***theLife* Project**

**Server API Design To Support External Devices**

March 11, 2013

Clarence Martens

March 11 Design Changes:

* Simplified the event stream.
* Expanded the database schema description in section II.
* Timestamp attribute added to GET methods for activities, groups, users and friends.
* Added an explanation section

March 9 Design Changes:

* user\_id and friend\_id and event\_id become unique across the system (and not just unique in a group)
  + this change complicates database sharding but is simpler to implement, easier to understand, and better follows the customer’s intent: when a user creates an event, it is shared with all fellow group members in all groups the user belongs to
* new get all activities call
* user database record now has password field

# I. Introduction

## General Design

* All device API calls to the server are REST based, and all parameters are JSON.
* Beyond standard HTTP return codes, (eg 404 and 500), all HTTP successful calls return an application status code and, if necessary, a translated and displayable description of status code
  + status code == 0 means no error
  + status code <> 0 means an error, plus an optional translated description
* Server copy of data is always master; it records the most correct values despite whatever may be on the device.
* User must be authenticated (log in) before using app, although there may be a way to view tutorials/help screens without logging in.
* If a translation is missing, the English translation is always present and used instead.
* Clients identify their version number in each API call. We only change the version number for a breaking API change. Versions numbers can be compared numerically.
* As much as possible, non-presentation elements are controlled and defined by the server; presentation is defined by the device

## Scaling Strategy

* Version 1 is a single server located on customer premises. It will be implemented using Ruby on Rails and ActiveRecord (PostgreSQL or perhaps MySQL).
* If further scaling is needed, separate web cache, web head and database servers can be utilized.
* If even further scaling is needed, separate systems or perhaps database sharding can be used. See sharding discussion below.

# II. Database

## Background

The database schema is as follows:

* a system has many users (could eventually be a million users).
* a system has many groups (could eventually be hundreds of thousands)
* a user can belong to 0 or more groups
* a group has 1 or more users, up to a maximum limit (max is something like 12 or 15)
  + the user who originated the group is the group’s leader
* a user has 0 or more friends (likely under 10, but could be 50 or more)
  + friends are not users
* each friend is characterized by a single threshold, according to the following list of thresholds:
  + new\_contact,
  + trusting,
  + curious,
  + open,
  + seeking,
  + entering,
  + Christian
* a system has many activities (likely around 100 activities at project launch)
* each activity is characterized by the thresholds for which it is applicable
* events are generated when a user undertakes an activity with one of their friends. The generated event is visible for all the user’s fellow group members to see (and pray for) in their event feed.

Users may undertake an activity with one of their friends, which generates an event for fellow group members to see (and pray for) in their event feed.

## Database Records Schema

* Group
  + group\_id
  + name (string, not translated)
  + set of user ids that belong to the group (for a relational database, this would be a group\_id field in a separate group\_user join table)
  + group leader user id
  + creation and updated times
* User
  + user id
  + photo, thumbnail *(TODO: UI designer needs to determine size)*
  + photo, larger *(TODO: UI designer needs to determine size)*
  + first name, string
  + last name, string
  + language (e.g., en, fr, etc – based on standard language codes)
  + flag: active/suspended (controlled by admin)
  + group ids for all the groups this user belongs to (for a relational database, this would be a user\_id field in a separate group\_user join table)
  + mobile phone number
  + email, this also doubles as the username
  + password (hashed version is stored in database)
  + creation and updated times
* Friend
  + friend id
  + photo, thumbnail *(TODO: UI designer needs to determine size)*
  + photo, larger *(TODO: UI designer needs to determine size)*
  + first\_name, string
  + last name, string
  + mobile phone number
  + email
  + threshold level (enum: new\_contact, trusting, curious, open, seeking, entering, christian)
  + associated user id (the friend of this user)
  + creation and updated times
* Activity (preliminary definition, enough for devices)
  + activity\_id
  + title (one line string, translated with placeholders; shown on the relevant activity list after selecting What’s Next)
  + summary (one line string, translated with placeholders, shown in the event feed on the Community screen)
  + full description (HTML paragraph or paragraphs, translated with placeholders)
  + category (one line string, translated)
  + thresholds that this activity is applicable to
  + Note: activities are created on the server, with translations. They are defined with the help of templates that contain placeholders. Here are some examples:
    - $uf = user’s first name
    - $ff = friend’s first name
    - $ul = user’s last name
    - $fl = friend’s last name
  + creation and updated times
* Event (seen on community screen)
  + event\_id
  + user\_id
  + friend\_id
  + activity\_id
  + Boolean: prayer requested
    - If true, a list of user\_ids that have pledged to pray for this event. (for a relational database, this would be a separate event\_user join table).
  + Activity summary (one line string, translated with placeholders, shown in the event feed on the Community screen)
  + Threshold change flag (enum: threshold move up, threshold move down, neither)
    - If not neither, then the new threshold level enum.
  + event creation time

## Schema Id Definitions

* group\_id: unique for each group in system. At some point in the future, may be turned into a sharding key to shard the overall database into database shards so that data for a group is localized and centered into one database shard. This implies that all groups that a user belongs to are contained in one shard.
  + E.g., integer
  + *Version 2 Idea: support sharding. This means there must be a central server to create group ids and route based on group ids. Group ids could be pre-partitioned according to geographical boundaries, eg, groups in the Philippines could have their own range of ids. Or we could let a database like MongoDB sort that out on its own. Or, even more simply, we could have a totally separate server in the Philippines. This last solution is simplest, acceptable to the customer, and only needs operational (not developmental) work.*
* friend\_id: unique for each friend in the system
  + E.g, integer
* user\_id: unique for each user across the system
  + E.g. integer
* activity\_id: unique for each activity in the system
  + E.g., integer
* event\_id: unique for each event in the system. It is sequenceable, so that events can be sorted by their occurrence.
  + E.g., **64 bit integer** or timestamp

## Security Notes

* Server is HTTPS.
* Use ProGuard for Android app.
* Authentication is a concern: Even if this is not a financial app, how to know that the client is who they say they are? At start of session, user must authenticate themselves with username (known) / password (secret). The server then replies with a token that the client provides with every subsequent API call. The token uniquely and securely identifies the user. At some point the token expires, and the user must re-authenticate. To make this usable, the token will be stored in the saved state when an app goes into the background, which is less secure, but still necessary. See OAuth2 examples.
  + Amount of time before session expiry is a server configuration parameter*.*
  + For even better security, we hash the password on the device before sending it to the server.
  + Server API: authenticate username=u password\_hash=p  
    **HTTP POST api/v1/authenticate?username=u&password=p**
    - **Returns authentication token**
  + Server API: each call, except authenticate, can fail with an authentication failure, meaning that the user must reenter their password
  + Admin: need way to suspend a user account (e.g., user reports stolen phone)

# III. Device API

## Community Screen API Calls

1. Get events for me after the given event id. This will return community events and pledge count events, which are seen on the community screen.  
   **HTTP GET api/v1/events?token=<token>&after=<event id>**
   1. Event stream is a sequence of community events and prayer pledge count events.
      1. Community event
         1. Like the event database record, but instead of the list of pledged user\_ids there is a flag for prayer requested events, indicating whether or not the user has already pledged to pray, and the count of users who have already pledged
      2. Prayer pledge count update event
         1. event\_id (must be an existing event, or the device will ignore it)
         2. new pledge count (integer)
      3. // DO NOT IMPLEMENT, NOT NEEDED FOR NOW New Friend / Update Friend event (would only be needed for caching thumbnails for other users’s friends)
         1. Event id
         2. Friend\_id
         3. Photo thumbnail
      4. // DO NOT IMPLEMENT, NOT NEEDED Delete Friend event
         1. Group id
         2. Friend id
      5. // DO NOT IMPLEMENT, NOT NEEDED FOR NOW New Group User / Update Group User event (to keep other users in group in sync)
         1. Group\_id
         2. Event id
         3. User\_id
         4. First name
         5. Last name
         6. Photo, larger and thumbnail
         7. Email
         8. Phone number
      6. // DO NOT IMPLEMENT, NOT NEEDED FOR NOW Delete Group User event (to keep other users in group in sync)
         1. Group id
         2. Event id
         3. User id
         4. Note: The user id can be the device user.
      7. // DO NOT IMPLEMENT, NOT NEEDED Delete Group
         1. Group id
         2. Event id
   2. To avoid having too many events from this call, the server would do a count before sending off to device client, and only send the most recent N events. N is a server configuration parameter.
2. Pledge to pray for event=event\_id  
   **HTTP PUT api/v1/pledge/<event id>?token=<token>**
3. Get earlier events for user=me\_id before given event ids. This will return an array of community events.  
   **HTTP GET api/v1/events?token=<token>&before=<event id>&max=30**
   1. Get community events before the given event\_id. This is needed to support the user scrolling to the earliest event and then wanting to see even earlier events.

## Friends Screens API Calls

1. Get all my friends. Returns a list of Friend records. A server timestamp is also returned.  
   **HTTP GET api/v1/friends?token=<token>**
2. Get all my friends according to the timestamp. This will return all the friend records if any change has been made to any of my friend records after the timestamp. If no change has been made to any of my friend records after the time stamp, no friend records are returned. The server’s timestamp is also returned.  
   **HTTP GET api/v1/friends?token=<token>&timestamp=<timestamp>**
3. // DO NOT IMPLEMENT, NOT NEEDED Get latest events for user=me\_id and friend=friend\_id and max\_number=n
   1. Not necessary, instead this can be filtered from the current event list, which is added to the front of the per user event list cache.
4. Create event by me with friend=friend\_id and activity=activity\_id  
   **HTTP POST api/v1/events?token=<token>  
   friend=<friend\_id>  
   activity=<activity\_id>  
   prayer\_requested=<true|false>**
5. Add my friend  
   **HTTP POST api/v1/friends?token=<token>  
   first\_name=<first name>  
   last\_name=<last name>  
   photo=<encoded>  
   thumbnail=<encoded>  
   email=<email address>  
   mobile=<mobile phone number>**
6. Delete my friend  
   **HTTP DELETE api/v1/<friend\_id>?token=<token>**
7. Get all activities. This will return an array of activity records, plus the server’s timestamp.  
   **HTTP GET api/v1/activities?token=<token>**
8. Get activities applicable to threshold=threshold\_enum. This will return an array of activity records, plus the server’s timestamp.  
   **HTTP GET api/v1/activities?token=<token>&threshold=<threshold\_enum>**
9. Get all activities [applicable to threshold=threshold\_enum] for the timestamp. This will return all the applicable activity records if any change has been made to any of the applicable activity records after the timestamp. If no change has been made to any of the applicable records after the time stamp, no activity records are returned. The server’s timestamp is also returned.  
   **HTTP GET api/v1/activities?token=<token>[&threshold=<threshold\_enum>] &timestamp=<timestamp>**
   1. Timestamp is UTC based, and it is based on a timestamp originating from the server.
10. Get earlier events for my friend=friend\_id. This will return an array of community events.  
    **HTTP GET api/v1/events?token=<token>&friend=<friend id>&before=<event id>&max=n**
    1. Get the community events before the given event id and up to a maximum n events. This is needed to support the user scrolling to the earliest event and then wanting to see even earlier events.

## Groups Screens API Calls

1. Get my groups and the users for each group. Returns group records and nested user records, and the server timestamp.**HTTP GET api/v1/groups?token=<token>**
2. Get my groups and the users for each group according to the timestamp. If any change has been made to my groups or to the users for each group since the timestamp, return all my groups and the users for each group. If no change has been made to my groups or to the users of each group since the timestamp, no records are returned. The server timestamp is also returned.**HTTP GET api/v1/groups?token=<token>&timestamp=<timestamp>**
3. Get users in my group. Returns a list of user records.  
   **HTTP GET api/v1/groups/<group\_id>/users?token=<token>**
4. Get users in my group according to the timestamp. If any change has been made to the users in my group since the timestamp, return all the users in the group. If no change has been made to the users of my group since the timestamp, no records are returned. The server timestamp is also returned.  
   **HTTP GET api/v1/groups/<group\_id>/users?token=<token>>&timestamp=<timestamp>**
5. Remove user from my group   
   **HTTP DELETE api/v1/groups/<group\_id>/users/<user\_id>?token=<token>**
   1. I must be the group leader
   2. cannot delete the group leader.
   3. will notify users via an event
   4. *Note that a group leader cannot delete another user’s friend.*
6. Create my group

**HTTP POST api/v1/groups**

**token=<token>**

**name=<group name>**

* 1. Returns group id
  2. User must not already be a part of max number of groups, which is a server configuration parameter.

1. Delete my group  
   **HTTP DELETE api/v1/groups/<group\_id>?token=<token>**
   1. I must be the group leader.
   2. Will notify any remaining users in group via an event.
2. Invite a user to my group. This will send an email or SMS to the invited person, asking them to join at a certain URL.   
   **HTTP POST api/v1/groups/<group\_id>/invite  
   token=<token>  
   email=<email> OR *phone =<phone\_number>***
   1. I must be group leader
   2. The given email or phone number must not already exist in the group.
   3. Not already at max number of users in group, which is a server configuration parameter.

## Settings Screen API Calls

1. Get my user record  
   **HTTP GET api/v1/user?token=<token>**
2. Set my user record  
   **HTTP POST api/v1/user  
   token=<token>  
   group=<group\_id>  
   first\_name=<first name>  
   last\_name=<last name>  
   photo=<encoded>  
   email=<email address>  
   mobile=<mobile phone number>**

## Other API Calls

1. Get System Settings from the server. Returns a list of name/value pairs. To Be Determined.  
   **HTTP GET api/v1/settings?token=<token>***The intention here is to inform devices that a system setting has been changed, for example polling frequency. This call is still To Be Determined.*
2. Use of Google / Apple notifications – To Be Determined.

# IV Explanations

## Timestamp Caching

Groups, Users, Friends and Activity records can all be accessed according to server timestamp. This allows the mobile device to quickly synchronize with the master dataset on the server.

For example, when the app on the device is first initialized, it can ask for the entire dataset as follows:

**HTTP GET api/v1/activities?token=<token>&threshold=<seeking>**

This will return all the activity records for the “seeking” threshold, and it will also return the server’s current timestamp.

Later on, the device may wish to check for more recent records, by supplying the server’s timestamp.

**HTTP GET api/v1/activities?token=<token>&threshold=<seeking>&timestamp=1234567**

If none of the activity records for the “seeking” threshold have changed since the timestamp, then no activity records are returned. But if any of the activity records for the “seeking” threshold have changed since the timestamp, then all the “seeking” threshold records are returned.

In any case, the server’s current timestamp is also returned.

## Event Stream

The event stream is a sequence of community events and prayer pledge count events, which are generated when users undertake an activity with a friend.

For example, lets say that a user is in a group and another member of the group, U1, undertakes an activity with friend F1. Then 3 members of that group pledge to pray for that activity. The event stream would be as follows:

* Community event id=12345, user=U1, friend=F1, activity=37 *(pledge count starts at zero)*
* Pledge event 12346, community event=12345, count=1 *(pledge count now one)*
* Pledge event 12347, community event=12345, count=2 *(pledge count now two)*
* Pledge event 12348, community event=12345, count=3 *(pledge count now three)*

If the user is watching their community screen, they will see the event with a pledge count at 0, then 1, then 2, and then 3.

If the user is not running the app, and later runs the app, they will get the event stream played back to them, **but the pledge count is always the current value for the event**, as follows:

* Community event id=12345, user=U1, friend=F1, activity=37 *(current pledge count is three)*
* Pledge event 12346, community\_event=12345, count=1 *(current pledge count is three)*
* Pledge event 12347, community\_event=12345, count=2 *(current pledge count is three)*
* Pledge event 12348, community\_event=12345, count=3 *(current pledge count is three)*

Watching their community screen, they will always see the event with a pledge count of 3.