USER-TO-USER MESSAGING

1. MESSAGING AUTHORIZATION PROCESS. Begins with authorization-to-message (ATM):

1.2. ask to message operates on a one-time authorization. If successful.

1.2.1 Requires proof of phone ownership, TouchID / Local Authentication, https://developer.apple.com/reference/localauthentication?language=objc

1.2.2 proof of phone to phone number access (via text message to phone)

1.2.2.1 user sends their phone # to our ML, which issues an SNS in reply, with an access code, randomly generated.

1.2.2.2 must send the user's SNS ARN, userid

1.2.2.3 ML returns a short code used to store locally, if they are in fact in recept of our text message to their phone #

1.2.2.4 User returns, via SDB, an authorization, creating them inside the `authenticatedUsers` domain

1.2.3 then offer a state-issued identity card.

1.2.3.1 Card data is combined with authorization from 1.2.2.4, updating their row in the `authenticatedUsers` domain

1.2.3.2 this creates the first verified identity moment in the lifecycle of the app. This row in `authenticatedUsers` represents their ID,

phone number, address, full name, age, DL # and userid, connected. This data is taken offline and stored internally in a later process.

1.3. when authorized (from 1.2), you are checked for ticket purchase for that event, locally (consumes an eventid)

1.4. then checks for a user-to-user (u2u) authorization string. IOS maintains a local string connecting you to another via userid, in this format [userid1 -> userid2]. This string is located in the `u2uAuthorizations` text file or related core data domain.

1.4.1 if the u2u string is missing in the local authorization database (or file), the user must move on to 1.5

1.5. acquire authorization from a user. Touch inside their profile page (accessed via Guest List) to request to message them.

1.5.1 A background request is logged within our `u2uAuthorizations` domain in SDB as a REQUEST type (we log REQUEST, APPROVAL, REJECTED), two different rows for each authorization request. Userid of both parties is captured, with timestamp, and a unique identifier is generated locally and sent as ItemID() -- this is the id of the authorization request.

1.5.2 ML loops through every 30 seconds to notify that user of an authorization request (so-n-so is asking to message with you on TR).

1.5.3 ML sends an SNS, with keys sufficient to trigger an authorization page, like you would see when inviting a promoter. The requested can see your information: that you are a verified user, age and initials, with pimg. They touch on an activation button, sending a confirmation back to SDB. Our ML picks this up and sends an ok SNS message back to the requestor, via ARN.An authorization request id is sent out, as generated by the requesting user (as itemID()). This authorization ID is used to make a new row encoded as REJECTED or APPROVAL.

1.5.4 ML checks for an additional key, `sent` in `u2uAuthorizations` domain. If the request was APPROVED and `sent` is NO, the approvee is sent an SNS.

1.5.5 The SNS from 1.5.4 opens an approval page in the approvee's IOS, as stated in 1.6

1.6. Approval Notification Page appears, as detailed in 1.7

1.7. An acceptance page pops up in response to the request, with the acceptor's picture, initials and age.

1.8. The userid of the acceptor is written to the `u2uAuthorizations` text file like so, indicating approval to communicate. in this format [userid1 -> userid2]. This string is located in the `u2uAuthorizations` text file or related core data domain.

1.9. A record of all authorizations is stored within the `u2uAuthorizations` domain. Cancels, bans, and approvals are set there. (Bans are issued by our company). Thus, if they reinstall the app, the authorizations file can be recreated, consulted online. Authentication can also be proven, again via the same process (phone #, DL showing)

1.10 the authorizations file should be augmented with the ARN of the approved u2u partner, which can be sought via lookup in SDB.

2. Messaging protocols (u2u)

2.1 The protocol possesses three parent objects:

2.1.1 a Message: eminating from one user to another (to/from), with timestamp, message body.

2.1.2 a Thread, which takes place when one messages receives a reply

2.1.3 a MessageUI, which is a UICV one cell per row, as in feed2. The Message appears in a bubble of text in a UITextView, which has variable height, embedded in a cGRect, measured to accomodate the Message's body. The photo of the sender or receiver appears. The sender's profile image (pimg) appears in the left side of the cell, message locked right. The receiver's pimg appears on the right side, to visualize the exchange, but their UITextView locks to the right of the screen in italic text. Thus, MessageUI appears as a checkerboard or messages and replies. A reply text field appears at the bottom, with keyboard enabled

2.2 MessagesUI data source is written via SNS/unpack in AZC. It contains the to/from in sequence, after you send and/or receive, thus painting the log of messages in order. Data source is read, then per row, if the log is sender->receiver (messages you sent), the cell appears with your picture on the right, receiver's picture on the left, with your message on left, their message reading right. Thus, for each row of communication, a cell appears corresponding to sender or receiver.

2.3 MessagesUI uses 2 types of messages: a received message and a sent message. Each one takes the data source row as input, integrates pictoral and message data from your existing authorization files, then paints the message ui.

2.4 A Message Queue is a UITV with the sender's name and an accessory. It represents each message received, initially without replies. Once a message receives a follow-up reply from either user, it becomes a thread. Threads refer to messages with replies or single, unreplied messages. Hence, when messages are replied to, they are sent into SNS with the message head (initial message id) encoded, to be directedd into a MessageUI sequence.