**OWASP Seraphimdroid Documentation**

**Implementation**

According to new android navigation guidelines best way to provide navigation in an application is navigation drawer. The drawer uses fragments for each layout that is displayed on the screen. Following the above guideline access to each service is given through the fragments. So like any other application SeraphimDroid starts with a MainActivity.java file which displays the navigation drawers consisting of 8 main fragments + an Additional fragment for the Project Info , namely,

1. PermissionsScannerFragment.java

2. SettingsCheckFragment.java

3. BlockerFragment.java

4. ApplicationLockerFragment.java

5. ServiceLockerFragment.java

6. GeoFencingFragment.java

7. EducateFragment.java

8. SettingsFragment.java

(i) . AboutFragment.java

Implementation detail for each fragment is given below:

**Permission Scanner**

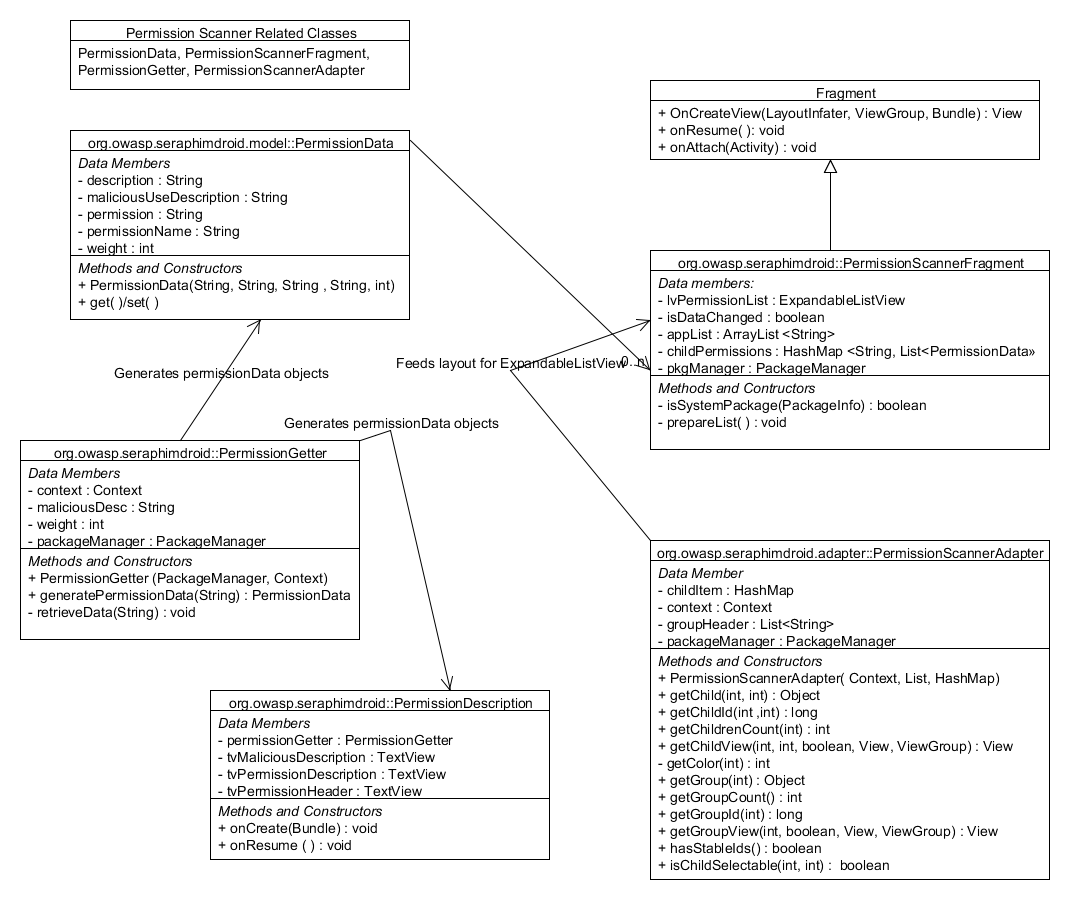
Permission scanner scans permission for all the installed application, for that, it need to get the list of all the installed application. **PackageManager** class provides the required method for that

List < ApplicationInfo> appList = getActivity().getPackageManager().packageManager.getInstalledApplication(PackageManager.GET\_META\_DATA);

appList stores the information of all the installed application, then for each item in the list permission are fetched and the details for these permissions are retrieved from the database using a custom class **PermissionGetter.** All this task is performed in a separate thread using class **AsyncTask**, which contains an object **svmModel**, which loads a pre-trained **SMO** Weka model. This instance is then used to predict the the nature of application, using it's permissions as a 0/1 feature vector. The Prediction accuracy of our current model is ~88%

**ExpandableListView** is used to display the installed applications and the child item for each item displays the permissions that is used by that application. With each installed application an indication is displayed to show the danger level for that app and user could uninstall the application by long pressing on the name.

**PermissionDescription** displays the details about each permission, the threat it poses, what access it provides to the application and how can it be used to damage your data or affect your privacy. It has been themed as a dialog box, because that suits best for displaying some information.



**Settings Checker**

Settings Checker scans the Device's settings, using the **Settings.Secure** API provided by Android. It can be used to fetch settings for different parameters, such as Settings.Secure.**ADB\_ENABLED** to check if USB Debugging is enabled, similarly Settings.Secure.**INSTALL\_NON\_MARKET\_APPS** to check if Application install from unknown sources is permitted.

Settings Checker also has a background service, implemented using Android's **Alarm Manager** class, which performs automated checks at regular time intervals. By default, the interval is set to 1 Day. It can be changed in the settings to a Week, Fortnight or a Month.

If the service scans and finds that any of the setting is not set to its optimal value, it fires a notification. Upon clicking this notification, the user is taken to the Settings Checker fragment, wherein he can fix the respective settings.

**Blocker logs**

The UI is a tabbed view for showing the logs for each Call, SMS and USSD that could be malicious. The tabbed layout is created using **TabHost** and **ViewPager** together in combination to achieve the required navigation flow and good UI design.

The broadcast receivers are used to keep check on the malicious activities, for example unaware message sending, premium calls, execution of harmful USSDs etc. These Broadcast receiver includes

1. CallRecepter

2. SMSRecepter

The **CallRecepter** class is used to intercept incoming and outgoing calls. Every call that is placed by any application is passed through this receiver and then it checks if the called number meets the requirement set by the user i.e. if the number is saved in users contact details or is not present in the user’s blacklist. User could access the preferences regarding call blocker in the settings. Any call not meeting the requirement is logged in the Call logs and is shown in the blocker logs, with the reason for the blocking.

The **SMSRecepter** class does pretty much the same but with messages. The receiver can only handle received messages, for outgoing messages another service is used because the Android mechanism for outgoing SMSs are different. In case of calls the calls could be blocked or cancelled but for messages its different and specially with outgoing messages. There is no way yet known which could be used to alter the content of outgoing messages, or better block them. The Incoming SMS are deemed malicious and is reported to the user, if the message content contains numbers that are not saved in the contact list of user.

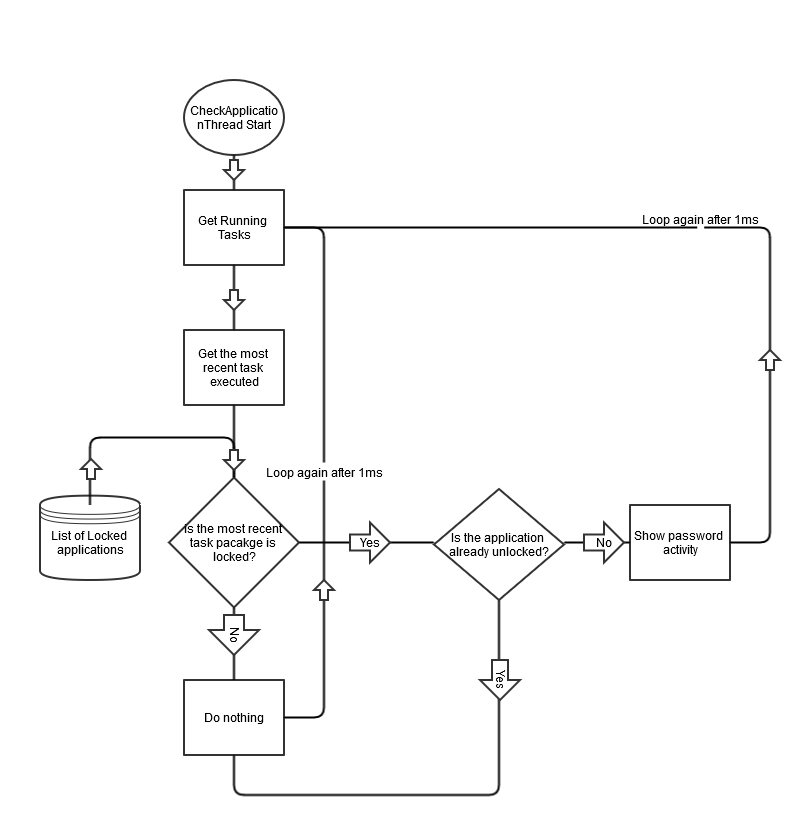
**OutgoingSMSRecepter** is the class which checks if the outgoing SMS is sent by the user of if the message is sent by some other application. If latter is true a notification is generated giving the application name which sent the message, also the content of the message it sent. More control over SMS can be expected in further development.

**Application Locker**

Application locker does just what the title suggests. The locker interface is simple and it displays the list of all the installed application. With each application there is toggle button which lets user to lock or unlock the particular application. The mechanism is simple as is, a database table is created which store the package name of all the locked application and is empty if no application is locked.

The heart and soul of application locker is the background service that keeps track of all the application that is launched. This service is sticky and need to be run always in the background. Android provides API to get the list of recent running task on the device, the **AppLockService** calls **getRunningTasks()** method of **ActivityManager** class object. The list is arrange with the top element being the most recently executed. Using the top most element of the list it is determined whether the application launched is locked or not. If locked the password activity is launched.

The flow chart for the process is shown in the diagram.



The password activity is implemented as given below.

**Password Activity**

The password prompt is implemented using a separate activity and is displayed each SeraphimDroid is started or a locked app is started. The password is stored in the database in the form of byte array, which is actually the SHA256 hashed passcode string. Java provides the required method to hash the string

MessageDigest digest = MessageDigest.getInstance("SHA-256");

hash = digest.digest(passwordConfirm.getBytes("UTF-8"));

*hash* is then stored in the Database.

Whenever the user enters the password it is validated by getting the byte array from the database and comparing it with the string hash of the passcode entered by the user. If they match password is confirmed and access is granted.

**Services Locker**

Services Locker's implementation involves a listener for each of the services. For ex. Wi-FiStateReceiver for Wi-Fi. This receiver listens to change in state of the Device's Wi-Fi. If there is a change (User switches on/off the Wi-Fi), SeraphimDroid password prompt is shown, and the service is returned to it's previous state.

For ex, if the user switches ON Wi-Fi, Password prompt will be shown, and Wi-Fi will be disabled by the command:-

Wi-FiManager.**setWifiEnabled**(false);

Switching ON/OFF these services require special permissions -

**Wi-Fi** - android.permission.**CHANGE\_WIFI\_STATE**

**Bluetooth** - android.permission.**BLUETOOTH\_ADMIN**

**Mobile Data** - android.permission.**CHANGE\_NETWORK\_STATE**

**Geo-Fencing**

Geo-fencing is the most resource intensive service, it requires you to provide Device Administrator Privilege to the application, which controls the device lock, and wiping of data. Moreover, it asks you to enable GPS for better location tracking, so GPS is must too. Once these features are enabled only then you could access the feature of this service to the full extent.

The class that does most of the work is **GPSTracker** class, this class gets the current device location from the GPS or network provider whichever is available. Using the location from this class the GeoFencing service keeps track of the device, if it is still inside the virtual fence or moved out of it. The center is set to the location at the time the user starts the service, which acts the center of the fence. After the service is started the location is queried every 10 seconds. That’s not too much battery draining but also not losing the timely location tracking that’s needed.

The other important class is **GeoFencingService**, this class takes care of all the actions that device must perform in case the device is assumed stolen. The device is said to be out of the range by calculating the distance from the center which is the location coordinates used at the time service was started and the current device location coordinates. The API provides the method to do so which is **Location1.distanceTo(Location2).** This class keeps track of the device location and in case the device is found to be out of the fence, it checks the location 3 more times and when it confirms the phone is out of the fence, it activates the alarming sound using **AlarmManager.** Similarly, current location is sent to a secure number set by user enables the feature.

As this service requires a working GPS, a prompt is made if GPS is unavailable and unless user enables the GPS the service is inaccessible. A secure number is required to enable the service to get location and Device Administrator privilege is also must.

**Remote Lock / Wipe**

The Geo-fencing requires user to enable the service first but if the user forgets to enable the service he could still lock or wipe data of his phone using remote wipe. Remote wipe uses the same locking mechanism of **DevicePolicyManager.** It uses the same broadcast receiver created for received SMSs. The receiver checks the message for the secret code set by the user, if the message contains the secret code, it activates the remote services and locks the device and if enabled wipes the user data from the device.

The main methods responsible for performing the remote actions are:

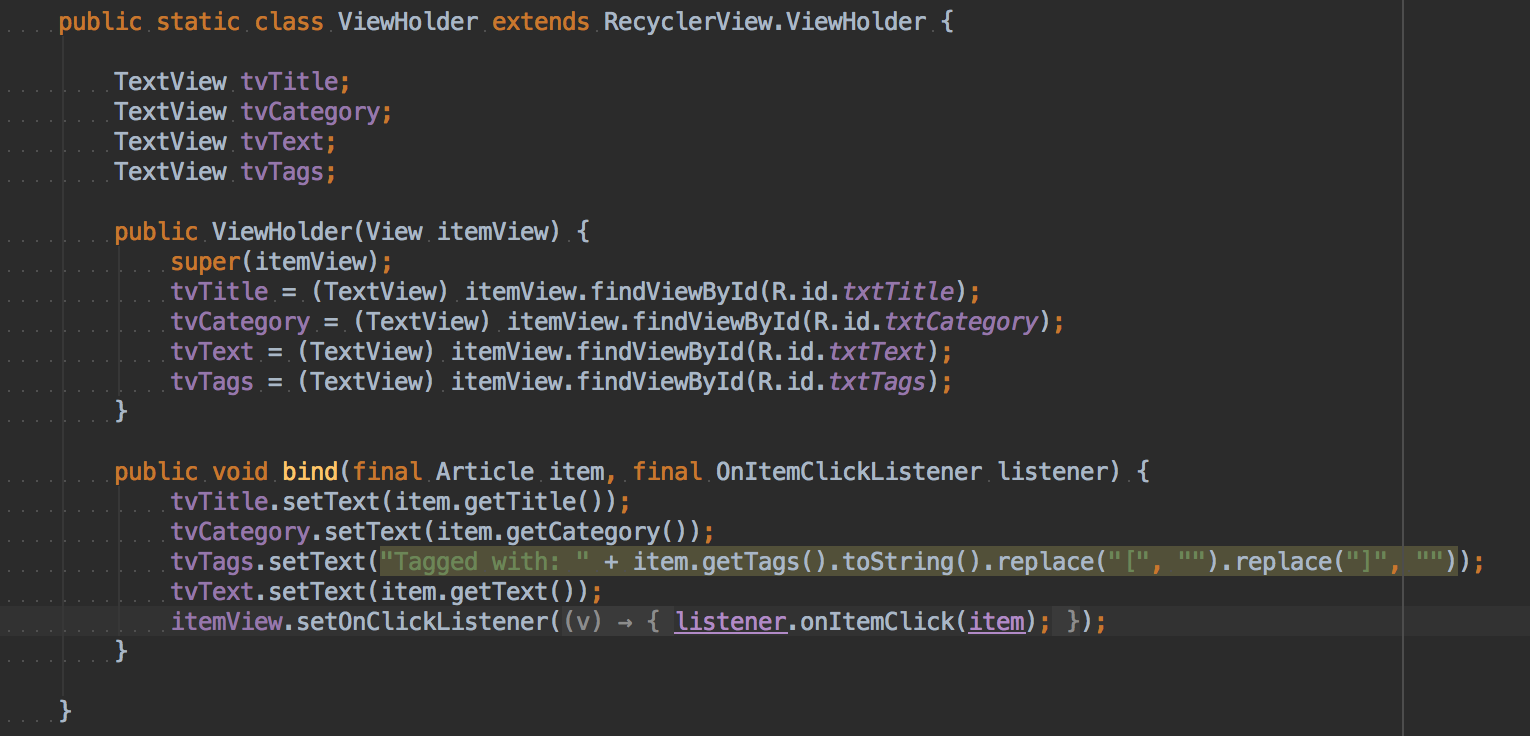
devicePolicyManager.wipeData (0);

devicePolicyManager.lockNow ();

For sending current location, current location is fetched using **GPSTracker**, and using **SMSManager** API a SMS is sent to the secure number that user believes could be reached in case the user’s device is stolen. The secret code is stored in application preferences which is retrieved each time is a new message is received to check if the message contains the secret code.

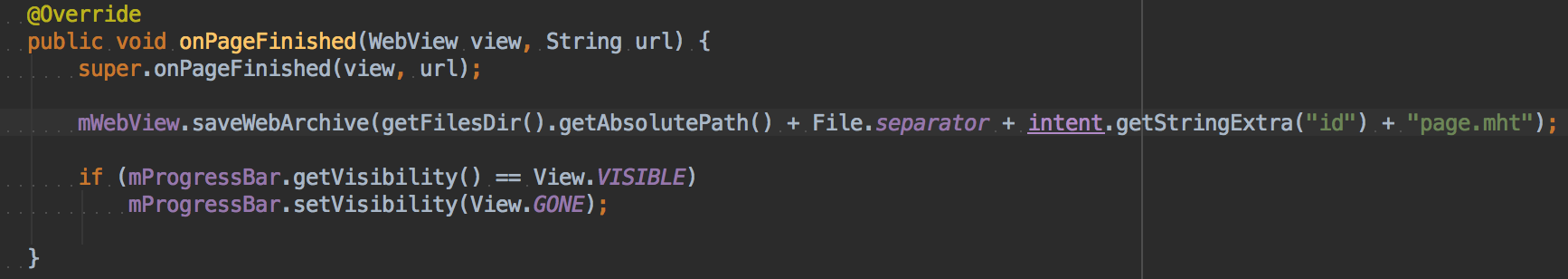
**Knowledge Base**

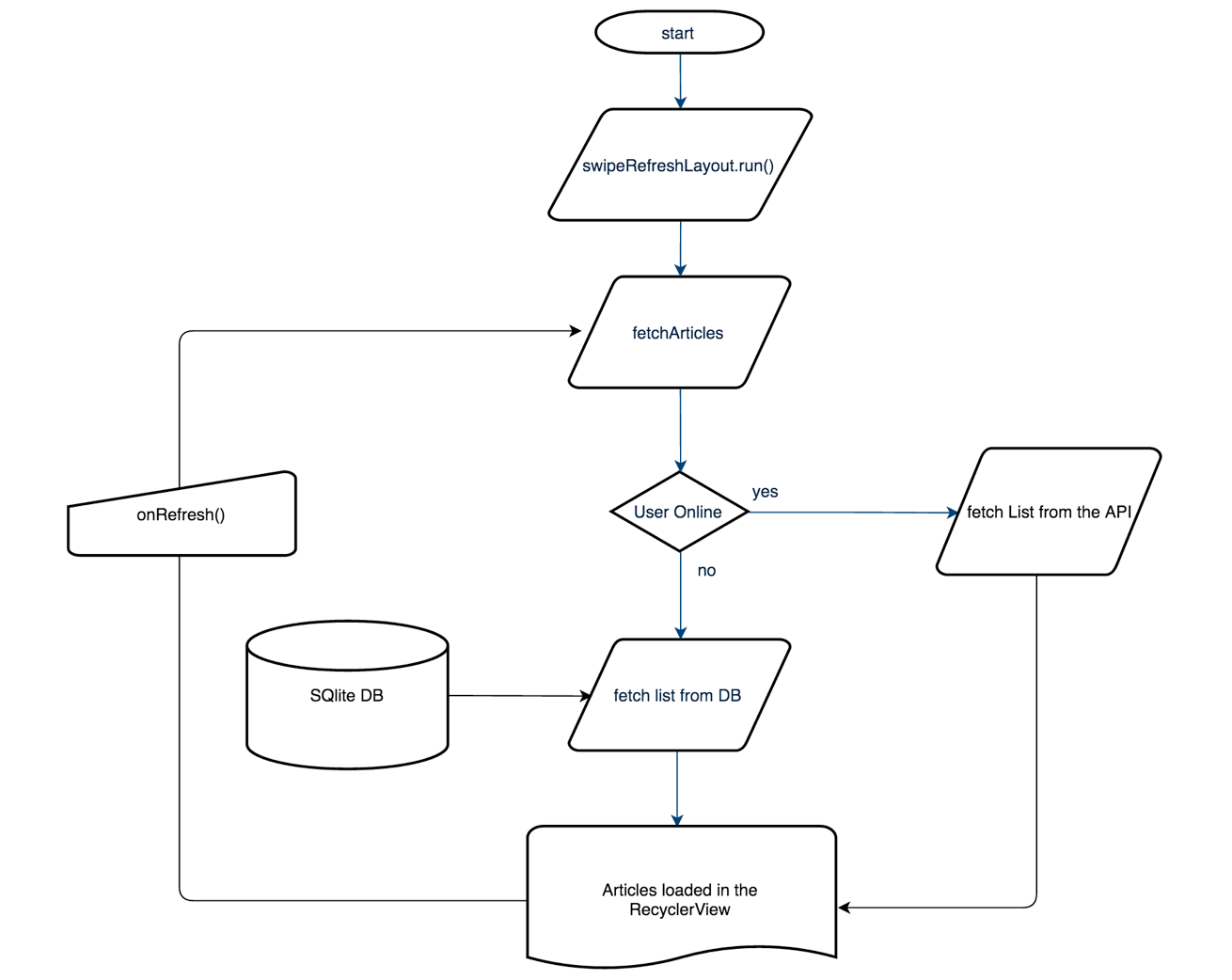
**This feature enables the User to read and view articles published by the writers on the Seraphimdroid Knowledge Base. This feature is fully integrated into the app as the articles displayed in the Educate Fragment. The articles are basically residing in a RecyclerView in a SwipeRefreshLayout and the list is cached in the SQlite db in the App.**



**Every article item is updated with this Viewholder in the Adapter**

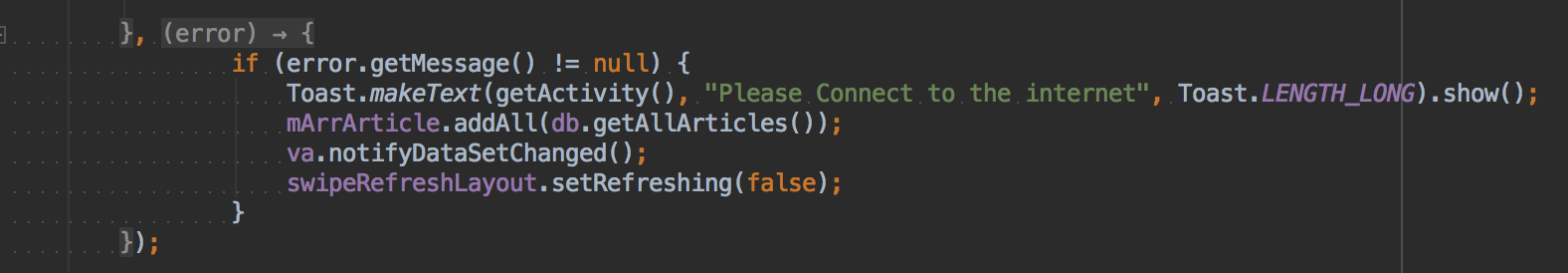
**When the User is offline, the adapter is loaded from the article objects cached in the Database and displays a Toast to the User that the User is Offline.**





**The articles from the Knowledge Base are loaded in a Webview and are also cached as Web Archive onPageFinished() and are saved as a .mht file. Whenever the Offline User clicks on the same Article, the Article is loaded from the .mht file. The Web Archive is updated whenever the user comes Online the next Time.**

**The Network requests are mostly JsonArrayRequests which are made using Google’s Volley Library and are parsed manually.**



**When Volley gives an Error while fetching, the article fetched from the db and the adapter is updated.**

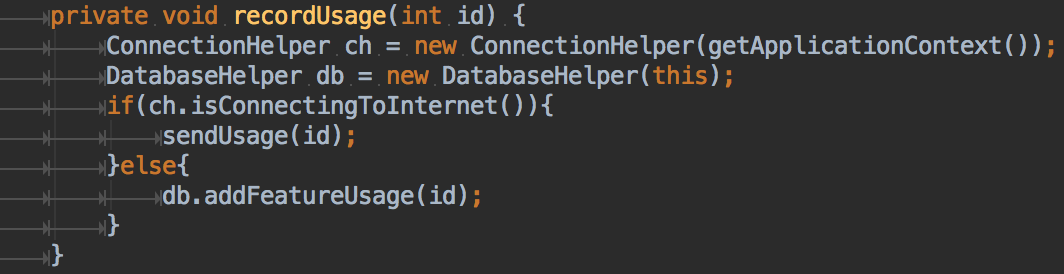


**Function for returning all articles list from the DB.**

**The articles are also maintained under a constant counter for maintaining statistics for the articles on the** [Knowledge Base](#kbas) **so the Editor knows how popular the article actually is. The Reads are also cached offline so that there is no miscounting of the article reads.**

**Statistical Functions**

**We have also developed statistical** [Recording API](#kbas) **for recording the feature Usage Count and Maintaining that data for the Admins to review the Feature Usage.  
The Usages are basically POST calls on the API** [Endpoints](#kbas) **on the** [Knowledge Base](#kbas) **so that they are recorded with the current Timestamp.**



**There is also support for recording the usages offline, by sending them to the Android Cache DB against that particular feature.  
And then Upload those Usages whenever the User comes online next time. By sending the usages as body of the request.**



**Search for the Knowledge Base**

**The Application uses a Compiled Library for searching through the articles present and available on the Knowledge Base. It basically performs a query on the Parsed and Indexed Data from the JSON in the Assets which is indexed and refreshed whenever the Article is Updated.  
  
The Library is basically a port for Lucene to work on Android using the JARs from the Lucene Core Library. The App uses Lucene so that it can perform faster and reliable search results based and sorted on the basis of Relevence.**

**Note: All the Information regarding the Knowledge Base sister project** [owasp-educate available here](https://github.com/addiittya2006/owasp-educate/wiki) **and the App is** [here](http://educate-seraphimdroid.rhcloud.com/) **for you to Visit.**