Creating Live Folders

To enable in-application live folder creation within your application, you need

- An application with data exposed as a content provider
- An application that acts as a content type handler for the type of data that is exposed in the live folder (for example, `VIEW` field notes)
- To implement an `Activity` class to handle live folder creation
- To update the application's content provider interface to handle live folder queries
- To configure the application's Android manifest file for live folders

Now let's look at some of these requirements in more detail.

Creating a Live Folder Activity

An application that supports live folders must include a live folder creation activity. This activity is launched anytime the application reacts to the intent action `ACTION_CREATE_LIVE_FOLDER`. The live folder creation activity is responsible for one thing: responding with a specific instance of a live folder configuration. If you recall how the `startActivityForResult()` method works, this is exactly how the live folder creation activity is called. The activity needs to retrieve the incoming intent that performed the live folder creation request, craft a new live folder (as an `Intent` object with suitable extras to specify the live folder configuration details) and set the Activity result using the `setResult()` method. The `setResult()` method parameters can be used to communicate whether or not the live folder creation was successful as the `resultCode` parameter and pass back the specific instance of the live folder as the accompanying result Intent `data` parameter.

Sounds a tad complex, eh? Well, let's look at a specific example. Here is the implementation of the live folder creation activity for the field notes application:

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.LiveFolders;

public class SimpleLiveFolderCreateActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        final Intent intent = getIntent();
        final String action = intent.getAction();
        if (LiveFolders.ACTION_CREATE_LIVE_FOLDER.equals(action)) {
            final Intent baseIntent = new Intent(Intent.ACTION_VIEW,
                SimpleFieldnotesContentProvider.CONTENT_URI);
            final Intent resultIntent = new Intent();
            resultIntent.setData(
                SimpleFieldnotesContentProvider.LIVE_URI);
```

```
            resultIntent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_NAME,
                getResources().getString(R.string.livefolder_label));
            resultIntent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_ICON,
                Intent.ShortcutIconResource.fromContext(
                this, R.drawable.foldericon));
            resultIntent.putExtra(
                LiveFolders.EXTRA_LIVE_FOLDER_DISPLAY_MODE,
                LiveFolders.DISPLAY_MODE_LIST);
            resultIntent.putExtra(
                LiveFolders.EXTRA_LIVE_FOLDER_BASE_INTENT,
                baseIntent);
            setResult(RESULT_OK, resultIntent);
        } else {
            setResult(RESULT_CANCELED);
        }
        finish();
    }
}
```

As you can see, the `SimpleLiveFolderCreateActivity` has a very short lifespan. It waits for `ACTION_CREATE_LIVE_FOLDER` requests and then crafts the appropriate `Intent` object to return as part of the activity result. The most important code in this activity is the code that creates the new intent called `resultIntent`. This `Intent` object contains all the configuration details for the new live folder instance. The `setData()` method is used to supply the live Uri (the `Uri` to query to fill the folder with data).

Several extras are set to provide the live folder instance with a `label` and `icon`, as well as specify the display mode of the live folder. Live folders have several canned display modes: The `DISPLAY_MODE_LIST` value causes all live folder content to display in `ListView` control (ideal for text content) and the `DISPLAY_MODE_GRID` displays live folder content in a `GridView` control—more appropriate if the live folder contents are graphics. Finally, the base `Intent` object for each live folder item is set. In this case, the base intent has an action type of `VIEW`, as you might expect, and therefore is compatible with the content type handler technique. For more information on the configuration details that can be applied to a live folder, see the Android SDK documentation for the `android.provider.LiveFolders` package.

Handling Live Folder Content Provider Queries

Each time the user opens the live folder, the system performs a content provider query. Therefore, the application's content provider interface needs to be updated to handle queries to fill the live folder with data. As with search suggestions, you need to define a projection in order to map the content provider data columns to those that the live folder expects to use to fill the list or grid (depending on the display mode) within the folder.

For example, the following code defines a project to map the field notes' unique identifiers and titles to the ID and name fields for the live folder items:

```
private static final HashMap<String, String>
FIELDNOTES_LIVE_FOLDER_PROJECTION_MAP;
static {
    FIELDNOTES_LIVE_FOLDER_PROJECTION_MAP = new HashMap<String, String>();
    FIELDNOTES_LIVE_FOLDER_PROJECTION_MAP
        .put(LiveFolders._ID, _ID + " AS "
        + LiveFolders._ID);
    FIELDNOTES_LIVE_FOLDER_PROJECTION_MAP.put(
        LiveFolders.NAME, FIELDNOTES_TITLE
        + " AS " + LiveFolders.NAME);
}
```

Whenever the live folder is opened by the user, the system executes a query on the `Uri` provided as part of the live folder configuration. Don't forget to define the live `Uri` address and register it in the content provider's `UriMatcher` object (using the `addURI()` method). For example, the field notes application used the `Uri`:

```
content:// com.androidbook.simplelivefolder.
SimpleFieldnotesContentProvider/fieldnotes/live
```

By providing a special live folder `Uri` for the content provider queries, you can simply update the content provider's query method to handle the specialized query, including building the projection, performing the appropriate query, and returning the results for display in the live folder. Let's take a closer look at the field notes content provider `query()` method:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    queryBuilder.setTables(SimpleFieldnotesDatabase.FIELDNOTES_TABLE);
    int match = sURIMatcher.match(uri);
    switch (match) {
    case FIELDNOTE_ITEM:
        String id = uri.getLastPathSegment();
        queryBuilder.appendWhere(_ID + "=" + id);
        break;
    case FIELDNOTES_LIVE:
        queryBuilder.setProjectionMap(
            FIELDNOTES_LIVE_FOLDER_PROJECTION_MAP);
        break;
    default:
        throw new IllegalArgumentException("Invalid URI: " + uri);
    }
    SQLiteDatabase sql = database.getReadableDatabase();
    Cursor cursor = queryBuilder.query(sql,
```

```
        projection, selection, selectionArgs, null,
        null, sortOrder);
    cursor.setNotificationUri(getContext().getContentResolver(), uri);
    return cursor;
}
```

This `query()` method implementation handles both regular content queries and special live folder queries (those that come in with the live `Uri`). When the live folder query occurs, we simply use the handy `setProjectionMap()` method of the `QueryBuilder` object to set and execute the query as normal.

Configuring the Android Manifest File for Live Folders

Finally, the live folder Activity class needs to be registered within the application Android manifest file with an intent filter with the `CREATE_LIVE_FOLDER` action. For example, here is an excerpt from the field notes Android manifest file that does just that:

```
<activity
    android:name="SimpleLiveFolderCreateActivity"
    android:label="@string/livefolder_label"
    android:icon="@drawable/foldericon">
    <intent-filter>
        <action
            android:name="android.intent.action.CREATE_LIVE_FOLDER" />
        <category
            android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

This type of `Activity` registration should look familiar. The `SimpleLiveFolderCreateActivity` class is responsible for handling the `CREATE_LIVE_FOLDER` intent (as dictated by the intent filter). You can also set the live folder's text `label` and `icon` using the attributes for the creation activity.

Note

The icon and text label shown in the Live Folder picker is set separately from the icon and label shown for a given instance of a live folder. You can set the information for the Live Folder in the picker using the `android:label` and `android:icon` attributes of the `<activity>` tag corresponding to the activity that handles the intent filter for `android.intent.action.CREATE_LIVE_FOLDER` action in the Android manifest file. The icon and label for each Live Folder instance on the Home screen (or other live folder host) is set as part intent extra fields when your `Activity` class handles the action `LiveFolders.ACTION_CREATE_LIVE_FOLDER`.

Installing a Live Folder

After the application is capable of handling the creation of a live folder, your development work is done. As a user, you can install a live folder on the Home screen using the following steps:

1. Long-press on an empty space in the Home Screen.
2. From the menu, choose the Folders option, as shown in Figure 22.8 (left).

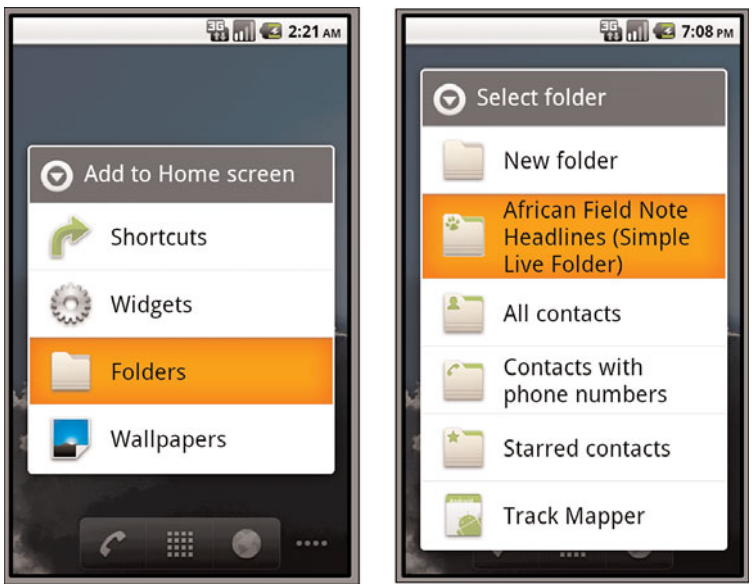

Figure 22.8 Installing a live folder on the Home screen.

3. From the Select folder menu, choose the folder to add, as shown in Figure 22.8 (right).
4. The live folder is now visible on your Home Screen, as shown in Figure 22.9.
5. If you click on the live folder, it opens and shows its contents (as dictated by the developer). Choosing an item from the live folder launches the underlying application. For example, choosing one of the field note titles from the list launches the `Activity` used to view that specific item and its details, as shown in Figure 22.10.

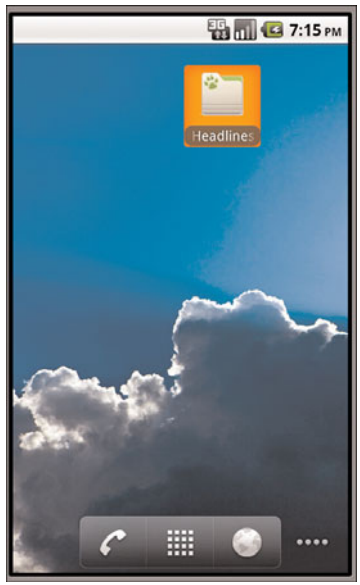

Figure 22.9 A live folder on the Home screen.

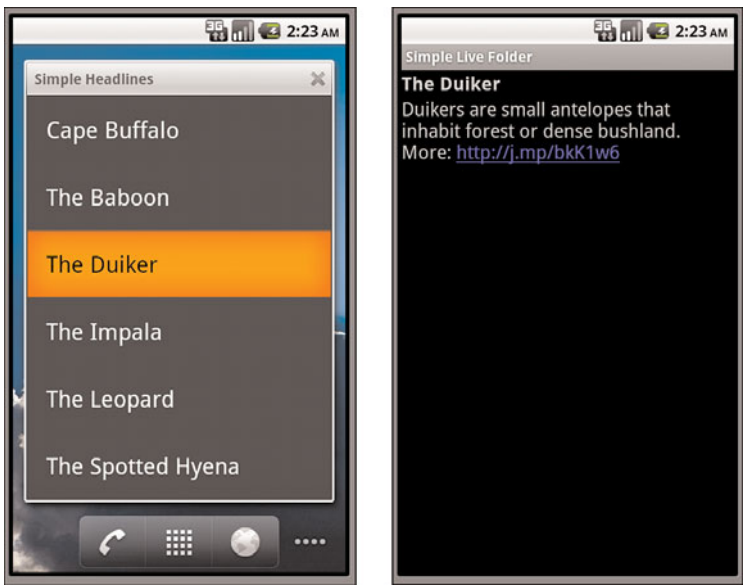

Figure 22.10 Choosing a specific field note from the live folder (left) launches the application to display its content (right).

Summary

The Android platform provides a number of ways to integrate your applications tightly into the operating system, enabling you to extend your reach beyond traditional application boundaries. In this chapter, you learned how to extend your application by creating simple App Widgets, live wallpapers, live folders, and more. You also learned how to enable search within your applications, as well as how to include your application content in global searches.

References and More Information

Our Series of Articles on App Widgets:
http://j.mp/a8mpdH
Android Dev Guide: App Widgets:
http://developer.android.com/guide/topics/appwidgets/index.html
Android Dev Guide: Search:
http://developer.android.com/guide/topics/search/index.html
Android Technical Articles: Live Wallpapers:
http://developer.android.com/resources/articles/live-wallpapers.html
Android Technical Articles: Live Folders:
http://developer.android.com/resources/articles/live-folders.html

This page intentionally left blank

23

Managing User Accounts and Synchronizing User Data

Android is cloud-friendly. Android applications can integrate tightly with remote services, helping users transition seamlessly. Android applications can synchronize data with remote cloud-based (Internet) services using sync adapters. Developers can also take advantage of Android's cloud-based backup service to protect and migrate application data safely and effectively. In this chapter, you learn about the account and synchronization features to sync data to built-in applications as well as protect application data using the backup and restore features available in Android.

Managing Accounts with the Account Manager

From a user perspective, the Android 2.0 platform introduced many exciting new device features. For instance, the user can register and use multiple accounts for email and contact management. This feature was provided through a combination of new synchronization and account services that are also available to developers. Although you can use the account and synchronization packages with any kind of data, the intention seems to be to provide a way for developers and companies to integrate their business services with the system that synchronizes data to the built-in Android applications.

Android user accounts are manipulated using the classes available within the `android.accounts` package. This functionality is primarily designed for accounts with services that contain contact, email, or other such information in them. A good example of this type of online service is a social networking application that contains friends' contact information, as well as other relevant information such as their statuses. This information is often delivered and used on an Android device using the synchronization service (we talk more about synchronization later in the chapter).

First, we talk about accounts. Accounts registered with the Android account manager should provide access to the same sort of information—contact information, for the most part. Different accounts can be registered for a given user using the Android `AccountManager` class. Each account contains authentication information for a service,

usually credentials for a server account somewhere online. Android services, such as the synchronization services built-in to the platform, can access these accounts, mining them for the appropriate types of information (again, primarily contact details, but also other bits of data such as social networking status).

Let's look at how using account information provided via the `AccountManager` and `Account` classes works. An application that needs to access the server can request a list of accounts from the system. If one of the accounts contains credentials for the server, the application can request an authentication token for the account. The application would then use this token as a way to log in to the remote server to access its services. This keeps the user credentials secure and private while also providing a convenience to the user in that they only need to provide their credentials once, regardless of how many applications use the information. All these tasks are achieved using the `AccountManager` class. A call to the `getAccountByType()` method retrieves a list of accounts and then a call to the `getAuthToken()` method retrieves the token associated with a specific account, which the application can use to communicate with a password-protected resource, such as a web service.

On the other side of this process, authenticating credentials against the back-end server are the account providers. That is, the services that provide users with accounts and with which user information is authenticated so the applications can get the auth tokens. In order to do all of this (handle system requests to authenticate an `Account` object against the remote server), the account provider must implement an account authenticator. Through the authenticator, the account provider requests appropriate credentials and then confirms them with whatever account authentication operations are necessary—usually an online server. To implement an account authenticator, you need to make several modifications to your application. Begin by implementing the `AbstractAccountAuthenticator` class. You also need to update the application's Android Manifest file, provide an authenticator configuration file (XML), and provide an authenticator preference screen configuration in order to make the authentication experience as seamless as possible for the user.

Tip

Learn more about creating system-wide accounts in the Android SDK documentation for the `AbstractAccountAuthenticator` class. Learn more about using accounts in the Android SDK documentation for the `AccountManager` class.

Synchronizing Data with Sync Adapters

The synchronization feature available in the Android SDK requires the use of the accounts classes we talked about earlier. This service is principally designed to enable syncing of contact, email, and calendar data to the built-in applications from a back-end datastore—you're "adapting" back-end server data to the existing content providers. That is, the service is not generally used for syncing data specific to your typical Android application. In theory, applications could use this service to keep data in sync, but they might be better served by implementing synchronization internally. You could do this using the

`AlarmManager` class to schedule systematic data synchronization via the network, perhaps using an Android service.

If, however, you are working with data that is well suited to syncing to the internal applications, such as contacts or calendar information that you want to put in the built-in applications and content providers, implementing a sync adapter makes sense. This enables the Android system to manage synchronization activities.

The account service must provide the sync adapter by extending the `AbstractThreadedSyncAdapter` class. When the sync occurs, the `onPerformSync()` method of the sync adapter is called. The parameters to this method tell the adapter what account (as defined by the `Account` parameter) is being used, thus providing necessary authentication tokens (auth token, for short) for accessing protected resources without the need for asking the user for credentials. The adapter is also told which content provider to write the data to and for which authority, in the content provider sense, the data belongs to.

In this way, synchronization operations are performed on their own thread at a time requested by the system. During the sync, the adapter gets updated information from the server and synchronizes it to the given content provider. The implementation details for this are flexible, and up to the developer.

Tip

Learn more about creating sync adapters by checking out the Sync Adapter sample application on the Android developer website: http://developer.android.com/resources/samples/SampleSyncAdapter/.

Using Backup Services

Android backup services were introduced in Android 2.2 (API Level 8). Applications can use the backup system service to request that application data such as shared preferences and files be backed up or restored. The backup service handles things from there, sending or retrieving the appropriate backup archives to a remote backup service.

Backup services should not be used for syncing application content. Backup and restore operations do not occur on demand. Use a synchronization strategy such as the sync adapter discussed earlier in this chapter in this case. Use Android backup services only to back up important application data.

Tip

Many of the code examples provided in this section are taken from the SimpleBackup application. The source code for this application is provided for download on the book website. Also, you need to use the `adb bmgr` command to force backups and restores to occur. For more information on `adb`, see Appendix C, "The Android Debug Bridge Quick-Start Guide."

Choosing a Remote Backup Service

One of the most important decisions when it comes to backing up application data is deciding where to back it up to. The remote backup service you choose should be secure, reliable, and always available. Many developers will likely choose the solution provided by Google: Android Backup Service.

> **Note**
>
> Other third-party remote backup services might be available. If you want complete control over the backup process, you might want to consider creating your own. However, this is beyond the scope of this book.

In order for your application to use Android Backup Service, you must register your application with Google and acquire a unique backup service key for use within the application's manifest file.

You can sign up for Google's backup service at the Android Backup Service website: http://code.google.com/android/backup/signup.html.

> **Warning**
>
> Backup services are available on most, but not all, Android devices running Android 2.2 and higher. The underlying implementation might vary. Also, different remote backup services might impose additional limitations on the devices supported. Test your specific target devices and backup solution thoroughly to determine that backup services function properly with your application.

Registering with Android Backup Service

After you have chosen a remote backup service, you might need to jump through a few more hoops. With Google's Android Backup Service, you need to register for a special key to use. After you've acquired this key, you can use it within your application's manifest file using the `<meta-data>` tag within the `<application>` block, like this:

```
<meta-data android:name="com.google.android.backup.api_key"
    android:value="KEY HERE" />
```

Implementing a Backup Agent

The backup system service relies upon an application's backup agent to determine what application data should be archived for backup and restore purposes.

Providing a Backup Agent Implementation

Now it's time to implement the backup agent for your particular application. The backup agent determines what application data to send to the backup service. If you only want to back up shared preference data and application files, you can simply use the `BackupAgentHelper` class.

Tip

If you need to customize how your application backs up its data, you need to extend the `BackupAgent` class, which requires you to implement two callback methods. The `onBackup()` method is called when your application requests a backup and provides the backup service with the appropriate application data to back up. The `onRestore()` method is called when a restore is requested. The backup service supplies the archived data and the `onRestore()` method handles restoring the application data.

Here is a sample implementation of a backup agent class:

```
public class SimpleBackupAgent extends BackupAgentHelper {
    @Override
    public void onCreate() {
        // Register helpers here
    }
}
```

Your application's backup agent needs to include a backup helper for each type of data it wants to back up.

Implementing a Backup Helper for Shared Preferences

To back up shared preferences files, you need to use the `SharedPreferencesBackupHelper` class. Adding support for shared preferences is very straightforward. Simply update the backup agent's `onCreate()` method, create a valid `SharedPreferencesBackupHelper` object, and use the `addHelper()` method to add it to the agent:

```
SharedPreferencesBackupHelper prefshelper = new
SharedPreferencesBackupHelper(this,
    PREFERENCE_FILENAME);
addHelper(BACKUP_PREFERENCE_KEY, prefshelper);
```

This particular helper backs up all shared preferences by name. In this case, the `addHelper()` method takes two parameters:

- A unique name for this helper (in this case, the backup key is stored as a `String` variable called `BACKUP_PREFERENCE_KEY`).
- A valid `SharedPreferencesBackupHelper` object configured to control backups and restores on a specific set of shared preferences by name (in this case, the preference filename is stored in a `String` variable called `PREFERENCE_FILENAME`).

That's it. In fact, if your application is only backing up shared preferences, you don't even need to implement the `onBackup()` and `onRestore()` methods of your backup agent class.

Tip

Got more than one set of preferences? No problem. The constructor for `SharedPreferencesBackupHelper` can take any number of preference filenames. You still need only one unique name key for the helper.

Implementing a Backup Helper for Files

To back up application files, use the `FileBackupHelper` class. Files are a bit trickier to handle than shared preferences because they are not thread-safe. Begin by updating the backup agent's `onCreate()` method, create a valid `FileBackupHelper` object, and use the `addHelper()` method to add it to the agent:

```
FileBackupHelper filehelper = new FileBackupHelper(this, APP_FILE_NAME);
addHelper(BACKUP_FILE_KEY, filehelper);
```

The file helper backs up specific files by name. In this case, the `addHelper()` method takes two parameters:

- A unique name for this helper (in this case, the backup key is stored as a `String` variable called `BACKUP_FILE_KEY`).
- A valid `FileBackupHelper` object configured to control backups and restores on a specific file by name (in this case, the filename is stored in a `String` variable called `APP_FILE_NAME`).

Tip

Got more than one file to back up? No problem. The constructor for `FileBackupHelper` can take any number of filenames. You still need only one unique name key for the helper. The services were designed to back up configuration data, not necessarily all files or media. There are currently no guidelines for the size of the data that can be backed up. For instance, a book reader application might back up book titles and reading states, but not the book contents. Then, after a restore, the data could be used to download the book contents again. To the user, the state appears the same.

You also need to make sure that all file operations within your application are thread-safe as it's possible a backup will be requested while a file is being accessed. The Android website suggests the following method for defining a lock from a simple `Object` array within your `Activity`, as follows:

```
static final Object[] fileLock = new Object[0];
```

Use this lock each and every time you are performing file operations, either in your application logic, or within the backup agent. For example:

```
synchronized(fileLock){
    // Do app logic file operations here
}
```

Finally, you need to override the `onBackup()` and `onRestore()` methods of your backup agent, if only to make sure all file operations are synchronized using your lock for

thread-safe access. Here we have the full implementation of a backup agent that backs up one set of shared preferences called `AppPrefs` and a file named `appfile.txt`:

```
public class SimpleBackupAgent extends BackupAgentHelper {
    private static final String PREFERENCE_FILENAME = "AppPrefs";
    private static final String APP_FILE_NAME = "appfile.txt";
    static final String BACKUP_PREFERENCE_KEY = "BackupAppPrefs";
    static final String BACKUP_FILE_KEY = "BackupFile";

    @Override
    public void onCreate() {
        SharedPreferencesBackupHelper prefshelper = new
            SharedPreferencesBackupHelper(this,
            PREFERENCE_FILENAME);
        addHelper(BACKUP_PREFERENCE_KEY, prefshelper);
        FileBackupHelper filehelper =
            new FileBackupHelper(this, APP_FILE_NAME);
        addHelper(BACKUP_FILE_KEY, filehelper);
    }
    @Override
    public void onBackup(ParcelFileDescriptor oldState,
        BackupDataOutput data, ParcelFileDescriptor newState)
        throws IOException {
            synchronized (SimpleBackupActivity.fileLock) {
                super.onBackup(oldState, data, newState);
            }
    }
    @Override
    public void onRestore(BackupDataInput data, int appVersionCode,
        ParcelFileDescriptor newState) throws IOException {
            synchronized (SimpleBackupActivity.fileLock) {
                super.onRestore(data, appVersionCode, newState);
            }
    }
}
```

To make the `doBackup()` and `doRestore()` methods thread-safe, we simply wrapped the super class call with a `synchronized` block using your file lock.

Registering the Backup Agent in the Application Manifest File

Finally, you need to register your backup agent class in your application's manifest file using the `android:backupAgent` attribute of the `<application>` tab. For example, if your backup agent class is called `SimpleBackupAgent`, you would register it using its fully-qualified path name as follows:

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:backupAgent="com.androidbook.simplebackup.SimpleBackupAgent">
```

Backing Up and Restoring Application Data

The `BackupManager` system service manages backup and restore requests. This service works in the background, on its own schedule. Applications that implement a backup agent can request a backup or restore, but the operations might not happen immediately. To get an instance of the `BackupManager`, simply create one within your `Activity` class, as follows:

```
BackupManager mBackupManager = new BackupManager(this);
```

Requesting a Backup

An application can request a backup using the `dataChanged()` method. Generally, this method should be called any time application data that is to be archived changes. It can be called any number of times, but when it's time to back up, the backup takes place only one time, regardless of how many times `dataChanged()` was called before the backup.

```
mBackupManager.dataChanged();
```

Normally, the user does not initiate a backup. Instead, whenever important application data changes, the `dataChanged()` method should be called as part of the data saving process. At some point in the future, a backup is performed "behind the scenes" by the backup manager.

Warning

Avoid backing up sensitive data to remote servers. Ultimately, you, the developer, are responsible for securing user data, not the backup service you employ.

Requesting a Restore

Restore operations occur automatically when a user resets his device or upgrades after "accidentally" dropping his old one in a hot tub or runs it through the washing machine (happens more often than you'd think). When a restore occurs, the user's data is fetched from the remote backup service and the application's backup agent refreshes the data used by the application, overwriting any data that was there.

An application can directly request a restore using the `requestRestore()` method as well. The `requestRestore()` method takes one parameter: a `RestoreObserver` object. The following code illustrates how to request a restore:

```
RestoreObserver obs = new RestoreObserver(){
    @Override
    public void onUpdate(int nowBeingRestored, String currentPackage) {
        Log.i(DEBUG_TAG, "RESTORING: " + currentPackage);
    }
```

```
    @Override
    public void restoreFinished(int error) {
        Log.i(DEBUG_TAG, "RESTORE FINISHED! ("+error+")");
    }

    @Override
    public void restoreStarting(int numPackages) {
        Log.i(DEBUG_TAG, "RESTORE STARTING...");
    }
};

try {
    mBackupManager.requestRestore(obs);
} catch (Exception e) {
    Log.i(DEBUG_TAG,
        "Failed to request restore. Try adb bmgr restore...");
}
```

Warning

Backup services are a fairly new feature within the Android SDK. There are currently some issues (exceptions thrown, services missing) with running backup services on the emulator. Testing of backup services is best done on a device running Android 2.2 or later, in conjunction with the `adb bmgr` command, which can force an immediate backup or restore to occur.

Summary

Android applications do not exist in a vacuum. Users demand that their data be accessible (securely, of course) across any and all technologies they use regularly. Phones fall into hot tubs (more often than you'd think) and users upgrade to newer devices. The Android platform provides services for keeping local application data synchronized with remote cloud services, as well as protecting application data using remote backup and restore services.

References and More Information

Wikipedia on Cloud Computing:
http://en.wikipedia.org/wiki/Cloud_computing
Android Reference: The AccountManager Class:
http://developer.android.com/reference/android/accounts/AccountManager.html
Android Sample App: Sample Sync Adapter:
http://developer.android.com/resources/samples/SampleSyncAdapter/
Android Dev Guide: Data Backup:
http://developer.android.com/guide/topics/data/backup.html
Google's Android Backup Service:
http://code.google.com/android/backup/index.html

This page intentionally left blank