Introduction

Managing contacts

This article takes on the subject of managing contacts on the user's phone. After reading it, you will be able to create a sample app that can find, add, delete and update contacts on his/her phone.

The author of this article assumes that the reader has basic programming knowledge and basic Android knowledge.

Overview

This article is divided into three parts: the first part shows how to find a contact in the phone's phonebook, the second one shows how to add new contacts and the last one shows how to delete contacts.

Application development

Contacts database

In Android the Contacts database is divided into three parts, with each storing different information:

- Data - Containing basic information like the phone number, name, or custom information designed by the developer.
- RawContacts - Containing information about a single account incorporating many Data fields. For example: John Smith, 555-666-222 and John Smith, 666-222-555 are two separate RawContacts.
- Contacts - Containing information from many sources about a single entry in the database formed from one or many RawContacts, for example the two RawContacts above, form a single contact record.

The Contacts database only contains the contacts that are visible on the phone (i.e. have not been deleted). Even if the contact has been deleted, its information is still stored in the Data table and in the RawContacts table.

You can see the Contacts database of the emulator, by using an SQLite Browser. The database is located in the data/data/com.android.providers.contacts/databases.

More details about this can be found at -
http://developer.android.com/reference/android/provider/ContactsContract.html .

Preparations

The first thing needed when creating an app that makes use of Contact data, is a custom contact class. The DCContact class is just a class containing fields for all the contact's data, e.g. getters and setters. The sample app implements only a few fields – the name, the last name and the phone number. Many more can be added like e-mail address, home address etc. Adding these additional fields is done in a similar way.

Finding contacts

The first thing that needs to be done is accessing the phone database and searching for contacts already there. In order to do this the ContactContract and ContentResolver are used. The ContentResolver can access all content stored on the phone, while ContactContract provides it with the necessary information to do a successful check. The Resolver performs a search using the URI supplied by the ContactContract and returns the list of all phone contacts on a Cursor. It is then just a matter of navigating the Cursor and comparing the data to the contact data desired.

```
private boolean checkIfExists(final DCContact pContact) {
         Uri conUri = Uri.withAppendedPath(ContactsContract.PhoneLookup.CONTENT_FILTER_URI,
                           Uri.encode(pContact.getPhoneNumber()));
         Cursor cur = getContentResolver().query(conUri, null, null, null, null);

         while (cur.moveToNext()) {
                  String displayName =
cur.getString(cur.getColumnIndex(ContactsContract.PhoneLookup.DISPLAY_NAME)).trim();

                  if (displayName.equals(pContact.getFullName())) {
                           return true;
                  }
         }
         return false;
}
```

The code to search for a contact

Adding new contacts

Searching for existing contacts is important, because a new contact must be checked for duplicates before being added. If this check isn’t performed, a duplicate contact may be added with redundant data.

The ContentResolver is used to add new contact data and read from the phonebook. Contact data is stored in a List of ContentProviderOperation. This List gathers all the values and fields of the contact like name, address, phone etc.

```
ArrayList<ContentProviderOperation> ops = new ArrayList<ContentProviderOperation>();
// Adding the contact info to the current account
ops.add(ContentProviderOperation.newInsert(RawContacts.CONTENT_URI)
         .withValue(RawContacts.ACCOUNT_TYPE, null)
         .withValue(RawContacts.ACCOUNT_NAME, null)
         .build());

// Adding the phone number
ops.add(ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
         .withValueBackReference(Data.RAW_CONTACT_ID, 0)
         .withValue(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE)
         .withValue(Phone.NUMBER, pContact.getPhoneNumber())
         .build());

// Adding the name
ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
         .withValueBackReference(Data.RAW_CONTACT_ID, 0)
         .withValue(Data.MIMETYPE, StructuredName.CONTENT_ITEM_TYPE)
         .withValue(StructuredName.DISPLAY_NAME, pContact.getFullName())
         .build());
```

Adding contact data to the List before applying it to the phonebook

The populated contacts list needs to be applied to the correct place (i.e. ContactsContract.AUTHORITY).

```
getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
```

Committing the contact data to the phonebook

A contact has been successfully created, with a name and phone number. Other data types that can be inserted can be found in the ContactsContract.CommonDataKinds class. More than one value can be assigned – for example adding more than one phone number can be achieved by adding another Phone.NUMBER position with the TYPE_CUSTOM type and a LABEL value to show that it is custom.

For example:

```
ops.add(ContentProviderOperation
        .newInsert(ContactsContract.Data.CONTENT_URI)
        .withValueBackReference(Data.RAW_CONTACT_ID, 0)
        .withValue(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE)
        .withValue(Phone.NUMBER, pContact.getSecondPhoneNumber())
        .withValue(Phone.TYPE, Phone.TYPE_CUSTOM)
        .withValue(Phone.LABEL, SECOND_NUMBER_LABEL)
.build());
```

Adding a custom field

Deleting contacts

The final operation with contacts is deleting them. Again, the ContentResolver is the easiest way to do this. Just like when finding contacts, the ContentResolver sends a query to the phonebook address and receives the list of contacts. Now we can either delete them all or locate the unwanted record and delete it. This will delete the contact from the phone, but the contacts that have been synchronized are safe, since they are stored on the Gmail account.

```
private void deleteContact(DCContact pContact) {
Cursor cur = getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, null, null,
null, null);
        if (cur.moveToFirst()) {
                Uri uri;
                do {
                uri = Uri.withAppendedPath(ContactsContract.Contacts.CONTENT_LOOKUP_URI,
cur.getString(cur.getColumnIndex(ContactsContract.Contacts.LOOKUP_KEY)));

                String displayName =
cur.getString(cur.getColumnIndex(ContactsContract.PhoneLookup.DISPLAY_NAME)).trim();

                if (displayName.equals(pContact.getFullName().trim())) {
```

```
                            getContentResolver().delete(uri, null, null);
                            }
                  } while (cur.moveToNext());
         }
}
```

Deleting a contact

To delete all of the contacts, the user doesn’t need to perform the check.

Updating contacts

The contacts located in the database can also be updated with new contact information. This is done in a very similar way to adding contacts, via the ContentProvider, only instead of using the newInsert function, use the newUpdate function.

This is done like in any database query - first find the appropriate field and insert a new value for it. Since the operation requires maneuvering between several tables, the first thing to do is finding the Raw_Contact_ID of the RawContact to be edited. In the sample, the search is based on the name.

```
Cursor dataCursor = getContentResolver().query(ContactsContract.Data.CONTENT_URI,
new String[] {
ContactsContract.Data.DISPLAY_NAME, ContactsContract.Data.RAW_CONTACT_ID },
ContactsContract.Data.DISPLAY_NAME + "='" + pContact.getName() + "'", null, null);

         if (dataCursor.moveToFirst()) {
                  rawContactID =
dataCursor.getString(dataCursor.getColumnIndex(Data.RAW_CONTACT_ID));
         }

         else { // There is no data returned from the Database
                  return false;
         }
```

The query method is used here to return the DISPLAY_NAME and RAW_CONTACT_ID columns (second parameter – projection) from the Data database (first parameter – database URI) where the name of the contact is the one that we are looking for (third parameter – selection).
The next step is updating the contact with the given RAW_CONTACT_ID, where the data is of number type.

```
ops.add(ContentProviderOperation
         .newUpdate(ContactsContract.Data.CONTENT_URI)
         .withSelection( ContactsContract.Data.RAW_CONTACT_ID + "=" + rawContactID + " AND "
+ ContactsContract.Data.MIMETYPE + "='"
+ ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE + "'", null)
.withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, pNewNumber).build());

getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
```

The above will replace a given contact’s phone number with the new one provided. Any data can be changed in a similar way (changing the name has been implemented in the attached app).

Final touches

For ease of use, a spinner has been added, which shows the current state of contacts, and allows selecting them for updating or deleting.

To set the spinner, a data structure is needed to keep all the contacts, a SparseArray seems to be an obvious choice, since it works like a HashMap but uses Integers as keys.

The query to get a contacts name, number and raw ID looks like this:

```
final Cursor cur = getContentResolver().query(
         ContactsContract.Data.CONTENT_URI,
         new String[] {ContactsContract.Data.DISPLAY_NAME,
ContactsContract.Data.RAW_CONTACT_ID,
         ContactsContract.Data.MIMETYPE, ContactsContract.Data.DATA1 }, null, null, null);
```

In the Data table the name is under the DISPLAY_NAME column, but the number is only in the DATA1 column, so we need to check that the data’s mimetype is of phone number type, and then we can get both the name and number.

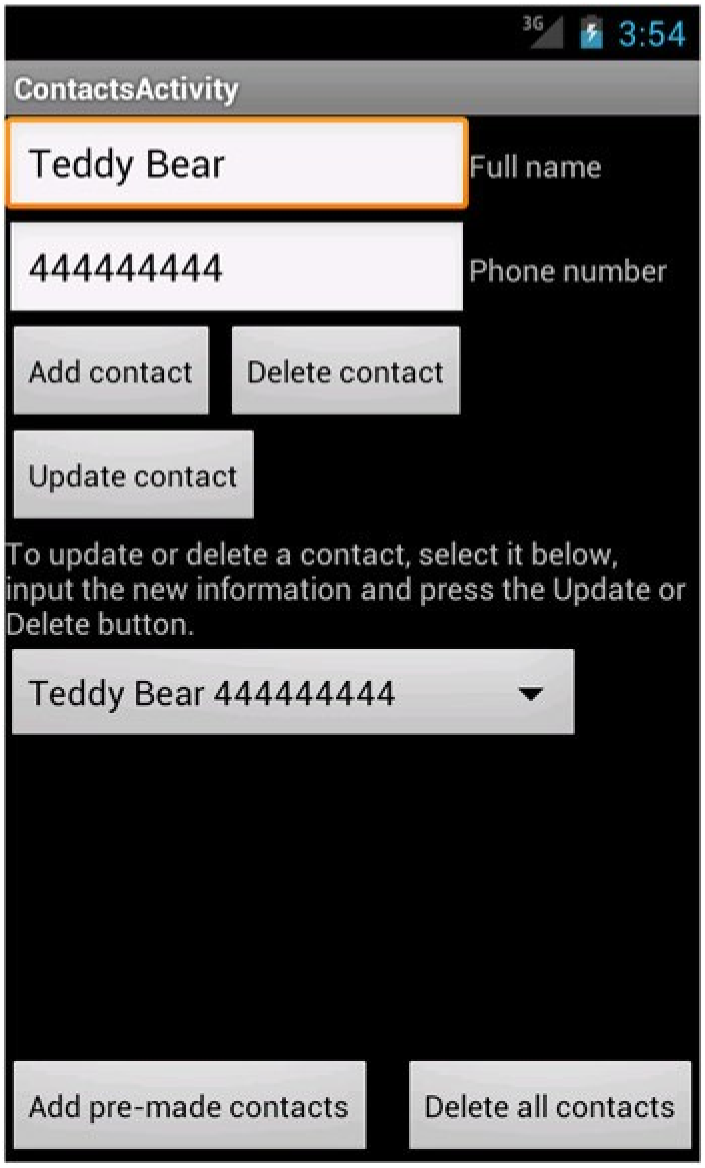
```
if (cur.getString(cur.getColumnIndex(Data.MIMETYPE)).equals(

ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)) {
         contactNumber = cur.getString(cur.getColumnIndex(Data.DATA1));

         if (contactName.contains(whiteSpace)) {
                  contact = new DCContact(contactName.split(whiteSpace)[0],
contactName.split(whiteSpace)[1],
                  contactNumber, rawContactID);
         } else {
                  contact = new DCContact(contactName, whiteSpace, contactNumber,
rawContactID);
         }
         mContactSparseArray.put(rawContactID, contact);
}
```

An adapter is needed to communicate between the spinner and the data structure. We have chosen an ArrayAdapter for this task.

```
mSparseArrayToSpinnerAdapter = new ArrayAdapter<DCContact>(this,
android.R.layout.simple_spinner_item, sparseArrayToArray());
```

The data we want the user to see in the spinner needs to be put in an Array, and that’s what the sparseArrayToArray() method is for.

After adding a basic user interface as seen in the picture below, an application that can add and delete contacts from the phone is complete. The code for this application has been attached.



To update or delete a contact, select it below, input the new information and press the Update or Delete button.

To learn more about contacts please visit - http://developer.android.com/reference/android/provider/ContactsContract.html

ref:http://developer.samsung.com/android/technical-docs/Managing-contacts