NFLogger
Author : Nilesh Agrawal

Index :

1. Introduction :

NFLogger is a client side logging iOS Library which provides APIs to application developers to log UI events to a remote server or persist log entries to disk if the client is offline, for later dispatch when the client comes online.

Motivation and application of such libraries :

- Usage Data : To know how applications are used by the user.
- Performance Metrics : Know how much time a particular event takes to complete, hence know performance metrics.
- Such libraries are foundation basis of analytics , performance management products.
- Enterprise Software Companies who build such libraries : Google Analytics, Flurry Analytics, CA Mobile Analytics, Appdynamics, New Relic.

2. Understanding of Problem and high level design consideration:

Logger should capture the following information for each event :

1. Type of event being logged.
2. Time the event occurred.
3. Period of time the active event spanned.
4. Data payload associated with the event.

The user uses the app, which involves clicking of some buttons i.e. generating some UI events. The implementation will have some "logEvent" apis called which will be called by UI Actions. This will help NFLogger library to collect data metrics, metrics will be enqueued for logging to server if client is online or stored and upload later if offline.

System Design Related notes:

1. Reduce Network Calls :
    a. Problem : Number of events generated every 5 minute >= 100, so making 100 network calls in 5 minute, will cost CPU. As well as, if the application has 100 million customers, there would be 100 million network request made per second. Also there can be cases where developers want to see data in real time,without much delay, so batching must be customized.
    b. Solution :
        - So we should do batching of data, while doing network request to reduce the number of calls.
        - This will require to store the events and upload later at certain event.

- We can collect data at certain time intervals, let's define this as **upload Intervals.**
- We can keep the upload interval parameter which can be set by user hence provides flexibility.

2. Timestamp selection :
    a. Problem : User wants to know when a particular event has happened, so we need a consistent way to define time, which is independent of the the app start time or any other app related timing events.
    b. Solution : We can use unix timestamp with the event.

3. Handle Active Spanning Events and Data Caching :
    a. Problem : Some events which are actively spanning will run in background to other events, how will we handle them ?
    b. Solution :
        i. We can need to store that data somewhere to cache it and then operate on it, when the end timestamp or upload event cycle is run.
        ii. We can use storage like sqlite, in memory or core data to store such events, and process them at upload event.
4. Don't block main thread :
    a. Problem : We should not to block main thread, and Sqlite will require only one thread to open close it at one time.
    b. Solution : We need to use Serial Queue and async threads.
5. Information to be associated with the event.
    a. Problem : We need to collect minimal and relevant data with each event.
    b. Solution : Add only relevant data as per design problem.
        i. Event Name
        ii. Event Parameters - It can be a dictionary of key value pairs.
        iii. Timestamp. - unix timestamp - kind of keeps things consistent.
        iv. Event Type.
6. What happens in background :
    a. Problem : Do we need to collect data in background ? Is the user relevant to us ?
        1. We only upload in foreground. Also, collection of events happen in foreground
        2. We should stop the timer, when user goes to background and start the timer when the user goes in foreground.
7. Failure Handling and Debugging (Logging) :
    a. Problem : How to know what is happening inside the sdk to debug it ?
    b. **Solution :**
        i. **Logging :** SDK should have different levels of logging (error, debug, verbose).
        ii. **Failure Handling :** The SDK should have a mechanism to disable it in case of failure handling . We need to have disabled SDK mode, rather than creating problems in host application.

iii. Know version of SDK integrated : App developers should know which version of SDK they are using.

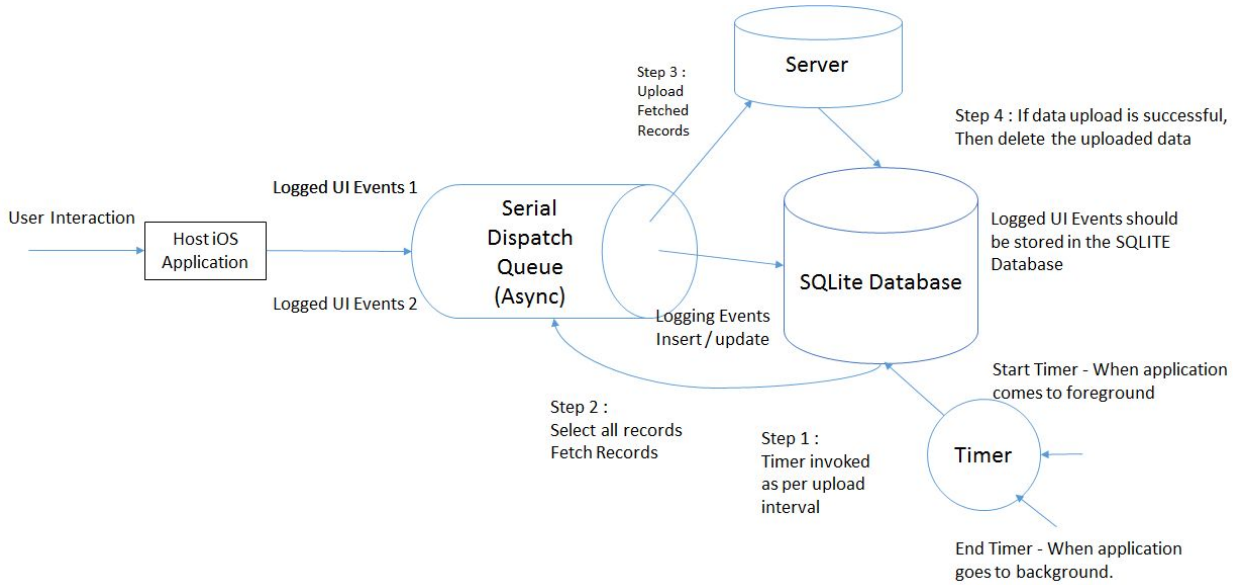
API Design Related Notes :

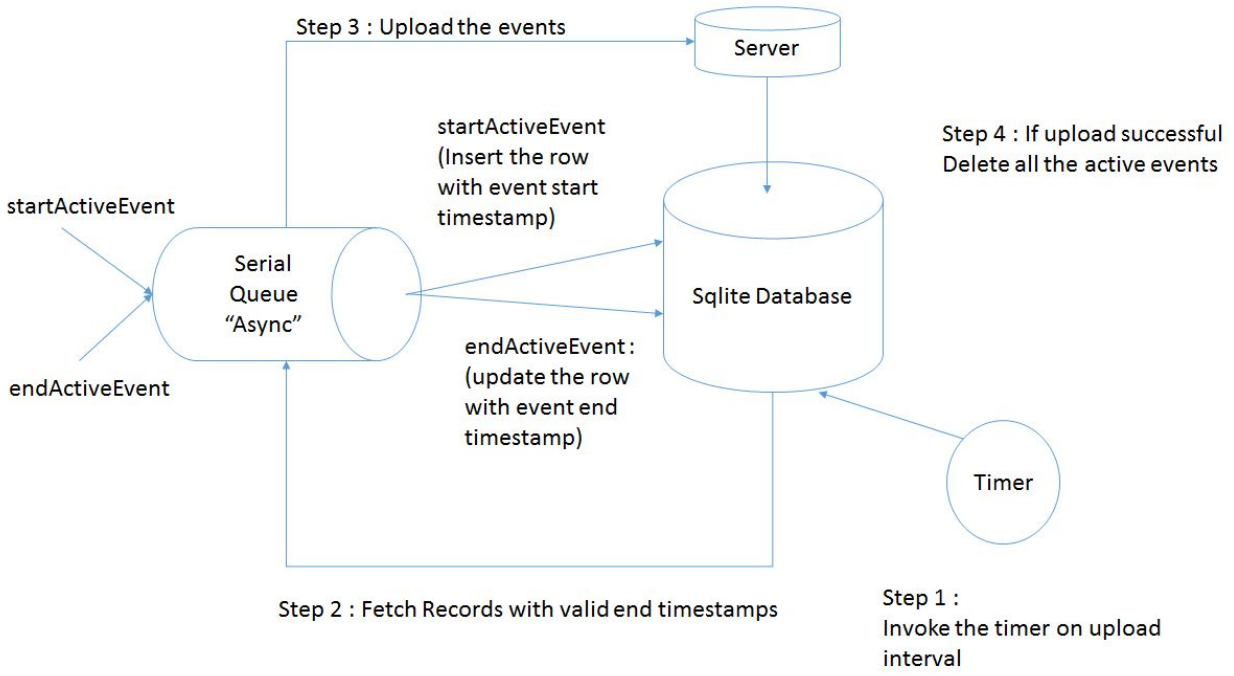
1. Send Status of Logged Events :
    a. Problem : User should know about the status of logged events, so that he can associate some event further in his UI or app.
    b. Solution :
        i. The Api should have a completion handler to the user to know what is status of recorded event.
        ii. User should know if the event is Recorded, Failed, or is the SDK Disabled.

Assumptions Made :

- Right now , I dont have backend server to accept server, so assumping successful upload case
- Also, due to shortage of time, I am not implementing AutoLogger, just putting a entry point for it in NFLOGAutoBehaviour Class.

3. High Level Design Architecture :





- Once the events are logged via NFLogger's API the user should dispatch the async task of insertion or update in the sqlite database through a serial dispatch queue
- We start the timer at the timer of initialization or the any time when the user comes to background and invalidate the timer, whenever the user goes to foreground.
- Uploading Action is based on timer. In case if the user is offline, the data is not uploaded, just persisted.

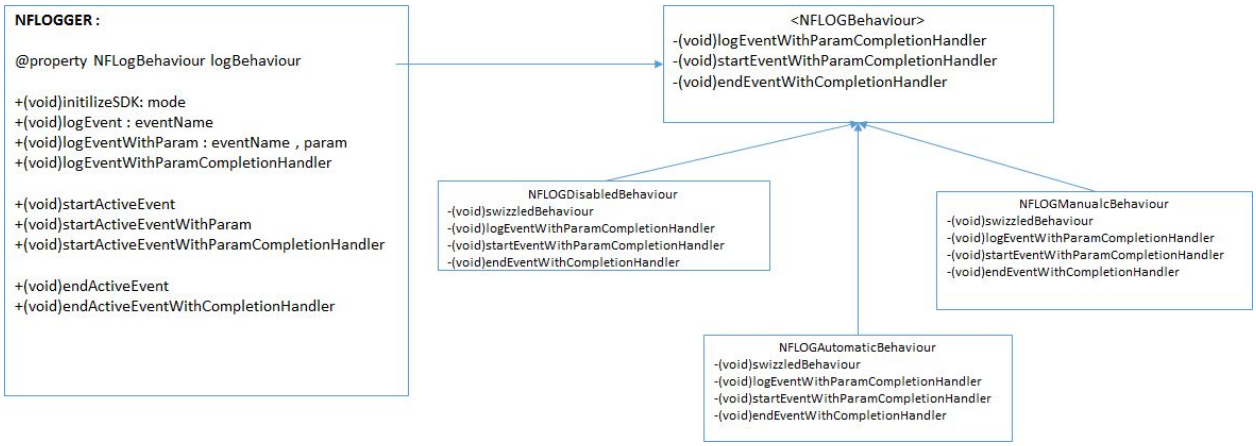
Important Design Choice :

1. Selection of Serial Queue over concurrent queue.
    a. If we refer the documentation of the serial queue https://www.sqlite.org/threadsafe.html we can find out that sqlite can be used by multi threading application with a restriction that only one thread can read and write at the same. In order to provide exclusion between the threads and avoid the hassle, we can use serialized queues over concurrent queues. Thus, through serial queues, we resolve the reader / writer problem too.
2. Choice of Async Task for Insertion /Updating / Deleting the data.
    a. Good thing about async task is that once it is dispatched in the serial queue, it will not block the current queue to accomplish the task, and hence it is more efficient.
    b. We are also planning to provide user status of our logged request through completion handler, so async thread is good option, as it does not block the main thread in completion handler.
3. Choice of Sync Task for Fetching Records.

    a. Before making this design choice, let us see, what are we trying to do with fetching of records : We are FETCHING - FORMING JSON - UPLOADING.
    b. All these three task need to be synced together, If I was about to use NSURLConnection syncRequest, using the same architecture as above would be fine, but in our case I am using NSURLSEssion, which itself operates on another Operation Queue.
    c. *So, In order to Sync this three task one after another , we have created sync thread. By this records are fetched in sync thread, then we are uploading the records, and then as know the record ids of this request, we use them to delete those specific records which are uploaded.*
4. Selection of SQLITE over Core Data
    a. Our database is quite simple , with no mapping between the tables. We are just adding deleting, updating, and selecting records, without any complex operations. The tables are just to means to Cache the data. So simple sqlite would help us gain good performance gain.
    b. Core data would just act as an abstraction over sqlite. Do we need that abstraction ? NO.
5. Selection of NSURLSession over NSURLConnection
    a. We are using NSURLSession.
6. Why do we have completion Handlers ?
    a. We want to notify app developers, if the event has been logged inside the library properly, one important reason is because they want to keep track of it, or they want to add some logger, or do some UI Action.
    b. Why completion Handlers over delegation ?
        i. Completion Handlers provide clean implementation for a particular API, the app developer does not have to implement all the methods in the protocol. Just that api in which he /she is interested in.
        ii. Handing threading is again more efficient in this case. Delegation will make it difficult for the USER.

4. Low Level Object Oriented Design :

1. Design of different SDK Behaviour :



- **The SDK behaviour is based on strategy design pattern.**
- NFLogBehaviour protocol defines mehod that concrete classes should have.
- NFLogDisabledBehaviour, NFLOGActiveBehaviour, NFLOGSpecificBehaviour implement NFLOGBehaviour and provide concrete implementation.
- The initliazeSDKWithMode sets the loggerBehaviour and thus whenever NFLogger APIs are called they call this methods from NFLOGBehaviour based on behaviour set.
- Thus follows isolation and clean design which can scale later on.

2. Design of Thread Management :

- NFLOGRequestManager is a singleton object which also acts as mediator between database, event logging and upload classes.
- NFLOGRequestManager creates serial queue, in which task are dispatched in sync fashion.
- As per high level design, based on timer, request are fetched and then sent to NSURLSession to upload.

3. Design of Database Management ;

- NFLOGDatabaseManager is a singleton design pattern which acts as single point of contact for SQLITE class.
- It opens the Sqlite Database, does SQL Operation and closes it.
- SQL Operations are further table class based.
- The NFLOGTableFactory gives instance of NFLOGEventTable based on eventType.
- The NFLOGDatabaseManager has a refernce to NFLOGEventTable, we set this table based on eventType, NFLOGDatabaseManager acts accordingly.
- Classes like NFLOGEventTableRow, NFLOGSqlParam,NFLOGSqlite acts as interpretor for data conversion to deal with sqlite methods.

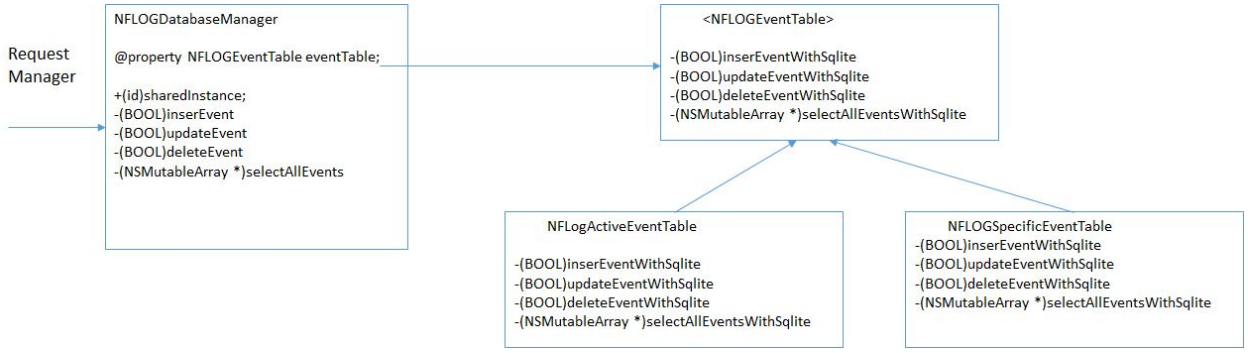


Table Schemas :

Columns in NFLOGSpecificEventTable

ROWID AUTOINCREMENT
EVENT_NAME
EVENT_TYPE
EVENT_PARAMETERS
EVENT_TIMESTAMP

Columns in NFLOGActiveEventTable :

ROWID AUTOINCREMENT
EVENT_NAME
EVENT_TYPE
EVENT_PARAMETERS
START_EVENT_TIME
END_EVENT_TIME

5. Discussion of Features that we can add in future :

1. Automatic Logging
    a. Automatic Logging means logging data events without developer requiring to use the apis
    b. Automatic Logging will require us to intercept the methods in runtime through method interception technique.
    c. Method Swizzling is a very popular method interception technique, where you can intercept the methods in runtime, where we can swizzle implementation App's method with our method and log the request and pass the control back to App's method.
2. Policy Management
    a. Some users don't like to capture all the events at once, they want to control which events are controlled, when to enable / disable SDK. This can be only achieved through policy management.

    b. SDK downloads the policy on a certain interval, based on that, the logBehaviour will exhibit different behaviours.
3. Configuration Management.
    a. Right now, I have hardcoded the API Key, Tenant ID, Policy URL amd Upload URL.
    b. It would be great if the end user, just downloads the file, drag and drop, we should read the file and store this important entries which can be helpful to upload and configure the SDK.
4. Javascript Requests
    a. Right now SDK just supports request from iOS Application, we can in future, add a java script interface for the user to call, and intercept those java script request through javascript injection.
5. SSL Pinning
    a. We need our server to accept request from those SDK which have valid certificate, to maintain security and sanity at server side.
6. Encryption
    a. If the developer is capturing some sensititve data, then we should provide some mechanism to encrypt the data whille storing in database. before uploading and decrypt it.
7. Average out the logging of data to reduce the number of data rows being sent.
    a. Right now we are sending lot of request .
    b. We can average out similar type of event, and this will reduce the amount of data being sent per request. Reducing the load drastically.
8. Rolling database concept.
    a. If the user is offline for several days and using our application. There will be so many entries stored, we need to delete the old data, which is not required and store only fewer entries as possible.
9. Location based disabling and enabling of the sdk
    a. In order to support user privacy, we should not capture data in personal location zones.
10. Compression of JSON
    a. We should use some libraries like GZIP to compress the data, right now I have avoided it just to avoid use of 3rd Party code.

6. Other Documentation :
    1. How to test SQLITE Database
    2. Tools used.
        a. Version Control System : GIT
        b. Continuous Integration : Travis CI
        c. Unit test code coverage : XCTest [Native iOS Unit Testing] 55% [Purposefully avoided OCMOCK to limit use of 3rd party for the test]
        d. Sample Test Application : NFLoggerDemo
        e. Documentation : Oxygen.