Development of mobile services to manage electronic health records

Auteur : Fransolet, Bastien
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité approfondie
Année académique : 2015-2016
URI/URL : http://hdl.handle.net/2268.2/1617

University of Liège - Faculty of Applied Science

Master's Thesis

Development of mobile services to manage electronic health records

Master's thesis submitted for
the degree of MSc in Computer Science

Bastien Fransolet

In collaboration with the A7 Software company

Academic year 2015-2016

Advisor: Prof. Laurent Mathy

Jury: L. Mathy, G. Leduc, P. Geurts, V. Keunen

Abstract

Development of mobile services to manage electronic health records

Bastien Fransolet
Master in computer science

University of Liège - Faculty of Applied Science
Academic year 2015-2016

Andaman7 is a mobile application for managing health records, developed by A7 Software company, based in Boncelles. This collaborative app has been designed to allow users to display, edit and exchange medical data, and by these means, to improve the communication between patients and doctors.

This master's thesis has been conducted as part of the Andaman7 project. The goal consisted in developing mobile services for Andaman7, focusing on the Android version, and more generally, providing new features and solutions to the application.

The goal of the work was reached, as many solutions have been provided. Indeed, different tasks were performed for building or improving the GUI of the Android app: the presentation menu was modified in order to be integrated in a navigation drawer, the handmade toolbar was replaced by an Android built-in Toolbar, the screen rotations and the restoration of the application state were taken in charge, and a splash screen was added. Moreover, the GUI of the rules defining which parts of the patients' records are shared with other users was re-designed, the languages and translations of the application were integrated. A notification system, based on Google Cloud Messaging, was also implemented. The confidentiality, integrity and origin of the medical data exchanged through the app server with other users was ensured, by integrating some cryptography mechanisms in the app. Finally, an analysis of two possible ways of performing secure backups of the users' electronic health records was realized.

The present paper reports the choices made and details the implementation realized throughout the thesis. It also presents some alternatives and perspectives for enhancing the provided features in the future.

Acknowledgment

I would first like to thank the A7 Software company, and particularly its CEO, Vincent Keunen, for allowing me to undertake this thesis, as well as Antoine Smolders, CTO at A7 Software, for his supervision, his availability and his good advice. I would also like to thank Sébastien Hannay and Pierre-Yves Derbaix, Software Engineers at A7 Software, with whom I have worked on some features, for their help and their disponibility too. I obviously thank the rest of the A7 team for their warm welcome.

I am also grateful to Prof. Laurent Mathy for having accepted to be my thesis advisor, and for his valuable suggestions. The door to Prof. Mathy's office was always opened whenever I had a question about my writing.

Finally, I would like to thank my family and friends for their encouragement and support throughout this master's thesis, and more generally, during my studies.

Contents

Chapter 1

Introduction

1.1 Presentation of A7 Software

A7 Software [1] is a software engineering company based in Boncelles and specializing in software for the health sector. Its founder, Vincent Keunen, was the head of Manex, a company specialized in the development of software based on advanced technologies and reliable open source software between 1986 and 2014.

In 2007, Vincent Keunen found out he was suffering from leukaemia, and 3 months later, his 10-year-old son was diagnosed with bone cancer. Both are healthy now, but during their convalescence, Vincent Keunen realized and deplored how the management of medical information was catastrophic. This alarming situation was thus the source of motivation to create Andaman7. Indeed, with this application, he decided to concentrate on helping the health sector to better communicate with patients and various trades. His wish was to create an effective tool, secure and easy to use for everyone.

Andaman7 was originally a project of Manex, but this activity was transferred in December 2014 to the new A7 Software company to focus its energies on that new project.

1.2 Presentation of Andaman7

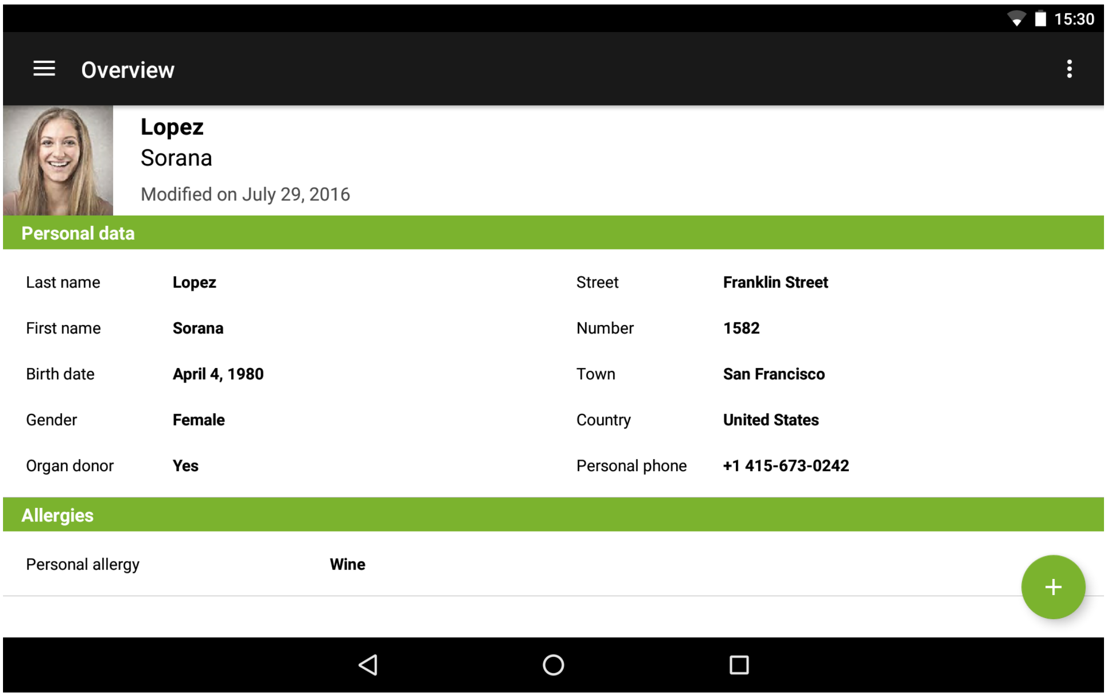
It is estimated that 30% of medication treatments are not correctly followed by patients. The will of A7 Software is to allow:

1. the patients to better follow their medications
2. doctors to have a better follow up on their patients compliance

This is the reason why A7 Software is currently working on Andaman7, a mobile application allowing to create, manage and share electronic health records to improve collaboration between doctors and patients.

Thanks to Andaman7, doctors will be able to initially enter what medication and how to intake it and share it with the patient. The system will have to display alerts to the patient when they need to take their medication and allow them to confirm the intake. That information will be transferred to the doctor.

Andaman7 is composed of a client application running on mobile devices and a back-end server written in Java, responsible for synchronizing data between them. The client application is, in fact, a combination of two mobile apps: an EHR (Electronic Health Record) for doctors and a PHR (Personal Health Record) for patients with a way to securely exchange information. Andaman7 for iPad was launched on the App Store in June 2015 and a first iPhone version was released in November 2015.



Lopez
Sorana
Modified on July 29, 2016

Personal data

Last name	**Lopez**	Street	**Franklin Street**
First name	**Sorana**	Number	**1582**
Birth date	**April 4, 1980**	Town	**San Francisco**
Gender	**Female**	Country	**United States**
Organ donor	**Yes**	Personal phone	**+1 415-673-0242**

Allergies

Personal allergy	**Wine**

Figure 1.1: Android app of Andaman7 - Overview of a patient's record

1.3 Initial state of the application

The development of the Android application had already been started by the company in the past. This code was composed of:

- a main activity,
- parts of the GUI, namely some layouts for the menu and fragments to display different sections of the application, as well as their associated Java classes,
- classes of the model, constituting the local database,
- DAO[1] classes for querying the database (cf. section 6.4.1), and

[1] Data Access Object

  - methods for accessing different web services proposed by the REST API of the server.

The (functional) code of the server was also provided, the latter being already used by the iOS app. As the server is meant to be used by both the iOS and the Android apps, some efforts were needed to upgrade the existing Android source code and maintain it compatible with the server implementation throughout the app development.

1.4 Objectives of the thesis

This master's thesis consists in the development of mobile services, focusing on the Android version of Andaman7, and more generally, in providing new features and solutions to the existing application. This application will have to interact with the existing back-end server through REST web services exposed by the server. However, the mobile application will be usable offline thanks to a client side database. The user interface will be similar to the interface of the iOS version while being consistent with the Android guidelines. Moreover, it will be adapted to tablets as well as smartphones.

At the beginning of the thesis, some existing Java code for the Android application (and, obviously, the entire server code) was provided. These pieces of code had been written between 2013 and 2014 by A7 Software's employees. Nevertheless, the development of the Android application was stopped to focus on the iOS versions. Thus the first step of the thesis consisted in actualizing and adapting the existing code. Then, the continuation of the application was done on this basis.

As the goal of A7 Software was to release, as soon as possible, both an iOS and an Android version of their application, the team also heavily worked on the Android application throughout the realization of the current master's thesis. Therefore, the Android app currently available on the Play Store integrates some of the features developed in the context of this thesis, but also and for a large part, the implementation produced by A7 Software's developers.

The present report details the various tasks and features on which I had to work in the scope of my master's thesis. The concepts needed to clearly understand each part of the work are voluntarily introduced as they are required, throughout this report, for the reader's comfort.

Chapter 2

First tasks and features

Some of the first tasks performed in the Android code are listed below. They helped to get familiarized with the project structure and the existing code.

2.1 Update of the API Level

The Android manifest [2] is a file written in XML that must be included in the root directory of every Android application. It presents essential information about the app to the Android system that is required before starting to execute the app's code. This information includes notably the Java package for the application, the name, icon and the activities of the application, permissions allowing interaction with other applications, as well as the target and the minimum level of the Android API that the application requires.

Those last two parameters are respectively set for the following reasons:

- `android:minSdkVersion`: prevents the user from installing the application if the system's API Level is lower than the value specified in this attribute. This parameter must always be declared.

- `android:targetSdkVersion`: informs the system that the app has been tested against the target version and the system should not enable any compatibility behaviours to maintain the app's forward-compatibility with the target version.

For Andaman7, the minSdkVersion was set to 16 (Jelly Bean), so as to target approximately 96% of the Android devices on the market. Since Google recommends to target the newest SDK possible, the targetSDKVersion was set to 23 (Marshmallow), as it is the last API version available for now.

Choosing the minSDKVersion is always compromising between compatibility with devices running on old Android versions and taking advantage of new mechanisms, features and widgets provided with more recent APIs, simplifying the programmers' life.

In the current configuration, less than 4% of all Android devices will not support the application, according to the following pie chart provided by Google [3]. This trade-off between development comfort and number of supported devices can be considered as fair.

Codename	API	Distribution
Froyo	8	0.1%
Gingerbread	10	1.7%
Ice Cream Sandwich	15	1.6%
Jelly Bean	16-18	16.7%
KitKat	19	29.2%
Lollipop	21-22	35.5%
Marshmallow	23	15.2%

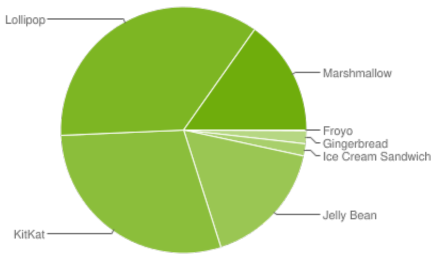


Figure 2.1: Distribution of devices running a given version of the Android platform (Data collected by Google during a 7-day period ending on August 1, 2016.) [3]

2.2 Integration of the AppCompat libraries

To backport features from newer framework releases on devices running previous APIs, but also to introduce widely-used classes not present in the framework, Android provides a set of support libraries [4].

Two major compatibility libraries are the v4 and v7-appcompat libraries. They are respectively designed to be used with API 4 and higher, and with API 7 and higher.

The v4 support library was already inserted in the project, and employed specifically to create the various app's fragments[1]. It was also needed to create a navigation drawer for the menu (cf. section 2.3.1). The v7-appcompat library was required for its implementation of the Toolbar, but also for its `AppCompatActivity` class, providing support for the `ActionBar` (cf. section 2.3.2).

2.3 Update and integration of UI components

Some existing UI components were updated in order to follow the design of current Android apps, but also to make use of facilities brought by the Support Library.

2.3.1 Navigation drawer

The Andaman7 menu was initially displayed full screen on smartphones, and as a fix portion of the UI on devices having a larger screen, such as tablets. Such a design was achieved by defining the menu as a Fragment. This way of designing the UI is encouraged by Google, as it allows to reuse a fragment in multiple activities

[1] A `Fragment` represents a behavior or a portion of user interface in an `Activity`. Multiple fragments can be combined in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. [5]

and efficiently uses the available screen space. This approach is also applied on the iOS version of Andaman7.

Nevertheless, displaying constantly the menu fragment on screen could still be problematic on some tablets or phablets, whose screens can be quite narrow. This is why the menu was integrated into a navigation drawer [6], namely a panel displaying the main navigation options on the left edge of the screen. This panel is hidden most of the time, but can be revealed when the user swipes a finger from the left to the right edges of the screen, or by touching the menu icon in the activity action bar, at the top of the app.

To add the navigation drawer to the activity, it has to be embedded as the second component of a `DrawerLayout` (support library v4) object [7]. Indeed, a `DrawerLayout` is a top-level container for window content which can only be constituted of two views: the first view contains the primary layout when the drawer is hidden, and the second one holds the contents of the navigation drawer.

This solution was finally applied to all types of devices (smartphones and tablets) in the Android app, as in every case space can be saved to show important information directly. This approach was also adopted as it is now a common practice in Android apps.

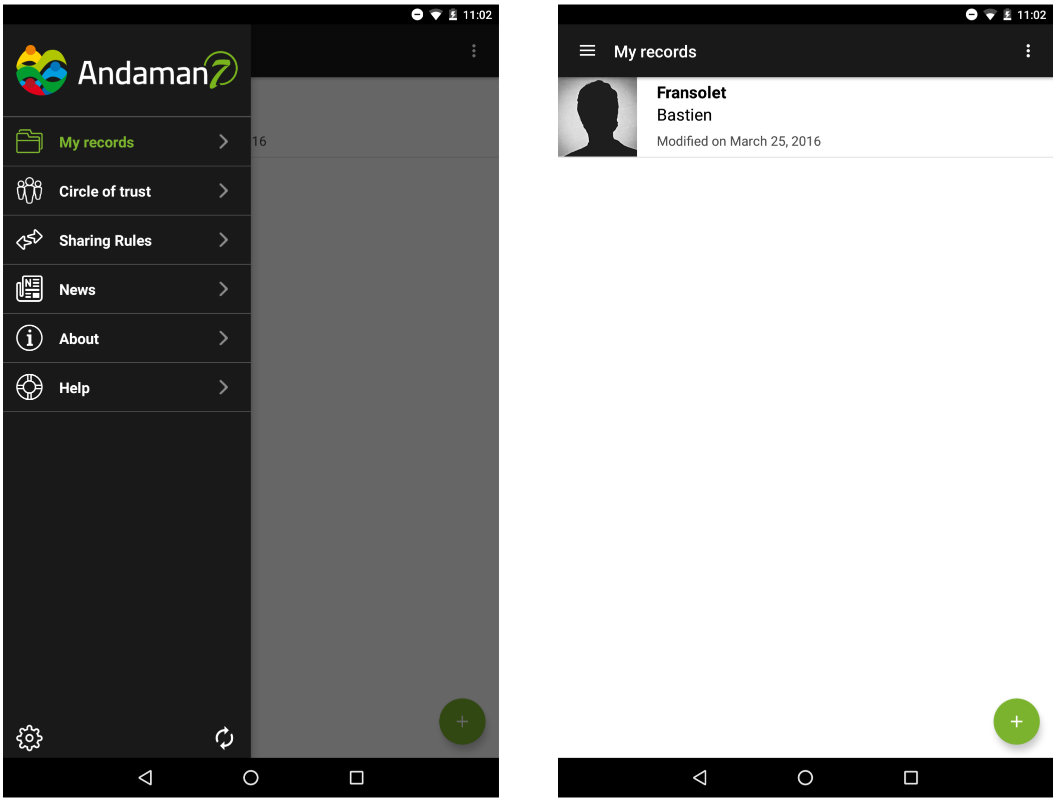


Figure 2.2: Menu's navigation drawer open (left) and closed (right)

2.3.2 Toolbar

A7 originally used for their Android app a custom toolbar, embedded as a view component in each fragment. But this solution was not using the `ActionBar` proposed by the Android framework.

The willingness of A7 being to produce a "pure Android" application, respecting the good practices and using the last stable tools provided by Google, it was decided to replace this handmade toolbar by the v7-appcompat's `Toolbar` [8]. This widget is a generalization of the traditional `ActionBar`, which provides more customization possibilities, and which notably embeds a navigation button to open or close a navigation drawer. Figure 2.2 gives an overview of the toolbar integrated in the application.

2.3.3 Splash screen and progress loader

When the application is launched for the first time on a given device, it has to

- create and initialize the local database,
- download all the translations and the supported languages from the distant server, insert them into the local database and load them in maps for quick retrievals,
- parse and load in maps the description of the panels and sections of every medical record, as well as the different `AMI`[2] types (aka TAMIs) being displayed in those sections. The idea is to be able to dynamically build the GUI and update its description from the server without having to update the entire application.

Even though those initializations are performed asynchronously in a background thread (called `InitA7Task`), the application has to wait for their completion for being operational. Therefore, a splash screen was created in order to show the progression of those tasks.

But, as the loading process should only be done at the first execution of the application, the splash screen also has to be shown only once. In this purpose, two layouts were realized: `activity_main.xml` containing the main GUI view (toolbar, drawer layout with menu and body fragment), and `activity_init.xml` which contains a ViewSwitcher [9]. A ViewSwitcher is a particular layout that can only have two child views, between which one can switch. In this case, the two children are a layout representing the splash screen, and the main view, re-used from `activity_main.xml`.

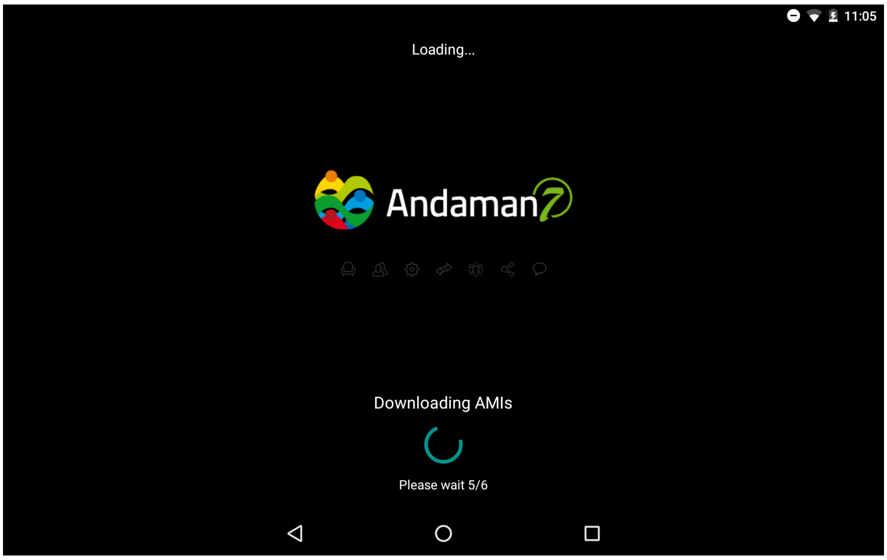
Moreover, to remember if the app was previously launched or not, a boolean called `activityStarted` is used in the `MainActivity`, and saved in the outstate bundle[3] before the activity is killed. This boolean is then recovered in the `onCreate` method of the `MainActivity` on restart.

[2] Atomic Medical Item

[3] A `Bundle` is an object containing a mapping from String keys to various Parcelable values. It is generally used for passing data between different Android fragments or activities.

If it is the first execution of the app, the bundle is null, and the `activity_init.xml` layout is defined as the activity content view. This allows to display the splash screen while loading the different data, and once the asynchronous task has finished its execution, to switch to the main view.

If it is not the first execution of the app (for example, the screen was rotated), then the boolean is recovered from the bundle and its value should be true. If it is the case, the `activity_main.xml` layout is set as the activity content view and the main view is thus directly displayed on screen. Indeed, this time, no data needs to be loaded, and thus users do not have to wait on the splash screen.



Loading...

Downloading AMIs

Please wait 5/6

Figure 2.3: Splash screen

The previous figure is a screenshot of the splash screen that has been described. The progress loader is displayed at the bottom of the screen, and is accompanied by a text explaining the task being currently performed, as well as the total loading progress.

2.3.4 Screen rotation and storage of the fragment state

Dealing with screen rotations is a feature only available in the Android version of the app. Indeed, on the iPhone version, the app can only be used in portrait, and on the iPad version, it can only be used in landscape, but it never switches on screen rotations.

The difficulty with rotations comes from the fact that activities (and their constituting fragments) are destroyed and then reconstructed on rotations. This means that the current state of the app, if not correctly saved on rotations, is lost after reconstruction of the activity.

Every fragment [5] has its own lifecycle, but this lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is destroyed, so are all the fragments in it. As both activities and fragments have their own lifecycle, both perform a callback to their `onSaveInstanceState(Bundle)` method before being destroyed. This method asks the activity or the fragment to save its current dynamic state, so it can later be reconstructed in a new instance of its process when restarted.

In the context of Andaman7, once the login or registration has been done, a single activity (`MainActivity`) is used to display in turn different fragments, as well as the menu fragment embedded in a navigation drawer. This is why it was needed to override the `onSaveInstanceState(Bundle)` method in the activity, but also in the displayed fragments.

In the activity, this method is used to save the activity state (the main fragment, and some status information about the application), and in the menu fragment and the main fragment, to store their own state, if needed. Doing so allows, at the next execution of the activity, to recover the current fragment and set it as the currently displayed fragment, restore the currently selected menu entry, and recover the state of the displayed fragment.

A second problem arose when rotating the device: the data loaded at application start up from the server (languages, translations, shared medical data, etc.) were each time re-downloaded from the server, re-inserted in the local database and then extracted and placed into maps by their corresponding controller. This was solved by checking if the bundle given as parameter of the `onCreate` method of the activity was null or not, and by keeping static references to the controllers in charge of extracting from the local database the needed information for storing them in maps. Indeed, those two measures respectively allow to know if it is the first execution of the app (in which case the bundle is null as nothing was stored in it previously) or not, and to avoid destroying the existing controllers and their maps on application restart.

2.4 Integration of the EventBus library

The traditional way for a fragment to communicate with its activity consists in defining an interface in the Fragment class [10], and implementing it within the Activity. An activity can also deliver a message to a fragment by capturing the `Fragment` instance with the `findFragmentById` method of the `FragmentManager` (an Android interface for interacting with Fragment objects inside of an Activity), and then by directly calling the required fragment's public methods.

But when two or more fragments have to interact with each other, no direct mechanism is provided by Android: the sender has to communicate its messages up to the hosting activity, which will then transmit them to each fragment in chain. The impracticality of such a solution was one of the reasons for which the EventBus library [11] was integrated to the project. Indeed, it deeply simplifies the communication between components thanks to a simple 3-step process:

1. Defining an event, possibly holding data or objects to be transmitted;

2. Preparing the subscribers (i.e. receiving activities or receiving fragments), by registering them to the bus, and by implementing event handling `onEvent` methods for each event they should listen to;
3. Posting events from the sending activity or fragment. All subscribers matching an event type will receive it.

EventBus also has the advantage of allowing one sender to directly communicate with multiple receiving entities, and this can be achieved even if the events are posted and received in different threads. Moreover, priorities can be assigned to the subscribers in order to define the order in which messages should be delivered among the receivers.

This library was used for different purposes in the code. One example of utilization is the notification of a language change. The generated event is received both by the `MainActivity`, which refreshes the toolbar title from the text of the selected menu entry, but also by the menu fragment which refreshes the text of its entries, and by the preferences fragment (from which one can change the app language) to refresh its text fields as well. As the toolbar title is based on the selected menu entry title, the menu fragment needs to be refreshed first. This is why a priority of 1 is set as second argument of the `register` method of the menu fragment. The default priority of a subscriber being 0, the menu fragment will be the first to receive the event.

Chapter 3

Application languages and translations

3.1 Android resources

Each Android project is composed of source code (collected in a `src/` directory) and resources (regrouped in a `res/` directory). The resources [12] can be of different types: layouts (xml files describing the user interface), drawables (i.e. graphics with bitmaps or xml), strings, etc. This externalization of the resources is very useful because it provides the independence between the executed code and the resources manipulated by this code. Furthermore, this separation allows to satisfy the specific needs of numerous devices that could potentially execute the application.

In particular, it is possible to provide resources adapted to the parameters of a device, such as the language or the screen size. In this purpose, one just has to group the resources in sub-directories of `src/`, by type and by configuration, and to append an appropriate configuration qualifier to the sub-directory name.

For example, while the default UI layout is saved in the `res/layout/` directory, it is possible to specify a different layout to be used when the screen is in landscape orientation, by saving it in the `res/layout-land/` directory. As a second example, in order to add support for multiple languages [13], we can create additional values directories inside `res/` that include a hyphen and an ISO language code at the end of the directory name. For example, `values-en/` is the directory containing simple resources for the Locales (language-country combinations) with the language code "en" (for English).

Android then automatically loads and applies the appropriate resources by matching the device's current configuration (in the case of the language, according to the locale settings of the device) to the resource directory names, at runtime.

3.2 Languages recovery

The languages currently supported by Andaman7 are French, English and Dutch. Nevertheless, once new languages will become available, the application will be automatically aware of it.

Indeed, the first time the application is launched, an HTTP GET request is performed by the mobile application in order to retrieve a set of languages from the server, and inserts them in the local client database. Some of those languages are activated (i.e. supported), some are not. Thereafter, the new languages are retrieved by a synchronizer, whose purpose is to regularly query the back-end server for updated or new entries in the database.

The URL of the resource called to perform such requests is the following: `{{protocol}}://{{hostname}}:{{port}}/api/context/v1/languages/`, where

- `protocol` can be "http" or "https",
- `hostname` is the IP address of the server,
- `port` is the port of the server,
- `api/context/v1/` is the web service path,
- `languages/` is the resource URI[1].

After this call, the set of supported languages is extracted from the database and put in a `Map`, associating to a language code the corresponding language name. This map is stored in the `LanguageController` class, and retrieved in the preferences to allow users to select the app language.

3.3 Translations recovery

In the same fashion as for the languages, the translations are recovered by the Android application from the server, and then inserted in the local database. The URI of the resource used in this case is the following (the rest of the URL being the same as for the languages): `translations?last-sync-date={{date}}`,

We can observe that this URI contains an additional parameter `last-sync-date`. This parameter can either be equal to `null` or to the date on which new translations were inserted in the local database for the last time:

- In the first case, all the translations are sent to the client application. The value of `last-sync-date` is set to the current date, and stored in the preferences.
- In the second case, only the translations which were created or updated after the `last-sync-date` are transmitted, in order to reduce the transfer size. The value of `last-sync-date` is only set to the current date if new translations were received.

Then, all the translations are inserted in the local database, and a map of translations is created for the currently selected language: it associates a translation key to a translation value. In this way, we have a fast way of recovering a translation for a specific text field.

[1] Unique Resource Identifier

3.4 Changing the application language

Generally, on Android, the language of an application is based on the current language of the device executing it, at least as long as this language is supported by the app. The Android system will automatically configure the application language by extracting, at runtime, the current device language from the Locale settings, and will then load the corresponding string resources necessary to build the UI.

However, in Andaman7, translations are not statically stored in string resource files located in different `values-xx/` directories as illustrated by the second example in section 3.1. They are rather retrieved dynamically from the server, and stored in the local database. This approach was adopted in the Android version to align with what had been done in the iOS app.

This technique has the disadvantage of being quite cumbersome at application start up, because it requires to contact the server and potentially store the received objects in the local database, before allowing the user to start using the app. But in return, such a method is very powerful to integrate new languages and translations to the application at any time, without having to constantly submit updates of the app on the Play Store. Another benefit of this method is that existing translations can be modified on the server, and loaded at the next synchronization of the application with the server on all client devices. Finally, it allows to use the application in a language that is different from the one of the system.

This behaviour was thus set up in the Android app as follows:

1. The `LocaleHelper` class was defined. This class is used to change the application locale and persist this change for the next time the app is going to be used. It is also used for recovering the code of the currently selected language in the Locale settings.

2. The `MainApplication` class (in Android), overriding `Application`, was created. It is the very first class that is loaded when the application is launched. Overriding this class allows to maintain some global application state.

   In our case, it restores the language selected in a previous execution of the app, or to the default one (French), if this is the first time the app is launched. This is done by calling the `onCreate` method of the `LocaleHelper`, which will modify the locale settings, and persist this change in the shared preferences for future executions.

   As `MainApplication` is the first class to be called, when the main activity is thereafter created, the application language is known (stored in the shared preferences and set up in the locale settings), and the UI can be dynamically created in the appropriate language. More precisely, every UI component displaying some text fields will, at creation, perform a call to the `getTranslation(String translationKey)` method (in `SingleInstances.java`), and receive the value of the required field in the current language. This value is extracted from a map containing the translations of the current language.

3. If the user modifies the language at runtime, through the app preferences, a call to `LocaleHelper.setLocale(Context context, String language)` is performed to update the language in `Locale` and in the shared preferences. Then, the map of translations for the new language is built after extraction from the database in a background thread. Finally, only the visible fragments, namely the menu and the preference fragments, will be refreshed by explicitly calling the `setTextFields` method defined in each of them. The other fragments do not require to be explicitly refreshed. Indeed, they will automatically be displayed in the new language when becoming visible, because the `onCreateView` method will be called in this case, and this method itself performs a call to `setTextFields`. The toolbar title is also updated with the text of the selected menu entry item, recovered from the already-refreshed `MenuFragment`.

Note that it was chosen to perform a "lazy loading" of the translations, namely the translations are downloaded at runtime from the local database for the current language only. All the translations could also have been loaded from the beginning in a "super map" containing the values for all languages. Nevertheless, it was decided not to fill up the RAM constantly with all translations, as only a fraction of them is useful at a given instant. Obviously, this decision implies some processing power to filter and load the translations on-the-fly, and it induces some delay for the user before the UI is translated. But it was felt this trade-off between processor and memory was the best choice, given the limited amount of memory embedded on current mobile devices, and the increasing power of their processors.

Chapter 4

Data representation and synchronization

In order to represent and synchronize the medical data of users, A7 have defined their own protocol [14]. This purely theoretical chapter introduces the main concepts of this protocol, and particularly those relevant for a clear understanding of the rest of this report.

4.1 Data structure

Each important Andaman7 data object is implemented as a child of `TrackedEntity`. TrackedEntity is a super-class that gives some traceability information regarding who created the object and when it was created, modified or invalidated, and specifies the corresponding user identifiers. TrackedEntity is itself a subclass of `IdentifiedEntity` that identifies each and every instance thanks to a UUID[1]. With this representation, any risk of "data conflict" is thus avoided. Note that all data classes presented in the next sections extend TrackedEntity.

4.1.1 Representation of users and their devices

An Andaman-7 account is called a `Registrar`. It is represented by a class composed, notably, of the user's email address and password, the user's profile type (doctor or patient), their associated medical record (i.e. `AmiContainer`, see subsection 4.1.2) and the list of devices in use.

The `Device` class represents a device the user has used at least once to connect to his/her registrar. This class contains fields such as the associated registrar, the device mac address, the date of the last connection to the server with this device, and other device properties.

4.1.2 Representation of the medical data

The most basic piece of information used to represent medical information, and more generally to represent each element which is part of a medical file, is the `AMI`

[1] A UUID (Universally Unique Identifier) is a string of 16 bytes allowing to uniquely identify an object in a given system, and nearly uniquely across different systems.

(Atomic Medical Item). Each AMI is a key-value pair that corresponds to a single piece of health information, such as the weight or tension of a person, a disease, an allergy, etc. An AMI also contains fields indicating the type of medical data (called Tami) it depicts.

Nevertheless, AMIs are in fact rarely directly manipulated, but are rather extended into AmiBases. An AmiBase extends the AMI class, and depicts a collection of AMIs, e.g. an AmiBase of type "allergy" can contain a collection of other AMIs, called Qualifiers, specifying the allergy (severity, result, etc.). Indeed, an AmiBase can contain Qualifiers, which provide additional information about the AMI, as well as AmiRefs, which are references to other (related) AmiBases.

Even though AmiBases are universally unique (as they extend the TrackedEntity class), they can never be modified. Indeed, if a new value is chosen for a given AmiBase, the later will be overridden in the database by a brand new one generated with the fresh value, but the old one will still be valid and visible in the history. The reason for doing so is to recover from a possible mistake in the future and thus restore the previously created AMI if necessary.

The medical records of a patien – or EHRs for Electronic Health Records – are finally represented by an instance of AmiContainer, which is a set of AmiBases about the patient. This AmiContainer contains thus the full medical description of a patient. A lightweight version of this class exists: AmiContainerCache. When a list of patients' medical records is downloaded by the application, only a part of those records will in fact be loaded, in the form of AmiContainerCache objects. If the full description is later needed, the corresponding AmiContainer will only then be downloaded.

4.1.3 Synchronization rules

In order to share medical records with each other, users have to define sharing rules. Those rules are objects describing, in a unidirectional flavour, the EHR or EHR group to be shared, which fields of those EHRs are included, and to which registrar or a group of registrars those data must be accessible.

Such rules are represented by Rule objects which are uploaded on the server in order to be known by all devices of a given registrar.

4.2 Data synchronization

The synchronization process is needed to synchronize some data stored in the distant database of the server with authorized registrar's devices, and conversely, to synchronize data stored in the local database of a registrar's device with the server. This process occurs periodically (the period can be configured in the app preferences), or can be triggered manually by users.

The philosophy of A7 is that "no medical data is kept on server, everything stays on the users' devices". One may then wonder how the synchronization of medical data can then be performed. The response to this question will be given later in this section.

The synchronization is in fact composed of two main subtasks:

- the context synchronization, and
- the EHR synchronization

Note that the different objects detailed in the previous section are only used locally by the server and the client app, but not sent through the network. Indeed, every object exchanged through web services are DTOs[2]. These correspond to lightweight or at least modified versions of the original objects, destined to be sent on the network.

4.2.1 Context synchronization

This part of the process consists in synchronizing the general registrar data stored at server side with a mobile device in both directions (i.e. client-to-server and server-to-client). During this task, a single date is used and compared to the creation or last modification date of the remote data to determine if they should or not be downloaded or uploaded.

The information exchanged during this first step includes the supported application languages and the translation updates, the registrar (groups) and device updates, the updates regarding the circle of trust of a registrar (i.e. the trusted registrars and the invites sent to become part of another registrar's community), as well as the sharing rules updates.

As can be seen, none of those data contains health records. They are thus non-critical data which can be stored on the server in the long term, without contradicting A7 Software's "anti-cloud" philosophy.

4.2.2 EHR synchronization

The second part of the process consists in the exchange of medical data. During this operation, health records are stored in and retrieved from table entries of the server database concerning the client device undertaking the synchronization. In this scenario, the server allocates, for each device, a kind of waiting queue in which medical data destined to it can be temporarily pushed before being retrieved by this device, or before expiring if timeout occurs. The server plays thus, to some extent, the role of a mail server receiving synchronization messages composed of a list of destinations and a set of medical records, and dispatching those records in the queue of each receiving device.

This second synchronization step is anew divided into three subtasks:

[2] Data Transfer Objects

EHR download

A client device downloads all the medical data waiting for it on the server. This operation is achieved by the server by simply checking its database for packets in the "waiting queue" of the client device, implemented by the entries of the `MedicalDataExchange` table in which the `deviceDestination` field corresponds to the calling device.

The medical records are then transmitted to the client device as a list of instances of the class `RegistrarSyncContentDTO`, each containing a set of JSON-formatted objects.

Intra-registrar EHR upload

As users can connect to their account from various devices, it is necessary to have an up-to-date and identical copy of their medical data on each of them. This is the reason why, as soon as medical records are created or modified on a registrar's device, those changes must be reflected on all the other devices of that registrar. This operation is achieved through the intra-registrar upload.

To perform such an upload, the sender device first requests the last synchronization date of each device of its registrar. Thanks to these dates, all EHRs stored on the current device and that were created after the last synchronization date of each other device can be uploaded on the server. The data to be synchronized are then sent as `DevicesSyncContentDTOs`, and stored on the server in the form of entries in the `MedicalDataExchange` table. The uploaded data will then be redistributed to the other ones at their next EHR synchronization.

Inter-registrar EHR upload

The process for synchronizing the medical data shared between different registrars is quite similar to the intra-registrar one, and is still based on the last synchronization date of each receiving device, but not only.

Indeed, in this case, the sender device also has to check if the modified EHR is concerned by a rule between its associated registrar and at least another registrar. If all those criteria are met, then medical data are sent to the server as `RegistrarSyncContentDTOs`, and entries are again inserted into the distant database as parts of the `MedicalDataExchange` table.

Chapter 5

Sharing rules GUI

5.1 Sharing rules

As a reminder, a sharing rule can be defined by a registrar for sharing all or parts of their medical data with registrars of their circle of trust. More precisely, those rules can be defined for a single recipient or for a group of recipients, and concern a single EHR or a group of EHRs, as well as all fields or only some specific fields of the EHR(s).

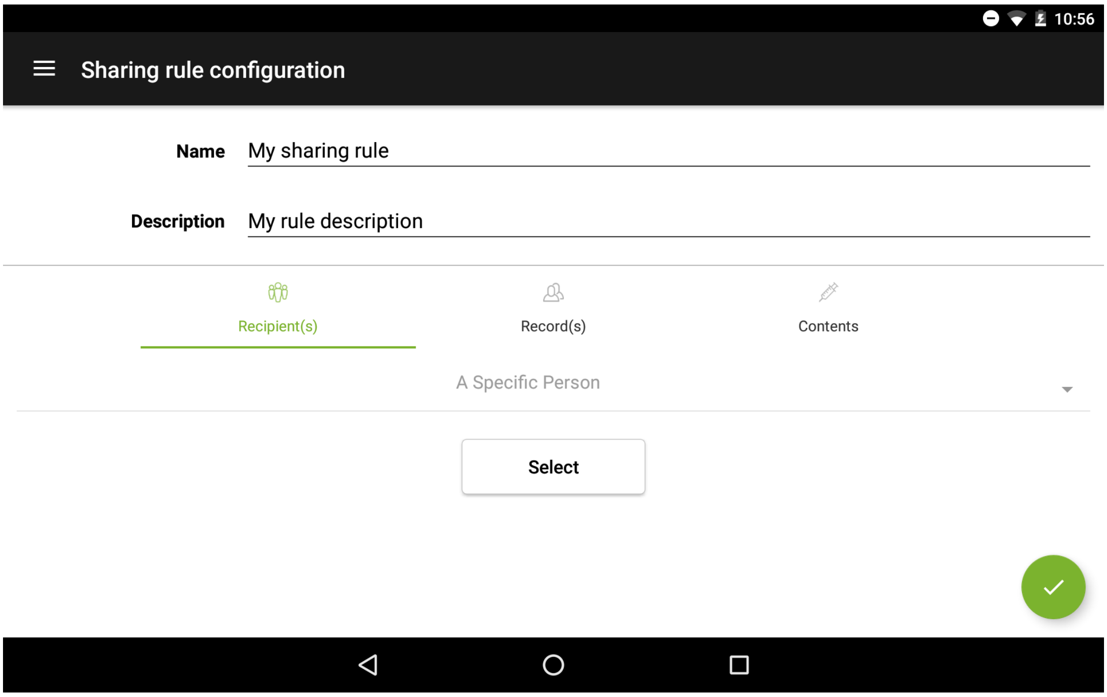
The rules are represented by `Rule` objects which are uploaded on the server in order to be known by all devices of a given registrar. They can be listed and managed in the application, under the “Sharing rules” section. A fragment, called `SharingRulesFragment` is first displayed for listing the set of existing rules. Those rules can be individually enabled or disabled thanks to switches. This fragment already existed and had been designed by the A7 team.

5.2 Rule creation and edition

Clicking on a rule or on the “Add” button opens a new fragment, `EditRuleFragment`. This fragment's GUI was the one that needed to be recreated. It had to closely resemble that of the iPhone version.

This fragment is used for both the creation of a new rule and the edition of existing ones. The only distinction between the uses is that an existing rule is provided in the first case as argument when instantiating the fragment, and not in the second case. But then, if no rule is passed to the `EditRuleFragment`, a new empty rule will be created by default at fragment initialization, and its fields will be filled in progressively as the user configures the rule. At the end of the configuration, and if the rule is valid, the latter will be inserted in the local database.

The following figure shows a screenshot of the GUI displayed when configuring a new or an existing rule.



Sharing rule configuration

Name My sharing rule

Description My rule description

A Specific Person

Figure 5.1: Screenshot of a rule creation or edition

As can be seen on the screenshot, the fragment layout is composed of two main sections:

- the name and description of the rule, and
- the configuration of the rule details.

Name and description fields are simple `EditText` views, filled in with a name and an optional description for the rule. A default text for name of a new rule is automatically generated. The interesting part of the layout is the second one. It is composed of a `TabLayout`, a `ViewPager` and a `FloatingActionButton`.

TabLayout A `TabLayout` [15] is simply a horizontal layout displaying some contents in different tabs. The `ViewPager`, instantiated when the `EditRuleFragment` is created, is then set up on the `TabLayout` by performing the following method call: `tabLayout.setupWithViewPager(mViewPager);`.

Viewpager A `ViewPager` [16], is a layout manager that allows the user to flip left and right through pages (or tabs, in this case). To generate the tabs that the `ViewPager` shows, an implementation of `PagerAdapter` must be supplied. In this case, this implementation is provided by an instance of `RulesTabAdapter`, which extends the `FragmentPagerAdapter` class [17], an implementation of `PagerAdapter` that represents each page as a Fragment that is persistently kept in the fragment manager as long as the user can return to the page. This solution is indeed recommended by the Android Developer's Guide when there is a handful of typically more static fragments to be paged through, such as a set of tabs.

RulesTabAdapter creates and keeps in a map the set of instantiated fragments representing each a single tab. When a tab is selected by the user when clicking on the corresponding tab icon, RulesTabAdapter returns the correct fragment instance to the EditRulesFragment, which selects it as the currently-displayed tab.

As all tabs exhibit a similar layout, namely a spinner, a selection button and a TextView for displaying the user's choice, all fragments instantiated by the RulesTabAdapter are children of the abstract RuleTab class, representing a fragment with this common layout. The specificity of each fragment, such as the proposed spinner entries, the specific action of the selection button and the verification of the validity of the entered entries are indeed specified in three children classes: RuleDestinationTab, RuleEhrTab and RuleContentTab. The action realized by the selection button in those fragments is to open, respectively, the fragment displaying the circle of trust or groups of registrars of the current user, the registrar's EHRs or groups of EHRs, and the list of possible AMI types to share.

With this organization of the code, the specific contents and actions of each tab are well separated from each other, and each tab is responsible for checking the validity of the values entered in it by users, while the common operations are regrouped in a single location (i.e. the RuleTab). The following UML diagram illustrates this organization.

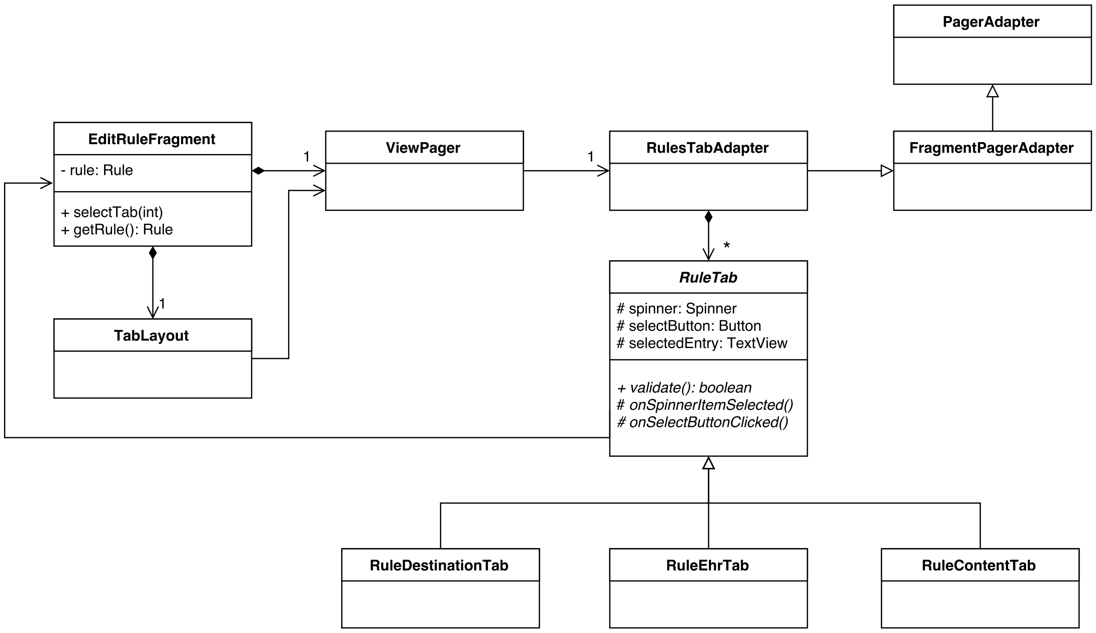


Figure 5.2: Class diagram: sharing rules GUI

FloatingButtonAction Finally, a FloatingButtonAction [18] was used, during the rule specification, originally to switch from the current tab to the next one, to check the values entered in the current tab, and validate the entire rule when the last tab was reached. Doing so required to disable the onClick listener intrinsically defined on each tab of the tab layout, to prevent users from switching tabs non-sequentially. Nevertheless, this could not be done as no built-in method of TabLayout allowed to disable tab selection, and as it was not possible to overwrite the TabLayout.Tabs internal class either, which defines this listener. The behaviour of the button was thus slightly modified by A7 thereafter: when clicking FloatingActionButton, the validity of the entire rule is checked by calling the "validate" method of each tab, and incorrect configurations are pointed out, if any. Users have thus to click on the different tabs to configure the different parts of the rule, and the button is used to validate the rule in its entirety, and trigger the storage of the new or modified rule in the local database.

Chapter 6

Notification system

6.1 Introduction and requirements

Another important feature implemented in the context of this thesis is the generation and storage of notifications for the application. Indeed, no notification mechanisms were proposed in the Android version of Andaman7. Only the iOS version already used APNS[1] for transmitting messages between the server and an iPhone or an iPad. Nevertheless, such a service is dedicated to Apple products and could not be adapted to the Android version of Andaman7. The team wanted thus a unique and centralized way of warning all users of important activities or news regarding their account, and by that means, encouraging them to open and use their application.

After discussion with the team, it was decided to initially create 3 different types of notifications:

- Notification for the reception of **new sharing rules**. Such a notification is generated when a new rule is created by a registrar for sharing (parts of) their medical data with at least one other registrar. This notification, displaying the name of the sender, must thus be sent to all recipients of the rule to let them know new medical data is available.
- Notification for the reception of a **new invitation**. This notification type is sent to a registrar who is invited by someone else to be part of their circle of truth. The sender's name constitutes the notification payload in this case too.
- Notification for a connection from a **new device**. This last type of notification is sent to a registrar who connects to their Andaman7 account with a new device. The notification is sent to all other devices of that registrar, and contains the model of the new device from which the user logged in. Such a notification can be useful especially to warn the user in case of credential theft.

In addition to displaying the notifications on an end-user's device as they are received, the A7 team wanted to create a history of the notifications received on a given device in a specific view of the application. Therefore a notification storage management was also needed at server and client sides. This functionality will be further described in the following sections.

[1] Apple Push Notification Service

6.2 Google Cloud Messaging

A notification service was needed to distribute messages from the server to various client devices. The Google Cloud Messaging (GCM) service [19] was chosen. GCM is a completely free service proposed by Google to transmit notifications between server and client applications, via so-called "downstream messages", but also from client apps to servers, via "upstream messages".

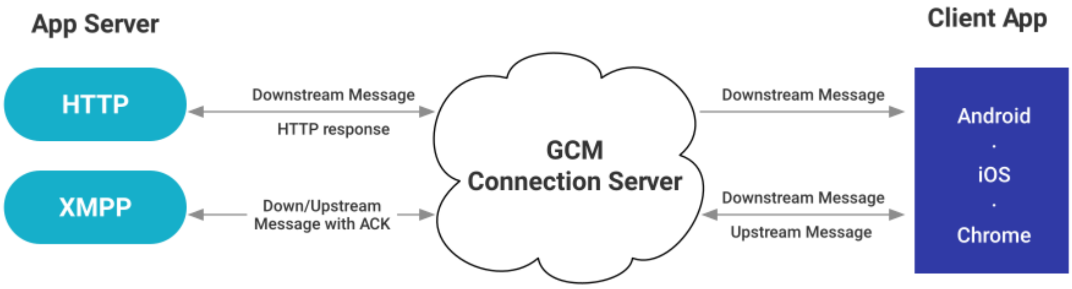


Figure 6.1: GCM architecture [20]

Figure 6.1 illustrates the three main elements constituting every GCM implementation:

1. The *GCM connection server*, accepting downstream (resp. upstream) messages from the application server (resp. the client app), queuing those messages, and finally forwarding them to the client application (resp. the app server) when the target device is online.

2. An *application server* (in this case, A7's back-end server) interacting with the GCM connection server.

3. A *GCM-enabled client application* (i.e. the Andaman7 application installed on Android or iOS), which must register with GCM to get a unique identifier called a registration token.

In the context of Andaman7, no upstream message (i.e. from client to server applications) had to be sent. Indeed, as a synchronization mechanism between the client application and the app server already existed, no further upstream communication was required. For this reason, the XMPP protocol, required for sending upstream messages, was not needed in this case, and the downstream communications were all performed using HTTP (the top part of the above diagram). For the rest of this chapter, only the HTTP protocol will be considered in the provided explanations.

Moreover, credentials are used in GCM to authenticate all parties. Those credentials include:

– A *sender ID*, which is a unique numerical value created when configuring the API project in the Google Developers Console, and used in the registration process to identify an app server allowed to send messages to the client app.

- An *API key*, saved on the app server and giving it authorized access to the Google services. This API key is included in the header of the HTTP POST requests sent to the GCM server.

- An *application ID*, corresponding to the client app registering to receive the notifications.

- *Registration tokens* (one for each receiver device). These are secret IDs issued by the GCM connection server to the client app that allow it to receive messages on a specific device.

Note that the sender ID and the API key are obtained when registering the client app with GCM on Google Developers' site. They must be stored on the application server and provided in the header of the HTTP messages sent to the client devices. Each receiver device should have its own registration token, and transmit it to the application server, so that it can later send notifications to specific devices, identified by their tokens.

Moreover, GCM provides two types of payloads for downstream messaging: the `notification` and `data` payloads. The `notification` payload is the most lightweight option, with a 2KB limit and a predefined set of user-visible keys, and allows GCM to automatically display the message on the end user devices on behalf of the client app. On the other hand, a `data` payload can contain up to 4KB of custom key/value pairs and requires the client app for processing the data messages. The app server can also send a hybrid message, including both notification and data payloads. With this last variant, GCM handles displaying the notification payload and the client app handles the data payload. This variant is the solution that was implemented in Andaman7. Indeed, the notification payload allows to simply display a message in the notification tray of a receiver device as soon as it is received, when the app is in the background. Only in cases where the user explicitly clicks on this notification or the client app is in the foreground when the notification is received, the data payload will be processed by the client app. Here is an example of JSON-formatted message containing both a notification and a data payload:

```
{
    "to" : "APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx...",
    "notification" : {
        "body" : "great match!",
        "title" : "Portugal vs. Denmark",
    },
    "data" : {
        "Nick" : "Mario",
        "Room" : "PortugalVSDenmark"
    }
}
```

In this example, the “`to`” field corresponds to the registration token of the receiver device. The first payload is the notification payload. It describes the body and the title of the notification that will be automatically displayed on the receiver device by

GCM. As for the data payload, two custom key/values pairs are defined and should be processed by the application.

Another really useful feature of GCM is the possibility to send a single message to multiple devices belonging to a **group**. Generally, a group is a set of devices owning to a single user. This feature has been heavily used here for sending a notification to a registrar that has the Andaman7 app installed on more than one device. For that purpose, all devices in a group must share a common notification key, which is the token that GCM uses to fan out messages to all devices in the group. The notification key is thus a mapping between a particular group (i.e. a registrar in the current case) and all of the group's associated registration tokens (i.e. the registrar's devices). With a notification key, instead of sending one message to one registration token at a time, the app server can send one message to the notification key, and GCM sends then the message to all of the group's registration tokens. The device groups, and their associated notification keys, can be created on the app server and each key can regroup a maximum of 20 members. It is also possible to add or remove devices from a group via specific HTTP POST requests that will be explained later.

GCM provides a bunch of other features such as topic messaging, as well as many message options. GCM topic messaging allows the app server to send a message to multiple devices that have subscribed to a particular topic. While being a useful feature for defining a few public notifications, topic messaging was not used in this app since notifications should only be sent to and received by specific devices. Furthermore, message options are parameters that can be specified in the notification properties. They notably allow to specify the lifespan of a message stored on the GCM server, assigning a (high or normal) priority to the messages. Some of those options were used in the implementation and will be further detailed.

In brief, GCM was selected for its compatibility with iOS and Android, its relative ease of use, its management of device groups, and because of its large number of features and options making it a complete tool for sending custom notifications.

6.3 Server-side implementation

Setting up the described notification system required to adapt the server application. The following sections describe how the solution was implemented on the server.

6.3.1 Adaptation of the server model

First of all, new tables in the server database were created to store notifications, as well as some Java classes to access and manage their content.

The Andaman7 database is an SQL database and is managed using PostgreSQL [21], which is a database management system. Hibernate ORM [22], an open source object/relational mapping framework for Java, implementing the Java Persistence API (JPA), is used on the server for the data persistence to the (relational) database.

Thanks to it, persistent classes can be developed following natural object-oriented idioms (such as inheritance, polymorphism and the Java collections framework), and using specific Java annotations. As an example, here is the beginning of the implementation of the `Notification` table:

```java
@Entity
@Audited
@Access(AccessType.FIELD)
public class Notification {

    @Id
    @Column
    private String uuid;

    @Column
    private Date creationDate;

    @OneToMany(cascade = CascadeType.ALL,
        targetEntity = NotificationProperty.class,
        mappedBy = "notification", fetch = FetchType.EAGER)
    private Set<NotificationProperty> properties = new HashSet<>();

    @ManyToMany(cascade =  CascadeType.ALL,
        mappedBy = "notifications")
    private Set<Device> receiverDevices = new HashSet<>();

    public Notification() {
        this.uuid = UUID.randomUUID().toString();
        this.creationDate = new Date();
    }

    // some methods...
}
```

One can first observe that a `Notification` is composed of a unique identifier (i.e. a UUID), a creation date, and a set of `NotificationProperty` objects. Moreover, a notification is linked to the device(s) it concerns. The `@Entity` annotation specifies that the class is an entity. `@Audited` means that a history of changes made to the entity will be held. `@Access(AccessType.FIELD)` specifies that non-persistent fields must be transient or annotated with the `@Transient` annotation. The `@Id` annotation above the UUID defines it as the primary key field of the entity. The `@Column` annotations specify mapped columns for the persistent fields. Two different types of associations are also defined thanks to two annotations: a one-to-many association specifying a relationship between a notification and a collection of notification properties, and a many-to-many association between notifications and devices. The `Device` class was adapted as well, to reference the notifications dedicated to a given device using a many-to-many annotation.

`NotificationProperty` is built in a similar way and contains 3 fields: a key, a value and a notification. The key and value are strings representing each a property of the notification, and the third field refers to the `Notification` the property belongs to. Currently, a Notification is composed each time of two `NotificationProperty` objects: one for the title and another one for the body of the notification. With this representation of a notification, adding new fields (i.e. new key/value pairs) to a notification in the future will be straightforward.

Furthermore, a new field was created for storing the registration token and the last synchronization date of the notifications of a device in the `Device` table (and in the `DeviceDTO` class). A string for storing the notification key of a registrar (i.e. a GCM group) was also inserted in table `Registrar`.

Finally, DTOs (Data Transfer Objects) were implemented for exchanging notifications and notification properties in a lightweight fashion between server and clients, respectively in classes `NotificationDTO` and `NotificationPropertyDTO`.

6.3.2 Integration of GCM helper classes

A new package, called `gcmNotification`, was created for collecting helper classes dealing with GCM notifications. More specifically, classes for managing device groups, creating GCM notifications and sending them to the GCM servers were built. Those classes were adapted from the GCM sample project proposed by Google to developers on its GCM's GitHub repository [23].

HttpRequest A convenience class allowing to build and send HTTP POST requests to given URLs and with a given body. This class also defines a set of constants for the different header fields required to contact a GCM server.

Message This class defines a subclass `Builder` that allows to build, piece by piece, the notification to be sent. Especially, it defines methods for building a `notification` payload thanks to predefined constant names, as well as an optional `data` payload based on user's custom key/value pairs. Once a `Builder` is constructed, calling `builder.build()` will create a `Message` with the different fields set. `Message` also provides getter methods for those fields.

DeviceGroupsHelper This helper class provides methods for performing HTTP requests in order to manage GCM groups. Methods for creating a group, adding or removing members from it on the GCM servers are defined. The HTTP requests are performed to the GCM group endpoint using methods of class `HttpRequest`.

GcmSender The last class of the package is `GcmSender`, which defines a method for sending a downstream notification to a given device or group of devices (respectively identified by its registration token or notification key), and based on a given `Message`. This method converts into a JSON payload the provided `Message`, and then contacts the GCM send endpoint using methods of class `HttpRequest`.

Note that GCM API key and the GCM sender ID, generated during the app registration, are copied in this class for being included in the header of the messages sent.

6.3.3 Gestion of device tokens and groups

Next, it was necessary to explicitly manage device groups based on the context synchronizations of the devices. As mentioned earlier, a device group is useful to send a single message to the GCM servers which will redistribute it to every single device in the group, instead of having to send multiple times the same message to each device of a registrar individually. In order to automate the creation of device groups, and as a group can already be created for a single device, it was decided to systematically build a GCM group for every registrar using at least one active device[2] with a valid registration token. It follows that every GCM notification will always be addressed to a notification key and never directly to a registration token, letting the contacted GCM servers perform the appropriate group-to-device(s) redistribution(s).

To this aim, a `NotificationController` class was implemented. As its name indicates, this class is a controller defining public methods to send notifications (cf. section 6.3.4), but also to manage device groups. The latter methods all make use of the `DeviceGroupsHelper` class for updating groups on GCM servers. Let us review the different methods dedicated to group management, and the situations in which they are needed:

createGroupForRegistrar(Registrar)

This method takes in parameter the registrar for which a GCM group must be created and builds a list of registration tokens for that registrar's active devices. If the list is not empty, a group is created using `DeviceGroupsHelper`, and the returned notification key is stored in the `Registrar` table of the database. This method is called as soon as a user creates a new Andaman7 account (i.e. a new registrar).

Note that for creating a device group, a *notification_key_name* is required by GCM. This name must be unique to a group of registration tokens. Its purpose is to protect users from accidentally using the incorrect notification key when adding or removing registration tokens in the case where multiple client apps have the same sender ID. For the sake of simplicity, the registrar's UUID is always used as the *notification_key_name* to avoid creating a dedicated field in the database for it, and because, by definition, a UUID is unique and can thus fulfill this role.

addDeviceToGcmGroup(Device)

This method simply adds a new device to the GCM group of its associated registrar. If, for some reason, no group exists for the registrar, a brand new one is created with its active devices (including the new device) using the previous method. The method can be called after a context synchronization for 2 possible reasons:

- Either because a new device has been created for an existing registrar, or put differently, because a user logged into his or her Andaman7 account from a new device. A notification must be sent in this scenario (cf. section 6.3.4).

[2] A device is active or enabled from the first time it is used to register or to log into a registrar account. It can be disabled at client side (this feature is currently available on the iOS app only), or at server side if it is lost, stolen or broken.

- Or because the registration token of a known device was updated. This second case divides again into two subcases:
    1. The updated device had no token previously: the token must simply be stored in the Device table of the database, and the device must be added to the GCM group of the associated registrar.
    2. The device token has been refreshed: in this case, the device must be first removed from the GCM group of its registrar, then the new device token must be updated in the database, and finally, the refreshed device must be re-added to the GCM group.

removeDeviceFromGcmGroup(Device)

A device can be removed from a GCM group because its token has been refreshed, as just mentioned, or because the device has been deleted or disabled. Note that a device will, nevertheless, be rarely deleted in practice. It is, in fact, a debug feature only accessible from the server, rather than a client app feature.

Removing a device from a GCM group amounts to removing its registration token using `DeviceGroupsHelper.removeMembers`. If after this operation the associated registrar does not contain any more active devices, the notification key is also removed from it in database.

deleteGcmGroup(Registrar)

This method simply removes each device of the specified registrar one by one, using the previous method. When the GCM group is empty, it is automatically deleted on Google's GCM servers. Deleting a GCM group is only done in the rare case where a registrar is deleted.

The following activity diagram summarizes the situations in which GCM groups are managed, as well as the interactions between the different operations performed:

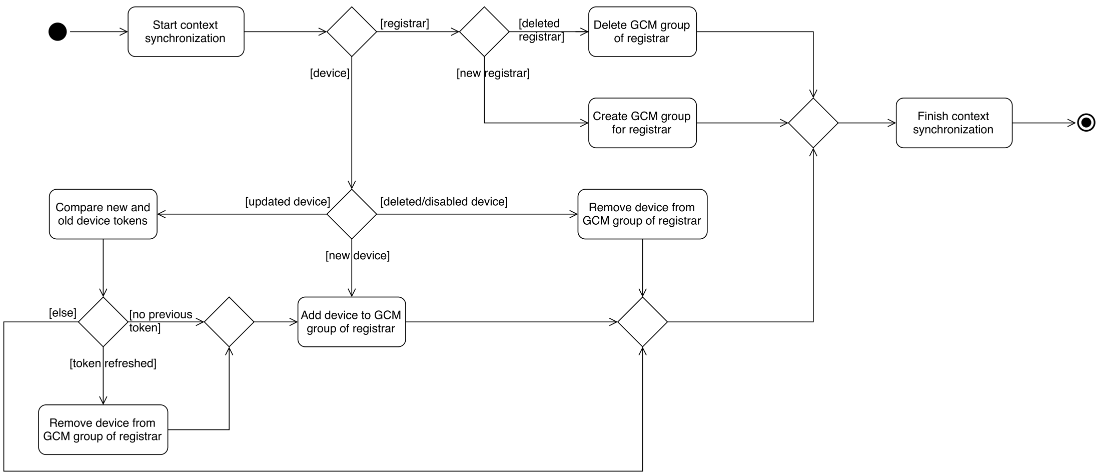

Start context synchronization
[registrar]
[deleted registrar]
Delete GCM group of registrar
[device]
[new registrar]
Create GCM group for registrar
Finish context synchronization
Compare new and old device tokens
[updated device]
[deleted/disabled device]
Remove device from GCM group of registrar
[new device]
[else]
[no previous token]
Add device to GCM group of registrar
[token refreshed]
Remove device from GCM group of registrar

Figure 6.2: Activity diagram: GCM group management

6.3.4 Generation and storage of the notifications

The 3 initial types of notifications to be generated were defined in an enumeration, namely NEW_RULE, NEW_INVITATION and NEW_DEVICE. The enumeration associates a name to each type, and two translation keys for the notification title and body. As a reminder, a translation in Andaman7 is stored in a table of the database associating a unique value for the translation to a translation key and a language. Thus, given the translation keys defined for the title and body of each notification, and given the preferred language of a destination registrar (i.e. the last language selected by the registrar on one device), it will be possible to send to a registrar a notification displayed in this language.

Let us now review the notification process per se, for each type of notification.

New device notification:

When a registrar has logged into their Andaman7 account with a new device, the `notifyNewDevice(Device)` method of the `NotificationController` is called. This method will set the notification type (NEW_DEVICE), and get the destination registrar and the new device's model to be used as the notification content.

Because the notification must be sent, in this case, to all devices of the registrar *except* the new device, it is important to first send the notification to the current registrar's associated GCM group (into which the new device has not been inserted yet), store the generated notification in the database for all devices but the new one, and, only then, add the new device to the GCM group for future notifications addressed to the registrar.

New rule notification:

When a registrar has created a new rule to share medical content with someone, the `notifyNewRule(Rule)` method is called with this new rule. Just as before, the notification type (NEW_RULE) is set by this method, as well as the notification content, which is this time the full name of the sender registrar. As seen in section 5.1, the destination of a rule can either be a single registrar or a group of registrars. It was thus necessary to look at the destination type of the rule in order to send and store as many notifications as the number of registrars concerned by the rule.

New invitation notification:

In this last case, `notifyNewInvitation` is called. This method takes as parameters the sender and destination registrars. The notification type (NEW_INVITATION) is again set, the sender's name is retrieved to be used as notification content, and the notification is sent to and stored in the table of each destination registrar's device.

As seen, all of the 3 above methods send and then store in database, sooner or later, a notification of a given type, with a given piece of information and to a (set of) destination device(s). In fact, all methods call the `createAndSendAsyncNotification` method of the `NotificationController`. This method is available in two versions:

one taking a registrar and a second one taking a list of devices as destination for the generated notification. Its purpose is to asynchronously send and then store, a notification of the specified type, with the provided piece of information included in its body, either to all devices of the destination registrar, or to the specified list of destination devices (special case for a new device notification).

Nevertheless, the principle is the same in both versions : first, the preferred language of the destination is retrieved, and thanks to it, the title and body of the notification are generated. Note that the provided piece of information (e.g. the sender's full name in the case of a new rule notification) is also appended to the end of the translated general body text. Then, the creation date and UUID of the notification are generated. Those four pieces of information constitute the `data` payload of the notification processed by the client applications if received while the app is in foreground, as well as the content of the `Notification` and `NotificationProperty` entries stored in the server database. Then, the `notification` payload is built by setting again the title and body of the notification. Moreover, a few notification options are set: a high priority and a time to live of 3 days are specified, the Android notification icon, color and sound to be used when the notification is played in the notification tray by GCM, as well as 2 iOS options to be used in the APNS payload in order to awaken an inactive client app and display a badge on the client app home icon when the notification is received. Finally, once the information constituting the data and notification payloads has been set to build a `Message`, the latter is transmitted as parameter to `GcmSender.sendHttpJsonDownstreamMessage` along with the notification key of the GCM group, in order to build and send via GCM the final JSON-formatted notification.

Here is, as a recap, the format and fields constituting every GCM notification sent by the app server to the GCM servers:

```
https://gcm-http.googleapis.com/gcm/send
Content-Type:application/json
Authorization:key=AIzaSyZ-1u...0GBYzPu7Udno5aA
{
    "to" : "APA91bEOcbeDnL2W6E1YbYdevqx5uqG...INUjz04",
    "time_to_live" : "259200",
    "priority" : "high",
    "content_available" : true,
    "notification" : {
        "title" : "Title text",
        "body" : "Body text",
        "icon" : "ic_notif_andaman7",
        "color" : "#7bb32e",
        "sound" : "default",
        "badge" : "1"
    },
    "data" : {
        "uuid" : "123456",
        "creation_date" : "2016-08-20T10:20:00+0000",
        "title" : "Title text",
        "body" : "Body text"
    }
}
```

6.3.5 Client-server synchronization of the notifications

As a reminder, a history of notifications should be created by the A7 team at client side, with the latest notifications generated for a given device. Nevertheless, a GCM notification might not be processed because the user does not click on it in the notification tray when the application is in the background, or simply because the notification is not received due to a GCM error, or yet because it could not be delivered to the (inactive) device within the lifespan of the notification. For those reasons, a web service, implemented on the server in the `NotificationService` class, was necessary in addition to GCM for a device to recover its notifications in any case. This class defines a service method accessible at the following URI: `/api/context/v1/notifications/`, with the UUID of the device whose notifications must be retrieved, given as query parameter.

When contacted, the service calls the `getDeviceNotifications` method defined in the `NotificationController` with the device UUID as parameter, and retrieves the device's latest notifications, namely the notifications generated since the last notification synchronization, by comparing the date of last notification synchronization of that device with the creation date of the device's stored notifications. If some are found, they are returned as a list of `NotificationDTOs` to the web service, which transmits them back to the client device. After the retrieval of notifications from the server database, the date of last notification synchronization is obviously updated in the `Device` table for future calls to the web service.

The following UML diagram provides a global view of the server implementation we have described.

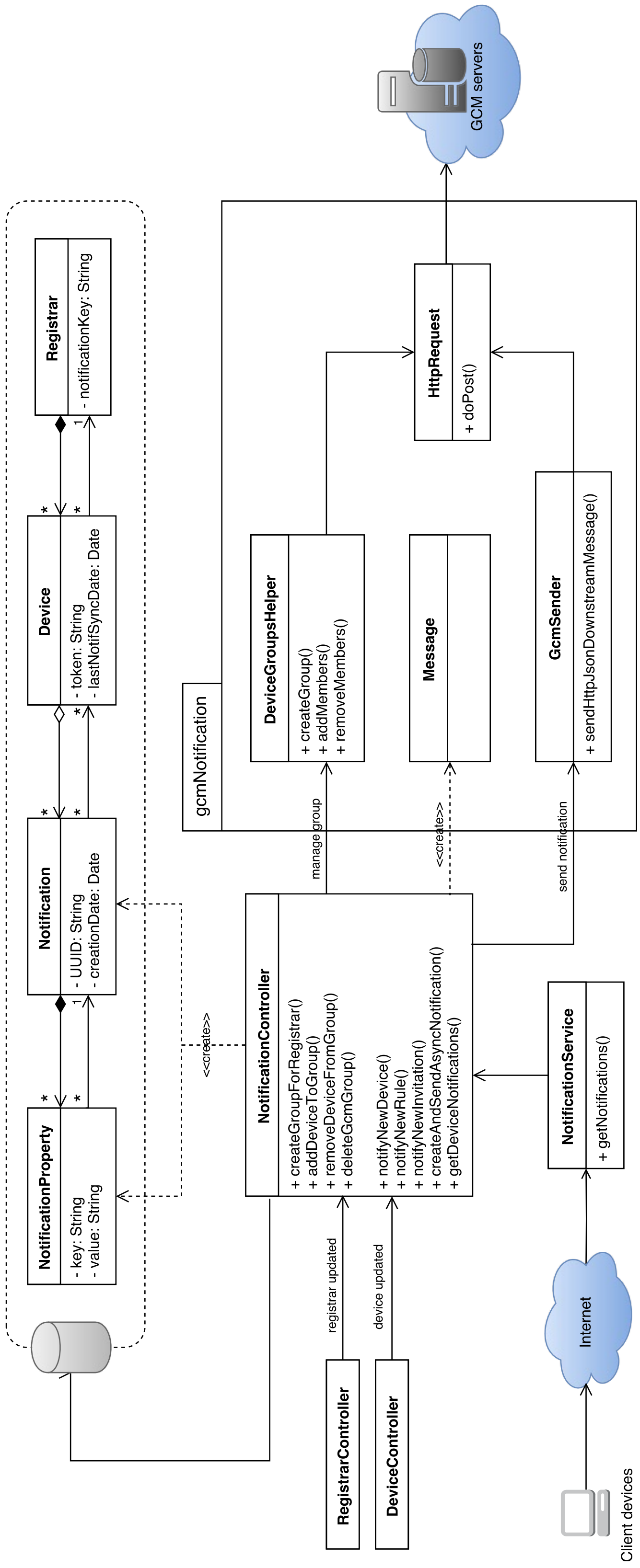


Figure 6.3: Class diagram: server implementation of the notifications

6.4 Android-side implementation

The second part of the implementation was for the mobile application. Note that, as the iOS version was not taken in charge in this thesis, the next sections only describe the adaptations and the implementation realized on the Android version of Andaman7.

6.4.1 Integration of GCM

The first step for using GCM on the Android app was to register the application on the Google Developer's website using the application's root package name, in order to generate the server API key, the sender ID, and a configuration file. The first two credentials need to be stored on the server. The configuration file, called `google-services.json`, is a JSON-formatted file providing service-specific information for the mobile app and must be added to the project. The Google Services plugin for Gradle parses configuration information from the `google-services.json` file. It was thus necessary to also add the plugin to the project by updating the top-level and app-level `build.gradle` files. The `GoogleCloudMessaging API` must be used to write the client application. In this purpose, the GCM Play Services library needed to be referenced by adding a dependency in the application's `build.gradle` file. Finally, the application manifest was also modified to add permissions for receiving messages from GCM and preventing other Android applications from registering and receiving the app's notifications. `GcmListenerService` and `InstanceIDListenerService` were also declared in the manifest to respectively enable various aspects of handling messages such as automatically displaying simple notifications on the app's behalf, and to handle the creation and update of registration tokens.

Next, three classes were defined in the `gcm` package of the app to handle the reception of notifications via GCM:

RegistrationIntentService.java This class implements `onHandleIntent(Intent)` of `IntentService`, which allows the Android application to register with the GCM connection servers and receive a unique registration token, using the `Instance ID API`[3]. Doing so is recommended by Google to avoid calling `instanceID.getToken()` in the main thread. This service is launched by the main activity at the application start. Once the token has been received, it is stored locally in the `Device` table of the local database, in order to be synchronized at the next context synchronization of the current device.

MyInstanceIDListenerService.java This second class implements a single method, `onTokenRefresh()`, of service `InstanceIDListenerService`, to update the device token, as stated before. A call to this method is automatically initiated by the `Instance ID` provider whenever the security of the previous token has been compromised. To this end, the method simply starts `RegistrationIntentService` in a new `Intent`.

[3] Google's Instance ID API provides services such as the generation of a unique ID per instance of an (Android or iOS) app, as well as security tokens for use with other services. [24]

MyGcmListenerService.java This service extends `GcmListenerService` to handle messages sent by GCM. In this purpose, the method `onMessageReceived(String from, Bundle data)` is implemented and called on notification receipt when the application is active (i.e. in the foreground). The first parameter (i.e. `from`) is the `SenderID` of the app server, and the second one (i.e. `data`) is a data bundle containing the message `data` payload as key/value pairs. The method first collects the different fields of the bundle and then calls the `displayNotification(String title, String body)` method to build and display a notification with the retrieved information. Finally, this information is passed to `NotificationController.storeNotification` to store the notification locally.

6.4.2 Adaptation of the client model

As mentioned earlier, the `Device` table had to be slightly extended to store the registration token generated by GCM for a device. If a new token is generated and inserted in this table, it is communicated to the application server at the next context synchronization.

Next, similarly to what has been done for the server database, two new classes, named `Notification` and `NotificationProperty`, were implemented for representing the tables of the local database used to store, respectively, the notifications and notification properties. The `Notification` class, simply extends `IdentifiedEntity`, an abstract class defined by A7 and heavily used to identify an entity with a unique identifier. In this case, the UUID of the notification generated at server side can be stored in it. A `Notification` is also composed of two other fields: the creation date of the notification, and a foreign collection of notification properties. As with the server database, a `NotificationProperty` contains 3 fields: a key, a value and a parent. The parent is a reference to the `Notification` the property belongs to.

As a reminder, the Android application of Andaman7 uses OrmLite to persist data in its local database. OrmLite [25] is a Java library allowing to persist Java objects into SQL databases. Compared to classical ORMs, OrmLite offers lightweight functionality to this aim, which is really appreciated when used on devices with limited performance, such as tablets and smartphones. The Java classes to be persisted in the database make use of Java annotations. More precisely, the `@DatabaseTable` annotation must be added on top of each class, and a `@DatabaseField` annotation is required before each of the fields in the class that have to be persisted to the database. The following piece of code taken from the implementation of the `Notification` class illustrates this behaviour.

```java
@DatabaseTable(daoClass = NotificationDao.class)
public class Notification extends IdentifiedEntity {

    @DatabaseField
    Date creationDate;

    @ForeignCollectionField
    ForeignCollection<NotificationProperty> properties;

    public Notification() {}

    public Notification(String uuid, Date creationDate) {
        super(uuid);
        this.creationDate = creationDate;
    }

    // some methods...
}
```

Once an annotated class has been created, a DAO[4] can then be defined for this class. A DAO provides query, create, delete, update methods specialized in the handling of a specific annotated class. Custom DAOs can also be created by extending the abstract implementation of DAO, in order to define methods performing specific requests in the database. Doing so isolates thus the database operations from the stored data.

In addition to the annotated classes `Notification` and `NotificationProperty`, two new classes, namely `NotificationDao` and `NotificationPropertyDao`, were thus also defined for querying the corresponding database tables. One can observe on the example provided above that the `NotificationDao` has been linked to the `Notification` class with the `@DatabaseTable` annotation. The same applies for `NotificationPropertyDao` and `NotificationProperty`.

6.4.3 Adaptation of the client controller

An HTTP GET request needs to be performed to the web service implemented on the server to retrieve the last notifications for the current client device. The client-server communication is performed by the method `getNotificationsForDevice()` implemented in `AndamanNotificationService`, which returns, as a response, a (possibly empty) list of `NotificationDTOs` retrieved from the server database.

Furthermore, two controllers were implemented, namely `NotificationController` and `NotificationPropertyController`. Both are singletons and define methods for respectively managing `Notification` and `NotificationProperty` objects. More specifically, the `NotificationController` offers a method `storeNotification` for storing a notification received via GCM. This method creates and inserts a new `Notification` entry, as well as the associated `NotificationProperty` entries in the database, based on the `data` payload of the notification. The controller also defines `getNotificationsFromServer()`, a public method called at the beginning of the context synchronization, to retrieve notifications from the server if some are available. For that purpose, this method calls `getNotificationsForDevice()` from `AndamanNotificationService` we have just described, and then parses the `NotificationDTO` list received from the server, if the latter is not empty. As some of the notifications received from the server might have already been received via GCM, a verification is done on the UUID of each `NotificationDTO` in the list to avoid inserting twice a same entry in the local database. The `NotificationPropertyController`

[4] Data Access Object

implements methods to get a property from a given notification and a given property key, as well as to create new `NotificationProperty` entries in the local database from the list of DTOs received from the server.

The following UML diagram summarizes all the concepts and entities that have been introduced, and that are required by the client application to deal with notifications.

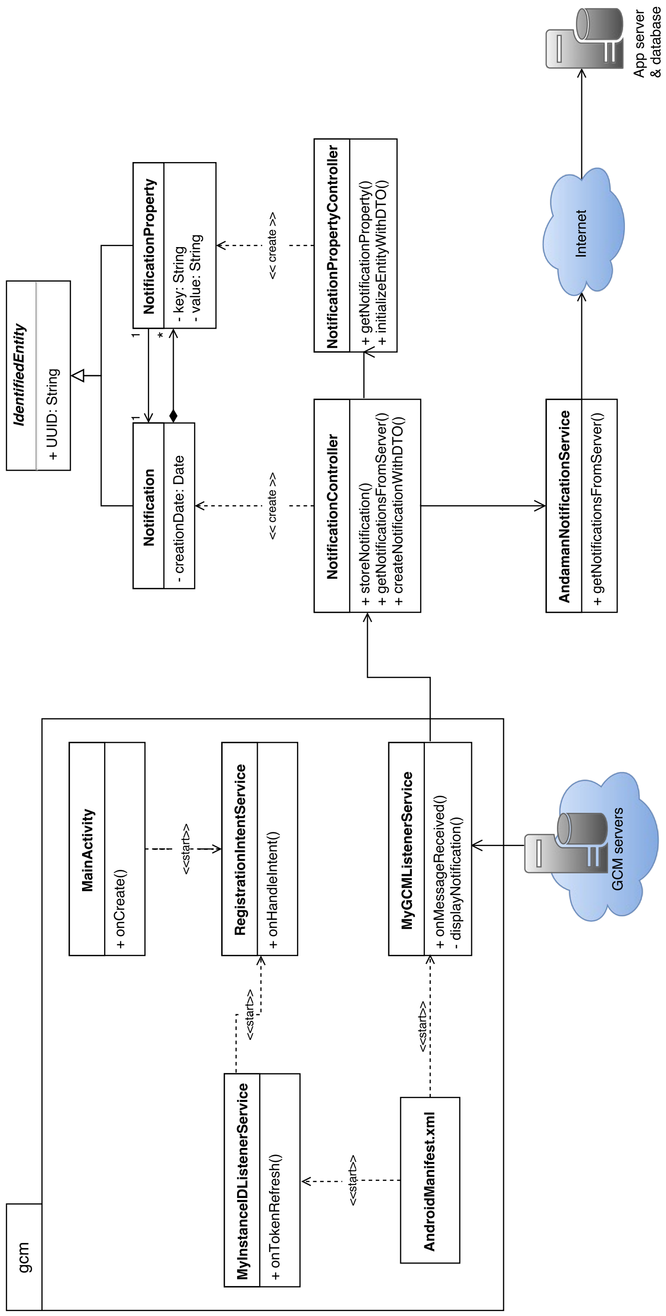


Figure 6.4: Class diagram: server implementation of the notifications

6.5 Test of the notification system

Once the implementation of the notifications was finished on the Android application, the reception of the different types of notifications and their display in the notification tray was tested. Those tests were performed using Postman [26], a Chrome app allowing the creation and sending of HTTP requests. Thanks to this application, the three types of notifications could be generated on the Android application, first using directly the device's registration token, and then the notification key of a GCM group created for the testing device. Making use of an external request builder app allowed to perform those tests while the server implementation was not completed yet, and thus to send test notifications as if they emerged from the app server.

Moreover, once the server implementation was realized, the A7 team, who wanted to integrate the notification mechanism for the next release on the Play Store, also tested the notification generation at server side. They added a new endpoint in `ContextService` which could be called to build a notification for a fake circle of trust invitation. This allowed to ensure the correct GCM notification generation and storage at server side, as well as its reception, at client side, through context synchronizations.

6.6 Integration and adaptations made by A7 Software

As just said, A7 Software have started to integrate the notification system to their app, once this first version was finished. During this integration, they performed some changes to the original code.

For example, today they do not add a `notification` payload for a new invitation notification or a new device notification, such that a notification of this type is not directly displayed on the destination device when received from the server. Instead, when received, a context synchronization is launched on the destination device in order to download from the server the new object (i.e. the new invitation, or the new device), and only then the notification is displayed. This modification allows users, when clicking on the notification, to be redirected to the new object, and thus to have the notification content on their devices when it is displayed on screen. This adaptation allowed A7 Software to remove, in the case of a new device, the device model which was originally displayed in the body of this type of notification, because clicking on it displays now directly the object referred to by the notification. Moreover, A7 Sotware added a new type of notification, generated when a new registrar validates their account after registration.

Finally, by May 2016, Google Cloud Messaging was integrated by Google to Firebase [27] and was thus renamed "Firebase Cloud Messaging" (FCM). This new version of GCM inherits its core infrastructure and its features, but simplifies the client development. As A7 Software developers had already started to integrate the developed system to the app, they also had to upgrade GCM to FCM. They notably removed the generation of registration tokens and permissions from the app manifest which are now automatically added by the new library, and they updated the server endpoints and the listener service.

6.7 Perspectives and improvements

Based on the provided implementation of GCM, it will be straightforward for A7 Software to create new types of notifications in the future. All the tools necessary to generate notifications with different options or properties, as well as managing devices groups have been set up. Those features can be easily extended to create new variants.

One point that was considered in the first place but finally put aside is the dismissal of notification replicas on different devices of the same registrar. More precisely, currently, when a notification is addressed to a given registrar, the message is sent and displayed on each of that registrar's devices when they are active, even if it has already been seen by the registrar on one device. In other words, the same notification can be seen multiple times by a registrar on different devices. Fortunately, as mentioned earlier, a time to live of 3 days limits this "issue" by deleting from the GCM servers the notifications that could not be transmitted within this period. Nevertheless, it would have been better to have a way to directly contact the GCM servers to dismiss the notification display on the remaining devices, when it has already been received on at least one other registrar's device. At present, no direct dismissal solutions are provided by Google, and all "manual" approaches considered are tricky and not really satisfactory (e.g. generating an upstream dismissal notification containing the UUID of the original notification to be discarded on the other devices). The notification system might thus be improved in this direction later.

Chapter 7

Security

7.1 Initial situation

At the beginning of the thesis, mechanisms already existed to secure the client-server exchanges, and to protect the access to some web services, both for the iOS and the Android applications. Those existing mechanisms are reviewed below:

TLS/SSL

In Andaman7, all client-server communications are ciphered with TLS/SSL.

TLS[1] and SSL[2] are two network protocols providing the confidentiality and the integrity of the data sent over TCP, as well as authentication of the communicating parties. Indeed, with TLS/SSL, the connection between client and server is made private using symmetric cryptography [28]. The unique keys and the algorithms used for encrypting the data in every connection are negotiated by the parties during the handshake phase, taking place at the beginning of the exchange. The integrity of the transmitted data can be checked thanks to a message authentication code computed on it and sent with the data to the receiver. Regarding authentication, the identity of both parties can be verified with public key cryptography during the handshake as well.

HTTP authentication

As a reminder, the Andaman7 server is composed of multiple web services. Those services are methods called by the client applications to manipulate some data on the server or to ask for the execution of specific operations.

The web services are said to be RESTful. REST[3] is a software architecture generally used with the HTTP protocol and allows the easy access to web services and to distributed contents. To this end, each resource is identified by a URI, defined by a standard HTTP method (among GET, PUT, POST, DELETE), and the objects exchanged are formatted in JSON.

[1] Transport Layer Security

[2] Secure Sockets Layer

[3] Representational State Transfer

For some API calls [29], for example to log in, end users have to be authenticated with their credentials, namely the email address and (a hash of) the password associated to their registrar. This authentication information is included as an additional HTTP header. The corresponding Web service will then only be accessible if the provided and the registered credentials are identical. By that means, the client registrar is authenticated when communicating with those services.

API key

An additional security point is the use of API keys for each request performed on the public API. An API key [30] is a unique code transmitted by the client to the server via a HTTP header, allowing to identify the calling device. Using those keys allows to track and control how the API is being used and prevents abuses. Obviously, API keys are encrypted with TLS during communication in order to keep secret.

7.2 Data on the server

Despite the aforementioned protections, no mechanisms were implemented for securing the medical data (EHR) transiting through the Andaman7's app server, when a user was sharing data with others. Indeed, in this case, the shared EHRs were temporarily stored in a table of the distant database waiting to be retrieved by the intended device(s) of the destination registrar(s). But during this time, any ill-intentioned person able to hack the server would have had the possibility to exploit those confidential pieces of information.

As stated before, the willingness of A7 Software was not to store the medical records of patients in a centralized server. Nevertheless, as peer-to-peer connections between devices were obviously not possible, the usage of an intermediate server was necessary. The positive point is the fact that the medical data stored on the server have a limited lifetime, which reduces their exposure to potential attacks of the server. However, and despite this limit, the probability of such attacks was non-negligible and had to be taken into account.

This is why an additional security layer needed to be applied directly on the data sent between devices, through the server.

7.3 Principle of the solution

To tackle the security flaw that has just been introduced, a solution inspired from PGP[4] was adopted. This solution is described piece by piece in the following sections, each of them dealing with an extra security aspect to be taken into account for a reliable and robust solution.

[4] Pretty Good Privacy

7.3.1 Data confidentiality

First, each device generates and owns a unique public/private key pair (K^+, K^-). The server centralizes the public key of each device, certifies its authenticity thanks to its certificate, and redistributes it to any device requesting it.

Now, consider the following scenario:

- Registrar A wants to share some EHRs with Registrar B,
- A performs this operation with his/her device *DeviceA*,
- B uses n different devices (say $DeviceB_1$, $DeviceB_2$, $\ldots$, $DeviceB_n$).

The following steps must thus be performed:

1. A generates a new symmetric key K_s.
2. With K_s, A encrypts the medical data to be shared.
3. A retrieves from the server the n public keys of B's devices: K^+_{B1}, K^+_{B2}, ..., K^+_{Bn}.
4. A successively encrypts K_s with every public key of B, producing thus n distinct encryptions of K_s.
5. A sends to the server the medical data encrypted in 2, along with the list of encryptions of K_s.
6. The server transmits to $DeviceB_i$ ($i \in \{1, ..., n\}$) the encrypted data received in 5, as well as the encryption of K_s destined to $DeviceB_i$.

   For each device of registrar B, $DeviceB_k$ ($k \in \{1, ..., n\}$):

7. B decrypts K_s with private key K^-_{Bk}.
8. Finally, B decrypts the medical data using K_s retrieved in 7.

Of course, the following schema can be applied to more than one receiver. In this case, the symmetric key K_s must be encrypted with the public key of every device of every destination registrar.

7.3.2 Data integrity

Up to now, no mechanism allows to detect unauthorized or accidental modifications to the shared EHRs that are temporarily stored on the server. Indeed, even if the data are now encrypted, and thus unintelligible for a potential intruder, nothing prevents the latter from modifying the content of the packets waiting on the server to be downloaded by the end devices.

Preventing such attacks on data integrity is done by computing a MAC[5] on the encrypted EHRs. A MAC can be generated thanks to a key and a hash function, namely a function that will map data of arbitrary size to data of fixed size. The transmitted packets are thus composed of the encrypted medical data, and the MAC computed on it.

Once a packet is retrieved by a destination device, the latter will then recompute the MAC on the received encrypted data using the same hash function and sender key, and compare it to the received MAC. If they are equal, then the data integrity is preserved. On the contrary, if these MACs do not correspond to each other, the data integrity is broken, and thus the data should not be exploited.

7.3.3 Non-repudiation

The non-repudiation (i.e. asserting the digital origin) of the transmitted data can, in addition, be ensured thanks to an additional authentication mechanism: a signature. A signature is in this case obtained by encrypting the MAC with the private key of the sender device.

When the data is received by an end device, the latter verifies the signature by decrypting the MAC with the sender device's public key, and then comparing it with the MAC recomputed on the encrypted medical data. If they coincide, the receiver not only knows that the data integrity is safe, but also has a strong guarantee that the message was sent by the announced sender, since only this sender knew the private key that was used to sign the MAC.

7.3.4 Summary

Let us summarize the full process on the next scenario:

- $DeviceA$ generates a symmetric key K_s, and owns a pair of public/private keys (K_A^+, K_A^-).
- B owns n devices, and each device i ($i \in \{1, ..., n\}$) has a unique pair of public/private keys (K_{Bi}^+, K_{Bi}^-).

$DeviceA$, who wants to send medical data to B, sends thus to the server, for each device i of B:

$$K_{Bi}^+(K_s), \text{ and}$$

$$[\quad K_A^-[MAC(K_s(data))] \quad || \quad K_s(data) \quad],$$

where || represents a data concatenation.

[5] Message Authentication Code

Note that, for debugging purposes, the symmetric key K_s is also encrypted with *DeviceA*'s public key K_A^+, i.e. the additional field $K_A^+(K_s)$ is also inserted in the message. This allows the sender to re-read the encrypted content in the future if needed.

7.4 Implementation of the solution

This section reports and justifies how the solution was concretely implemented in the Android application and on the server. JCE[6] was heavily used while implementing the solution, notably for the key generation, the encryption, and the MACs.

7.4.1 Cryptographic algorithms

Asymmetric keys

The asymmetric keys are generated with RSA[7]. In such a cryptosystem, the private key must be kept secret by its owner, while the public key is by definition freely available to anyone requiring it. In order to decrypt some data with RSA, only the private (resp. public) key associated to the public (resp. private) key with which the data was encrypted can be used.

In the context of this application, the private key is used by the sender to sign some medical data to be shared, and their associated public key is retrieved and used by the receiver to check this signature. In addition, the receiver's public key is needed to encrypt the symmetric key, and the corresponding private key is used by the receiver to decrypt it.

Symmetric keys

As for the encryption of the medical data strictly speaking, AES[8] was used in counter (CTR) mode [31]. This algorithm requires a nonce or an initialization vector (IV) and a secret symmetric key K_s to encrypt the data. The encryption process of this algorithm is depicted on the following schema:

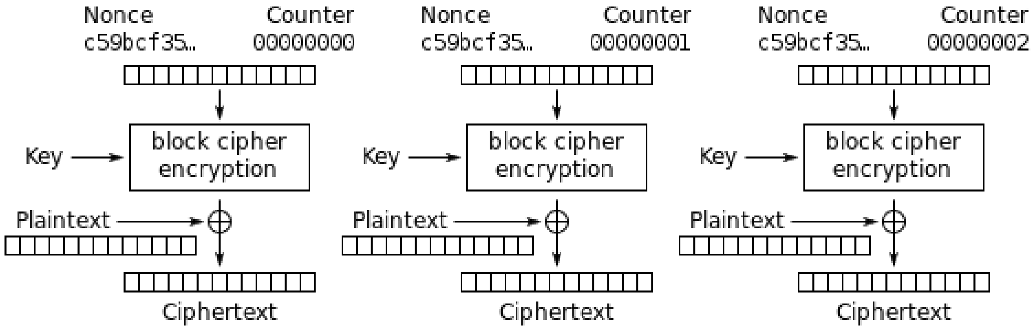


Figure 7.1: Counter (CTR) mode encryption [31]

[6] Java Cryptography Extension
[7] Rivest Shamir Adleman
[8] Advanced Encryption Standard

This algorithm is considered to turn a block cipher into a stream cipher as it generates blocks of cipher text by encrypting successive values of a counter. The counter can be any function producing a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. In the above schema, the nonce is in our case an initialization vector (IV), namely a fixed-size pseudo-random vector of bits, combined with the counter to break determinism and thus produce unique counter blocks for the encryption.

In Andaman7, 256-bit AES keys are generated and used with 16-byte IVs. Both are transmitted with the encrypted data to the receivers to allow them to perform the reverse operation, as represented hereafter:

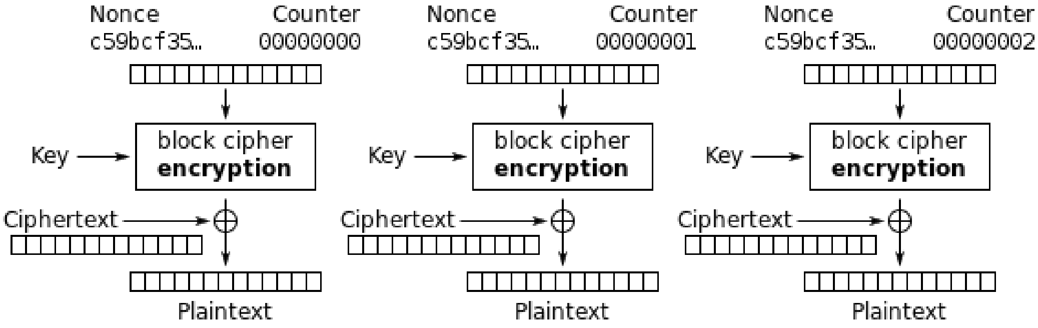


Figure 7.2: Counter (CTR) mode decryption [31]

AES is commonly used for data encryption because of its computational simplicity, compared to RSA. The CTR mode was chosen for its robustness and for its parallelization capabilities allowing it to perform efficiently on multiprocessor devices.

Message authentication codes

Finally, the HMAC[9] SHA-512 algorithm was chosen for computing a 512-bit fingerprint of the encrypted data. Such a type of message authentication code is computed thanks to a cryptographic hash function (here SHA-512) combined with a shared secret key. HMAC was chosen for its cryptographic strength; indeed HMAC is substantially less affected by collisions than its underlying hashing algorithms alone.

Given the different cryptographic algorithms and parameters we have just discussed, the exact structure of a message sent from one device to another is thus the following:

$$[\quad signed\_mac\_size \quad || \quad signed\_mac \quad || \quad IV \quad || \quad encrypted\_data \quad],$$

[9] keyed-Hash Message Authentication Code

where

- *signed_mac_size* is a 4-byte array containing the size of the signed fingerprint,
- *signed_mac* is the fingerprint computed with HMAC on the encrypted data, and signed with the sender's private key,
- IV is the 16-byte initialization vector used to encrypt the medical data,
- *encrypted_data* is the medical data encrypted using IV and K_s.

As for the encrypted symmetric key K_s transmitted with the encrypted data, a similar structure is used:

$$[\quad \textit{encrypted\_key\_size} \quad || \quad \textit{encrypted\_key} \quad],$$

where

- *encrypted_key_size* is a 4-byte array containing the size of the encrypted symmetric key,
- *encrypted_key* is the symmetric key encrypted with the receiver's public key.

Indeed, because of the variable length of an encryption or a signature, it is necessary to communicate to the receiver the exact structure of the transmitted message. This is the reason why the size of the HMAC and of the encrypted symmetric key need to be included in the messages sent.

7.4.2 Key pair generation and storage

Two different methods for generating and storing the RSA key pairs are implemented in the Android application. The choice of one method is determined by the Android platform version run by each device.

First method

The first method consists in the traditional generation of pairs of 2048-bit public/private keys. The public key is stored in clear text in the `Device` table entry corresponding to the current device, in the local database. Note that, when set or updated, this field will be synchronized with the server database at the next context synchronization. Regarding the private key, it is stored in this table entry as well, but this time encrypted with AES. In the same way as for the encryption of medical data, CTR mode is used to encrypt the private key, but the symmetric key consists, in this case, of a 256-bit digest of the user's password. By that means, even if an unauthorized person gains access to the local database files, the private key will remain confidential and will not be usable for encrypting misused contents, as long as he/she does not know the owner's password.

Second method

The second method consists in using the Android Keystore system [32] to generate and store key pairs. This solution provides a secure container which can be used by applications to store their cryptographic keys in a way that is quite difficult for malicious users and applications to retrieve. An app can store multiple keys in the Keystore, identified by a unique alias, and later retrieve them to perform cryptographic operations.

The main advantage of the Keystore is the fact that an app can only view and query its own keys, as the Keystore mitigates unauthorized use of key material outside of the Android device by preventing extraction of the key material from application processes. Moreover, the Keystore itself can be encrypted using the user's lock screen password, making it unavailable if the screen is locked.

The following piece of code is extracted from the app and shows how a new pair of RSA keys is generated:

```
Calendar cal = Calendar.getInstance();
Date start = cal.getTime();
cal.add(Calendar.YEAR, 10);
Date end = cal.getTime();

KeyPairGenerator kpg =
    KeyPairGenerator.getInstance("RSA", "AndroidKeystore");

kpg.initialize(new KeyPairGeneratorSpec
    .Builder(SingleInstances.getContext())
        .setAlias(KEYSTORE_ALIAS)
        .setStartDate(start)
        .setEndDate(end)
        .setSerialNumber(BigInteger.ONE)
        .setSubject(new X500Principal("CN=andaman7,
                                       O=A7-Software,
                                       C=Belgium"))
        .build());

kpg.generateKeyPair();
```

One can observe that a certificate is needed to store the public key in the Keystore. This certificate is a self-signed X.500 Principal certificate in the present case, set valid for 10 years after its generation.

The RSA keys can then be retrieved from the secure container as follows:

```
Keystore keyStore = Keystore.getInstance("AndroidKeystore");
keyStore.load(null);

if (!keyStore.containsAlias(KEYSTORE_ALIAS))
    return null;

PrivateKey privateKey =
    (PrivateKey) keyStore.getKey(KEYSTORE_ALIAS, null);
PublicKey publicKey = keyStore.getCertificate(alias).getPublicKey();
```

The different operations realized in this piece of code are the loading of the Keystore instance using the “AndroidKeystore” provider, the verification that the searched key exists thanks to its alias, the retrieval of the private key, and finally the extraction of the public key from the associated certificate.

Nevertheless, as mentioned earlier, the Keystore cannot be used on each Android device. Indeed, to use these functionalities, the AndroidKeystore provider is required and was only introduced in Android 4.3 (API level 18). This platform version being supported by more than 80% of all Android devices currently on the market, and the minSdkVersion of Andaman7 being set to 16, it was decided to integrate the Keystore for its security aspects, but not to block users whose devices run lower API levels. This is the reason why these two different methods were implemented.

7.4.3 Public key distribution

Client-server distribution:

Every time a new key pair is generated on a client device, the public key must be communicated to the server. These operations are required and performed in different situations:

- Either during the registration of a new Andaman user, in which case the latter is using a device which must be registered on the server too, and which requires its own pair of asymmetric keys.

- Or, when the user logs in, but the associated registrar entry is not found in the local database, and must then be authenticated with the server.

  This scenario can occur if the user logs in to his or her account with a new device. In this case, the device is not known by the server yet and must thus be registered on it, along with its public key.

  It may also occur if the current device API level is less than 18, the registrar and their device are known by the server, but the local database does not contain the encrypted private key anymore. Therefore, a new key pair must be generated and transmitted to the server. Such a situation can happen if the local database was deleted (e.g. if the application was uninstalled from the device and then reinstalled on it) and, therefore, does not contain anymore the entry corresponding to its device, nor the associated private and public keys.

  Finally, a new key pair must also be generated and sent to the server if the device API level is higher than 18, but the Keystore entry is empty, because either the app was uninstalled from the device, and thus all the Keystore entries associated to it were deleted, or because the device API was recently upgraded to a version equal to or higher than 18, and thus was not using the Keystore previously.

In the case of a registration, the generated public key is transmitted to the server through the `RegistrarDTO` sent to the server for the new registrar. This operation is performed during a context synchronization.

In the case of a login, the public key is communicated through the `DeviceDTO` sent to the server either to add a new device or update an existing device of the known registrar, again during a context synchronization.

Note that, unlike the public key, the private key will never be synchronized and stored in the server database, because this key must remain secret and can only be known by its owner device.

Server-client distribution:

In fact, public keys are not explicitly distributed by the server on a client device request. Indeed, these keys being stored in the device entries of the distant database on creation or update, they are implicitly distributed during context synchronizations of a client device with the server.

So, when a device needs the public key of any other device, it will look for the device entries that have been inserted in the associated registrar entry of its local database during a previous synchronization, and recover the public key from the appropriate device entry.

Security strength

Because devices must communicate their API key when querying the server, but also since users need to be connected with their credentials (email address and password) for using the application, it is always possible to track the origin of any data sent to the server, including the publication of the public key.

The other way round, the public key redistributed by the server to the different devices is certified by the latter thanks to a SSL/TLS certificate. So the identity of the associated registrar should not be doubted by the receiver devices when using a public key.

The only security point that cannot be ensured is the identity of the user manipulating a registrar's account. Indeed, nothing prevents the owner of an Andaman7 account to communicate his/her credentials to someone else, who will then be potentially able to handle the registrar's data.

7.4.4 Modification of the EHR synchronization process

The three types of EHR synchronizations (EHR download, intra- and inter-registrar uploads) are affected by the new security feature that has just been described. Previously, for being uploaded to the server, the medical data were simply formatted in JSON, then converted into a base-64 string and used to build an appropriate `Device-` or `Registrar-SynContentDTO`, depending on the upload type. The reverse operation was obviously performed in the case of a download of EHRs.

But now, as data must be encrypted before being uploaded, the JSON string is first converted into a byte array. Those bytes are then encrypted with a symmetric key, and the resulting byte array is then converted into a base-64 string. This final string is put in the `SynContentDTO`, along with the Map of encryptions of the symmetric key with every receiver device's public key, and the encryption of the symmetric key with the sender's public key (for debugging purpose).

Those additional operations are performed in `putMedicalDataInEnvelope()` of classes `EhrDeviceUploadSynchronizer` and `EhrRegistrarUploadSynchronizer`. Note that the "envelope", in the method name refers to the `SynContentDTO`. This method calls, in turn, the `encrypt()`, and `getCipherData()` methods respectively for the encryption and base-64 conversion of the medical data, as well as methods `getReceiversEncryptedKeys()` and `getSenderEncryptedKey()` for the encryption of the symmetric key with every receiver and sender's public key. Those methods are all defined in a new controller, called `CryptoController`.

Regarding the download of EHRs, the reverse operations are performed by the method `decryptMedicalRecords()` of class `EhrDownloadSynchronizer`. As data are received on a per-device basis, one has first to retrieve the encrypted symmetric key corresponding to its device identifier, from the map of encrypted keys transmitted in the `SynContentDTO` by the server. One has also to recover from its database the public key of the sender, based on the device UUID provided with the `SyncContentDTO`. This public key, and the decrypted symmetric key will respectively be used to recompute the MAC on the received data, and to decrypt them. Those operations are performed by the `decrypt()` method of `CryptoController`. A JSON string must be recovered as a result of the decryption process, which is then parsed into the original set of medical records.

Finally, it is noteworthy to mention that, to allow the transition from the unencrypted to the encrypted version of EHRs, A7 Software asked for a mechanism allowing to choose between both versions at first. This is why, it is currently possible, for debug purposes, to select in the app settings to encrypt or not the medical data exchanged. If enabled, the EHRs are sent in an encrypted way during synchronization, and an additional boolean field of the envelope indicates if the transmitted data should be decrypted by the receiver or not. This way, devices can exchange data encrypted or not, independently from each other.

7.5 Possible improvement: discussion

The solution described in this chapter only concerns the protection of medical data during their transmission, and especially when they temporarily transit on the application server. This solution was provided to fulfill A7's willingness to keep medical records away from their server and to store them as much as possible on users' devices. However, nothing actually protects the data stored on user's mobile devices. Indeed, the data are stored unencrypted in the local database of a client app, and nothing – except maybe encrypting the entire device [33], if this option is enabled and a lock screen password has been defined on the device in use – prevents a hacker from reading the app data stored on a device, if the latter were to be stolen or lost.

Therefore, another possible solution would be to just encrypt the medical data before storing them in database, as they are received from the server. Compared to the solution detailed above, this process would only require to encrypt the symmetric key K_S, with each receiver's public key, at the time of a synchronization, as the data to be sent would already be encrypted. Nevertheless, doing so could also slow down a bit the app performance when consulting the medical data, since they would have to be decrypted on-the-fly every time the user accesses them.

In any case, most cryptographic elements are now available in the code thanks to the implemented system, and can be freely and quite easily used as a basis by A7 Software if they decide later to encrypt the medical data on user's devices.

Chapter 8

Secure backup: analysis

8.1 Context and motivation

This last chapter aims to provide an analysis of the possible solutions that could be implemented in Andaman7 to perform backups of a registrar's EHR. Indeed, there is currently no mechanism for backing up the medical data of a given user. The motivation of implementing such a mechanism is twofold:

1. First, and above all, if a user looses, breaks his/her unique device, or uninstalls the application (maybe accidentally) from it, this user will definitely loose his/her medical records, as A7 Software do not store permanently any medical data on their server. Such a situation could be really annoying for someone who has used the application for a long time and has built a complete health file over time.
2. Second, having a means of storing one user's records would not only benefit this specific user as explained in the first point, but also any other users or organizations optionally designated and authorized by this owner. For example, a hospital could access the entire record of a patient, in specific situations (e.g. the patient is injured or unconscious), by retrieving their medical data from a backup made in Andaman7, without requiring the patient's credentials and maybe quicker than via the patient's treating doctor.

The solutions proposed in this chapter will thus have to focus on the easiness of implementation, the protection of the sensitive data against unauthorized uses, as well as the integrity and availability of these data.

Note that, in the scope of this thesis, the goal was not to implement concretely the solutions explained, but rather to propose and consider high-level ways to achieve a secure and distributed backup of users' medical data, which could potentially serve as a basis for A7 Software in the future.

8.2 First solution: backups distributed on other registrars' devices

8.2.1 Presentation of the solution

The first solution considered consists in benefiting from free disk space of other registrars' mobile devices, through the storage of users' medical records on those devices. Put differently, the medical part of a patient's record (representing a few kilobytes) would be stored on other users' devices. Obviously, this data would be encrypted with AES before being sent, for ensuring the data confidentiality. Moreover, in order to limit the exposure of those sensitive data, which will be called "a backup" once encrypted, the idea would be to store them only on devices of registrars being part of the owner's circle of trust.

In practice, the medical data would be first serialized, and then encrypted with a secret AES key (K_S). This symmetric key would then be ciphered with the public key of every user authorized to read the data (e.g. a specific registrar or a hospital). Note that a member of the owner's circle of trust, who will thus store a backup of the owner's medical records, will not necessarily be able to decrypt this backup. Indeed, it will depend on the potential sharing rules defined between the owner and the registrar, and more precisely, the hosting member will only be able to decrypt the backup if the owner had previously defined a rule for sharing their *entire* medical record with this registrar.

Moreover, for allowing the owner to restore his backup, the symmetric key K_S would also be ciphered with another key known only by the owner (let's call it K^{Owner}). This second key would again be an AES key, but derived from the user's password this time. Indeed, the goal of a backup is, for its owner, to restore the medical data if the latter has broken or lost his/her device and, consequently, the private key that had been generated for this device. One can therefore understand the necessity to have a new key that does not depend on the lost or broken device's RSA keys, and based on a piece of information that the owner should be the only to know, i.e. their Andaman7 password. AES keys of different lengths can be derived from a given password, generally by computing a hash of the password (possibly using a salt[1]) and then by making use of this hash to generate a secret key (e.g. with JCE's `SecretKeySpec` class).

Finally, the encrypted data, along with the set of ciphers of the symmetric encryption key would be sent to the server during a synchronization. The server would then redistribute to each member of the owner's circle of trust a single copy of the encrypted backup, a copy of the symmetric key K_S ciphered with the password-based key K^{Owner}, and the symmetric key ciphered with each destination device's public key (e.g. $K^{Hospital}$). Once it is done, the server will store, in a new database table, called for example "BackupMetadata", the list of devices which successfully received their backup copy.

[1] A salt is a random string which can be used as an additional parameter of hash functions, in order to extend the length and potentially the complexity of the hash. A salt does not need to remain secret and is generally stored along with the hash, or is part of the hash itself.

Later on, when a hospital or the owner will contact the server for retrieving the backup, either through the app or via an authenticated HTTP request to the corresponding web resource, the server will look inside the "BackupMetadata" table for devices having the intended user's backup. It will then contact them, one after the other, until one active device is found and can send the data to it. If a backup copy can be retrieved from one device by the server, along with the symmetric key ciphered with the calling user's key (either with a public or a password-based key), the server will send them back to this user. The latter will finally be able to decipher the received key, and decrypt the backup with it.

Note that, for the integrity and non-repudiation of the data origin, a MAC would be computed, again in this case, on the encrypted data, and then encrypted with the owner's private key. This implies for the server to communicate the owner's public key, or at least the owner's device UUID, to the calling device asking for the backup. Indeed, the owner's RSA public key will still be available in the server database and usable for checking the integrity and the origin of the backup sent, similarly to what is done in sections 7.3.2 and 7.3.3 of the previous chapter.

To sum up, the next figure illustrates the principle of the first backup solution.

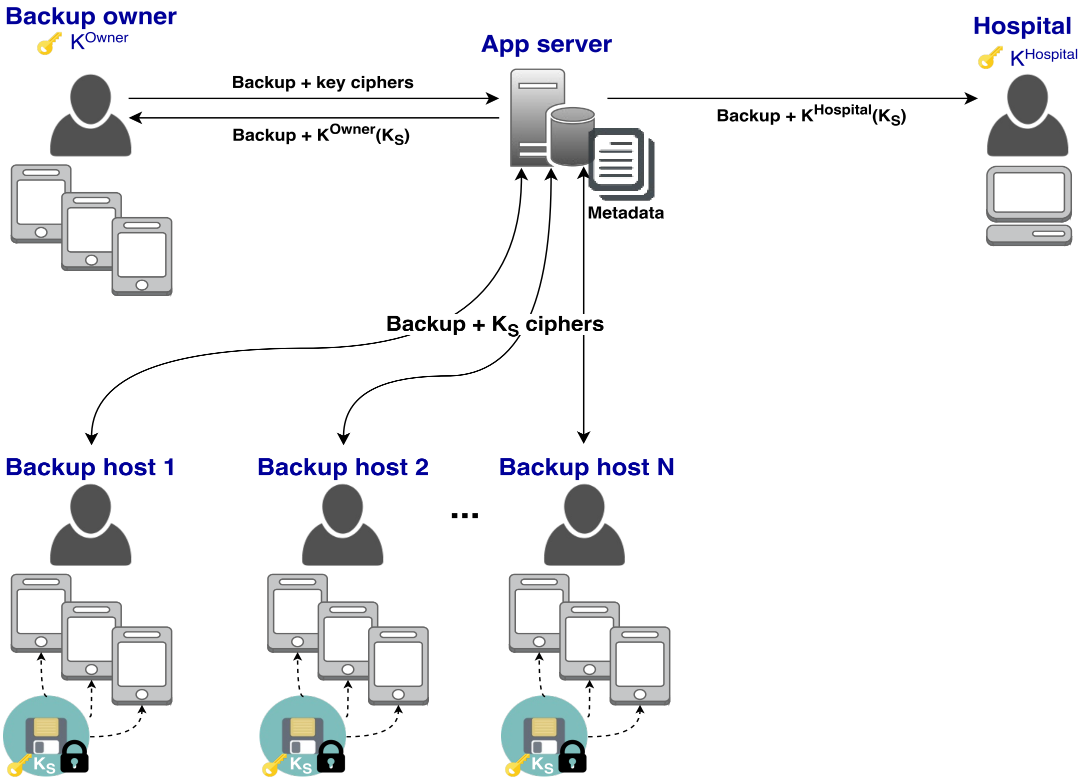


Figure 8.1: Principle of the first backup solution

8.2.2 Pros, cons and alternatives

Pros

- The "anti-cloud philosophy" of A7 Software is satisfied and the solution is costless. Indeed, by storing backups on users' devices, A7 avoids the need for making use of external storage services. This solution will make them save money and time as the solution will be directly implementable on their mobile apps and their app server.

- The probability of a user having the app installed on more than one device, as well as the intra-registrar synchronization will implicitly favour the reliability of this solution, by distributing multiple replicas of each backup among registrars' devices.

- All cryptography mechanisms implemented previously (cf. chapter 7) can be reused. They confer the confidentiality (via data encryption/decryption), authenticity (via a Message Authentication Code) and non-repudiation (via a signature of the MAC) of the backups.

- Storing backups on devices of the owner's circle of trust members limits the exposure of data to attacks, and removes the need to trust third-party persons (i.e. third-party registrars) or services (i.e. cloud providers).

- Finally, the backup data are never decrypted by intermediary hosts, except if they are specifically authorized to do so, as only the destination registrars (i.e. the owner itself or hospitals) have the possibility to decipher the symmetric key. This property provides thus an enhanced protection of the medical data exchanged.

Cons and alternatives

- This solution requires at least one member in the owner's circle of trust to function. However, this drawback can be tempered by the fact that the primary purpose of Andaman7 is to exchange medical data between users. Therefore, having no members in one's circle of trust should not be prevalent.

  Alternative: One could store the backup on devices of an arbitrary registrar, not necessarily part of the owner's circle of trust, but this alternative would remove the advantage of not having to trust a third party. Another idea would be to allow the user to store their backup in a shared folder of one of their cloud storage services (e.g. Dropbox, Google Drive, etc.), and the server would record, in the metadata, the URL of this shared folder. Nonetheless, this second option breaks, this time, the "anti-cloud philosophy" of the company.

- The availability of a backup at a given time is not ensured, if no devices supposed to store a copy of this backup can be reached at this moment. However, the probability of such a scenario to occur is decreased by the possible existence of replicas on different devices of the hosting registrars.

Alternative: Again, the backup could be stored in a cloud storage of the user, sharing the URL of the folder containing this backup with the server.

- The more friends a user will have in his/her circle of trust, the more backups this user is prone to store. Even though a backup should only weight a few kilobytes, storage space on mobile devices (i.e. tablets and smartphones) is generally quite scarce and should thus not be filled up with other user's medical data.

  Alternative: An idea to circumvent this problem would be to define a threshold on the number of backups that a given device can store or on the maximum storage space allocated for storing backups on a device, so as to prevent saturating its storage space. It would thus be possible for one (or more) device(s) of a given registrar not to have a copy of a backup `X`, because this device already holds the maximum number of backups allowed or because the allocated backup space is full, whereas `X` would be available on other devices of that registrar. This is also one of the purposes of recording on the server, in `BackupMetadata`, the exact list of *devices* having a copy of the backup.

- Encryptions of entire patients' medical records are stored on registrars' devices. This implies that the entire backup file needs to be re-generated and re-transmitted to all hosts when a single piece of information is added or updated in the medical record of a patient.

  Alternative: Alternatives to this issue can be considered, such as *differential* or *incremental* backups [34], in which one can back up the modified and/or new data based, respectively, on the first (or last) *full* backup, or on the last (incremental) backup, instead of updating the whole backup at every change. Given the representation of data adopted by A7 Software (cf. section 4.1), setting up such types of backups would be simplified by the existence of a creation date and a last modification date for every object. Indeed, at every backup, one would have to store, in the server metadata, the most recent creation and modification date among all the previously backed up data, for a given device. At the next backup synchronization, the owner would anew retrieve and send those two dates to the server, and if at least one of those values is more recent than the last creation or modification date for a given device holding a copy of the owner's backup, then the latter will be updated with the new data, and effectively perform an incremental or differential backup. Doing so would limit the contact between the server and the end devices.

8.2.3 Conclusion

Even though this first solution has some drawbacks, it would be quite straightforward to be implemented as it would reuse lots of already-existing mechanisms. Moreover, for each issue pinpointed in the previous section, alternatives can be found, but sometimes at the expense of another advantage. In short, even if quite naive, this solution carries a number of advantages making it a worthwhile possibility for backing up medical data.

8.3 Second solution: backups in the cloud

Given the problems of availability and storage capacity the first solution entails, it seems logical to consider an alternative way to back up users' data. One of the easiest solutions at hand is to consider the storage of data in the cloud, as more and more hosting providers propose reliable cloud services for free or at affordable prices.

8.3.1 Presentation of the solution

The basic idea of this second solution is that every user makes accessible a certain amount of storage on one or multiple of their cloud services to the Andaman7 application. This cloud storage would then be available for the owner's backups, via URLs stored on the server as metadata, and referencing those shared storage locations.

More specifically, assume that registrar A owns a Dropbox account (cloud storage 1) and a Google Drive account (cloud storage 2). A will thus create a shared folder at a given location on both cloud services, and reference those shared folders via their URLs (i.e. URL_1 and URL_2 respectively) in the Andaman7 app. When configuring the shared location(s), A will also have the possibility, if more than one cloud storage has been shared by A, to choose between storing the full backup in one piece in a given cloud folder, or in multiple *chunks* spread among different folders.

Indeed, *striping* is the technique which consists in segmenting logically some sequential data into stripes or chunks of a given size, so that consecutive segments are stored on different physical storage devices [35]. Doing so would allow users to store in different locations parts of their data, such that if an intruder gains access to one of those chunks, only parts of the medical records will be stolen (but still unintelligible because each chunk will also be encrypted, as explained below). Another benefit of using data striping is that, by spreading chunks across multiple devices which can be accessed concurrently, total data throughput is increased. Whereas striping can also generate a higher rate of data corruption (i.e. the full data sequence can become unusable, as soon as one storage chunk is compromised), the rest of this section will consider that a user always has more than one cloud storage and has enabled the striping functionality, disabling it being anyhow *not* in contradiction with the concepts presented hereafter. In the current example, A's medical record will thus be divided into 2 equal-sized chunks, one for each storage location configured by A.

Once the chunks are created, each of them will be encrypted with a *different* secret AES key (in this case, K_{S1} and K_{S2}). In the same way as for the first solution, each symmetric key will again be ciphered with the public key of every authorized user or hospital, as well as the password-based key of A, for allowing the personal recovery of A's backup.

Finally, each chunk will be sent to a cloud location, and the metadata, along with the list of ciphered encryption keys will be sent to the server. The metadata and the list of keys will form an entry to be inserted in the "BackupMetadata" table of the server database, composed of the following pieces of information:

- The id of registrar A (i.e. the backup owner's UUID).
- The ordered list (e.g. a `LinkedHashMap`) of folder URLs and filenames corresponding to the different encrypted chunks composing A's backup, in the order they must be assembled to reconstitute the backup file.
- A map associating to the UUID of every authorized registrar, the ordered list of symmetric keys allowing to decrypt the chunks, and ciphered with the (private or password-based) key of that registrar.

With this representation, when an authorized registrar, say B (possibly A itself), contacts the server for obtaining A's backup, the server sends to B the ordered list of folder URLs and chunk filenames, retrieved from the "BackupMetadata" entry created for registrar A, along with the ordered list of symmetric keys ciphered with B's personal key. B will then decipher the keys, retrieve the different chunks from their cloud storage, decrypt each chunk with its associated deciphered key, and merge the different chunks to rebuild the backup. As before, the integrity and origin of each chunk would be verified before decrypting it, thanks to an encrypted MAC inserted in each chunk.

Here is an illustration of the proposed solution:

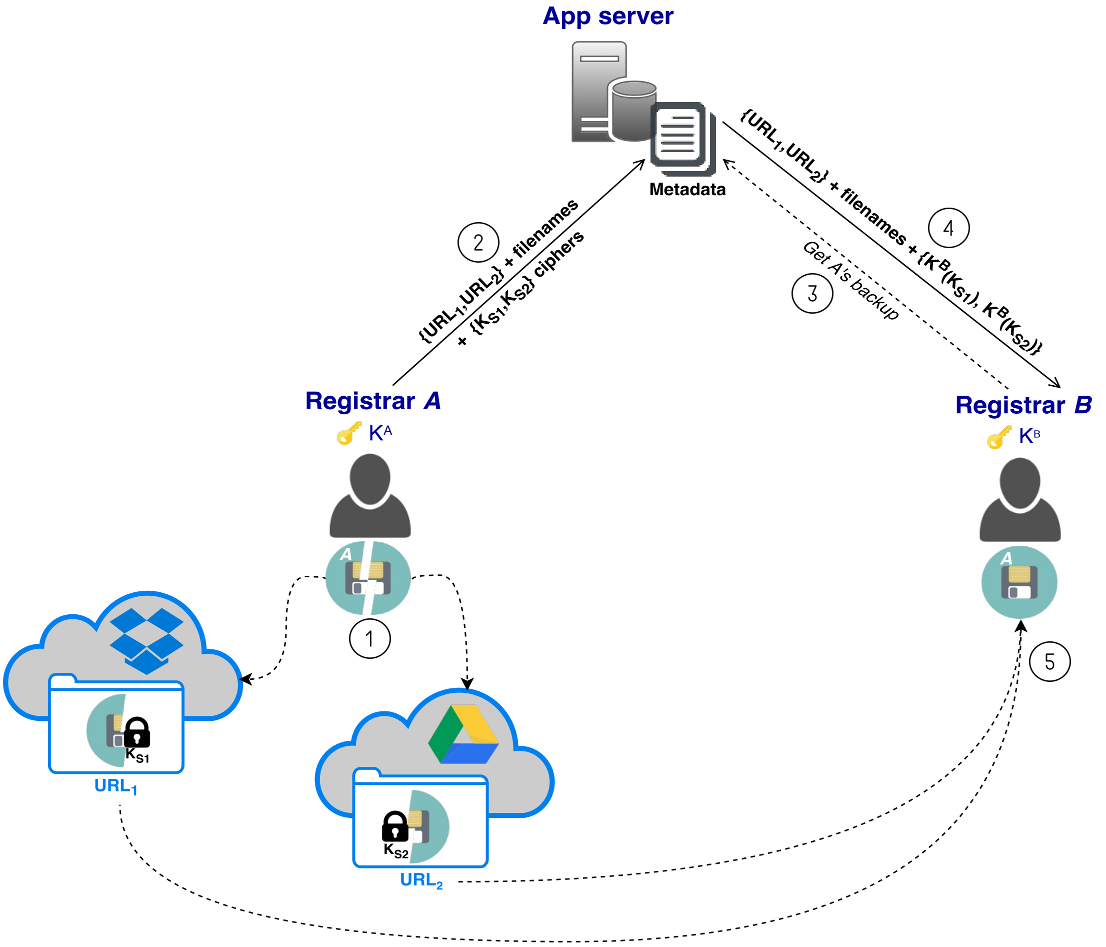

App server
Metadata
2
$\{URL_1, URL_2\}$ + filenames + $\{K_{S1}, K_{S2}\}$ ciphers
3
Get A's backup
4
$\{URL_1, URL_2\}$ + filenames + $\{K^B(K_{S1}), K^B(K_{S2})\}$
Registrar A
K^A
Registrar B
K^B
A
A
1
5
K_{S1}
URL_1
K_{S2}
URL_2

Figure 8.2: Principle of the second backup solution

8.3.2 Pros and cons

Pros

- Once again, this solution is costless for A7 Software which don't have to pay for external cloud services, since they would rely on user's personal cloud services to store backups.

- With this solution, A7 Software do not store medical data on their server, only metadata and encrypted keys. The app server never even manipulates the backups, as chunks (or the entire backups) are created, uploaded and downloaded by the end-clients on their mobile devices. And in the case where some encrypted keys were to be stolen from their server by a hacker, the latter would still require the private key of the intended registrar before being able to decrypt any medical data.

- A7 Software can rely, in addition to the encryption of the medical records before sending them, on features provided by cloud services for the replication, but also the encryption of data stored on those services. In fact, files are generally re-encrypted by cloud services when uploaded, and replicated on different disks by cloud services [36][37].

- As said before, segmenting data into chunks and scattering them across multiple cloud services enforces the data protection. Indeed, if a chunk is stolen by an unauthorized user, only part of the medical record will be revealed. Moreover, striping allows for improved performance thanks to the parallelization of transfers over different storage locations.

- Finally, the availability of the data is ensured thanks to this solution, as cloud services are supposed to be continuously accessible, as well as the Andaman7 app server which should distribute metadata to the authenticated and authorized users when required.

Cons

- The proposed solution requires the mobile app to be compliant with some cloud storage services (e.g. Dropbox, Google Drive, Amazon Drive, etc.) for storing the encrypted chunks of medical records. So, A7 Software's "anti-cloud" philosophy is not entirely respected. However, it is noteworthy that users will be aware of the fact that their medical record will be stored in the cloud, as they will configure themselves the locations in the app.

- Another difficulty is to deal with the backup updates. As the health records can be stored in chunks at different storage locations, updating a patient's record will require to replace all the chunks with the modified or new values, and update the metadata and the list of keys on the server, if only full backups are implemented. In the case where incremental or differential backups are implemented, it would be necessary to store the backup updates in the shared cloud folders as well, and reference those additional update chunks as such on the server, in order to distinguish them from the original full backup chunks.

8.3.3 Variants

It is possible to consider a lot of variants of the solution that has just been described. Some of them are proposed in the next few paragraphs.

First, instead of using shared folders in cloud services for their personal purposes only, users could also share them with any other Andaman7 users, much like in the first solution where users could lend some space on their mobile devices for storing backups of other registrars. The URLs of those shared cloud folders would thus have to be stored in a particular table of the server database, for having an inventory of available storage locations where users' backups can be stored. Nevertheless, as the cloud solution is based on the assumption that users have a free and limited cloud storage capacity, a threshold on the number of backups or on the total backup size that a shared folder can store would again be defined.

Second, the ciphered keys could also be stored elsewhere than in the app server. An idea would be to store those keys on some registrar's devices. Nevertheless, problems of availability occur if a set of keys is required while all devices storing them are inactive. Therefore, another possibility would be to store keys in the cloud too: whether dispersed in different shared folders, along with backup chunks, or all in a single dedicated folder. Either way, the app server would no longer have to store the ciphered keys, but instead, additional metadata would be needed to indicate the storage locations of those keys in the cloud.

One could also imagine having more than one type of metadata, contributing to the reconstruction of a backup: for example, a first metadata file, stored on the Andaman7 server, could indicate the way to retrieve the different (possibly encrypted) parts of a main metadata file. Once all of its chunks recovered (and maybe decrypted), the latter would be reassembled into a single piece, indicating how to retrieve the backup encryption keys and the backup (chunks).

8.3.4 Conclusion

As seen in the previous section, it is possible to add as many levels of indirection as desired in order to strengthen the data confidentiality. However, it is also important to realize that each of those levels can reduce, or at least complicate, the availability of the data (which represented one of the points to be improved when introducing this second solution), and can make the backup updates more difficult too. Therefore, A7 Software will have to find the best possible balance between data confidentiality and availability, as well as simplicity of implementation.

Chapter 9

Conclusion

This master's thesis has been conducted as part of the Andaman7 project, a mobile application for managing health records, developed by the A7 Software company in Boncelles. More precisely, this collaborative app, destined to patients and doctors, has been designed for allowing its users to display, edit and exchange medical data. The goal of this work consisted in the development of mobile services, focusing on the Android version of Andaman7, and more generally, in providing new features and solutions to the application. The latter had to interact with the existing back-end server through REST web services exposed by the server.

Given the fact that no concrete functionality had been explicitly mentioned in the title nor in the description of this master's thesis, one can consider that the goal of this work has been reached, as many features and solutions (being new or adapted from solutions already developed for the iOS version of the app) have been provided and implemented. Indeed, different tasks were performed for building or improving the GUI of the Android app: the menu presentation was modified, the handmade toolbar was replaced by an Android built-in Toolbar, the screen rotations and the restoration of the application state were taken in charge, a splash screen was added, and the sharing rules's GUI was re-designed. In addition, the languages and translations of the application were integrated, and a notification system, based on GCM, was implemented for alerting users of events regarding them. On another hand, the protection of the exchanged medical data was ensured, while being transmitted through and temporarily stored on the app server. Finally, an analysis of possible ways to perform secure backups of the users' electronic health records was realized: a first solution considers the possibility to distribute backups on other user's devices, and a second solution proposes to perform backups in the cloud.

Obviously, it is possible to improve the notification system, the security of the exchanged EHRs and other implemented features. Some of those potential improvements have been proposed in their respective sections. Sometimes it was also necessary to adapt the provided functionality so as to better stick to A7 Software's requirements and representation of data.

Moreover, the Android version of Andaman7 being simultaneously developed by A7 Software since the beginning of this master's thesis, and in production since late 2015, it was not always simple to integrate new functionalities within a code in constant evolution, nor because the requirements were not clearly defined by A7 Software. Nevertheless, this work was really interesting as it had a practical and very topical value. I consider this project as a rewarding experience. as it provided me the opportunity to establish a first contact with the professional world, which constituted one of my motivations to undertake it. It was indeed really exciting to see the company growing at a fast pace throughout the thesis, as well as the Android application growing simultaneously on the company's side and on my side.

Finally, I have developed new knowledge and skills thanks to the present thesis, whether technical knowledge (in Android programming and the use of new development tools, frameworks and APIs), theoretical knowledge (new programming concepts as well as new concepts defined by the company), or written and oral communication skills.

Bibliography

[1] A7 Sofwtare. URL: http://www.a7-software.com.

[2] Android Developers. *App Manifest*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro.html.

[3] Android Developers. *Dashboards*. Aug. 2016. URL: https://developer.android.com/about/dashboards/index.html.

[4] Android Developers. *Support Library*. URL: https://developer.android.com/topic/libraries/support-library/index.html.

[5] Android Developers. *Fragments*. URL: https://developer.android.com/guide/components/fragments.html.

[6] Android Developers. *Navigation Drawer*. URL: https://developer.android.com/training/implementing-navigation/nav-drawer.html.

[7] Android Developers. *Drawer Layout*. URL: https://developer.android.com/reference/android/support/v4/widget/DrawerLayout.html.

[8] Android Developers. *Toolbar*. URL: http://developer.android.com/reference/android/support/v7/widget/Toolbar.html.

[9] Android Developers. *ViewSwitcher*. URL: https://developer.android.com/reference/android/widget/ViewSwitcher.html.

[10] Android Developers. *Communicating with other fragments*. URL: https://developer.android.com/training/basics/fragments/communicating.html.

[11] GreenRobot. *EventBus*. URL: https://github.com/greenrobot/EventBus.

[12] Android Developers. *Resources Overview*. URL: https://developer.android.com/guide/topics/resources/overview.html.

[13] Android Developers. *Supporting different languages*. URL: https://developer.android.com/training/basics/supporting-devices/languages.html.

[14] Andaman7 Developer Portal. *A7 Protocol*. URL: http://developers.andaman7.com/a7-protocol.html.

[15] Android Developers. *TabLayout*. URL: https://developer.android.com/reference/android/support/design/widget/TabLayout.html.

[16] Android Developers. *ViewPager*. URL: https://developer.android.com/reference/android/support/v4/view/ViewPager.html.

[17] Android Developers. *FragmentPagerAdapter*. URL: https://developer.android.com/reference/android/support/v4/app/FragmentPagerAdapter.html.

[18] Android Developers. *FloatingActionButton.* URL: https://developer.android.com/reference/android/support/design/widget/FloatingActionButton.html.

[19] Google Developers. *Google Cloud Messaging.* URL: https://developers.google.com/cloud-messaging/.

[20] Google Developers. *Google Cloud Messaging: Overview.* URL: https://developers.google.com/cloud-messaging/gcm.

[21] PostgreSQL. URL: https://www.postgresql.org/.

[22] Hibernate. *Hibernate ORM.* URL: http://hibernate.org/.

[23] Google. *GCM libraries and samples.* URL: https://github.com/google/gcm/.

[24] Google Developers. *What is Instance ID?* URL: https://developers.google.com/instance-id/.

[25] OrmLite. URL: http://ormlite.com/.

[26] Postman. URL: https://www.getpostman.com/.

[27] Firebase. *Firebase Cloud Messaging.* URL: https://firebase.google.com/docs/cloud-messaging/.

[28] Wikipedia. *Transport Layer Security.* URL: https://en.wikipedia.org/wiki/Transport_Layer_Security.

[29] Andaman7 Developer Portal. *Developer's guide: Overview.* URL: http://developers.andaman7.com/guide/overview.html.

[30] Wikipedia. *Application programming interface key.* URL: https://en.wikipedia.org/wiki/Application_programming_interface_key.

[31] Wikipedia. *Block cipher mode of operation.* URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[32] Android Developers. *Android Keystore System.* URL: https://developer.android.com/training/articles/keystore.html.

[33] Android Open Source Project. *Full Disk Encryption.* URL: https://source.android.com/security/encryption/.

[34] Antony Adshead (ComputerWeekly.com). *Incremental vs. differential backup: A comparison.* Feb. 2009. URL: http://www.computerweekly.com/news/1347703/Incremental-vs-differential-backup-A-comparison.

[35] Wikipedia. *Data striping.* URL: https://en.wikipedia.org/wiki/Data_striping.

[36] Dropbox. *How secure is Dropbox?* URL: https://www.dropbox.com/help/27?path=security_and_privacy.

[37] Google. *Is Google Drive secure?* URL: https://support.google.com/drive/answer/141702?co=GENIE.Platform=Desktop&hl=en&oco=1.