**theLife Project**

**Server API Design To Support External Devices**

March 5, 2013

Clarence Martens

# I. Introduction

## General Design

* All device API calls to the server are REST based, and all parameters are JSON.
* Beyond 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, superseding what may be on 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, database sharding can be used in version 2. See sharding discussion below.

# II. Database

### Database 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
* User
  + user id
  + photo, thumbnail *(TODO: need to determine size)*
  + photo, larger *(TODO: need 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
* Friend
  + friend id
  + photo, thumbnail *(TODO: need to determine size)*
  + photo, larger *(TODO: need 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)
  + associated group id (the group of the user)
* 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)
  + 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
* Event (seen on community screen)
  + event\_id
  + group\_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: sharding key, unique for each group in system. Can be used to shard the overall database into database shards so that data for a group is localized and centered into one database 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.*
  + *Version 2 Idea: 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.*
* event\_id: unique for each event in the group. It is sequenceable, so that events can be sorted by their occurrence.
  + E.g., integer or timestamp
  + Group id plus event id is unique across system
* friend\_id: unique for each friend in the group
  + E.g, integer
  + Group id plus friend id is unique across system
  + *TODO: Verify with customer: a friend is associated with a user in a particular group, and not just with a user. This means that a user could talk about friend “Joe” in one group, but that would not immediately get passed on to all of the user’s groups.*
* user\_id: unique for each user in the group
  + E.g. integer
  + Group id plus user id is unique across system
  + *TODO: Controversial: This means that if a person is a user in N groups, they will have N user ids, implying that they have N user records. The purpose is to aid in sharding. Reads from the user record can go to any shard – writes must be to each user record. This makes sense because reads are much more common than writes*.
* activity\_id: unique for each activity in the system
  + E.g., integer
  + Never re-used, but can be retired on server.

## 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 APIs

## Community Screen APIs

1. Get events for me after the given event ids. This will return community events, seen on the community screen, as well as other group state synchronization events.  
   **HTTP GET api/v1/events?token=<token>&aev1=<group id>,<event id>&aev2=<group id>,<event id>**
   1. Event stream is a sequence of different event types, but each is sequenced by an group\_id plus event\_id
      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. group\_id
         2. event\_id
         3. new count (integer)
      3. New Group Friend / Update Group Friend event (needed for caching thumbnail for other users in group)
         1. Group\_id
         2. Event id
         3. Friend\_id
         4. Photo thumbnail
         5. TODO: decide if this is really needed; if not needed just put thumbnail in community event stream
      4. // DO NOT IMPLEMENT, NOT NEEDED Delete Group Friend event
         1. Group id
         2. Friend id
      5. 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. 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. Delete Group
         1. Group id
         2. Event id
         3. TODO: decide if this is really needed; if not just use Delete Group User instead.
   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.
   3. *Version 2: Need way to combine separate event streams, because of separate groups, into one stream ordered by time. Perhaps order result set by event timestamp – for multiple servers this requires keeping the clock between servers mostly in sync. This will only be an issue in version 2.*
2. Pledge to pray for event=group\_id,event\_id  
   **HTTP PUT api/v1/pledge/<group id>,<event id>?token=<token>**
3. Get earlier events for user=me\_id before event ids. This will return an array of community events.  
   **HTTP GET api/v1/events?token=<token>&bev1=<group id>,<event id>&bev2=<group id>,<event id>&max=30**
   1. Get community events before the given before\_event\_id. This is needed to support the user scrolling to the earliest event and then wanting to see even earlier events.

## Friends Screen APIs

1. Get friend list for me  
   **HTTP GET api/v1/friends?token=<token>**
   1. *Returns all my groups and my friends in each*
   2. *Called when coming to this screen and (no data or it has been a while since last sync)*
   3. *Note that a friend of a fellow group member is not also my friend.*
2. // DO NOT IMPLEMENT 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.
3. Create event by me with friend=friend\_id and activity=activity\_id  
   **HTTP POST api/v1/events**  
   **token=<token>  
   group=<group\_id>  
   friend=<friend\_id>  
   activity=<activity\_id>  
   prayer\_requested=<true|false>**
4. Add my friend into group  
   **HTTP POST api/v1/groups/<group\_id>/friends  
   token=<token>  
   group=<group\_id>  
   first\_name=<first name>  
   last\_name=<last name>  
   photo=<encoded>  
   thumbnail=<encoded>  
   email=<email address>  
   mobile=<mobile phone number>**
5. Delete my friend from group   
   **HTTP DELETE api/v1/groups/<group\_id>/friend\_id?token=<token>**
6. Get activities for threshold=threshold\_enum. This will return an array of activity records.  
   **HTTP GET api/v1/activities?token=<token>&threshold=threshold\_enum**
7. Get earlier events for my friend=friend\_id. This will return an array of community events.  
   **HTTP GET api/v1/events?token=<token>&group=<group id>&friend=<friend id>&bev=<group id>,<event id>&max=n**
   1. Get the community events before the given before\_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 Screen APIs

1. Get my groups *(including users for each group)***HTTP GET api/v1/groups?token=<token>**
   1. *Called when coming to this screen and (no data or it has been a while since last sync).*
2. Get users in my group   
   **HTTP GET api/v1/groups/<group\_id>/users?token=<token>**
3. 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.*
4. 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 APIs

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>**