***theLife* Project**

**Server API Design To Support External Devices**

April 8, 2013

Clarence Martens

April 8 Design Changes: all changes in yellow

* Simplified the handling of a request to join a group
* Added a one page listing of the REST calls

April 4 Design Changes: all changes in yellow

* Added a target event\_id field to the event record; if this is nonzero it is used to support pledge\_count update events.
* Created separate fields for the request record; this prevents different kinds of data being stored in the same field.
* Simplified the use of a threshold enum field in the event stream record; if it is there it indicates a threshold change as a result of the event.
* Simplified the event stream so that it contains only one type of record.
* Marked some API calls as lower priority.
* Make some API calls more RESTful.
* Make a difference between API GET calls for my objects and API GET calls for all objects. For example, HTTP GET /v1/my\_groups returns only the groups I belong to, while GET /v1/groups returns all groups in the system.

April 2 Design Changes: all changes in yellow

* Clarified that any placeholders inside activity summaries of event records are replaced with the real values before being sent to the mobile device.
* Group queries only return group records, and not also user records.
* New user queries API calls.
* Activity queries only return activity records, and not also category records.
* New category queries API calls.
* Added thumbnail PNG to activity record, and clarified that images are in PNG format.
* Added image access API calls.

March 20 Design Changes: all changes in yellow

* Database changes:
  + Added priority field to Activity database record,
  + Added description field to Group database record,
  + Category changes from a field in Activity into a new database record
  + New database record for admin user accounts
* Support for a threshold when adding a friend, and changing a friend’s threshold when creating an event
* New API call to change the group leader.
* New API call to search for groups.
* Clarified and simplified the user authentication and registration workflow.
* Clarified the mechanism to invite a person to a group, and request to join a group using a new Request record type. Also added explanations to describe how a person becomes a new user.

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

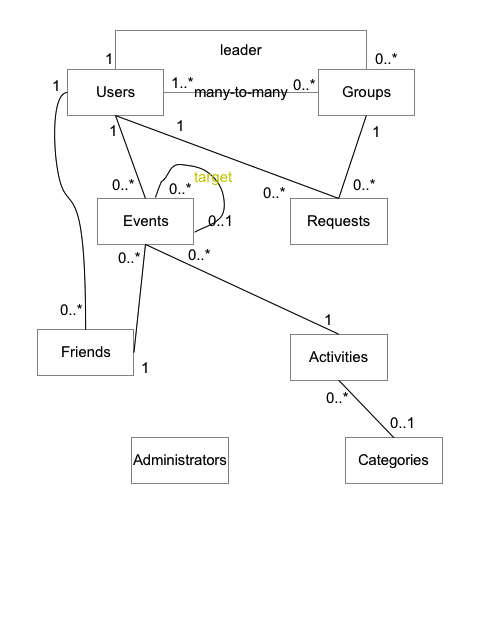
# I. Introduction

## 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)
  + description (string, 255 characters max)
  + group leader user id
  + creation and updated times
* User
  + user id
  + thumbnail PNG *(TODO: UI designer needs to determine size)*
  + image PNG *(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
  + thumbnail PNG *(TODO: UI designer needs to determine size)*
  + image PNG *(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
  + activity\_id
  + thumbnail PNG (TODO: UI designer needs to determine size)
  + 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)
  + thresholds that this activity is applicable to
  + priority (integer from 1 to 10, 1 is highest priority, 10 is lowest priority)
  + category\_id
  + 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
* Category
  + category\_id
  + category name, translated
  + full description (HTML paragraph or paragraphs, translated with placeholders)
* 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, shown in the event feed on the Community screen)
  + Target event\_id – this will only be nonzero when this event is about an earlier target event. Currently, this only happens when a user pledges to pray for an event X, which will create a “pledge event” with the new prayer count and target event\_id = X.
  + event creation time
* Request (when requesting to join a group, or requesting a person join a group)
  + request\_id
  + user\_id (the user creating the request)
  + group\_id (the group to join)
  + type (an enum: INVITE or REQUEST\_MEMBERSHIP)
  + email (if the group leader is asking a person to join the group leader’s group, then this is the person’s email, else it is null)
  + sms (if the group leader is asking a person to join the group leader’s group, then this is the person’s SMS, else it is null)
  + description (one line string describing the request)
* Administrators (only for logging into web interface for administration)
  + administrator\_id
  + email (this is the user name)
  + password
  + first name
  + last name

## 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? When the user first registers to get a user account, the user must register themselves with an email (known) and 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. In the first version of this application, the token does not expire, so the user does not need to ever re-authenticate. On the device, the app stores the token in protected storage.
  + For even better security, the password is hashed on the device before sending it to the server.
  + Admin: need way to suspend a user account (e.g., user reports stolen phone)

# III. Device API

## Authentication and Registration API Calls

1. Register a new user to get a user account.  
   **HTTP POST v1/register   
   username=<email>  
   password=<password hash>  
   first\_name=<first name>  
   last\_name=<last name>  
   language=<language>  
   image=<image\_url>  
   mobile=<mobile phone number>**
   1. Returns authentication token or a fail message if this username is already taken.
2. Login the user. This is not normally necessary, although in extreme situations, such as when a user deletes their app by mistake and then reinstalls the app, the user may need to login with this API call.  
   **HTTP POST v1/authenticate   
   username=<email>  
   password=<password hash>**
   1. Returns authentication token or a fail message if this username is already taken.

## Community Screen API Calls

1. Get events applicable to me after the given event id. This will return events that are seen on the community screen.  
   **HTTP GET v1/my\_events?token=<token>&after=<event id>**
   1. Events are as follows:
      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. Also a threshold enum field: if it is not null then it indicates that the friend’s threshold has changed because of this event and this is the new threshold.
      3. If the target event\_id field is not zero it records the event with the new pledge count.
      4. On the server the activity’s summary field may have template placeholders; these are first replaced with real values before being sent to the mobile device.
   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 v1/events/<event id>/pledge?token=<token>**
3. Get events applicable to me before given event id. This will return an array of events.  
   **HTTP GET v1/my\_events?token=<token>&before=<event id>&max=<number>**
   1. Get 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 v1/my\_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 v1/ my\_friends?token=<token>&timestamp=<timestamp>**
3. Create event by me with friend=friend\_id and activity=activity\_id  
   **HTTP POST v1/events?token=<token>  
   friend=<friend\_id>  
   activity=<activity\_id>  
   prayer\_requested=<true|false>  
   [new\_threshold=<threshold enum>]**
4. Add my friend  
   **HTTP POST v1/friends?token=<token>  
   first\_name=<first name>  
   last\_name=<last name>  
   photo=<encoded>  
   thumbnail=<encoded>  
   email=<email address>  
   mobile=<mobile phone number>  
   threshold=<threshold enum>**
5. Delete my friend  
   **HTTP DELETE v1/friends/<friend\_id>?token=<token>**
6. Get all activities in the system. This will return an array of activity records, plus the server’s timestamp. This does not return category records.  
   **HTTP GET v1/activities?token=<token>**
7. Get all activities in the system applicable to threshold=threshold\_enum. This will return an array of activity records, plus the server’s timestamp. This does not return category records.  
   **HTTP GET v1/activities?token=<token>&threshold=<threshold\_enum>**
8. Get all activities in the system [applicable to threshold=threshold\_enum] according to 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. This does not return category records.  
   **HTTP GET v1/activities?token=<token>[&threshold=<threshold\_enum>] &timestamp=<timestamp>**
   1. Timestamp is UTC based, and it is a timestamp originating from the server.
9. Get all categories. In the system This will return an array of category records, plus the server’s timestamp.  
   **HTTP GET v1/categories?token=<token>**
10. Get all categories in the system ccording to the timestamp. This will return all category records if any change has been made to any of the category records after the timestamp. If no change has been made to any category records after the time stamp, no records are returned. The server’s timestamp is also returned.  
    **HTTP GET v1/categories?token=<token>&timestamp=<timestamp>**
    1. Timestamp is UTC based, and it is a timestamp originating from the server.
11. Get earlier events for my friend=friend\_id. This will return an array of events.  
    **HTTP GET v1/my\_events?token=<token>&friend=<friend id>&before=<event id>&max=n**
    1. Get the 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.
    2. <friend\_id> must be my friend.

## Groups Screens API Calls

1. Get my groups. Returns group records and the server timestamp. Does not return user records.**HTTP GET v1/my\_groups?token=<token>**
2. Get my groups according to the timestamp. If any change has been made to my groups since the timestamp, return all my groups. If no change has been made to my groups since the timestamp, no records are returned. The server timestamp is also returned. Does not return user records.**HTTP GET v1/my\_groups?token=<token>&timestamp=<timestamp>**
3. Get users in my group. Returns a list of user records.   
   **HTTP GET v1/my\_groups/<group\_id>/users?token=<token>**
   1. <group\_id> must be one of my groups
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 v1/my\_groups/<group\_id>/users?token=<token>>&timestamp=<timestamp>**
   1. <group\_id> must be one of my groups.
5. LOWER PRIORITY: Get all the users in all my groups. Returns a list of user records.  
   **HTTP GET v1/my\_groups/0/users?token=<token>**
6. LOWER PRIORITY: Get all the users in all my groups according to the timestamp. If any change has been made to any of the users of any of my groups since the timestamp, return all the users in all the groups. If no change has been made to any of the users of any of my groups since the timestamp, no records are returned. The server timestamp is also returned.  
   **HTTP GET v1/my\_groups/0/users?token=<token>>&timestamp=<timestamp>**
7. Remove user from my group   
   **HTTP DELETE 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
8. Create my group

**HTTP POST 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 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. Create a request for a person to join my group. This will send an email or SMS to the invited person, asking them to join. See explanation *Becoming a New User.*  
   **HTTP POST v1/requests   
   token=<token>  
   group=<group\_id>  
   type=INVITE  
   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. The group must not already have max number of users, which is a server configuration parameter.
3. Create a request to join a group. This request will go the group’s leader, who may accept or reject my request.  
   **HTTP POST v1/requests   
   token=<token>  
   group=<group\_id>  
   type=REQUEST\_MEMBERSHIP**
   1. I must not already belong to the group.
   2. The group must not already have max number of users, which is a server configuration parameter.
4. Get requests sent to me, ordered by request\_id.  
   **HTTP GET v1/my\_requests?token=<token>**
   1. Returns a list of request records
5. Handle an existing request to join a group.  
   **HTTP POST v1/requests/<request\_id>/process  
   token=<token>  
   accept=true | false  
   user\_id=<user\_id>  
   group\_id=<group\_id>**
   1. Check that the request\_id points to a valid request.
   2. Ensure that the user is not already in the group.
   3. If the request is valid and accepted is true, add the user to the group.
   4. If the request is valid, delete the request (whether or not accepted is true).
6. Change the group leader to the given user.  
   **HTTP PUT v1/groups/<group\_id>/leader/user\_id   
   token=<token>**
   1. I must be group leader.
   2. The new group leader must already belong to the group.
7. Search all groups in the system.  
   **HTTP GET v1/groups?token=<token>&query=<query string>**

Return all group records that match the query string, either in the group name field or in the group description field.

## Reading Images API Calls

1. Get image from the Friend record.  
   **HTTP GET v1/my\_friends/<friend\_id>/image?token=<token>  
   HTTP GET v1/my\_friends/<friend\_id>/thumbnail?token=<token>**
   1. <friend\_id> must be one of my friends.
2. Get image from the User record.  
   **HTTP GET v1/my\_users/<user\_id>/image?token=<token>  
   HTTP GET v1/my\_users/<user\_id>/thumbnail?token=<token>**
   1. <user\_id> must be one of users in one my groups.
3. Get image from the Activity record.  
   **HTTP GET v1/activities/<activity\_id>/thumbnail?token=<token>**

## Settings Screen API Calls

1. Get my User record  
   **HTTP GET v1/my\_users/<user\_id>?token=<token>**
   1. <user\_id> must be me or one of the users in my groups.
2. Set my User record  
   **HTTP PUT v1/users/<user\_id>  
   token=<token>  
   first\_name=<first name>  
   last\_name=<last name>  
   photo=<encoded>  
   email=<email address>  
   mobile=<mobile phone number>**
   1. <user\_id> must be me

## Other API Calls

1. LOWER PRIORITY: Get System Settings from the server. Returns a list of name/value pairs. To Be Determined.  
   **HTTP GET 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

## Becoming a New User – Usual Case

The usual way that people become new users is when they receive an invite in their email (or SMS). Here are the steps this happens.

1. A group leader wishes to add a new user to their group. Using their mobile device, they choose the “Invite New Group Member” feature, giving the email (or SMS) of the new person.
2. This results in a **POST requests** **(type = INVITE)** call to the server.
3. The server sends a translated email/SMS to the person, describing the app and the invite, and telling the user how to download the app. The server also makes a request record in its database, recording that the group leader of a certain group invited a person with given email/SMS.
4. When the person runs the app the initial time, they register a new user account using the **POST register** call to the server.
5. After the new user is registered, the mobile device asks for any requests for the user using the **GET** **my\_requests** call to the server. (At this point, the only requests to the user will be invite requests.) If there are invite requests for the new user, the user will be given a choice whether or not to become a member of the group. If the user chooses to accept the invitation, they will become a member of the group by the **POST requests/<request\_id>/process?accept=true** call sent to the server. If the user rejects the invitation, a **POST requests/<request\_id>/process?accept=false** call is sent to the server.
6. When the server receives the call, the server will check for a valid invite request, and then add the user to the group.

## Becoming a New User – Special Case

The less common way that a person becomes a new user is by downloading the app and running it (without receiving any invitations). The app will ask them to *log in* or to *register*.

* If they choose to *log in* (perhaps the user deleted their app by mistake)
  + They will provide their username and password, which will be used by the **POST authenticate** call to the server. If this call succeeds, the app will receive the user’s security token and information.
* If they choose to *register* (more common case)
  + They will provide their user account information, which will be used by the **POST register** call to the server. If this call succeeds, the app will receive the user’s security token and information. The user will not belong to any group.

## 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 v1/activities?token=<token>&threshold=<seeking>**

This will return all the activity and category 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 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 events, which are generated when users do an activity with a friend.

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

* Event id=12345, user=U1, friend=F1, activity=37, pledge\_count=0 *(pledge count starts at zero)*
* Event id=12346, target event=12345, pledge\_count=1 *(pledge count now one)*
* Event id=12347, target event=12345, pledge\_count=2 *(pledge count now two)*
* Event id=12348, target event=12345, pledge\_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:

* Event id=12345, user=U1, friend=F1, activity=37, pledge\_count=3 *(current pledge count is three)*
* Event id=12346, target event=12345, pledge\_count=1 *(current pledge count is three)*
* Event id=12347, target event=12345, pledge\_count=2 *(current pledge count is three)*
* Event id=12348, target event=12345, pledge\_count=3 *(current pledge count is three)*

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

REST API Listing

Session

1. POST v1/register
2. POST v1/authenticate

Events

1. GET v1/my\_events?token=<token>&after=<event\_id>[&max=n]  
   GET v1/my\_events?token=<token>&before=<event\_id>[&max=n]
2. POST v1/events/<event\_id>/pledge?token=<token>
3. GET v1/my\_events?token=token&friend=<friend\_id>&before=<event\_id>&max=n

Friends

1. GET v1/my\_friends?token=token[&timestamp=<timestamp>
2. POST v1/friends?token=<token>
3. DELETE v1/friends/<friend id>

Activities

1. GET v1/activities?token=token[&threshold=<threshold enum>][&timestamp=<timestamp>]
2. GET v1/categories?token=token[&timestamp=<timestamp>]

Groups and Users

1. GET v1/my\_groups?token=<token>[&timestamp=<timestamp>]
2. GET v1/my\_groups/<group\_id>/users?token=<token>>[&timestamp=<timestamp>]
3. DELETE v1/groups/<group\_id>/users/<user\_id>?token=<token>
4. DELETE v1/groups/<group\_id>?token=token
5. PUT v1/groups/<group\_id>/leader/<user\_id>
6. GET v1/groups?token=<token>&query=<query string>
7. POST v1/groups?token=token

Requests

1. POST v1/requests?token=<token>&type=INVITE
2. POST v1/requests?token=<token>&type=REQUEST\_MEMBERSHIP
3. GET v1/my\_requests?token=<token>
4. POST v1/requests/<request\_id>/process *this is new, will also delete request*

Images

1. GET v1/my\_friends/<friend id>/image?token=<token>
2. GET v1/my\_friends/<friend id>/thumbnail?token=<token>
3. GET v1/my\_users/<friend id>/image?token=<token>
4. GET v1/my\_users/<friend id>/thumbnail?token=<token>
5. GET v1/activities/<activityid>>/thumbnail?token=<token>

User Record

1. GET v1/my\_users/<user\_id>?token=<token>
2. PUT v1/users/<user\_id>?token=<token>