CDP

**App Infra Code Design Spécification**

**Authors : Ashok Kasthuri, Anurag Gautam**

**Version : 1.1.0**

**Date : 2016-07-19**

**Status : Proposed**

**Platforms: Android**

**Components covered: Secure storage, Tagging, Logging,**

**App Identity , Service Discovery & Time sync**

**Table of Contents**

1. Introduction 3

Purpose & Scope

Target Audiance

1. Secure Storage 4

3. Tagging 8

4. Logging 10

5. App Identity 15

6. Service Discovery 17

**Introduction:**

**Purpose & Scope**

This document basically contains the information to help the verticals to understand code design flow of each feature of Chassis (AppInfra).

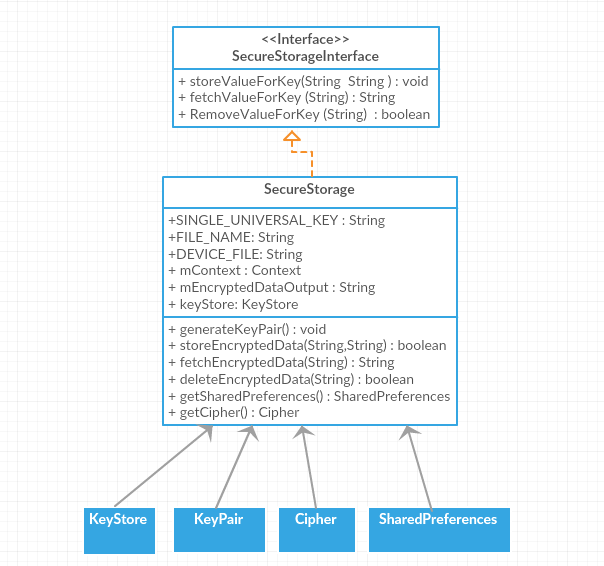
**Target Audiance:**

  All component and App developer.

**2. Secure Storage**

Secure Storage is used to store secret value in device storage with encrypted way using RSA. It uses key value pair concept to store data inside the apps.

1. Secure Storage Class Hierarchy



*SecureStorage class implements SecureStorageInterface interface and aggregates KeyStore, keyPair, Cipher & SharedPreferences classes.*

Initialization

*SecureStorageInterface ssInterface = AppInfraSingleton.getInstance().getSecureStorage();*

1. Threading

In present version all method call happens on main thread and all methos calls are synchronous so its thread safe.

1. Dependance

No third party has been used.

1. Open Source Library

NO

1. Memory managment

Android's Dalvik virtual machine performs routine garbage collection.

1. Storage

We are using Android keystore to store encryption/decryption key. We are using SharedPreferences for storing secured encrypted data. When user tries to store encrypted data, the alias name of key is searched in keystore. If key is not present(first run) then asymmetric RSA key pair is generated using RSA algorithm and stored in keystore. Thereafter stored key pair is retrived for each encryption and decryption.

When developer stores a value in device witha a user key or tag (storeValueForKey(userKey, value)), it is first encrypted using RSA public key and RSA algorithm further it is stored in device at SharedPreferences with user given key.

Later when developer fetch the stored value with user key or tag (fetchValueForKey), the encrypted data is fetched from SharedPreferences and decrypted with RSA private key further decrypted value is given back to developer.

1. Use Cases

Developer can perform three operations:

1. Store value in device

Developer can store value in encrypted format assigning a key name to value.

SecureStorageError ssError = new SecureStorageError();

**boolean void** storeValueForKey(String userKey,String valueToBeEncrypted);

**eg boolean result=**storeValueForKey(“myPasswordKey“,“abc@123“, ssError);

1. Fetch value from device

Developer can fetch stored value from device using assigned key name.

SecureStorageError ssError = new SecureStorageError();

**public** String fetchValueForKey(String userKey);

eg **boolean result=**fetchValueForKey(“myPasswordKey“, ssError);

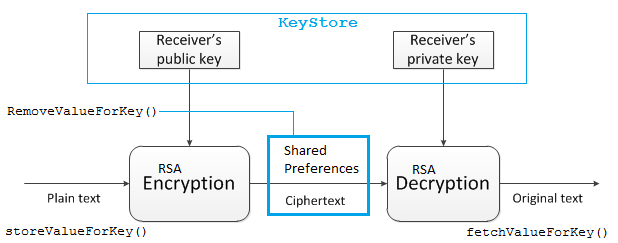
1. Delete value from device

Developer can delete stored value from device using assigned key name.

**public boolean** RemoveValueForKey(String userKey);

eg RemoveValueForKey(“myPasswordKey“);

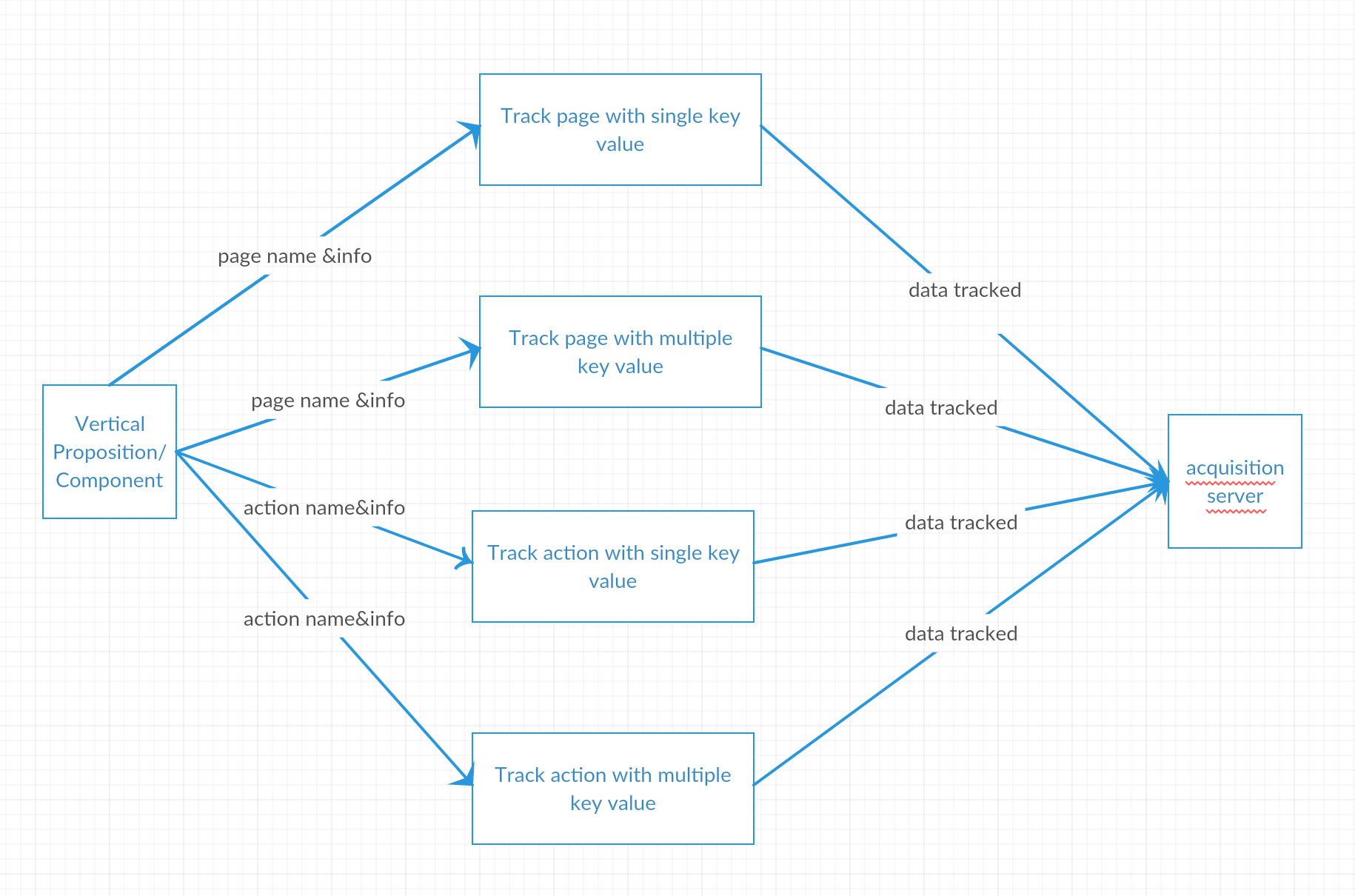
1. Data Flow



1. **Tagging:**

Purpose & Scope:

As part of Connected Digital Proposition, App Tagging is a component that can be reused across various applications. The subsequent sections provide the detail of the App Tagging as a re-usable component. it uses the Adobe library (https://marketing.adobe.com/developer/get-started/mobile/c-measuring-mobile-applications) to track and report analytics



Use Cases:

Use case: Track Page with key value

Used to track the pages along with additional data in the form of key values

Use case: Track Page with multiple key value

Used to track the pages along with additional data in the form of dictionary (multiple key values)

Use case: Track Event with key value

used to track button action events ,can be tracked with optional additional info parameters

Use case: Track Event with multiple key value

Used to track button action events along with additional data in the form of dictionary (multiple key values)

1. **Logging**

Purpose & Scope:

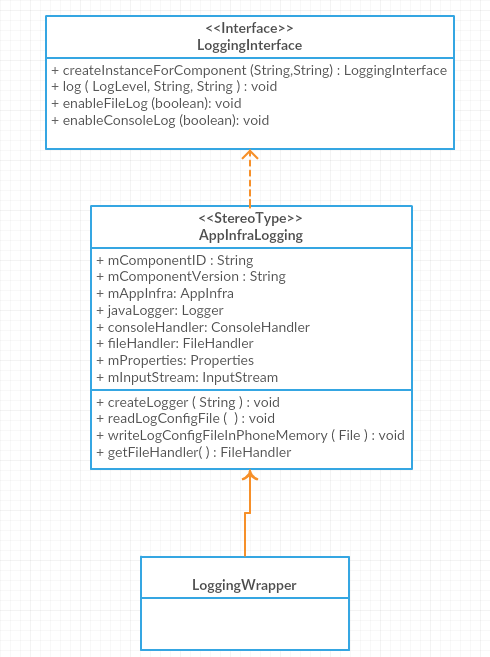
Logging will be used by various component and App Framework to log events and message in multiple sinks like console and file.

Logging provides filtering at log levels and component level.

Java native **Java Logger library** is used for logging logs.

Log can be done at five levels {VERBOSE, DEBUG, INFO, WARNING, ERROR};

Logging Class Hierarchy



*AppInfraLogging implements LoggingInterface and LoggingWrapper extends AppInfraLogging*

Prerequisite: Add logging.properties file from Documents\External to Application assets folder. Application can modifiy this file for console and file level/configuration/filter .



Threading:

Currently log works on main UI thread.

Dependence:

No third party library required. Native Java Logger used.

Open Source Library

NO

Memory managment

Android's Dalvik virtual machine performs routine garbage collection.

Storage:

Logs are stored under AppInfra Logs directory in device. Maximum five files(AppInfra.log.0 , AppInfra.log.1 .. AppInfra.log.4 ) are created each of 2 MB. Once fifth file reaches size limit, over writing starts on first file. These parameters are configurable.

Use Cases:

Developer can perform these operations:

1. Initialize log at component level

Prerequisite: Initialize AppInfra object in App framework and gets its reference by single class getInstance.Single App Infra object will be created in App Framework, and its instance can be accessed in app and libray.

*AppInfraInterface appInfra =AppInfraSingleton.getInstance();*

Then component will use this instance to create their Log class(i.e LoggingInterface ).

*LoggingInterface loggingInterface = appInfra.getLogging().createInstanceForComponent("philips.di.cl.appframework.UiKit", "3.1.0");*

***Note: Each component must give their root package name in createInstanceForComponent method****.*

1. Enable Console Log:

Developer can enable/disable console log for Android Console output:

*loggingInterface.enableConsoleLog(true);*

1. Enable File Log:

Developer can enable/disable console log for device file output:

*loggingInterface.enableFileLog(true);*

1. Write Logs:

Developer can write logs on console , file or both:

*loggingInterface.log (LoggingInterface.LogLevel.INFO,”Event”,message”);*

1. Filter Logs:

Developer can use logging.properties file to filter Logs based on:

a)Log level

{VERBOSE, DEBUG, INFO, WARNING, ERROR}

*java.util.logging.ConsoleHandler.level=FINE // all five log Levels*

*java.util.logging.FileHandler.level = INFO // INFO, WARNING & ERROR*

*java.util.logging.ConsoleHandler.level=OFF // no output*

b) Component Level

*philips.di.cl.appframework.UiKit.level=WARNING // only WARNING and ERROR log will output*

*philips.di.cl.appframework.UiKit.level=OFF // No log output for this component*

Logging.properties file will be read first time from App asset/ logging.properties file which is read only.After reading it a copy of logging.properties file will be written to device memory under AppInfra Logs directory (read/write).

Second time onwards logging.properties will be read from this device memory. If any runtime configuration/filter change is required, developer can do those configuration/filter change on this logging.properties under AppInfra Logs directory and restart the app to adapt change.

(This model works only for rooted device where device data folded is accesible)

1. **App Identity**

The App identity feature shall provide an API to get the app release status: development, test, acceptance, production.

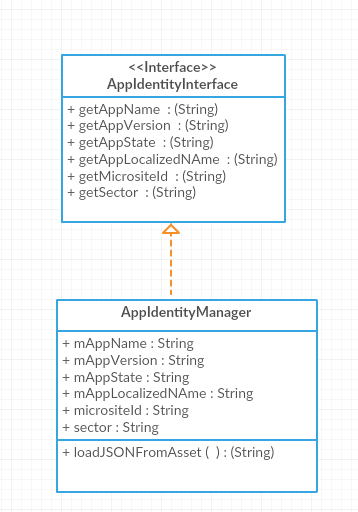
The App identity feature shall obtain the technical app name, app version and app release status automatically from the build application build process.

developer needs to be crerate appidentity.json and add microsite, sector & AppState key value:

*{  
 "micrositeId" : "12345",  
 "sector" : "B2C",  
 "AppState" : "DEVELOPMENT"  
}*

Remaining AppVersion and AppName will be written from gradle & AppLocalName will be written manifest file.

**App Identity Class Hierarchy**

****

**API**

*public String getAppName();*

Fetch technical App name.

*public String getAppVersion();*

Fetch App version

*public String getAppState();*

Fetch App state (development, test, acceptance, production)

*public String getAppLocalizedNAme();*

Fetch app localized commercial app name.

*public String getMicrositeId();*

Fetch micrositeID

*public String getSector();*

Fetch sector

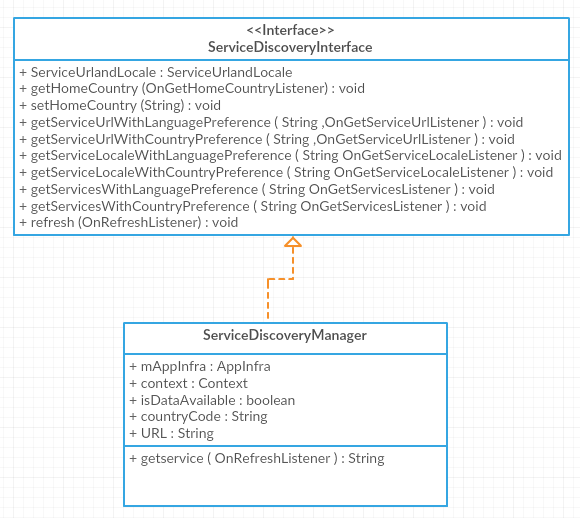
public String loadJSONFromAsset()

Fetch Appidentity.json file from asset folder and assigns key value pairs to class variables

**6. Service Discovery**

Service Discovery reduces the hard dependency between app and cloud services. The main idea is that the list of URLs that are to be used by an application is maintained server side, at the service discovery server. The app only has to download this list from **one single global location**, this list tells the app where all other cloud services can be found. It is the service discovery server's responsibility to ensure that the correct URLs are returned for the country and our language in which that app is being used. If cloud services are relocated, only the list at that service discovery server needs to be updated, no changes on app side are required

Service Discovery Class Hierarchy



**Service Discovery API (Android)**

1. void getHomeCountry(OnGetHomeCountryListener listener)

GetHomeCountry will get the country either from SIM or GEOIP. The country is saved in preferences. The listener, OnGetHomeCountryListener will get the results back.

1. void getServiceUrlWithLanguagePreference(String serviceId, OnGetServiceUrlListener listener)

getServiceUrlWithLanguagePreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServiceUrlListener will get the results back.

1. void getServiceUrlWithCountryPreference(String serviceId, OnGetServiceUrlListener listener)

getServiceUrlWithCountryPreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServiceUrlListener will get the results back.

1. void getServiceLocaleWithLanguagePreference(String serviceId, OnGetServiceLocaleListener listener)

getServiceLocaleWithLanguagePreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServiceLocaleListener will get the results back.

1. void getServiceLocaleWithCountryPreference(String serviceId, OnGetServiceLocaleListener listener)

getServiceLocaleWithCountryPreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServiceLocaleListener will get the results back.

1. void getServicesWithLanguagePreference(String serviceIds, OnGetServicesListener listener)

getServicesWithLanguagePreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServicesListener will get the results back.

1. void getServicesWithCountryPreference(String serviceIds, OnGetServicesListener listener);

getServicesWithCountryPreference will get the URL’s from the response filtering with given ServiceID. The listener, OnGetServicesListener will get the results back.

1. void refresh(OnRefreshListener listener)

The refresh to Webservice call happens here. And the results will get back to OnRefreshListener.

1. public String getservice(OnRefreshListener listener)

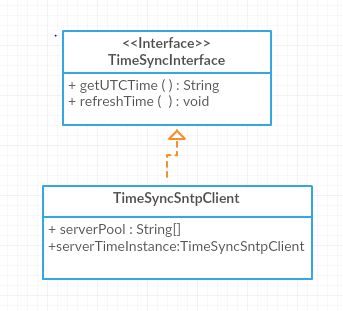
Call the service discovery hard coded single URL without(first run) country code and fetch the country code and save it in shared preference.

There after this service discovery hard coded single URL is called with ‚country‘ also as a parameter to get complete list of service urls

for given service.

**Time Sync**

The feature does not maintain and internal real time clock. Rather, the feature 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 feature automatically synchronizes the time at first instantiation, when a large local time change is detected, and every 24 hours



To fetch the NTP network time using SNTP client

**public** String getUTCTime();

To refresh NTP time.

Offset Time= NTP time – Device time

**public void** refreshTime();

Time sync is used by Logging and Tagging feature