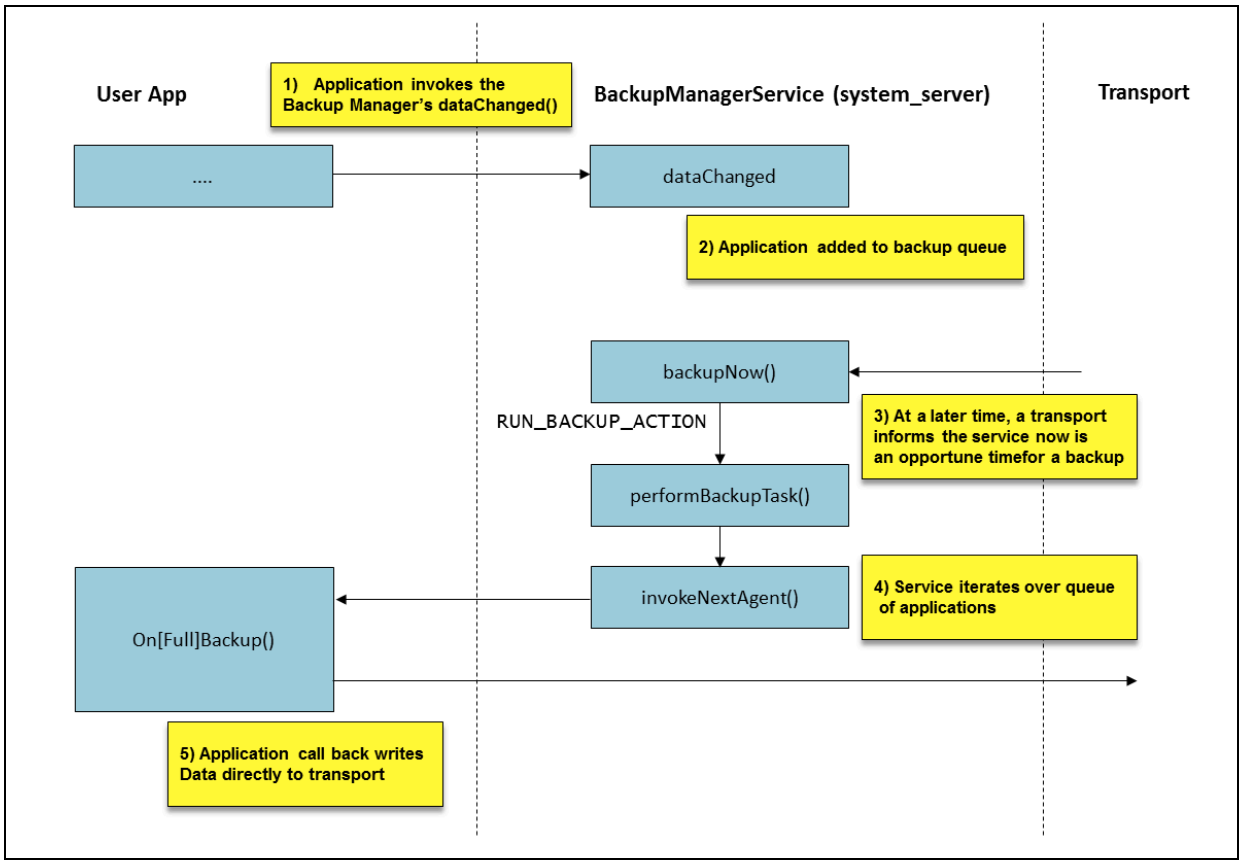
Application Backup & Restore

Just as humans grapple with sickness, Operating Systems face the risk of data corruption, or outright loss. Backup and Restore is therefore an important functionality which an operating system needs to provide. Applications need an ability to save and recover their configuration and data, and power users require a similar ability to backup the entire device to a well known, bootable configuration or a system checkpoint which can be rolled back to in case of calamity.

Indeed, as of API level 8, Android provides Applications with the `BackupManagerService`, a framework service which provides both per-application backups, as well as full backups of all apps. The internals of the framework service, including the Application Programming Interface it provides, is covered (along with the rest of the framework services) in Volume II. The backup architecture is quite elegant, delegating the responsibility of selecting which data is to be backed up to the application: The application notifies the backup manager when data has changed, and the backup manager adds the application to a queue.

Figure 3-4: A simplified view of the Android backup architecture



At some later time, when the `BackupManagerService` gets a request to actually perform a backup, it creates a **backup set**, grouping together the one or more applications that were queued. For each application, it invokes the `onBackup()` callback. The `BackupService` passes the application a file descriptor in the callbacks, which the application is expected to use in order to write out (or read from) the backup data. The descriptor provided is connected to a **transport**, to which the application remains entirely oblivious. Data is written and read to the transport while leaving its implementation opaque - Data can be backed up either locally, or to "the Cloud" (i.e. Google's servers, or the device vendor's), but the choice of where to back up to remains at the system (or vendor) level. The common transports are shown in table 3-7:

Table 3-7: Android Transports

Transport	Backs up to
`com.google.android.backup/.BackupTransportService`	Google's servers. Application needs a special API key to use this service
`com.android.server.enterprise/.EdmBackupTransport`	Enterprise backup, for managed devices
`android/com.android.internal.backup.LocalTransport`	Local backup, to device

Command line tools

From the perspective of the power user, there's a far simpler interface to backup and restore, in the form of two Dalvik upcall scripts, the `bmgr` and `bu` utilities. Both utilities require Java to facilitate communication with the `BackupManagerService`, which they perform over Binder (as discussed in Chapter 7). The `bmgr` utility is well documented, and explains its usage in detail when invoked with no arguments. A summary of its arguments is shown in Table 3-8:

Table 3-8: Commands and arguments understood by the `bmgr` upcall script

Command	Purpose
`backup` *`package`*	Mark package to be backed up on next run
`enable` *`0|1`*	Enable/disable the backup mechanism
`enabled`	Report if backup mechanism is enabled or disabled
`list` *`transports`*	List available transports, * specifying default (q.v. Table 3-7)
`list` *`sets`*	List restore sets
`transport` *`transportName`*	Set default transport
`restore` *`set [App]`*	Restore from a specific set - all apps, or only App specified.
`run`	Perform pending backups now
`wipe` *`transportName package`*	Erase all backups of *package* from *transportName*
`fullbackup` *`package`*	Perform a full backup of specified package

By contrast, the `bu` utility is entirely undocumented, and provides no user facing output, preferring instead to use the Android logging system. expects only one argument - `backup` or `restore`, but can handle quite a few switches when backing up. The switches expected by `bu` are shown in Table 3-9, with the defaults in **bold**:

Table 3-9: Switches understood by `bu backup`

Switch	Purpose
`-[no]apk`	Save or omit application `.apk` files
`-[no]obb`	Save or omit application opaque binary blobs (`.obb`) files
`-[no]shared`	Save or omit shared resources
`-[no]`**`system`**	Save or omit system applications in full backups
`-[no]widgets`	Save or omit widgets (default: -nowidgets)
`-[no]`**`compress`**	Compress backup
`-all`	Backup everything (requires user confirmation)

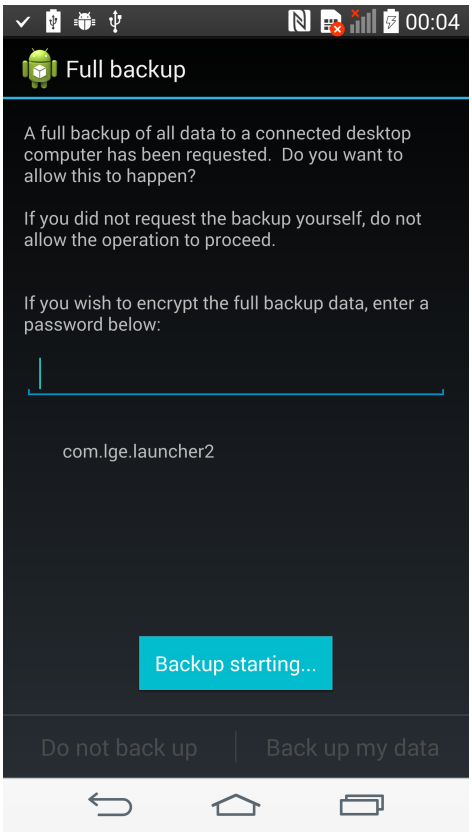
If the switches seem vaguely familiar, it's because they are the same as those passed to `adb backup` (though the latter does not advertise `-nocompress` as an option). Backups through `adb` are just direct invocations of the `bu` upcall script, which helps explain why it's not as user-friendly as `bmgr`.

Local backups

Using `adb backup -all`, triggers a full backup of all applications. Doing so causes the `bu` utility to call the `BackupManagerService`'s `fullBackup()` method, which pops up a customizable UI notification to the user.

The default notification UI activity is hardcoded to `com.android.backupconfirm`, and shown in Figure 3-5. Using a UI requires the device to be unlocked, adding a measure of security for users, by mitigating the chance a device could be taken for a minute or two, backed up and returned to the unwitting user. Another measure of security offers the user a chance to cancel the backup, as well as enter a password.

Figure 3-5: The default Backup UI (LG G3 running KitKat)



If the user approves the backup operation, a toast notification informs that the backup started, and the current package progress is displayed.

When backing up to a connected host, `adb` connects the other end of the transport file descriptor to a local file on the host, specified by the `-f` switch, or simply the `backup.ab` default. The backup file uses a proprietary format, which differs slightly if the backup is encrypted or not. The format's only documentation is embedded in the source of the `BackupManagerService` class, but this provides comprehensive detail, as shown in Listing 3-bufor:

Listing 3-bufor: The format of an Android backup file

```
// Write the global file header.  All strings are UTF-8 encoded; lines end
// with a '\n' byte.  Actual backup data begins immediately following the
// final '\n'.
//
// line 1: "ANDROID BACKUP"
// line 2: backup file format version, currently "2"
// line 3: compressed?  "0" if not compressed, "1" if compressed.
// line 4: name of encryption algorithm [currently only "none" or "AES-256"]
//
// When line 4 is not "none", then additional header data follows:
//
// line 5: user password salt [hex]
// line 6: master key checksum salt [hex]
// line 7: number of PBKDF2 rounds to use (same for user & master) [decimal]
// line 8: IV of the user key [hex]
// line 9: master key blob [hex]
//     IV of the master key, master key itself, master key checksum hash
//
// The master key checksum is the master key plus its checksum salt, run through
// 10k rounds of PBKDF2.  This is used to verify that the user has supplied the
// correct password for decrypting the archive:  the master key decrypted from
// the archive using the user-supplied password is also run through PBKDF2 in
// this way, and if the result does not match the checksum as stored in the
// archive, then we know that the user-supplied password does not match the
// archive's.
```

Experiment: Examining Android Backups

The Android backup file header is easy to figure out using Listing 3-bufor, but its contents are compressed by default. Using the semi-documented `-nocompress`, which is supported by the `bu` upcall script but not readily advertised by `adb`, you can create an uncompressed backup:

Output 3-15: Creating and inspecting an uncompressed backup

```
morpheus@Forge (~) % adb backup -nocompress -all
# UI is displayed on device...
Now unlock your device and confirm the backup operation.
morpheus@Forge (~) % ls -l backup.ab
-rw-r-----  1 morpheus  staff  17158168 Jan  1 23:37 backup.ab
morpheus@Forge (~) % head -6 backup.ab
ANDROID BACKUP   # MAGIC
3                # Version (3 = L)
0                # Compression (0 = False)
none             # Encryption (none)
apps/android/_manifest000600 01750017500000000036240010767 0ustar001
android
..
```

The header is straightforward enough, but what of the actual backup contents? The first line looks suspiciously like meta data. We therefore strip the header, and try our luck with `file(1)`:

Output 3-16: Stripping the header from an Android archive

```
# The header was four lines long, so start as of line 5...
morpheus@Forge (~) % tail +5 backup.ab > a.ab
# Attempt to auto identify the file..
morpheus@Forge (~) % file a.ab
a.ab: POSIX tar archive
# Check file contents:
morpheus@Forge (~) % tar tvf a.ab | more
-rw-------  0 1000   1000     1940 Dec 31  1969 apps/android/_manifest
-rw-------  0 1000   1000       99 Jan  1 21:29 apps/android/r/wallpaper_info.xml
-rw-------  0 1000   1000     1961 Dec 31  1969 apps/com.android.browser.provider/_manifest
...
```

And thus we see that Android backups, internally, are nothing more than good ol' UN*X `tar` archives. Using compression applies the Deflate algorithm after the `tar`.

If you do use encryption, the header size and complexity both increase. The following shows the header of the same archive, when compressed and encrypted with "password":

Output 3-17: Examining an encrypted backup

```
# Note this time, no nocompress implies compress by default
morpheus@Forge (~) % adb backup -all
# UI is displayed on device... enter "password"
Now unlock your device and confirm the backup operation.
# File is significantly smaller this time
morpheus@Forge (~) % ls -l backup.ab
-rw-r-----  1 morpheus  staff  7518645 Jan  1 23:50 backup.ab
morpheus@Forge (~) % head -9 backup.ab
ANDROID BACKUP
3
1         # This time, compressed
AES-256   # Encryption algorithm
FBAEB6CF..# 128 hex digits = 512-bit salt
98A4BF42..# 128 hex digits = 512-bit master key checksum
10000     # Number of PBKDF2 key derivations
0B0D638F9856C5D4F040399AB28A0C5F # Random IV (32 hex digits = 128bit)
E8AD4E9948F356E15A1E41AA265660.. # 192 hex digits = 768 bit Master key blob
```

Monitoring backup operations

The `BackupManagerService` stores its configuration in two main locations:

- **The system secure settings:** common to all Android framework services, and accessible via the `Settings` class. The manager defines the following settings (with constants in the `Settings` class identical to the string values, uppercased:

Setting	Purpose
backup_enabled	Is backup enabled? Equivalent to `bmgr enable`
backup_transport	Default transport. Settable by `bmgr transport ..`
backup_provisioned	Is backup provisioned? Useful for managed devices
backup_auto_restore	Can application data be automatically restored?

- **The `/data/backup` directory:** containing the list of transports (as directories), and backup queues.

Normally, you won't need to go into the directory or settings yourself, as you can use `bmgr` (or `settings`) to toggle the settings, and `dumpsys backup` to get verbose information on the queues. The annotated output is shown below:

Output 3-18: Using `dumpsys` to display the backup status

```
shell@flounder:/ $ dumpsys backup
Backup Manager is disabled / provisioned / not pending init
Auto-restore is enabled
Last backup pass started: 0 (now = 1420171109885)
  next scheduled: 0
# List of transports. Google cloud is default, but requires account
Available transports:
  * com.google.android.backup/.BackupTransportService
       destination: Need to set the backup account
       intent: Intent { act=com.google.android.backup.SetBackupAccountActivity }
    android/com.android.internal.backup.LocalTransport
       destination: Backing up to debug-only private cache
       intent: null
Pending init: 0
# List of applications that can request backup, sorted by AID:
Participants:
  uid: 1000
    com.android.providers.settings
    android
  uid: 1027
    com.android.nfc
...
# Ancestral refers to full backups, which serve as a point of
# departure for incremental backup/restore operations
Ancestral packages: none
Ever backed up: 0
Pending key/value backup: 13
    BackupRequest{pkg=com.google.android.gm}
    BackupRequest{pkg=com.google.android.talk}
    ..
Full backup queue:47
# Last backup : package name
    0 : com.android.providers.downloads.ui
    0 : com.android.externalstorage
    0 : com.google.android.nfcprovision
    ..
```

Experiment: Delving deeper into backups

To get a better grip of backups on Android, have a look at the /data/backup directory, which is where the `BackupManagerService` maintains its metadata. As root, you should see something similar to the following:

Output 3-19: The /data/backup directories

```
root@flounder:/data/backup # ls -l
drwx------ system   system       ...  com.android.internal.backup.LocalTransport
drwx------ system   system       ...  com.google.android.backup.BackupTransportServic
-rw------- system   system  1881 ...  fb-schedule
drwx------ system   system       ...  pending
```

Getting the default transport is a simple matter, either by calling on the `bmgr` upcall script, or querying the value directly from the system's secure settings:

Output 3-20: Finding the default transport

```
root@flounder:/data/backup # bmgr list transports
  * com.google.android.backup/.BackupTransportService
    android/com.android.internal.backup.LocalTransport
root@flounder:/data/backup # settings get secure backup_transport
com.google.android.backup/.BackupTransportService
```

The backup queue is maintained in memory, but also written to the `pending` directory, as a journal-*xxxx*.tmp temporary file, to provide recovery in case the backup service itself crashes. The file format is simply a concatenation of package names to be backed up. Since the package names are preceded by a length byte and NULL terminated, use `cat -tv` to display this file:

Output 3-21: Displaying the backup journal

```
root@flounder:/data/backup # cat  -tv pending/journal-168056423.tmp
^@$com.android.providers.userdictionary^@'com.google.android.googlequicksearchbox^@"
com.google.android.marvin.talkback^@$com.google.android.inputmethod.latin^@^Ucom.
google.android.gm^@^Ocom.android.nfc^@^Scom.android.vending^@^Gandroid^@^Wcom.google.
android.talk^@^_com.android.sharedstoragebackup^@)com.google.android.apps.genie.
geniewidget^@^[com.google.android.calendar^@^^com.android.providers.settingsroot@
```

Lastly, the fb-schedule file schedule is used to maintain a list of all installed packages which are backup eligible (i.e. declared a `BackupAgent` in their manifest, as we discuss in Volume II, and is well documented in the Android Developer Website). The file format is very similar to that of the journal (albeit with a few more fields), but this is where `dumpsys` gets handy (which is even more useful since you don't need root privileges to use it)