Lec #5: Content Provider

Tamer Nadeem
Dept. of Computer Science
Objective

• Data Storage
  • SQLite
  • Content Provider

• Two Presentation
  – Case Studies (7-13) - cont’d
    • Presenter: Ashok Kumar
  – Your Apps Are Watching You
    • Presenter: Elliott Peay
SQLite
SQLite

• Embedded database for Android
• Supports 3 main data types
  – TEXT (like Java String)
  – INTEGER (like Java long)
  – REAL (like Java double)
• No type checking inherent in SQLite
• Each DB is private to App that created it

Documentation on SQLITE available at: http://www.sqlite.org/sqlite.html
Good GUI tool for SQLITE available at: http://sqliteadmin.orbmu2k.de/
SQL and SQLite Language Syntax at: http://www.sqlite.org/lang.html
Using SQLite

• Create a subclass of **SQLiteOpenHelper**
  – Override at least **onCreate()** - where we can create tables
  – Can also override **onUpgrade()** - make a modification to the tables structures

• Call **getWritableDatabase()** to **get read/write instance of** **SQLiteDatabase**
  – Can then call insert(), delete(), update()
public class DatabaseHelper extends SQLiteOpenHelper {
    static final String dbName = "demoDB";
    static final String employeeTable = "Employees";
    static final String colID = "EmployeeID";
    static final String colName = "EmployeeName";
    static final String colAge = "Age";
    static final String colDept = "Dept";
    static final String deptTable = "Dept";
    static final String colDeptID = "DeptID";
    static final String colDeptName = "DeptName";
    
    public DatabaseHelper(Context context) {
        super(context, dbName, null, 1);
    }

    …
}

Data Stored at:
/data/data/<project name>/databases/<db name>
onCreate(SQLiteDatabase db)

public void onCreate(SQLiteDatabase db) {
    // TODO Auto-generated method stub
    db.execSQL("CREATE TABLE " + deptTable + " (" + 
               colDeptID + " INTEGER, " + colDeptName + " TEXT);\n    ");

    db.execSQL("CREATE TABLE " + employeeTable + " (" + 
               colID + " INTEGER, " + 
               colName + " TEXT, " + colAge + " INTEGER, " + colDept + 
               " INTEGER);\n    ");
}
Handling Records

SQLiteDatabase db = this.getWritableDatabase();

    //Inserting Records
    ContentValues cv = new ContentValues();
    cv.put(colDeptID, 1);
    cv.put(colDeptName, "Sales");
    db.insert(deptTable, colDeptID, cv);

    db.execSQL("INSERT INTO " + deptTable + "(" + colDeptID + "," +
    colDeptName + ") values ("1", 'Sales');");

    //Deleting Record
    db.delete(deptTable, colDeptID + "=" + dept_1, null);  //dept_1="1"

    //Updating Record
    cv.put(colDeptName, "newIT");
    db.update(deptTable, cv, colDeptID + "=?", dept_2);  //dept_2="1"

    db.close();

Querying SQL

SQL Select Syntax (see http://www.sqlite.org/lang.html)

SQL-select statements are based on the following components

```
select    field₁, field₂, ... , fieldₙ
from      table₁, table₂, ... , tableₙ
where     ( restriction-join-conditions )
order by  fieldₙ₁, ..., fieldₙₘ
group by  fieldₘ₁, ... , fieldₘₖ
having    (group-condition)
```

The first two lines are mandatory, the rest is optional.
Querying SQLite

• SQLiteDatabase.query() returns a **Cursor object** that points to results
  – Cursor is a **iterator**. Call moveToNext() to get to next element.
  – Cursor can be examined at runtime
    • getCount(), getColumnCount()
    • getColumnIndex(String name)
    • getColumnName(int index)
    • getInt(int index), getLong(int index), getString(int index), etc.
Querying SQLite – query()

Android Simple Queries

The signature of the Android’s simple query method is:

```java
query( String table,
      String[] columns,
      String selection,
      String[] selectionArgs,
      String groupBy,
      String having,
      String orderBy )
```

Android Simple Queries Example

Query the EmployeeTable, find the average salary of female employees supervised by 123456789. Report results by Dno. List first the highest average, and so on, do not include depts. having less than two employees.

```java
String[] columns =  
    {"Dno", "Avg(Salary) as AVG"};
String[] conditionArgs =  
    {"F", "123456789"};
Cursor c = db.query(  
    "EmployeeTable",  
    columns,  
    "sex = ? And superSsn = ? " ,  
    conditionArgs,  
    "Dno",  
    "Count(*) > 2",  
    "AVG Desc "
);  
```
String [] columns = {"recID", "name", "phone"};

Cursor myCur = db.query("tblAMIGO", columns,
                        null, null, null, null, null, "recID");

int idCol = myCur.getColumnIndex("recID");
int nameCol = myCur.getColumnIndex("name");
int phoneCol = myCur.getColumnIndex("phone");

while (myCur.moveToNext()) {
    columns[0] = Integer.toString((int)myCur.getInt(idCol));
    columns[1] = myCur.getString(nameCol);
    columns[2] = myCur.getString(phoneCol);

    txtMsg.append("\n" + columns[0] + " "
                + columns[1] + " "
                + columns[2] );
}

Content Provider
Content Provider

• A content provider makes a specific set of the application's data available to other applications

  => Share data to other apps

• Any app with appropriate permission, can read and write the data.

• Many native databases are available via the content providers, for example Contact Manager

• Common interface for querying the data
About Content Provider

• Content providers expose their data as a simple table on a database model

<table>
<thead>
<tr>
<th>_ID</th>
<th>NUMBER</th>
<th>NUMBER_KEY</th>
<th>LABEL</th>
<th>NAME</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>(425) 555 6677</td>
<td>425 555 6677</td>
<td>Kirkland office</td>
<td>Bully Pulpit</td>
<td>TYPE_WORK</td>
</tr>
<tr>
<td>44</td>
<td>(212) 555-1234</td>
<td>212 555 1234</td>
<td>NY apartment</td>
<td>Alan Vain</td>
<td>TYPE_HOME</td>
</tr>
<tr>
<td>45</td>
<td>(212) 555-6657</td>
<td>212 555 6657</td>
<td>Downtown office</td>
<td>Alan Vain</td>
<td>TYPE_MOBILE</td>
</tr>
<tr>
<td>53</td>
<td>201.555.4433</td>
<td>201 555 4433</td>
<td>Love Nest</td>
<td>Rex Cars</td>
<td>TYPE_HOME</td>
</tr>
</tbody>
</table>

• Every record includes a numeric _ID field that uniquely identifies the record within the table.
About Content Provider

• Content provider exposes a public URI that uniquely identifies its data set:

```plaintext
content://<Authority>/(n path)/[instance identifier]
```

• the URI starts with content:// scheme.

• the Authority is a unique identifier for the content provider (fully-qualified class name)

• the Authority can be followed by one or more paths (optional) refer to data paths within the content.

• there can be an instance identifier that refers to a specific data instance.
  • content://media/internal/images - return the list of all internal images.
  • content://com.android.contacts/data/phones - return the list of all contact names.
  • content://contacts/people/45 - return the single result row, the contact with ID=45.
Android Native ContentProvider

• **Browser**—Read or modify bookmarks, browser history, or web searches.

• **CallLog**—View or update the call history.

• **Contacts**—Retrieve, modify, or store the personal contacts. Three-tier data model of tables under a `ContactsContract` object:
  • `ContactsContract.Data`—Contains all kinds of personal data.
  • `ContactsContract.RawContacts`—Contains a set of Data objects associated with a single account or person.
  • `ContactsContract.Contacts`—Contains an aggregate of one or more RawContacts, presumably describing the same person.

• **MediaStore**—Access audio, video, and images.

• **Setting**—View and retrieve Bluetooth settings, ring tones, and other device preferences.
Android defines CONTENT_URI constants for all the providers that come with the platform.

- `ContactsContract.CommonDataKinds.Phone.CONTENT_URI` (content://com.android.contacts/data/phones)
- `Browser.BOOKMARKS_URI` (content://browser/bookmarks)

*Classes section under:*

*(if a provider, CONTENT_URI field exist)*
### Constants

<table>
<thead>
<tr>
<th>String</th>
<th>CONTENT_ITEM_TYPE</th>
<th>This constant is deprecated. see ContactsContract</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>CONTENT_TYPE</td>
<td>This constant is deprecated. see ContactsContract</td>
</tr>
<tr>
<td>String</td>
<td>DEFAULT_SORT_ORDER</td>
<td>This constant is deprecated. see ContactsContract</td>
</tr>
<tr>
<td>String</td>
<td>PERSON_ID</td>
<td>This constant is deprecated. see ContactsContract</td>
</tr>
</tbody>
</table>

### Inherited Constants

- From interface android.provider.BaseColumns
- From interface android.provider.Contacts.PeopleColumns
- From interface android.provider.Contacts.PhonesColumns

### Fields

<table>
<thead>
<tr>
<th>public static final Uri</th>
<th>CONTENT_FILTER_URL</th>
<th>This field is deprecated. see ContactsContract</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static final Uri</td>
<td>CONTENT_URI</td>
<td>This field is deprecated. see ContactsContract</td>
</tr>
</tbody>
</table>

### Public Methods

- `final static CharSequence getDisplayLabel(Context context, int type, CharSequence label, C` This method is deprecated. see ContactsContract
- `final static CharSequence getDisplayLabel(Context context, int type, CharSequence label, C` This method is deprecated. see ContactsContract

### Inherited Methods

- From class java.lang.Object
Querying Native Content Provider

• You need
  – URI
    • ContactsContract.Contacts.CONTENT_URI
  – Names of data fields (result comes in table)
    • ContactsContract.Contacts.DISPLAY_NAME
  – Data types of those fields
    • String

• Remember to modify the manifest file for permissions!

To retrieve the contacts:

<uses-permission android:name="android.permission.READ_CONTACTS"/>
</uses-permission>
import android.provider.Contacts.People;
import android.content.ContentUris;
import android.net.Uri;
import android.database.Cursor;

// Use the ContentUris method to produce the base URI for the contact with _ID == 23.
// → "content://com.android.contacts/people/23"
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);

// Alternatively, use the Uri method to produce the base URI.
// It takes a string rather than an integer.
// → "content://com.android.contacts/people/23"
Uri myPerson = Uri.withAppendedPath(People.CONTENT_URI, "23");

// Then query for this specific record:
Cursor cur = managedQuery(myPerson, null, null, null, null);

For more information on URI handling functions:
import android.provider.Contacts.People;
import android.database.Cursor;

// Form an array specifying which columns to return.
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};

// Get the base URI for the People table in the Contacts content provider.
Uri contacts = People.CONTENT_URI;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection,  // Which columns to return
    null,       // Which rows to return (all rows)
    null,       // Selection arguments (none)
    // Put the results in ascending order by name
    People.NAME + " ASC");
Insert/Update/Delete

private void insertRecords(String name, String phoneNo) {
    ContentValues values = new ContentValues();
    values.put(People.NAME, name);
    Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
}

private void updateRecord(int recNo, String name) {
    Uri uri = ContentUris.withAppendedId(People.CONTENT_URI, recNo);
    ContentValues values = new ContentValues();
    values.put(People.NAME, name);
    getContentResolver().update(uri, values, null, null);
}

private void deleteRecords() {
    Uri uri = People.CONTENT_URI;
    getContentResolver().delete(uri, null, null);
}
Implementing your own Content Provider

1. Set up a system for storing the data
   - e.g., SQLite using SQLiteOpenHelper

2. Extend ContentProvider class to provide access:
   - query()—Allows third-party applications to retrieve content.
   - insert()—Allows third-party applications to insert content.
   - update()—Allows third-party applications to update content.
   - delete()—Allows third-party applications to delete content.
   - getType()—Allows third-party applications to read each of URI structures supported.
   - onCreate()—Creates a database instance to help retrieve the content.

3. Declare the Content Provider in manifest
public class MyContentProvider extends ContentProvider {
    public static final String AUTHORITY = "edu.odu.phonenumber";
    public static final Uri CONTENT_URI =
        Uri.parse("content://"+ AUTHORITY + "/phones");
    private MySQLDatabase mDB;
    private static final int PHONES = 1;
    private static final int PHONES_ID = 2;
    private static final UriMatcher uriMatcher;
    static{
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI(AUTHORITY, "phones", PHONES);
        uriMatcher.addURI(AUTHORITY, "phones/#", PHONES_ID);
    }
    public boolean onCreate() {
        mDB = new MySQLDatabase(getContext());
        return true;
    }
    ...
}
public class MyContentProvider extends ContentProvider {
...
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        Cursor c=null;
        SQLiteDatabase db = mDB.getWritableDatabase();
        switch (uriMatcher.match(uri)) {
            case PHONES: //get all phones records
                ...
                break;
            case PHONES_ID: //get a particular phone record
                ...
                break;
        }
...
    }

    public String getType(Uri uri) {...}
    public int delete(Uri uri, String selection, String[] selectionArgs) {...}
    public Uri insert(Uri uri, ContentValues values) {...}
    public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {...}
}
Manifest

```xml
<application android:icon="@drawable/icon" android:label="@string/app_name">
  ...

  <provider android:name=".MyContentProvider"
           android:authorities="edu.odu.phonenumber"></provider>

</application>
```
Debugging

The main components that comprise Android debugging environment are:

- **Device** or **Android Virtual Device**

- **Delvik Debug Monitoring Service (DDMS)**: graphical program that can capture screenshots, gather thread and stack information, spoof incoming calls and SMS messages, and other features.

- **The Android Debug Bridge (ADB)**: provides various device management capabilities.

- **JDWP debugger**: allows you to step through code, view variable values, and pause execution of an application. A JDWP-compliant debugger is already included in Eclipse.
DDMS (Window > Open Perspective > Other... > DDMS)

The Log.e() method is used to log errors.
The Log.w() method is used to log warnings.
The Log.i() method is used to log informational messages.
The Log.d() method is used to log debug messages.
The Log.v() method is used to log verbose messages.
The Log.wtf() method is used to log terrible failures that should never happen. (“WTF” stands for “What a Terrible Failure!” of course.)

```java
private static final String TAG = "MyApp";
Log.i(TAG, "I am logging something informational!");
```
Debug (Window > Open Perspective > Debug)

```xml
<application android:icon="@drawable/icon" android:label="@string/app_name" android:debuggable="true" />
```
Questions?
TO DO

• Project teams (team’s name, team’s member) – due date: Friday Sep 16\textsuperscript{th}, 11:59pm

• Assignment #3: Boggle - due date: Sun Sep 25\textsuperscript{th}, 11:59pm