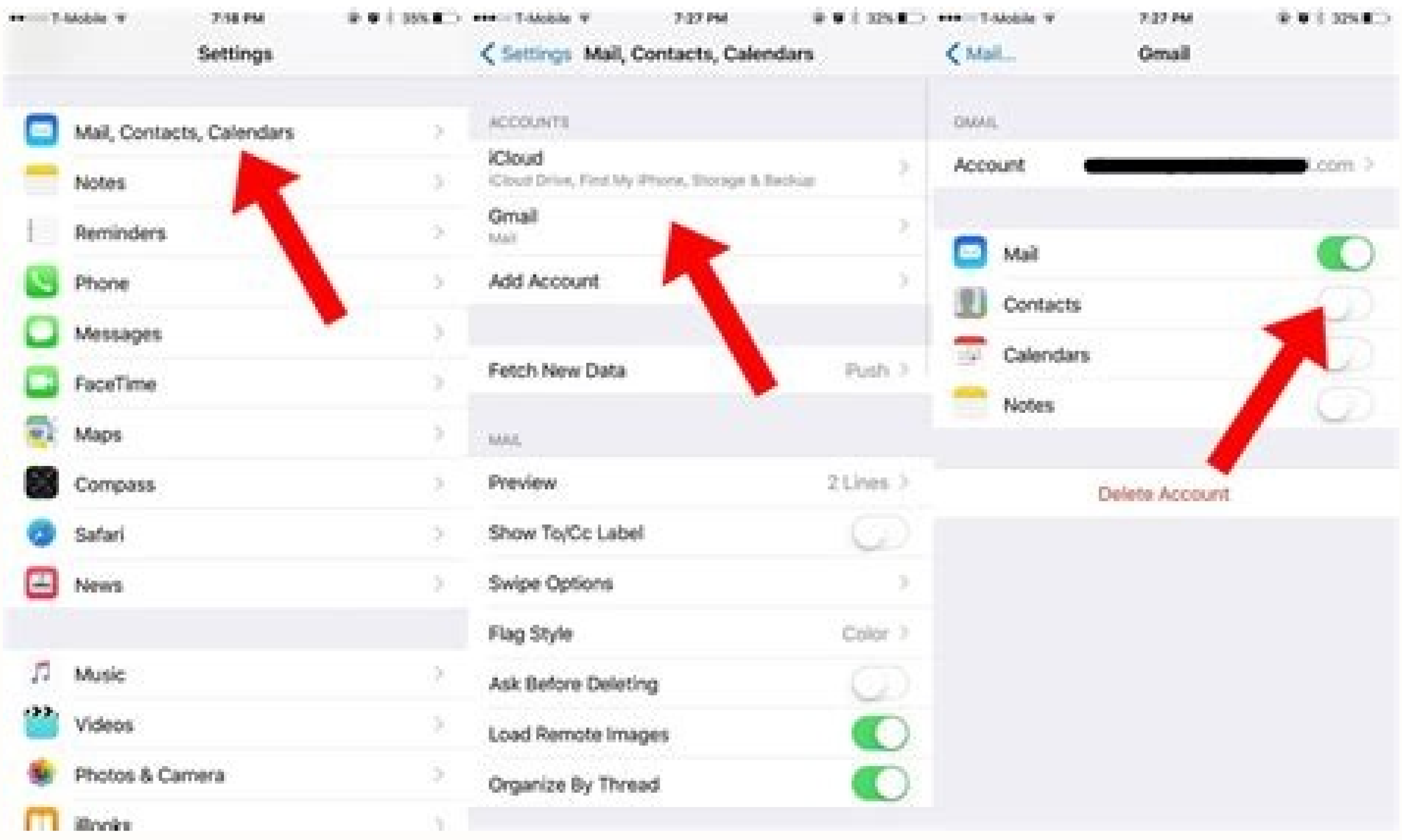
How do i transfer contacts from one android phone to another android phone

Continue


12% 6:33 PM
Import/Export Contacts
COPY CONTACTS FROM
SUN
SUN 01
Phone contact
SD card
Next


Layers
Layer Properties
Blending options
Opacity
Blend Mode
Normal
Add clipped adjustment
Effects
Smart filters



Contacts Provider is a powerful and flexible Android component that maintains a central repository of people data on your device. The contact provider is the source of data you see in the Contacts app on your device, and you can also access its data in your own app and transfer data between your device and online services. The provider covers a wide range of data sources and tries to manage as much data as possible for each individual, resulting in a comprehensive organization. For this reason, the Provider API includes a rich set of contract classes and interfaces that facilitate both data retrieval and manipulation. This guide provides the following information: Basic Vendor Structure. How to recover data from provider. How to edit supplier data. How to write a sync adapter to sync data from a server to a contact provider. This guide assumes that you are familiar with the basics of Android content providers. To learn more about content providers on Android, see the Content Providers Basic Guide. The contact provider is part of the Android content provider. It stores three types of personal data, each of which corresponds to a table offered by the provider, as shown in Figure 1: Figure 1. Contact provider table structure. These three tables are commonly referred to by the names of their contract classes. The classes define constants for content URIs, column names and column values used by tables: ContactsContract.Contacts table rows representing different people based on an aggregation of raw contact rows. ContactsContract.RawContacts table Rows containing summary information about a person specific to the user account and user type. ContactsContract.Data table Rows of raw contact data, e.g. B. E-mail addresses or telephone numbers. The other tables represented by the contract classes in ContactsContract are auxiliary tables that the contact provider uses to manage its operations or to process certain onesin your device contacts or phone apps. A raw contact represents a person's data from one account type and account name. Because the contacts provider allows more than one online service as a data source for a person, the contacts provider allows multiple raw contacts for the same person. Multiple raw contacts also allow the user to combine information about a person from more than one account of the same type. Most of the data for the original contact is not stored in the ContactsContract.RawContacts table. Instead, it is stored in one or more rows in the ContactsContract.Data table. Each data row has a Data.RAW_CONTACT_ID column containing the RawContacts._ID value of the ContactsContract.RawContacts parent row. The important columns in the ContactsContract.RawContacts table are listed in Table 1. Read the notes that follow the table: Table 1. Important raw contact columns. Column Name Usage Notes ACCOUNT_NAME Account name for the account type that is the source of this raw contact. For example, the Google account name is one of the device owner's Gmail addresses. See the next ACCOUNT_TYPE entry for more information. The format of this name is specific to the account type. It doesn't have to be an email address. ACCOUNT_TYPE The type of account that is the source of this original contact. For example, the Google account type is com.google. Always tag your account type with a domain ID for a domain you own or control. This ensures that your account type is unique. The account type that offers contact information usually has an associated sync adapter that syncs with the contact data provider. DELETED The "deleted" flag for an unprocessed contact. This flag allows the contact provider to store the row internally until the sync adapters can delete the row from their servers, and then permanently remove the row from storage. Notes In the following text you will find important notes aboutTable: Raw contact name is not stored in its string in ContactsContract.RawContacts. Instead, it is stored in the ContactsContract.Data table in the ContactsContract.CommonDataKinds.StructuredName row. A raw contact has only one row of this type in the ContactsContract.Data table. Note. To use your account information in a raw contact string, you must first register it with your account administrator. To do this, prompt the user to add the account type and account name to the list of accounts. If you don't, the contact provider will automatically delete your raw contact line. For example, if you want your application to store contact information for your web service with the domain com.example.dataservice and the user account for your service is becky.sharp@dataservice.example.com, the user must first add the account type (com .example .dataservice) and account name (becky.smart@dataservice.example.com) before your app can add raw contact strings. You can explain this requirement to the user in the documentation, or you can ask the user to specify the type and Add names, or both. Account types and account names are described in more detail in the next section. To understand how Raw Contacts works, consider the user "Emily Dickinson" who has the following three accounts defined on his device: emily.dickinson@gmail.com emilyd@gmail.com Twitter account "belle_of_amherst" This user has contacts sync enabled for all three of these accounts in Account Settings. Suppose Emily Dickinson opens a browser window, logs into Gmail as emily.dickinson@gmail.com, opens Contacts, and adds "Thomas Higginson". She later logs into Gmail as emilyd@gmail.com and sends an email to "Thomas Higginson" which automatically adds him to her contact list. He also follows "colonel_tom" (Thomas Higginson's Twitter ID) on Twitter. As a result of this work, the contact provider creates three raw contacts: A raw contact for "Thomas Higginson" associated with emily.dickinson@gmail.com. user accountit's google. Second raw contact 'Thomas Higginson' associated with emilyd@gmail.com. The user account type is also Google. There is a second unverified contact, although the name is identical to the previous one, because the person was added to a different user account. Third raw contact "Thomas Higginson" related to "belle_of_amherst". User account type is Twitter. Data As mentioned above, raw contact data is stored in the ContactsContract.Data string associated with the _ID value of the raw contact. Thus, one raw contact has multiple instances of the same type of data, such as email addresses or phone numbers. For example, if "Thomas Higginson" emilyd@gmail.com (the raw contact string of Thomas Higginson associated with the Google account emilyd@gmail.com) has a home email address of thigg@gmail.com and a work email address of thomas. higginson Provider Contacts @gmail.com stores two rows of email addresses and connects both to a raw contact. Note that this one table contains different types of data. The display name, phone number, email, mailing address, photo, and website detail rows are located in the ContactsContract.Data table. To make it easier to manage, the ContactsContract.Data table has columns with descriptive names and additional columns with generic names. The content of a column with a descriptive name has the same meaning regardless of the data type of the row, while the content of a column with a generic name has a different meaning depending on the data type. Descriptive Column Names Some examples of descriptive column names: RAW_CONTACT_ID The raw _Contact ID value of the column for this data. MIMETYPE The type of data stored in this string, expressed as a custom MIME type. The Contacts provider uses the MIME types defined in the ContactsContract.CommonDataKinds subclasses. These MIME types are open source and can be used by any application or synchronization adapter that works with the contact provider. IS_PRIMARYBecause a data row of this type can appear more than once in a raw contact, the IS_PRIMARY column identifies the data row that contains this type of primary data. For example, if a user long-presses a contact's phone number and selects Set Default, the IS_PRIMARY column of the ContactsContract.Data row that contains that number is set to a non-zero value. Generic Column Names There are a total of 15 generic columns named DATA1–DATA15 and four additional generic columns SYNC1–SYNC4 that should only be used by sync adapters. Generic column name constants always work regardless of the type of data contained in the row. Column DATA1 is indexed. The contact data provider always uses this column for data that the provider considers to be the most frequently used query target. For example, in the email address row, this column contains the actual email address. By convention, the DATA15 column is used to store binary large object (BLOB) data, such as B. Photo thumbnails, proprietary. Type-Specific Column Names To facilitate working with columns for a given row type, the Contacts provider also provides type-specific column name constants, which are defined in the ContactsContract.CommonDataKinds subclasses. Constants simply give the same column name another constant name, making it easier to access data in a particular row type. For example, the ContactsContract.CommonDataKinds.Email class defines type-specific column name constants for the row ContactsContract.Data with MIME type Email.CONTENT_ITEM_TYPE. The class contains the constant ADDRESS for the e-mail address column. The actual ADDRESS value is "data1", which is a generic column name. Attention! Do not add your custom data to the ContactsContract.Data table using a row that contains one of the provider's predefined MIME types. Otherwise, you may lose data or cause disruption to your service provider. For example, don't add lines with a MIME typewhich contains a username instead of an email address in the DATE1 column. If you use your own MIME type for the string, you can define your own column names for the specific type and use the columns however you like. Figure 2 shows how descriptive columns and data columns appear in the ContactsContract.Data row and how column names of certain types overlap common column names. Rice. 2. Enter dependent column names and common column names. Column name classes for specific types Table 2 lists the most commonly used column name classes for specific types: Table 2. Column name classes for specific types The contact provider combines raw contact strings from all account types and account names to form a contact. This allows all data that the user has collected about the person to be easily viewed and changed. The contact provider controls the creation of new contact strings and the merging of raw contacts into an existing contact string. Neither apps nor sync adapters can add contacts, and some columns in the contact row are read-only. Note. If you try to add a contact to the contact provider using the insert() function, you will get an UnsupportedOperationException. If you try to update a read-only column, the update will be ignored. The contact provider creates a new contact in response to adding a new raw contact that doesn't match any of the existing contacts. The service provider also does this if existing raw contact data is changed in such a way that it no longer matches the contact to which it was previously assigned. When the sync app or adapter creates a new raw contact that doesn't match an existing contact, the new raw contact is merged into the existing contact. The contact provider joins the contact row with the raw contact rows to the contact row's _ID column in the contact table. The CONTACT_ID column of the ContactsContract.RawContacts raw contact table contains the _ID values for the contact row associated with each row of raw contacts.The ContactsContract.Contacts table also has a LOOKUP_KEY column, which is a "permanent" reference to the Contacts row. Because the contact provider maintains contacts automatically, it can change the _ID value of a contact row in response to an aggregation or synchronization. Even if this happens, CONTENT_LOOKUP_URI with contact LOOKUP_KEY will still point to the contact row, so you can use LOOKUP_KEY to store links to "favorite" contacts, etc. This column has its own format, unrelated to the format of the _ID column. Figure 3 shows how the three main tables are related. Figure 3 Contact table, raw contacts and detailed relationships. Note. If you publish your app to the Google Play Store, or if your app is installed on Android 10 (API level 29) or later, be aware that the limited set of contact fields and methods is deprecated. Under the following conditions, the system periodically removes all values stored in these data fields: The APIs used to set the above data fields are also deprecated: In addition, the following fields no longer return frequent contacts. Note that some of these fields only affect contact ratings if the contacts are part of a certain type of metric. If your applications access or update these fields or APIs, use other methods. For example, you may fulfill certain use cases by using private content providers or other data stored in your application or back-end systems. You can manually clear these data fields and verify that these changes do not affect the functionality of your application. To do this, run the following ADB command on a device running Android 4.1 (API level 16) or higher: delete adb shell content \ --uri content://com.android.contacts/contacts/delete_usage Sync data for adapters Users enter contact information directly on the device, but data to the contact provider also flows from Internet services using synchronization adapters that automate the transfer of data between the device and services. Synchronizationthey run in the background under system control and call ContentResolver methods to manipulate the data. In Android, the web service that the sync adapter communicates with is identified by the account type. Each sync adapter works with one account type, but can support multiple account names for that type. Account types and account names are briefly described in the Raw Contact Sources chapter. The following definitions describe in more detail how the account type and name are associated with adapters and synchronization services. Account type Specifies the service in which the user stores data. In most cases, the user must be authenticated with the service. For example, Google Contacts is an account type identified by the code google.com. This value corresponds to the account type used by the AccountManager. Account name Identifies a specific account or credentials for an account type. Google Contacts accounts are similar to Google accounts with an email address as the account name. Other services may use a single-word username or a numeric identifier. Account types do not have to be unique. You can set up multiple Google Contacts accounts and upload your details to a contact provider; this can happen if the user has one set of personal contacts for the personal account name and another set for work. Account names are usually unique. Together, they define a specific data flow between the contact provider and the external service. If you want to pass overhead to the contact provider, you need to write your own sync adapter. This is described in more detail in the Contact Provider Sync Adapters section. On fig. Figure 4 shows how the contact provider fits into the flow of people data. In the Sync Adapters field, each adapter is labeled with an account type. Figure 4. Contact provider data flow. Required Permissions Applications that want to access the contacts provider must request the following permissions: Read access to one or more tables.specified in the AndroidManifest.xml file with the element as . Write access to one or more WRITE_CONTACTS tables listed in AndroidManifest.xml with the element as . These permissions do not apply to user profile data. The user profile and required permissions are described in the next section User profile. Please note that user contact information is personal and confidential. Users are concerned about their privacy, so they don't want apps to collect data about them or their contacts. If it's not clear why you need permission to access their contact information, they may give the app a low rating or simply refuse to install it. User Profile The ContactsContract.Contacts table contains one row containing the user profile data for the device. This data describes the user of the device, not one of the user's contacts. The profile's contact queue is linked to the raw contact queue for each system using the profile. Each raw contact row in a profile can contain multiple rows of data. User profile access constants are available in the ContactsContract.Profile class. Access to the user profile requires special permissions. In addition to the READ_CONTACTS and WRITE_CONTACTS permissions required for read and write, access to the user profile requires read and write permissions android.Manifest.permission#READ_PROFILE and android.Manifest.permission#WRITE_PROFILE. Remember that the user profile should be treated as confidential. The android.Manifest.permission#READ_PROFILE permission allows access to the device user's private data. In the application description, be sure to tell the user why you need access rights to the user profile. To get the contact string containing the user's profile, call ContentResolver.query(). TuneURI to CONTENT_URI and don't specify any selection criteria. You can also use this content URI as the base URI to retrieve raw contact or profile data. For example, this snippet loads data for a profile: // Specifies the columns to load for a user profile) // Loads a profile from a contact provider profile profileCursor = contentResolver.query(ContactsContract.Profile.CONTENT_URI, project, null, null , null) // Specifies columns to load for user profile project = new String[] { Profile. _ID , profile.DISPLAY_NAME_PRIMARY, profile.LOOKUP_KEY , profile.PHOTO_THUMBNAIL_URI }; // Get profile from contact provider profile profileCursor = getContentResolver().query(Profile.CONTENT_URI, project, null, null, null); Note. If you are retrieving multiple contact rows and want to determine if one of them is a user profile, check the IS_USER_PROFILE row column. This column is set to "1" if the contact is a user profile. The contact provider maintains data that keeps track of the state of the saved contact information. This repository metadata is stored in various places, including rows in the Raw Contacts, Data, and Contacts tables, the ContactsContract.Settings table, and the ContactsContract.SyncState table. The following table shows the impact of each of these metadata: Table 3. Metadata in the contact providers table Column Values Value ContactsContract.RawContacts DIRTY "0" - no change since last sync. Displays raw contacts that have been modified on the device and need to be synced to the server. This value is automatically set by the contacts provider when Android apps update the string. Synchronization adapters that modify raw contact or data tables should always append the string CALLER_IS_SYNCADAPTER to the content URI.This prevents the provider from marking lines as dirty. Otherwise, changes to the sync adapter appear as local changes and are sent to the server, even if the change comes from the server. "1" - Changed since last sync, need to resync with server. ContactsContract.RawContacts VERSION The version number of this string. The contact provider automatically increments this value each time a row or its data changes. ContactsContract.Data DATA_VERSION The version number of this string. The contact data provider automatically increments this value each time the data row changes. ContactsContract.RawContacts SOURCE_ID A string value that uniquely identifies this raw contact for the account it was created under. When the sync adapter creates a new raw contact, this column should contain the unique server ID for the raw contact. When the Android app creates a new raw contact, this column should be left blank. This signals the sync adapter to create a new raw contact on the server and obtain a value for SOURCE_ID. In particular, the Source ID must be unique for each account type and stable during synchronization: Unique: Each raw contact for an account must have its own Source ID. If you don't, you'll cause problems in the Contacts app. Note that two raw contacts for the same account type may have the same Source ID. For example, the raw contact "Thomas Higginson" for the account "emily.dickinson@gmail.com" might have the same Origin ID as the raw contact "Thomas Higginson" for the account "emilyd@gmail.com". Stable: Source IDs are a permanent part of the online service data for raw contacts. For example, if the user clears the contact store in the app settings and re-syncs, the recovered raw contacts should have the same original IDs as before. If you don't, the shortcuts will stop working. ContactsContract.Groups GROUP_VISIBLE "0" - Contacts in this groupThey are not visible in Android app user interfaces. This column is for compatibility with servers that allow the user to hide contacts in specific groups. "1" - Contacts in this group can be visible in the application user interfaces. ContactsContract.Settings UNGROUPED_VISIBLE "0" - For this account and account type, ungrouped contacts are invisible in Android application UIs. Contacts are invisible by default if none of their raw contacts belong to a group (a raw contact's group membership is indicated by one or more rows in the ContactsContract.CommonDataKinds.GroupMembership table in the ContactsContract.Data table). You can make contacts without groups visible by setting this flag in the ContactsContract.Settings table row for Account Type and Account. One use of this flag is to show contacts from servers that don't use groups. "1" - For this account and account type, non-group contacts are visible in the application user interfaces. ContactsContract.SyncState (all) Use this table to store metadata for your sync adapter. You can use this table to permanently store sync status and other sync-related data on your device. Accessing the contact provider This section describes instructions for accessing data from the contact provider, with a focus on: Querying entities. dose adjustment. Load and edit on purpose. data reliability. Making customizations through the sync adapter is also described in the Contact Provider Sync Adapter section. Querying Entities Because contact provider tables are organized in a hierarchy, it is often useful to retrieve a row and all of its "child" rows. For example, to display all information about a person, you might want to retrieve all ContactsContract.RawContacts rows for a single ContactsContract.Contacts row, or all ContactsContract.CommonDataKinds.Email rows for a single ContactsContract.RawContacts row. To facilitate this, the contact provider offers an entitywhich act like database connections between tables. An entity is like a table consisting of selected columns from a parent table and its child table. When you query an object, you provide predictions and search criteria based on the columns available for the object. The result is a cursor that contains one row for each row in the loaded child table. For example, if you query ContactsContract.Contacts.Entity for the contact name and all rows in ContactsContract.CommonDataKinds.Email for all raw contacts with that name, you will get a cursor with one row for each row in ContactsContract.CommonDataKinds.Email. Entities simplify queries. With an entity, you can get all the contact details for a contact or raw contact at once, instead of first querying the ID in the parent table and then querying that ID in the child table. In addition, the contact provider handles the entity request in a single transaction, ensuring that the retrieved data is internally consistent. Note. An entity usually does not contain all the columns of the parent and child tables. If you try to work with a column name that is not in the persistent list of entity column names, an exception will be thrown. This snippet shows how to get all raw contact rows. A snippet is part of a larger application that has two activities: "master" and "detail". The main action displays a list of contact strings; when the user selects one, the activity sends its id to the detailed activity. The Expand action uses the ContactsContract.Contacts.Entity to display all rows of data from all raw contacts associated with the selected contact. This snippet is taken from the "details" action: ... /* * Adds the object's path to the URI. In the case of a contact provider, the intended URI is * content://com.google.contacts/#/entity (where # is the value of the identifier). */ contactUri = Uri.withAppendedPath(contactUri,) // Initializes the loader identified by LOADER_ID. loaderManager.initLoader(LOADER_ID, // Loader ID to initialize from null, // Loader arguments (none in this case) this // Operation context) // Creates a new cursor adapter to include in the list view cursorAdapter = SimpleCursorAdapter(this, // context actions R.layout.detail_list_item, // view item with detail widgets mCursor, // base cursor fromColumns, // columns in cursor providing data for views, // views in view item showing data 0) // Flags // Sets the ListView backup adapter. rawContactList.adapter = cursorAdapter ... override fun onCreateLoader(id: Int, args: Bundle?): Loader { /* * Specifies the columns to load. * RAW_CONTACT_ID is included to identify the raw contact associated with the row of data. * DATA1 contains the first column in the data row (usually the most important). * MIMETYPE indicates the type of data in the data string. */ project val: Array = arrayOf(ContactsContract.Contacts.Entity.RAW_CONTACT_ID, ContactsContract.Contacts.Entity.DATA1, ContactsContract.Contacts.Entity.MIMETYPE) /* * Sorts the resulting cursor by the raw contact ID, to keep all rows of data for a single raw* contact added together. */ val sortOrder = "${ContactsContract.Contacts.Entity.RAW_CONTACT_ID} ASC" /* * Returns a new CursorLoader. The arguments * are the same as the ContentResolver.query() argument, except for the Context argument, * which specifies the location of the ContentResolver to use. */ return CursorLoader(applicationContext, // action context ContactUri, // entity content URI for single contact projection, // columns to get null, // get all raw contacts and their data rows. null, // sortOrder // sort by to raw contactid) } ... /* * Adds the object's path to the URI. For the contact provider *, the expected URI is content://com.google.contacts/#/entity.identifier value). */ contactUri = Uri.withAppendedPath(contactUri, ContactsContract.Contacts.Entity.CONTENT_DIRECTORY); // Initializes the loader specified by LOADER_ID. getLoaderManager().initLoader(LOADER_ID, // loader ID for initialization null, // loader arguments (none in this case) this); // Operation context // Creates a new cursor adapter to add to the list view cursorAdapter = new SimpleCursorAdapter(this, // operation context R.layout.detail_list_item, // view element containing detail widgets mCursor, // fallback cursor fromColumns, // cursor columns that pass views // views of view elements that display data 0); // flags // Sets the backing adapter of the ListView. rawContactList.setAdapter(cursorAdapter); ... @Override public Loader onCreateLoader(int id, Bundle args) { /* * Sets the columns to load. * RAW_CONTACT_ID is included to identify the raw contact associated with the data row. * DATA1 contains the first column in the data row (usually the most important). * MIMETYPE specifies the type of data in the data string. */ String [] projection = { ContactsContract.Contacts.Entity.RAW_CONTACT_ID, ContactsContract.Contacts.Entity.DATA1, ContactsContract.Contacts.Entity.MIMETYPE }; /* * Sorts the resulting cursor by the raw contact id so that all data rows of one * raw contact are gathered together. */ String sortOrder = ContactsContract.Contacts.Entity.RAW_CONTACT_ID + "ASC"; /* * Returns a new CursorLoader. The arguments are the same as the * ContentResolver.query() argument, except that the Context argument specifies the location of the * ContentResolver to use. */ return a new CursorLoader(getApplicationContext(), // ContactUri action context, // Object content URI for one contact projection, // Columns to wrap null values, // Get all raw contacts and their data rows. , // Sort order) ; // Sort by raw contact id. } After loadingfinishes, the LoaderManager calls the onLoadFinished() callback. One of the input arguments to this method is a cursor containing the results of the query. You can view or continue working with this cursor data in your application. Batch Change Whenever possible, you should insert, update, and delete data in a contact provider in a "batch" mode, creating an array list of ContentProviderOperation objects and calling applyBatch(). Because the contacts provider does everything applyBatch() does in a single transaction, your changes will never leave the contacts repository in an inconsistent state. Batch modification also facilitates the simultaneous insertion of a raw contact and its details. Note. To edit a single raw contact, consider sending the intent to the device's contacts app instead of handling the change in the native app. This is described in more detail in the Extraction and Intended Modification section. Break Even Points A batch modification containing a large number of operations can block other processes, resulting in an overall poor user experience. In order to organize all the changes you want to make into as few separate lists as possible, while avoiding blocking the system, you need to set floating points for one or more operations. The yield point is a ContentProviderOperation object with isYieldAllowed() set to True. When the contact provider reaches the breakeven point, it stops its work to allow other processes to run and closes the current transaction. When the provider restarts, it moves to the next step in the ArrayList and starts a new transaction. Yield points generate more than one transaction per AppBatch() call. Because of this, a series of related rows must establish a break-even point for the last operation. For example, you need to set a break-even point for the last step in a set that adds raw contact rows and related data.or the last action in a series of rows related to a single contact. Yield points are also a unit of atomic activity. All accesses between two yield points will pass or fail as a single entity. If you don't set any yield points, the smallest atomic operation is the entire set of operations. By using yield points, you prevent features from degrading system performance while ensuring that a subset of features is atomic. Undoing changes After inserting the new raw contact row and its associated data rows as a set of ContentProviderOperation objects, bind the data rows to the raw contact row by inserting the raw contact_ID value as the RAW_CONTACT_ID value. However, this value is not available when creating a ContentProviderOperation for a data row because the ContentProviderOperation has not yet been applied to the raw contact row. As a workaround, the ContentProviderOperation.Builder class has a method withValueBackReference() . You can use this method to insert or change the column with the result of the previous operation. withValueBackReference() takes two arguments: key The key of the key-value pair. The value of this argument must be the name of a column in the table you are modifying. previousResult The 0-based index of the values in the array of ContentProviderResult objects from applicationBatch(). When using batch operations, the result of each operation is stored in an intermediate result array. The previousResult value is the index of one of these results retrieved and stored with the key value. This allows you to insert a new raw contact record and retrieve its _ID value, then create a "back reference" to the value by adding a row to ContactsContract.Data. The entire result array is created the first time applicationBatch() is called and is the same size as the specified ArrayList of the ContentProviderOperation object. However, all the elementsarray is set to null and if you try to refer to the result of an operation that has not yet been applied, withValueBackReference() will throw an exception. The following snippets show how to batch insert a new raw contact and data. They contain code that sets the yield point and uses backreference. The first snippet retrieves contact information from the UI. At this point, the user has already selected the account to which he wants to add the new raw contact. // Create a contact entry from the current UI values using the currently selected account. private fun createContactEntry() { /* * get value from UI */ val name = contactNameEditText.text.toString() val phone = contactPhoneEditText.text.toString() val email = contactEmailEditText.text.toString() val phoneType: String = contactPhoneTypes[mContactPhoneTypeSpinner.s selectedItemPosition] val emailType: String = contactEmailTypes[mContactEmailTypeSpinner.s selectedItemPosition] // Create a contact entry from the current UI values using the currently selected account. protected void createContactEntry() { /* * Get values from UI */ String name = contactNameEditText.getText().toString(); phone string = contactPhoneEditText.getText().toString(); Email String = contactEmailEditText.getText().toString(); int phoneType = contactPhoneTypes.get (contactPhoneTypeSpinner.getSelectedItemPosition()); int emailType = contactEmailTypes.get(contactEmailTypeSpinner.getSelectedItemPosition()); The following code snippet creates an operation to insert a raw contact row into the ContactsContract.RawContacts table: /* * Prepares a batch operation to insert a new raw contact and its details. Even if the *contact provider doesn't have any data for that person, you can't add a contact, *only raw contact. The contact provider then adds the contact automatically. */ // Creates a new array of ContentProviderOperation objects. val ops = arrayListOf() /* * Creates a new raw contactits account type (server type) and account name * (user account). Note that the display name is not stored in this row, but in the *StructuredName data row. No additional data is required. */ var op: ContentProviderOperation.Builder = ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI).withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, selectedAccount.name) .withValue(RawCcountsadd arra to op.build()) /* * Prepare batch operation to insert a new raw contact and its details. Even if the *contact provider doesn't have that person's details, you can't add the contact, *only raw contact. The contact provider then automatically adds the contact. */ // Creates a new array of ContentProviderOperation objects. ArrayList ops = new ArrayList(); /* * Creates a new raw contact with account type (server type) and account name * (user account). Note that the display name is not stored in this row, but in the *StructuredName data row. No additional data is required. */ ContentProviderOperation.Builder op = ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI).withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, selectedAccount.getType()); // Creates an operation and adds it to the ops array ops.add(op.build()); The code then creates data rows for the Display Name, Phone Number, and Email Address rows. Each action builder object uses sValueBackReference() to obtain the RAW_CONTACT_ID. The reference points back to the ContentProviderResult object from the first step, which adds the raw contact row and returns its new _ID value. As a result, each data row is automatically associated by its RAW_CONTACT_ID with the new ContactsContract.RawContacts row to which it belongs.The ContentProviderOperation.Builder object that adds the email string is marked with a YieldAllowed() function that sets the exit point: // Creates a display name for the new raw contact as a StructuredName data string. op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI) /* * withValueBackReference sets the value of the first argument to the * ContentProviderResult value indexed by the second argument. In this particular call, the * raw contact ID column of the StructuredName data row is set to the * value of the result returned by the first operation, which actually adds * the raw contact string. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Set the MIME type of the data string to the StructuredName data string for the user interface name. .withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, name) // Creates an operation and adds it to the operations array ops.add(op.build()) // Adds the specified phone number and type as a phone data string op = ContentProviderOperation. newInsert (ContactsContract.Data.CONTENT_URI) /* * Sets the value of the raw contact id column to the new raw contact id returned * from the first operation in the batch. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Set the data string MIME type to Phone (ContactsContract.CommonDataKinds.Phone.NUMBER, phone) .withValue(ContactsContract.CommonDataKinds.Phone.TYPE, phoneType) // Creates operation and adds it to the ops array ops.add(op.build()) // Inserts the specified email address and enters it as a phone data string op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI) / * * Sets the value of the original id column contactthe new raw contact id returned by * the first operation in the batch. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Set the MIME type of the data string to Email (ContactsContract.CommonDataKinds.Email.ADDRESS, email).withValue(ContactsContract.CommonDataKinds.Email.TYPE, emailType) /* * Shows starting point. At the end of this insert, the batch thread * prioritizes other threads. Use * after each group of operations involving a contact to avoid performance degradation. */ op.withYieldAllowed(true) // Creates an operation and adds it to the ops array ops.add(op.build()) // Creates a friendly name for the new raw contact as a StructuredName data string. op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI) /* * withValueBackReference sets the value of the first argument to the * ContentProviderResult value indexed by the second argument. In this particular call, the raw contact ID column of the StructuredName data row is *set to the value of the result returned by the first operation, which *actually adds the raw contact string. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Sets the MIME type of the data string to StructuredName in the UI. .withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, Name); // Create an operation and add it to the ops array ops.add(op.build()); // Inserts the specified phone number and type as a phone data string op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI) /* * Sets the value of the Raw Contact ID column to the new Raw Contact ID used by the first * operation is returnedBroadcast. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Sets the MIME type of the data string to phone .Phone.NUMBER, phone).withValue(ContactsContract.CommonDataKinds.Phone.TYPE, phoneType); // Build an operation and add it to the operations array ops.add(op.build()); // Inserts the given email address and writes it as a phone data string op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI) /* * Sets the value of the raw contact id column to the new raw contact id that * returned by the first operation on the stack. */ .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0) // Set the MIME type of the data string to email (ContactsContract.CommonDataKinds.Email.ADDRESS, email) .withValue(ContactsContract.CommonEmailT.TYPE); /* * Shows the yield curve. At the end of this insert, the batch operation thread * gives precedence to other threads. Use after each set of operations that affect * a pin to avoid performance degradation. */op.withYieldAllowed(true); // Build an operation and add it to the operations array ops.add(op.build()); The last snippet shows an applyBatch() call that inserts new raw contact and data rows. // Ask the contact provider to create a new contact Log.d(TAG, "Selected account: ${mSelectedAccount.name} (${mSelectedAccount.type})") Log.d(TAG, "Create contact: $name") / * * Applies an array of ContentProviderOperation objects in batch mode results * are discarded */ try { contentResolver.applyBatch(ContactsContract.AUTHORITY, ops) } catch (e: Exception) { // show warning val txt: String = getString(R .string.contactCreationFailure)txt, Toast.LENGTH_SHORT).show() // Log an exception Log.e(TAG, "An exception occurred while inserting a contact: $e") } } // Ask your service provider to create a new contact log. d(TAG, "Selected account: " + selectedAccount.getName() + " (" + selectedAccount.getType() + ")"); Log.d(TAG,"Create contact: " + name); /* * An array of ContentProviderOperation objects is used in the package. The results are * discarded. */ try { getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops); } catch (Exception e) { // Display notification context ctx = getApplicationContext(); CharSequence txt = getString(R.string.contactCreationFailure); int duration = Toast.LENGTH_SHORT; Toast toast = Toast.makeText(ctx, txt, duration); toast.show(); // Log exception Log.e(TAG, "Exception caught while inserting contact: " + e); } } Batch operations also allow you to implement optimistic concurrency control, a method of applying modification transactions without closing the underlying storage. To use this method, commit the transaction and then verify that no other changes were made at the same time. If you find inconsistent changes, roll back the transaction and try again. Optimistic concurrency control is useful for mobile devices where there is only one user at a time and concurrent access to data storage is rare. Since no locking is used, there is no need to waste time setting up locks or waiting for other transactions to release locks. To use optimistic concurrency control when updating a single row in ContactsContract.RawContacts, do the following: If a raw contact row is updated by another operation between loading the row and trying to modify it, the ContentProviderOperation "commit" operation will fail and the entire group of operations will be duplicated. You can then choose to retry the batch or take another action. This snippet shows how to create a "confirmation" for a ContentProviderOperation.query a single raw contact with CursorLoader: /* * The application uses CursorLoader to query the raw contact table. The system calls this method * when the download is complete. */ override fun onLoadFinished(loader: Loader, cursor: Cursor) { // Get raw _ID and VERSION values of the contact. RawContactID = cursor.getLong(cursor.getColumnIndex(BaseColumns._ID)) mVersion = cursor.getInt (cursor. getColumnIndex(SyncColumns.VERSION)) } ... // Set URI for confirmation operation val rawContactUri: Uri = ContentUris. withAppendedId(ContactsContract .RawContacts.CONTENT_URI, rawContactID) // Creates the creator of the val assertion action. assertOp : ContentProviderOperation.Builder = ContentProviderOperation .newAssertQuery(rawContactUri).apply { // Adds assertions to the assert operation: checks the version with Value(SyncColumns.VERSION, mVersion) // and the number of rows checked with /Counted with Exp. ArrayList containing ContentProviderOperation objects val ops = arrayListOf() ops.add(assertOp.build()) // This is where the remaining batch operations are added to the operation ... // Use . If the assertion fails, an exception is thrown try { val Results: Array = contentResolver.applyBatch(AUTHORITY, ops) } catch (e: OperationApplicationException) { // Here are the actions to take if the assertion fails } / * * The application uses CursorLoader to query the raw contact table. The system calls this method * when the download is complete. */ public void onLoadFinished(Loader loader, Cursor cursor) { // Fetches the raw values of the contact's _ID and VERSION. RawContactID = cursor.getLong(cursor.getColumnIndex(BaseColumns._ID)); mVersion = cursor.getInt(cursor.getColumnIndex(SyncColumns.VERSION)); } ... // Set the Uri for the assertion action Uri rawContactUri = ContentUris.withAppendedId(RawContacts.CONTENT_URI, rawContactID); // Creates an assertion action constructorasseOp = ContentProviderOperation.newAssertQuery(rawContactUri); // Add assertions to the assert operation: checks the version and number of rows checked assessOp.withValue(SyncColumns.VERSION, mVersion); claimOp.withExpectedCount(1); // Creates an ArrayList to store ContentProviderOperation objects ArrayList ops = new ArrayList; ops.add(assertOp.build()); // For other batch operations you would add "ops" here... // Uses batch. If the assertion fails, an exception is thrown try { ContentProviderResult[] results = getContentResolver().applyBatch(AUTHORITY, ops); } catch (OperationApplicationException e) { // Here are the actions you want to take if the validation operation fails. } You can access the contacts provider indirectly by sending an intent to the Contacts app on the device. The intent launches the Contacts user interface on the device where users can perform contacts-related tasks. Using this access method, users can: Select a contact from the list and return it to their application for further work. Edit information about an existing contact. Insert a new raw contact for any of your accounts. Delete a contact or contact details. If a user posts or updates data, you can first collect the data and send it as part of an intent. If you use intents to access the contact provider through the Contacts app on the device, you don't need to write your own UI or code to access the provider. You also don't need to ask your ISP for read or write permission. The Contacts app on your device can delegate read access to the contact to you, and since you're making changes to the provider through another app, you don't need write access. The general process of submitting an access intent to a provider is detailed in the Content Provider Fundamentals Guide under "Accessing Data Using Intents". The actions, MIME type, and data values used for the available jobs are listed in Table 4.other values you can use with putExtra() are listed in the ContactsContract.Intents.Insert reference documentation: Table 4. Contacts provider intents. Task Action MIME Data Type Notes Pick a contact from the ACTION_PICK list One of: Not Used Displays a list of raw contacts or a list of data from a raw contact depending on the content URI type you specified. A call to startActivityForResult() that returns the URI of the content of the selected row. The URI of the form is the URI of the table content appended to the LOOKUP_ID line. The Contacts app on the device delegates read and write permissions to this content URI at runtime. Please refer to your content provider's main information guide for more information. Insert new raw contact Insert.ACTION N/A RawContacts.CONTENT_TYPE, MIME type for the set of raw contacts. Displays the Add Contact screen of the Device Contacts application. Additional values that you add to the intent will appear. When sent with startActivityForResult(), the content URI of the newly added raw contact is passed back to your activity's onActivityResult() callback method in the Intent argument of the 'data' field. To get the value, call getData(). Edit Contact Contact ACTION_EDIT CONTENT_LOOKUP_URI. The editor action will allow the user to edit all data associated with this contact. Contacts.CONTENT_ITEM_TYPE, one contact. Displays the contact editing screen in the Contacts application. Additional values that you add to the intent will appear. When the user clicks "Done" to save the changes, your activity will return to the foreground. Show a selector that can also add data. ACTION_INSERT_OR_EDIT N/A CONTENT_ITEM_TYPE This intent always displays the contact application selection screen. The user can select a contact to edit or add a new contact. Depending on the user's choice, the edit or add screen will appear and display the additional data you pass to the intent. If your app displays contact information such as an email address or phone number, use it.so that the user can add data to an existing contact. Contact, Note. There is no need to send a name value in the additions of this intent, as the user will always choose an existing name or add a new one. If you submit a name and the user decides to change it, the Contacts app displays the submitted name and overwrites the previous value. If the user doesn't notice and saves the change, the old value will be lost. The Contacts app on your device does not allow you to intentionally delete a pending contact or its data. To delete an unhandled contact, use ContentResolver.delete() or ContentProviderOperation.newDelete() instead. The following snippet shows how to create and send an intent that inserts a new raw contact and data: // Get values from the UI val name = contactNameEditText.text.toString() val phone = contactPhoneEditText.text.toString() val email = contactEmailEditText .text.toString() val company = companyName.text.toString() val jobtitle = jobTitle.text.toString() /* * Shows adding data strings as a list of fields associated with a DATA key. */ // Defines a List array that will contain ContentValues objects for each row val contactData = arrayListOf() /* * Defines a raw contact row */ // Sets a row as a ContentValues object val rawContactRow = ContentValues( ).apply { // Add the account type and name to the row put(ContactsContract.RawContacts.ACCOUNT_TYPE, selectedAccount.type) put(ContactsContract.RawContacts.ACCOUNT_NAME, selectedAccount.name) } // Add the row to the contactData array row.add(rawContactRow) /* * Sets the phone number data string */ // Sets the string as a ContentValues object val phoneRow = ContentValues().apply { // Specifies the MIME type for this data string (all data strings must be identified by their type) put(ContactsContract.Data.MIMETYPE,ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE) // Add the phone number and type to the string put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone.) } // Add d Adding a linearray contactData.add(phoneRow) /* * Configures the email data row */ // Configures the row as a ContentValues object val emailRow = ContentValues().apply { // Determines the MIME type for this data row (all data lines must be checked with their type) put(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE) // Add the email address and its type to the line put(ContactsContract.CommonDataKinds.Email.ADDRESS, email) } / / Add a row to the contactData field .add(emailRow) // Create a new intent to send to the device's contacts application val insertIntent = Intent(ContactsContract.Intents.Insert.ACTION).apply { // Set the MIME type to this expected by insert action type = ContactsContract. RawContacts.CONTENT_TYPE // Set new contact name putExtra(ContactsContract.Intents.Insert.NAME, name) // Set new company and job title putExtra(ContactsContract.Intents.Insert. COMPANY, company) putExtra(ContactsContract.Intents.Insert. JOB_TITLE , jobname) /* * Appends fields to add intent. It must be a wrapable object to *travel between processes. Device Contacts is expected to have a key * Intents.Insert.DATA */ putParcelableArrayListExtra(ContactsContract.Intents.Insert.DATA, contactData) } // In the Add Contacts step, pass an intent to launch Device Contacts. startActivity(insertIntent) // Get values from UI String name = contactNameEditText.getText().toString(); string phone = contactPhoneEditText.getText().toString(); String email = contactEmailEditText.getText().toString(); String company = companyName.getText().toString(); string task_name = task_name.getText().toString(); // Creates a new intent to send to the Contacts app on the device. Intent insertIntent = new Intent(ContactsContract.Intents.Insert.ACTIONS); // Set the MIME type to what the embed operation expects. insertIntent.setType(ContactsContract.RawContacts.CONTENT_TYPE); // Setting up a new contactinsertIntent.putExtra(ContactsContract.Intents.Insert.NAME, Name); // Set the new company and position insertIntent.putExtra(ContactsContract.Intents.Insert.COMPANY, company); insertIntent.putExtra(ContactsContract.Intents.Insert.JOB_TITLE, job title); /* * Indicates adding rows of data as a list of fields associated with the DATA key */ // Defines a list of fields containing ContentValues objects for each row ArrayList contactData = new ArrayList () ; /* * Defines a raw contact row */ // Sets the row as a ContentValues object ContentValues rawContactRow = new ContentValues(); // Add account type and name to the row rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_TYPE, selectedAccount.getType()); rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_NAME, selectedAccount.getName()); // Add a row to the array contactData.add(rawContactRow); /* * Sets the row with phone number data */ // Sets the row as a ContentValues object ContentValuesphoneRow = new ContentValues(); // Get the MIME type of this data row (all data rows must be tagged with their type) phoneRow.put(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE); // Add the phone number and its type to the row phoneRow.put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone); // Add a row to the array contactData.add(phoneRow); /* * Sets the email data row */ // Sets the row as a ContentValues objectContentValuesemailRow = new ContentValues(); // Determines the MIME type of this data row (all data rows must be tagged with their type) emailRow.put(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE); // Adds the email address and its type to the row emailRow.put(ContactsContract.CommonDataKinds.Email.ADDRESS, email); // Add a row to the contactData.add(emailRow) array; /* * Add fields to intent extras. It must be a packageable item for * travelprocesses. The device's Contacts app expects its key to be * Intents.Insert.DATA */ insertIntent.putParcelableArrayListExtra(ContactsContract.Intents.Insert.DATA, contactData); // Send the intent to launch the Contacts app on the device in the Add Contacts activity. startActivity(insertIntent); Since the contact repository contains sensitive and sensitive data that users believe to be accurate and up-to-date, the contact service provider has well-defined data integrity policies. You are responsible for adhering to this policy when changing your contact details. Here are the important rules: Always add a ContactsContract.CommonDataKinds.StructuredName line for each ContactsContract.RawContacts line you add. A ContactsContract.RawContacts row without a ContactsContract.CommonDataKinds.StructuredName row in the ContactsContract.Data table can cause aggregation issues. Always combine new ContactsContract.Data rows with their ContactsContract.RawContacts parent row. A ContactsContract.Data string that is not associated with ContactsContract.RawContacts will not appear in the device's contacts app, which can cause issues with sync adapters. Change data only for raw contacts you have. Please note that the contact service provider typically manages data from several different types of online accounts/services. You must ensure that your application only changes or deletes data for rows that you own, and that it only inserts data with the account type and name you control. Always use the constants defined in ContactsContract and its subclasses for credentials, content URIs, URI paths, column names, MIME types, and TYPE values. Using these constants will help avoid errors. You will also be notified by compiler warnings if any of the constants are deprecated. Custom data strings By creating and using your own custom MIME types, you can insert, edit, delete, and retrieve your own data strings in the ContactsContract.Data table. Your strings are limited in usedefined in ContactsContract.DataColumns, although you can map your own type-specific column names to the default column names. The Devices app shows the contact information for your lines, but it can't be edited or deleted, and users can't add more data. To allow users to edit their own data rows, you must expose editor behavior in your application. To display custom data, upload a Contacts.xml file that contains a element and one or more of its child elements. This is described in more detail in the element. To learn more about custom MIME types, read the Content Provider Creation Guide. Contact Provider Synchronization Adapters The Contact Provider service is specifically designed to support the synchronization of contact data between your device and an online service. This allows users to download existing data to a new device and upload existing data to a new account. Synchronization also ensures that users have access to the latest data regardless of the source of additions and changes. Another advantage of synchronization is that contact information is available even when the device is not connected to the network. Although synchronization can be implemented in many ways, Android provides a plug-in synchronization system that automates the following tasks: Check for network availability. Scheduling and performing synchronization based on user preferences. Restarting a paused sync. You must provide a sync adapter plug-in to use this system. Each sync adapter is unique to the service and content provider, but can support multiple account names for the same service. The framework also allows multiple synchronization adapters for the same service and service provider. Sync Adapter Classes and Files The sync adapter is implemented as a subclass of the AbstractThreadedSyncAdapter class and is installed as part of the Android application. The system learns about the sync adapterin the application manifest and from a special XML file pointed to by the manifest. The XML file defines the online service account type and content provider credentials, which together uniquely identify the adapter. The sync adapter does not become active until the user adds the account to the sync adapter account type and enables sync for the content provider that the sync adapter is syncing with. At this point, the system begins to manage the adapter, calling it as needed to synchronize the content provider and the server. Note. Using the account type as part of the sync adapter identification allows the system to identify and group sync adapters that access different services within the same organization. For example, all Google Online Services sync adapters have the same com.google account type. When users add a Google account to their devices, all installed Google sync adapters are shown together;

each of the sync adapters listed syncs with a different content provider on the device. Because most services require users to verify their identity before accessing data, Android provides an authentication framework similar to, and often used in conjunction with, the sync adapter. The authentication system uses pluggable authenticators, which are subclasses of the AbstractAccountAuthenticator class. The authenticator verifies the user's identity by performing the following steps: Gathers a username, password, or similar information (user credentials). Sends credentials to the service. Checks the response of a service. If the service accepts credentials, the authenticator can store the credentials for later use. With a pluggable authentication framework, the AccountManager can provide access to all authenticators supported by the authenticator and chosen to be exposed, such as OAuth2 credentials. While authentication is not required, most contact services require it. However, you don't needAndroid authentication system for authentication. Implementing a Sync Adapter To implement a sync adapter for a contact provider, start by creating an Android application that contains: A service component that responds to system requests to bind to a sync adapter. When the system wants to start synchronization, it calls the onBind() method of the service to obtain an IBinder for the synchronization adapter. This allows the system to make interprocess calls to adapter methods. The actual sync adapter, implemented as a concrete subclass of AbstractThreadedSyncAdapter. This class handles loading data from the server, sending data from the device, and conflict resolution. The main work of the adapter is done by the onPerformSync() method. This class must be created as a single class. Subclass application. This class acts as a single synchronization adapter factory. Use the onCreate() method to synchronize the adapter and provide a static "derived" method that returns the onBind() method of the synchronization adapter service singleton. Optional: A service component that responds to system requests for user authentication. AccountManager starts this service and starts the authentication process. The onCreate() method of the service creates an instance of the authentication object. When the system wants to authenticate an Application Sync Adapter user account, it calls the service's onBind() method to obtain the authenticator's IBinder. This allows the system to make cross-process calls to authentication methods. Optional: A concrete subclass of AbstractAccountAuthenticator that handles authentication requests. This class provides methods that AccountManager uses to verify user credentials against the server. The details of the authentication process vary greatly depending on the server technology used. See the server software documentation for more information.XML files that define the system synchronization adapter and authenticator. The synchronization adapters and authentication service components described above are defined in elements in the application manifest. These elements contain child elements that provide the system with specific data: The element for the sync adapter service refers to the XML file res/xml/syncadapter.xml. This file in turn defines the URI of the web service that is synchronized with the contact provider and the account type of the web service. Optional: The authenticator element points to the XML file res/xml/authenticator.xml. This file, in turn, specifies the type of account supported by this authenticator, as well as the user interface resources that are provided during the authentication process. The account type specified in this element must match the account type specified in the sync adapter. Social Stream Data The android.provider.ContactsContract.StreamItems and android.provider.ContactsContract.StreamItemPhotos tables manage incoming social media data. You can write a sync adapter that adds streaming data from your network to these tables, or you can read streaming data from these tables and display them in your application, or both. These features allow you to integrate social networking services and applications with the Android social networking experience. The text stream elements of a social stream are always associated with a raw contact. android.provider.ContactsContract.StreamItemsColumns#RAW_CONTACT_ID is bound to the _ID value of the raw contact. The stream element string also stores the account type and account name of the raw contact. Store data from your stream in the following columns: android.provider.ContactsContract.StreamItemsColumns#ACCOUNT_TYPE Required. The user account type of the raw contact associated with this flow element. Don't forget to set this value when inserting the flow element.Required. The user account name of the raw contact associated with this flow element. Don't forget to set this value when inserting the flow element. Identity columns are required. When inserting a stream element, the following identity columns must be inserted: android.provider.ContactsContract.StreamItemsColumns#CONTACT_ID: The android.provider.BaseColumns#_ID value of the contact this stream element is associated with. android.provider.ContactsContract.StreamItemsColumns#CONTACT_LOOKUP_KEY: The value of android.provider.ContactsContract.ContactsColumns#LOOKUP_KEY for the contact this stream item is associated with. android.provider.ContactsContract.StreamItemsColumns#RAW_CONTACT_ID: The android.provider.BaseColumns#_ID value for the raw contact this stream item is associated with. android.provider.ContactsContract.StreamItemsColumns#COMMENTS Optional. Stores summary information that you can view at the beginning of the flow element. android.provider.ContactsContract.StreamItemsColumns#TEXT The text of the stream item, either the content published by the item's source, or a description of the action that created the stream item. This column can contain any formatting and inline resource images that can be rendered with fromHtml(). The service provider may shorten or lengthen long content, but tries not to break the tags. android.provider.ContactsContract.StreamItemsColumns#TIMESTAMP A text string containing the time in milliseconds since the stream item was inserted or updated. Applications that insert or update stream elements are responsible for maintaining this column; it is not automatically supported by the contact provider. To display the IDs of your stream items, use android.provider.ContactsContract.StreamItemsColumns#RES_ICON, android.provider.ContactsContract.StreamItemsColumns#RES_LABEL and android.provider.ContactsContract.StreamItemsColum to bind resources in your application. #RES_Column.the table also contains android.provider.ContactsContract.StreamItemsColumns#SYNC1 using android.provider.ContactsContract.StreamItemsColumns#SYNC4 for the exclusive use of sync adapters. Social media stream photos The android.provider.ContactsContract.StreamItemPhotos table stores photos associated with a stream item. The table column android.provider.ContactsContract.StreamItemPhotosColumns#STREAM_ITEM_ID is combined with the values in the _ID column of the android.provider.ContactsContract.StreamItems table. References to photos are stored in the table in the following columns: android.provider.ContactsContract.StreamItemPhotos#PHOTO (BLOB). A binary representation of a photo that has been resized by a storage and display service provider. This column is available for backward compatibility with previous versions of the contacts provider that used it to store photos. However, in the current version, this column should not be used to store photos. Instead, use android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_FILE_ID or android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_URI to save photos to a file (both described in the points below). This column now has a legible photo thumbnail. android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_FILE_ID Numeric photo ID of the raw contact. Add this value to the DisplayPhoto.CONTENT_URI constant to get the content URI that points to a single photo file, then call openAssetFileDescriptor() to get the holder of the photo file. android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_URI A content URI pointing directly to the photo file of the photo displayed in this row. Call the openAssetFileDescriptor() method with this URI to get the holder of the image file. Using Social Streaming Tables These tables work like other main contact provider tables with the exception: these tables require additional access permissions. readyour app must have android.Manifest.permission#READ_SOCIAL_STREAM. Your app must have android.Manifest.permission#WRITE_SOCIAL_STREAM permission to edit them. The android.provider.ContactsContract.StreamItems table has a limited number of rows stored for each raw contact. When this limit is reached, the contacts provider makes room for new stream item rows by automatically deleting the rows with the oldest android.provider.ContactsContract.StreamItemsColumns#TIMESTAMP. To get the limit, query the content URI android.provider.ContactsContract.StreamItems#CONTENT_LIMIT_URI. You can set all non-content URI arguments to null. The query returns a cursor containing one row with one column android.provider.ContactsContract.StreamItems#MAX_ITEMS. The android.provider.ContactsContract.StreamItems.StreamItemPhotos class defines a subtable of android.provider.ContactsContract.StreamItemPhotos that contains rows of photos per stream item. Interaction with Social Streaming Social streaming data, managed by the contact provider in conjunction with the contact application on the device, offers an efficient way to connect the social networking system to existing contacts. The following features are available: When synchronizing a social network service with a contact provider using the sync adapter, the user's recent contact activity can be retrieved and stored in the android.provider.ContactsContract.StreamItems and android.provider.ContactsContract.StreamItemPhotos tables for later use. In addition to normal synchronization, you can activate the synchronization adapter to download additional data when the user selects a contact to view. This allows the sync adapter to download high-resolution photos and recent stream items for that contact. After you register notifications in your device's contacts app and contact provider, you can receive notifications when a contact appears andThis item updates the contact status from your service. This approach can be faster and uses less bandwidth than a full sync using a sync adapter. Users can add a contact to your social networking service by viewing it in the Contacts app on their device. You enable this with the Invite Contact feature, which you enable by combining an action that adds an existing contact to your network and an XML file that provides the contact app on your device and the contact provider with details about your app. The periodic synchronization of Stream items with the contact provider is the same as other synchronizations. For more information about synchronization, see Contact Service Provider Sync Adapters. Registering notifications and inviting contacts is covered in the next two sections. Registering for social browsing To register the sync adapter to receive notifications when a user views a contact managed by the sync adapter: Create a file named Contacts.xml in the res/xml/ directory of your project. If you already have this file, you can skip this step. Add to this file. If this item already exists, you can skip this step. To register one of your activities to handle a stream item, in the device's Contacts app, click and add the viewStreamItemActivity="activityclass" attribute to the item, where activityclass is the fully qualified name of the activity class that the intent should get from the device Contacts app. . To record one of your activities to process a user's click on a device-streamed photo in the Contacts app, add the viewStreamItemPhotoActivity="activityclass" attribute to the element, where activityclass is the fully qualified name of the activity class that the intended to receive. from your device's Contacts app. The element is described in more detail in the element. The incoming intent contains the content URI of the item or photo that the user clicked. To perform separate operations on text elements and images, use both attributes in the same file. Social interaction Users do not have to leave the device's Contacts app to invite a contact to a social network. Instead, you can send the People app on your device to invite a contact to one of your activities. To set this up: Create a file called Contacts.xml in your project's res/xml/ directory. If you already have this file, you can skip this step. Add contained in: res/xml/contacts.xml may contain: Description: Declares Android components and UI markup that allows contacts to invite a user on social networks, notify users, when one of their social media streams gets updated, etc. Note that the attribute prefix android: is not required Attributes: KutsuContactActivity The fully qualified name of the activity class in your app that you want to activate when the user selects Add Connection text vatContactActionLabel in the app ko ntacts on the device The string to display for the action specified in the Add connection menu. For example, you can use "Follow on my network". You can use a string resource ID for this label. viewContactNotifyService The fully qualified name of class ain an application that should receive a notification when a user views a contact. This notification is sent from the contacts application on the device; allows your application to defer data-intensive operations until they are needed. For example, your application can respond to this notification by loading and displaying the contact's high-resolution photo and the latest social media items. This feature is described in more detail in the Social Stream Interactions section. viewGroupActivity The fully qualified name of the activity class in your application that can display group information. When a user taps a group feature in the People app on their device, the user interface for that action is displayed. viewGroupActionLabel The label that the Contacts application displays for a UI control that allows the user to view groups in your application. A string resource identifier is allowed for this attribute. viewStreamItemActivity The fully qualified name of the activity class in your app that the contact app runs on the device when the user taps a stream item in a raw contact. viewStreamItemPhotoActivity The fully qualified name of the activity class in your app that the contact app runs on the device when a user clicks a photo in a stream item to get raw contact. The Element The element controls the display of your application's custom data rows in the Contacts UI. It has the following syntax: Contained in: Description: This element is used to specify the Contact The application displays the custom data content row as part of the raw contact information. Each child element of the element represents a custom data row type that your sync adapter adds to ContactsContract.Data.Add a element for each custom MIME type you use. If you have a custom row of data that you don't want to display, you don't need to add an entry. Attributes: android:mimeType A custom MIME type that you defined for one of the custom data row types in the ContactsContract.Data table. For example, the value vnd.android.cursor.item/vnd.example.locationstatus can be a custom MIME type for a data line that records the contact's last known location. android:icon A drawable Android resource that displays the Contacts app next to your information. Use it to indicate to the user that the data comes from your service. android:summaryColumn The column name for the first of the two values retrieved from the data row. The value appears as the first line of the record for that row of data. The first row is intended as a summary of the data, but is optional. See also android:detailColumn. android:detailColumn The column name for the second of the two values retrieved from the data row. The value appears as the second row of the record for that row of data. See also android:summaryColumn. Other features of the contact provider In addition to the main features described in the previous sections, the contact provider provides the following useful features for working with contact data: contact groups photo features of contact groups The contact provider can optionally mark collections of related contacts with group data. If the server associated with the user account wants to manage groups, the account type synchronization adapter must transfer the group data between the contact provider and the server. When users add a new contact to the server and then place that contact in a new group, the synchronization adapter must add the new group to the ContactsContract.Groups table. The group or groups to which the raw contact belongs are stored in the ContactsContract.Data table using the ContactsContract.CommonDataKinds.GroupMembership MIME type. If you want to syncwhich adds raw contact data from the server to the contact provider and you do not use groups, you must instruct the provider to share your data. In the code that runs when the user adds the account to the device, update the ContactsContract.Settings line that the contact provider added to the account. In this row, set the value in the Settings.UNGROUPED_VISIBLE column to 1. If you do this, the contact provider will always make your contact information visible, even if you don't use groups. Contact Photos The ContactsContract.Data table stores photos as rows with MIME type Photo.CONTENT_ITEM_TYPE. The row's CONTACT_ID column is concatenated with the _ID column of the raw contact it belongs to. The ContactsContract.Contacts.Photo class defines a ContactsContract.Contacts subtable that contains information about the contact's primary photo, which is the primary photo of the contact's primary contact. Similarly, the ContactsContract.RawContacts.DisplayPhoto class defines a ContactsContract.RawContacts subtable that contains the photo information for the contact's original photo. The reference documentation for ContactsContract.Contacts.Photo and ContactsContract.RawContacts.DisplayPhoto provides examples of how to get information about a photo. There is no appropriate class to get the primary thumbnail for a raw contact, but you can query the ContactsContract.Data table by selecting Raw Contact_ID, Photo.CONTENT_ITEM_TYPE and the IS_PRIMARY column to find the primary contact row with photos. A person's social stream data may also include photos. They are stored in the android.provider.ContactsContract.StreamItemPhotos table, which is described in more detail in the Social Stream Photos section. Pictures.