Software Architecture Design
App Infra

Author	:	Raymond Kloprogge
Document ID	:	RayKlo20151216-06V02
Version	:	~~3.0~~3.1
Date of modification	:	~~2016-12-06~~2017-01-25
Status	:	Proposed
Classification	:	For internal use only

Approval

Name Reviewer(s):	Role Reviewer(s):	Review date:	Signature:
Bhargavi Upadhya	Chapter architect mobile		
Sumant Bhargav	Product owner		
Joost Westra	Tech lead		
Poornachandra Kallare	Platform architect		
Chritiene Aarts	Proposition architect		
Name Approver	**Role Approver**	**Approval date:**	**Signature:**
Sangeetha Sathiamoorthy	RTE mobile common components		

Software Architecture Design
App Infra

Revision History

Date	Revision number	Status	Author	CR/PR ID	Changes/Comments
2016-03-01	0.1	Draft	Raymond Kloprogge		• Separated off from app framework document • Added context diagram • Updated template
2016-06-10	0.2	Draft	Raymond Kloprogge		• Added more context information • Removed modules not included in PI16.3 App Infra release • Adding details on dependency injection, builder pattern, strict interface use • Significant rework in preparation of review
2016-08-10	0.3	Draft	Raymond Kloprogge		• Renamed 'feature' to 'module' in accordance to requirements document review • Reworked according to review remarks
2016-10-06	1.0	Approved	Raymond Kloprogge		• Document approved
2016-10-06	1.1	Draft	R. Kloprogge		• Reformulated service discovery • Added plain encrypt/decrypt to secure storage • Added app config module • Added REST client module
2016-10-18	2.0	Approved	R. Kloprogge		• Document approved, no review changes
2016-11-22	2.1	Proposed	R. Kloprogge		• Added A/B testing • Content loader • Secure API signing • Added data handling sections to all modules • Slight restructure to emphasize module hierarchy
2016-12-06	3.0	Approved	R. Kloprogge		• Updated architecture diagram • Improved A/B test and tagging data flow explanation • Document approved
2017-01-25	3.1	Proposed	R. Kloprogge		• Added PRX client and Secure DB • Added missing SOUP • Updated secure storage responsibilities and data handling • Added service discovery's splitting off platform URL set • Added app config cloud update • Added responsibility to API signing

Open Issues and/or Actions

N/A

Software Architecture Design
App Infra

Table of Contents

Software Architecture Design App Infra

1 Introduction

1.1 Purpose

This document describes the high-level software architecture of the mobile App Infra component.

1.2 Scope

The scope of this document is limited to the mobile App Infra component, which is to be used by common components and applications. This document does not describe the application as a whole, nor does it describe in detail any cloud services that may be related to the functionalities covered in the App Infra component.
This document does not include an extensive API description; please refer to [AppInfraInterfaceAndroid] and [AppInfraInterfaceiOS].

1.3 References

Reference ID	Document ID	Document Title
[AppInfraDesignAndroid]		*Mobile App Infra Android design specification* App Infra Android Team
[AppInfraDesigniOS]		*Mobile App Infra iOS design specification* App Infra iOS Team
[AppInfraInterfaceAndroid]		*Mobile App Infra Android interface specification* App Infra Android Team
[AppInfraInterfaceiOS]		*Mobile App Infra iOS interface specification* App Infra iOS Team
[AppInfraSWReqSpec]	v4.0	*Software Requirements Specification* Raymond Kloprogge RayKlo20151216-08V40

1.4 Definitions & Abbreviations

Term	Description
app	Mobile application
App Infra	Mobile application infrastructure library
CoCo	CDP2 Common Component
HTTPS	Hypertext Transport Protocol through TLS
OS	Operating system
REST	REpresentational State Transfer
SW	Software
TLS	Transport Layer Security

2 System Overview

The following block diagram provides a high-level overview of App Infra in the wider context of the proposition app, the App Framework, micro apps, embedded devices and cloud services:

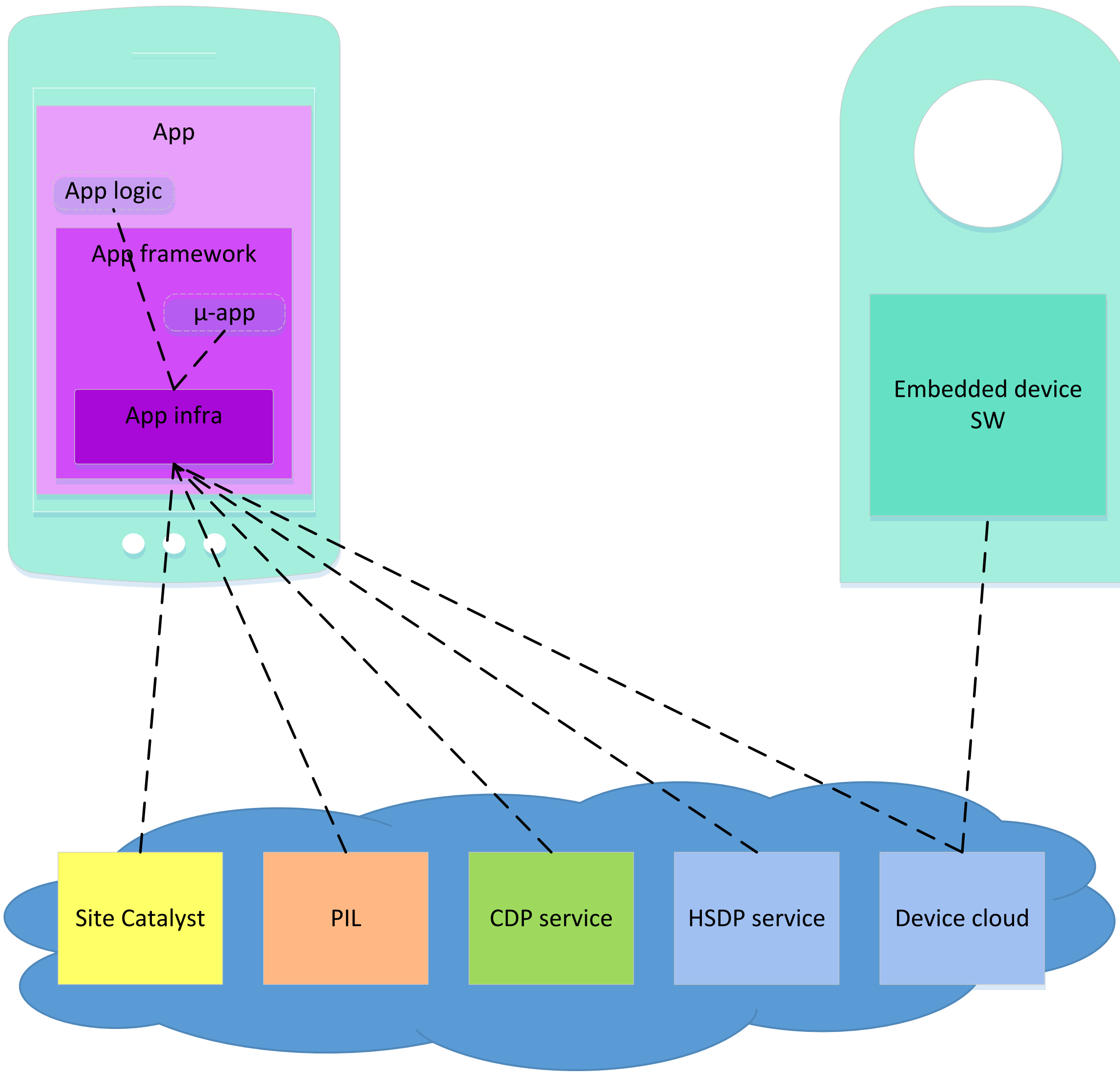

Figure 1 App Infra system context diagram

App Infra is a library providing a collection of modules to the app in which it is embedded. The App Infra modules are also used by the micro apps, which are part of the app.
App Infra assists in connecting to cloud services, but also to provide standardized methods for example for logging, or provide internationalization primitives to assist in optimizing the UI for the consumer's locale. The following diagram zooms in a bit more on App Infra and its relation with the app and OS:

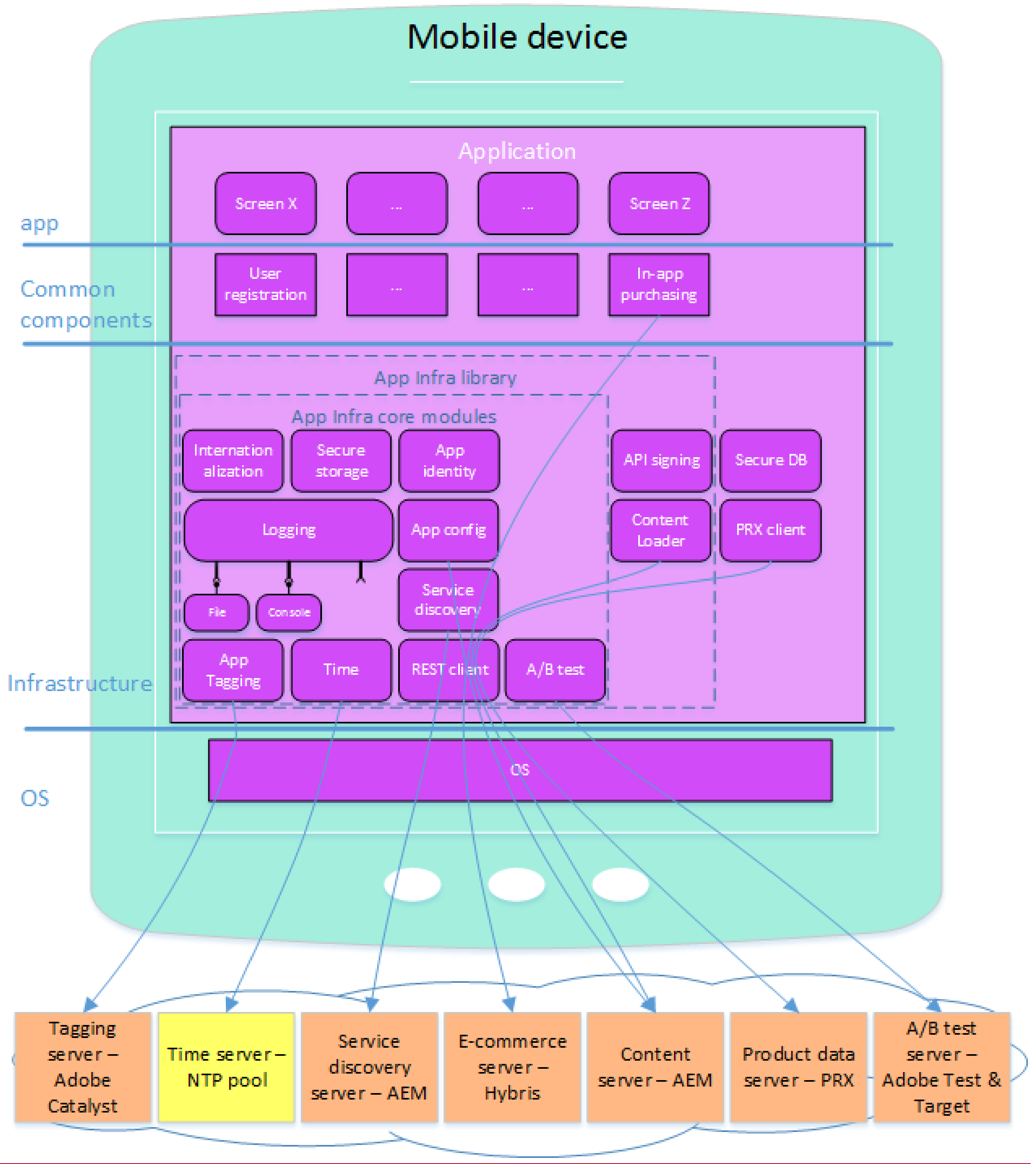

Figure 2 App Infra in app context diagram

3 Software Architecture Design

3.1 Top-level software architecture design

App Infra provides a range of modules that are the basis for any mobile application. App Infra is integrated as one single library in the application. App Infra in itself is not an operational entity but only exists in the context of an application. App Infra ensures that all modules provided by App Infra are linked to each other where needed. All modules together are exposed to the application through a single set of well-defined interfaces. App Infra is not only used by the application but also by all common components that are integrated in the application.
As such App Infra can be seen as a basic layer of functionality in the SW stack that is positioned somewhere between the app and the device's OS. App Infra is not designed to abstract the operating system; App Infra provides additional functionality on top of the operating system making use of the operating system.
Some of the App Infra depend on cloud servers to provide the required functionality. In those cases, App Infra abstracts the cloud server such that the users of App Infra are not directly exposed to the typical problems of remote services.

3.1.1 App Infra injection

App Infra is provided as an external software package (distributed via managers like Artifactory or CocoaPods). An application has to ingest App Infra as an external dependency.
It is the responsibility of the app to create one single instance of App Infra. The modules provided by App Infra are to be used throughout the app and integrated common components, in order to have one consistent view on data and way of working. Therefore, components integrated into the application need access to the application's App Infra instance and have a dependency on App Infra. To achieve this, App Infra is provided by the app to the components through dependency injection. The components do not create an instance of App Infra themselves but expect the application to provide a reference to App Infra.
App Infra is injected as a whole instead of module-by-module to reduce code complexity. It will be the component's responsibility to maintain the injected reference.
This concept is depicted in Figure 3.

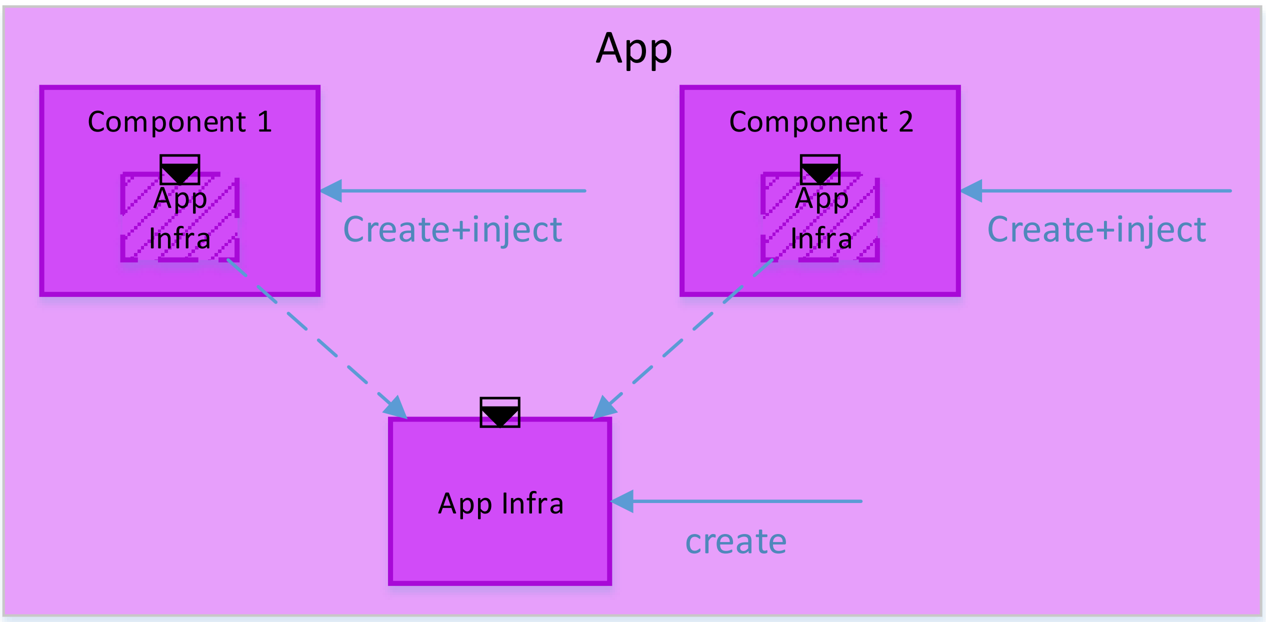

Figure 3 App Infra dependency injection

3.1.2 App Infra builder

To enable an application developer to create his own implementation for specific App Infra modules and have all components integrated in the app use that alternative module implementation; App Infra supports a builder pattern. By the use of the builder pattern, it is possible to create an instance of App Infra with alternative module implementations that overwrite one or more of the default module implementations.
The most common use case for providing alternative implementations is for testing purposes where a (component test-) app wants to test its functionality in isolation without having to implicitly test the App Infra implementation or any cloud services abstracted by App Infra. In such a case, the app developer can create an App Infra instance with dummy implementations.
Another use case for implementation replacement is to provide the ability to maintain compatibility with another cloud back-end (version).

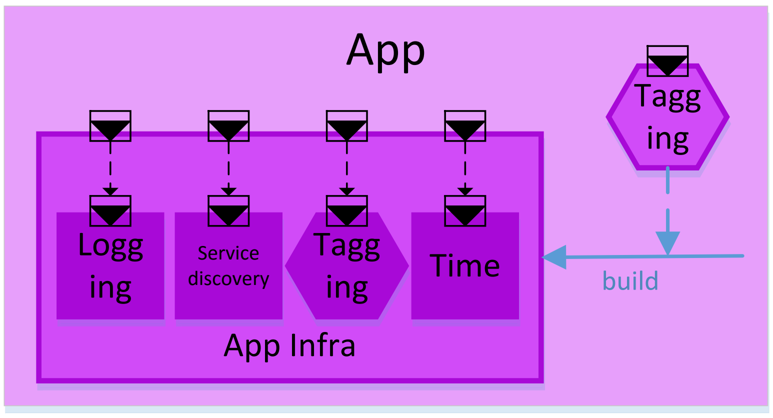

Figure 4 App Infra builder pattern

3.1.3 App Infra interfaces

To enable the dependency injection and builder patterns, App Infra fully relies on well-defined API interfaces. Any API call to App Infra or to any of the modules collected by App Infra shall be part of the abstract interface (in objective-C/Swift this concept is called protocols). By strictly adhering to the use of interfaces, implementations of App Infra (or its modules) can easily be exchanged without the application nor integrated components being aware of the alternative implementation and having to be recompiled.
Strict adherence to using these well-defined interfaces not only applies to the components and the app using the App Infra modules, but also in between the modules of App Infra. Modules within App Infra cannot make use of private/hidden interfaces of their peers.
This concept is strengthened by applying the separation of concern principle. Meaning a module is responsible to deliver all functionality related to that module. Other modules will fully depend on functionality provided by the responsible module. For example, Secure Storage is responsible to store data securely, other modules shall not directly store data in the database used by Secure Storage.

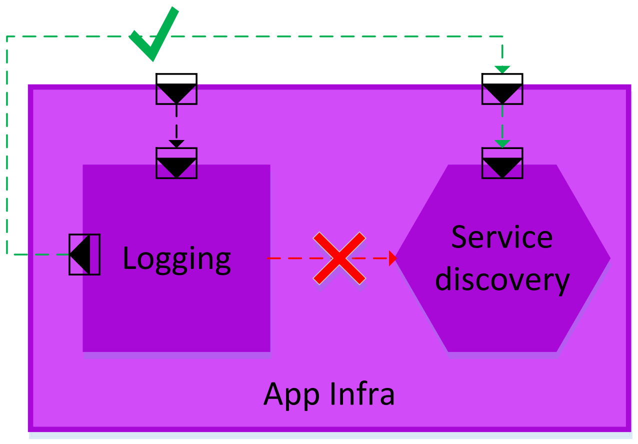

Figure 5 App Infra interface dependency

The encompassing App Infra object is not aware of the detail interfaces of the modules it holds, it only provides access to an object that implements a specific module's interface.

3.1.4 App Infra internal dependencies

Although the modules in App Infra appear to be largely independent of each other, they nevertheless depend on the services provided by their peer modules. The following diagram provides a quick overview of the module dependencies within App Infra.

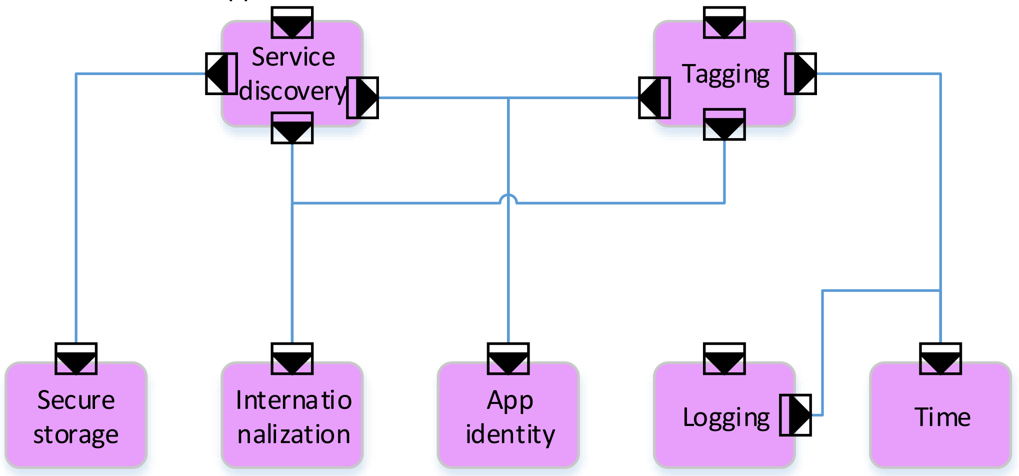

Figure 6 App Infra internal dependencies

As all modules depend on logging, this dependency is not made explicit.
Dependencies are controlled by mandating that no module may make use of functionality provided by any of its peers during its creation. App Infra will create instances for all modules in its builder in one atomic action. Only after completion of the builder process App Infra becomes accessible. Thus, it is ensured that the depended module is available when another module addresses it.
The diagram shows that Logging depends on Time, while Time depends on Logging (not show); this leads to a circular dependency. Commonly this should be avoided. This specific case can unfortunately not be prevented; it is resolved through manually preventing the circle from being closed.

3.1.5 App Infra initialization

At initialization, App Infra ensures that for all modules there is an instance created that provides the defined interface. Either it is an instance provided by the app through the builder pattern, or a default implementation as included in the App Infra library. This way, no exception handling is required in App Infra when providing access to its modules. To limit memory consumption and initialization time, the constructor of the App Infra modules is kept limited to the bare minimum. Second reason why there cannot be very complex logic in the constructor is that there is no guarantee that any of the other modules of App Infra have already been created and initialized, so during the initialization of App Infra internal dependencies cannot be resolved nor can the modules be used.

3.1.6 App Infra stand-alone modules

Not all modules provided by the App Infra library warrant being placed in the top-level App Infra component. Main reasons why these modules are not place in the top-level component are:

- Multiple instances may be instantiated in the context of a proposition app.
- They are created by the proposition and therefore are proposition specific; hence, App Infra cannot guarantee their availability.

For these reasons, the distribution of instances of these modules via the App Infra core component is not a good pattern. The App Infra library only provides the implementation but does not instantiate these modules. It is up to the proposition (or possibly other CoCos) to create them and separately inject them when required.

3.2 Software detailed design App Infra library

This document provides an overview of the functionality of the various modules of the App Infra component, the interfaces provided by every module and the required interfaces for every module. Most of the required interfaces are provided by other modules in App Infra, in that case these dependencies are resolved internal to App Infra. Interfaces that cannot be resolved internal to the App Infra component are to be resolved at app level.
This document does not describe the detailed design of the modules; please refer to [AppInfraDesignAndroid] and [AppInfraDesigniOS].
For interface documentation of App Infra, please refer to [AppInfraInterfaceAndroid] and [AppInfraInterfaceiOS].
The detailed requirement specification for App Infra is documented in [AppInfraSWReqSpec].
This section will indicate the interfaces delivered by App Infra and its various modules.

3.2.1 App Infra

Introduction:
App Infra is a simple collection object that mainly gathers all App Infra modules into one object. After creation, the App Infra instance cannot be modified.

Main responsibility:
The App Infra object maintains information that is required by all modules inside App Infra, in the case of Android for example this can be the activity context.
All log output generated by App Infra modules is logged under the App Infra component ID. To facilitate this, the App Infra object contains a logging wrapper instance.
Every module in App Infra is represented by an object implementing the defined interface. App Infra provides an API to obtain a reference to each object.
App Infra provides a builder that must be used to create an App Infra instance. The builder has the option to provide alternative module instances. For all modules where no alternative is delivered, the builder creates an instance with the default App Infra implementation. At completion, the builder delivers an App Infra instance.

Foreseen interfaces (dependencies):
Provides:

- App Infra modules access interface
- Build factory making it possible to provide alternative implementations for the modules.

Requires:
N/A.

3.2.1.1 Secure storage module

Introduction:
The secure storage module provides in encrypting and decrypting data using securely managed keys.

Main responsibility:
Secure encryption and decryption of application data. The application is responsible for storing the data.
The module provides functions for:

- Persistently store a key-value pair where the value is encrypted using an app instance encryption key that is managed in an OS specific secure key store.
- Retrieve and decrypt previously stored key-value pair based on a given key.
- Delete a key-value pair based on a given key.
- Encrypt/decrypt provided data and return the result (without storing).
- Create and securely manage encryption keys.

Note, secure storage does not provide any key-indexing, listing, or iteration modules.

Foreseen interfaces (dependencies):
Provides:

- Secure Storage interface: store/retrieve/delete key-value pairs

Requires:
N/A.

Data handling:
Storage:

- The master key ~~used to encrypt/decrypt the data~~ is stored in the device's secure key store.
- Per key-value pair an encryption key is created.
- The encrypted value of each key-value pair~~s~~ is~~are~~ stored in the OS's user app preferences storage.
- Encryption keys are securely wrapped with the master key and persisted in the OS's user app preferences storage

Transmission:
N/A.
Privacy:

- This module can be used to store privacy sensitive data.

3.2.1.2 Service discovery module

Introduction:
The service discovery module provides a soft coupling between application and the cloud services they use. Cloud services can be relocated over time. To compensate for these effects, the service discovery module maintains a list of base URLs for the various cloud services that are used by the application. The location of the cloud services is maintained in the backend and communicated to the app via Service discovery. It is the intention that the app and common components do not hard code any cloud service URL, but instead retrieve the URL from service discovery using a unique identifier for that service.

This list of base URLs is obtained from a cloud service at a hard coded location. The URLs for the services are influenced by the identity and state of the application, but also consumer specific parameters like home country and locale. So even with the same local and home country, different applications may use different URLs for exactly the same cloud service. This fully depends on the configuration of the service discovery server.
Next to the URL, service discovery also provides the recommended locale to be used for that specific service.
To facilitate easy and central configuration of all platform specific URLs, the service discovery module provides the ability to download two sets of URLs. One set is intended for all platform URLs, while the other set is intended to contain all proposition specific URLs. This enables the platform to centrally manage the URLs used by the platform components for all propositions. At the same time the amount of effort required by propositions to configure service discovery is significantly reduced. In case of conflicts between the two URL sets, the service discovery module will give preference to the proposition set.
The AEM service is used to host the service discovery URL sets.

Main responsibility:

- Provide locale and home country specific URL for a given cloud service
- Determine and maintain home country
- Synchronize URL configuration with a cloud service.

Foreseen interfaces (dependencies):
Provides:

- Service discovery interface: obtain URL+locale for specific service, get/set home country

Requires:

- Secure storage: to store home country
- App identity: identify application
- Internationalization: current UI locale plus fallback languages
- REST client: download service discovery information from server

Data handling:
Storage:

- Downloaded URLs are only cached in memory.
- User's home country is stored using secure storage module's key-value API.

Transmission:

- All communication is via HTTPS.

Privacy:

- The user's locale (maintained by OS) and home country are exposed to the service discovery server as query parameter.
- The service discovery server might store the home country and requesting client IP address.

3.2.1.3 App tagging module

Introduction:
The app tagging module facilitates the tracking of consumer behavior within the app UI. Actions and page changes triggered by the consumer – click stream – can be tracked and are sent to the server infrastructure for remote analysis.

Main responsibility:
The app tagging module receives page and action tracking requests from the application, adds common data parameters to the request and forwards the request to the cloud. Additionally the application can add global data parameters or instance level data parameters that will be added to each request.
The module provides for:

- Ensuring app global data parameters (key/value pairs) are added to each tracking request.

- Ensuring that the component performing a tag request can be identified in the tracking data.
- Tracking a given consumer action; the created tag is augmented by a set of data parameters provided in the call. Further, the component local parameters and global parameters are added to the tag.
- Tracking a consumer page transition; the new page is provided in the call, the previous page is maintained by the module. The created tag is augmented by a set of data parameters provided in the call. Further, the component local parameters and global parameters are added to the tag.

Foreseen interfaces (dependencies):
Provides:

- App tagging interface: perform tag requests

Requires:

- Internationalization: provides app UI locale
- Time: for UTC and local time
- App identity: app name, ID, version

Detailed design:
To make it convenient for component to perform tag requests and have their component ID and version included in every tag request they do, tagging provides a simple class that wraps around the base tagging implementation providing exactly the same APIs. The component can set its ID and version in that wrapper class, and every tag request the component makes via that wrapper class will automatically be extended with the component's ID and version.

Data handling:
Storage:

- Incoming tag requests while the device is offline is cached locally on the device by the Adobe SDK. Local cache is not additionally encrypted.
- User acceptance state is stored by the Adobe SDK.
- User consent for privacy sensitive data is stored using secure storage module's key-value API.

Transmission:

- In production state, all communication to tagging server is via HTTPS.
- Data is send to the Adobe Analytics server environment.

Privacy:

- The module provides an API to set tagging acceptance, until consent given no data is sent to the server. If no acceptance is given, no tagging requests are maintained. If acceptance state is unknown, tag requests are cached locally until state is clear.
- The module provides a method to define a filter for removing privacy sensitive until user consent has been given.
- App instance and time zone information is exposed to the tagging server.

3.2.1.4 Logging module

Introduction:
The logging module provides for logging actions in the SW of the application and its components. The log lines can be send to one or more sinks. Per sink, the logging can be filtered for level and component.

Main responsibility:
The logging module is responsible for aggregating log actions. The log output is formatted such that it can be parsed by external systems for automatic analysis.
Background threads are used to output the log lines to one or more destinations. Typical destinations:

- Console
- Local file, the maximum amount of local log data can be configured

- Web

A request for a log entry will have to specify the following parameters:

- Log level
  Severity of the event: error, warning, info, debug, verbose.
- Component ID
  Id of the common component generating the log statement.
- Event
  Agreed string identifying the specific event being logged.
- Comment
  Human readable string providing additional explanation, it shall not contain information required for automated analysis.
- Data parameters
  Key/value pairs providing more detailed information.

Foreseen interfaces (dependencies):
Provides:

- Logging interface: logging APIs

Requires:

- Time: UTC time

Detailed Design:
The logging module consists out of two layers:

- Distribution: queues all incoming log requests and distributes them in the background over all connected output classes.
- Zero or more output classes: filters the log requests per configured, formats the accepted log requests to a single log string, and outputs the string to a specific destination. Typical destinations: console, local file, cloud.

To make it convenient for component to perform log requests and have their component ID and version included in every log request they do, tagging provides a simple class that wraps around the base logging implementation providing exactly the same APIs. The component can set its ID and version in that wrapper class, and every log request the component makes via that wrapper class will automatically be extended with the component's ID and version.

Data handling:
Storage:

- Local file logging stores the files in app private storage without additional encryption.
- App shall disable file and console logging when released.

Transmission:
N/A.
Privacy:

- File and console logging shall be disabled at app release.

3.2.1.5 Time module

Introduction:
Commonly the device's local time is synchronized with an accurate time source, for example GSM network or GPS. However, the consumer may have setup the device to use a manually configured time. In order to correlate output data of the device with data of other devices (e.g. tags, or logs) an accurate UTC time is needed.

Main responsibility:
The time synchronization module obtains an accurate UTC time from an independent source. Multiple sources have been identified:

- HSDP Device cloud
- Generic NTP

As not every proposition needs a dependency on the device cloud, generic NTP is selected. No mechanism is foreseen switch to device cloud.
This time is synchronized at regular intervals (e.g. application start, local time modification, every 24hours).
The module provides the following information:

- UTC time: accurate time obtained from a web server.
- Local time (as provided by OS)
- Local time zone

Foreseen interfaces (dependencies):
Provides:

- Time interface: provides local and UTC time plus time zone

Requires:
N/A.

Detailed Design:
The module does not maintain and internal real time clock. Rather, the module determines a time delta between the device local time and the independent accurate time source when the time is synchronized. When the current UTC time is requested, the UTC time is derived from the device local time plus the calculated delta. The module automatically synchronizes the time at first instantiation, when a large local time change is detected, and every 24hours.

Data handling:
Storage:

- Time offset between device local and server UTC time is stored in OS's user app preferences storage.

Transmission:

- NTP protocol is used to retrieve server UTC time.

Privacy:

- This module does not expose any privacy sensitive data.

3.2.1.6 Internationalization module

Introduction:
This module provides primitives that assist in providing a UI that matches the locale as expected by the consumer.

Main responsibility:
The internationalization module currently only provides an API to retrieve the locale currently used for the app UI.

Foreseen interfaces (dependencies):
Provides:

- Internationalization interface

Requires:
N/A.

Data handling:
Storage:
N/A.
Transmission:
N/A.

Privacy:
- This module does not expose any privacy sensitive data.

3.2.1.7 App identity module

Introduction:
An application is identified by various parameters; this module maintains all these parameters. The parameters are used amongst others for tagging and service discovery but are also shown in the UI.

Main responsibility:
This module obtains the parameters from the build environment and makes them available to other modules in the application. The following parameters characterizing the application are provided:
- App name
- App version
- App ID
- App status: e.g. development, test, production

Foreseen interfaces (dependencies):
Provides:
- App identity interface: provide app information

Requires:
- The parameters maintained in this interface are provided by the build process; hence, this module has a dependency on the build environment.

Data handling:
Storage:
- App identity information is stored in static App config asset file.

Transmission:
N/A.
Privacy:
- This module does not expose any privacy sensitive data.

3.2.1.8 App config module

Introduction:
An application and the included common components can have multiple configuration parameters that determine their behavior. The app config module centrally maintains these configuration parameters. The majority of the configuration is compile time defined and provided in a static configuration file. Some parameters are determined run time. The app config module provides read and write access to the configuration parameters, both static and runtime additions/overwrite, on a key-value pair basis. Runtime changes are stored in a secure way as they may contain sensitive information.
The module has the ability to download configuration updates from a cloud service. The URL for the configuration update is retrieved through service discovery based on the user’s home country. The AEM service is used to host the configuration update files. Downloaded configuration updates have priority over static configurations but are overruled by runtime configurations.
Neither configuration downloads nor runtime configurations are irreversible. Removal of runtime changes will expose the lower prioritized downloaded or static configuration values; similarly removal of downloaded configuration values will expose the static configuration value again.

Main responsibility:
- This module maintains and exposes configuration parameters for all components included in an application. Configuration parameters are identified by:
    - Configuration group name

  - Configuration key name
- Download configuration updates from a cloud service.

Foreseen interfaces (dependencies):
Provides:
- App config interface: provide read/write access to configuration parameters.

Requires:
- Secure storage: encrypt run time configuration settings

Data handling:
Storage:
- Static configuration is stored in asset file.
- Run-time changes are stored using secure storage module's key-value pair API.

Transmission:
- ~~N/A.~~The home country might be send to the AEM server in order to obtain country based configuration updates (depends on level of detail in URL configurations).
- The AEM server may store the configuration download request in combination with the client IP address.

Privacy:
- This module can be used to store privacy sensitive data.
- Depending on the URL configuration for configuration download the home country might be exposed.

3.2.1.9 REST client module

Introduction:
An application has a strong dependency on cloud services. These cloud services are accessed using a REST-based API. The REST client module provides a REST-based interface to channel all communication to cloud services.

Main responsibility:
This module provides for a communication path to cloud services. The module ensures that all communication goes through a properly secured channel. For servers that require authorization, the REST client module has the ability to include an authorization token provider in the request. The REST client module also include caching capabilities adhering to the caching guidance provided by the server in order to: reduce network bandwidth, improve responsiveness, and assist data availability when offline. To ensure security for all data, the cache will be encrypted.
The REST client module will strongly depend on well-known external libraries. There is little added value in providing a CDP2 proprietary interface on top of the selected library as it only adds effort and risk for bugs. Further, it reduces familiarity with the API by its users, as most developers already have used the selected library; and the library itself is already well documented and many external help is available. Hence, the REST client module API strongly follows that of the selected libraries (differing per platform).
The REST client module mainly tunes the selected libraries with respect to the main responsibilities and features: security, authorization, and cache encryption.

Foreseen interfaces (dependencies):
Provides:
- REST client interface: perform REST calls, use authorization token provider.

Requires:
- Service Discovery: optional, used when the REST request directly refers to a service ID.
- Secure storage: encrypting the cache.

Data handling:
Storage:
- The REST client maintains a cache of hashed URL and encrypted response data in app private storage.
- Secure storage module's encryption functions are used to encrypt/decrypt response data

Transmission:
- All communication is forced to use HTTPS.

Privacy:
- This module in itself does not expose any privacy sensitive data.

3.2.1.10 A/B test module

Introduction:
Mobile apps only achieve their goal through providing a great user experience. Although one can extensively investigate how to obtain that experience, several viable alternatives can be defined. The best way to determine the best option is to test all of them live in the field and select the one with the best results.
This means implementing all alternatives and randomly select one or the other over the complete population of app users. The selection which user gets which experience is controlled by a cloud service based on controlling factors like country or locale. The selection is cloud controlled such that distribution percentages and timeframe in which the test is executed can be controlled.
The A/B test feature selects which experience to show to the user. App tagging is used to feedback the results from the experience to the cloud in order to measure which experience obtained the best results.

Main responsibility:
The A/B test module obtains the test selection value for all A/B tests embedded in the app. The A/B test module ensures that the user always obtains a consistent experience even if the cloud server is not available to provide the test selection information. The module obtains all test experience selections before the UI flow needs to decide which experience to render to prevent delaying the UI experience waiting for the server to provide the experience selection.
The measurement feedback loop is closed through app tagging:
1. A/B test module retrieves which experience is to be shown to the consumer from the test control server (Adobe Test & Target).
2. The CoCo or application uses that value to render the appropriate UI.
3. By the use of app tagging the UI usage by the consumer is returned to the tagging server.
4. Every related tag is supplemented with the test identity and used experience value to assist the analysis of the experience differences.

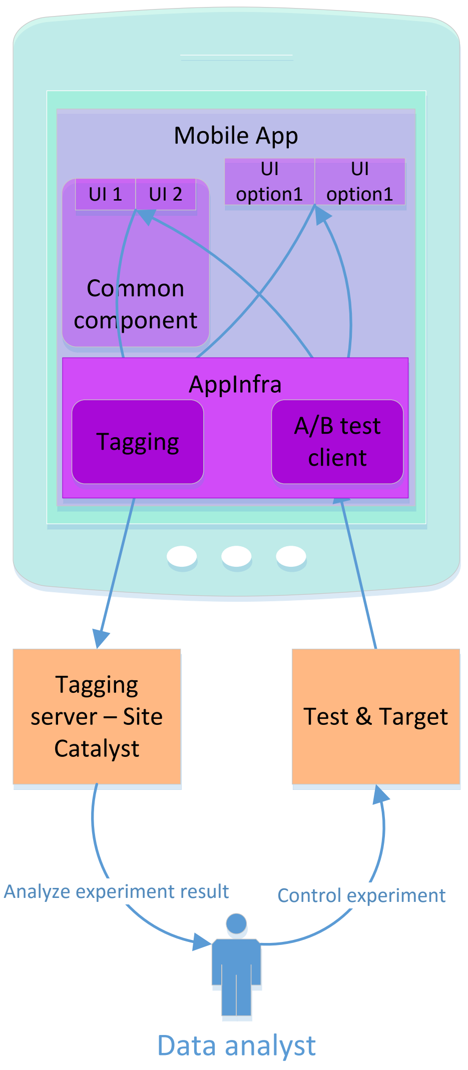

Figure 7 A/B test data flow

Foreseen interfaces (dependencies):
Provides:

- Configuration: option to define the list of tests embedded in the app
- A/B test interface: refresh all test experience values from server, get test experience value

Requires:

- App tagging: A/B test client uses same Adobe SDK as used for App tagging.

Data handling:
Storage:

- The downloaded test experience values are stored in OS's user app preferences storage.

Transmission:

- In production state, all communication to server is via HTTPS.
- Experience values are retrieved from the Adobe Test & Target server environment.

- A one-time randomly generated application instance ID (identifying that single app installation) is send to the server for every requested.
- The application instance ID is maintained on the server to ensure the server can always return the same experience value to that application instance.

Privacy:

- This module in itself does not expose any privacy sensitive data.

3.2.2 Content loader module

Introduction:
The content loader module is not an integral part of the App Infra core but must be instantiated separately.
A large set of information that can be shown to the user resides on a cloud server. This information is not embedded in the app itself because it has a different lifecycle than the app itself (created and modified after the app is released) and due to its size including multi locale versions will have a significant impact on the size of the app.
The method to download this information from a cloud server is generalized even though the data model of each content element may differ per type of content. Multiple content elements can be retrieved from the server in one call. It is not possible to download all content in one call, as that will have a negative impact on the server response time; therefore, a pagination concept is used.
Typical content are informational articles: ‘how to shave a specific beard style’, ‘how to feed your baby’, ‘frequently asked questions’, etc. Content can be tagged to split it into categories (breast-feeding, bottle feeding, etc.).
To achieve a good user experience, the content must be presented immediately, waiting on a server to provide the content would hamper the performance. For that reason, the content is to be cached locally on the device.
Content elements can be a combination of human readable text, properties and images. The content loader will only cache the text and metadata; due to their size, images and such are not cached and must be downloaded on the fly.
An app can instantiate zero or more content loaders.

Main responsibility:
The content loader downloads through a page-based interface content metadata from a cloud server. It maintains a timestamp of the cached content such that it can signal to the app that a refresh is recommended. During a download, the content loader will create/update/delete the cached content using the newly obtained content. The location of the content is obtained via Service Discovery. Hence, the content is locale dependent. The app is given access to the content based on the content ID, content tag.
Because the content loader must be flexible enough to be used for different types of content each using a different data model, the content loader is designed as a template class. The app has to provide the data model class that is maintained by the content loader. The data class must implement a minimum interface such that the content loader can feed the retrieved data into the class and retrieve the minimum required data to fulfill its interface requirements.

Foreseen interfaces (dependencies):
Provides:

- Content loader interface: create instance using service ID, max age, content type; refresh content; retrieve content

Requires:

- Service Discovery: resolve service ID to URL
- REST client: download content.

Data handling:
Storage:

- The downloaded content is stored in app private storage.

Transmission:

- In production state, all communication to server is via HTTPS.

Privacy:

- This module in itself does not expose any privacy sensitive data.

3.2.3 API signing module

Introduction:
To ensure only authenticated clients can make use of our cloud services, the APIs are protected. Commonly this is achieved using oAuth2, where the user needs to login to the cloud environment and the resulting token is used to authenticate the app to the different cloud services.
However, some service must be available prior to the user having logged in or do not depend on a user identity. To ensure that only authenticated clients make use of the server APIs, the server must be able to determine whether the incoming request is coming from a known client. This is achieved by including a secret in the client and use that secret to sign the API request. This secret is to be well protected to prevent it being easily abused.

Main responsibility:
The API signing module embeds a client secret through a white box algorithm and provides an API to sign REST requests. The application only has to provide a key to identify the type of app, this security level of this key is low as the secret information is part of the API signing module. Thereby the security restrictions imposed on the app developer are less strict.
The API signing module supports multiple signing strategies, each based on HMACSha256.

Foreseen interfaces (dependencies):
Provides:

- API signing interface: set app keys, calculate signature for given REST request.

Requires:
N/A.

Data handling:
Storage:

- The key pair provided by the app is only maintained in RAM.
- The key secret is embedded in the white box signing algorithm of this module.

Transmission:
N/A.
Privacy:

- This module in itself does not expose any privacy sensitive data.

3.3 Software detailed design PRX client

The detailed requirement specification for PRX client are documented in [AppInfraSWReqSpec].
This section will indicate the interfaces delivered by PRX client.

3.3.1 PRX client

Introduction:
The PRX client provides an abstracted interface of the product information stored on the PRX cloud service.
The PRX client is provided as a separate component in order to have no data model dependencies of various systems in the App Infra library.

Main responsibility:
The PRX client retrieves product asset, summary, and support information from the PRX cloud service given a sector, catalog and product CTN.

Foreseen interfaces (dependencies):
Provides:
- APIs to retrieve product asset, summary, and support information.

Requires:
- AppInfra REST client: retrieve data from PRX cloud service.
- AppInfra service discovery: locate PRX cloud service based on home country.

Data handling:
Storage:
N/A.
Transmission:
- All communication is via HTTPS.

Privacy:
- The product CTN is sent to the PRX server.
- The PRX server might store the product CTN and requesting client IP address.

3.4 Software detailed design Secure DB

The detailed requirement specification for Secure DB are documented in [AppInfraSWReqSpec].
This section will indicate the interfaces delivered by Secure DB.

3.4.1 Secure DB

Introduction:
Secure storage provides ability to store simple key-value pairs where the value is encrypted. For larger amounts of structured data, a relational database (SQLite compatible) is more convenient. Secure DB provides an encrypted relational database including an object modeling layer. Secure DB uses Secure storage to manage the password used to encrypt the database.
Secure DB is delivered as a stand-alone component that works in combination with AppInfra. It is created as a stand-alone component as it may not be required by all propositions while it does have a significant footprint.
Secure DB is currently only delivered for the Android platform.

Main responsibility:
The Secure DB component provides a relational database with object model based interface. It will encrypt the data before persistently storing the data on the device.

Foreseen interfaces (dependencies):
Provides:
- Object based development framework for database access.
- API to instantiate an encrypted database given an encryption key ID.

Requires:
- App Infra Secure storage: securely creating and managing encryption keys.

Data handling:
Storage:
- The data stored in the database is encrypted using AES.
- The key used to encrypt the data is maintained in secure storage.

Transmission:

N/A.
Privacy:

- This module in itself does not expose any privacy sensitive data.

~~3.3~~3.5 Third-party Software, Software Of Unknown Provenance (SOUP)

~~3.3.1~~3.5.1 CocoaLumberjack

iOS specific: this library is used internal to the logging module as basis framework for filtering, distributing, and outputting log requests.

~~3.3.2~~3.5.2 Adobe Mobile SDK

The Adobe SDK is used for communicating the App tagging events to the Catalyst cloud back-end.

~~3.3.3~~3.5.3 Volley

Android specific: the Google Volley library is used for managing the REST client request queue and cache.

~~3.3.4~~3.5.4 GSON

Android specific: the Google GSON library is used to convert JSON data structures received by the content loader module to data model class.

~~3.3.5~~3.5.5 NHNetworkTime

iOS specific: the NHNetworkTime library is used by the Time module to sync the time with NTP servers.

~~3.3.6~~3.5.6 CocoaAsyncSocket

iOS specific: the CocoaAsyncSocket library is used by NHNetworkTime to setup a plain socket connection to a NTP server.

~~3.3.7~~3.5.7 AFNetworking

iOS specific: the AFNetworking library is used for managing the REST client request queue and cache.

3.5.8 SQLCipher

Android specific: the SQLCipher library is used to provide an encrypted SQLite compatible database for Secure DB.

3.5.9 ORMLite

Android specific: the ORMLite library is used to provide an object modeling framework on top of SQLCipher for Secure DB.

4 Algorithms

This SW component does not contain any algorithms.

5 Requirements Traceability

The next table shows the mapping of each requirement from the [SwRS] to the software module(s) identified in the software architecture that are responsible for the implementation of it.

Table 1 Requirements traceability cross reference to [SwRS]

Requirement	Software Module(s)
R-AI-1, R-AI-2, R-AI-3	3.2.1 App Infra
R-AI-SEC-*	3.2.1.1 Secure storage module
R-AI-SLM-*, R-AI-SD-*	3.2.1.2 Service discovery module
R-AI-AT-*	3.2.1.3 App tagging module
R-AI-LOG-*	3.2.1.4 Logging module
R-AI-T-*	3.2.1.5 Time module
R-AI-INT-*	3.2.1.6 Internationalization module
R-AI-AI-*	3.2.1.7 App identity module
R-AI-AC-*	3.2.1.8 App config module
R-AI-RC-*	3.2.1.9 REST client module
R-AI-AB-*	3.2.1.10 A/B test module
R-AI-CL-*	3.2.2 Content loader module
R-AI-AS-*	3.2.3 API signing module
R-AI-PRX-*	3.3.1 PRX client
R-AI-SDB-*	3.4.1 Secure DB