Development of a Privacy Preserving Liferay Portal document synchronizer for Android

by

Max Perry Perinato - 816507

Submitted to the Department of Environmental Sciences, Informatics and Statistics
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

CA' FOSCARI UNIVERSITY OF VENICE

A.Y. 2011/2012

Author ....
Department of Environmental Sciences, Informatics and Statistics
January 10, 2013

Certified by..
Michele Bugliesi
Full Professor & Head of Department
Thesis Supervisor

Accepted by ....
Augusto Celentano
Full Professor
Thesis Examiner

Development of a Privacy Preserving Liferay Portal document synchronizer for Android

by

Max Perry Perinato - 816507

Submitted to the Department of Environmental Sciences, Informatics and Statistics
on January 10, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In recent years we have seen an overwhelming spread of mobile devices, both for private and corporate use. To push this diffusion were the operating systems iOS by Apple and Android by Google, which currently dominate the market. This year in particular we saw Android reach a share of about 60% of all smartphones sold and about 50% of all tablets. Being these devices well suited for accessing all kinds of information and contents while on-the-go, many enterprises have started developing their own applications to give their employees the ability to access corporate data from anywhere. This however raises the problem of security of sensitive and confidential data, which is considered of great importance in the corporate domain. In this dissertation the problem of protecting data is addressed treating a real world corporate case study: the development of a native Android application (the client) for the synchronization of documents with Liferay Portal (the server), a leading open source web platform for enterprises. This application raises a number of security related issues, including, transferring data from server to client and vice versa, caching of information and off-line access, copying and sharing of data, managing and controlling user permission over each individual information (information rights management), revoking data access to untrusted users, and protecting information from theft or tampering (i.e. rooting) of the mobile device. Some of these problems can be eliminated, whereas many of them represent risks that can only be mitigated. The purpose of this work is to implement a synchronizer able to provide security of confidential data with the highest possible criterion, by harnessing software technologies compatible with the Android platform.

Thesis Supervisor: Michele Bugliesi
Title: Full Professor & Head of Department

Contents

List of Figures

Listings

Chapter 1

Introduction

In recent years we assisted to a rapid growth of mobile platforms. Smartphones and tablets are to the early 21st century what the PC was to the late 20th century - a multi-functional tool valued for its productivity and entertainment factor. As a consequence to their rise in popularity and capabilities, mobile devices are becoming a desirable work asset for enterprises who foresee in their adoption an added value to foster mobility and collaboration. However, when used in the enterprise, these devices are likely to process sensitive information. As an example, a common use case for an enterprise is to allow their employees to access and share documents anytime and anywhere, from their own mobile devices. This therefore raises the question of whether this data is secure with current mobile platforms and what actions an enterprise should take to secure sensitive information on mobile devices - whether to protect the enterprise's intellectual property or the privacy of employees and clients.

This thesis describes the development of Lifery Safe, a native application for Android that implements a client for document synchronization with servers running Liferay Portal, the leading open source portal for the enterprise. The purpose of this work is to design an acceptable solution for providing preservation of private documents and caching of data for offline use in a mobile environment.

Chapter two gives some background information about the Android platform, with particular interest in the security model and component framework.

Chapter three provides a brief overview of the Liferay open source enterprise portal, as well as introducing the document management capabilities integrated in the *"Document and Media Library"* portlet.

Chapter four describes the client-server architecture of the application, with particular attention to the security of the communication channel, including the details about user authentication and secure interaction with protected data.

Chapter five explains the design and implementation of the file synchronizer, which is based on Android's SyncAdapter pattern - an efficient synchronization mechanism perfectly integrated in the Android system.

Chapter six focuses on the dual problem of preserving private documents and providing offline access to these data. A proposed approach to this problem is described, which consists in caching private files in a virtual encrypted disk and entrusting the management of the encryption key to a secure service component. The design of this approach gave rise to several security risks, many of which are related to the Android platform. The chapter discusses how some of these risks can be circumvented, and what assumptions and constraints are necessary for those that can only be mitigated.

1.1 Motivations

The rapid adoption of high end smartphones and tablets, along with the diffusion of mobile applications and cloud based services, is driving a growing trend in the Bring Your Own Device (BYOD) concept among enterprises; as employees become more accustomed to using their own mobile computing devices in their jobs, and accessing corporate private data.

A recent report from Juniper Research [1] has found that the number of employee owned smartphones and tablets used in the enterprise will more than double by 2014, reaching 350 million, compared to almost 150 million this year (about 23% of the total installed base). The report also found that the majority of employees smart devices did not have any form of security software loaded nor were company materials protected. While BYOD is becoming an inevitable trend for the enterprise, it also poses potential security risks. Enterprises need to define their own end-user policies and address the key security issues emerging.

Employees demand access to documents and different types of content anytime, anywhere, from their mobile devices. For companies, however, this introduces significant compliance risks if the proper controls to protect this information are not in place. In particular, confidentiality and liability issues may arise when documents contain private information concerning the company, or personal information about its employees or clients. Organizations of all sizes need a simple, scalable way to distribute, manage and secure documents on smartphones and tablets.

The mobile device security model is erroneously based on the security model of their technological predecessor: the laptop computer [2]. Unlike the laptop, mobile devices are rarely shut down or hibernated. They are always turned on and are almost always connected, making the laptop security model insufficient. This connectivity also raises a new set of security risks with new threats and attack vectors. As illustrated by viaForensics security company (Figure 1-1) a mobile attack can involve the device layer, the network layer, the data center, or a combination of these [3]. In the hands of an attacker, a powered-on device is susceptible to information disclosure via its flash memory (internal/external) and on some devices the RAM. Additionally, privilege escalation bugs and public exploits for rooting a device are commonplace.

The primary security issue with BYOD centers around corporate data leakage and misuse. Mobile Device Management (MDM) solutions (e.g. Samsung SAFE) - often referred as *"Secure Containers"* - have been developed to facilitate the adoption of mobile devices into the enterprise and try to mitigate the risks inherent to the platform. However, recent security audits have discovered that such security solutions might still be vulnerable to a number of attacks that plague most third-party applications [2, 4], including, privilege esca-

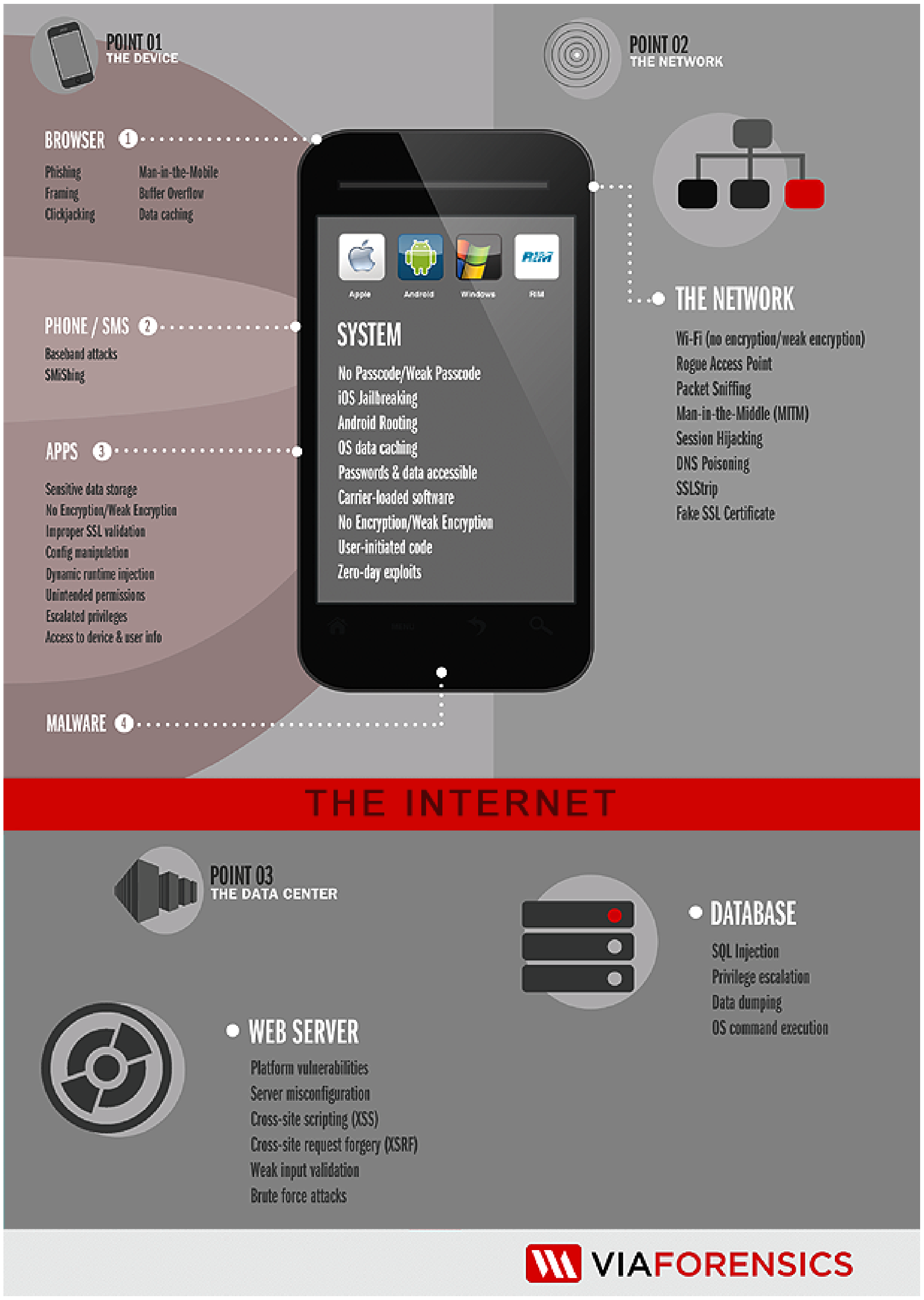


Figure 1-1: Anatomy of a Mobile Attack

lation, bad implementation of SSL, usage of vulnerable OS libraries and leakage of data into unprotected parts of the system. Above all, the main weakness of these secure containers stands in the lack of proper implementation of data encryption, which unfortunately results to be hindered by the same limitations of the platform's security model. However, in most cases security risks arise from bad design and insecure coding practices. While developing a secure mobile application is not absolutely straightforward, fortunately there are groups of security experts such as the Open Web Application Security Project (OWASP) who are actively collaborating to *"classify mobile security risks and provide developmental controls to reduce their impact or likelihood of exploitation"* [5]. OWASP's annual list of top ten mobile risks (Figure 1-2) can be a starting point for many enterprises that need to assess and mitigate potential risks posed by mobile devices to sensitive information.

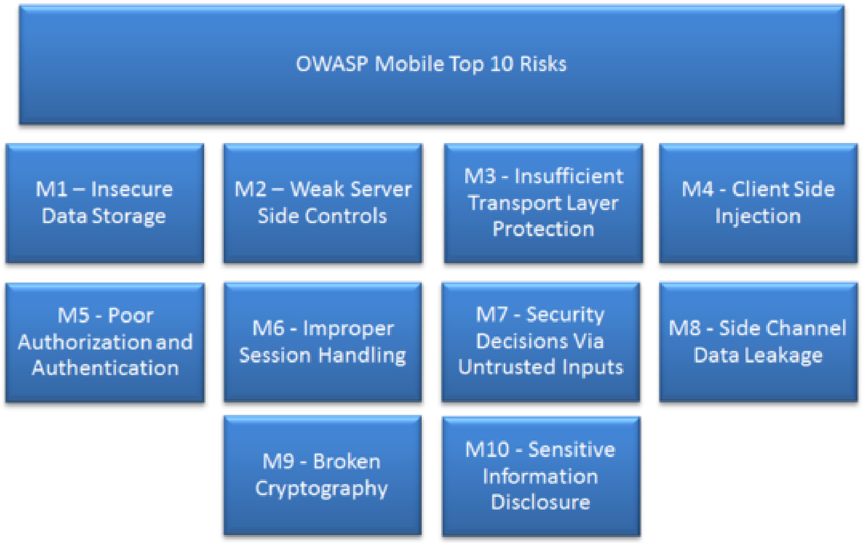

OWASP Mobile Top 10 Risks
M1 – Insecure Data Storage
M2 – Weak Server Side Controls
M3 - Insufficient Transport Layer Protection
M4 - Client Side Injection
M5 - Poor Authorization and Authentication
M6 - Improper Session Handling
M7 - Security Decisions Via Untrusted Inputs
M8 - Side Channel Data Leakage
M9 - Broken Cryptography
M10 - Sensitive Information Disclosure

Figure 1-2: OWASP Top Ten Mobile Attacks

1.1.1 The Android platform

Android is the open mobile platform developed by Open Handset Alliance (OHA). The most innovative feature of Android is its openness. This allows Android to be embraced by a large number of smartphone vendors, mobile operators, and third-party developers, driving shipment volumes to an impressive 75% share in the worldwide smartphone market, with over 135 million units shipped in Q3 2012 [6]. However, this popularity comes at a price. In one of its latest reports on mobile security, TrendMicro stated that:

> *"Malware targeting Googles Android platform increased nearly sixfold in the third quarter of 2012. What had been around 30,000 malicious and potentially dangerous or high-risk Android apps in June increased to almost 175,000 between July and September."* [7]

While the openness provides various benefits to developers and users, apparently it also increases security concerns. Due to the lack of a control in the application development and distribution processes, it is quite possible for a user to download and install malicious applications. As an attempt to reduce this risk, Google has recently introduced the Bouncer service, which provides automated scanning of the Google Play store for potentially malicious software. However, security experts have promptly demonstrated that the Bouncer can be easily bypassed [8]. Although it's not an easy problem to solve, clearly the consequences to the lack of effective moderation in the application store can include exposure of private information.

The security model of the Android system is based on application-oriented mandatory access control and sandboxing. By default, components within an application are sandboxed by Android, and other applications may access such components only if they have the required permissions to do so. This allows developers and users to restrict the execution of an application to the privileges it has (statically) assigned at installation time. However, enforcing permissions is not sufficient to prevent security violations, since permissions may be misused, intentionally or unintentionally, to introduce insecure data flows [9]. In fact, Davi et Al. [10], have shown that Android's sandbox model is conceptually flawed and actually allows privilege escalation attacks. In confirmation of these findings, a later survey provided a taxonomy of attacks to the platform demonstrated by real attacks that guarantee privileged access to the device [11].

The Android operating system also provides a rich inter-application message passing system. This encourages inter-application collaboration and reduces developer burden by facilitating component reuse. Unfortunately, message passing is also an application attack surface. The content of messages can be sniffed, modified, stolen, or replaced, which can lead to breaches of user data and violation of application security policies [12].

A number of security frameworks have been developed with the purpose to enhance Android's security architecture. Saint [13] addresses the current limitations of Android security through install-time permission granting policies and run- time inter-application communication policies. Kirin [14] extracts an application's security policy from its manifest file to detect data flows allowing privilege escalation attacks. SCanDroid [9] statically analyzes data flows through Android applications to help the developer make security-relevant decisions, based on such flows. ComDroid [12] detects application communication vulnerabilities. TaintDroid [15] extends the Android platform to track the flow of privacy sensitive data through third-party applications.

These tools can be useful for developers to certify the security of their Android applications and detect vulnerabilities due to bad design and insecure coding practices. Still, companies supporting BYOD must understand that mobile devices have a very large threat surface which makes full-protection of sensitive information stored in uncontrolled (and possibly compromised) mobile devices an inherently impossible task, even when providing access to enterprise resources through certified secure containers. An ideal approach already pursued by security and military agencies [16] would be for enterprises to build their own security enhanced version of Android and give company-owned devices to their employees, yet giving up the productivity benefits of the trending BYOD model.

1.2 Problem statement

Considered the several areas of risk to which mobile platforms are exposed, and considered the issues inherent to Android's security model, the problem to be asked is the following:

Can documents organized in an enterprise portal be synchronized with an Android device and still have their privacy preserved? Is it also possible to cache these private data for offline access without loosing their control? Finally, can users be effectively revoked and data synchronized on their devices be consistently removed?

This thesis will discuss the development of Liferay Safe, a software for the synchronization on Android devices of documents stored in a Liferay portal. The purpose of the thesis is to provide and explain the implementation of a concrete solution to the problems stated above.

Chapter 2

File synchronization

One of the capabilities that most prominently characterizes the paradigm of mobile computing is the possibility of working with a variety of content (such as documents and media) while being "on the go"; that is, accessing shared and private files anytime and everywhere directly from a mobile device. Indeed, this is a very common scenario in the ecosystem of mobile applications, since many of these applications wouldn't be much interesting without being connected in some way to content repositories or data services on the cloud.

In the design and implementation of the Liferay Safe client, we face directly the problem of file synchronization; more specifically, files are added to a Liferay Portal server application, through the *"Documents and Media Library"* portlet, and synchronized with client applications installed on a diverse set of mobile devices running Android. The synchronization is *two-way*, since updated files are copied in both directions - and across many clients, as the same user can sync multiple devices of his own - with the purpose of keeping the two locations identical to each other (i.e. synced).

However, dealing with any kind of data synchronization in a mobile context is way different than dealing with it in a traditional (desktop) computing environment. In fact, keeping local and remote files synced requires to pose attention to many aspects that are exclusive to the world of mobile computing. The most relevant of these derive from considering some characteristics about the networks, the hardware, and the operating system of a mobile device, including, but not limited to, the following:

- Prolonged or intermittent, network connection unavailability;
- Cost of cellular data connection;
- Limited hardware resources - processor, memory and storage - and network bandwidth;
- Multi-tasking: resources and data connection are shared among multiple concurrently running applications;
- Limited battery power time;
- Implications on software design imposed by the underlying software platform (i.e. Android);

- Lack of a desktop environment: the user interface for file operations must be built from scratch as part of the application's user interface;

- Increased data security risks (wrt. traditional desktop environments).

These aspects, all together, imply that a mobile client for file synchronization should track and persist all file modifications even when there is no connectivity, and automatically sync in the background these changes, with the server counterpart on the cloud, as soon as the connectivity returns - while consistently detecting and managing possible conflicts. Actually, due to the limited storage space, not all files can be automatically downloaded and synchronized, instead only the one's demanded by the user should be. Therefore, the client has to manage a size-adjustable cache for user's favorite and files uploaded from local - to be kept in sync. Additionally, the user should still be able to view a list of all files in the synced repository and browse any folders. Thus, the client has to keep in sync the logical description of every file (i.e. UUID, title, extension, UUID of parent folder, etc.) and locally persist this entry along with a value indicating the synchronization state of the corresponding file's content. When all file entries are retrieved, the full repository structure can be represented and automatically be kept in sync with its evolution over time - without needing to download the content of all files. Finally, the client sync process should be optimized to have the least resource footprint possible, in order to reduce battery consumption and efficiently coexist alongside other applications, with which has to coordinate to responsibly use the available connection bandwidth and perform the needed sync operations.

In short, a mobile client for file synchronization should include at least the following features:

- Automatic synchronization of file entries and cached file contents;

- Detection and tracking of file operations (edit, rename, move, delete);

- Conflict detection;

- Selective download of remote files;

- Import and upload of local files;

- New folder creation;

- Encryption for security of private files, at both transport and storage levels.

In this chapter, we will explore a particular synchronization pattern leveraged by Android, called the SyncAdapter, while also illustrating the criterium of synchronization and its algorithmic implementation. Besides, we will define how the entry of a file is represented across various states and locally persisted, and finally, discuss how downloads and uploads are performed. Later in the next chapter, we will unravel the cache architecture and propose an approach for security of private files.

2.1 SyncAdapter pattern

The Android platform provides, through its component framework, several patterns for implementing a client to synchronize data with HTTP Web Services. In this section, we focus on the most interesting of these patterns, the one using both the ContentProvider API and a SyncAdapter, simply called the SyncAdapter pattern. This pattern was first introduced by Google engineer Virgil Dobjanschi in a session during Google IO 2010 [17]. Of the patterns presented by him, the SyncAdapter was the most interesting due to its level of integration with the Android system.

Through the SyncAdapter, an application can register with the system to perform any synchronization work with a remote server, delegating to the system itself all the burden of managing eventual sync errors and consequent reattempts. In fact, the difference from a pattern not using the SyncAdapter is that with the latter, the available resources and connection bandwidth are efficiently shared among all applications registered with the system through this mechanism. That is, the system will automatically listen for sync requests, enqueue them, and subsequently perform them as soon as connectivity is available. This ensures that any synchronization would not interfere with other running syncs or applications using data connection, while minimizing the network usage. If a synchronization could not be started or successfully completed due to an error, the system will automatically arrange a new attempt - an exponential backoff algorithm will set a priority in the work queue for the reattempt, based on the gathered error information. Moreover, with the ContentProvider API, synchronized data can be neatly persisted and conveniently updated to the User Interface by means of the ContentResolver with small effort. Therefore, the developer can relief from all this dull work, yet keep the code cleaner and be assured that her application is complying with the limitations of the underlying mobile platforms which we discussed before.

Despite Engr. Dobjanschi during his talk hasn't displayed any demo application nor has provided any example code - actually a sample SyncAdapter for the Contacts Provider was later released along the SDK - he illustrated the pattern as clearly as needed to be correctly implemented and customized by any experienced developer. The pattern is schematized in Figure 2-1 below.

In this pattern, the SyncAdapter is the central component responsible for performing the synchronization operations. It relies on the ContentProvider to retrieve all the items that need to be synced (1.), this is the preliminary step of every sync task necessary to send to the server any local pending updates and make the synchronization two-way. After all items are retrieved, the sync algorithm can be started from the SyncAdapter's onPerformSync() method (2.), which was originally called by the system or from another component by means of the ContentResolver. During this phase all communication with the server's Web Services API is performed by the HTTP client via HTTP request methods (3.). Local updates are pushed to the server (POST/PUT/DELETE) and remote modifications are retrieved (GET). All retrieved data must be unmarshaled, compared with existing data to detect eventual conflicts, and finally persisted (4. and 5.). A Processor component should take these responsibilities and interact with the ContentProvider's API (insert, update,

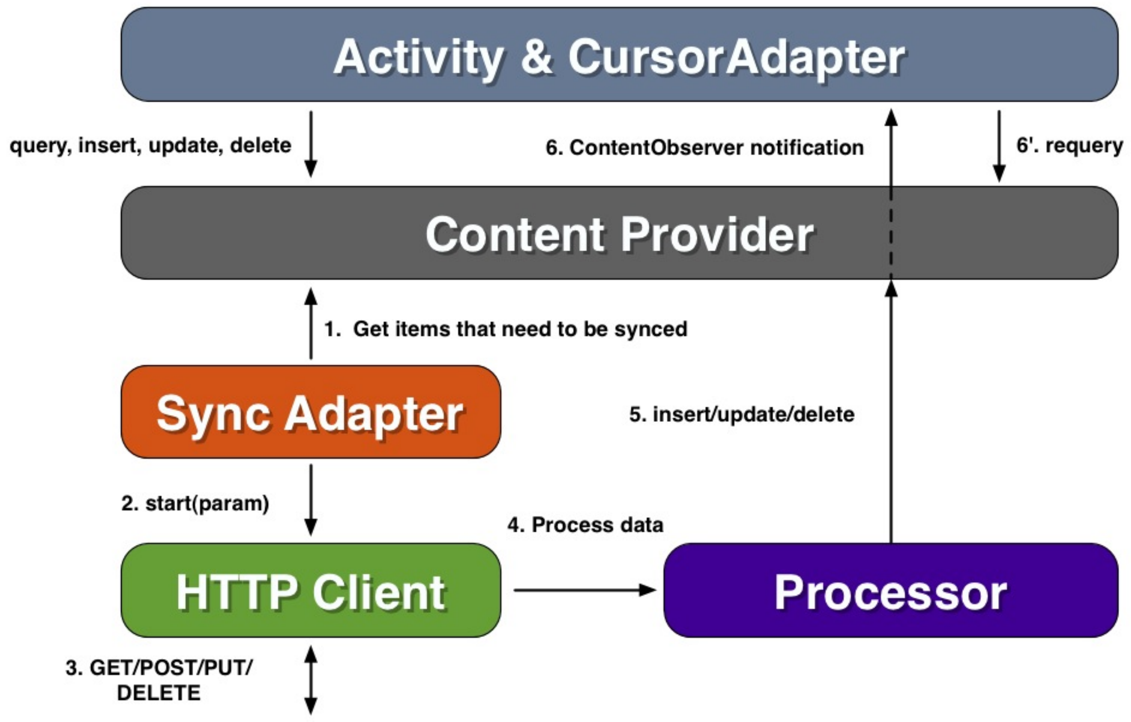


Figure 2-1: Android SyncAdapter pattern

delete). Any Activity can rely on a CursorAdapter to update its views with new synced data. A ContentObserver object will receive callbacks from the ContentProvider whenever its content is changed, and notify the CursorAdapter (6.) which can then requery the dataset (6'.).

In pills, the SyncAdapter pattern allows to decouple the synchronization logic from activities and execute background operations which are not time critical, while minimizing the network usage. Additionally, by relying on the ContentProvider, data can be persisted early and often in a convenient way, thus keeping the application functionality sound wrt. its lifecycle - an application (or one of its components) can be terminated by the user or the system anytime.

2.2 Implementation

Implementing the SyncAdapter pattern might still get a little bit challenging despite its simplicity, as there are not many examples and documentation explaining how the concept works. Fortunately, there are a few articles and the SampleSyncAdapter included in the SDK providing some examples useful to set off [18, 19, 20, 21]. So let's start illustrating the implementation steps and details of the SyncAdapter for Liferay Safe, which is schematized in Figure 2-2 below.

Simply put, the SyncAdapter is just a service that is started by the Sync Manager, which

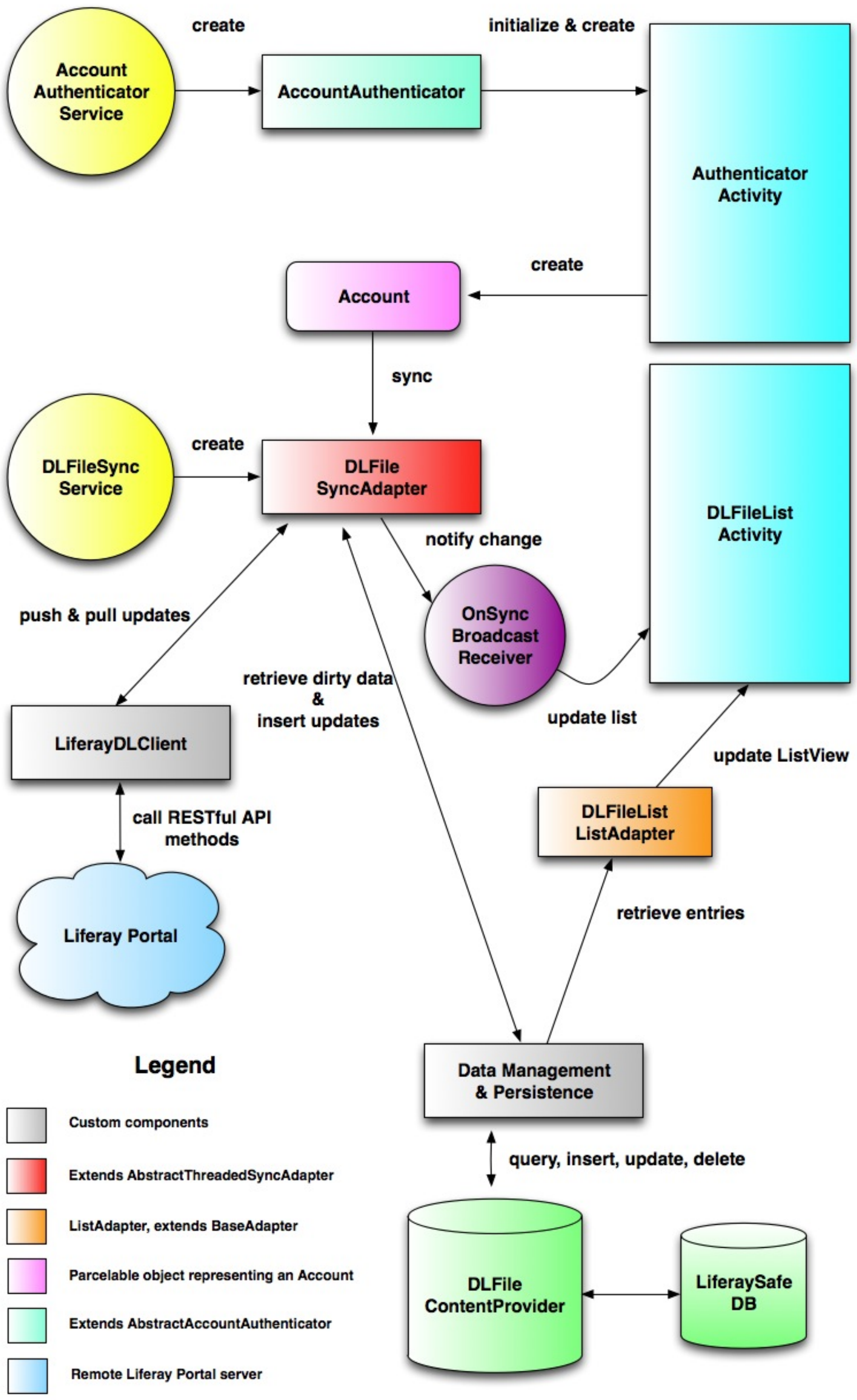


Figure 2-2: Implementation of the SyncAdapter pattern

in turn maintains a queue of SyncAdapters and is responsible for choosing when to perform the sync, and for scheduling resyncs according to the errors reported by the SyncAdapter.

The basic idea is that remote data is mirrored in a local database. Activities can access this data through the ContentProvider and display it to the user - for example, with a ListView and a ListAdapter - allowing him to perform various operations. The SyncAdapter will be in charge of making the remote and local data match; it will push the local changes to the server and fetch the new data.

There are several components involved in the implementation of this pattern. In general, one first needs to write its own implementation of the components, then has to glue all the components together using the AndroidManifest file, and finally make the SyncAdapter interact with the application. Hereafter are described all the steps involved in the implementation, including a description of all the necessary components.

2.2.1 Account creation

In order to use a SyncAdapter an Account is needed. On Android an Account can be exactly any kind of account a user owns to access some online service. It is essentially a pair of strings; one is the username identifying the user on the remote server, and the other is the type distinguishing it from the others available on the system. Different types of online user accounts are organized and stored in a centralized system registry, the component responsible for providing access to this registry is the AccountManager.

Applications interact with the AccountManager to add or retrieve particular types of accounts they are interested in. Since different online services have different ways of handling accounts and authentication, the AccountManager uses pluggable authenticator modules - extending AbstractAccountAuthenticator - for different account types. Authenticators handle the actual details of validating account credentials and storing account information.

To interact with the user to prompt for credentials, present options, or ask the user to add an account, an activity extending AccountAuthenticatorActivity is needed. This class provides the basic implementation necessary to handle requests for and pass responses to an authenticator. A user can also add, modify or remove her accounts from the "Accounts & sync" section of the "Settings" application.

When an account has to be created, an Intent is sent to the system specifying the type of account requested. The system will look up for the authenticator associated exactly with that type (exposed through the AndroidManifest file) and start a service from which the authenticator itself can be instantiated and binded. Next the authenticator's addAccount() method can be called and subsequently the AccountAuthenticatorActivity will be started with an Intent, passing a response and other account related values. The activity will be responsible to collect the user's credentials and perform authentication with server. If the credentials are correct, the last step is to call the AccountManager's addAccountExplicitly() method and effectively add the account by passing in the new Account object and the password. Other user data can be associated to account and stored in the AccountManager with a key by calling its setUserData() method. This is useful for example to store the server's host URL. Finally the activity has to call setAccountAuthenticatorResult() passing the response back to the authenticator with the name and type of the account that was added, or an error code and message if an error occurred.

Security Remark - Account data protection is based on the Linux user id (UID) of the process making the request. Each account is associated with an authenticator (that has a UID), and the process calling getPassword() (or several other methods) must have the same UID as the authenticator. This prevents extraneous applications to steal the user's credentials from the AccountManager.
Still, it is NOT safe to store in the AccountManager user's passwords and other sensitive data. In fact, all data is stored "as is" (and so eventually in plain text) in a SQLite database located in /data/system. Thus, after gaining root access on the device (which is often extremely simple), a malicious user can easily access these system files via Android's adb and retrieve all stored account info [22].
For the record, in August 2010 was reported an issue on AOSP stating: "The password for email accounts is stored into the SQLite DB which in turn stores it on the phone's file system in plain text." [23]. The issue was closed one year later proposing full-disk encryption as the "most appropriate solution", starting with Honeycomb. More recent comments in fact still confirm the database is stored in plain text.
An alternative and safer approach would be to store user's sensitive information through the Android KeyStore API. This provides access to a credential storage encrypted using an AES 128 bit master key, which in turn is derived from the device unlock password or PIN. This means that secrets cannot be extracted even after gaining root access, unless the password is known. However, the master encryption key is not tied to the device, so it's possible to copy the encrypted key files and perform a brute force attack on more powerful machine(s) [24].

Many servers support some notion of an authentication token, which can be used to authenticate a request to the server without sending the user's actual password. AccountManager can generate auth tokens for applications, so the application doesn't need to handle passwords directly. In Liferay Safe this functionality is not used. As discussed earlier authentication tokens are handled independently by the application and in a different way wrt the AccountManager, in order to mitigate related security risks. For the same security reasons, the user's password is never stored in the device (a dummy string will be passed when calling addAccountExplicitly() on the AccountManager); therefore, the user will be prompted for credentials whenever the authentication token has expired.

The Account is an important piece of the SyncAdapter pattern, because it ties the SyncAdapter and the ContentProvider together. In fact, *"AccountManager, SyncAdapter and ContentProvider go together. You cannot use an AccountManager without a SyncAdapter. You cannot use a SyncAdapter without an AccountManager. You cannot have a SyncAdapter without a ContentProvider"*[25].

2.2.2 SyncAdapter implementation

In order to write a SyncAdapter one must extend the AbstractThreadedSyncAdapter class, provide implementations for the abstract onPerformSync() method and write a service that returns the result of getSyncAdapterBinder() in the service's onBind() when invoked with an intent with action android.content.SyncAdapter.

When a requestSync() is received (from the SyncManager), a thread will be started to run the operation and onPerformSync() will be invoked on that thread.

- If a sync operation is already in progress, then an error will be returned to the new request and the existing request will be allowed to continue.
- If a cancelSync() is received that matches an existing sync operation, then the thread that is running that sync operation will be interrupted.

All the synchronization logic resides in the body of the onPerformSync() method, including communication with the server and interaction with the ContentProvider for data persistence. There is no one standard way for sync logic, as it depends strongly on the server's API.

Instead, setting up a service that filters SyncAdapter intents is relatively straightforward.

Listing 2.1: DLSyncService

```
public class DLFileSyncService extends Service {
        private static final Object sSyncAdapterLock = new Object();
        private static SyncAdapter sSyncAdapter = null;
        @Override
        public void onCreate() {
        synchronized (sSyncAdapterLock) {
                if (sSyncAdapter == null) {
                sSyncAdapter = new DLFileSyncAdapter(getApplicationContext(), true);
                }
        }
        }
        @Override
        public IBinder onBind(Intent intent) {
                return sSyncAdapter.getSyncAdapterBinder();
        }
}
```

The service definition given above must be registered in the AndroidManifest file and specify the following intent filter and metadata tags:

Listing 2.2: Manifest service tag

```
$\langle$ service
        android:name=".syncadapter.DLFileSyncService"
        android:exported="true" $\rangle$
$\langle$ intent−filter$\rangle$
             $\langle$ action android:name="android.content.SyncAdapter" /$\rangle$
$\langle$ /intent−filter$\rangle$

$\langle$ meta−data
             android:name="android.content.SyncAdapter"
             android:resource="@xml/syncadapter" /$\rangle$
$\langle$ /service$\rangle$
```

Where the android:resource attribute points to the following resource in the res/xml folder

Listing 2.3: Manifest sync-adapter tag

```
$\langle$ sync−adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.liferay.safe"
    android:accountType="liferaysafe"
    android:userVisible="false"
    android:supportsUploading="true"
    android:allowParallelSyncs="false"
 /$\rangle$
```

The android:contentAuthority and the android:accountType property are particularly important because they tie the SyncAdapter respectively with the ContentProvider and the account type. In fact, the value of the first property should match the android:authorities property value of the ⟨ provider /⟩ resource declared in the AndroidManifest file, while the value of the second should be exactly the one declared in the ⟨ account-authenticator /⟩ resource located in res/xml. As a matter of fact, the instances of the Account and the ContentProvider are passed as parameters to the onPerfomSync().

As for the remaining properties of the ⟨ sync-adapter /⟩ resource,

- android:userVisible="false" specifies that the SyncAdapter should not show up in the "Accounts & sync settings" screen;

- android:supportsUploading="true" enables an upload-only sync to be requested whenever the associated authority's ContentProvider does a notifyChange() with syncToNetwork set to true;

- android:allowParallelSyncs="false" indicates that the SyncAdapter cannot handle syncs for multiple accounts at the same time.

Also note that the service's android:exported is set to true, this means that other applications can invoke the service and interact with it. Although it might be interpreted as a security risk, this is necessary in order to receive sync requests with the intent filter.

An ulterior parameter of onPerformSync() which is of particular interest, is the SyncResult. This object is used to communicate the results of a sync operation to the SyncManager. Based on the values in the SyncResult, the SyncManager can determine - by means of an exponential backoff algorithm - whether to reschedule the sync in the future or not. More specifically, via the SyncResult, the SyncManager evaluates also the SyncStats object, which records various statistics about the result of a sync operation, including, and not limited to:

- authentication exceptions occurring when authenticating the Account specified in the sync request (numAuthExceptions);
- IO exceptions related to network connectivity or timeout while waiting for a network response (numIoExceptions);
- exceptions while processing the data received from the server, usually due to malformed or corrupted data (numParseExceptions).

Using the SyncResult object in a clever manner will allow the SyncManager to be more effective.

A SyncAdapter can be started manually:

Listing 2.4: SyncAdapter manual start

```
Bundle bundle = new Bundle();
bundle.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true);

ContentResolver.requestSync(account, "com.liferay.safe", bundle);
```

Or alternatively, it can be scheduled to start periodically, for example every hour:

Listing 2.5: SyncAdapter periodic start

```
Bundle params = new Bundle();
params.putBoolean(ContentResolver.SYNC_EXTRAS_EXPEDITED, false);
params.putBoolean(ContentResolver.SYNC_EXTRAS_DO_NOT_RETRY, false);
params.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, false);

```

```
ContentResolver.addPeriodicSync(account, "com.liferay.safe", params, 3600);
ContentResolver.setSyncAutomatically(account, "com.liferay.safe", true);
```

Finally, any active or pending syncs that match an account and an authority (or any of them) can be canceled:

Listing 2.6: SyncAdapter cancel sync

```
ContentResolver.cancelSync(account, "com.liferay.safe");
```

2.2.3 UI notification

If a synchronization is performed when the related application is running, the user should be notified of the affected changes and the UI should be updated. There are several ways to carry out this task, the most convenient of them involves using a CursorAdapter and ContentObserver notifications.

Whenever data is inserted, updated or deleted, clients should be notified about these changes in the underlying datastore of the ContentProvider. This is accomplished by calling notifyChange() on the ContentResolver, specifying the URI of the content that was changed.

Listing 2.7: ContentResolver notifyChange()

```
getContext().getContentResolver().notifyChange(uri, null);
```

To receive these notifications, Android provides the ContentObserver class following the object-observer pattern, which can be implemented as a subclass, and registered to the ContentResolver to listen for changes [26].

Whenever a change occurs, the onChange() method is called, therefore the implementation has to override this method and perform there all the logic necessary to update the UI. To avoid executing this code on the UI thread, it's recommended to create the handler for the ContentObserver on a separate thread, for example using an AsyncTask.

Then, registering a ContentObserver is as simple as calling the ContentResolver's registerContentObserver() method:

Listing 2.8: Registering the ContentObserver

```
getContentResolver().registerContentObserver(SOME_URI, true, yourObserver);
```

Where the second parameter specifies if changes to URIs beginning with SOME_URI (descendents) will also cause notifications to be sent. Content URIs can be directory-based or id-based. The first one is more appropriate when needing to update a list of data, while the second should be used for updating a detail screen. However, the choice of the URI depends on how the ContentProvider was implemented.

A ContentObserver can be used almost by any component, but it turns out very useful with a CursorAdapter. A CursorAdapter is an adapter that exposes data from a Cursor (returned from a ContentProvider) to a ListView. When using a CursorAdapter, an activity can register a ContentObserver to receive notifications and call the adapter's changeCursor() to update a ListView whenever the underlying data returned by the Cursor has changed.

Listing 2.9: Custom ContentObserver

```java
class MyObserver extends ContentObserver {
    public MyObserver(Handler handler) {
        super(handler);
    }
    @Override
    public void onChange(boolean selfChange, Uri uri) {
        Cursor myCursor = managedQuery(uri, projection, where, whereArgs, sortBy);
        myAdapter.changeCursor(myCursor);
    }
}
```

Note that the original pattern suggested calling requery() on the Cursor, and hence the ContentObserver could have been registered directly on the Cursor itself calling registerContentObserver(). However, requery() was deprecated suggesting infact to *"Just request a new cursor, so you can do this asynchronously and update your list view once the new cursor comes back."*

This mechanism also provides an effective model to show to the user the progress of a synchronization. In fact, by storing a column in each record in the ContentProvider indicating the sync state of that record, it's possible to have a per-row granularity for information, allowing, for example, to have an independent spinner for each item in the ListView, telling the user exactly what data is being synced.

Unfortunately there are two main downsides when using content observers, one is that for directory-based URIs it's not possible to get a list of id's that have changed, which can be a problem for big datasets if only a few records have changed, second, it's often difficult to reduce the number of notifications of bulk operations.

There is one more issue about UI notification which is more closely related to the SyncAdapter, namely that, at the state of the art, there is no sound way to get notified when the overall sync process has started or ended. This is a major problem especially when the SyncAdapter is started periodically.

Scrolling through the ContentResolver's API we find the addStatusChangeListener (int mask, SyncStatusObserver callback) method. By calling this method a component can request notifications when different aspects of the SyncManager change, such as when a sync is active, pending or the sync settings have been changed. The status change will cause the onStatusChanged() method of the SyncStatusObserver object to be invoked. An example can be found here [27] and here [28].

However, this utility is pretty useless for a SyncAdapter, since addStatusChangeListener() notifies the callback object when the sync status of ANY SyncAdapter changes, not a particular sync identified by an account + authority combination [29, 30]. This would cause too many refresh apart from the fact that it would not be possible know what is being synced nor distinguish the start of the sync event from the end.

An alternative, but dirty, approach would be to spawn a thread which continuously checks the result of isSyncActive() and isSyncPending() from the ContentResolver. These methods return true if there is currently a sync operation for the given account or authority actively being processed, or in the pending list.

In Liferay Safe we have adopted a different approach which consists in using a BroadcastReceiver, and a ListAdapter (instead of a CursorAdapter) to populate the ListView. In the activity managing the ListView, we register the broadcast receiver when onResume() is called:

Listing 2.10: Registering an OnSyncBroadcastReceiver

```
IntentFilter syncIntentFilter = new IntentFilter(DLFileSyncService.SYNC_MESSAGE);
mSyncBroadcastReceiver = new OnSyncBroadcastReceiver();
LocalBroadcastManager.getInstance(this).registerReceiver(mSyncBroadcastReceiver,
syncIntentFilter);
```

And unregister it in onPause() to avoid memory leaks:

Listing 2.11: Unregistering an OnSyncBroadcastReceiver

```
if (mSyncBroadcastReceiver != null) {
LocalBroadcastManager.getInstance(this).
unregisterReceiver(mSyncBroadcastReceiver);
        mSyncBroadcastReceiver = null;
}
```

The BroadcastReceiver is defined as a private inner class of the activity:

Listing 2.12: OnSyncBroadcastReceiver

```
private class OnSyncBroadcastReceiver extends BroadcastReceiver {

@Override public void onReceive(Context context, Intent intent) {
        boolean inProgress =
                        intent.getBooleanExtra( DLFileSyncService.IN_PROGRESS, false);

        String accountName = intent .getStringExtra(DLFileSyncService.ACCOUNT_NAME);

        if (accountName.equals(AccountUtils.getCurrentLiferayAccount(context).name)) {

                String synchFolderRemotePath =
                        intent.getStringExtra(DLFileSyncService.SYNC_FOLDER_REMOTE_PATH);

                boolean fillBlankRoot = false;
                if (mCurrentFolder == null) {
                        mCurrentFolder = getDLFileManager().getDLFile(PathHelper.PATH_ROOT);
                        fillBlankRoot = (mCurrentFolder != null);
                }
                if ((synchFolderRemotePath != null
                        && mCurrentFolder != null
                        && (PathHelper.fixPath(mCurrentFolder.getFilePath())
                                                .equals(synchFolderRemotePath)))
                    || fillBlankRoot ) {
                        if (!fillBlankRoot)
                                mCurrentFolder = getDLFileManager().getDLFile(synchFolderRemotePath);

                        DLFileListFragment fileListFragment =
                                        (DLFileListFragment) getSupportFragmentManager()
                                                .findFragmentById(R.id.fileList);
                        if (fileListFragment != null) {
                                fileListFragment.listFolder(mCurrentFolder);
                        }
                }
                setSupportProgressBarIndeterminateVisibility(inProgress);
        }
 }

}
```

When a broadcast intent is received, the receiver checks the progress status and the account name, sent from the SyncAdapter in the intent's extras, and sets a progress indicator in the activity. When the synchronization is notified as completed, the receiver will invoke the update of the ListView.

The SyncAdapter sends a broadcast intent both at the beginning and at the end of the sync operation. Additionally, the SyncAdapter can also notify the activity when a folder has been updated, by setting its path as an extra in the intent. This allows to refresh the view displaying the contents of that folder, when visible to the user, avoiding to wait the full sync event to complete.

Listing 2.13: SyncAdapter UI notification local broadcast

```
private void notifyUI(boolean inProgress, String dirRemotePath) {
        Intent i = new Intent(getContext(), DLFileListActivity.class);
        i.setAction(DLFileSyncService.SYNC_MESSAGE);
        i.putExtra(DLFileSyncService.IN_PROGRESS, inProgress);
        i.putExtra(DLFileSyncService.ACCOUNT_NAME, getAccount().name);
        if (dirRemotePath != null) {
                i.putExtra(DLFileSyncService.SYNC_FOLDER_REMOTE_PATH, dirRemotePath);
        }
        LocalBroadcastManager.getInstance(getContext()).sendBroadcast(i);
}
```

By using the LocalBroadcastManger to publish and subscribe for broadcasts, we avoid all the security issues related to global broadcasts, such as leaking private data or receiving the same broadcasts from other applications [31]. Besides, since local broadcast intents never go outside of the current process, the communication is also more efficient [32].

Finally, instead of using Sticky Broadcasts (which are insecure [33]), we can still detect a periodic sync that was started when the application was paused or closed, by calling the ContentResolver's isSyncActive() and eventually update the UI. The application will still register for sync broadcast and be notified by the SyncAdapter when the sync operation has finished.

2.2.4 Required permissions

The final step for implementing the SyncAdapter pattern is to declare in the AndroidManifest file a few permissions to allow several aspects of the application to function appropriately: ability to read/write accounts, ability to interact with the network, and ability to read/write sync settings.

Here is a snippet from Liferay Safe's AndroidManifest.xml:

Listing 2.14: SyncAdapter pattern permissions

```
$\langle$ uses-permission android:name="android.permission.INTERNET" /$\rangle$
$\langle$ uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" /$\rangle$

$\langle$ uses-permission android:name="android.permission.GET_ACCOUNTS" /$\rangle$
```

```
$\langle$ uses−permission android:name="android.permission.USE_CREDENTIALS" /$\rangle$
$\langle$ uses−permission android:name="android.permission.MANAGE_ACCOUNTS" /$\rangle$
$\langle$ uses−permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" /$\rangle$

$\langle$ uses−permission android:name="android.permission.READ_SYNC_STATS" /$\rangle$
$\langle$ uses−permission android:name="android.permission.READ_SYNC_SETTINGS" /$\rangle$
$\langle$ uses−permission android:name="android.permission.WRITE_SYNC_SETTINGS" /$\rangle$
```

In the next sections we will focus on the synchronization logic and we'll cover the remaining building blocks of the SyncAdapter pattern: the Data Management & Persistence layer providing access to the ContentProvider, and the HTTP client executing the methods of the Web Services API (LiferayDLClient).

2.3 Synchronization algorithm

A preliminary step necessary for designing the logic of a synchronization client is to analyze the sync policy enforced by the server. For sync policy we intend all the rules that define the behaviour of the client when performing two-way synchronization of operations carried on syncable items.

In Liferay Safe there are two main classes of syncable items, user sites (UserSite) and file entries (FileEntry). A user site represents a site available in Liferay Portal that is identified by a "groupId". File entries are records of the "Document and Media Library" (DML) that describe file contents stored in a user site's repository or directories defined in the DML. A file entry is identified by a "fileEntryId" and is tied exactly to a user site with a "repositoryId" that corresponds to the user site's "groupId", whereas it is tied to a file content by means of the file's "uuid". For file content we mean precisely a file stored in the server's Content Repository; we may call it simply a file, a document, a media or a content.

A change in a file entry can give rise to three types of synchronization events: "add", "update" or "delete". Synchronization events subsequent to a given timestamp are pulled together from the "Document and Media Library" as a collection of document library sync entries (DLSyncUpdate). A document library sync entry (DLSync) is tied to a user site by a "repositoryId" and to a file entry by a "fileId".

The responsibilities of a client when carrying out these synchronization events depend on the role of the server. Two main issues for a file synchronizer are conflict resolution and file consistency on one hand, and controlling user permissions on the other. Fortunately the responsibilities of Liferay Safe are simplified for the fact that Liferay Portal provides a centralized control of permissions on user operations and a centralized versioned repository.

This practically means that Liferay Safe in most cases can delegate to the server the burden of managing synchronization conflicts. In fact, the DML keeps track of every version of a file entry and its file content; therefore all clients, when synchronized, will always have the latest version of that entry. This means that if a client misses some updates and uploads a new modification before syncing, the server will accept the modification as the latest version, without creating any conflict. Other clients can still retrieve the previous versions and the user can take care by himself of merging the modifications. Similarly, if a new file is uploaded and has the same full path of an existing one, it will be treated as a new version of that existing file without raising a conflict.

Being Liferay Portal a collaborative platform where more than one person can be working on a file at the same time, "editing conflicts" may occur very often. File versioning allows to keep contents consistent across many clients and more importantly it ensures that newer modifications are never lost by overwrites - while eliminating the clutter of file duplications due to unresolvable conflicts.

In Liferay, contents of the DML can be accessed also via WebDAV, which can be used by applications to write directly on remote files as if they were local. Often, applications use file locking to protect files that are in use. But, having a document open in any application that lock its files can increase the likelihood of conflicting edits taking place. Any changes made in an application that has a file locked will create inevitably a conflict. In this case, the client uploading changes (not holding the lock) will have to abort the upload and create a local duplicate of the file to be sorted out by the user.

For security implications, instead, it's not possible to completely rely on a centralized user permission control. The reason is straightforward: the client needs to know if user's permissions have changed after a file content has already been downloaded. However, since Android devices have a very limited internal storage, most of the times application data has to be stored in the external SD storage, but there, all contents are world-readable because the file system lacks of user permissions. Thus, at this point it is impossible for the client to effectively enforce user's permissions defined on the server, not least it would be worthless.

We postpone to the next section the discussion of a trade-off approach to this problem, when we'll introduce the concept of private file content. For now, let's define a private file content as a file content whose file entry is marked as confidential.

To respond correctly to synchronization events we define the following rules relying on the server's policy, which apply to all file entries.

- Local updates and uploads must be pushed to the server immediately (if possible) or always before pulling remote updates;

- Contents should be downloaded ("cached") only upon user request; only the updates of contents whose entries are marked as "keep in sync" must be downloaded automatically;

- Local entries must track and persist the synchronization state of their associated cacheable content;

- Remote deletions must be applied immediately, whatever is the file entry's sync state;

- Local deletions are voluntarily not supported in Liferay Safe. The user is only allowed to remove a file content from the local cache storage.

Based on the synchronization policy defined above, we can now explain the sync mechanism in algorithmic form, which should be implemented in the SyncAdapter's onPerformSync() method.

Listing 2.15: Synchronization algorithm: main

```
1  Notify the UI that the sync event has started
2  Request "get-user-sites", update local user site entries and notify the UI
3  For each UserSite site
4          If site.lastAccessDate = 0
5                  GetFolders(site.groupId, 0)
6                  GetFileEntries(site.groupId, 0)
7                   Assign 1 to site.lastAccessDate
8                  Jump to 23.
9          Get local entries marked as "pending upload"
10         For each "pending upload" entry
11                 If entry.fileId = -1
12                         Request "add-file-entry"
13                 Else
14                         Request "update-file-entry"
15         Request "get-dl-sync-update" and store result in DLSyncUpdate remoteUpdates
16         For DLSync remoteEntry in remoteUpdates and remoteEntry.type = "folder"
17                 HandleDLSync(remoteEntry, site.groupId)
18         For DLSync remoteEntry in remoteUpdates and remoteEntry.type = "file"
19                 HandleDLSync(remoteEntry, site.groupId)
20         Assign remoteUpdates.lastAccessDate to site.lastAccessDate
21         Notify the UI that site is synced
22 Get local entries marked as "pending download" and start the Downloader service
23 Notify the UI that the sync event has completed
```

Listing 2.16: Synchronization algorithm: GetFolders

```
GetFolders(repositoryId, parentFolderId)

Request "get-folders" and store result in List$\langle$ Folder$\rangle$ folders
For each Folder folder in folders where folder.folderId $\rangle$ 0
        If local entry for folder does not exist
                Create local entry
                Create folder in cache storage
        Update local entry and mark it "downloaded"
        GetFolders(repositoryId, folder.folderId)
        GetFileEntries(repositoryId, folder.folderId)
        Notify UI that folder is synced
```

Listing 2.17: Synchronization algorithm: GetFileEntries

```
GetFileEntries(repositoryId, folderId)

Request "get-file-entries" and store result in List$\langle$ FileEntry$\rangle$
fileEntries
For each FileEntry fileEntry in fileEntries
        If local entry for fileEntry does not exist
                Create local entry
        If fileEntry.version $\rangle$ localFileEntry.version and localFileEntry.keepInSync
                Update local entry and mark it "pending download"
        Else
                Mark local entry "downloaded"
```

Listing 2.18: Synchronization algorithm: HandleDLSync

```
HandleDLSync(remoteEntry, repositoryId)

Get local entry DLFile localEntry from remoteEntry.fileId and repositoryId
If remoteEntry.event = "delete"
        PerformDelete(localEntry)
        Return
If localEntry does not exist
        If remoteEntry.type = "folder"
                HandleFolder(localEntry, remoteEntry)
        Else
                HandleFileContent(localEntry, remoteEntry)
        Return
If localEntry.filePath != remoteEntry.filePath
        PerformRename(localEntry, remoteEntry)
If remoteEntry.version $\rangle$ localEntry.version
```

```
      HandleFileContent(localEntry, remoteEntry)
If localEntry.directory = true
      HandleFolder(localEntry, remoteEntry)
```

Listing 2.19: Synchronization algorithm: HandleFileContent

```
HandleFileContent(localEntry, remoteEntry)

If localEntry does not exist
      Create local entry
Update localEntry with remoteEntry data
If localEntry.keepInSync = true
      Mark localEntry "pending download"
```

Listing 2.20: Synchronization algorithm: HandleFolder

```
HandleFolder(localEntry, remoteEntry)

If localEntry does not exist
      Create local entry with type folder
      Create folder in cache storage
Update localEntry with remoteEntry data
Mark localEntry "downloaded"
```

Listing 2.21: Synchronization algorithm: PerformRename

```
PerformRename(localEntry, remoteEntry)

Rename cached content
If localEntry.parentId != remoteEntry.parentId
      Move cached content
Update localEntry with remoteEntry data
Recursively update children's paths
```

Listing 2.22: Synchronization algorithm: PerformDelete

```
PerformDelete(localEntry)

If localEntry.directory = true
       Recursively delete children's entries and cached contents
Delete localEntry and respective cached content
```

In the next sections we will define the all the object models involved in this algorithm and explain how the HTTP requests to the server's Web Services are carried out by the LiferayDLClient component.

2.4 Logical file representation and persistence

The second most important aspect in the SyncAdapter pattern, after the synchronization algorithm, is the management and persistence of synchronized data. The diagram in Figure 2-3, depicts the overall process of retrieving data from the server and storing it on the client device for local use.

As we have seen earlier, there are two main types of logical data to keep in sync (apart from the contents themselves), which are user sites and file entries. These data are retrieved from the Liferay Portal server through its Web Services API as four different types of responses in JSON format, depending on the API method invoked. JSON is open standard alternative to XML, designed for language-independent, lightweight data interchange. It is derived from the JavaScript scripting language for representing objects as simple data structures and associative arrays.

In order for the Android client to manage and persist these data (in a ContentProvider), every JSON message must be parsed and converted to a Java object of the corresponding type. The responses differ for the fact that their JSON representations have different field names, although they can still be mapped to a single object type. In fact, we have the following method to object mappings:

- "get-user-sites" returns an array of user sites. A user site is converted to a UserSite object;
- "get-dl-sync-update" is converted to a DLSyncUpdate, which contains an array of sync entries, in turn converted to corresponding DLSync objects;
- "get-file-entries" returns an array of JSON structures that are converted to FileEntry objects;
- "get-folders" is mapped to Folder objects.

The JSON to Java object conversion is carried out by the "google-gson" library [34].

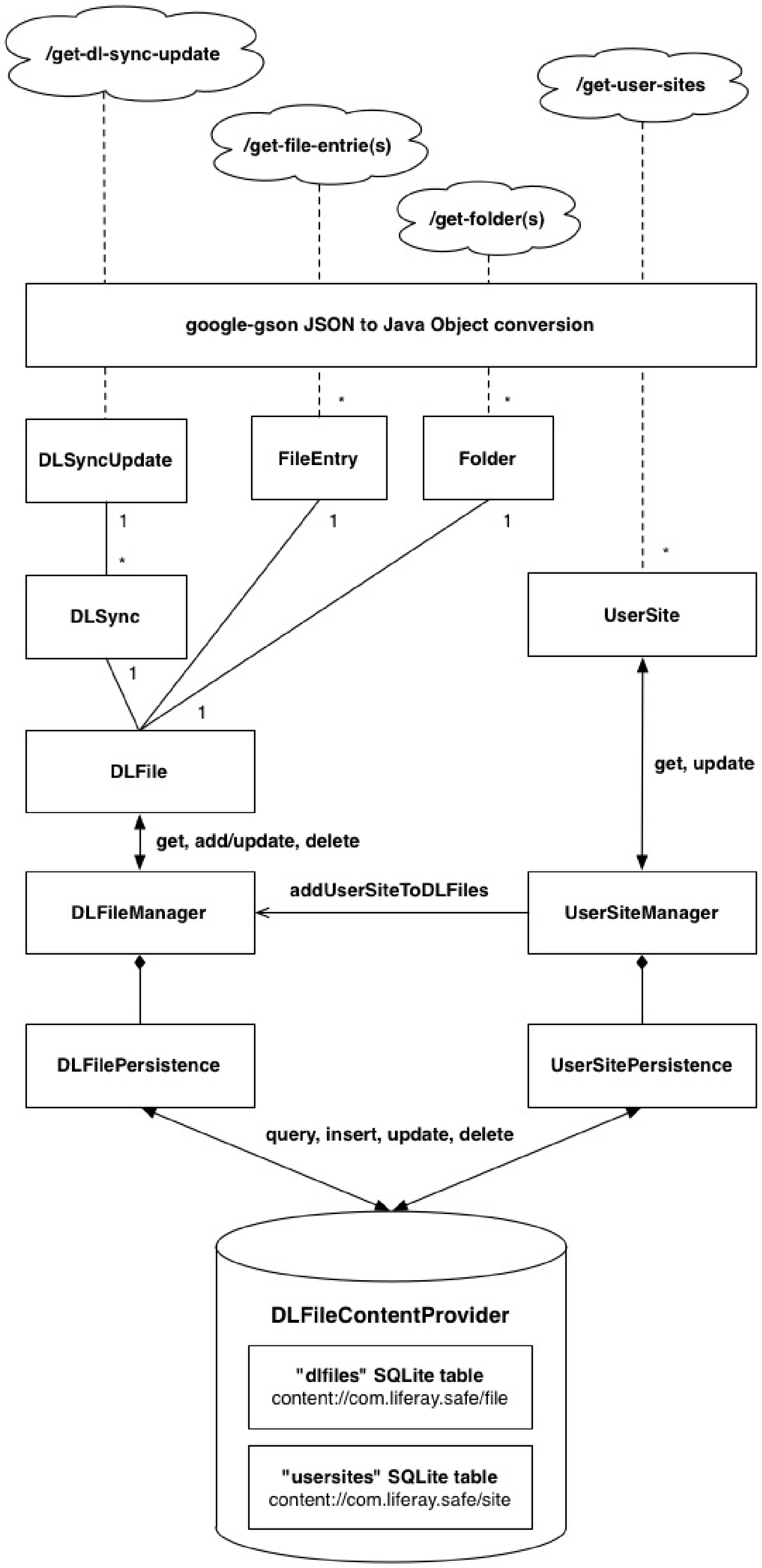


Figure 2-3: Data management and persistence

The following snippet of code shows how a “get-user-sites” response entity is converted to a list of UserSite objects:

Listing 2.23: JSON to Java object conversion

```
InputStream instream = entity.getContent();
Gson gson = new Gson();
Reader reader = new InputStreamReader(instream);
Type collectionType = new TypeToken$\langle$ List$\langle$ UserSite$\rangle$ $\rangle$ () {}.getType();
List$\langle$ UserSite$\rangle$ userSites = gson.fromJson(reader, collectionType);
reader.close();
instream.close();
```

The use of the library requires no additional step other than defining the class of the Java object itself, as long as the field names correspond exactly to the ones of the JSON object. When they are not, it’s sufficient to annotate the Java fields with the name of the corresponding JSON fields, for example:

Listing 2.24: @SerializedName annotation

```
@SerializedName("DLSyncs")
private List$\langle$ DLSync$\rangle$ dlSyncs;
```

This maps “DLSyncs” from JSON to the dlSyncs field of the DLSyncUpdate Java object.

Since FileEntry, Folder and DLSync objects in the end all refer to a file entry, in Liferay Safe all these data are simplified to two main object types: a file entry is represented by a DLFile, while a user site still by a UserSite. The class diagram below illustrates the data model package which includes all the objects just discussed.

Hence a DLFile is the local representation of a file entry stored in the “Documents and Media Library” of a Liferay Portal server. A DLFile has also a fileState property that reflects the synchronization state of the file content it is associated to. The state is a value from an enumeration (DLFileState) and describes either a transition from a “pending” download/upload to a “completed” download/upload or an “error” occurred during an upload. By default the state of a DLFile is NONE, and this means that the corresponding file content is not available yet in the local cache storage.

DLFile and UserSite objects are persisted in a ContentProvider (DLFileContentProvider), which manages a SQLite database holding two tables: the “dlfiles” table for DLFile entries and the “usersites” table for UserSite entries.

All the application logic related to these two types of information, including the access to the ContentProvider, is implemented in a manager-persistence pattern.

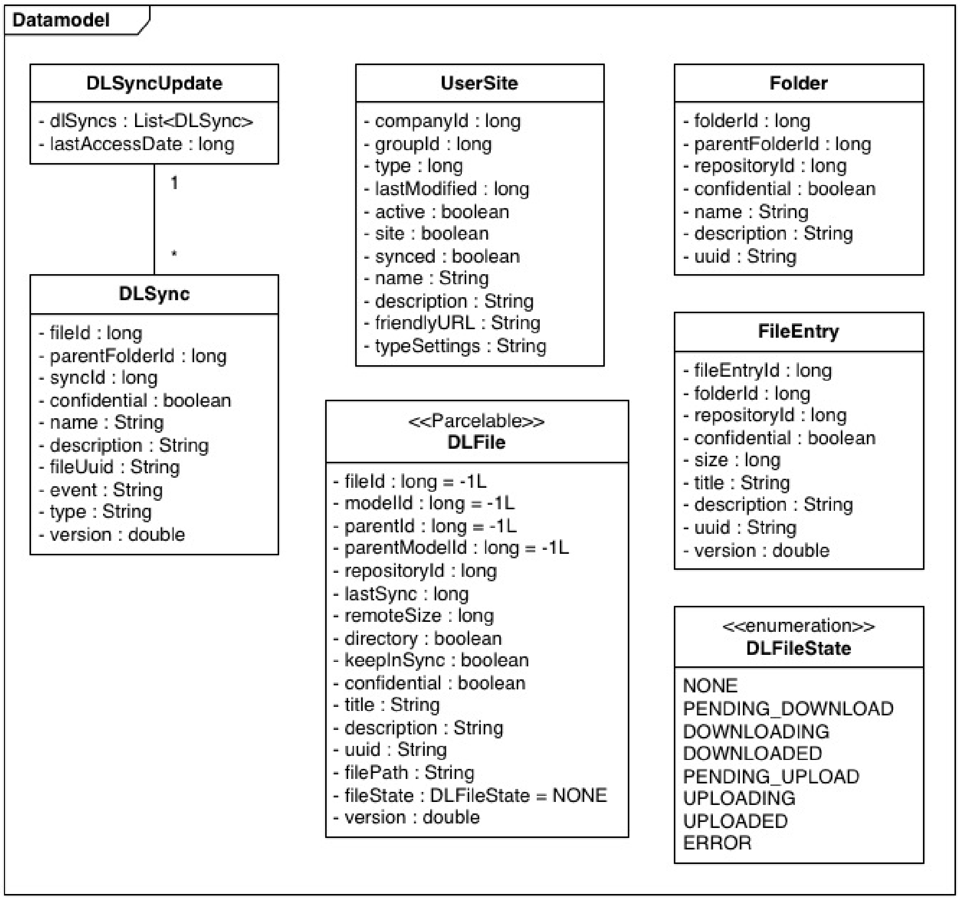


Figure 2-4: Data model

The manager object has the responsibility of performing all the operations defined on the type of model object it is managing. The manager also guarantees that any modification made is immediately persisted on the ContentProvider. For this purpose, it's tightly coupled to a persistence object which has the sole responsibility of calling the ContentProvider's API to persist the modifications. In fact, the manager (or any other application component) never talks directly to the ContentProvider, only the persistence object is allowed to.

DLFileManager and DLFilePersistence manage and persist DLFile objects, while UserSiteManager and UserSitePersistence manage and persist UserSite objects. Their operations are listed in the class diagrams reported below in the next pages.

Finally, since the first screen of the application has to list the sites available at the host Liferay Portal, which work as different content repositories, for convenience, a UserSite is also persisted as a DLFile. The UserSiteManager calls the DLFileManager's addUserSiteToDLFiles() method passing the UserSite to be converted to a DLFile and stored in the corresponding ContentProvider's table.

2.4.1 The Content Provider

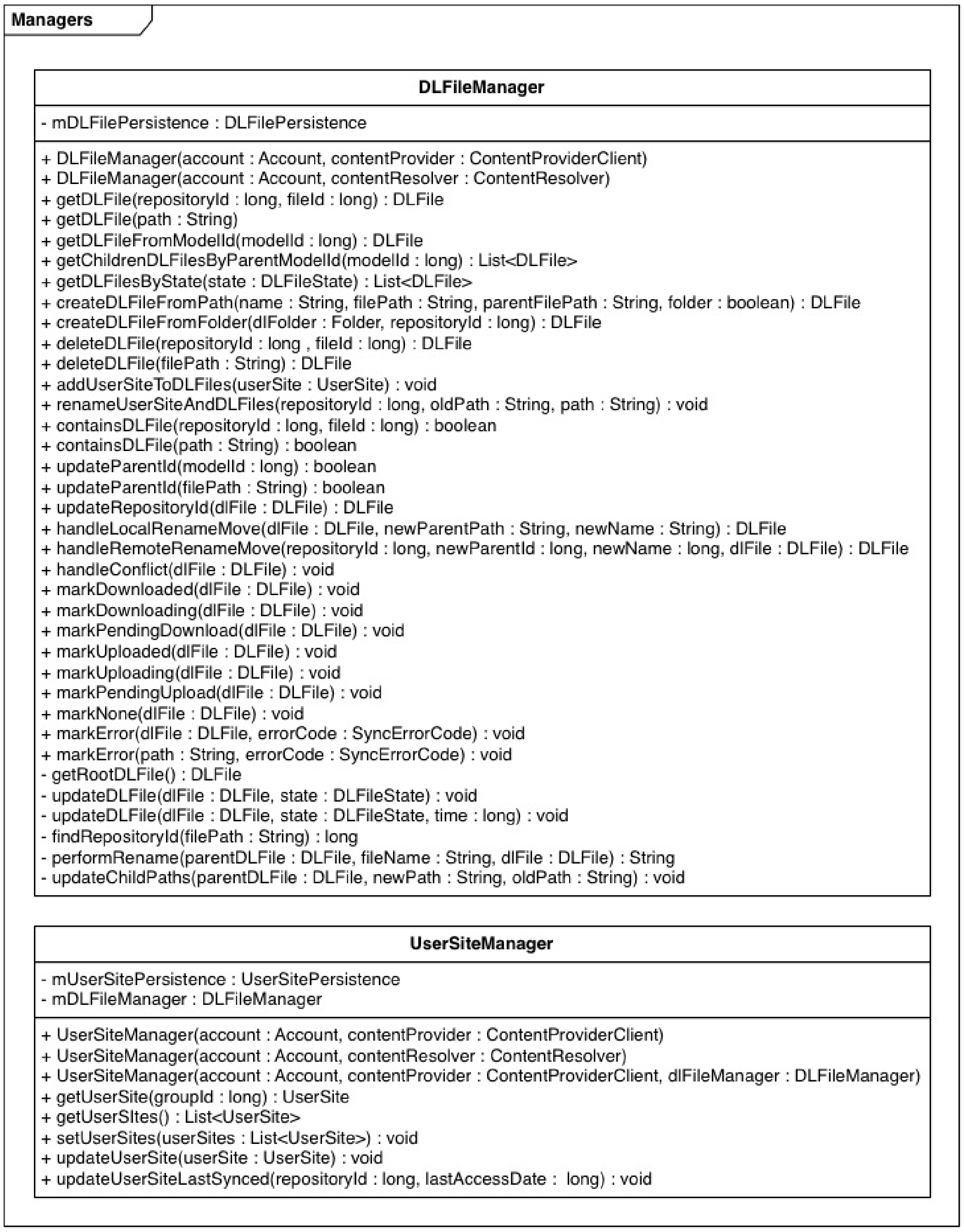
The DLFileContentProvider is a custom implementation of the abstract ContentProvider class providing an abstraction from the underlying SQLite database where UserSite and DLFile models are stored in separate SQLite tables.

Content providers support the four basic operations, normally called CRUD-operations. CRUD is the acronym for create, read, update and delete. DLFileContentProvider extends ContentProvider and implements the interface:

- onCreate() which is called to initialize the provider;
- query(Uri, String[], String, String[], String) which returns data to the caller;
- insert(Uri, ContentValues) which inserts new data into the content provider;
- update(Uri, ContentValues, String, String[]) which updates existing data in the content provider;
- delete(Uri, String, String[]) which deletes data from the content provider;
- getType(Uri) which returns the MIME type of data in the content provider.

The persistence classes (DLFilePersistence and UserSitePersistence) do not always use the Content Provider directly (for example, when the interaction started from an Activity), rather they use the Content Resolver.

The Content Resolver is a single, global instance in an application that provides access to content providers. It accepts requests from clients, and resolves these requests by directing the them to the content provider with the given authority. To do this, the Content Resolver stores a mapping from authorities to Content Providers. The ContentResolver class includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, delete, query, update) in the ContentProvider class.



```
DLFileManager
- mDLFilePersistence : DLFilePersistence
+ DLFileManager(account : Account, contentProvider : ContentProviderClient)
+ DLFileManager(account : Account, contentResolver : ContentResolver)
+ getDLFile(repositoryId : long, fileId : long) : DLFile
+ getDLFile(path : String)
+ getDLFileFromModelId(modelId : long) : DLFile
+ getChildrenDLFilesByParentModelId(modelId : long) : List<DLFile>
+ getDLFilesByState(state : DLFileState) : List<DLFile>
+ createDLFileFromPath(name : String, filePath : String, parentFilePath : String, folder : boolean) : DLFile
+ createDLFileFromFolder(dlFolder : Folder, repositoryId : long) : DLFile
+ deleteDLFile(repositoryId : long , fileId : long) : DLFile
+ deleteDLFile(filePath : String) : DLFile
+ addUserSiteToDLFiles(userSite : UserSite) : void
+ renameUserSiteAndDLFiles(repositoryId : long, oldPath : String, path : String) : void
+ containsDLFile(repositoryId : long, fileId : long) : boolean
+ containsDLFile(path : String) : boolean
+ updateParentId(modelId : long) : boolean
+ updateParentId(filePath : String) : boolean
+ updateRepositoryId(dlFile : DLFile) : DLFile
+ handleLocalRenameMove(dlFile : DLFile, newParentPath : String, newName : String) : DLFile
+ handleRemoteRenameMove(repositoryId : long, newParentId : long, newName : long, dlFile : DLFile) : DLFile
+ handleConflict(dlFile : DLFile) : void
+ markDownloaded(dlFile : DLFile) : void
+ markDownloading(dlFile : DLFile) : void
+ markPendingDownload(dlFile : DLFile) : void
+ markUploaded(dlFile : DLFile) : void
+ markUploading(dlFile : DLFile) : void
+ markPendingUpload(dlFile : DLFile) : void
+ markNone(dlFile : DLFile) : void
+ markError(dlFile : DLFile, errorCode : SyncErrorCode) : void
+ markError(path : String, errorCode : SyncErrorCode) : void
- getRootDLFile() : DLFile
- updateDLFile(dlFile : DLFile, state : DLFileState) : void
- updateDLFile(dlFile : DLFile, state : DLFileState, time : long) : void
- findRepositoryId(filePath : String) : long
- performRename(parentDLFile : DLFile, fileName : String, dlFile : DLFile) : String
- updateChildPaths(parentDLFile : DLFile, newPath : String, oldPath : String) : void

UserSiteManager
- mUserSitePersistence : UserSitePersistence
- mDLFileManager : DLFileManager
+ UserSiteManager(account : Account, contentProvider : ContentProviderClient)
+ UserSiteManager(account : Account, contentResolver : ContentResolver)
+ UserSiteManager(account : Account, contentProvider : ContentProviderClient, dlFileManager : DLFileManager)
+ getUserSite(groupId : long) : UserSite
+ getUserSItes() : List<UserSite>
+ setUserSites(userSites : List<UserSite>) : void
+ updateUserSite(userSite : UserSite) : void
+ updateUserSiteLastSynced(repositoryId : long, lastAccessDate :  long) : void
```

Figure 2-5: Data management classes

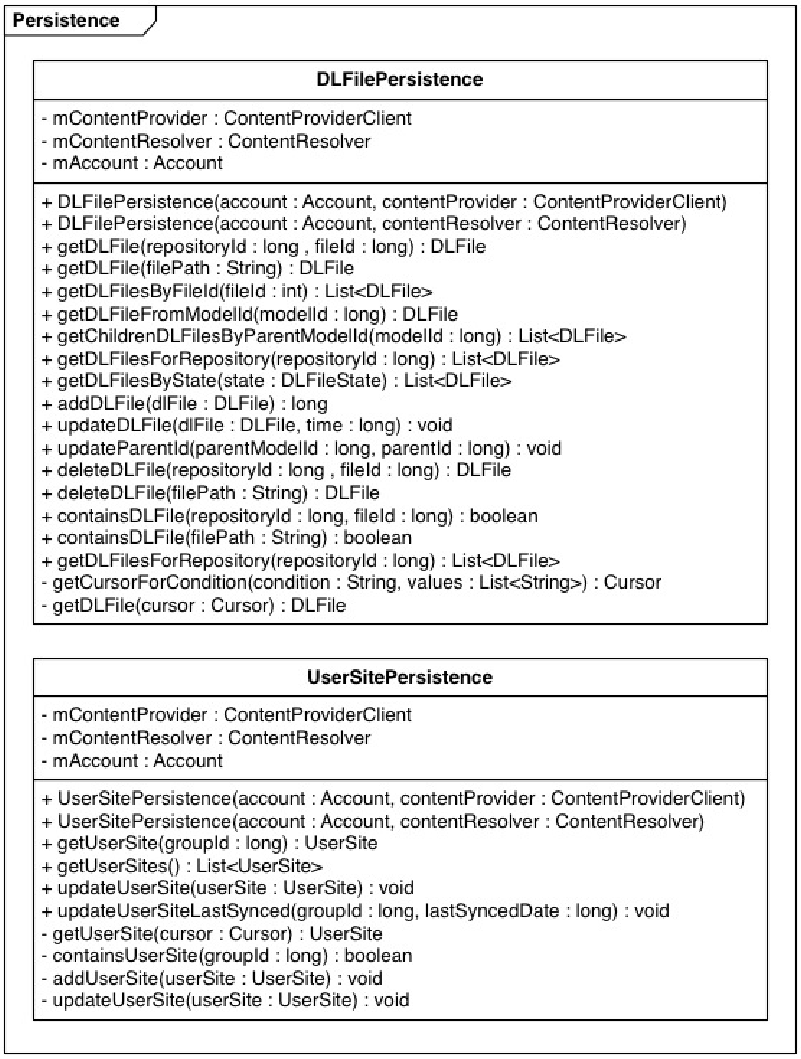

Persistence
DLFilePersistence
- mContentProvider : ContentProviderClient
- mContentResolver : ContentResolver
- mAccount : Account
+ DLFilePersistence(account : Account, contentProvider : ContentProviderClient)
+ DLFilePersistence(account : Account, contentResolver : ContentResolver)
+ getDLFile(repositoryId : long , fileId : long) : DLFile
+ getDLFile(filePath : String) : DLFile
+ getDLFilesByFileId(fileId : int) : List<DLFile>
+ getDLFileFromModelId(modelId : long) : DLFile
+ getChildrenDLFilesByParentModelId(modelId : long) : List<DLFile>
+ getDLFilesForRepository(repositoryId : long) : List<DLFile>
+ getDLFilesByState(state : DLFileState) : List<DLFile>
+ addDLFile(dlFile : DLFile) : long
+ updateDLFile(dlFile : DLFile, time : long) : void
+ updateParentId(parentModelId : long, parentId : long) : void
+ deleteDLFile(repositoryId : long , fileId : long) : DLFile
+ deleteDLFile(filePath : String) : DLFile
+ containsDLFile(repositoryId : long, fileId : long) : boolean
+ containsDLFile(filePath : String) : boolean
+ getDLFilesForRepository(repositoryId : long) : List<DLFile>
- getCursorForCondition(condition : String, values : List<String>) : Cursor
- getDLFile(cursor : Cursor) : DLFile
UserSitePersistence
- mContentProvider : ContentProviderClient
- mContentResolver : ContentResolver
- mAccount : Account
+ UserSitePersistence(account : Account, contentProvider : ContentProviderClient)
+ UserSitePersistence(account : Account, contentResolver : ContentResolver)
+ getUserSite(groupId : long) : UserSite
+ getUserSites() : List<UserSite>
+ updateUserSite(userSite : UserSite) : void
+ updateUserSiteLastSynced(groupId : long, lastSyncedDate : long) : void
- getUserSite(cursor : Cursor) : UserSite
- containsUserSite(groupId : long) : boolean
- addUserSite(userSite : UserSite) : void
- updateUserSite(userSite : UserSite) : void

Figure 2-6: Data persistence classes

The Content Resolver does not know the implementation of the Content Providers it is interacting with (nor does it need to know); each method is passed an URI that specifies the Content Provider to interact with.

Data sets of a Content Provider are known by their URI. Each URI uniquely identifies a data set (a single table), or even a specific record in the data set. It's necessary to specify a URI whenever data needs to be accessed from a Content Provider.

URIs for Content Providers have a standardized format that is structured in four parts: `content://authority/optionalPath/optionalId`

- The first part is the **scheme**, that for Content Providers is always "content".
- The next part is the **authority** for the Content Provider. Authorities have to be unique for every content provider. Thus the naming conventions should follow the Java package name rules.
- The third part, the **optional path**, is used to distinguish the kinds of data in a Content Provider. This way a content provider can support different types of data that should be related. If the URI ends with this part it is called a directory-based URI. They are used to access multiple elements of the same type.
- The last element is the **optional id**, a numeric value used to access a single record. If the underlying store is a SQLite database, this corresponds to the mandatory numeric _ID field that uniquely identifies a record within a table. URIs that include this part are called id-based URIs.

Besides defining the content URI patterns, Content Providers based on structured data have to define also their **content types**. Content types are MIME types in Android's vendor-specific MIME format, which consists of three parts:

- Type part: **vnd**
- Subtype part:
  - If the URI pattern is for a single row: **android.cursor.item/**
  - If the URI pattern is for more than one row: **android.cursor.dir/**
- Provider-specific part: **vnd.⟨ name⟩ .⟨ type⟩**
  - The ⟨ name⟩ value should be globally unique, and the ⟨ type⟩ value should be unique to the corresponding URI pattern. A good choice for ⟨ name⟩ is the company's name or some part of the application's Android package name. A good choice for the ⟨ type⟩ is a string that identifies the table associated with the URI.

Implementing a Content Provider working on a SQLite database involves the following steps:

1. Create a class that extends ContentProvider;
2. Define the authority, URIs and database attributes;
3. Create constants for table name and columns;
4. Create a class that extends SQLiteOpenHelper;
5. Implement the getType() method;
6. Implement the CRUD methods;
7. Add the Content Provider to the AndroidManifest.xml.

Based on the defined data model, in the next pages we will expand on all these steps. We assume that the DLFileContentProvider class that extends ContentProvider was already created, therefore we focus on the remaining steps.

Define Content Provider attributes and table constants

Since there isn't a common standard for dealing with these attributes and constants, we opted to define a ProviderMeta class as a container for the authority, database attributes, and two inner classes defining URIs, table name and table column names for the two subtypes of our provider: DLFile and UserSite. The class is reported below:

Listing 2.25: ProviderMeta

```java
public class ProviderMeta {

    public static final String AUTHORITY = "com.liferay.safe";
    public static final String DB_FILE = "liferaysafe.db";
    public static final String DB_NAME = "liferaysafe";
    public static final int DB_VERSION = 1;

        public static final Uri CONTENT_URI = Uri.parse("content://"
            + AUTHORITY + "/");

    static public class DLFileTableMeta implements BaseColumns {
        public static final String FILE_TABLE = "dlfiles";

        public static final Uri CONTENT_URI_FILE = Uri.parse("content://"
                + AUTHORITY + "/file");
        public static final Uri CONTENT_URI_FOLDER = Uri.parse("content://"
                + AUTHORITY + "/folder");

        public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.liferay.safe.file";
        public static final String CONTENT_TYPE_ITEM = "vnd.android.cursor.item/vnd.liferay.safe.file";

        public static final String FILE_DESCRIPTION = "description";
        public static final String FILE_IS_DIRECTORY = "is_directory";
        public static final String FILE_ID = "file_id";
        public static final String FILE_PATH = "path";
        public static final String FILE_STATE = "state";
        public static final String FILE_LAST_SYNC = "last_sync";
                public static final String FILE_KEEP_IN_SYNC = "keep_in_sync";
                public static final String FILE_CONFIDENTIAL = "confidential";
        public static final String FILE_PARENT_ID = "parent_id";
        public static final String FILE_PARENT_MODEL_ID = "model_parent_id";
        public static final String FILE_SIZE = "size";
        public static final String FILE_REPOSITORY_ID = "repository_id";
        public static final String FILE_TITLE = "title";
        public static final String FILE_UUID = "uuid";
        public static final String FILE_VERSION = "version";
        public static final String FILE_ACCOUNT_OWNER = "account";

        public static final String DEFAULT_SORT_ORDER = FILE_TITLE
                + " collate nocase asc";
    }

    public static class UserSiteTableMeta implements BaseColumns {
        public static final String SITE_TABLE = "usersites";

        public static final Uri CONTENT_URI_SITE = Uri.parse("content://"
                + AUTHORITY + "/site");

        public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.liferay.safe.site";
        public static final String CONTENT_TYPE_ITEM = "vnd.android.cursor.item/vnd.liferay.safe.site";

        public static final String SITE_GROUP_ID = "group_id";
        public static final String SITE_ACTIVE = "active";
        public static final String SITE_COMPANY_ID = "company_id";
        public static final String SITE_DESCRIPTION = "description";
        public static final String SITE_FRIENDLY_URL = "friendly_url";
        public static final String SITE_NAME = "name";
        public static final String SITE_SITE = "site";
        public static final String SITE_SOCIAL_OFFICE_SITE = "social_office_site";
        public static final String SITE_SYNCED = "synced";
        public static final String SITE_TYPE = "type";
        public static final String SITE_TYPE_SETTINGS = "type_settings";
        public static final String SITE_LAST_MODIFIED = "last_modified";
        public static final String SITE_ACCOUNT_OWNER = "account";

        public static final String DEFAULT_SORT_ORDER = SITE_NAME
                + " collate nocase asc";
    }
}
```

Essentially we have defined two different content types and three different content URIs:

- For DLFile data:
  - vnd.android.cursor.item/vnd.liferay.safe.file
  - vnd.android.cursor.dir/vnd.liferay.safe.file
  - content://com.liferay.safe/file
  - content://com.liferay.safe/folder
- For UserSite data:
  - vnd.android.cursor.item/vnd.liferay.safe.site
  - vnd.android.cursor.dir/vnd.liferay.safe.site
  - content://com.liferay.safe/site

To deal with these multiple URIs Android provides the helper class UriMatcher, which eases the parsing of URIs. In the DLFileProvider class the UriMatcher is initialized by adding a set of paths with corresponding int values. The UriMatcher is very important when implementing the CRUD methods, because it allows to detect the type of data is being queried, inserted, updated or deleted, and consequently set the correct table name, where clause and arguments of the corresponding SQL operation.

The following code snippet shows how the UriMatcher is defined:

Listing 2.26: UriMatcher

```
private static final int FILE = 1;
private static final int FILE_WITH_ID = 2;
private static final int FOLDER = 3;
private static final int SITE = 4;

private static final UriMatcher mUriMatcher;
static {
        mUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        mUriMatcher.addURI(ProviderMeta.AUTHORITY, "file/", FILE);
        mUriMatcher.addURI(ProviderMeta.AUTHORITY, "file/#", FILE_WITH_ID);
        mUriMatcher.addURI(ProviderMeta.AUTHORITY, "folder/#", FOLDER);
        mUriMatcher.addURI(ProviderMeta.AUTHORITY, "site/", SITE);
}
```

Whenever it is asked if a URI matches, the UriMatcher returns the corresponding int-value to indicate which one matches. It is common practise to use constants for these int-values in order to use them with switch statements inside the methods of the Content Provider.

Create a class that extends SQLiteOpenHelper

The SQLiteOpenHelper is an helper class used to manage database creation and version management. A subclass can be created by implementing onCreate, onUpgrade and optionally onOpen, and this class takes care of opening the database if it exists, creating it if it does not, and upgrading it as necessary.

We created the DataBaseHelper subclass as an inner class of the DLFileContentProvider class, and implemented the onCreate() method to create the two database tables using the constants defined in the previous step. Below is the source code of the helper class.

Listing 2.27: DataBaseHelper

```
class DataBaseHelper extends SQLiteOpenHelper {

        public DataBaseHelper(Context context) {
                super(context, ProviderMeta.DB_NAME, null, ProviderMeta.DB_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
                db.execSQL("CREATE TABLE " + DLFileTableMeta.FILE_TABLE + "("
                                + DLFileTableMeta._ID + " INTEGER PRIMARY KEY, "
                                + DLFileTableMeta.FILE_ACCOUNT_OWNER + " TEXT, "

                                + DLFileTableMeta.FILE_DESCRIPTION + " TEXT, "
                                + DLFileTableMeta.FILE_IS_DIRECTORY + " INTEGER, "
                                + DLFileTableMeta.FILE_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_PATH + " TEXT, "
                                + DLFileTableMeta.FILE_STATE + " INTEGER, "
                                + DLFileTableMeta.FILE_LAST_SYNC + " INTEGER, "
+ DLFileTableMeta.FILE_KEEP_IN_SYNC + " INTEGER, "
+ DLFileTableMeta.FILE_CONFIDENTIAL + " INTEGER, "
                                + DLFileTableMeta.FILE_PARENT_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_PARENT_MODEL_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_SIZE + " INTEGER, "
                                + DLFileTableMeta.FILE_REPOSITORY_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_TITLE + " TEXT, "
                                + DLFileTableMeta.FILE_UUID + " TEXT, "
                                + DLFileTableMeta.FILE_VERSION + " REAL );");

                db.execSQL("CREATE TABLE " + UserSiteTableMeta.SITE_TABLE + "("
                                + UserSiteTableMeta._ID + " INTEGER PRIMARY KEY, "
                                + UserSiteTableMeta.SITE_ACCOUNT_OWNER + " TEXT, "

                                + UserSiteTableMeta.SITE_GROUP_ID + " INTEGER, "
                                + UserSiteTableMeta.SITE_ACTIVE + " INTEGER, "
                                + UserSiteTableMeta.SITE_COMPANY_ID + " INTEGER, "
                                + UserSiteTableMeta.SITE_DESCRIPTION + " TEXT, "
                                + UserSiteTableMeta.SITE_FRIENDLY_URL + " TEXT, "
                                + UserSiteTableMeta.SITE_NAME + " TEXT, "
                                + UserSiteTableMeta.SITE_SITE + " INTEGER, "
                                + UserSiteTableMeta.SITE_SOCIAL_OFFICE_SITE + " INTEGER, "
                                + UserSiteTableMeta.SITE_SYNCED + " INTEGER, "
                                + UserSiteTableMeta.SITE_TYPE + " INTEGER, "
                                + UserSiteTableMeta.SITE_TYPE_SETTINGS + " TEXT, "
                                + UserSiteTableMeta.SITE_LAST_MODIFIED + " INTEGER );");
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
                // Nothing to do here, no previous version exists.
        }

}
```

Note that there is no native boolean data type for SQLite, so it's necessary to use integer values instead.

A reference to the DataBaseHelper is initialized in the provider's onCreate() method, and is used in the CRUD methods to get an instance of the database, in read-only or writeable mode.

Implement the getType() and CRUD methods

Every content provider must return the content type for its supported URIs, this is done by implementing the getType() method as follows:

Listing 2.28: ContentProvider getType()

```
public String getType(Uri uri) {
        switch (mUriMatcher.match(uri)) {
        case FILE: case FOLDER:
                return DLFileTableMeta.CONTENT_TYPE;
        case FILE_WITH_ID:
                return DLFileTableMeta.CONTENT_TYPE_ITEM;
        case SITE:
                return UserSiteTableMeta.CONTENT_TYPE;
        default:
                throw new IllegalArgumentException("Unknown Uri id."
                                        + uri.toString());
        }
}
```

Note that content URIs for files and folders are associated to the same content type, since they are both modeled as a DLFile.

To give an example of implementation of a CRUD method we take the code of the update operation, which we report below.

Listing 2.29: ContentProvider update(

```
@Override
public int update(Uri uri, ContentValues values, String selection,
                        String[] selectionArgs) {

        int updateCount = 0;
        String table;
        switch (mUriMatcher.match(uri)) {
        case FILE:
                table = DLFileTableMeta.FILE_TABLE;
                break;
        case SITE:
                table = UserSiteTableMeta.SITE_TABLE;
                break;
        default:
                throw new IllegalArgumentException("Unknown uri id: " + uri);
        }

        updateCount = mDbHelper.getWritableDatabase().update(
                        table, values, selection, selectionArgs);
        if (updateCount > 0) {
                getContext().getContentResolver().notifyChange(uri, null);
        }
        return updateCount;
}
```

In order to notify other application components when there has been a change in the underlying datastore of a content provider it's necessary to notify the Content Resolver with notifyChange(), passing the URI of the modified record. This notification should be triggered as well when deleting or inserting data.

The CRUD methods are called directly on an instance of the Content Provider (usually when the call originated is the SyncAdapter) or through the ContentResolver. The persistence (DLFilePersistence and UserSitePersistence) will be responsible of performing the call and deal with the necessary paramaters. As an example, below is reported the updateDLFile() method of the DLFilePersistence class.

Listing 2.30: DLFileManager updateDLFile()

```
public void updateDLFile(DLFile dlFile, long time)
                        throws RemoteException {

        ContentValues cv = new ContentValues();
       cv.put(DLFileTableMeta.FILE_DESCRIPTION, dlFile.getDescription());
       cv.put(DLFileTableMeta.FILE_IS_DIRECTORY, dlFile.isDirectory());
       cv.put(DLFileTableMeta.FILE_ID, dlFile.getFileId());
       cv.put(DLFileTableMeta.FILE_PATH, dlFile.getFilePath());
       cv.put(DLFileTableMeta.FILE_STATE, dlFile.getFileState().ordinal());
        cv.put(DLFileTableMeta.FILE_LAST_SYNC, time);
       cv.put(DLFileTableMeta.FILE_PARENT_ID, dlFile.getParentId());
       cv.put(DLFileTableMeta.FILE_PARENT_MODEL_ID, dlFile.getParentModelId());
       cv.put(DLFileTableMeta.FILE_SIZE, dlFile.getRemoteSize());
       cv.put(DLFileTableMeta.FILE_REPOSITORY_ID, dlFile.getRepositoryId());
       cv.put(DLFileTableMeta.FILE_TITLE, dlFile.getTitle());
       cv.put(DLFileTableMeta.FILE_UUID, dlFile.getUuid());
       cv.put(DLFileTableMeta.FILE_VERSION, dlFile.getVersion());

       if (mContentProvider != null)
               mContentProvider.update(DLFileTableMeta.CONTENT_URI_FILE,
                                cv,
                                DLFileTableMeta._ID + "=? AND " +
                                DLFileTableMeta.FILE_ACCOUNT_OWNER + "=?",
                                new String[] { String.valueOf(dlFile.getModelId()),
 mAccount.name });
        else
               mContentResolver.update(DLFileTableMeta.CONTENT_URI_FILE,
                                cv,
                                DLFileTableMeta._ID + "=? AND " +
                                DLFileTableMeta.FILE_ACCOUNT_OWNER + "=?",
                                new String[] { String.valueOf(dlFile.getModelId()),
 mAccount.name });
}
```

Add the Content Provider to the AndroidManifest.xml

The last step is to register the Content Provider within the AndroidManifest.xml file. The next code snippet shows how this is done.

Listing 2.31: Manifest provider tag

```
$\langle$ provider
          android:name=".provider.DLFileContentProvider"
```

```
            android:authorities="com.liferay.safe"
            android:enabled="true"
            android:exported="false"
            android:label="@string/sync_string_files"
            android:syncable="true" $\rangle$
$\langle$ /provider$\rangle$
```

The Content Provider is not exported, which means that only components of the same application, or applications that have the same user ID as the provider will have access to it.

The android:syncable property tells that the provider can be synced using a SyncAdapter. The android:authorities property is important to tie the provider with the SyncAdapter, and in fact must have the same value of android:contentAuthority declared in the ⟨ sync-adapter /⟩ component.

Although Content Providers should be used to share data with other applications, there cannot be a SyncAdapter without a ContentProvider and an AccountManager, these three components must be tied together.

Additionally there are a number of benefits to using ContentProvider, such as:

- ContentProvider schedules the database access in a background thread, preventing ANR errors while not requiring you to explicitly handle threading [25].
- ContentProvider ties into ContentResolver's observer: this means it is easy to notify views when content is changed [25].
- It enables to decouple application layers from the underlying data layers, making the application data-source agnostic by abstracting the underlying data source [55].
- It provides mechanisms for defining data security (i.e. by enforcing read/write permissions) and offer a standard interface that connects data in one process with code running in another process [55].

2.5 JSON Web Services API

In Liferay Portal JSON Web Services provide convenient access to portal service methods by exposing them as JSON HTTP API. This makes service methods easily accessible using HTTP requests from any JSON-speaking client [35].

JSON (JavaScript Object Notation) is a lightweight data-interchange text format. JSON is both easy for humans to read and write, and easy for machines to parse and generate. Although it is based on a subset of the JavaScript Programming Language, JSON is completely language independent [36].

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

Each service method registered in the JSON Web Services, is bound to one HTTP method type. All service methods having names starting with get, is or has are assumed to be read-only methods and are therefore mapped as GET HTTP methods, by default. All other service methods are mapped as POST HTTP methods. By default, Liferay Portal works in “non-strict HTTP method” mode; this means that it does not check HTTP methods when invoking a service call, and services may be invoked using any HTTP method.

Furthermore, each service method has the responsibility to determine for itself whether it can be executed by unauthenticated/unauthorized users and whether a user has adequate permission for the chosen action. Most of portal’s read-only methods are open to public access.

JSON Web Services can be invoked simply by passing parameters as request parameters directly in the URL. Parameter values are sent as strings using the HTTP protocol. Before a matching Java service method is invoked, each parameter value is converted from a string to its target Java type. Except for methods returning a java.io.InputStream, any returned objects are serialized to a JSON string and returned to the caller.

In the following paragraphs, we list all the service methods used by Liferay Safe, and introduce the HTTP client component mentioned in the SyncAdapter pattern, which is responsible for communicating with the server. We will also provide the sample code of an API invocation.

2.5.1 Currently used API methods

In Liferay Portal a list of all available JSON Web Services can be retrieved from http://localhost:8080/api/jsonw
The resulting page lists all registered and exposed service methods of the portal.

As already discussed at the beginning of this chapter, for security constraints we require that every invocation from the Liferay Safe client to the server’s JSON API must be verified and authorized by a portlet implementing an OAuth 2.0 provider & resource server.

In fact, a method exposed by the Document and Media Library services, for example, will never be invoked directly by Liferay Safe, but instead, the client will invoke the exact same method exposed by the Provider Portlet services, passing also the required OAuth parameters. If the client is authorized, the call will be unmarshaled and forwarded to the corresponding Java method of the Local Services API. The response will be returned to the client again from the Provider Portlet, which must first convert the result of the local service call to JSON.

Hence, the JSON HTTP methods of the Provider are actually wrappers of several methods exposed by other different services. These methods are:

- /group/get-user-sites

- /dlsync/get-dl-sync-update
- /dlapp/get-folders
- /dlapp/get-file-entries
- /dlapp/add-folder
- /dlapp/update-folder
- /dlapp/add-file-entry
- /dlapp/update-file-entry
- /dlfileentry/get-file-as-stream

Where group, dlsync, dlapp and dlfileentry are the names of different JSON Web Services. In the case of Liferay Safe, which invokes only the Provider's services, these names are substituted with dlprovider.

In the next pages we report an example of JSON response for each of the first four methods listed; we will focus on the remaining ones when discussing the downloader and uploader services.

/get-user-sites

Returns the guest or current user's layout set group, organization groups, inherited organization groups, and site groups.

Listing 2.32: /get-user-sites JSON response

```
[
        {
                active: true,
                classNameId: 10,
                classPK: 19,
                companyId: 1,
                creatorUserId: 5,
                description: "",
                friendlyURL: "/guest",
                groupId: 19,
                liveGroupId: 0,
                name: "Guest",
                parentGroupId: 0,
                site: true,
                type: 1,
                typeSettings: ""
        }
]
```

/get-dl-sync-update?companyId=1&repositoryId=19&lastAccessDate=1355499996163

Returns all sync updates occurred in the repository after the last sync.

Listing 2.33: /get-dl-sync-update JSON response

```
{
        DLSyncs: [
                {
                        companyId: 1,
                        confidential: true,
                        createDate: 1342279216432,
                        event: "add",
                        fileId: 10683,
                        fileUuid: "c1708549-9672-4f53-aa09-932215d03c34",
                        modifiedDate: 1342279216432,
                        name: "research_report.pdf",
                        parentFolderId: 0,
                        repositoryId: 19,
                        syncId: 10684,
                        type: "file",
                        version: "-1"
                }
        ],
        lastAccessDate: 1359763011283
}
```

/get-folders?repositoryId=1&parentFolderId=0

Returns all immediate subfolders of the parent folder.

Listing 2.34: /get-folders JSON response

```
[
        {
                companyId: 1,
                confidential: false,
                createDate: 1342792580838,
                defaultFileEntryTypeId: 0,
                description: "",
                folderId: 12816,
                groupId: 19,
                lastPostDate: 1359252171687,
                modifiedDate: 1342792580838,
                mountPoint: false,
                name: "Folder",
                overrideFileEntryTypes: false,
                parentFolderId: 0,
                repositoryId: 19,
                userId: 2,
                userName: "",
                uuid: "bc22d971-f4b4-4119-bf46-1138e5db8936"
        }
]
```

/get-file-entries?repositoryId=19&folderId=0

Returns all the file entries in the folder.

Listing 2.35: /get-file-entries JSON response

```
[
    {
        companyId: 1,
        confidential: false,
        createDate: 1342279217320,
        custom1ImageId: 0,
        custom2ImageId: 0,
        description: "",
        extension: "png",
        extraSettings: "",
        fileEntryId: 10813,
        fileEntryTypeId: 0,
        folderId: 10683,
        groupId: 19,
        largeImageId: 0,
        mimeType: "image/png",
        modifiedDate: 1342279217320,
        name: "12",
        readCount: 25,
        repositoryId: 19,
        size: 19163,
        smallImageId: 0,
        title: "social_network.png",
        userId: 5,
        userName: "",
        uuid: "8e978614-5d2e-4105-a4a4-32a36e621dfe",
        version: "1.0",
        versionUserId: 5,
```

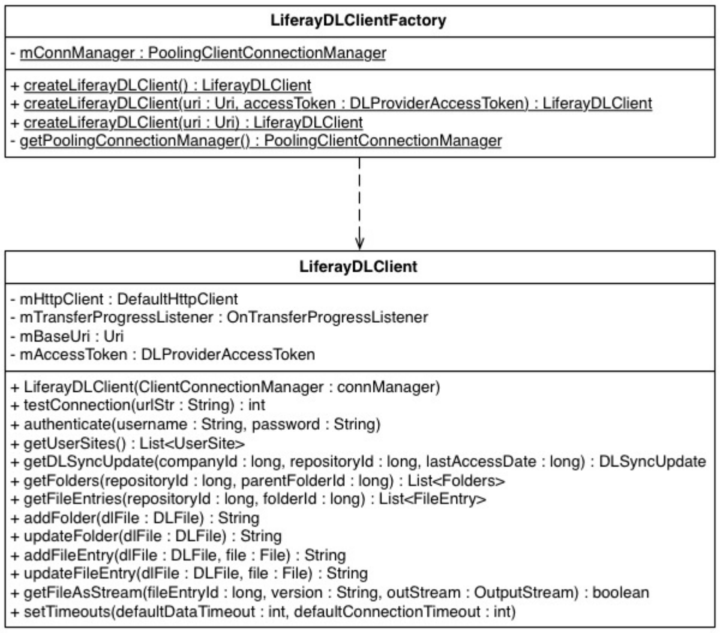

LiferayDLClientFactory
- mConnManager : PoolingClientConnectionManager
+ createLiferayDLClient() : LiferayDLClient
+ createLiferayDLClient(uri : Uri, accessToken : DLProviderAccessToken) : LiferayDLClient
+ createLiferayDLClient(uri : Uri) : LiferayDLClient
- getPoolingConnectionManager() : PoolingClientConnectionManager
LiferayDLClient
- mHttpClient : DefaultHttpClient
- mTransferProgressListener : OnTransferProgressListener
- mBaseUri : Uri
- mAccessToken : DLProviderAccessToken
+ LiferayDLClient(ClientConnectionManager : connManager)
+ testConnection(urlStr : String) : int
+ authenticate(username : String, password : String)
+ getUserSites() : List<UserSite>
+ getDLSyncUpdate(companyId : long, repositoryId : long, lastAccessDate : long) : DLSyncUpdate
+ getFolders(repositoryId : long, parentFolderId : long) : List<Folders>
+ getFileEntries(repositoryId : long, folderId : long) : List<FileEntry>
+ addFolder(dlFile : DLFile) : String
+ updateFolder(dlFile : DLFile) : String
+ addFileEntry(dlFile : DLFile, file : File) : String
+ updateFileEntry(dlFile : DLFile, file : File) : String
+ getFileAsStream(fileEntryId : long, version : String, outStream : OutputStream) : boolean
+ setTimeouts(defaultDataTimeout : int, defaultConnectionTimeout : int)

Figure 2-7: HTTP client

```
                versionUserName: ""
        }
]
```

An additional field in the JSON objects returned by the Provider Portlet is confidential. This field specifies if the file is marked as confidential in the Documents and Media Library. If it is, Liferay Safe will have to take all necessary measures to preserve the privacy of that file. We discuss how this is done in the remainder of this chapter.

2.5.2 HTTP Client

The component responsible for sending requests to the server, converting JSON responses and processing file streams is LiferayDLClient.

LiferayDLClient is based on Apache HttpClient 4.2.1, a library used also by Web browsers which simplifies the handling of HTTP requests.

In the class diagram below we see that LiferayDLClient encapsulates an instance of DefaultHttpClient. This is the default implementation of the HttpClient class, through which data is sent and received as a HttpUriRequest, such as HttpGet and HttpPost. The response of the HttpClient instead is returned as an InputStream.

To handle HTTP connections, the HttpClient uses an implementation of the ClientConnectionManager interface. The purpose of an HTTP connection manager is to serve as a factory for new HTTP connections, manage persistent connections and synchronize access to persistent connections making sure that only one thread of execution can have access to a connection at a time.

In Liferay Safe we use a PoolingClientConnectionManager [39] that manages a pool of client connections and is able to service connection requests from multiple execution threads. Connections are pooled on a per route basis. A request for a route for which the manager already has a persistent connection available in the pool will be serviced by leasing a connection from the pool rather than creating a brand new connection. This allows to share a single connection manager among multiple instances of LiferayDLClient and manage multiple connections more efficiently.

The LiferayDLClientFactory is a factory class providing static methods that create different instances of the LiferayDLClient which use a single shared PoolingClientConnectionManager. There are different factory methods depending on whether the HTTP client is used to test the connection to a server or to invoke an API method either passing an access token or not.

The most important method in the factory (reported below) is getPoolingConnectionManager(), which is used to retrieve the instance of the connection manager necessary to initialize the HttpClient.

Listing 2.36: PoolingConnectionManager

```
private static PoolingClientConnectionManager getPoolingConnectionManager() {
        if (mConnManager == null) {

                                SSLContext sslContext = createSslContext(useClientAuth);
                                CustomSSLSocketFactory sslSocketFactory = new CustomSSLSocketFactory(
                                sslContext, new BrowserCompatHostnameVerifier());

                                ConnPerRoute connPerRoute = new ConnPerRouteBean(10);
                ConnManagerParams.setMaxConnectionsPerRoute(params, connPerRoute);
                ConnManagerParams.setMaxTotalConnections(params, 20);

                SchemeRegistry schemeRegistry = new SchemeRegistry();
                schemeRegistry.register(new Scheme("https", sslSocketFactory, 443));

                                mConnManager = new PoolingClientConnectionManager(schemeRegistry);
        }
    return mConnManager;
}
```

Note from the code that the connection manager is initialized with a single scheme to support only the HTTPS protocol. Additionally the scheme is supplied with the CustomSSLSocketFactory discussed in section 6.3, which is used to validate the identify of the HTTPS server against a list of trusted certificates stored in a TrustManager.

To explain how the LiferayDLClient works, we consider the following code used to invoke the getDLSyncUpdate method of the JSON Web Services.

Listing 2.37: getDLSyncUpdate()

```java
public DLSyncUpdate getDLSyncUpdate(long companyId, long repositoryId,
                        long lastSyncMarker) throws ParseException, IOException,
                        AuthenticationException {

        DLSyncUpdate dlSyncUpdate = null;

        String url = String.format(URL_GET_DL_SYNC_UPDATE, companyId,
                                repositoryId, lastSyncMarker);

        // Prepare a request object
        HttpGet httpget = new HttpGet(mUri.toString() + url);

        // Set the OAuth token in the header
        httpget.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());

        // Execute the request
        HttpResponse response = mHttpClient.execute(httpget);

        // Examine the response status
        int status = response.getStatusLine().getStatusCode();
        boolean result = status == HttpStatus.SC_OK;

        if (result) {
                // Get hold of the response entity
                HttpEntity entity = response.getEntity();

                if (entity != null) {
                        InputStream instream = entity.getContent();

                        try {
                                Reader reader = new InputStreamReader(instream);

                                Gson gson = new Gson();

                                // Parse the JSON data
                                dlSyncUpdate = gson.fromJson(reader, DLSyncUpdate.class);

                        } catch (RuntimeException ex) {
                                httpget.abort();
                                throw ex;

                        } finally {
```

```
                    // Ensure that the entity content is fully consumed and
                    // the content stream, if exists, is closed.
                    EntityUtils.consume(entity);

                    // Release the connection back to the connection manager
                    httpget.releaseConnection();
                }
            }
        } else if (status == HttpStatus.SC_UNAUTHORIZED) {
            throw new AuthenticationException();

        } else {
            throw new IOException();

        }
        return dlSyncUpdate;
}
```

Two important steps when invoking an HTTP method are, first to provide the access token value in the request header, as discussed in section 6.4, and second to make sure the connection at the end is always released.

The flow is analogous for the remaining GET methods; an example of POST execution will be given in the next section after introducing the downloader and uploader services.

2.6 File download and upload

File downloads and uploads are performed by two distinct services of the IntentService class. IntentService is a subclass of Service that uses a worker thread to handle all start requests, one at a time. In particular IntentService does the following:

- Creates a default worker thread that executes all intents delivered to onStartCommand() separate from the application's main thread.
- Creates a work queue that passes one intent at a time to the onHandleIntent() implementation.
- Stops the service after all start requests have been handled, so there's no need to call stopSelf().
- Provides default implementation of onBind() that returns null.
- Provides a default implementation of onStartCommand() that sends the intent to the work queue and then to the onHandleIntent() implementation.

Thus, the only method to implement is onHandleIntent(), which receives the intent for each start request so that work can be done in the background. This is convenient for us because we don't want to handle multiple requests simultaneously.

File downloads are initiated on demand of the user, or whenever a new sync update is received, if the file's keepInSync value is true. The file downloader service must also take care of managing private files differently than public ones, this is done based on the value of the file's confidential property. Similarly, file uploads are started by the user, or whenever a local modification to a file is detected. However, only public files can be uploaded.

In the next paragraphs we look in more detail to the implementation of these services.

2.6.1 FileDownloader service

The FileDownloader service is started with an explicit intent containing an instance of Account, necessary to retrieve the server's base URI, and the DLFile instance of the file being requested for download. Both objects implement the Parcelable interface in order to be bundled with the intent, which is reported below.

Listing 2.38: FileDownloader service intent

```
Intent intent = new Intent(this.getContext(), FileDownloader.class);
intent.putExtra(FileDownloader.EXTRA_ACCOUNT, account);
intent.putExtra(FileDownloader.EXTRA_DLFILE, dlFile);
getContext().startService(intent);
```

If the service is being started for the first time, then the service's onCreate() callback is called. At this the server creates a NotificationManager to display notifications about the download progress on the notification bar, and a LiferayDLClient to communicate with the server. Additionally, the service binds to the CacheGuard service (discussed later in section 6.7.4) responsible for managing the access token and for storing private files. The onCreate() method is reported in the next code listing.

Listing 2.39: FileDownloader service onCreate()

```
@Override
public void onCreate() {
super.onCreate();

        // Init notification manager
        mNotificationMngr = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

        // Bind to the CacheGuard service
        bindService(new Intent(CacheGuardService.class,
                        mConnection, Context.BIND_AUTO_CREATE);

        // Create the HTTP client
        mLiferayDLClient = LiferayDLClientFactory.createLiferayDLClient();
        mLiferayDLClient.setTransferProgressListener(this);
}
```

Next the intent is passed to the onHandleIntent() method, where all the work should be performed. Here the service extracts both the Account and the DLFile bundled in the intent, and sets the LiferayDLClient with the current access token returned from the bound CacheGuard service. Next two different methods are called for performing the download, depending on whether the file is marked as confidential or not.

Listing 2.40: FileDownloader service onHandleIntent()

```
@Override
public void onHandleIntent(Intent i) {
        if (!intent.hasExtra(EXTRA_ACCOUNT) || !intent.hasExtra(EXTRA_DL_FILE)) {
                return;
        }

        mAccount = intent.getParcelableExtra(EXTRA_ACCOUNT);

        // Set the current base URI and access token
        String baseUri = AccountManager.get(context).getUserData(mAccount,
        AccountAuthenticator.KEY_HOST_BASE_URL);
        mLiferayDLClient.setBaseUri(baseUri);
        mLiferayDLClient.setAccessToken(mCacheGuard.getAccessToken());

        DLFile dlFile = intent.getParcelableExtra(EXTRA_DLFILE);

        boolean result = false;

        // Notify download start

        if (dlFile.getConfidential())
                result = processPrivateFileDownload(dlFile);
        else
                result = processPublicFileDownload(dlFile);

        // Broadcast result to registered receivers
        sendFinalBroadcast(result);
}
```

If the file is a private file, then processPrivateFileDownload() is called. The method calls the LiferayDLClient's getFileAsStream() method passing an OutputStream where the downloaded stream should be written to. The OutputStream is managed by the CacheGuard service, which will take care of writing to an encrypted storage, and mark the file as downloaded in the persistence if the operation was completed with success.

The outcome of the download is then notified to the user. If any error occurred, the state of the cache is reverted and the file is marked with a specific error code.

Listing 2.41: Private file download

```
private boolean processPrivateFileDownload(DLFile dlFile) {
        boolean result = false;

        try {
                result = mLiferayDLClient.getFileAsStream(dlFile.getFileId(),mDLFile.getVersion,
                                                        mCacheGuard.initFileCaching(dlFile).getOutputStream());

        } catch (Exception e) {
                // Set notification error message
        }

        if(result) {
                mCacheGuard.finalizeFileCaching(dlFile);
                // Notify download success
        }
        else {
                mCacheGuard.abortFileCaching(dlFile);
                // Notify download error
        }

        return result;
}
```

For downloads of public files the procedure is similar, yet the OutputStream is returned from the CacheManager and the result of the download is persisted through the DLFileManager.

Listing 2.42: Public file download

```
private boolean processPublicFileDownload(DLFile dlFile) {
        boolean result = false;

        try {
                result = mLiferayDLClient.getFileAsStream(dlFile.getFileId(),dlFile.getVersion,
                                        new FileOutputStream(getCacheManager.openLocalFile(dlFile)))

        } catch (Exception e) {
                // Notify download error (I/O or storage space)
        }

        if (result) {
                getDLFileManager().markDownloaded(dLFile);
                // Notify download success
        }
        else {
                getDLFileManager().markError(dlFile, SyncErrorCode.DOWNLOAD_UNKNOWN);
                getCacheManager.deleteLocalFile(dlFile);
                // Notify download success
        }

        return result;
}
```

The getFileAsStream() method of the HTTP client writes the server's response stream exactly in the OutputStream supplied as an argument. During this operation the client calls the transferProgress() callback defined in the OnTransferProgressListener interface, passing the number of bytes written to the registered listener implementing the interface, which in this case is the FileDownloader service. This allows the service to update a progress bar while the file is being download.

Listing 2.43: HTTP client getFileAsStream()

```
public boolean getFileAsStream(long fileEntryId, String version, OutputStream outStream)
                throws IOException, AuthenticationException {
        boolean result = false;

        String url = String.format(URL_GET_FILE_AS_STREAM, fileEntryId, version);

        HttpGet httpget = new HttpGet(mUri.toString() + url);
        httpget.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());

        HttpResponse response = mHttpClient.execute(httpget);

        int status = response.getStatusLine().getStatusCode();

        if (status == HttpStatus.SC_OK) {
                HttpEntity entity = response.getEntity();

                if (entity != null) {
                        InputStream instream = entity.getContent();

                        try {
                                byte[] b = new byte[1024];
                            int len = 0;
                            while ((len = instream.read(b)) != -1) {
                                        if (mTransferProgressListener != null)
                                                mTransferProgressListener.transferProgress(len);
                                outstream.write(b, 0, len);
                            }
                                result = true;

                        } catch (RuntimeException ex) {
                                httpget.abort();
                                throw ex;

                        } finally {
                                EntityUtils.consume(entity);
                                outstream.close();

                                httpget.releaseConnection();
                        }
                }

        } else if (status == HttpStatus.SC_UNAUTHORIZED) {
                throw new AuthenticationException();

        } else {
                throw new IOException();

        }

        return result;
}
```

Finally, besides notifying the user about the result, when a download is completed the service sends a local broadcast to all other components registered with a BroadcastReceiver.

2.6.2 FileUploader service

Uploading a new file to the remote repository involves first selecting such file either directly from the device's local storage or from other installed applications. Additionally, a user might select a single file or multiple files at once to upload.

For simplicity we will deal with the upload of a single file imported from other applications. Instead of implementing our own local file browser we will assume one of the many file explorers available for free from the Google Play store is already installed on the device. Anyhow, our implementation can be entirely reused to support other more sophisticated requirements.

The first step for uploading a file is to display to the user a list of available applications where a file can be selected from. This is done with the intent in the next code snippet:

Listing 2.44: FileUploader service intent

```
Intent action = new Intent(Intent.ACTION_GET_CONTENT).
                .setType("*/*")
                .addCategory(Intent.CATEGORY_OPENABLE);
                startActivityForResult(Intent.createChooser(action,
                                getString(R.string.upload_chooser_title)),
                                ACTION_SELECT_CONTENT_FROM_APPS);
```

This intent will start the activity matching the application selected by the user, expecting as a result the Uri of the file that was selected, returned with an intent to the onActivityResult() callback. If the result is valid a new entry in the Content Provider can be created based on the path of the folder the user decided to upload the file to, and the FileUploader service can be started.

Listing 2.45: File upload onActivityResult()

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {

    if (requestCode == ACTION_SELECT_FILE_FROM_APPS && resultCode == RESULT_OK) {
        Uri fileUri = data.getData();

                String name = new File(fileUri).getName();
                String filePath = mCurrentDir.getFilePath() + FILE_SEPARATOR + name;
                DLFile uploadFile = mDLFileManager().createDLFileFromPath(name, filePath,
                                mCurrentDir.getFilePath(), false);

                Intent intent = new Intent(this, FileUploader.class);
                intent.putExtra(FileDownloader.EXTRA_ACCOUNT, account);
                intent.putExtra(FileDownloader.EXTRA_DLFILE, uploadFile);
                intent.putExtra(FileDownloader.EXTRA_FILE_PATH, fileUri.getPath());
                startService(intent);
    }
}
```

Note that when a new file is uploaded the file itself isn't cached in Liferay Safe. For this reason, when the user want's to access the file from Liferay Safe, a new synced version of the file will have to be downloaded to the cache from the server, possibly creating a double copy in the local storage.

The logic implemented by the FileUploader service is simpler than its downloader counterpart, since we don't support uploading private files. The file upload consists in executing from the HTTP client either the addFileEntry() or updateFileEntry() method depending on whether the file is being added or it was modified.

The server's response, which is returned to the service from the HTTP client, can either be a string representing a server exception or a FileEntry as a JSON object. In the first case the service will handle the exception and set the state of the local file to an error value, accordingly. In the second case the service can proceed to parse the FileEntry from JSON and use it to update the local file entry. Finally, if no errors occurred, the file is marked as uploaded.

Listing 2.46: FileUploader onHandleIntent()

```
@Override
public void onHandleIntent(Intent i) {
        if (!intent.hasExtra(EXTRA_ACCOUNT) || !intent.hasExtra(EXTRA_DL_FILE)) {
                return;
        }

        mAccount = intent.getParcelableExtra(EXTRA_ACCOUNT);

        // Set the current base URI and access token
        String baseUri = AccountManager.get(context).getUserData(mAccount,
                                                                AccountAuthenticator.KEY_HOST_BASE_URL);
        mLiferayDLClient.setBaseUri(baseUri);
        mLiferayDLClient.setAccessToken(mCacheGuard.getAccessToken());

        DLFile dlFile = intent.getParcelableExtra(EXTRA_DLFILE);
        String filePath = intent.getStringExtra(EXTRA_FILE_PATH);

        if (filePath == null)
                filePath = CacheManager.getLocalFilePath(dlFile);

        // Notify upload start

        String response;
        if (dlFile.getFileId() == -1)
                response = mLiferayDLClient.addFileEntry(dlFile, filePath);
        else
                response = mLiferayDLClient.updateFileEntry(dlFile, filePath);

        // Check server errors
        boolean result = checkServerException(response);

        if (!result) {
                // Parse JSON response
                Gson gson = new Gson();
                FileEntry fileEntry = gson.fromJson(response, FileEntry.class);

                if (fileEntry.getFileEntryId() > 0) {
                        DLFileUtil.updateDLFile(dlFile, fileEntry);
                        getDLFileManager().markUploaded(dlFile);
                        result = true;
                } else {
                        getDLFileManager.markError(dlFile, SyncErrorCode.UPLOAD_UNKNOWN);
                        // Notify upload error
                }
        }

        // Broadcast result to registered receivers
        sendFinalBroadcast(result);
}
```

The file is actually uploaded as a FileBody content in a MultipartEntity sent with an HTTP POST method. Since we have to notify the user about the upload progress, we extended the MultipartEntity class in order to override the writeTo() method and post the progress to the FileUploader implementing the OnTransferProgressListener interface. In particular we used a custom FilterOutputStream to get the number of bytes written to the output stream.

Listing 2.47: HTTP client MultipartEntity subclass

```
public class DLMultiPartEntity extends MultipartEntity
{

        private final OnTransferProgressListener listener;

        @Override
        public void writeTo(final OutputStream outstream) throws IOException
        {
                super.writeTo(new ProgressOutputStream(outstream, this.listener));
        }

        public static class ProgressOutputStream extends FilterOutputStream
        {

                private final OnTransferProgressListener listener;
                private long transferred;

                public void write(byte[] b, int off, int len) throws IOException
                {
                        out.write(b, off, len);
                        this.transferred += len;
                        if (this.listener != null)
                                this.listener.transferProgress(this.transferred);
                }
                ...
        }
...
}
```

Finally to complete this section, reported below is the implementation of the addFileEntry() method.

Listing 2.48: HTTP client addFileEntry

```
public String addFileEntry(DLFile dlFile, String filePath) {

        HttpPost httpPost = new HttpGet(mUri.toString() + URL_ADD_FILE_ENTRY);

        httpPost.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());

        DLMultipartEntity multipartEntity = new DLMultipartEntity(mTransferProgressListener);

        StringBody changeLogBody = createStringBody(dlFile.getVersion());
        multipartEntity.addPart("changeLog", changeLogBody);

        String description = dlFile.getDescription();
        StringBody descriptionBody;
        if (description != null) {
                descriptionBody = createStringBody(description);
        }
        else {
                descriptionBody = createStringBody("");
        }
        multipartEntity.addPart("description", descriptionBody);

        StringBody fileEntryIdBody = createStringBody(dlFile.getFileId());
        multipartEntity.addPart("fileEntryId", fileEntryIdBody);

        StringBody majorVersionBody = createStringBody("false");
        multipartEntity.addPart("majorVersion", majorVersionBody);
        
        StringBody titleBody = createStringBody(dlFile.getTitle());
        multipartEntity.addPart("sourceFileName", titleBody);

        String mimeType = MimeTypeUtil.getMimeType(dlFile.getFilePath());
        StringBody mimeTypeBody = createStringBody(mimeType);
        multipartEntity.addPart("mimeType", mimeTypeBody);

        multipartEntity.addPart("title", titleBody);

        try {
                File file = new File(filePath);
                FileBody filePart = new FileBody(file, mimeType);
                multipartEntity.addPart("file", filePart);
        } catch (Exception e) {
                return null;
        }

        httpPost.setEntity(multipartEntity);

        HttpResponse response = mHttpClient.execute(httpPost);

        int status = response.getStatusLine().getStatusCode();

        if (status == HttpStatus.SC_OK)
                return EntityUtils.toString(response.getEntity());

        return null;
}
```

The implementation is analogous for the remaining POST invocations to the server's JSON Web Services API that we mentioned in the previous section.

Chapter 3

Offline usage and preservation of private data

There are two important concepts that make the development of Liferay Safe for Android an interesting case of study, these are offline usage and preservation of private data.

The concept of offline usage (or offline mode) is very common in the context of a client-server architecture. In fact, offline usage is the capability of the client application to deliver its functionalities - or at least, part of them - to the user when a connection to the server cannot be established. Furthermore, an application in offline mode must provide the same security level as if operating in online mode. This directly implies that, under this mode of operation, the application must still be able to identify and authenticate the user, as well as preserving any managed private data.

To better define what we mean for private data and its preservation, we should first recall that Liferay Portal (the server) is "The Leading Open Source Portal for the Enterprise". This tagline is important because it clearly states that our domain of study is exactly the Enterprise domain. This means that the party owning and administering the server is not a group of common consumer users, but a company or organization of any size and complexity.

Liferay Portal allows the development of collaborative websites for teams and an entire enterprise. Liferay users can be grouped into a hierarchy of "organizations" or cross-organizational "user-groups". A Liferay user can also create or join one or more communities and organize all work collaboration within that community.

Liferay Portal also provides a central platform for determining enterprise content policy depending on a user's role, including who can edit and publish Web content, files, communities, and applications.

Therefore, Liferay is also a full workflow-enabled Web Content Management System (WCMS). Including a unified document repository (the "Documents and Media Library") that can be leveraged across an enterprise, within a specific group, or for a single user as a web repository. Liferay users can store documents, video, audio, images, and other media types all in one place, and make them available on the portal websites, or across different client applications through the HTTP APIs, to download them for offline use.

The confidentiality of documents (intended as "data" in general) stored in Liferay Portal can be scaled to various level - unless they are of public domain, of course. A document can be private to the single user, or it can be confidentially disclosed within a specific group or the entire enterprise. However, the concept of privacy for some documents overlaps with the concept of private proprietary information.

> *"Proprietary information is sensitive information that is owned by a company and which gives the company certain competitive advantages. Proprietary information assets are critical to the success of many, perhaps most businesses. [...] Proprietary information, also known as a trade secret, is information a company wishes to keep confidential."* [35]

Whereas private personal information, is currently defined by the European Union as:

> *"any information relating to an identified or identifiable natural person ('data subject'); an identifiable person is one who can be identified, directly or indirectly, in particular by reference to an identification number or to one or more factors specific to his physical, physiological, mental, economic, cultural or social identity;"* Article 2(a) of the Data Protection Directive [36]

Hence, in Liferay we deal with two types of private data. On one hand, data that is associated with, and belongs to a user as a private individual, which was uploaded to her own repository and that may be shared with others for personal or business reason. On the other hand, data that is proprietary information of the enterprise, which can be uploaded to a collaborative repository by a user as an enterprise representative.

For preservation of private data we mean information security, which indeed may be defined as:

> *"Preservation of confidentiality, integrity and availability of information; in addition, other properties such as authenticity, accountability, non-?repudiation and reliability can also be involved."* [42]

The first three primary concepts in information security - also known as CIA triad - are defined in ISO 27001 [42] as follows:

- Confidentiality - *"the property that information is not made available or disclosed to unauthorized individuals, entities or processes"*;
- Integrity - *"the property of safeguarding the accuracy and completeness of assets"*;
- Availability - *"the property of being accessible and usable upon demand by an authorized entity"*.

Private data are those information assets for which all three attributes are important; security of private data focuses on provision of protection mechanisms that preserve those three attributes.

In the development of Liferay Safe - a Liferay Portal document synchronizer for Android - we must face different problems regarding the preservation of CIA over private data. Some of these issues are inherent in a client-server architecture, other are more particularly related to mobility and to the Android platform. Moreover, the requirement of offline access capability poses an inherent challenge to privacy preservation, because it typically results in losing control other data.

When considering a mobile client application for Android, relevant issues related to privacy preservation arise, including and not limited to: data usage and proliferation, data theft, inadequate data retention, inadequate data deletion, communication spoofing (security of data in transit), client/server impersonation, dynamic provisioning of trust (user revocation), unauthorized access, lack of a root of trust, device tampering, Android malware, privilege escalation, and notification of breach.

Some of these issues can be eliminated, other represent risks that can only be mitigated. In the following sections we propose a trade-off approach towards the challenging task of simultaneously providing preservation and offline access to private data in a mobile Android application.

3.1 A trade-off approach

Considering the issues related to privacy preservation that we mentioned before, the goal of our approach is to build a secure implementation of Liferay Safe for Android, that is, one capable of preserving the privacy of private user and corporate data at multiple levels, and against multiple risk factors.

In synthesis, our approach consists in applying security coding best practices to harness features specific to Android - such as the component framework, the security model and lower kernel capabilities - and to integrate state of the art technologies compatible with the Android platform. In particular, the approach aims to give the following guarantees:

- Security of data in transit: client-server communication is secured with HTTPs. Man In The Middle attacks (MITM) and impersonation are mitigated with the use of SSL certificates. See section 6.3;

- Prevention from unauthorized data access: the OAuth 2.0 protocol provides an authorization mechanism that allows to control, restrict and revoke user access to protected data, while preserving their account credentials. The automatic screen lock functionality protects retained data from unwanted eyes when the device is left unattended. See section 6.4.

- Protection of data at rest: private data cached in the local storage must be protected in the event of its unauthorized access or theft. Encrypted cache storage and secure encryption key management are the core solutions to prevent data visibility and mitigate privacy leakage. See subsections 6.7.5 and 6.7.6.

- Control over the use of data: as any other asset, private data has no value to the user if it cannot be used at least for the purpose of viewing. Unfortunately, it's not possible to entrust either a third party, or the system itself, to provide such usage functionalities, unless without taking into account all the security risks involved. Though, security of private data can be preserved if all functionalities required by the user are implemented by a trusted party, and if there is a single entity holding the control of private data during all its use in memory. See subsection 6.7.7.

- Reduction of tampering exposures: Android-compatible devices have proven to be particularly open to user customization. At the same time, Android's architecture stack is considered an interesting attack surface where tools of exploitation and reverse engineering common to the Linux environment can be put in practice to easily gain root privileges and violate user's private data. There is no final protection against tampering of the device, this risk can only be assumed or mitigated. See subsection 6.9 for examples.

We call our approach a trade-off because at the state of the art on Android devices it's not possible to provide a fully functional solution able to protect private data and enable offline usage at the same time; at least without posing any constraints and limitations to the application, or taking initial assumptions.

Furthermore, a key aspect of this approach is that the resulting solution does not absolutely imply forking the original Android Open Source Project (AOSP), or building a custom firmware, to leverage the above security provisions.

In fact, an important requirement is that enterprises interested in this solution should not be constrained to provide their employees a specific device tailored for this application, but instead employees should be allowed to use their own Android devices in their jobs and install Liferay Safe to access corporate private data.

However, this implies that there is little or no control over how devices running Liferay Safe are used, what other applications are installed, and if the devices are rooted or have a customized firmware. Besides, in this scenario end-user policies defined by enterprises turn out to be both highly ineffective and very difficult to be enforced.

Taking all this into account, there is no surprise in the fact that the result of our approach should not be deemed perfect, and consequently we shall now state the initial assumptions, constraints and limitations of our design:

- Minimum Operating System: lacking a root of trust, we heavily rely on the underlying kernel security to store in memory expiring encryption keys and access tokens. Starting from Android 4.1 (Jelly Bean) there is full support for 32-bit Address Space Layout Randomization (ASLR) and Position-independent executables (PIE) [43], which hinder some types of security attacks (such as Return Oriented Programming) by preventing an attacker from being able to easily predict target addresses [44].

- 24-hours limited offline access: to enforce revocation of a user, we limit the offline access to 24-hours, after which the application is locked to the authentication screen and the private cache is erased. The public cache can still be accessed by mounting the external storage.

- Maximum size of private data: because private data is never written to disk unencrypted, private files can only be accessed in such state from the volatile memory. Therefore, the maximum size of a single private file cannot be larger than the available RAM on the device.

- No editing of private data: to avoid losing control over usage of private data, and prevent its proliferation, a trusted proprietary solution for editing files is needed. However, this would imply to develop and maintain a full "office suite" solution, which is out of the scope of this project. Moreover, there are no open source projects available of that kind.

- PDF format limitation: for the same reason above, at the moment Liferay Safe supports only private files in PDF format. The functionality is delegated to a separate viewer application based on an open source project. Fortunately, Liferay Portal supports automated conversion of several file formats to PDF.

- Vulnerability to pre-rooted devices: there is no 100% effective way to detect rooting [45]. If a device was already rooted prior to the installation of Liferay Safe, any private data subsequently downloaded could be compromised based on the attacker's skills. Enterprise end-user policies should include responsibilities related to device tampering.

- Enterprise policy enforcement: the enterprise must promptly revoke user access when a user isn't trusted anymore, and in case of policy breach.

- Mandatory screen lock and disabled "USB debugging" setting: the Android Debug Bridge (ADB) is the simplest vector for rooting a device, but it requires the "USB debugging" setting to be enabled [11]. The application enforces the use of a screen lock to protect this settings and requires that it is disabled for the private cache to be available. Whenever the ADB is enabled, the private cache is deleted.

- Vulnerability to remote exploitation: an attacker can still gain root access through malicious applications containing privilege escalation attacks, or by taking advantage of vulnerabilities in system components, such as the browser [11]. However, we assume these techniques would require a reasonable amount of time and a highly experienced attacker to be performed.

- Vulnerability to memory dump: after gaining root access, an attacker can dump the application memory and analyze it to find a valid private cache encryption key or a server access token [46, 47]. We assume that memory analysis is a difficult and time consuming task. The risk is mitigated by limiting the validity of sensitive data in memory and reducing its duplication.

- Device whitelisting: some devices expose specific vulnerabilities related to hardware or firmware bugs (such as the latest Samsung Exynos exploit [48]), and additionally, vendors adopt different cycles for releasing security patches and system updates. Maintaining a whitelist of devices can reduce the risk of distributing private data to reportedly compromised devices.
- Impact on battery life: several operations must be executed periodically to ensure the security of private data, resulting in a more rapid battery discharge.

3.2 Marking files as "confidential"

The procedure for marking a file as confidential is directly integrated in Liferay's "Documents and Media Library" (DML) without any necessary modification to the source code. To mark a file confidential, a user has to simply add the "confidential" tag from the document's properties screen, as shown below.

New and existent documents can be tagged and untagged exclusively from Liferay Portal, and not from the mobile client. The tag information applied to each file entry is evaluated by the Provider Portlet in order to set the boolean value of the confidential field that must be included in the JSON objects returned as a response to the client. Since the original API methods don't include this information, the Provider must retrieve it separately from other services.

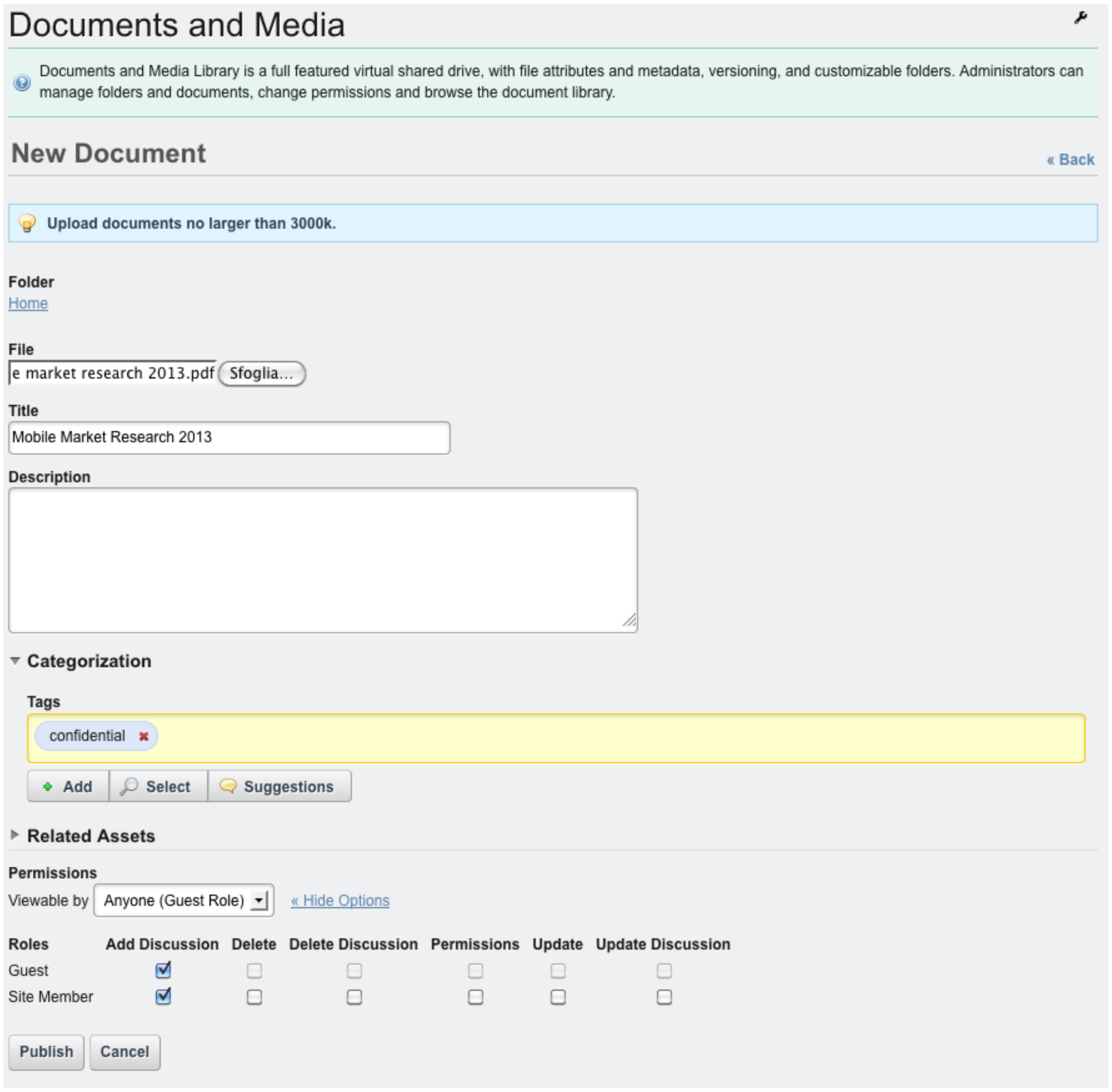
Fortunately, the AssetTagService allows us to get the list of all tags associated to any asset stored in the portal, including documents of the DML. This local service is based on Liferay's Asset Framework, which was created exactly to allow portal applications and custom portlets to be able to add tags to any kind of asset in the portal without having to reimplement this same functionality over and over. The term asset defines any type of content regardless of whether it's purely text, an external file, a URL, an image, a document, etc.

If for example we call this method via JSON HTTP, by specifying the asset's class and id, the output would be the following JSON array:

Listing 3.1: get-tags JSON response

```
/get-tags?className=com.liferay.portlet.documentlibrary.model.DLFileEntry&classPK=16722

[
        {
                assetCount:1,
                companyId:1,
                createDate:1359251455864,
                groupId:19,
                modifiedDate:1359251455864,
                name:"confidential",
                tagId:17218,
```



Documents and Media

Documents and Media Library is a full featured virtual shared drive, with file attributes and metadata, versioning, and customizable folders. Administrators can manage folders and documents, change permissions and browse the document library.

New Document

« Back

Upload documents no larger than 3000k.

Folder
Home

File
e market research 2013.pdf Sfoglia...

Title
Mobile Market Research 2013

Description

Categorization

Tags
confidential

Add Select Suggestions

Related Assets

Permissions
Viewable by Anyone (Guest Role) « Hide Options

Roles	Add Discussion	Delete	Delete Discussion	Permissions	Update	Update Discussion
Guest	✓	☐	☐	☐	☐	☐
Site Member	✓	☐	☐	☐	☐	☐

Publish Cancel

Figure 3-1: Marking a file as confidential

```
                userId:12604,
                userName:"Max"
        }
]
```

Hence, the Provider can call this service for any file entry, check if it's marked as "confidential" and finally set the corresponding field in the JSON object returned to the client.

3.3 Cache architecture

The local cache is probably the most important feature in Liferay Safe. The reasons for this are multiple and most of them have already been discussed in the previous sections, however, let's summarize them here all together:

- Reduce network traffic and file access time;
- Provide offline access to files and improve data availability;
- Enable opening files in other applications for viewing or editing.

Basically, the responsibility of the cache is to provide offline access to both public and private files. This distinction has important implications on the architecture of the cache, because all private files must always be encrypted before being stored to disk. Additionally it's required that, *the unencrypted version of a file must never be written disk* under any circumstance or operation; it can only be *retained in volatile memory and for a short amount of time*, after which the memory must be correctly zeroized.

In order to implement these requirements in a correct and sound way, we opted to have two distinct caches: a secure one for private files and a *world-readable* one for the rest. Although these two are stored and managed differently, they share the following functional characteristics:

- Both caches are stored in Android's external storage;
- A file stored in a cache has the same relative path of the corresponding remote file, with respect to the parent site repository;
- The user is be able to configure the size of the overall storage space allocated to the two caches (public + private). This value is persisted in Android's SharedPreferences;
- All file downloads are automatically cached;
- When the limit is reached, the cache will try to free up space by deleting the oldest synced/modified files, starting from the public cache first;

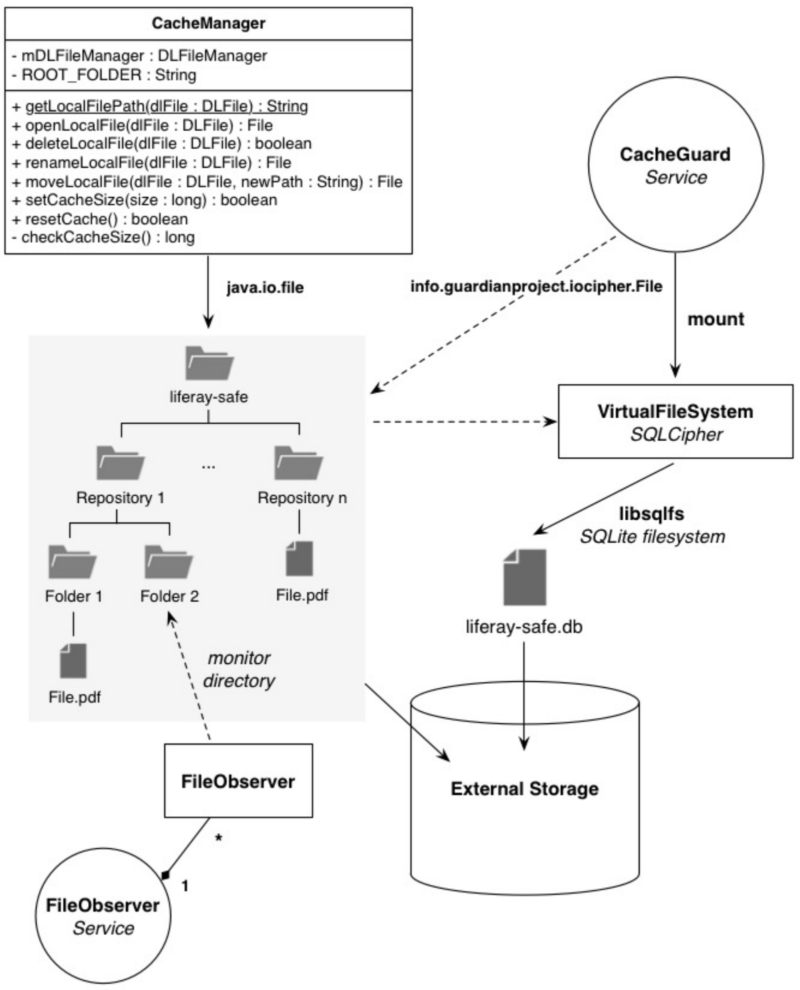

CacheManager
- mDLFileManager : DLFileManager
- ROOT_FOLDER : String
+ getLocalFilePath(dlFile : DLFile) : String
+ openLocalFile(dlFile : DLFile) : File
+ deleteLocalFile(dlFile : DLFile) : boolean
+ renameLocalFile(dlFile : DLFile) : File
+ moveLocalFile(dlFile : DLFile, newPath : String) : File
+ setCacheSize(size : long) : boolean
+ resetCache() : boolean
- checkCacheSize() : long
CacheGuard
Service
java.io.file
info.guardianproject.iocipher.File
mount
liferay-safe
VirtualFileSystem
SQLCipher
Repository 1
...
Repository n
libsqlfs
SQLite filesystem
Folder 1
Folder 2
File.pdf
liferay-safe.db
monitor
directory
File.pdf
FileObserver
External Storage
*
1
FileObserver
Service

Figure 3-2: Cache architecture

- The user can remove a file from the local cache individually.
- The architecture of the local cache is illustrated in Figure 3-2 below. In the next paragraphs we will explain the functioning of the public cache, while the private encrypted cache will be covered more thoroughly in the next two sections.

The diagram provides us a big picture of the components involved in the implementation of the two caches. Focusing for now on the public cache, the CacheManager is the only component entitled to perform operations that change the state of the public cache, such as creating, deleting, renaming and moving a cached file. Including, setting and maintaining the available storage space.

When starting a new download task, the DownloaderService calls the CacheManager's openLocalFile() method, passing the entry of the file going to be downloaded. This causes the CacheManager first to check if the cached file already exists, and then to verify if it's necessary to free up space to complete the download, according to the entry's remoteSize property. Eventually, a list of all synced cached files, ordered by lastSync, can be retrieved from DLFileManager by calling getDLFilesByState(DLFileState) for DOWNLOADED and UPLOADED file states. The combined results can be used to calculate the current occupation size and to select eventual candidates for removal. Finally, a local file is opened (and eventually created) from the entry's filePath property, and returned to the DownloaderService.

Public files are cached to the external storage. This can be thought as a file system supported by every Android-compatible device, that can hold a relatively large amount of data and that is shared across all applications. Traditionally this is an SD card, but it may also be implemented as built-in storage in a device that is distinct from the protected internal storage and can be mounted as a file system on a computer.

Accessing files on external storage is as easy as getting the absolute path to the external storage directory. The following snippet shows how to open a file from the public cache given a filePath:

Listing 3.2: Accessing the external storage

```
File sdCardRoot = Environment.getExternalStorageDirectory();
File file = new File(sdCardRoot.getAbsolutePath() + "/liferay-safe/" + filePath);
```

The absolute path of any file stored in the public cache can be conveniently retrieved as a String calling CacheManager's getLocalFilePath() static method, by any other component. File operations require no particular coding, since Android uses the standard java.io.File class as an "abstract" representation of a file system entity identified by a pathname. Reading and writing to external storage requires the WRITE_EXTERNAL_STORAGE permission to be declared in the AndroidManifest file.

There are three main issues to take care of when using Android external storage to store files [37]:

- Before doing any work with the external storage, Environment.getExternalStorageState() should always be called before to check whether the media is available. The media might be mounted to a computer, missing, read-only, or in some other state;
- There's no security enforced upon files saved to the external storage. All applications can read and write files placed on the external storage and the user can remove them;
- These files will not be deleted when the application is uninstalled.

Notwithstanding, the use of external storage allows the user to more easily access public files from other applications and to transfer these files quickly on another device or computer.

A very important aspect about the public cache is the detection of modifications to the cached files. Liferay Safe should be able to detect when a cached file has been changed and subsequently upload the new version to the server. Modifications include also renaming and moving a file.

Fortunately Android provides the FileObserver as a convenient way to detect these events. A FileObserver monitors files to fire an event after files are accessed or changed by any process on the device. Each FileObserver instance monitors a single file or directory. If a directory is monitored, events will be triggered for all files and subdirectories inside the monitored one.

The monitoring mechanism is based on inotify, a Linux kernel feature that extends file systems to notice changes and report them to applications (an interesting introduction can be found here: [38]). The main drawback about inotify is that it does not support recursively watching directories, meaning that a separate inotify watch must be created for every subdirectory.

Additionally, to ensure keeping a FileObserver sending events and avoiding the risk of garbage collection, it's necessary to hold a reference to the instance from some other live object.

For the above reasons, the public cache architecture includes a FileObserverService which manages a list of FileObserver's. Each FileObserver is associated to a single file in the cache, but only the files having the keepInSync property set to true in their corresponding entry stored in the ContentProvider are monitored by a FileObserver.

A user can either decide to download a file content for a single visualization or explicitly request the synchronizer to keep the file in sync (both-ways with the server) after the first download. In this case, modifications on the file will be monitored by a FileObserver which will start the UploaderService for any event of change detected.

The FileObserverService also manages a list of broadcast receivers to receive notifications from the DownloaderService when the download of a file has been completed. This is necessary for a file that needs to be synced before a FileObserver starts to monitor its changes.

Below are reported the class definitions for both the FileObserverService and the implementation of the abstract FileObserver.

Listing 3.3: FileObserver service

```java
public class FileObserverService extends Service {

    public final static String EXTRA_CMD = "EXTRA_CMD";
    public final static String EXTRA_PATH = "EXTRA_FILE_PATH";

    public final static int INIT_OBSERVED_LIST = 1;
    public final static int ADD_OBSERVED_FILE = 2;
    public final static int RM_OBSERVED_FILE = 3;

    private static List<CacheFileObserver> mObservers;
    private static List<DownloadCompletedReceiver> mDownloadReceivers;
    private static Object mReceiverListLock = new Object();

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if (intent == null) {
            initializeObservedList();
            return Service.START_STICKY;
        }

        if (!intent.hasExtra(EXTRA_CMD)) {
            return Service.START_STICKY;
        }

        switch (intent.getIntExtra(EXTRA_CMD, -1)) {
            case INIT_OBSERVED_LIST:
                initializeObservedList();
                break;
            case ADD_OBSERVED_FILE:
                addObservedFile(intent.getStringExtra(EXTRA_FILE_PATH));
                break;
            case DEL_OBSERVED_FILE:
                removeObservedFile(intent.getStringExtra(EXTRA_FILE_PATH));
                break;
        }

        return Service.START_STICKY;
    }

    private void initializeObservedList() {

        /*
         * 1. Instantiate mObservers and mDownloadReceivers.
         * 2. Retrieve file entries having keepInSync = true.
         * 3. For each entry, get the absolute path and check if file exists.
         * 4. If yes, create a new CacheFileObserver and set the target file.
         * 5. Call startWatching() to start the observer.
         */
    }

    private void addObservedFile(String path) {
     /*
            *  1. Check if an observer for this path already exists.
            *  2. If not, retrieve the file entry for this path and create
            *  a CacheFileObserver.
            *  3. And the observer to the list of observers.
            *  4. Register a new DownloadCompleteReceiver for this file.
            */
    }

    private void removeObservedFile(String path) {
            /* 1. Find the observer of this path in the list.
                 * 2. Call stopWatching() to stop the monitoring.
         * 3. Remove the observer from the list.
            */
    }

    private static void addReceiverToList(DownloadCompletedReceiver r) {
        synchronized(mReceiverListLock) {
            mDownloadReceivers.add(r);
        }
    }

    private static void removeReceiverFromList(DownloadCompletedReceiver r) {
        synchronized(mReceiverListLock) {
            mDownloadReceivers.remove(r);
        }
    }

    private class DownloadCompletedReceiver extends BroadcastReceiver {
        String mPath;
        CacheFileObserver mObserver;

        public DownloadCompletedReceiver(String path, CacheFileObserver observer) {
            mPath = path;
            mObserver = observer;
            addReceiverToList(this);
        }

        @Override
        public void onReceive(Context context, Intent intent) {
            if (mPath.equals(intent.getStringExtra(FileDownloader.EXTRA_FILE_PATH))) {
                context.unregisterReceiver(this);
                removeReceiverFromList(this);
                mObserver.startWatching();
            }
        }

                @Override
        public boolean equals(Object o) {
            if (o instanceof DownloadCompletedReceiver)
                return mPath.equals(((DownloadCompletedReceiver)o).mPath);
            return super.equals(o);
        }
    }
}
```

Listing 3.4: FileObserver subclass

```
public class CacheFileObserver extends FileObserver {

    public static int CHANGES = CLOSE_WRITE | MOVED_FROM | MODIFY;

    private String mFilePath;
    private int mMask;
    Account mAccount;
    DLFile mDLFile;
    static Context mContext;

    public CacheFileObserver(String path) {
        this(path, ALL_EVENTS);
    }

    public CacheFileObserver(String path, int mask) {
        super(path, mask);
        mPath = path;
        mMask = mask;
    }

    public void setAccount(Account account) {
        mAccount = account;
    }

    public void setDLFile(DLFile dlFile) {
        mDLFile = dlFile;
    }

    public void setContext(Context context) {
        mContext = context;
    }

    public String getPath() {
        return mPath;
    }

    @Override
    public void onEvent(int event, String path) {
        if ((event | mMask) == 0) {
            return;
        }
        Intent i = new Intent(mContext, FileUploader.class);
        i.putExtra(FileUploader.KEY_ACCOUNT, mAccount);
        i.putExtra(FileUploader.KEY_DLFILE, mDLFile);
        i.putExtra(FileUploader.KEY_FILE_PATH, mFilePath);
        mContext.startService(i);
    }
}
```

Note also that the CacheFileObserver's event mask is specifically set as CLOSE_WRITE — MOVED_FROM — MODIFY to report exactly those event type changes.

In order to correctly watch a file, every FileObserver must be started immediately after the device has booted and should continue to detect any changes as long as the system is running.

To cope with that, a BootupBroadcastReceiver is used to receive the Intent.ACTION_BOOT_COMPLETED broadcast from the system and subsequently start the FileObserverService:

Listing 3.5: BootupBroadcastReceiver

```
public class BootupBroadcastReceiver extends BroadcastReceiver {

    private static String TAG = "BootupBroadcastReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        if (!intent.getAction().equals(Intent.ACTION_BOOT_COMPLETED)) {
            return;
        }
        Intent i = new Intent(context, FileObserverService.class);
        i.putExtra(FileObserverService.EXTRA_CMD,
                    FileObserverService.INIT_OBSERVED_LIST);
        context.startService(i);
    }
}
```

Where the BootupBroadcastReceiver is registered in the AndroidManifest file, along with the required permission declaration:

Listing 3.6: Registering the bootup broadcast receiver

```
$\langle$uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/$\rangle$
...
$\langle$receiver android:name=".receivers.BootupBroadcastReceiver"$\rangle$
$\langle$intent-filter$\rangle$
        $\langle$action android:name="android.intent.action.BOOT_COMPLETED"/$\rangle$
       $\langle$/intent-filter$\rangle$
$\langle$/receiver$\rangle$
$\langle$service android:name=".cache.FileObserverService"/$\rangle$
...
```

Furthermore, we make the service return Service.START_STICKY from onStartCommand() to tell the system to keep the service running in the background. However, the system can still decide to kill the service in certain low memory conditions, and therefore all FileObserver's must be reinitialized when the it is restarted.

Finally, when a user sets a file to be kept in sync, the FileObserverService is started with the following intent, passing the absolute path to the cached file.

Listing 3.7: FileObserver service start intent

```
Intent intent = new Intent(getActivity().getApplicationContext(),
FileObserverService.class);
intent.putExtra(FileObserverService.EXTRA_CMD,
(dlFile.getKeepInSyn() ?
                FileObserverService.ADD_OBSERVED_FILE:
                FileObserverService.RM_OBSERVED_FILE));
intent.putExtra(FileObserverService.EXTRA_FILE_PATH, CacheManager.getLocalFilePath(dlFile));
startService(intent);
```

3.4 Encrypted file cache

Liferay Safe's encrypted cache is based on the IOCipher library developed by The Guardian Project group [49], and released under the LGPL 2.1 license. IOCipher brings app-level virtual encrypted disks to Android applications without requiring the device to be rooted. It uses a clone of the standard java.io API for working with files in a secure and transparent way.

To use IOCipher encrypted storage, there are only two things the developer needs to do: manage the password and mount the volume using VirtualFileSystem.mount(); and replace the relevant java.io import statements with info.guardianproject.iocipher. All in all using IOCipher is in fact very simple, as the following example demonstrates:

Listing 3.8: Virtual Encrypted Disk operations

```
// Create the virtual encrypted disk in the app's external storage
java.io.File db = new java.io.File(mContext.getExternalFilesDir(null),
 "liferay-safe.db");

VirtualFileSystem vfs = new VirtualFileSystem(db.getAbsolutePath());

```

```
// Mount the virtual file system
vfs.mount();

// Get the root of the vfs
info.guardianproject.iocipher.File ROOT = new File("/");

// Create a new file
info.guardianproject.iocipher.File f = new File(randomFileName("testFile"));

// Write to the new file
info.guardianproject.iocipher.FileOutputStream out = new FileOutputStream(f);
out.write(123);
out.close();

// Unmount the vfs
vfs.unmount();
```

IOCipher is built on top two important libraries: SQLCipher and libsqlfs.

3.4.1 SQLCipher

SQLCipher is an SQLite extension that provides transparent, secure 256-bit AES encryption of database files. It is an open source project, sponsored and maintained by Zetetic LLC, which has been adopted by many organizations, making it one of the most popular encrypted database platforms for mobile, embedded and desktop applications.

As claimed by The Guardian Project group [50] - who maintains the Android porting of SQLCipher - the data stored in this type of encrypted databases will be less vulnerable to access by malicious apps, protected in case of device loss or theft, and highly resistant to mobile data forensics tools.

SQLCipher's encryption is transparent and on-the-fly. Applications use the standard SQLite API to manipulate tables using SQL, while behind the scenes the library silently manages the security. SQLCipher does not implement its own encryption. Instead it uses OpenSSL libcrypto for all cryptographic functions.

The main security features of SQLCipher are [51]:

- The default algorithm is 256-bit AES in CBC mode.
- Each database page is encrypted and decrypted individually.
- Each page has it's own random initialization vector, generated by OpenSSL's RAND_bytes.
- IVs are regenerated on write so that the same IV is not reused on subsequent writes of the same page data.
- Every page write includes a Message Authentication Code (HMAC_SHA1) of the ciphertext and the initialization vector at the end of the page.

- The MAC is checked when the page is read back from disk.
- When initialized with a passphrase SQLCipher derives the key data using PBKDF2 (OpenSSL's PKCS5_PBKDF2_HMAC_SHA1), with a default number of 4000 iterations.
- Each database is initialized with a unique random salt in the first 16 bytes of the file, used for key derivation.
- The key used to calculate page HMACs is different that the encryption key. It is derived from the encryption key and using PBKDF2 with 2 iterations and a variation of the random database salt.
- When encrypted, the entire database file appears to contain random data.

3.4.2 Libsqlfs

Created as part of PalmSource's ALP mobile platform, the libsqlfs library implements a POSIX style file system on top of an SQLite database. It allows applications to have access to a full read/write file system in a single file, complete with its own file hierarchy and name space, without using SQL statements [52].

Libsqlfs can accommodate small preference values such as a number, and large binary objects such as an video clip. The library provides a generic file system layer that maps a file system onto a SQLite database, and supports a POSIX file system semantics.

IOCipher wraps the SQLite encryption features of SQLCipher on top of the SQLite-based file system implemented in the libsqlfs library, providing a secure transparent app-level virtual encrypted disk, with few limitations:

- files cannot be larger than the available RAM on the device;
- no users, groups, or permissions implemented;
- folder renaming is not supported (by libsqlfs);
- crashes possible under extremely heavy, concurrent load.

Currently The Guardian Project is working side to side with Zetetic to improve the performance and stability of IOCipher. They have recently released some optimizations on the libsqlfs library giving it a huge improvement in read/write concurrency and performance [53].

3.4.3 CacheGuard service

Similarly to the CacheManager seen for the public cache, the component responsible for managing the virtual encrypted disk is the CacheGuard service. Just like the CacheManager, it takes care of opening, storing, deleting, renaming and moving files. But more importantly, it is responsible for preserving the privacy stored in the virtual file system (VFS).

Inside the VFS, private files are stored according to the remote directory structure, with respect to their repository membership. The VFS is abstracted by the libsqlfs library from a single SQLite database file encrypted with SQLCipher. This virtual encrypted disk is stored in the /Android/data/⟨package_name⟩/files path in the external filesystem - retrieved from the Context's getExternalFilesDir() method. The directories returned here are owned by the application, and their contents will be removed when the application is uninstalled.

There is no risk in storing the VFS in the external storage, since the underlying SQLite database once initialized with a key is always in an encrypted state.

Alternatively, Android's internal storage could have been used, where files are private to the application, and other applications (and the user) cannot access them. However, this storage space is very limited in most devices and cannot be used to store large files. Infact, when it comes short, other applications could stop functioning. Therefore, it's better to store the cache in the external storage to benefit of a larger and more versatile space.

The CacheGuard service is the only component communicating with the VFS. Other application components must bind to this service in order to store (when downloading) or retrieve (for viewing) any private file. During these operations a private file is never written to disk. The reasons thereof are twofold.

First of all the NAND Flash technology used in mobile devices and SD cards tends to hold onto data longer than traditional spinning hard drives [54]. This is due to the wear leveling technique used for enhancing the drive's longevity, which works by arranging data so that erasures and re-writes are distributed evenly across the memory. Therefore, to completely erase any written data from a NAND Flash memory, it's necessary to rewrite the full medium, which of course is not possible in real conditions.

Secondly, the external storage is often implemented as a removable storage media (eg. an SD card). If private files are written to this storage, even temporarily, when performing an operation, it is possible that the file is not removed at the end of the operation if a particular event occurred such as, for example, the storage medium was removed, the application or the system stopped responding, or the device was powered off.

Instead, a private file is written to a MemoryFile, both when being downloaded from the server, and when loaded from the VFS. MemoryFile is a wrapper for the Linux ashmem driver. Android Shared Memory (ashmem) is a component of the Android operating system that facilitates memory sharing and conservation [56].

A MemoryFile can be shared with other application components or applications that bind to the CacheGuard service, as described later in section 6.7.6. Ashmem has reference counting so that if many processes use the same area the area will not be removed until all processes has released it, the memory is also virtual and not physically contiguous.

The CacheGuard service is responsible for ensuring that private data is visible for the least amount of time necessary, in particular it must take care of:

- Mounting and unmounting the VFS;

- Zeroing out and closing a MemoryFile;
- Managing crashes of the VFS and errors gracefully;
- Controlling the cache usage and erase least recently used files.

Beyond that, the CacheGuard service is responsible also for managing the key used to encrypt the VFS and the OAuth access token necessary to interact with the server. In the next section we discuss exactly how such service component must be structured to perform these security-critical tasks.

3.5 CacheGuard encryption key management

3.6 Viewing and sharing files

Liferay Safe allows the user to perform different operations on cached files depending on whether a file is marked as "confidential". In particular:

- Public files - can be shared with any other application (eg. an email client), both from within Liferay Safe with an intent, or directly from the desired application by browsing the external storage. Any application can access a public file either for reading or writing. The user can also take screenshots and copy parts of text. The content of the public cache can also be accessed from another computer that mounts the external storage.
- Private files - cannot be shared with other third party or system applications and cannot be accessed for writing. Only trusted applications can read from private files. Additionally, both the screen capture and the clipboard are disabled, and the content of the private cache cannot be mounted from other computers.

To open a public file in another application from within Liferay Safe it's sufficient to create an intent with specified the Intent.ACTION_VIEW action, the file's path Uri and MIME type, like in the following example:

Listing 3.9: Opening a public file in other applications

```
String filePath = CacheManager.getLocalFilePath(dlFile);
String fileType = null;
String extension = MimeTypeMap.getFileExtensionFromUrl(filePath);
if (extension != null) {
        MimeTypeMap mime = MimeTypeMap.getSingleton();
        fileType = mime.getMimeTypeFromExtension(extension);
}


```

```
Intent i = new Intent(Intent.ACTION_VIEW);

i.setDataAndType(Uri.parse("file://" + filePath), fileType);
i.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION |
Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
startActivity(i);
```

The standard activity picker will be displayed by the system listing all installed applications whose intent filters match the intent's properties, allowing the user to pick the desired application before proceeding. The selected activity will be started and pushed on top of the activities backstack, so that when the user presses the back button the Liferay Safe screen that had left is restored.

Alternatively the user can access the entire public cache with any file browser application available free on the Google Play store.

Dealing with private files is a lot more different and difficult. We assumed initially that Liferay Safe should provide a secure way to allow the user to view these files at least in PDF format. Still, building a PDF viewer for Android is not straightforward, since there are no system components providing this functionality.

Fortunately, there are a number of open source applications available that could be modified and embedded in Liferay Safe, such as DroidReader [57] and EBookDroid [58]. Many of these are based on MuPDF library [59], which provides accurate and fast PDF rendering.

In spite this, being the rendering library licensed under GNU GPL v3, any derived work must be licensed, as a whole, under the same license (subsection 5c).

Since this could not be compatible with Liferay Safe's licensing, instead the open source code could be modified in a separate application, called Liferay Safe Viewer, and be released independently under the GPL v3 license.

The open source code of the viewer will have to be modified in order to communicate with Liferay Safe to read files from the private cache, and to ensure that any functionality that could compromise the privacy of the shared data is completely blocked or removed - such as opening the file in other applications, capturing screenshots or copying text.

Additionally the viewer application must guarantee that any opened file (or parts of it) is never written to disk, and that any allocated volatile memory is correctly zeroized when the file is closed - leaving no traces of the unencrypted file around.

To make this possible, we rely on Android's Binder Inter-process Communication. An intent from Liferay Safe will start the viewer's activity passing as an extra the id of the private file to be opened. The viewer will then bind to the CacheGuard service using an AIDL interface for remote communication (since the two applications run on different processes), and request the service to open the file. CacheGuard will instantiate a file stream in the shared memory as a MemoryFile and return it directly to the viewer, which can then execute the rendering.

No data is written to disk during all these steps. The interesting thing about using the shared memory (ashmem) is that the allocated space is never counted in the VM budget, therefore as much space can be allocated as the amount available in memory without incurring in OutOfMemory errors.

To allow the viewer to bind CacheGuard without needing to declare an IntentFilter for the service in the AndroidManifest file and making it exported, it is necessary to run both applications under the same Linux User ID. For this purpose we can use the sharedUserId attribute in the AndroidManifest.xml's manifest tag of each package to have them assigned the same user ID.

Only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID. The two packages are then treated as being the same application, with the file permissions and user ID.

The certificate with which an application is digitally signed, and whose private key is held by the application's developer, is used by the Android system as a means of identifying the author of an application and establishing trust relationships between applications. In this way, it's not necessary to define a specific permission for binding the CacheGuard from the viewer - to avoid impersonation - because the identity of the application is already ensured by the application signing.

The viewer can then bind to the service using the AIDL interface as in the following example:

Listing 3.10: Binding to CacheGuard from the PDF viewer application

```
interface CacheGuard {
        void openCachedFile(long fileId);
}

public class PDFViewer extends Activity implements OnFileReadyListener{
        private CacheGuard service = null;

        private ServiceConnection svcConn=new ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                                        IBinder binder) {
                service=CacheGuard.Stub.asInterface(binder);
        }

        public void onServiceDisconnected(ComponentName className) {
                service=null;
        }
        };

        @Override
        public void onCreate(Bundle icicle) {
                super.onCreate(icicle);
```

```

		bindService(new Intent(CacheGuard.getName(),
			svcConn, Context.BIND_AUTO_CREATE);

		Intent i = getIntent();
		service.openCachedFile(i.getIntExtra(FILE_ID, −1);
	}

	@Override
	public void onFileReady(MemoryFile result) {
		InputStream input;
		if (result != null) {
			input = result.getInputStream();
		}
		// Render the file
	}

	@Override
	public void onDestroy() {
		super.onDestroy();

		unbindService(svcConn);
	}
}
```

A benefit from running the PDF rendering in another process is that this operation can be often resource intensive and result in a crash. If the viewer was running in the same process of Liferay Safe, and consequently in the same process of CacheGuard, a crash of the viewer would cause a restart of the service and access to private cache would be lost.

Finally, Liferay Safe Viewer should not have a launcher activity specified in the AndroidManifest.xml, since this application is not meant to be opened directly from the user but it can only be started from Liferay Safe.

Bibliography and URLs

[1] *Security Issues to Escalate as 350m Employees to Use Personal Mobile Devices at work by 2014*, available at `http://www.juniperresearch.com/viewpressrelease.php?pr=330`

[2] Marc Blanchou, *Auditing Enterprise Class Applications and Secure Containers on Android*, iSEC Partners, 2012.

[3] *Anatomy of a Mobile Attack*, available at `https://viaforensics.com/resources/reports/best-practices-ios-android-secure-mobile-development/mobile-security-primer/`

[4] *Security in a BYOD Era*, available at `https://viaforensics.com/mobile-security/security-byod-era-webinar-questions-answers.html`

[5] *OWASP Mobile Security Project*, available at `https://www.owasp.org/index.php/OWASP_Mobile_Security_Project`

[6] *Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter*, available at `http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UQlAn-jfYQU`

[7] *TrendLabs 3Q 2012 Security Roundup*, available at `www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-3q-2012-security-roundup-android-under-siege-popularity-comes-at-a-price.pdf`

[8] Jon Oberheide and Charlie Miller, *Dissecting the Android Bouncer*, SummerCon 2012.

[9] Adam P. Fuchs, Avik Chaudhuri, Jeffrey S. Foster, *SCanDroid: Automated Security Certification of Android Applications*, University of Maryland, College Park, 2009

[10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy, *Privilege Escalation Attacks on Android*, Ruhr-University Bochum, Germany, 2010

[11] Timothy Vidas, Daniel Votipka, Nicolas Christin, *All Your Droid Are Belong To Us: A Survey of Current Android Attacks*, CyLab, Carnegie Mellon University, 2011

[12] Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner, *Analyzing Inter-Application Communication in Android*, University of California, Berkeley, CA, USA, 2011

[13] Machigar Ongtang, Stephen McLaughlin, William Enck and Patrick McDaniel, *Semantically Rich Application-Centric Security in Android*, Pennsylvania State University, 2009

[14] William Enck, Machigar Ongtang, Patrick McDaniel, *Understanding Android Security*, Pennsylvania State University, 2009

[15] William Enck, Peter Gilbert, Byung-Gon-Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth, *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*, Proceedings of the 9th USENIX conference on Operating systems design and implementation, USENIX Association, 2010

[16] *NSA Releases a Security-enhanced Version of Android*, available at `http://www.pcworld.com/article/248275/nsa_releases_a_securityenhanced_version_of_android.html`

[17] Virgil Dobjanschi, *Developing REST client applications*, Google IO 2010, San Francisco, available at `http://www.google.com/events/io/2010/sessions/developing-RESTful-android-apps.html`

[18] *Writing an Android Sync Provider*, available at `http://www.c99.org/2010/01/23/writing-an-android-sync-provider-part-1/`

[19] *Connecting The Dots with Android SyncAdapter*, available at `http://ericmiles.wordpress.com/2010/09/22/connecting-the-dots-with-android-syncadapter/`

[20] *The revenge of the SyncAdapter: Synchronizing data in Android*, available at `http://naked-code.blogspot.it/2011/05/revenge-of-syncadapter-synchronizing.html`

[21] *Concept of SyncAdapter*, available at `https://sites.google.com/site/andsamples/concept-of-syncadapter-androidcontentabstractthreadedsyncadapter`

[22] *Security Issue Exposed by Android AccountManager*, available at `http://security-n-tech.blogspot.it/2011/01/security-issue-exposed-by-android.html`

[23] *Issue 10809: Password is stored on disk in plain text*, available at `http://code.google.com/p/android/issues/detail?id=10809`

[24] *Storing application secrets in Android's credential storage*, available at `http://nelenkov.blogspot.it/2012/05/storing-application-secrets-in-androids.html`

[25] *What should I use Android AccountManager for?*, available at `http://stackoverflow.com/questions/2720315/what-should-i-use-android-accountmanager-for/8614699#8614699`

[26] *Use Androids ContentObserver in Your Code to Listen to Data Changes*, available at `http://www.grokkingandroid.com/use-contentobserver-to-listen-to-changes/`

[27] *Obtaining Sync Status Information using SyncStatusObserver or by other means?*, available at http://stackoverflow.com/questions/7214142/obtaining-sync-status-information-using-syncstatusobserver-or-by-other-means

[28] *How to know when sync is finished?*, available at http://stackoverflow.com/questions/6622316/how-to-know-when-sync-is-finished

[29] *SyncAdapter questions: monitoring sync status; getting sync settings; sync icon*, available at https://groups.google.com/forum/?fromgroups=#!topic/android-developers/uT93EUsKXSA

[30] *How does one listen for progress from Android SyncAdapter?*, available at http://stackoverflow.com/questions/5268536/how-does-one-listen-for-progress-from-android-syncadapter

[31] http://developer.android.com/reference/android/content/BroadcastReceiver.html

[32] http://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html

[33] *Android sticky broadcast perils*, available at http://porcupineprogrammer.blogspot.it/2012/09/android-sticky-broadcast-perils.html

[34] http://code.google.com/p/google-gson/

[35] *Liferay Portal 6.1 - JSON Web Services*, available at http://www.liferay.com/documentation/liferay-portal/6.1/development/-/ai/json-web-services

[36] *Introducing JSON*, available at http://www.json.org

[37] *Using the External Storage*, available at http://developer.android.com/guide/topics/data/data-storage.html#filesExternal

[38] *Intro to inotify*, available at http://www.linuxjournal.com/article/8478

[39] Oleg Kalnichevski, *HttpClient Tutorial*, available at http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html#d5e627

[40] *Proprietary Information Law & Legal Definition*, available at http://definitions.uslegal.com/p/proprietary-information/

[41] *Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995*, available at http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML

[42] *ISO (2005) 27001: Information Security Management Specification With Guidance for Use.*

[43] *Android Security Overview - Memory Management Security Enhancements*, available at `http://source.android.com/tech/security/index.html#memory-management-security-enhancements`

[44] Jon Oberheide, *Exploit Mitigations in Android Jelly Bean 4.1*, available at `https://blog.duosecurity.com/2012/07/exploit-mitigations-in-android-jelly-bean-4-1/`

[45] *How can you detect if the device is rooted in the app?*, available at `http://stackoverflow.com/questions/3576989/how-can-you-detect-if-the-device-is-rooted-in-the-app`

[46] J. Sylve, A. Case, L. Marziale, G. G. Richard, *Acquisition and analysis of volatile memory from Android devices*, University of New Orleans, 2012.

[47] A. M. de L. Simao, F. C. Sicoli, L. P. de Melo, R. T. de Sousa Junior, *Acquisition of digital evidence in Android smartphones*, University of Brasilia, 2011.

[48] *Root exploit on Exynos*, available at `http://forum.xda-developers.com/showthread.php?t=2048511`

[49] *IOCipher: Virtual Encrypted Disks*, available at `https://guardianproject.info/code/iocipher/`

[50] *SQLCipher: Encrypted Database*, available at `https://guardianproject.info/code/sqlcipher/`

[51] *SQLCipher - Design*, available at `http://sqlcipher.net/design/`

[52] `http://www.nongnu.org/libsqlfs/`

[53] *Report on IOCipher beta dev sprint*, available at `https://guardianproject.info/2013/01/31/report-on-iocipher-beta-dev-sprint/`

[54] *Security in a BYOD Era*, available at `https://viaforensics.com/mobile-security/security-byod-era-webinar-questions-answers.html`

[55] *Content Providers and Content Resolvers*, available at `http://www.androiddesignpatterns.com/2012/06/content-resolvers-and-content-providers.html`

[56] *An Introduction to Android Shared Memory*, available at `http://notjustburritos.tumblr.com/post/21442138796/an-introduction-to-android-shared-memory`

[57] `http://code.google.com/p/droidreader/`

[58] `http://code.google.com/p/ebookdroid/`

[59] `http://www.mupdf.com/`