Storage Options

Android provides several options for you to save persistent application data. The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications (and the user) and how much space your data requires.

Your data storage options are the following:

Shared Preferences
: Store private primitive data in key-value pairs.

Internal Storage
: Store private data on the device memory.

External Storage
: Store public data on the shared external storage.

SQLite Databases
: Store structured data in a private database.

Network Connection
: Store data on the web with your own network server.

Android provides a way for you to expose even your private data to other applications — with a content provider. A content provider is an optional component that exposes read/write access to your

STORAGE QUICKVIEW

- Use Shared Preferences for primitive data
- Use internal device storage for private data
- Use external storage for large data sets that are not private
- Use SQLite databases for structured storage

IN THIS DOCUMENT

SEE ALSO

Content Providers and Content Resolvers

application data, subject to whatever restrictions you want to impose. For more information about using content providers, see the Content Providers documentation.

Using Shared Preferences

The `SharedPreferences` class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types. You can use `SharedPreferences` to save any primitive data: booleans, floats, ints, longs, and strings. This data will persist across user sessions (even if your application is killed).

To get a `SharedPreferences` object for your application, use one of two methods:

- `getSharedPreferences()` - Use this if you need multiple preferences files identified by name, which you specify with the first parameter.
- `getPreferences()` - Use this if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, you don't supply a name.

To write values:

1. Call `edit()` to get a `SharedPreferences.Editor`.
2. Add values with methods such as `putBoolean()` and `putString()`.
3. Commit the new values with `commit()`

To read values, use `SharedPreferences` methods such as `getBoolean()` and `getString()`.

Here is an example that saves a preference for silent keypress mode in a calculator:

> **User Preferences**
>
> Shared preferences are not strictly for saving "user preferences," such as what ringtone a user has chosen. If you're interested in creating user preferences for your application, see `PreferenceActivity`, which provides an Activity framework for you to create user preferences, which will be automatically persisted (using shared preferences).

```
public class Calc extends Activity {
```

```
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

Using the Internal Storage

You can save files directly on the device's internal storage. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user). When the user uninstalls your application, these files are removed.

To create and write a private file to the internal storage:

1. Call `openFileOutput()` with the name of the file and the operating mode. This returns a `FileOutputStream`.
2. Write to the file with `write()`.
3. Close the stream with `close()`.

For example:

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

`MODE_PRIVATE` will create the file (or replace a file of the same name) and make it private to your application. Other modes available are: `MODE_APPEND`, `MODE_WORLD_READABLE`, and `MODE_WORLD_WRITEABLE`.

To read a file from internal storage:

1. Call `openFileInput()` and pass it the name of the file to read. This returns a `FileInputStream`.
2. Read bytes from the file with `read()`.
3. Then close the stream with `close()`.

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

Saving cache files

If you'd like to cache some data, rather than store it persistently, you should use `getCacheDir()` to open a `File` that represents the internal directory where your application should save temporary cache files.

When the device is low on internal storage space, Android may delete these cache files to recover space. However, you should not rely on the system to clean up these files for you. You should always maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB. When the user uninstalls your application, these files are removed.

Other useful methods

`getFilesDir()`
: Gets the absolute path to the filesystem directory where your internal files are saved.

`getDir()`
: Creates (or opens an existing) directory within your internal storage space.

`deleteFile()`
: Deletes a file saved on the internal storage.

`fileList()`
: Returns an array of files currently saved by your application.

Using the External Storage

Every Android-compatible device supports a shared "external storage" that you can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

It's possible that a device using a partition of the internal storage for the external storage may also offer an SD card slot. In this case, the SD card is *not* part of the external storage and your app cannot access it (the extra storage is intended only for user-provided media that the system scans).

> **Caution:** External storage can become unavailable if the user mounts the external storage on a computer or removes the media, and there's no security enforced upon files you save to the external storage. All applications can read and write files placed on the external storage and the user can remove them.

Checking media availability

Before you do any work with the external storage, you should always call `getExternalStorageState()` to check whether the media is available. The media might be mounted to a computer, missing, read-only, or in some other state. For example, here's how you can check the availability:

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all we need
    //  to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

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

Accessing files on external storage

If you're using API Level 8 or greater, use `getExternalFilesDir()` to open a `File` that represents the external storage directory where you should save your files. This method takes a `type` parameter that specifies the type of subdirectory you want, such as `DIRECTORY_MUSIC` and `DIRECTORY_RINGTONES` (pass `null` to receive the root of your application's file directory). This method will create the appropriate directory if necessary. By specifying the type of directory, you ensure that the Android's media scanner will properly categorize your files in the system (for example, ringtones are identified as ringtones and not music). If the user uninstalls your application, this directory and all its contents will be deleted.

If you're using API Level 7 or lower, use `getExternalStorageDirectory()`, to open a `File` representing the root of the external storage. You should then write your data in the following directory:

```
/Android/data/<package_name>/files/
```

The ***<package_name>*** is your Java-style package name, such as "`com.example.android.app`". If the user's device is running API Level 8 or greater and they uninstall your application, this directory and all its contents will be deleted.

Saving files that should be shared

If you want to save files that are not specific to your application and that should *not* be deleted when your application is uninstalled, save them to one of the public directories on the external storage. These directories lay at the root of the external storage, such as `Music/`, `Pictures/`, `Ringtones/`, and others.

In API Level 8 or greater, use `getExternalStoragePublicDirectory()`, passing it the type of public directory you want, such as `DIRECTORY_MUSIC`, `DIRECTORY_PICTURES`, `DIRECTORY_RINGTONES`, or others. This method will create the appropriate directory if necessary.

> **Hiding your files from the Media Scanner**
>
> Include an empty file named `.nomedia` in your external files directory (note the dot prefix in the filename). This will prevent Android's media scanner from reading your media files and including them in apps like Gallery or Music.

If you're using API Level 7 or lower, use `getExternalStorageDirectory()` to open a `File` that represents the root of the external storage, then save your shared files in one of the following directories:

`Music/` - Media scanner classifies all media found here as user music.

`Podcasts/` - Media scanner classifies all media found here as a podcast.

`Ringtones/` - Media scanner classifies all media found here as a ringtone.

`Alarms/` - Media scanner classifies all media found here as an alarm sound.

`Notifications/` - Media scanner classifies all media found here as a notification sound.

`Pictures/` - All photos (excluding those taken with the camera).

`Movies/` - All movies (excluding those taken with the camcorder).

`Download/` - Miscellaneous downloads.

Saving cache files

If you're using API Level 8 or greater, use `getExternalCacheDir()` to open a `File` that represents the external storage directory where you should save cache files. If the user uninstalls your application, these files will be automatically deleted. However, during the life of your application, you should manage these cache files and remove those that aren't needed in order to preserve file space.

If you're using API Level 7 or lower, use `getExternalStorageDirectory()` to open a `File` that represents the root of the external storage, then write your cache data in the following directory:

```
/Android/data/<package_name>/cache/
```

The ***`<package_name>`*** is your Java-style package name, such as "`com.example.android.app`".

Using Databases

Android provides full support for SQLite databases. Any databases you create will be accessible by name to any class in the application, but not outside the application.

The recommended method to create a new SQLite database is to create a subclass of `SQLiteOpenHelper` and override the `onCreate()` method, in which you can execute a SQLite command to create tables in the database. For example:

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

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

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

You can then get an instance of your **`SQLiteOpenHelper`** implementation using the constructor you've defined. To write to and read from the database, call **`getWritableDatabase()`** and **`getReadableDatabase()`**, respectively. These both return a **`SQLiteDatabase`** object that represents the database and provides methods for SQLite operations.

You can execute SQLite queries using the **`SQLiteDatabase query()`** methods, which accept various query parameters, such as the table to query, the projection, selection, columns, grouping, and others. For complex queries, such as those that require column aliases, you should use **`SQLiteQueryBuilder`**, which provides several convienent methods for building queries.

> Android does not impose any limitations beyond the standard SQLite concepts. We do recommend including an autoincrement value key field that can be used as a unique ID to quickly find a record. This is not required for private data, but if you implement a

Every SQLite query will return a `Cursor` that points to all the rows found by the query. The `Cursor` is always the mechanism with which you can navigate results from a database query and read rows and columns.

> content provider, you must include a unique ID using the `BaseColumns._ID` constant.

For sample apps that demonstrate how to use SQLite databases in Android, see the Note Pad and Searchable Dictionary applications.

Database debugging

The Android SDK includes a `sqlite3` database tool that allows you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases. See Examining sqlite3 databases from a remote shell to learn how to run this tool.

Using a Network Connection

You can use the network (when it's available) to store and retrieve data on your own web-based services. To do network operations, use classes in the following packages:

- `java.net.*`
- `android.net.*`

PREVIOUS NEXT