Some Name";
SharedPreferences sharedPreferences =
getSharedPreferences("pk.edu.riu.e4031.share_preference_file_ke
y", MODE_PRIVATE);
SharedPreferences.Editor editor=sharedPreferences.edit();
editor.putString("full_name",fn);
editor.commit();
```

Shared Preferences

Retrieve Shared Preferences:

```
SharedPreferences sharedPreferences =
getSharedPreferences("pk.edu.riu.e4031.share_preference_file_ke
y",MODE_PRIVATE);

String helloText =
sharedPreferences.getString("full_name",“Default Value");
```

DATA PERSISTENCE WITH SQLITE DATABASE

What is SQLite?

- SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.
  - It requires very minimal support from external libraries or from the operating system
  - With SQLite, the process that wants to access the database reads and writes directly from the database files on disk.
  - SQLite does not need to be "installed" before it is used.
  - With SQLite if anything goes wrong during a transaction, it is rolled back automatically.
- More information about SQLite: http://www.sqlite.org/

What is SQLite?

- SQLite supports the data types TEXT (similar to String in Java), INTEGER (similar to long in Java) and REAL (similar to double in Java).
- All other types must be converted into one of these fields before getting saved in the database.

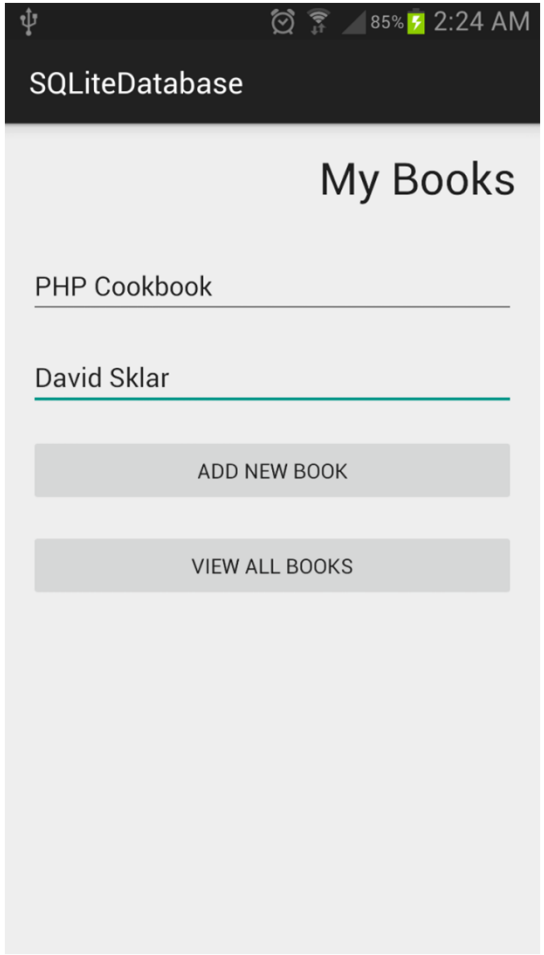
SQLite in Android

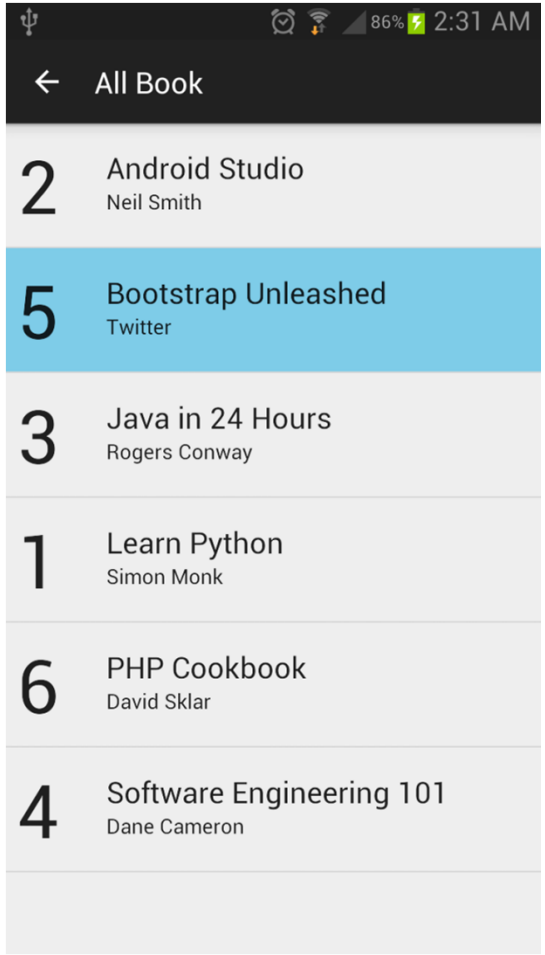
- Android provides full support for SQLite databases.
- Any databases you create will be accessible by name to any class in the application, but not outside the application.
- The APIs you'll need to use a database on Android are available in the android.database.sqlite package.
- Database file is stored in /data/data/<package_name>/databases folder in the device.

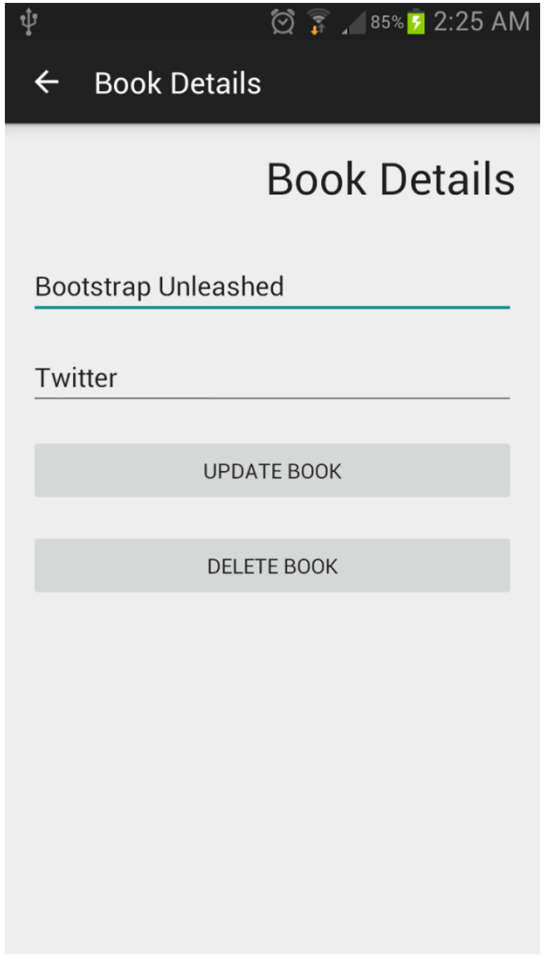
SQL

- SQL (Structured Query Language) is a language designed to manage relational databases.
- SELECT, INSERT, UPDATE, DELETE, etc
- http://www.w3schools.com/sql/

Sample Application







How to Use SQLite Database

- **Define a Schema**
  - Scheme is a formal declaration of how the database is organized.
- **Get an instance of your SQLite database using SQLiteOpenHelper class.**
  - SQLiteOpenHelper class is used for database creation and version management.
- **Perform CREATE, UPDATE, DELETE, SELECT, etc. operations using SQLiteDatabase object.**
  - You can create your own database adapter to handle these operations.

Define Schema

Define Schema

- The schema is reflected in the SQL statements that you use to create your database.
  - CREATE TABLE mytable (id INTEGER PRIMARY KEY AUTOINCREMENT, fullname TEXT NOT NULL, email TEXT)
  - DROP TABLE IF EXISTS mytable
- You may find it helpful to create a companion class, known as a **contract class**, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

Contract Class

- A contract class is a container for constants that define names for tables and columns.
- The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.
- A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

Contract Class

```
public final class DatabaseContract {

    public DatabaseContract() { }

    public static abstract class Books implements BaseColumns {

        public static final String TABLE_NAME="books";
        public static final String COL_TITLE="title";
        public static final String COL_AUTHOR="author";

    }

}
```

Define Schema in Contract Class

- By implementing the BaseColumns interface, your inner class can inherit a primary key field called _ID that some Android classes such as cursor adaptors will expect it to have.
- It's not required, but this can help your database work harmoniously with the Android framework.

SQLiteOpenHelper

SQLiteOpenHelper

- Once you have defined how your database looks, you should implement methods that create and maintain the database and tables.
- A useful set of APIs is available in the SQLiteOpenHelper class.
- To use SQLiteOpenHelper, create its subclass and override the onCreate() , and onUpgrade() callback methods.

SQLiteOpenHelper

```
public class DatabaseHelper extends SQLiteOpenHelper {
   public static final int DATABASE_VERSION = 1;
   public static final String DATABASE_NAME = "library.db";
   public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }
   @Override
   public void onCreate(SQLiteDatabase db) {
        // TODO Auto-generated method stub
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
   newVersion) {
        // TODO Auto-generated method stub
   }
}
```

Provide custom cursor or enter **null** to use default cursor

onCreate()

- Called only once when database is created for the first time.

onCreate()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   public static final int DATABASE_VERSION = 1;
   public static final String DATABASE_NAME = "library.db";

   private static final String CREATE_TABLE_BOOKS="CREATE TABLE "
            + Books.TABLE_NAME + " ("
            + Books._ID +" INTEGER PRIMARY KEY AUTOINCREMENT, "
            + Books.COL_TITLE + " TEXT, "
            + Books.COL_AUTHOR + " TEXT)";

   public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

   . . .
}
```

onCreate()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   . . .

   @Override
   public void onCreate(SQLiteDatabase db) {
        // TODO Auto-generated method stub
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
   newVersion) {
        // TODO Auto-generated method stub
   }
}
```

onCreate()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   . . .

   @Override
   public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_BOOKS);
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
   newVersion) {
        // TODO Auto-generated method stub
   }
}
```

onUpgrade()

- Called when database needs to be upgraded.
- You indicate to Android system that database has been upgraded by changing the Version Number of the database.

  For Example:

  - `public static final int DATABASE_VERSION = 2;`
  - `public static final int DATABASE_VERSION = 3;`

onUpgrade()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   public static final int DATABASE_VERSION = 2;
   public static final String DATABASE_NAME = "EmailUser.db";
   private static final String CREATE_TABLE_BOOKS="CREATE TABLE "
            + Books.TABLE_NAME + " ("
            + Books._ID +" INTEGER PRIMARY KEY AUTOINCREMENT, "
            + Books.COL_TITLE + " TEXT, "
            + Books.COL_AUTHOR + " TEXT, “
            + Books.COL_NOTES + " TEXT)";

   private static final String ALTER_TABLE_BOOKS = "ALTER TABLE "
   + Users.TABLE_NAME + " ADD COLUMN "
   + Users.COL_NOTES + " TEXT;

   . . .
}
```

onUpgrade()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   . . .

   @Override
   public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_BOOKS);
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
   newVersion) {
         // TODO Auto-generated method stub



   }
}
```

onUpgrade()

```
public class DatabaseHelper extends SQLiteOpenHelper {
   . . .

   @Override
   public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_BOOKS);
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
   newVersion) {
        if (oldVersion<2) {
               db.execSQL(ALTER_TABLE_BOOKS);
        }
   }
}
```

Use SQLiteDatabase

Using SQLiteDatabase

- To access your database, create an instance of extended SQLiteOpenHelper class:

```
DatabaseHelper dbHelper = new DatabaseHelper(getContext());
```

- Now to get the data repository all you need to do is call getWritableDatabase() or getReadableDatabase() methods of SQLiteDatabase object.

```
SQLiteDatabase db = dbHelper.getWritableDatabase();

SQLiteDatabase db = dbHelper.getReadableDatabase();
```

SQLiteDatabase insert() Method

- **Method for inserting a row into the database.**

```
public long insert (String table, String nullColumnHack,
ContentValues values)
```

- **Parameters**
  - table the table to insert the row into
  - nullColumnHack optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your values is empty.
  - values this map contains the initial column values for the row. The keys should be the column names and the values the column values
- **Returns** the row ID of the newly inserted row, or -1 if an error occurred

SQLiteDatabase query() Method

- **Method to query the given table, returning a Cursor over the result set.**

```
public Cursor query (String table, String[] columns, String
whereClause, String[] whereArgs, String groupBy, String having,
String orderBy)
```

- **Returns** A Cursor object, which is positioned before the first entry

SQLiteDatabase delete() Method

- **Method for deleting rows in the database.**

```
public int delete (String table, String whereClause, String[]
whereArgs)
```

- **Parameters**
  - table the table to delete from
  - whereClause the optional WHERE clause to apply when deleting. Passing null will delete all rows.
  - whereArgs You may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.
- **Returns** the number of rows affected if a whereClause is passed in, 0 otherwise.

SQLiteDatabase update() Method

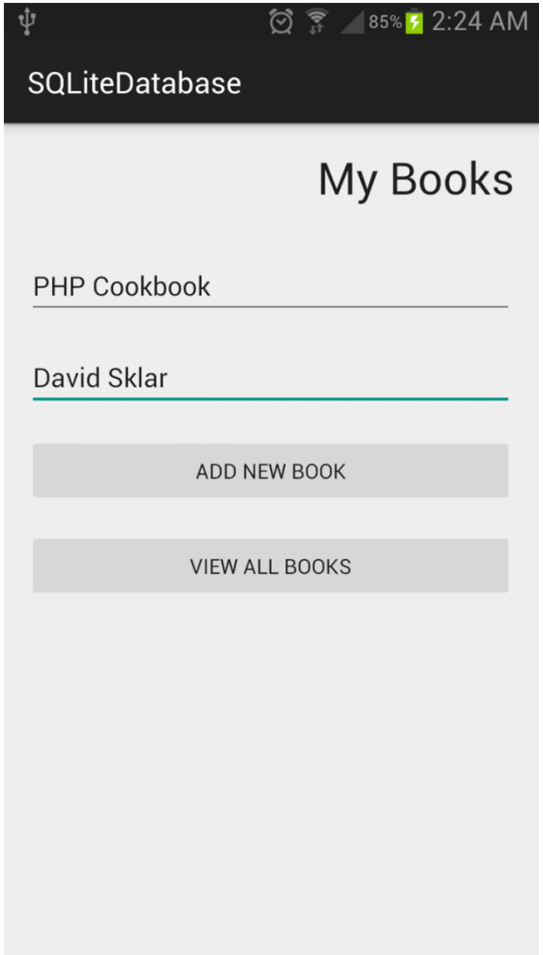
- **Method for for updating rows in the database.**

```
public int update (String table, ContentValues values, String
whereClause, String[] whereArgs)
```

- **Parameters**
  - table the table to update in
  - values a map from column names to new column values. null is a valid value that will be translated to NULL.
  - whereClause
  - whereClause the optional WHERE clause to apply when updating. Passing null will update all rows.
  - whereClause You may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings..
- **Returns** the number of rows affected

ADDING NEW RECORD

Adding New Record



SQLiteDatabase insert() Method

- **Method for inserting a row into the database.**

```
public long insert (String table, String nullColumnHack,
ContentValues values)
```

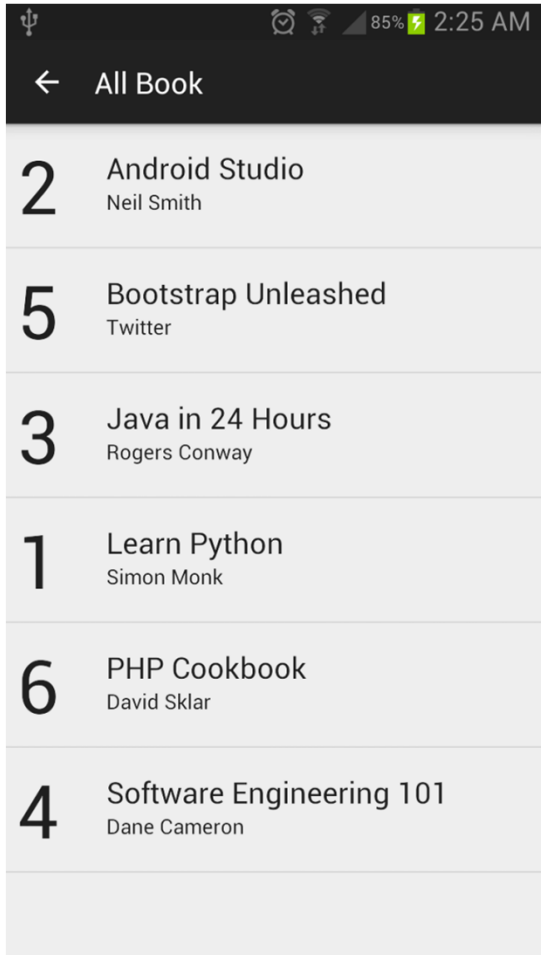
- **Parameters**
  - table the table to insert the row into
  - nullColumnHack optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your values is empty.
  - values this map contains the initial column values for the row. The keys should be the column names and the values the column values
- **Returns** the row ID of the newly inserted row, or -1 if an error occurred

Adding New Record

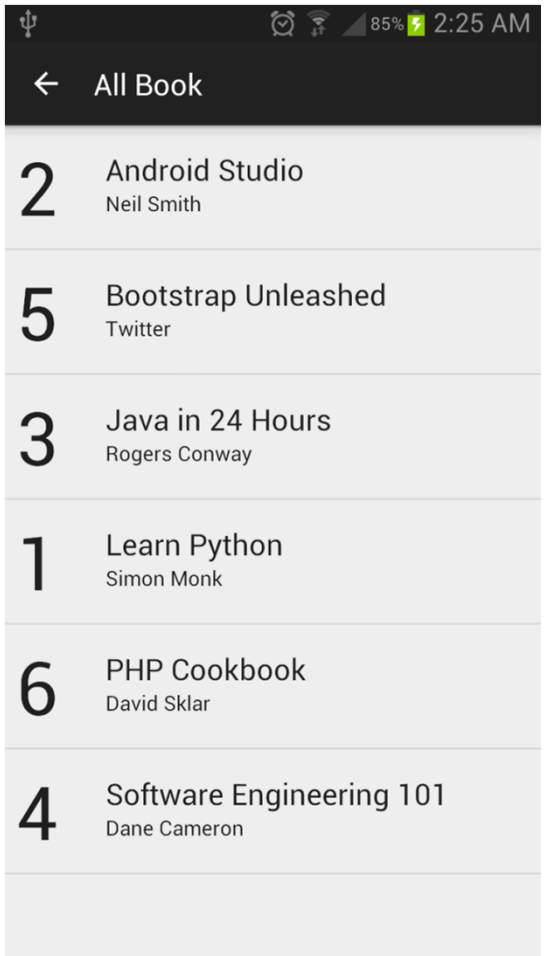
```
String val1="Some Book Title";
String val2="Some Author";

SQLiteDatabase db = dbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(Books.COL_TITLE, val1);
values.put(Books.COL_AUTHOR, val2);

// insert(String table, String nullColumnHack, ContentValues values)
long id=db.insert(Books.TABLE_NAME, null, values);

if (id>0) {
   Toast.makeText(this, "New Record Inserted: " + id,
   Toast.LENGTH_SHORT).show();
}
db.close(); // Closing database connection
```

QUERY ALL RECORDS

Query All Records



Query All Records



- Add ListView
- Get Cursor from SQLiteDatabase query() Method
- Create XML Layout for List Items
- Create Custom CursorAdapter by extending CursorAdapter
- Set CursorAdapter on ListView
- SetOnItemClickListener

Add ListView

```
<RelativeLayout ...>

    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/listView"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

</RelativeLayout>
```

SQLiteDatabase query() Method

- **Method to query the given table, returning a Cursor over the result set.**

```
public Cursor query (String table, String[] columns, String
whereClause, String[] whereArgs, String groupBy, String having,
String orderBy)
```

- **Returns** A Cursor object, which is positioned before the first entry

Get Cursor from query() Method

```
SQLiteDatabase db = dbHelper.getWritableDatabase();

String[] columns={Books._ID,Books.COL_TITLE,Books.COL_AUTHOR};
// String whereClause="";
// String[] whereArgs={};
// String having="";
// String groupBy="";
String orderBy= Books.COL_TITLE + " ASC";

Cursor cursor=db.query(Books.TABLE_NAME, columns, null, null,
   null, null, orderBy);
```

Create XML Layout for List Items

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . .>
    <TextView
        android:layout_width="60dp"
        android:layout_height="wrap_content"
        android:textSize="50sp"
        android:text="id"
        android:id="@+id/txt_id" />

    <LinearLayout . . .>
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:paddingTop="8dp"
            android:textSize="20sp"
            android:text="Book Title"
            android:id="@+id/txt_title"/>
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Book Author"
            android:id="@+id/txt_author"/>
    </LinearLayout>
</LinearLayout>
```

Create Custom CursorAdapter

```
public class BooksCursorAdapter extends CursorAdapter {
    public BooksCursorAdapter(Context context, Cursor cursor) {
        super(context,cursor,0);
    }
    @Override
    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        // TODO
    }

    @Override
    public void bindView(View view, Context context, Cursor cursor) {
        // TODO
    }
}
```

Create Custom CursorAdapter

```
public class BooksCursorAdapter extends CursorAdapter {
 . . .

@Override
public View newView(Context context, Cursor cursor, ViewGroup parent)
{
   return
   LayoutInflater.from(context).inflate(R.layout.all_books,parent,false);

}
  . . .
}
```

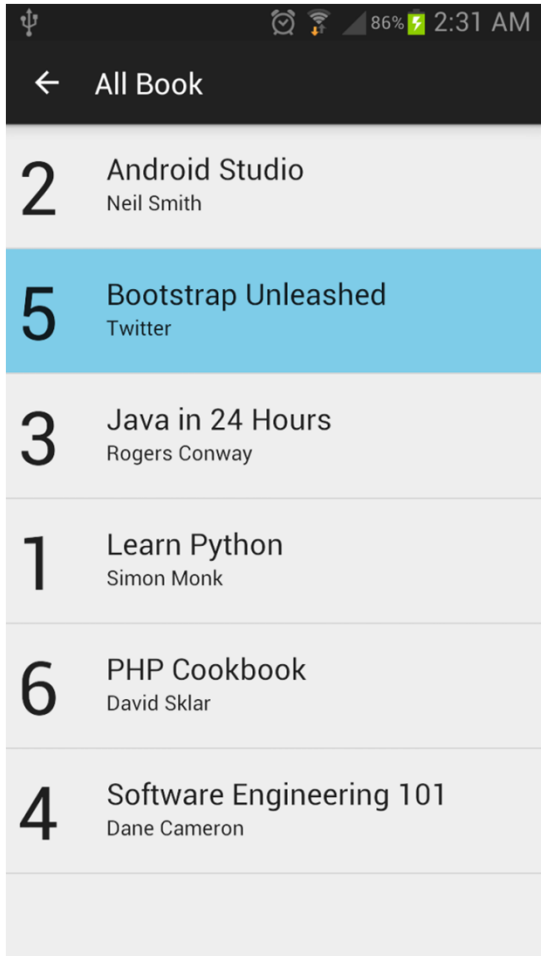
Create Custom CursorAdapter

```
public class BooksCursorAdapter extends CursorAdapter {
 . . .

@Override
 public void bindView(View view, Context context, Cursor cursor) {
   TextView tvId=(TextView) view.findViewById(R.id.txt_id);
   TextView tvTitle=(TextView) view.findViewById(R.id.txt_title);
   TextView tvAuthor=(TextView) view.findViewById(R.id.txt_author);


   tvId.setText(String.valueOf(cursor.getInt(cursor.getColumnIndexOrThrow(B
   ooks._ID))));

   tvTitle.setText(cursor.getString(cursor.getColumnIndexOrThrow(Books.COL_
   TITLE)));

   tvAuthor.setText(cursor.getString(cursor.getColumnIndexOrThrow(Books.COL
   _AUTHOR)));
    }
}
```

Set CursorAdapter on ListView

```
SQLiteDatabase db = dbHelper.getWritableDatabase();

String[] columns={Books._ID,Books.COL_TITLE,Books.COL_AUTHOR};
String orderBy= Books.COL_TITLE + " ASC";

Cursor cursor=db.query(Books.TABLE_NAME, columns, null, null,
   null, null, orderBy);

ListView allBooks=(ListView) findViewById(R.id.listView);
booksCursorAdapter=new BooksCursorAdapter(this,cursor);
allBooks.setAdapter(booksCursorAdapter);
```

SetOnItemClickListener



SetOnItemClickListener

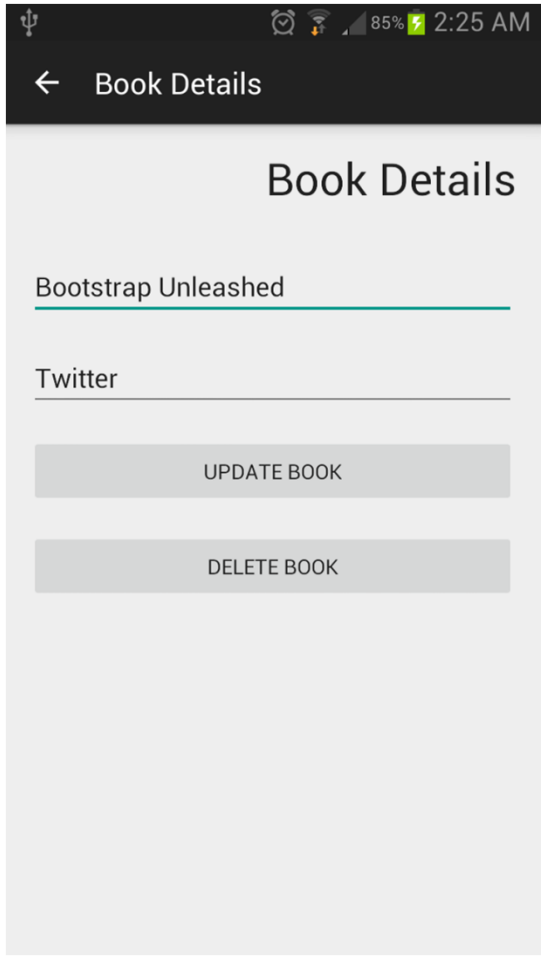
```
allBooks.setOnItemClickListener(new AdapterView.OnItemClickListener() {
   @Override
   public void onItemClick(AdapterView<?> parent, View view, int position,
   long id) {

        Intent i = new Intent(getApplicationContext(), BookActivity.class);
        i.putExtra("bookId", id);
        startActivity(i);

        }
   });
```

BookActivity.class

```
. . .
Intent i=getIntent();
long bid=i.getLongExtra("bookId",0);

. . .
```

QUERY SINGLE RECORD

Query Single Record

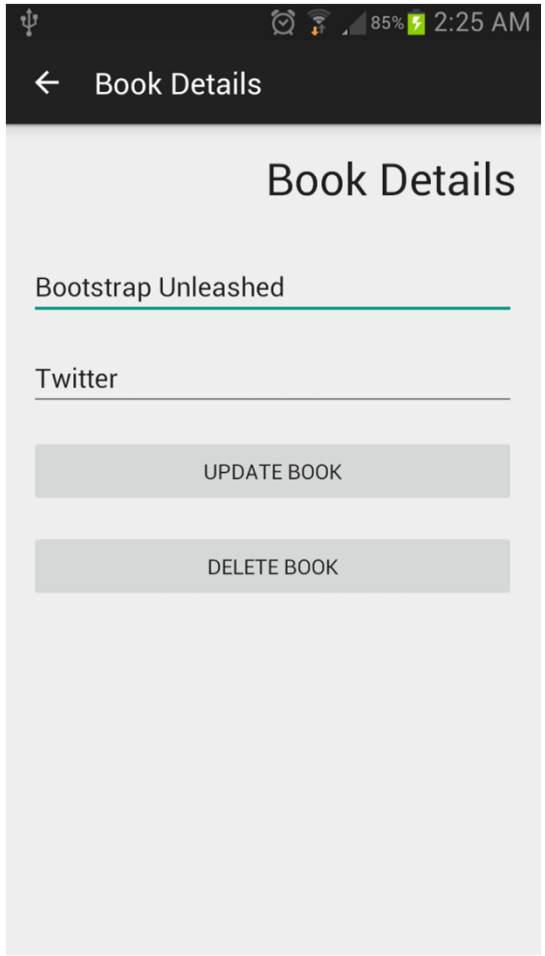


Query Single Record

```
String[] columns={Books._ID,Books.COL_TITLE,Books.COL_AUTHOR};
String whereClause=Books._ID + "=?";
String[] whereArgs={ String.valueOf(bid) };
// String having="";
// String groupBy="";
// String orderBy="";
Cursor cursor=db.query(Books.TABLE_NAME, columns, whereClause,
   whereArgs, null, null, null);

EditText et1=(EditText) findViewById(R.id.bookTitle);
EditText et2=(EditText) findViewById(R.id.bookAuthor);

if (cursor!=null) {
   cursor.moveToFirst();
   et1.setText(cursor.getString(cursor.getColumnIndexOrThrow(Books.COL
   _TITLE)));
   et2.setText(cursor.getString(cursor.getColumnIndexOrThrow(Books.COL
   _AUTHOR)));
}
cursor.close();
```

DELETE RECORD

Delete Record



SQLiteDatabase delete() Method

- **Method for deleting rows in the database.**

```
public int delete (String table, String whereClause, String[]
whereArgs)
```

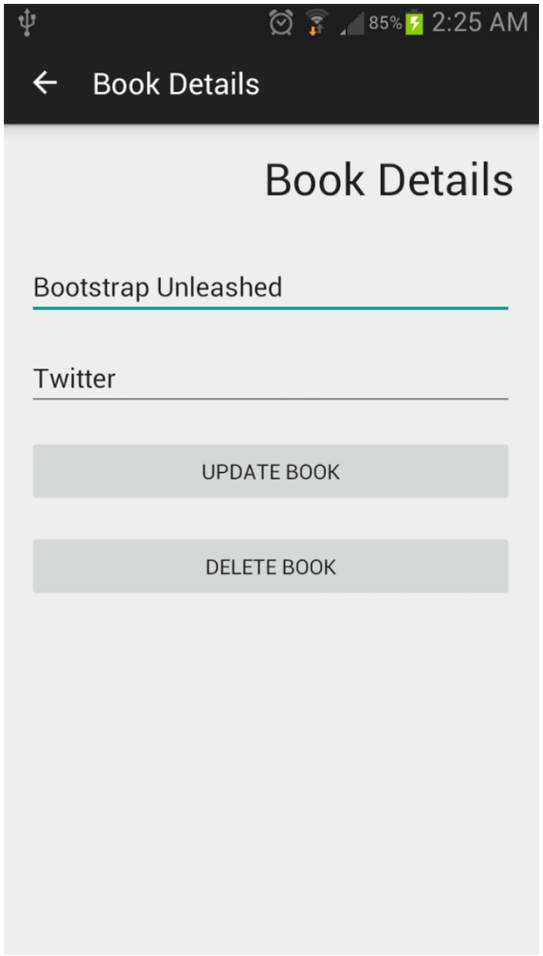
- **Parameters**
  - table the table to delete from
  - whereClause the optional WHERE clause to apply when deleting. Passing null will delete all rows.
  - whereArgs You may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.
- **Returns** the number of rows affected if a whereClause is passed in, 0 otherwise.

Delete Record

```
SQLiteDatabase db = dbHelper.getWritableDatabase();


String whereClause=Books._ID + "=?";
String[] whereArgs={ String.valueOf(bid) };

int rows=db.delete(Books.TABLE_NAME, whereClause, whereArgs);
```

UPDATE RECORD

Update Record



SQLiteDatabase update() Method

- **Method for for updating rows in the database.**

```
public int update (String table, ContentValues values, String
whereClause, String[] whereArgs)
```

- **Parameters**
  - table the table to update in
  - values a map from column names to new column values. null is a valid value that will be translated to NULL.
  - whereClause
  - whereClause the optional WHERE clause to apply when updating. Passing null will update all rows.
  - whereClause You may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings..
- **Returns** the number of rows affected

Update Record

```
EditText et1=(EditText) findViewById(R.id.bookTitle);
EditText et2=(EditText) findViewById(R.id.bookAuthor);
String title=et1.getText().toString();
String author=et2.getText().toString();

ContentValues values=new ContentValues();
values.put(Books.COL_TITLE,title);
values.put(Books.COL_AUTHOR,author);

String whereClause=Books._ID + "=?";
String[] whereArgs={ String.valueOf(id) };
int rows=db.update(Books.TABLE_NAME, values, whereClause,
   whereArgs);
```

Review

- **Define a Schema**
  - Use a Contract class to define Scheme – Create inner class (subclass of BaseColumns) for each table.
- **Get an instance of your SQLite database using SQLiteOpenHelper class.**
  - Implment onCreate() and/or onUpgrade() methods.
- **Perform CREATE, UPDATE, DELETE, SELECT, etc. operations using SQLiteDatabase object.**
  - Get SQLiteDatabase object using getWritableDatabase() or getReadableDatabase() methods.

References

- http://developer.android.com/training/basics/data-storage/databases.html
- http://developer.android.com/guide/topics/data/data-storage.html#db
- http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html
- http://developer.android.com/reference/android/content/ContentValues.html
- http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html
- http://developer.android.com/training/basics/data-storage/shared-preferences.html

Q & A