ANDROID APPS DEVELOPMENT FOR MOBILE GAME

Lecture 7: Data Storage and Web Services

Peter Lo

Overview

- Android provides several options for you to save persistent application data.

| Storage Option	Description
Shared Preferences	Store private primitive data in key-value pairs.
Internal Storage	Store private data on the device memory.
External Storage	Store public data on the shared external storage (e.g. SD Card).
SQLite Databases	Store structured data in a private database.
Network Connection	Store data on the web with network server

The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications and how much space your data requires

2

Android Folder Structure

- User the emulator's **File Explorer** to see and manage your device's storage structure

Internal Main Memory

For old version of Android

External SD Card
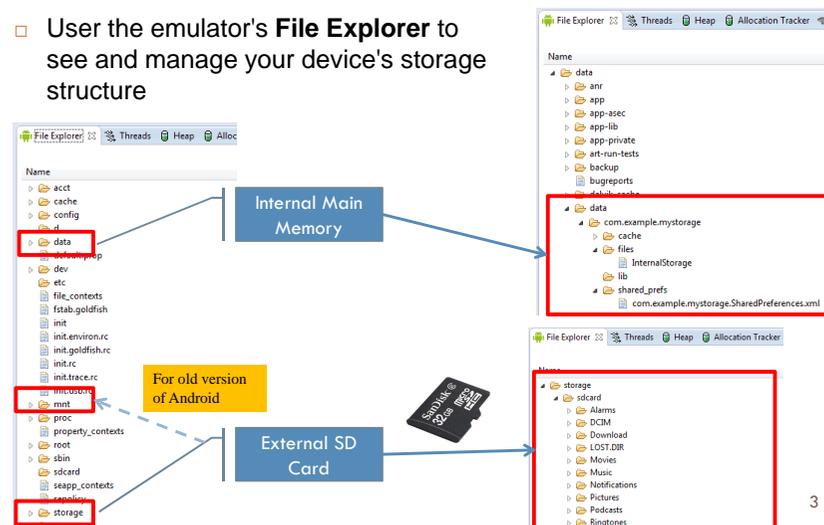
3

Android File Structure

- Android allows to persists application data via the file system.
- For each application the Android system creates a *data/data/[application package]* directory.
- On Android, all internal application data objects (including files) are private to that application.

SQLite Database

Internal storage file

Shared Preference

4

Shared Preferences

- Is an Android lightweight mechanism to store and retrieve <Key-Value> pairs of primitive data types.

- If you have a relatively small collection of key-values that you'd like to save, you should use the Shared Preferences.

- It is typically used to keep state information and shared data among several activities of an application.

- In each entry of the form <key-value> the key is a string and the value must be a primitive data type.

Shared preferences are not strictly for saving "user preferences" such as what ringtone a user has chosen.

key	value
firstName	Bugs
lastName	Bunny
location	Earth

Get a Handle to a SharedPreferences

- There are two methods to access the preference:
  - getPreferences()
    - Use this from an Activity if you need to use only **one** shared preference file for the activity.
    ```
    SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
    ```
  - getSharedPreferences()
    - Use this if you need **multiple** shared preference files identified by name, which you specify with the first parameter.
    ```
    Context context = getActivity();
    SharedPreferences sharedPref = context.getSharedPreferences(
            getString(R.string.preference_file_key), Context.MODE_PRIVATE);
    ```

When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE_FILE_KEY"
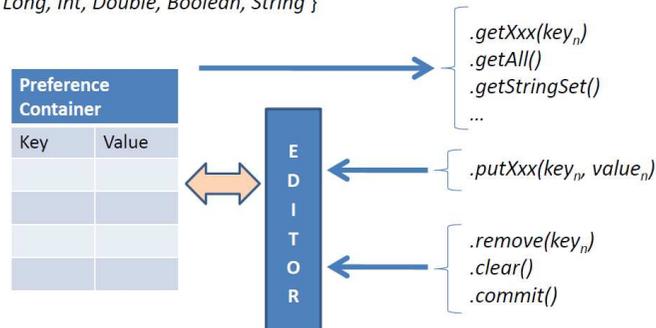
6

Using Preferences API calls

- All of the get preference methods return a preference object whose contents can be manipulated by an editor that allows put and get commands to place data in and out of the preference container.

Xxx = { Long, Int, Double, Boolean, String }

```
.getXxx(key_n)
.getAll()
.getStringSet()
...
```

```
.putXxx(key_n, value_n)
```

```
.remove(key_n)
.clear()
.commit()
```

Preference Container

Key	Value

EDITOR

                                                                7

Read from Shared Preferences

- To retrieve values from a shared preferences file, call methods such as *getInt()* and *getString()*, providing the key for the value you want, and optionally a default value to return if the key isn't present.

```
// Retrieve and hold the contents of the preferences file
SharedPreferences preferences = getSharedPreferences(PREFERENCE_FILE, Context.MODE_PRIVATE);

// Retrieve a String value from the preferences
mDataString = preferences.getString(PREFERENCE_KEY, "");
```

Default value to return if the preference not exist

Operating mode
- Use 0 or MODE_PRIVATE for the default operation,
- MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE to control permissions.
- MODE_MULTI_PROCESS used if multiple processes are mutating the same SharedPreferences file.

8

Write to Shared Preferences

- To write to a shared preferences file, create a *SharedPreferences.Editor* by calling *edit()* on your Shared Preferences.
- Pass the keys and values you want to write with methods such as *putInt()* and *putString()*.
- Call *commit()* to save the changes.

```
// Retrieve and hold the contents of the preferences file
SharedPreferences preferences = getSharedPreferences(PREFERENCE_FILE, Context.MODE_PRIVATE);

// Create a new Editor for these preferences
SharedPreferences.Editor editor = preferences.edit();

// Set a String value in the preferences editor
editor.putString(PREFERENCE_KEY, mDataString);

// Commit your preferences changes
editor.commit();
```

Android File Storage

Internal Storage	External Storage
- Always available.	- Not always available (e.g. USB storage)
- Files saved here are accessible by only your app by default.	- Files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes all your app's files from internal storage.	- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from *getExternalFilesDir()*.
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Write a File to Internal Storage

- To create and write a private file to the internal storage:
  - Call *openFileOutput()* with the name of the file and the operating mode. This returns a *FileOutputStream*.
  - Write to the file with *write()*.
  - Close the stream with *close()*.

```
// Open a private file for writing
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);

// Constructs a new DataOutputStream on the OutputStream out
DataOutputStream out = new DataOutputStream(fos);

// Write to file in Unicode format
out.writeUTF(mDataString);

// Close file
out.close();

// Closes the file stream.
fos.close();
```

MODE_PRIVATE will create the file and make it private to your application.
Other modes available are: MODE_APPEND, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE.

Read a File from Internal Storage

- To read a file from internal storage:
  - Call *openFileInput()* and pass it the name of the file to read. This returns a *FileInputStream*.
  - Read bytes from the file with *read()* or *readUTF()*.
  - Close the stream with *close()*.

```
// Open a private file for reading
FileInputStream fos = openFileInput(FILENAME);

// Constructs a new DataInputStream on the InputStream in
DataInputStream in = new DataInputStream(fos);

// Read the content in Unicode format
mDataString = in.readUTF();

// Closes this stream
in.close();

// Closes the file stream.
fos.close();
```

If you want to save a static file in your application at compile time, save the file in your project *res/raw/* directory. You can open it with *openRawResource*(), passing the *R.raw.<filename>* resource ID. This method returns an *InputStream* that you can use to read the file (but you cannot write to the original file).

Checking Media Availability

- Before you do any work with the external storage, you should always check whether the media is available.
- The media might be mounted to a computer, missing, read-only, or in some other state.

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

The *getExternalStorageState()* method returns other states that you might want to check, such as whether the media is being shared (connected to a computer), is missing entirely, has been removed badly, etc. You can use these to notify the user with more information when your application needs to access the media.

13

Files Categories of External Storage

Public Directory	Private Directory
- Files that should be freely available to other apps and to the user. - When the user uninstalls your app, these files should remain available to the user. - E.g. photos captured by your app or other downloaded files.	- Files that rightfully belong to your app and should be deleted when the user uninstalls your app. - Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app. - When the user uninstalls your app, the system deletes all files in your app's external private directory. - E.g. additional resources downloaded by your app or temporary media files.

14

Creating Files in Public Directory

- If you want to save public files on the external storage, use the *getExternalStoragePublicDirectory()* method to get a File representing the appropriate directory on the external storage.
- The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as *DIRECTORY_MUSIC* or *DIRECTORY_PICTURES*.

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

15

Permission for Access SD Card

- When you deal with external files you need to request permission to read and write on the SD card.
- The following clauses need to add to the AndroidManifest.xml

```
<uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

16

Creating Files in Private Directory

- If you want to save files that are private to your app, you can acquire the appropriate directory by calling *getExternalFilesDir()* and passing it a name indicating the type of directory you'd like.
- Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

```java
public File getExternalFilesDir(String FolderName) {
    // Get the directory for the private directory
    File folder = new File(Environment.getExternalStorageDirectory(), FolderName);

    // Create the folder
    if (!folder.mkdirs()) {
        Log.e("LOG_TAG", "Directory not created");
    }

    // Return the folder name
    return folder;
}
```

SQLite

- Using an SQLite database in Android does not require a setup procedure or administration of the database.
  - SQLite is embedded into every Android device.
- You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.

create a database,	Insert rows,
define SQL tables,	delete rows,
indices,	change rows,
queries,	run queries and
views,	administer a SQLite database file
triggers	

Beware of Sharing Issues

- You cannot access internal databases belonging to other people (instead use Content Providers or external SD resident DBs).
- An SD resident database requires the declaration of following permissions in AndroidManifest file:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

SQLITE (as well as most DBMS) is not case sensitive

Create a Database Using a SQL Helper

```java
public class MySQLiteOpenHelper extends SQLiteOpenHelper {

    public static final int DATABASE_VERSION = 1;

    public static final String DATABASE_NAME = "MyDatabase";

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

    public void onCreate(SQLiteDatabase db) {
        String SQL_CREATE_ENTRIES = "CREATE TABLE ...";
        db.execSQL(SQL_CREATE_ENTRIES);
    }

    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS ...");
        onCreate(db);
    }
}
```

If you change the database schema, you must increment the database version.

Called by the framework if the database not yet created

If the database version is increased in your application code. This method allows you to update an existing database schema or to drop the existing database and recreate it

Put Information into a Database

□ Insert data into the database by passing a *ContentValues* object to the *insert()* method:

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

ContentValues values = new ContentValues();
values.put(COL_ID, id);
values.put(COL_NAME, value);

long newRowId = db.insert(TABLE_NAME, null, values);

db.close();
```

Get reference to writable database

Create a new map of values, where column names are the keys

Insert the new row, returning the primary key of the new row
• *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 not set to null, this 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

Close the database connection

Delete Information from a Database

□ To delete rows from a table, you need to provide selection criteria that identify the rows.

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

String selection = COL_NAME + " LIKE ?";

String[] selectionArgs = { String.valueOf(rowId) };

db.delete(TABLE_NAME, selection, selectionArgs);

db.close();
```

Get reference to writable database

Define 'where' part of query. Passing "1" will remove all rows.

Specify arguments in placeholder order

Delete the row, returning the number of rows affected

Close the database connection

Update a Database

□ When you need to modify a subset of your database values, use the *update()* method. Updating the table combines the content values syntax of *insert()* with the where syntax of *delete()*.

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

ContentValues values = new ContentValues();
values.put(COL_STUDENT, student);
values.put(COL_MARK, mark);

String selection = COL_ID + " = ?";

String[] selectionArgs = new String[] { String.valueOf(id) };

int rowAffected = db.update(TABLE_NAME, values, selection, selectionArgs);
db.close();
```

Get reference to writable database

Create a new map of values, where column names are the keys

Define 'where' part of query. Passing null will update all rows.

Specify arguments in placeholder order

Updating the table, returning the number of rows affected

Close the database connection

Read Information from a Database

□ To read from a database, use the *query()* method, passing it your selection criteria and desired columns.

□ The results of the query are returned to you in a *Cursor* object.

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

Cursor cursor = db.query(
        TABLE_NAME,
        projection,
        selection,
        selectionArgs,
        groupBy,
        having,
        sortOrder );

db.close();
```

Get reference to writable database

Query the given table, returning a Cursor over the result set.
• *Table* – The table name to compile the query against.
• *projection* – A list columns to return. Passing null will return all columns.
• *selection* – Define 'where' part of query. Passing null will return all rows for the given table.
• *selectionArgs* – Specify arguments in placeholder order
• *groupBy* – Declaring how to group rows, formatted as an SQL GROUP BY clause. Passing null will cause the rows to not be grouped.
• *having* – Declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause. Passing null will cause all row groups to be included.
• *sortOrder* – Define the sorting order for the results, formatted as an SQL ORDER BY clause. Passing null will use the default sort order, which may be unordered.

Close the database connection

Using Cursor

- A Cursor represents the result of a query and basically points to one row of the query result. This way Android can buffer the query results efficiently; as it does not have to load all data into memory.
- To look at a row in the cursor, use one of the Cursor move methods, which you must always call before you begin reading values.

```
if (cursor.moveToFirst()) {
    do {
        int id = cursor.getInt(0);
    } while (cursor.moveToNext());
}
```

Start by calling *moveToFirst()*, which places the read position on the first entry in the results.

For each row, you can read a column's value by calling one of the Cursor get methods, such as *getString()* or *getInt()*.

Using *moveToNext()* methods to move between individual data rows.

Web Service

- Web services are software systems that have taken the concept of services delivered over the web to support interoperable machine-to-machine interaction over a network
- It is a method of communications between two electronic devices over the WWW.
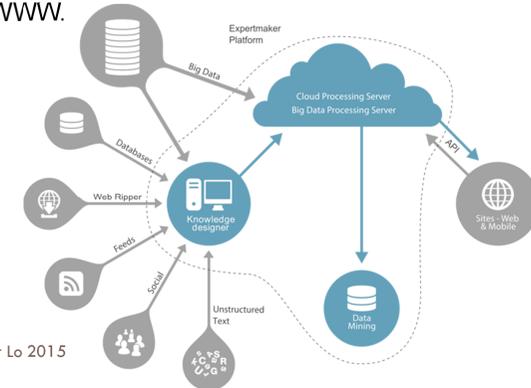
RESTful Web services

- REST (Representational State Transfer) is a technology relies on a stateless, client-server, cacheable communications technology that uses the HTTP protocol
- In REST, the web services are viewed as resources and can be identified by their URLs.
- Web service clients that want to use these resources and access a particular representation will need to use a globally defined set of remote methods that describe the action to be performed on the resource
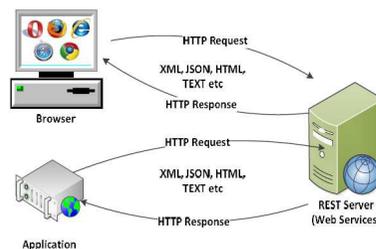
HTTP GET

- The GET method means retrieve whatever information is identified by the Request-URI.
- If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.
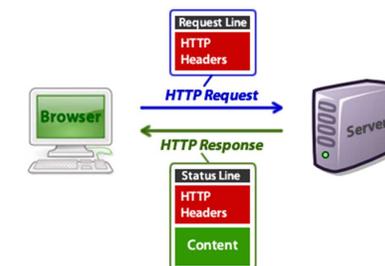
Create a HTTP GET Request

Create HTTP client object to send request to server

Construct URL string

```
HttpClient Client = new DefaultHttpClient();

String URL = "http://www.website.com/demo.php?var1=value1";

HttpGet httpget = new HttpGet(URL);

ResponseHandler<String> responseHandler = new BasicResponseHandler();

String SetServerString = Client.execute(httpget, responseHandler);
```
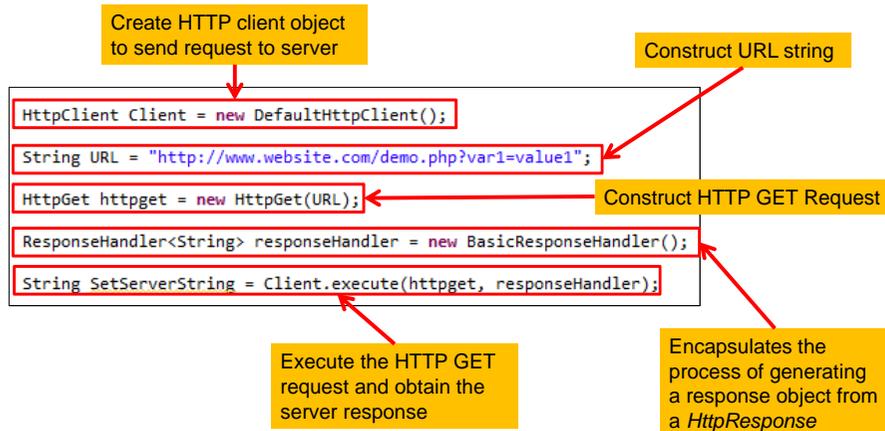
Construct HTTP GET Request

Execute the HTTP GET request and obtain the server response

Encapsulates the process of generating a response object from a *HttpResponse*

JSON (JavaScript Object Notation) Structure

□ JSON is built on two structures:

- **JSON Object:** A collection of name/value pairs which represented by { }.

```
{
    "employee": {
        "name":    "sachin",
        "salary":   56000,
        "married":  true
    }
}
```

- **JSON Array:** An ordered list of values which represented by [ ].

```
{ "Employee" :
    [
        {"id":"101","name":"Sonoo Jaiswal","salary":"50000"},
        {"id":"102","name":"Vimal Jaiswal","salary":"60000"}
    ]
}
```

JSON String

□ Consider this sample JSON string:

```
{"FirstObject": { "attr1":"one value" ,"attr2":"two value",
"sub": { "sub1":[ {"sub1_attr":"sub1_attr_value" },{"sub1_attr":"sub2_attr_value" }]}
}
}
```

□ Or we can reformat it as follow:

```
{
    "FirstObject": {
        "attr1": "one value",
        "attr2": "two value",
        "sub": {
            "sub1": [
                {
                    "sub1_attr": "sub1_attr_value"
                },
                {
                    "sub1_attr": "sub2_attr_value"
                }
            ]
        }
    }
}
```

Parsing JSON Object

Create a JSONObject with the received response string

Get the main object from the created JSON object by using *getJSONObject()* method

```
JSONObject jsonObject = new JSONObject(ResponseString);

JSONObject object = jsonObject.getJSONObject("FirstObject");

String attr1 = object.getString("attr1");
String attr2 = object.getString("attr2");

JSONObject subObject = object.getJSONObject("sub");

JSONArray subArray = subObject.getJSONArray("sub1");

for (int i=0; i<subArray.length(); i++) {
    String attr3 = subArray.getJSONObject(i).getString("attr").toString();
}
```

Get 2 strings by using *getString()* method.

Get the sub object by using the *getJSONObject()* method.

Get this JSON array by using *getJSONArray()* method

Process this array same as simple string array to obtain the values