Zigurd MEDNIEKS
Series Editor

ANDROID™
DEEP
DIVE

Adam STROUD

ANDROID™ DATABASE BEST PRACTICES

Android™ Database Best Practices

While the device IDs are displayed to the screen, it is not always clear which device is represented by a given ID. Passing `-l` to the `adb devices` command causes `adb` to print additional information to help identify different devices. Listing 4.10 shows usage of the `adb devices` command with the `-l` flag.

Listing 4.10 Getting a List of Attached Devices with Device Names

```
bash-4.3$ adb devices -l
List of devices attached
HT4ASJT00075          device product:volantis model:Nexus_9 device:flounder
ZX1G22PJGX            device product:shamu model:Nexus_6 device:shamu

bash-4.3$
```

With the addition of the `-l` flag, it is now clear which device is represented by which ID so that the shell can be opened on the correct device.

Once the ID for the desired device has been determined, the `adb shell` command can be used along with the `-s` flag to indicate which device is the target for the `adb` command. The following snippet shows use of the `adb shell` command with the `-s` flag:

```
adb -s HT4ASJT00075 shell
```

> **Note**
>
> The `-s` flag passed to the `adb` command applies to `adb` itself and not the subcommand of `shell`. This means that the `-s` flag can be used with any `adb` subcommand.

Once a shell to the desired device is connected, the file system hierarchy can be navigated using standard Linux commands, such as `cd` to change directories and `ls` to retrieve a directory listing.

Permissions and the Android File System

When using the `adb shell` command, it is important to remember that the Android environment is heavily based on Linux. Each app is treated like a user on a Linux system with a home directory and permissions set on the home directory to disallow other apps from reading its private data. This is an intentional security feature that is built into Android in the same way that this is a security feature built into Linux to protect user data. On Android, apps have home directories in the `/data/data` folder. The actual name of the app's home directory is the same as the package name that uniquely identifies the app in the system. Listing 4.11 shows a part of a directory listing of `/data/data`, including the permissions of each subdirectory.

Listing 4.11 `/data/data` Directory Listing

```
root@generic_x86_64:/ # ls -l /data/data
drwxr-x--x u0_a0   u0_a0    2015-12-16 14:04 com.android.backupconfirm
```

```
drwxr-x--x u0_a15  u0_a15   2015-12-16 14:04 com.android.backuptester
drwxr-x--x u0_a17  u0_a17   2015-12-16 14:04 com.android.browser
drwxr-x--x u0_a18  u0_a18   2015-12-16 14:04 com.android.calculator2
drwxr-x--x u0_a19  u0_a19   2015-12-16 14:04 com.android.calendar
drwxr-x--x u0_a33  u0_a33   2015-12-16 14:04 com.android.camera
drwxr-x--x u0_a20  u0_a20   2015-12-16 14:04 com.android.captiveportallogin
drwxr-x--x u0_a21  u0_a21   2015-12-16 14:04 com.android.certinstaller
drwxr-x--x u0_a2   u0_a2    2016-03-24 20:40 com.android.contacts
drwxr-x--x u0_a22  u0_a22   2015-12-16 14:04 com.android.customlocale2
drwxr-x--x u0_a3   u0_a3    2015-12-16 14:05 com.android.defcontainer
drwxr-x--x u0_a23  u0_a23   2015-12-16 14:04 com.android.deskclock
drwxr-x--x u0_a24  u0_a24   2015-12-16 14:04 com.android.development
drwxr-x--x u0_a4   u0_a4    2015-12-16 14:04 com.android.dialer
drwxr-x--x u0_a1   u0_a1    2015-12-16 14:04 com.android.providers.calendar
drwxr-x--x u0_a2   u0_a2    2015-12-16 14:04 com.android.providers.contacts
drwxr-x--x u0_a5   u0_a5    2015-12-16 14:04 com.android.providers.media
```

In Listing 4.11, each directory in the listing represents the home directory of an app that is installed on the device. The home directory is where local data, such as databases, preferences, and cache information, is saved. Because the data is specific to the app, Android assigns permissions that prevent other apps from accessing local app data. The permissions for the directories in `/data/data` (`rwxr-x--x`) allow any app to enter the directory but not add or remove anything in the directory. Only the app that "owns" the directory can add or remove content from the directory.

Drilling down into an app directory a little deeper provides more detail on how Android protects files. Listing 4.12 shows the permissions of files in the `databases` directory of `/data/data/com.android.providers.contacts`.

Listing 4.12 File Permissions

```
root@generic_x86_64:/ # ls -l \
> data/data/com.android.providers.contacts/databases
-rw-rw---- u0_a2    u0_a2      348160 2016-03-24 20:42 contacts2.db
-rw-rw---- u0_a2    u0_a2           0 2016-03-24 20:42 contacts2.db-journal
-rw-rw---- u0_a2    u0_a2      348160 2015-12-16 14:04 profile.db
-rw-rw---- u0_a2    u0_a2       16928 2015-12-16 14:04 profile.db-journal
root@generic_x86_64:/ #
```

Notice that in Listing 4.12 the file permissions (`rw-rw----`) are even more restrictive than the directory permissions. The file permissions for every file in the `databases` directory ensure that no other app can read or write to the file.

File permissions are important when exploring a device with `adb` because when the `adb` shell is launched, it may be launched as an "unprivileged" user. In Linux terms, an unprivileged user means that the user account used to run the shell is not able to override any of the permissions of the files or directories on the device. If a file has permissions that prevent it from being read for all users except the app that owns the file, the `adb` shell is not able to read the file. This means that the `adb` shell is not able to perform tasks like read and write to databases in an app's home directory.

While the `adb` shell does tend to run as an unprivileged user on most devices, it is run as the "root" user on the emulator, or if the device is "rooted" (modified to allow apps and programs to run as root). On Linux systems, the root user can override file and directory permissions. It can be thought of as an administrative account with almost limitless access to the system and its contents.

When the `adb` shell is run as root, it has access to an app's private files and directories. This is important because it means that on an emulator or rooted device, the `adb` shell is able to access an app-private database. However, the `adb` shell is not able to access app-private databases on a device that is not rooted. Because most devices are not rooted, accessing a database on most devices can be problematic.

> **Note**
>
> All the `adb shell` commands listed in this chapter were run on either the Android emulator or a rooted device. The commands used to read and copy files from an app's home directory are not able to run on a non-rooted device/emulator due to a lack of permissions.

Finding a Database Location with `adb`

To connect to an app's SQLite database, the location of the SQLite database file must be known. Recall from Chapter 3 that SQLite stores an entire database in a single file (with the possibility of some temporary files used for transaction support). The database file is stored in the app's home directory.

An example of accessing an app's database would be accessing the database of one of Android's internal databases: the contacts database. Before the contacts database can be accessed, the location of the database file must be known. The database file location can be determined using `adb`.

As for most system-level databases, Android provides a content provider to access the contacts database. While the concept of content providers will be explored more deeply in later chapters, for now it is enough to know that a content provider provides a data abstraction layer and contains an **authority** that uniquely identifies the type of data in the `ContentProvider`. This authority is typically defined in a contract class that provides the public API for using a `ContentProvider`.

In the case of the contacts content provider, the contract class is `ContactsContract`. Examining the documentation for the `ContactsContract` class

(https://developer.android.com/reference/android/provider/ContactsContract.html#AUTHORITY) reveals that `ContactsContract` defines an `AUTHORITY` constant which has a value of `com.android.contacts`. We can use the value of `ContactsContract.AUTHORITY` along with `adb shell dumpsys` to find the location of the database that supports the contacts `ContentProvider`.

The `adb shell dumpsys` subcommand can be used to display information about the Android system. Listing 4.13 shows how to use `adb shell dumpsys` to get information about the registered content providers in the system as well as a snippet of the output from the command.

Listing 4.13 `adb shell dumpsys` Subcommand

```
bash-4.3$ adb shell dumpsys activity providers

ACTIVITY MANAGER CONTENT PROVIDERS (dumpsys activity providers)

…

* ContentProviderRecord{2f0e81e u0 com.android.providers.contacts/.Contacts
➥Provider2}

    package=com.android.providers.contacts process=android.process.acore

    proc=ProcessRecord{ad8d91a 11766:android.process.acore/u0a2}

    launchingApp=ProcessRecord{ad8d91a 11766:android.process.acore/u0a2}

    uid=10002 provider=android.content.ContentProviderProxy@c8028ff

    authority=contacts;com.android.contacts

…


bash-4.3$
```

The output from the `adb shell dumpsys` subcommand provides all the content provider information for the entire device, listing all the providers provided by the various apps that are installed. To find the correct content provider entry, the `authority` field from the `adb shell dumpsys` output needs to be examined. Specifically, the `ContentProviderRecord` with an `authority` that includes `com.android.contacts` provides the details needed to find the contacts SQLite database file. The correct `ContentProviderRecord` is highlighted in Listing 4.13.

In most cases, the `adb shell dumpsys` command prints a lot of information that needs to be searched to find the correct `authority` string. After the `ContentProviderRecord` is identified, the package of the app that provides the content provider can be resolved. In the case of the contacts content provider, the package name for the app that provides the content provider is `com.android.providers.contacts`. Once the package name is known, finding the actual SQLite database file is easy since it is in the home directory of the app. In this case, the home directory is `/data/data/com.android.providers.contacts`.

Listing 4.14 shows using the `cd` command to change to the contacts provider app home directory, then using the `ls` command to perform a directory listing.

Listing 4.14 Home Directory Listing

```
root@generic_x86_64:/ # cd /data/data/com.android.providers.contacts
root@generic_x86_64:/data/data/com.android.providers.contacts # ls
cache
code_cache
databases
files
shared_prefs
root@generic_x86_64:/data/data/com.android.providers.contacts #
```

The contents of the contacts content provider app can be seen in Listing 4.14. The directory `/data/data/com.android.providers.contacts` contains the following entries:

- `cache`
- `code_cache`
- `databases`
- `files`
- `shared_prefs`

The `cache` and `code_cache` directories are used to store temporary information. The `files` directory is where app-specific files get stored. `shared_prefs` contains the XML files used to persist preferences in Android, and the `databases` directory is used to store the SQLite database files for the app. Recall from the discussion of `SQLiteOpenHelper` that the name of the database is specified when opening the database. The file in the database directory matches the file names used in the `SQLiteOpenHelper` class. Listing 4.15 shows a directory listing of the `databases` directory for the contacts provider app.

Listing 4.15 `databases` Directory Listing

```
root@generic_x86_64:/data/data/com.android.providers.contacts# ls databases
contacts2.db
contacts2.db-journal
profile.db
profile.db-journal
root@generic_x86_64:/data/data/com.android.providers.contacts#
```

In Listing 4.15, there are two database files in the `databases` directory that make up the contacts database, `contacts2.db` and `profile.db`. Each database file also has a journal file to support transactions. With the knowledge of the location of the database file, the `sqlite3` command can now be used to connect to the database in the shell.

Connecting to a Database with `sqlite3`

The `sqlite3` command is part of the SQLite package and is included in the Android SDK. To use the `sqlite3` command to connect to a database, simply pass the database file name as an argument to `sqlite3`. Listing 4.16 shows how to use the `sqlite3` command to connect to the contacts database from Listing 4.15.

Listing 4.16 Connecting to the Contacts Database

```
root@generic_x86_64:/data/data/com.android.providers.contacts # sqlite3 \
> databases/contacts2.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

Once the connection to the database has been made, `sqlite3` displays a prompt. Notice in Listing 4.16 that the `sqlite3` program provides a hint at how to get help: typing `.help` and pressing Enter. Listing 4.17 shows what is printed when the `.help` command is issued to `sqlite3`.

Listing 4.17 Issuing `.help` to `sqlite3`

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.binary on|off         Turn binary output on or off.  Default OFF
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dbinfo ?DB?           Show status information about the database
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.eqp on|off            Enable or disable automatic EXPLAIN QUERY PLAN
.exit                  Exit this program
.explain ?on|off?      Turn output mode suitable for EXPLAIN on or off.
                         With no args, it turns EXPLAIN on.
```