Shared Preferences

Shared Preferences allow you to save and retrieve data in the form of key,value pair. In order to use shared preferences, you have to call a method getSharedPreferences() that returns a SharedPreference instance pointing to the file that contains the values of preferences.

```
SharedPreferences sharedpreferences = getSharedPreferences(MyPREFERENCES,
Context.MODE_PRIVATE);
```

You can save something in the sharedpreferences by using SharedPreferences.Editor class. You will call the edit method of SharedPreference instance and will receive it in an editor object. Its syntax is −

```
Editor editor = sharedpreferences.edit();
editor.putString("key", "value");
editor.commit();
```

Apart from the putString method, there are methods available in the editor class that allows manipulation of data inside shared preferences. They are listed as follows −

Sr.No	Mode & description
1	**apply()** It is an abstract method. It will commit your changes back from editor to the sharedPreference object you are calling
2	**clear()** It will remove all values from the editor
3	**remove(String key)** It will remove the value whose key has been passed as a parameter
4	**putLong(String key, long value)** It will save a long value in a preference editor
5	**putInt(String key, int value)** It will save a integer value in a preference editor
6	**putFloat(String key, float value)** It will save a float value in a preference editor

Session Management through Shared Preferences

In order to perform session management from shared preferences, we need to check the values or data stored in shared preferences in the **onResume** method. If we don't have the data, we will start the application from the beginning as it is newly installed. But if we got the data, we will start from the where the user left it. It is demonstrated in the example below −

Example

The below example demonstrates the use of Session Management. It crates a basic application that allows you to login for the first time. And then when you exit the application without logging out, you will be brought back to the same place if you start the application again. But if you logout from the application, you will be brought back to the main login screen.

To experiment with this example, you need to run this on an actual device or in an emulator.

Steps	Description
1	You will use android studio IDE to create an Android application under a package com.example.sairamkrishna.myapplication.
2	Modify src/MainActivity.java file to add progress code to add session code.
3	Create New Activity and it name as second.java.Edit this file to add progress code to add session code.
4	Modify res/layout/activity_main.xml file to add respective XML code.
5	Modify res/layout/second_main.xml file to add respective XML code.
7	Run the application and choose a running android device and install the application on it and verify the results.

Here is the content of **MainActivity.java**.

```
package com.example.sairamkrishna.myapplication;

import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {
   EditText ed1,ed2,ed3;
   Button b1;
   Intent in;

   public static final String MyPREFERENCES = "MyPrefs" ;
   public static final String Name = "nameKey";
   public static final String Phone = "phoneKey";
   public static final String Email = "emailKey";
   SharedPreferences sharedpreferences;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      ed1=(EditText)findViewById(R.id.editText);
      ed2=(EditText)findViewById(R.id.editText2);
      ed3=(EditText)findViewById(R.id.editText3);

      b1=(Button)findViewById(R.id.button);
      sharedpreferences = getSharedPreferences(MyPREFERENCES,
Context.MODE_PRIVATE);

      b1.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            String n  = ed1.getText().toString();
            String ph  = ed2.getText().toString();
            String e  = ed3.getText().toString();

            SharedPreferences.Editor editor =
sharedpreferences.edit();

            editor.putString(Name, n);
            editor.putString(Phone, ph);
            editor.putString(Email, e);
            editor.commit();
            in = new 
Intent(MainActivity.this,second_main.class);
            startActivity(in);
         }
      });
   }
}
```

Here is the content of **second_main.java**.

```
package com.example.sairamkrishna.myapplication;

import android.app.Activity;
import android.content.Context;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class second_main extends Activity {
   Button bu=null;
   Button bu2=null;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.second_main);

      bu=(Button)findViewById(R.id.button2);
      bu2=(Button)findViewById(R.id.button3);
   }

   public  void logout(View view){
      SharedPreferences sharedpreferences = 
getSharedPreferences(MainActivity.MyPREFERENCES, 
Context.MODE_PRIVATE);
      SharedPreferences.Editor editor = 
sharedpreferences.edit();
      editor.clear();
      editor.commit();
   }

   public void close(View view){
      finish();
   }
}
```

Here is the content of **activity_main.xml**.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
   android:layout_height="match_parent"
android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   android:paddingBottom="@dimen/activity_vertical_margin"
tools:context=".MainActivity">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Shared Preference"
      android:id="@+id/textView"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="35dp" />

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials Point"
      android:id="@+id/textView2"
      android:layout_below="@+id/textView"
      android:layout_centerHorizontal="true"
      android:textSize="35dp"
      android:textColor="#ff16ff01" />

   <EditText
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/editText"
      android:layout_below="@+id/textView2"
      android:layout_marginTop="67dp"
      android:hint="Name"
      android:layout_alignParentRight="true"
      android:layout_alignParentEnd="true"
      android:layout_alignParentLeft="true"
      android:layout_alignParentStart="true" />

   <EditText
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/editText2"
      android:layout_below="@+id/editText"
      android:layout_alignParentLeft="true"
      android:layout_alignParentStart="true"
      android:layout_alignParentRight="true"
      android:layout_alignParentEnd="true"
      android:hint="Pass" />

   <EditText
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/editText3"
      android:layout_below="@+id/editText2"
      android:layout_alignParentLeft="true"
      android:layout_alignParentStart="true"
      android:layout_alignParentRight="true"
      android:layout_alignParentEnd="true"
      android:hint="Email" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="login"
      android:id="@+id/button"
      android:layout_below="@+id/editText3"
      android:layout_centerHorizontal="true"
      android:layout_marginTop="50dp" />

</RelativeLayout>
```

Here is the content of **second_main.xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
android:layout_width="match_parent"
   android:layout_height="match_parent">

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Logout"
      android:onClick="logout"
      android:id="@+id/button2"
      android:layout_gravity="center_horizontal"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:layout_marginTop="191dp" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Close"
      android:onClick="close"
      android:id="@+id/button3"
      android:layout_below="@+id/button2"
      android:layout_centerHorizontal="true"
      android:layout_marginTop="69dp" />

</RelativeLayout>
```

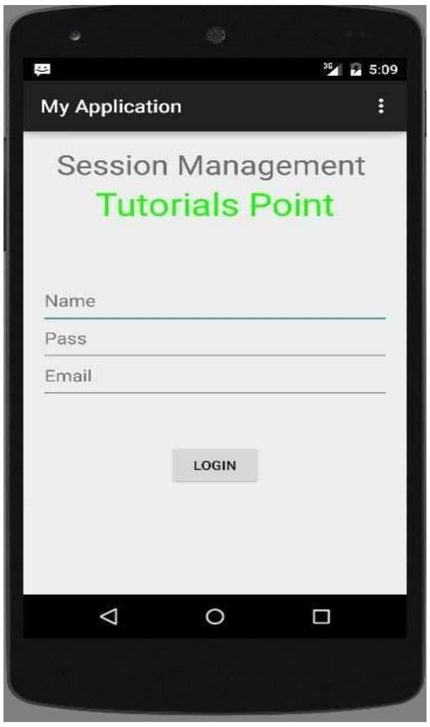
Here is the content of **Strings.xml**.

```
<resources>
   <string name="app_name">My Application</string>
</resources>
```

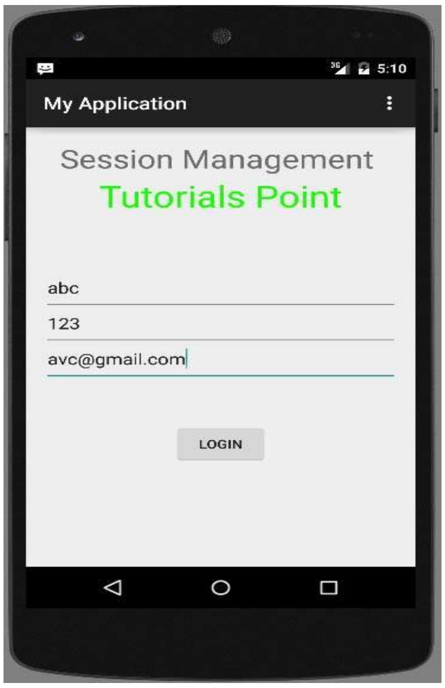
Here is the content of **AndroidManifest.xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.sairamkrishna.myapplication" >

   <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name=".MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN"
/>
            <category
android:name="android.intent.category.LAUNCHER" />
         </intent-filter>

      </activity>

      <activity android:name=".second"></activity>

   </application>
</manifest>
```

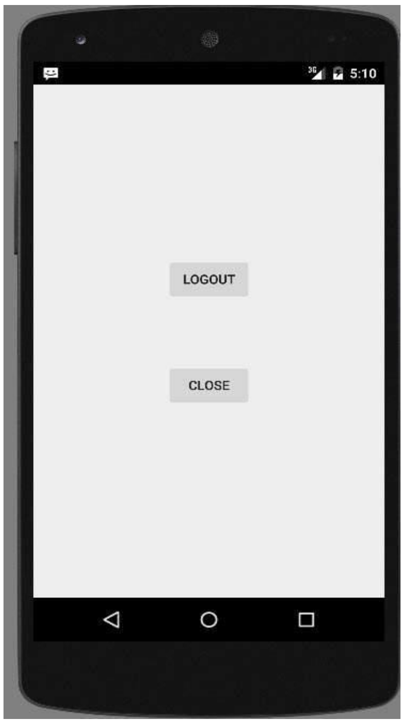
Let's try to run your application. I assume you had created your AVD while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run icon from the tool bar. Android studio installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window –



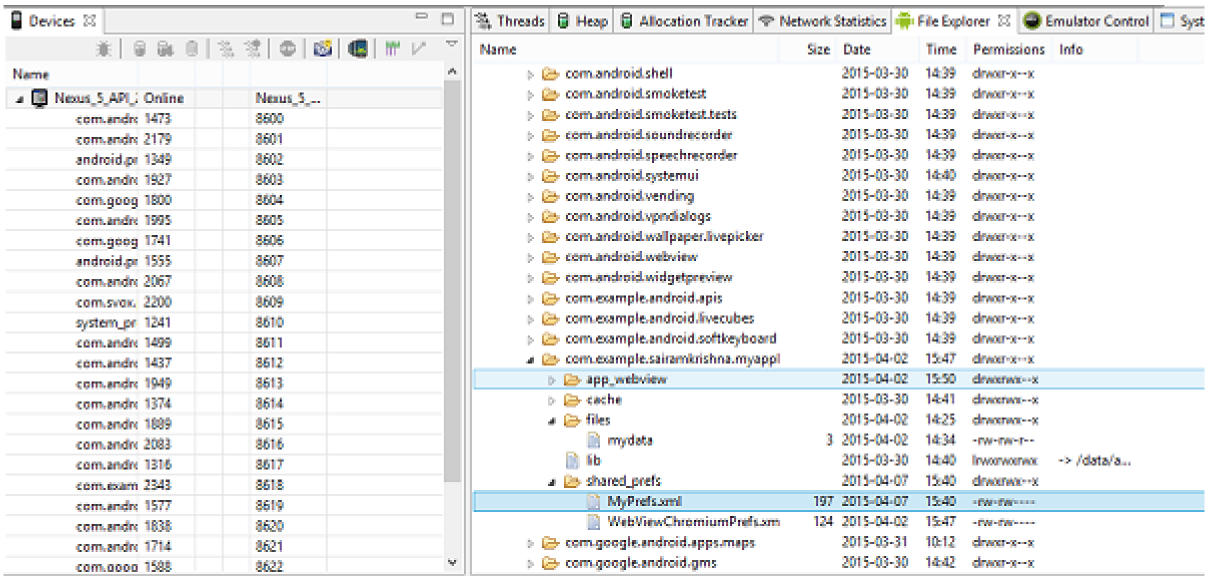
Type in your username and password **(type anything you like, but remember what you type)**, and click on login button. It is shown in the image below –



As soon as you click on login button, you will be brought to this Welcome screen. Now your login information is stored in shared preferences.



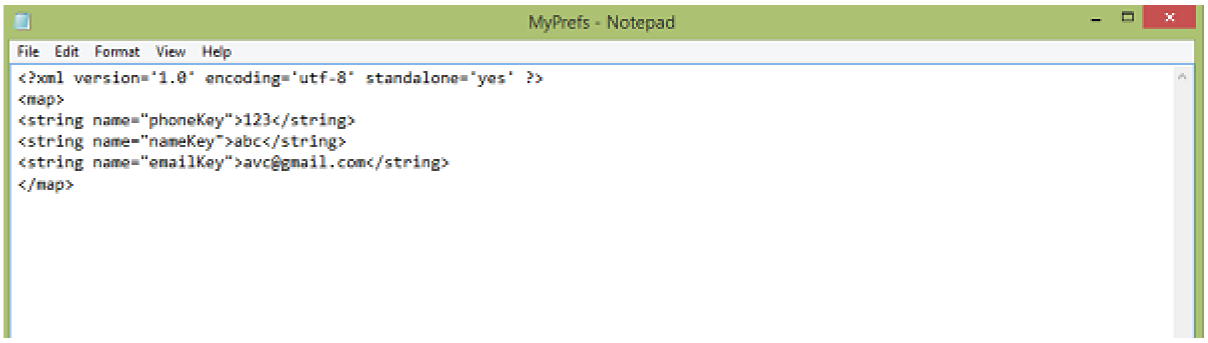
Now click on **Exit without logout** button and you will be brought back to the home screen and in preference file out put would be as shown below image



Name		
Nexus_5_API_:	Online	Nexus_5_...
com.andr	1473	8600
com.andr	2179	8601
android.p	1349	8602
com.andr	1927	8603
com.goog	1800	8604
com.andr	1995	8605
com.goog	1741	8606
android.p	1555	8607
com.andr	2067	8608
com.svox.	2200	8609
system_pr	1241	8610
com.andr	1499	8611
com.andr	1437	8612
com.andr	1949	8613
com.andr	1374	8614
com.andr	1889	8615
com.andr	2083	8616
com.andr	1316	8617
com.exam	2343	8618
com.andr	1577	8619
com.andr	1838	8620
com.andr	1714	8621
com.goog	1588	8622

Name	Size	Date	Time	Permissions	Info
com.android.shell		2015-03-30	14:39	drwxr-x--x	
com.android.smoketest		2015-03-30	14:39	drwxr-x--x	
com.android.smoketest.tests		2015-03-30	14:39	drwxr-x--x	
com.android.soundrecorder		2015-03-30	14:39	drwxr-x--x	
com.android.speechrecorder		2015-03-30	14:39	drwxr-x--x	
com.android.systemui		2015-03-30	14:40	drwxr-x--x	
com.android.vending		2015-03-30	14:39	drwxr-x--x	
com.android.vpndialogs		2015-03-30	14:39	drwxr-x--x	
com.android.wallpaper.livepicker		2015-03-30	14:39	drwxr-x--x	
com.android.webview		2015-03-30	14:39	drwxr-x--x	
com.android.widgetpreview		2015-03-30	14:39	drwxr-x--x	
com.example.android.apis		2015-03-30	14:39	drwxr-x--x	
com.example.android.livecubes		2015-03-30	14:39	drwxr-x--x	
com.example.android.softkeyboard		2015-03-30	14:39	drwxr-x--x	
com.example.sairamkrishna.myappl		2015-04-02	15:47	drwxr-x--x	
app_webview		2015-04-02	15:50	drwxrwx--x	
cache		2015-03-30	14:41	drwxrwx--x	
files		2015-04-02	14:25	drwxrwx--x	
mydata	3	2015-04-02	14:34	-rw-rw-r--	
lib		2015-03-30	14:40	lrwxrwxrwx	-> /data/a...
shared_prefs		2015-04-07	15:40	drwxrwx--x	
MyPrefs.xml	197	2015-04-07	15:40	-rw-rw----	
WebViewChromiumPrefs.xm	124	2015-04-02	15:47	-rw-rw----	
com.google.android.apps.maps		2015-03-31	10:12	drwxr-x--x	
com.google.android.gms		2015-03-30	14:42	drwxr-x--x	

If you open myPref.xml file as note file, it would be as follows



```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="phoneKey">123</string>
<string name="nameKey">abc</string>
<string name="emailKey">avc@gmail.com</string>
</map>
```

If you click on logout button, it will erase preference values. and if you entered different values as inputs,it will enter those values as preference in XML.

Access app-specific files

In many cases, your app creates files that other apps don't need to access, or shouldn't access. The system provides the following locations for storing such *app-specific* files:

- **Internal storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. The system prevents other apps from accessing these locations, and on Android 10 (API level 29) and higher, these locations are encrypted. These characteristics make these locations a good place to store sensitive data that only your app itself can access.
- **External storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. Although it's possible for another app to access these directories if that app has the proper permissions, the files stored in these directories are meant for use only by your app. If you specifically intend to create files that other apps should be able to access, your app should store these files in the shared storage part of external storage instead.

When the user uninstalls your app, the files saved in app-specific storage are removed. Because of this behavior, you shouldn't use this storage to save anything that the user expects to persist independently of your app. For example, if your app allows users to capture photos, the user would expect that they can access those photos even after they uninstall your app. So you should instead use shared storage to save those types of files to the appropriate media collection.

The following sections describe how to store and access files within app-specific directories.

Access from internal storage

For each app, the system provides directories within internal storage where an app can organize its files. One directory is designed for your app's persistent files, and another contains your app's cached files. Your app doesn't require any system permissions to read and write to files in these directories.

Other apps cannot access files stored within internal storage. This makes internal storage a good place for app data that other apps shouldn't access.

Keep in mind, however, that these directories tend to be small. Before writing app-specific files to internal storage, your app should query the free space on the device.

Access persistent files

Your app's ordinary, persistent files reside in a directory that you can access using the filesDir property of a context object. The framework provides several methods to help you access and store files in this directory.

Access and store files

You can use the File API to access and store files.

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to use the File API:

Java

```
File file = new File(context.getFilesDir(), filename);
```

Store a file using a stream

As an alternative to using the File API, you can call openFileOutput() to get a FileOutputStream that writes to a file within the filesDir directory.

The following code snippet shows how to write some text to a file:

Java

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

To allow other apps to access files stored in this directory within internal storage, use a FileProvider with the FLAG_GRANT_READ_URI_PERMISSION attribute.

Access a file using a stream

To read a file as a stream, use openFileInput():

Java

```
FileInputStream fis = context.openFileInput(filename);
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Error occurred when opening raw file for reading.
} finally {
    String contents = stringBuilder.toString();
}
```

Note: If you need to access a file as a stream at install time, save the file in your project's `/res/raw` directory. You can open these files with `openRawResource()`, passing in the filename prefixed with `R.raw` as the resource ID. This method returns an `InputStream` that you can use to read the file. You cannot write to the original file.

View list of files

You can get an array containing the names of all files within the filesDir directory by calling fileList(), as shown in the following code snippet:

Java

```
Array<String> files = context.fileList();
```

Create nested directories

You can also create nested directories, or open an inner directory, by calling getDir() in Kotlin-based code or by passing the root directory and a new directory name into a File constructor in Java-based code:

Java

```
File directory = context.getFilesDir();
File file = new File(directory, filename);
```

Note: `dataDir` is always an ancestor directory of this new directory.

Create cache files

If you need to store sensitive data only temporarily, you should use the app's designated cache directory within internal storage to save the data. As is the case for all app-specific storage, the files stored in this directory are removed when the user uninstalls your app, although the files in this directory might be removed sooner.

Note: This cache directory is designed to store a small amount of your app's sensitive data. To determine how much cache space is currently available for your app, call `getCacheQuotaBytes()`.

To create a cached file, call File.createTempFile():

Java

```
File.createTempFile(filename, null, context.getCacheDir());
```

Your app accesses a file in this directory using the cacheDir property of a context object and the File API:

Java

```
File cacheFile = new File(context.getCacheDir(), filename);
```

Remove cache files

Even though Android sometimes deletes cache files on its own, you shouldn't rely on the system to clean up these files for you. You should always maintain your app's cache files within internal storage.

To remove a file from the cache directory within internal storage, use one of the following methods:

- The delete() method on a File object that represents the file:

  Java

  ```
  cacheFile.delete();
  ```
- The deleteFile() method of the app's context, passing in the name of the file:

  Java

  ```
  context.deleteFile(cacheFileName);
  ```

Access from external storage

If internal storage doesn't provide enough space to store app-specific files, consider using external storage instead. The system provides directories within external storage where an app can organize files that provide value to the user only within your app. One directory is designed for your app's persistent files, and another contains your app's cached files.

On Android 4.4 (API level 19) or higher, your app doesn't need to request any storage-related permissions to access app-specific directories within external storage. The files stored in these directories are removed when your app is uninstalled.

Caution: The files in these directories aren't guaranteed to be accessible, such as when a removable SD card is taken out of the device. If your app's functionality depends on these files, you should instead store the files within internal storage.

On devices that run Android 9 (API level 28) or lower, your app can access the app-specific files that belong to other apps, provided that your app has the appropriate storage permissions. To give users more control over their files and to limit file clutter, apps that target Android 10 (API level 29) and higher are given scoped access into external storage, or scoped storage, by default. When scoped storage is enabled, apps cannot access the app-specific directories that belong to other apps.

Verify that storage is available

Because external storage resides on a physical volume that the user might be able to remove, verify that the volume is accessible before trying to read app-specific data from, or write app-specific data to, external storage.

You can query the volume's state by calling Environment.getExternalStorageState(). If the returned state is MEDIA_MOUNTED, then you can read and write app-specific files within external storage. If it's MEDIA_MOUNTED_READ_ONLY, you can only read these files.

For example, the following methods are useful to determine the storage availability:

Java

```
// Checks if a volume containing external storage is available
// for read and write.
private boolean isExternalStorageWritable() {
   return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED);
}

// Checks if a volume containing external storage is available to at least read.
private boolean isExternalStorageReadable() {
    return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED) ||

Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED_READ_ONLY);
}
```

On devices without removable external storage, use the following command to enable a virtual volume for testing your external storage availability logic:

```
adb shell sm set-virtual-disk true
```

Select a physical storage location

Sometimes, a device that allocates a partition of its internal memory as external storage also provides an SD card slot. This means that the device has multiple physical volumes that could contain external storage, so you need to select which one to use for your app-specific storage.

To access the different locations, call ContextCompat.getExternalFilesDirs(). As shown in the code snippet, the first element in the returned array is considered the primary external storage volume. Use this volume unless it's full or unavailable.

Java

```
File[] externalStorageVolumes =
        ContextCompat.getExternalFilesDirs(getApplicationContext(), null);
File primaryExternalStorage = externalStorageVolumes[0];
```

Note: If your app is used on a device that runs Android 4.3 (API level 18) or lower, then the array contains just one element, which represents the primary external storage volume.

Access persistent files

To access app-specific files from external storage, call getExternalFilesDir().

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to call getExternalFilesDir():

Java

```
File appSpecificExternalDir = new File(context.getExternalFilesDir(null), filename);
```

Note: On Android 11 (API level 30) and higher, apps cannot create their own app-specific directory on external storage.

Create cache files

To add an app-specific file to the cache within external storage, get a reference to the externalCacheDir:

Java

```
File externalCacheFile = new File(context.getExternalCacheDir(), filename);
```

Remove cache files

To remove a file from the external cache directory, use the delete() method on a File object that represents the file:

Java

```
externalCacheFile.delete();
```

Media content

If your app works with media files that provide value to the user only within your app, it's best to store them in app-specific directories within external storage, as demonstrated in the following code snippet:

Java

```
@Nullable
File getAppSpecificAlbumStorageDir(Context context, String albumName) {
    // Get the pictures directory that's inside the app-specific directory on
    // external storage.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (file == null || !file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

It's important that you use directory names provided by API constants like DIRECTORY_PICTURES. These directory names ensure that the files are treated properly by the system. If none of the pre-defined sub-directory names suit your files, you can instead pass null into getExternalFilesDir(). This returns the root app-specific directory within external storage.

Query free space

Many users don't have much storage space available on their devices, so your app should consume space thoughtfully.

If you know ahead of time how much data you're storing, you can find out how much space the device can provide your app by calling getAllocatableBytes(). The return value of getAllocatableBytes() might be larger than the current amount of free space on the device. This is because the system has identified files that it can remove from other apps' cache directories.

If there's enough space to save your app's data, call allocateBytes(). Otherwise, your app can request the user to remove some files from the device or remove all cache files from the device.

The following code snippet shows an example of how your app can query free space on the device:

Java

```
// App needs 10 MB within internal storage.
private static final long NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10L;

StorageManager storageManager =
        getApplicationContext().getSystemService(StorageManager.class);
UUID appSpecificInternalDirUuid = storageManager.getUuidForPath(getFilesDir());
long availableBytes =
        storageManager.getAllocatableBytes(appSpecificInternalDirUuid);
if (availableBytes >= NUM_BYTES_NEEDED_FOR_MY_APP) {
    storageManager.allocateBytes(
            appSpecificInternalDirUuid, NUM_BYTES_NEEDED_FOR_MY_APP);
} else {
    // To request that the user remove all app cache files instead, set
    // "action" to ACTION_CLEAR_APP_CACHE.
    Intent storageIntent = new Intent();
    storageIntent.setAction(ACTION_MANAGE_STORAGE);
}
```

Save data using SQLite

bookmark_border

Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the android.database.sqlite package.

Define a schema and contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table. Each inner class enumerates the corresponding table's columns.

Note: By implementing the `BaseColumns` interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as `CursorAdapter` expect it to have. It's not required, but this can help your database work harmoniously with the Android framework.

For example, the following contract defines the table name and column names for a single table representing an RSS feed:

Java

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // make the constructor private.
    private FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    }
}
```

Create a database using an SQL helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

Java

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_TITLE + " TEXT," +
    FeedEntry.COLUMN_NAME_SUBTITLE + " TEXT)";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Just like files that you save on the device's internal storage, Android stores your database in your app's private folder. Your data is secure, because by default this area is not accessible to other apps or the user.

The SQLiteOpenHelper class contains a useful set of APIs for managing your database. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call getWritableDatabase() or getReadableDatabase().

Note: Because they can be long-running, be sure that you call `getWritableDatabase()` or `getReadableDatabase()` in a background thread. See Threading on Android for more information.

To use SQLiteOpenHelper, create a subclass that overrides the onCreate() and onUpgrade() callback methods. You may also want to implement the onDowngrade() or onOpen() methods, but they are not required.

For example, here's an implementation of SQLiteOpenHelper that uses some of the commands shown above:

Java

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
   // If you change the database schema, you must increment the database version.
   public static final int DATABASE_VERSION = 1;
   public static final String DATABASE_NAME = "FeedReader.db";

   public FeedReaderDbHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }
   public void onCreate(SQLiteDatabase db) {
      db.execSQL(SQL_CREATE_ENTRIES);
   }
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      // This database is only a cache for online data, so its upgrade policy is
      // to simply to discard the data and start over
      db.execSQL(SQL_DELETE_ENTRIES);
      onCreate(db);
   }
   public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      onUpgrade(db, oldVersion, newVersion);
   }
}
```

To access your database, instantiate your subclass of SQLiteOpenHelper:

Java

```
FeedReaderDbHelper dbHelper = new FeedReaderDbHelper(getContext());
```

Put information into a database

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

Java

```
// Gets the data repository in write mode
SQLiteDatabase db = dbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle);

// Insert the new row, returning the primary key value of the new row
long newRowId = db.insert(FeedEntry.TABLE_NAME, null, values);
```

The first argument for insert() is simply the table name.

The second argument tells the framework what to do in the event that the ContentValues is empty (i.e., you did not put any values). If you specify the name of a column, the framework inserts a row and sets the value of that column to null. If you specify null, like in this code sample, the framework does not insert a row when there are no values.

The insert() methods returns the ID for the newly created row, or it will return -1 if there was an error inserting the data. This can happen if you have a conflict with pre-existing data in the database.

Read information from a database

To read from a database, use the query() method, passing it your selection criteria and desired columns. The method combines elements of insert() and update(), except the column list defines the data you want to fetch (the "projection"), rather than the data to insert. The results of the query are returned to you in a Cursor object.

Java

```
SQLiteDatabase db = dbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    BaseColumns._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE
    };

// Filter results WHERE "title" = 'My Title'
String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?";
String[] selectionArgs = { "My Title" };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_SUBTITLE + " DESC";

Cursor cursor = db.query(
    FeedEntry.TABLE_NAME,   // The table to query
    projection,             // The array of columns to return (pass null to get all)
    selection,              // The columns for the WHERE clause
    selectionArgs,          // The values for the WHERE clause
    null,                   // don't group the rows
    null,                   // don't filter by row groups
    sortOrder               // The sort order
    );
```

The third and fourth arguments (selection and selectionArgs) are combined to create a WHERE clause. Because the arguments are provided separately from the selection query, they are escaped before being combined. This makes your selection statements immune to SQL injection. For more detail about all arguments, see the query() reference.

To look at a row in the cursor, use one of the Cursor move methods, which you must always call before you begin reading values. Since the cursor starts at position -1, calling moveToNext() places the "read position" on the first entry in the results and returns whether or not the cursor is already past the last entry in the result set. For each row, you can read a column's value by calling one of the Cursor get methods, such as getString() or getLong(). For each of the get methods, you must pass the index position of the column you desire, which you can get by calling getColumnIndex() or getColumnIndexOrThrow(). When finished iterating through results, call close() on the cursor to release its resources. For example, the following shows how to get all the item IDs stored in a cursor and add them to a list:

Java

```
List itemIds = new ArrayList<>();
while(cursor.moveToNext()) {
  long itemId = cursor.getLong(
      cursor.getColumnIndexOrThrow(FeedEntry._ID));
  itemIds.add(itemId);
}
cursor.close();
```

Delete information from a database

To delete rows from a table, you need to provide selection criteria that identify the rows to the delete() method. The mechanism works the same as the selection arguments to the query() method. It divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

Java

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { "MyTitle" };
// Issue SQL statement.
int deletedRows = db.delete(FeedEntry.TABLE_NAME, selection, selectionArgs);
```

The return value for the delete() method indicates the number of rows that were deleted from the database.

Update a database

When you need to modify a subset of your database values, use the update() method.

Updating the table combines the ContentValues syntax of insert() with the WHERE syntax of delete().

Java

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

// New value for one column
String title = "MyNewTitle";
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the title
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
String[] selectionArgs = { "MyOldTitle" };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

The return value of the update() method is the number of rows affected in the database.

Persisting database connection

Since getWritableDatabase() and getReadableDatabase() are expensive to call when the database is closed, you should leave your database connection open for as long as you possibly need to access it. Typically, it is optimal to close the database in the onDestroy() of the calling Activity.

Java

```
@Override
protected void onDestroy() {
    dbHelper.close();
    super.onDestroy();
}
```

Content provider basics

bookmark_border

A content provider manages access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data. However, content providers are primarily used by other applications, which access the provider using a provider client object. Together, providers and provider clients offer a consistent, standard interface to data that also handles interprocess communication and secure data access.

Typically you work with content providers in one of two scenarios: implementing code to access an existing content provider in another application or creating a new content provider in your application to share data with other applications.

This page covers the basics of working with existing content providers. To learn about implementing content providers in your own applications, see Create a content provider.

This topic describes the following:

- How content providers work.
- The API you use to retrieve data from a content provider.
- The API you use to insert, update, or delete data in a content provider.
- Other API features that facilitate working with providers.

Overview

A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

A content provider coordinates access to the data storage layer in your application for a number of different APIs and components. As illustrated in figure 1, these include the following:

- Sharing access to your application data with other applications
- Sending data to a widget
- Returning custom search suggestions for your application through the search framework using SearchRecentSuggestionsProvider
- Synchronizing application data with your server using an implementation of AbstractThreadedSyncAdapter
- Loading data in your UI using a CursorLoader

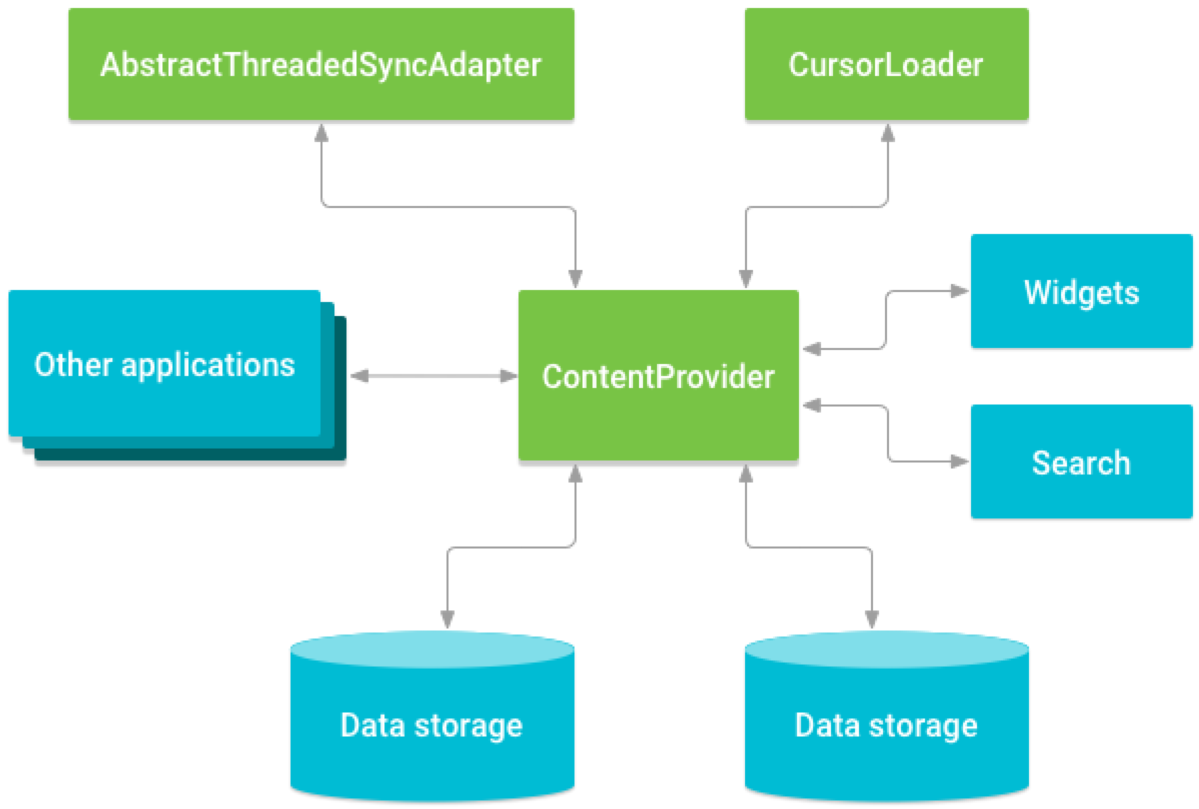


Figure 1. Relationship between a content provider and other components.

Access a provider

When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client. The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider.

The provider object receives data requests from clients, performs the requested action, and returns the results. This object has methods that call identically named methods in the provider object, an instance of one of the concrete subclasses of ContentProvider. The ContentResolver methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

A common pattern for accessing a ContentProvider from your UI uses a CursorLoader to run an asynchronous query in the background. The Activity or Fragment in your UI calls a CursorLoader to the query, which in turn gets the ContentProvider using the ContentResolver.

This lets the UI continue to be available to the user while the query is running. This pattern involves the interaction of a number of different objects, as well as the underlying storage mechanism, as illustrated in figure 2.

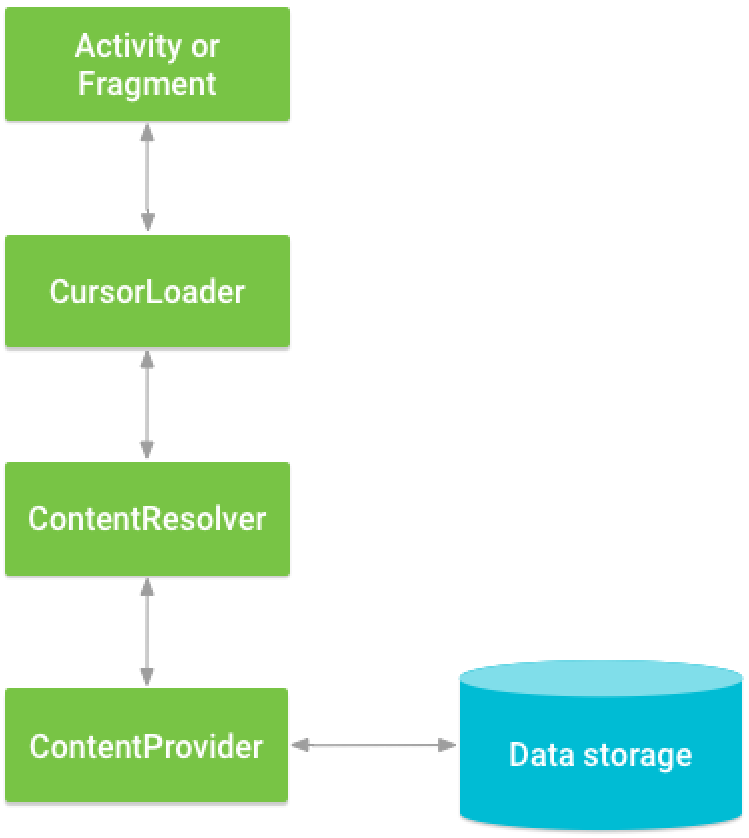


Figure 2. Interaction between ContentProvider, other classes, and storage.

Note: To access a provider, your application usually has to request specific permissions in its manifest file. This development pattern is described in more detail in the Content provider permissions section.

One of the built-in providers in the Android platform is the User Dictionary Provider, which stores the nonstandard words that the user wants to keep. Table 1 illustrates what the data might look like in this provider's table:

Table 1: Sample user dictionary table.

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

In table 1, each row represents an instance of a word not found in a standard dictionary. Each column represents a piece of data for that word, such as the locale in which it was first encountered. The column headers are column names that are stored in the provider. So to refer to a row's locale, for example, you refer to its locale column. For this provider, the _ID column serves as a *primary key* column that the provider automatically maintains.

To get a list of the words and their locales from the User Dictionary Provider, you call ContentResolver.query(). The query() method calls the ContentProvider.query() method defined by the User Dictionary Provider. The following lines of code show a ContentResolver.query() call:

KotlinJava

```
// Queries the UserDictionary and returns results
cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
    projection,                        // The columns to return for each row
    selectionClause,                   // Selection criteria
    selectionArgs,                     // Selection criteria
    sortOrder);                        // The sort order for the returned rows
```

Table 2 shows how the arguments to query(Uri,projection,selection,selectionArgs,sortOrder) match a SQL SELECT statement:

Table 2: query() compared to SQL query.

query() **argument**	**SELECT keyword/parameter**	**Notes**
Uri	FROM *table_name*	Uri maps to the table in the provider named *table_name*.
projection	*col,col,col,...*	projection is an array of columns that is included for each row retrieved.
selection	WHERE *col* = *value*	selection specifies the criteria for selecting rows.
selectionArgs	No exact equivalent. Selection arguments replace ? placeholders in the selection clause.	
sortOrder	ORDER BY *col,col,...*	sortOrder specifies the order in which rows appear in the returned Cursor.

Content URIs

A *content URI* is a URI that identifies data in a provider. Content URIs include the symbolic name of the entire provider—its *authority*—and a name that points to a table—a *path*. When you call a client method to access a table in a provider, the content URI for the table is one of the arguments.

In the preceding lines of code, the constant CONTENT_URI contains the content URI of the User Dictionary Provider's Words table. The ContentResolver object parses out the URI's authority and uses it to *resolve* the provider by comparing the authority to a system table of known providers. The ContentResolver can then dispatch the query arguments to the correct provider.

The ContentProvider uses the path part of the content URI to choose the table to access. A provider usually has a path for each table it exposes.

In the previous lines of code, the full URI for the Words table is:

```
content://user_dictionary/words
```

- The content:// string is the *scheme*, which is always present and identifies this as a content URI.
- The user_dictionary string is the provider's authority.
- The words string is the table's path.

Many providers let you access a single row in a table by appending an ID value to the end of the URI. For example, to retrieve a row whose _ID is 4 from the User Dictionary Provider, you can use this content URI:

KotlinJava

```
Uri singleUri = ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

You often use ID values when you retrieve a set of rows and then want to update or delete one of them.

Note: The `Uri` and `Uri.Builder` classes contain convenience methods for constructing well-formed URI objects from strings. The `ContentUris` class contains convenience methods for appending ID values to a URI. The previous snippet uses `withAppendedId()` to append an ID to the User Dictionary Provider content URI.

Retrieve data from the provider

This section describes how to retrieve data from a provider, using the User Dictionary Provider as an example.

For the sake of clarity, the code snippets in this section call `ContentResolver.query()` on the UI thread. In actual code, however, do queries asynchronously on a separate thread. You can use the `CursorLoader` class, which is described in more detail in the Loaders guide. Also, the lines of code are snippets only. They don't show a complete application.

To retrieve data from a provider, follow these basic steps:

1. Request read access permission for the provider.
2. Define the code that sends a query to the provider.

Request read access permission

To retrieve data from a provider, your application needs read access permission for the provider. You can't request this permission at runtime. Instead, you have to specify that you need this permission in your manifest, using the <uses-permission> element and the exact permission name defined by the provider.

When you specify this element in your manifest, you are requesting this permission for your application. When users install your application, they implicitly grant this request.

To find the exact name of the read access permission for the provider you're using, as well as the names for other access permissions used by the provider, look in the provider's documentation.

The role of permissions in accessing providers is described in more detail in the Content provider permissions section.

The User Dictionary Provider defines the permission `android.permission.READ_USER_DICTIONARY` in its manifest file, so an application that wants to read from the provider must request this permission.

Construct the query

The next step in retrieving data from a provider is to construct a query. The following snippet defines some variables for accessing the User Dictionary Provider:

KotlinJava

```
// A "projection" defines the columns that are returned for each row
String[] mProjection =
{
    UserDictionary.Words._ID,    // Contract class constant for the _ID column name
    UserDictionary.Words.WORD,   // Contract class constant for the word column name
    UserDictionary.Words.LOCALE  // Contract class constant for the locale column name
};

// Defines a string to contain the selection clause
String selectionClause = null;

// Initializes an array to contain selection arguments
String[] selectionArgs = {""};
```

The next snippet shows how to use `ContentResolver.query()`, using the User Dictionary Provider as an example. A provider client query is similar to a SQL query, and it contains a set of columns to return, a set of selection criteria, and a sort order.

The set of columns that the query returns is called a *projection*, and the variable is `mProjection`.

The expression that specifies the rows to retrieve is split into a selection clause and selection arguments. The selection clause is a combination of logical and boolean expressions, column names, and values. The variable is `mSelectionClause`. If you specify the replaceable parameter `?` instead of a value, the query method retrieves the value from the selection arguments array, which is the variable `mSelectionArgs`.

In the next snippet, if the user doesn't enter a word the selection clause is set to `null` and the query returns all the words in the provider. If the user enters a word, the selection clause is set to `UserDictionary.Words.WORD + " = ?"` and the first element of selection arguments array is set to the word the user enters.

KotlinJava

```
/*
 * This defines a one-element String array to contain the selection argument.
 */
String[] selectionArgs = {""};

// Gets a word from the UI
searchString = searchWord.getText().toString();

// Remember to insert code here to check for invalid or malicious input

// If the word is the empty string, gets everything
if (TextUtils.isEmpty(searchString)) {
    // Setting the selection clause to null returns all words
    selectionClause = null;
    selectionArgs[0] = "";

} else {
    // Constructs a selection clause that matches the word that the user entered
    selectionClause = UserDictionary.Words.WORD + " = ?";

    // Moves the user's input string to the selection arguments
    selectionArgs[0] = searchString;

}

// Does a query against the table and returns a Cursor object
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI, // The content URI of the words table
    projection,                       // The columns to return for each row
    selectionClause,                  // Either null or the word the user entered
    selectionArgs,                    // Either empty or the string the user entered
    sortOrder);                       // The sort order for the returned rows

// Some providers return null if an error occurs, others throw an exception
if (null == mCursor) {
    /*
     * Insert code here to handle the error. Be sure not to use the cursor! You can
     * call android.util.Log.e() to log this error.
     *
     */
// If the Cursor is empty, the provider found no matches
} else if (mCursor.getCount() < 1) {

    /*
     * Insert code here to notify the user that the search is unsuccessful. This isn't necessarily
     * an error. You can offer the user the option to insert a new row, or re-type the
     * search term.
     */

} else {
    // Insert code here to do something with the results

}
```

This query is analogous to the following SQL statement:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY word ASC;
```

In this SQL statement, the actual column names are used instead of contract class constants.

Protect against malicious input

If the data managed by the content provider is in a SQL database, including external untrusted data into raw SQL statements can lead to SQL injection.

Consider the following selection clause:

KotlinJava

```
// Constructs a selection clause by concatenating the user's input to the column name
String selectionClause = "var = " + userInput;
```

If you do this, you let the user potentially concatenate malicious SQL onto your SQL statement. For example, the user can enter "nothing; DROP TABLE *;" for `mUserInput`, which results in the selection clause `var = nothing; DROP TABLE *;`.

Since the selection clause is treated as a SQL statement, this might cause the provider to erase all the tables in the underlying SQLite database, unless the provider is set up to catch SQL injection attempts.

To avoid this problem, use a selection clause that uses `?` as a replaceable parameter and a separate array of selection arguments. This way, the user input is bound directly to the query rather than being interpreted as part of a SQL statement.

Because it's not treated as SQL, the user input can't inject malicious SQL. Instead of using concatenation to include the user input, use this selection clause:

KotlinJava

```
// Constructs a selection clause with a replaceable parameter
String selectionClause =  "var = ?";
```

Set up the array of selection arguments like this:

KotlinJava

```
// Defines an array to contain the selection arguments
String[] selectionArgs = {""};
```

Put a value in the selection arguments array like this:

KotlinJava

```
// Sets the selection argument to the user's input
selectionArgs[0] = userInput;
```

A selection clause that uses `?` as a replaceable parameter and an array of selection arguments array is the preferred way to specify a selection, even if the provider isn't based on a SQL database.

Display query results

The `ContentResolver.query()` client method always returns a `Cursor` containing the columns specified by the query's projection for the rows that match the query's selection criteria. A `Cursor` object provides random read access to the rows and columns it contains.

Using `Cursor` methods, you can iterate over the rows in the results, determine the data type of each column, get the data out of a column, and examine other properties of the results.

Some `Cursor` implementations automatically update the object when the provider's data changes, trigger methods in an observer object when the `Cursor` changes, or both.

Note: A provider can restrict access to columns based on the nature of the object making the query. For example, the Contacts Provider restricts access for some columns to sync adapters, so it doesn't return them to an activity or service.

If no rows match the selection criteria, the provider returns a `Cursor` object for which `Cursor.getCount()` is 0—that is, an empty cursor.

If an internal error occurs, the results of the query depend on the particular provider. It might return `null`, or it can throw an `Exception`.

Since a `Cursor` is a list of rows, a good way to display the contents of a `Cursor` is to link it to a `ListView` using a `SimpleCursorAdapter`.

The following snippet continues the code from the previous snippet. It creates a SimpleCursorAdapter object containing the Cursor retrieved by the query, and sets this object to be the adapter for a ListView.

KotlinJava

```
// Defines a list of columns to retrieve from the Cursor and load into an output row
String[] wordListColumns =
{
    UserDictionary.Words.WORD,   // Contract class constant containing the word column name
    UserDictionary.Words.LOCALE  // Contract class constant containing the locale column name
};

// Defines a list of View IDs that receive the Cursor columns for each row
int[] wordListItems = { R.id.dictWord, R.id.locale};

// Creates a new SimpleCursorAdapter
cursorAdapter = new SimpleCursorAdapter(
    getApplicationContext(),               // The application's Context object
    R.layout.wordlistrow,                  // A layout in XML for one row in the ListView
    mCursor,                               // The result from the query
    wordListColumns,                       // A string array of column names in the cursor
    wordListItems,                         // An integer array of view IDs in the row layout
    0);                                    // Flags (usually none are needed)

// Sets the adapter for the ListView
wordList.setAdapter(cursorAdapter);
```

Note: To back a `ListView` with a `Cursor`, the cursor must contain a column named `_ID`. Because of this, the query shown previously retrieves the `_ID` column for the `Words` table, even though the `ListView` doesn't display it. This restriction also explains why most providers have a `_ID` column for each of their tables.

Get data from query results

In addition to displaying query results, you can use them for other tasks. For example, you can retrieve spellings from the User Dictionary Provider and then look them up in other providers. To do this, you iterate over the rows in the Cursor, as shown in the following example:

KotlinJava

```
// Determine the column index of the column named "word"
int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);

/*
 * Only executes if the cursor is valid. The User Dictionary Provider returns null if
 * an internal error occurs. Other providers might throw an Exception instead of returning null.
 */
```

```
if (mCursor != null) {
    /*
     * Moves to the next row in the cursor. Before the first movement in the cursor, the
     * "row pointer" is -1, and if you try to retrieve data at that position you get an
     * exception.
     */
    while (mCursor.moveToNext()) {

        // Gets the value from the column
        newWord = mCursor.getString(index);

        // Insert code here to process the retrieved word
        ...
        // End of while loop
    }
} else {

    // Insert code here to report an error if the cursor is null or the provider threw an exception
}
```

Cursor implementations contain several "get" methods for retrieving different types of data from the object. For example, the previous snippet uses getString(). They also have a getType() method that returns a value indicating the data type of the column.

Release query result resources

Cursor objects must be closed if they are not needed anymore, so that resources associated with them are released sooner. This can be done either by calling close(), or by using a try-with-resources statement in the Java programming language or the use() function in the Kotlin programming language.

Content provider permissions

A provider's application can specify permissions that other applications must have to access the provider's data. These permissions let the user know what data an application tries to access. Based on the provider's requirements, other applications request the permissions they need in order to access the provider. End users see the requested permissions when they install the application.

If a provider's application doesn't specify any permissions, then other applications have no access to the provider's data, unless the provider is exported. Additionally, components in the provider's application always have full read and write access, regardless of the specified permissions.

The User Dictionary Provider requires the android.permission.READ_USER_DICTIONARY permission to retrieve data from it. The provider has a separate android.permission.WRITE_USER_DICTIONARY permission for inserting, updating, or deleting data.

To get the permissions needed to access a provider, an application requests them with a <uses-permission> element in its manifest file. When the Android Package Manager installs the application, the user must approve all of the permissions the application requests. If the user approves them, Package Manager continues the installation. If the user doesn't approve them, Package Manager stops the installation.

The following sample <uses-permission> element requests read access to the User Dictionary Provider:

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

The impact of permissions on provider access is explained in more detail in Security tips.

Insert, update, and delete data

In the same way that you retrieve data from a provider, you also use the interaction between a provider client and the provider's ContentProvider to modify data. You call a method of ContentResolver with arguments that are passed to the corresponding method of ContentProvider. The provider and provider client automatically handle security and interprocess communication.

Insert data

To insert data into a provider, you call the ContentResolver.insert() method. This method inserts a new row into the provider and returns a content URI for that row. The following snippet shows how to insert a new word into the User Dictionary Provider:

KotlinJava

```
// Defines a new Uri object that receives the result of the insertion
Uri newUri;
...
// Defines an object to contain the new values to insert
ContentValues newValues = new ContentValues();

/*
 * Sets the values of each column and inserts the word. The arguments to the "put"
 * method are "column name" and "value".
 */
newValues.put(UserDictionary.Words.APP_ID, "example.user");
newValues.put(UserDictionary.Words.LOCALE, "en_US");
newValues.put(UserDictionary.Words.WORD, "insert");
newValues.put(UserDictionary.Words.FREQUENCY, "100");

newUri = getContentResolver().insert(
    UserDictionary.Words.CONTENT_URI,   // The UserDictionary content URI
    newValues                           // The values to insert
);
```

The data for the new row goes into a single ContentValues object, which is similar in form to a one-row cursor. The columns in this object don't need to have the same data type, and if you don't want to specify a value at all, you can set a column to null using ContentValues.putNull().

The previous snippet doesn't add the _ID column, because this column is maintained automatically. The provider assigns a unique value of _ID to every row that is added. Providers usually use this value as the table's primary key.

The content URI returned in newUri identifies the newly added row with the following format:

```
content://user_dictionary/words/<id_value>
```

The <id_value> is the contents of _ID for the new row. Most providers can detect this form of content URI automatically and then perform the requested operation on that particular row.

To get the value of _ID from the returned Uri, call ContentUris.parseId().

Update data

To update a row, you use a ContentValues object with the updated values, just as you do with an insertion, and selection criteria, as you do with a query. The client method you use is ContentResolver.update(). You only need to add values to the ContentValues object for columns you're updating. If you want to clear the contents of a column, set the value to null.

The following snippet changes all the rows whose locale has the language "en" to a have a locale of null. The return value is the number of rows that were updated.

KotlinJava

```
// Defines an object to contain the updated values
ContentValues updateValues = new ContentValues();

// Defines selection criteria for the rows you want to update
String selectionClause = UserDictionary.Words.LOCALE +  " LIKE ?";
String[] selectionArgs = {"en_%"};

// Defines a variable to contain the number of updated rows
int rowsUpdated = 0;
...
/*
 * Sets the updated value and updates the selected words.
 */
updateValues.putNull(UserDictionary.Words.LOCALE);

rowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,  // The UserDictionary content URI
    updateValues,                      // The columns to update
    selectionClause,                   // The column to select on
    selectionArgs                      // The value to compare to
);
```

Sanitize user input when you call ContentResolver.update(). To learn more about this, read the Protect against malicious input section.

Delete data

Deleting rows is similar to retrieving row data. You specify selection criteria for the rows you want to delete, and the client method returns the number of deleted rows. The following snippet deletes rows whose app ID matches "user". The method returns the number of deleted rows.

KotlinJava

```
// Defines selection criteria for the rows you want to delete
String selectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] selectionArgs = {"user"};

// Defines a variable to contain the number of rows deleted
int rowsDeleted = 0;
...
// Deletes the words that match the selection criteria
rowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,  // The UserDictionary content URI
    selectionClause,                   // The column to select on
    selectionArgs                      // The value to compare to
);
```

Sanitize user input when you call ContentResolver.delete(). To learn more about this, read the Protecte against malicious input section.

Provider data types

Content providers can offer many different data types. The User Dictionary Provider offers only text, but providers can also offer the following formats:

- integer
- long integer (long)
- floating point
- long floating point (double)

Another data type that providers often use is a binary large object (BLOB) implemented as a 64 KB byte array. You can see the available data types by looking at the Cursor class "get" methods.

The data type for each column in a provider is usually listed in its documentation. The data types for the User Dictionary Provider are listed in the reference documentation for its contract class, UserDictionary.Words. Contract classes are described in the Contract classes section. You can also determine the data type by calling Cursor.getType().

Providers also maintain MIME data type information for each content URI they define. You can use the MIME type information to find out whether your application can handle data that the provider offers or to choose a type of handling based on the MIME type. You usually need the MIME type when you work with a provider that contains complex data structures or files.

For example, the ContactsContract.Data table in the Contacts Provider uses MIME types to label the type of contact data stored in each row. To get the MIME type corresponding to a content URI, call ContentResolver.getType().

The MIME type reference section describes the syntax of both standard and custom MIME types.

Alternative forms of provider access

Three alternative forms of provider access are important in application development:

- Batch access: you can create a batch of access calls with methods in the ContentProviderOperation class and then apply them with ContentResolver.applyBatch().
- Asynchronous queries: do queries in a separate thread. You can use a CursorLoader object. The examples in the Loaders guide demonstrate how to do this.
- Data access using intents: although you can't send an intent directly to a provider, you can send an intent to the provider's application, which is usually best-equipped to modify the provider's data.

Batch access and modification using intents are described in the following sections.

Batch access

Batch access to a provider is useful for inserting a large number of rows, for inserting rows in multiple tables in the same method call, and in general for performing a set of operations across process boundaries as a transaction, called an *atomic operation*.

To access a provider in batch mode, create an array of ContentProviderOperation objects and then dispatch them to a content provider with ContentResolver.applyBatch(). You pass the content provider's authority to this method, rather than a particular content URI.

This lets each ContentProviderOperation object in the array work against a different table. A call to ContentResolver.applyBatch() returns an array of results.

The description of the ContactsContract.RawContacts contract class includes a code snippet that demonstrates batch insertion.

Data access using intents

Intents can provide indirect access to a content provider. You can let the user access data in a provider even if your application doesn't have access permissions by either getting a result intent back from an application that has permissions or by activating an application that has permissions and letting the user do work in it.

Get access with temporary permissions

You can access data in a content provider, even if you don't have the proper access permissions, by sending an intent to an application that does have the permissions and receiving back a result intent containing URI permissions. These are permissions for a specific content URI that last until the activity that receives them is finished. The application that has permanent permissions grants temporary permissions by setting a flag in the result intent:

- **Read permission:** FLAG_GRANT_READ_URI_PERMISSION
- **Write permission:** FLAG_GRANT_WRITE_URI_PERMISSION

Note: These flags don't give general read or write access to the provider whose authority is contained in the content URI. The access is only for the URI itself.

When you send content URIs to another app, include at least one of these flags. The flags provide the following capabilities to any app that receives an intent and targets Android 11 (API level 30) or higher:

- Read from, or write to, the data that the content URI represents, depending on the flag included in the intent.
- Gain package visibility into the app containing the content provider that matches the URI authority. The app that sends the intent and the app that contains the content provider might be two different apps.

A provider defines URI permissions for content URIs in its manifest, using the android:grantUriPermissions attribute of the <provider> element as well as the <grant-uri-permission> child element of the <provider> element. The URI permissions mechanism is explained in more detail in the Permissions on Android guide.

For example, you can retrieve data for a contact in the Contacts Provider, even if you don't have the READ_CONTACTS permission. You might want to do this in an application that sends e-greetings to a contact on their birthday. Instead of requesting READ_CONTACTS, which gives you access to all the user's contacts and all of their information, let the user control which contacts your application uses. To do this, use the following process:

1. In your application, send an intent containing the action ACTION_PICK and the "contacts" MIME type CONTENT_ITEM_TYPE, using the method startActivityForResult().
2. Because this intent matches the intent filter for the People app's "selection" activity, the activity comes to the foreground.

3. In the selection activity, the user selects a contact to update. When this happens, the selection activity calls setResult(resultcode, intent) to set up an intent to give back to your application. The intent contains the content URI of the contact the user selected and the "extras" flags FLAG_GRANT_READ_URI_PERMISSION. These flags grant URI permission to your app to read data for the contact pointed to by the content URI. The selection activity then calls finish() to return control to your application.
4. Your activity returns to the foreground, and the system calls your activity's onActivityResult() method. This method receives the result intent created by the selection activity in the People app.
5. With the content URI from the result intent, you can read the contact's data from the Contacts Provider, even though you didn't request permanent read access permission to the provider in your manifest. You can then get the contact's birthday information or email address and then send the e-greeting.

Use another application

Another way to let the user modify data to which you don't have access permissions is to activate an application that does have permissions and let the user do the work there.

For example, the Calendar application accepts an ACTION_INSERT intent that lets you activate the application's insert UI. You can pass "extras" data in this intent, which the application uses to pre-populate the UI. Because recurring events have a complex syntax, the preferred way of inserting events into the Calendar Provider is to activate the Calendar app with an ACTION_INSERT and then let the user insert the event there.

Display data using a helper app

If your application *does* have access permissions, you might still use an intent to display data in another application. For example, the Calendar application accepts an ACTION_VIEW intent that displays a particular date or event. This lets you display calendar information without having to create your own UI. To learn more about this feature, see the Calendar provider overview.

The application you send the intent to doesn't have to be the application associated with the provider. For example, you can retrieve a contact from the Contact Provider, then send an ACTION_VIEW intent containing the content URI for the contact's image to an image viewer.

Contract classes

A contract class defines constants that help applications work with the content URIs, column names, intent actions, and other features of a content provider. Contract classes aren't included automatically with a provider. The provider's developer has to define them and then make them available to other developers. Many of the providers included with the Android platform have corresponding contract classes in the package android.provider.

For example, the User Dictionary Provider has a contract class UserDictionary containing content URI and column name constants. The content URI for the Words table is defined in the constant UserDictionary.Words.CONTENT_URI. The UserDictionary.Words class also contains column name constants, which are used in the example snippets in this guide. For example, a query projection can be defined like the following:

KotlinJava

```
String[] projection =
{
   UserDictionary.Words._ID,
   UserDictionary.Words.WORD,
   UserDictionary.Words.LOCALE
};
```

Another contract class is ContactsContract for the Contacts Provider. The reference documentation for this class includes example code snippets. One of its subclasses, ContactsContract.Intents.Insert, is a contract class that contains constants for intents and intent data.