CSE 486/586 Distributed Systems
Content Providers & Services

Content Providers

- A content provider provides a table view of data.
- If you write a content provider, any client application with the permission can enter/read/update/delete data items in your content provider.
- A client application (that uses your content provider) uses *ContentResolver* to interact with your content provider.
- You need to extend *ContentProvider* and implement necessary methods.

How a Client Interacts

- Table identification → URI (android.net.Uri)
  – E.g., content://user_dictionary/words
- Insert
  – public final Uri ContentResolver.insert (Uri url, ContentValues values)
- Update
  – public final int ContentResolver.update (Uri uri, ContentValues values, String where, String[] selectionArgs)
- Query
  – public final Cursor ContentResolver.query (Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
- Delete
  – public final int ContentResolver.delete (Uri url, String where, String[] selectionArgs)

How to Write a Content Provider

1. Declare in AndroidManifest.xml
2. Define a URI that client apps will use
3. Define permissions
4. Implement necessary methods in *ContentProvider*
5. When implementing *ContentProvider*, use either the Android file system or SQLite as the actual data storage.

Declare in AndroidManifest.xml

```
<manifest ... >
  ...
  <application ... >
      <provider android:name=".ExampleProvider" />
      ...
  </application>
</manifest>
```

Defining a URI

- Typical format
  – content://<authority>/<table name>
  – Authority: a global (Android-wide) name for the provider
    » E.g., edu.buffalo.cse.cse486.proj1.provider
  – Table name: the name of a table that the provider exposes
    » Note: a provider can expose more than one table.
- Should be added to AndroidManifest.xml
  – E.g., <provider android:authorities="edu.buffalo.cse.cse486.proj1.provider" …>…</provider>

Define Permissions

- Should define permissions (for others) in AndroidManifest.xml
- android:permission: Single provider-wide read/write permission.
  - E.g., <provider android:permission="edu.buffalo.cse.cse486.proj1.provider.permission.USE_PROJ1_PROVIDER" …>…</provider>
- android:readPermission: Provider-wide read permission.
- android:writePermission: Provider-wide write permission.

Necessary Methods

- query()
  - Retrieve data from your provider.
- insert()
  - Insert a new row into your provider.
- update()
  - Update existing rows in your provider.
- delete()
  - Delete rows from your provider.
- getType()
  - Return the MIME type corresponding to a content URI.
- onCreate()
  - Initialize your provider. The Android system calls this method immediately after it creates your provider. Notice that your provider is not created until a ContentResolver object tries to access it.
- These need to handle concurrent accesses (need to be thread-safe)

Storage Options

- Internal storage: file system, private to the app
- External storage: file system, open to everybody
- SQLite: database, private to the app
- Read: http://developer.android.com/guide/topics/data/data-storage.html

Internal Storage

- Saving files directly on the device's internal storage.
- User uninstallation → files are removed.
- 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().
- E.g.,

```
String FILENAME = "hello_file";
String string = "hello world!";
FileOutputStream fos = openFileOutput(FILENAME,
   Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

Internal Storage

- MODE_PRIVATE → create the file (or replace a file of the same name) and make it private to your application.
- Other modes available are:
  - MODE_APPEND, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE.
- 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().
  - Then close the stream with close().

External Storage

- Shared "external storage"
  - E.g., a removable storage media (such as an SD card) or an internal (non-removable) storage.
- Files saved to the external storage are:
  - World-readable
  - Can be modified by the user when they enable USB mass storage to transfer files on a computer.
- Checking media availability
  - Before you do any work with the external storage, you should always call getExternalStorageState() to check whether the media is available.

External Storage

- Accessing files on external storage (API Level 8 or greater)
  - Use getExternalFilesDir() to open a File that represents the external storage directory where you should save your files.
  - This method takes a type parameter that specifies the type of subdirectory you want, such as DIRECTORY_MUSIC and DIRECTORY_RINGTONES (pass null to receive the root of your application's file directory). This method will create the appropriate directory if necessary.
  - If the user uninstalls your application, this directory and all its contents will be deleted.

External Storage

- Saving files that should be shared
  - For files not specific to your application and that should not be deleted when your application is uninstalled
  - Save them to one of the public directories on the external storage.
  - These directories lay at the root of the external storage, such as Music/, Pictures/, Ringtones/, and others.
- (API Level 8 or greater)
  - Use getExternalStoragePublicDirectory(), passing it the type of public directory you want, such as DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_RINGTONES, or others. This method will create the appropriate directory if necessary.

Services

- A service runs in the background with no UI for long-running operations.
  - Playing music, sending/receiving network messages, …
  - Subclass of android.app.Service
- Started service
  - A service is "started" when an application component (such as an activity) starts it by calling startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
- Bound service
  - A service is "bound" when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

How to Write a Service

- Declare in AndroidManifest.xml
- Implement necessary methods in *Service*

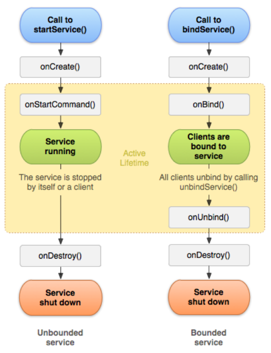
Declare in AndroidManifest.xml

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Necessary Methods

- onStartCommand()
  - The system calls this method when another component, such as an activity, requests that the service be started, by calling startService().
- onBind()
  - The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling bindService().
- onCreate()
  - The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either onStartCommand() or onBind()).
- onDestroy()
  - The system calls this method when the service is no longer used and is being destroyed.

Service Lifecycle


Call to startService()
onCreate()
onStartCommand()
Service running
The service is stopped by itself or a client
onDestroy()
Service shut down
Unbounded service
Call to bindService()
onCreate()
onBind()
Clients are bound to service
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
Service shut down
Bounded service
Active Lifetime