Contxt – iPhone App High Level Design

# Purpose

The purpose of this document is to provide an overview of the structure of the iPhone app and to give insight into the workflow, as well as point out specific details for design choices.

# Domain Model

The contxt iPhone app uses Core Data for storing and manipulating data within the app. The model can be viewed within XCode, but below is a screenshot of the model. All objects except for AnnotationPoint and AnnotationSize extend from Object. Those two specific classes were created to make storing points easy, but there is no value in having “keys” or shared object data for them.

The Project class was originally created with the idea that there may be multiple projects, each with many “starting images” that contain embedded images, conversations, markups, or any combinations thereof. This was cut from scope, but significant code had been written to handle this concept. Thus, it remains and is available for use by future revisions of the app. When the app launches for the first time, it creates the concept of an “Untitled Project” as a starting project. In the current revision (1.0), that is the only Project class used, and users will not know the difference.

*Object* – Abstract parent object that contains data relevant to almost all other data types in the object model.

*AnnotationDocument* – This is the highest level object (besides project) utilized in the code. Everything starts with an AnnotationDocument, which contains any number of Annotation objects. Each instance must contain an ImageInfo object, as this is what the user sees in the “Images” list in the app.

*ImageInfo* – Represents a physical image on disk. Note: The image itself is not stored in the Core Data object. Rather, a path to the location on disk is maintained so that that image can be loaded into memory when used and purged when unused, rather than Core Data controlling when to let it go (along with unnecessary reads/writes from/to disk when searching and returning objects).

*Annotation* – This is an abstract class for the concrete annotation types. It captures the common attributes amongst all annotation types.

*ImageAnnotation* – Represents an embedded image as a pin. This is implemented by simply containing another AnnotationDocument.

* NOTE: An AnnotationDocument can have any number of ImageAnnotations, but an embedded AnnotationDocument within an ImageAnnotation can NOT have any ImageAnnotation objects. The purpose behind this decision was to keep from nesting multiple levels to avoid confusion for the user.

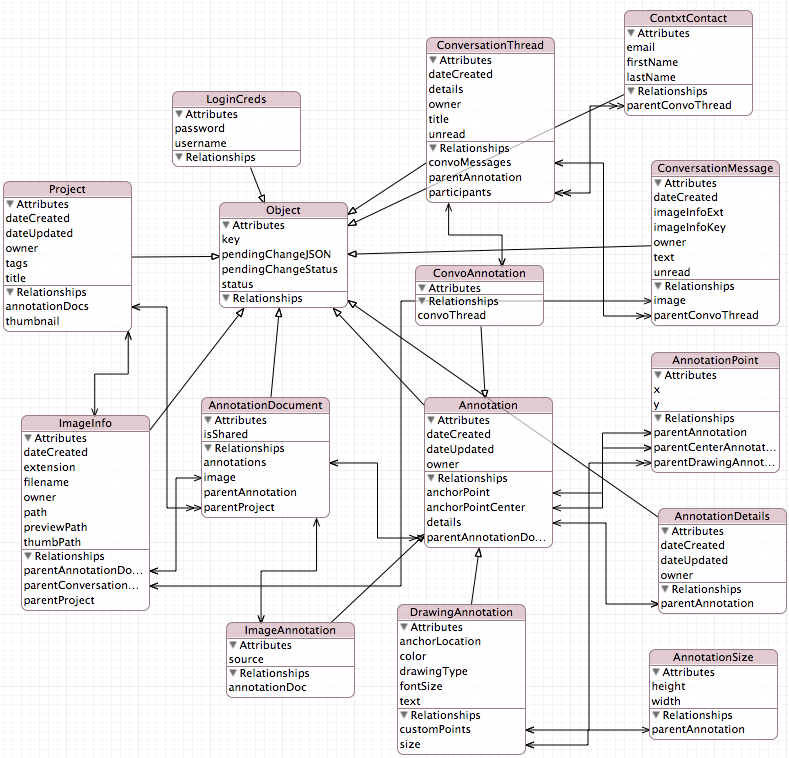
*DrawingAnnotation* – Represents all types of drawings that can be created within app. Those include Rectangle, Ellipse, Line, Leader Line, Dimension Line (with Text), Text, Text with Leader Line, Brush (freehand), and Pen (freehand). The combined concepts, such as “Text with Leader Line” are captured as two separate objects in code. For the custom (freehand) drawing types, AnnotationPoints are captured and stored in an ordered set.

*ConvoAnnotation* – Represents a conversion within an AnnotationDocument. It simply contains a ConversationThread object, which has the important/relevant information. The main purpose of this class was to keep convention of the <type>Annotation naming scheme for clarity.

*ConversationThread* – Represents the overall thread for the conversation. It contains details on who owns (created) the thread, a list of thread participants (as ContxtContact objects), and a list of ConversationMessage objects. Note that certain attributes are not used in this revision of the app (v1.0), such as details and title. These attributes were considered for scope at one point, but later cut from this version.

*ConversationMessage* – Represents the messages passed amongst participants. It can either have text or an ImageInfo object, but not both (though this is not constrained in code). Each message has an owner, and a read/unread status is tracked for this object.

NOTE: As recommended by Apple’s documentation, two-way relationships are tracked for objects.



# UI / Workflow

When first starting, the app loads either the signup screen (*SignUpViewController*) if first launch or no account has been created, or the camera control to take an image. The *AnnotationDocListVC* is the root view controller, but the user does not see this loaded when first launching.

From the *AnnotationDocListVC*, the user can perform a number of operations:

* Launch the Photo Library Import tool to import a photo
* Launch the Camera Control to take a photo
* View a list of all conversations (in all AnnotationDocuments) by touching the chat icon. This launches the *ConvoListViewController* class, which determines what messages to load based on which controller pushed this controller.
* View details for a specific AnnotationDocument by touching the image for that that document.

From the *AnnotationViewController*, the user can also do a number of operations. There are two pages to the *AnnotationViewController*:

1. *AnnotationPinView* – This view contains pins for ImageAnnotation and ConvoAnnotation objects.
   1. The user can touch and hold to drop a pin, which then allows the user the option of taking a new image, importing an image from the photo library, or starting a new conversation thread.
   2. For existing image pins, the user can launch an embedded *AnnotationViewController* by touching an image pin, or launch the *ConvoViewController* by touching on a conversation pin.
2. *AnnotationDrawingView* – This view contains all drawings associated with the selected AnnotationDocument.
   1. The user can create any number of DrawingAnnotation objects.
   2. The objects are drawn using the DrawingView class or the DrawingFreehandView class, which is a subclass of DrawingView.

* NOTE: The *AnnotationPinView* and *AnnotationDrawingView* classes contain instances of the *PZPhotoView* class, which handles all of the manipulation of the image that is shown (zoom/scroll/eventing).
* In either of these pages, the user can take the following actions:
  + Share the AnnotationDocument with other users (launching the *COPeoplePickerViewController*) by touching the share icon.
  + View a list of threads (most recent message in each) by touching the chat icon (launches the *ConvoListViewController*, which in this workflow will only show threads associated with the currently selected AnnotationDocument).

From the *ConvoListViewController*, a user can launch the *ConvoViewController*, which contains the actual ConversationMessage objects.

The *ConvoViewController* provides the conversation history and allows users to create new message (text or photo), or add or remove participants.

* Note: Only the owner of a ConversationThread (thread starter) can remove people from the list of participants.

# Data Controller

The *DataController* class handles all of the fetching and storing of NSManagedObjects from the persistent store. It provides methods for interacting with the primary MOC (ModelObjectContext, as NSMainQueueConcurrencyType), as well as for the call to supply their own MOC. This is significant with regards to server communication, which can alter the data. To avoid freezing the UI on the main thread, background communication is handled in separate threads, and private contexts are created for updating the model in background processing. Those changes are then saved to the main MOC (captured in the DataController in the “managedObjectContext” property) and observers notified of the changes.

The only methods in the *DataController* class that interact with the *ServerComms* class, and thus controlling data being published to the server are:

- (void)deleteAnnotation:(Annotation \*)annotation;

- (void)deleteAnnotationDocument:(AnnotationDocument \*)doc

The purpose behind allowing the DataController to handle this interaction is due to the fact that the other classes should NOT delete any objects prior to getting acknowledgement from the server that the delete has been processed. Otherwise, we may lose information by it being deleted from the persistent store, only to find out after the fact that something went wrong (like network connection failed) and the delete couldn’t be processed on the server.

\*NOTE: It is absolutely the responsibility of the other classes to handle saves of data and determining when to push to the server.

# Server Communication

## ServerComms Class

Communication to/from the web server is contained in the *ServerComms* class. The purpose of this class is strictly to handle all processing with the server using asynchronous calls. Rather than list all methods here (since all are equally important in this class), it seems more appropriate to list the types of calls this class processes:

* Login/Signup calls (validating credentials or signing up new users)
* Fetching (all “get” or pull operations from the server)
* Sharing (all “push” operations as they relate to simply sharing AnnotationDocuments or Annotations)
* Saving (all “push” operations as they relate to saving objects)
* Deleting (all “push” operations as they relate to deleting objects)

This will be documented in a separate API document for the App/Web Server communication, but communication should be acknowledged both ways (App to Server and Server to App) to confirm successful processing of a request.

When data is updated in the background, the ServerComms class notifies all observers of changes data so that the UI can take the appropriate action. For example, if a new message is received, the UI should change the “chat icon” to “new chat” icon to signify to the user that there is a new message to view. Another example would be if an DrawingAnnotation was moved by a shared user, and the update comes through while viewing that drawing. The drawing should be moved accordingly.

* **NOTE:** It is extremely important to update this data ASAP so that the user doesn’t run the risk of overwriting data or getting data out of sync in the MOCs, and thus causing a fault to occur in code in Core Data.

## Execution

The various methods are invoked at different times within the app, as noted below:

* When loading the AnnotationDocListVC, the following operations are executed:
  + Process pending deletes
  + Process pending changes
  + Check for new AnnotationDocument objects (shared from others)
  + Check for new ConversationMessage objects (for all threads, since this is the list of all AnnotationDocuments, not just a specific document)
* When importing an image or taking a new photo from the *AnnotationDocListVC*, the following operation is executed:
  + Save AnnotationDocument
  + Delete AnnotationDocument
* In the *AnnotationDrawingView* class
  + Save annotation – when annotation is created
  + Save annotation – when annotation is updated, be it color, text, or location
  + Delete annotation
* In the *AnnotationPinView* class
  + Save annotation – when creating a new ConvoAnnotation
* In the *AnnotationViewController* class
  + Share AnnotationDocument
  + Download ConversationThread for a specific ConvoAnnotation
  + Download embedded AnnotationDocument for a specific ImageAnnotation
  + Get all new/updated Annotation objects for the specified AnnotationDocument
  + Save annotation – when creating a new ImageAnnotation
  + DeleteAnnotation
* In the *ConvoViewController* class
  + Remove participant
  + Add participants
  + Save ConversationMessage (including image, if applicable)
  + Delete Annotation (self)

## Integrity

The integrity of data in app is preserved by keeping the app and server in sync via acknowledgements. More details will be provided in the API documentation. However, the following should be noted from an iPhone App developer perspective:

* If a push of data to the server is successful, the object maintains an OBJ\_STATUS\_SAVED state captured in the “status” field of the *Object* class. No future considerations are required.
* If a push of a created or updated object fails to return success from the server, the object maintains an OBJ\_STATUS\_PENDING state captured in the “status” field of the *Object* class. This request is attempted again when the AnnotationDocListVC is loaded (by calling processPendingChanges).
* If a push of a deleted objects fails to return success from the server, the object maintains an OBJ\_STATUS\_DELETED state captured in the “status” field of the *Object* class. This request is attempted again when the AnnotationDocListVC is loaded (by calling processPendingDeletes).
* If a push of adding or removing a participant from a conversation thread fails to receive a successful status from the server, the state is tracked slightly differently that for other objects (as noted above).
  + Since the change could have been adding *or* removing a participant from a conversation, the specific request must be tracked. Additionally, more than one add or remove request could be pending at any given time for the same contact, so it must be captured. In this case, the “status” field is set to OBJ\_STATUS\_PENDING, the “pendingChangeStatus” field (also on *Object* class) is set to either OBJ\_CHANGE\_TYPE\_ADD or OBJ\_CHANGE\_TYPE\_DELETE, and the appropriate ConvoThreadKey and ContxtContactKey are stored in an JSON object in the “pendingChangesJSON” field of the *Object* class.
  + When the update is process, that particular part of the changelist is removed from the “pendingChangesJSON” field.
  + Once all changes for a particular contact have been processed, the “status” is set to OBJ\_STATUS\_SAVED, and the “pendingChangeStatus” is set to OBJ\_CHANGE\_TYPE\_NONE.
* If a pull from the server is attempted, but the request could not be completed (such as for insufficient data from the server), no acknowledgement is sent to the server. Thus, the server should not clear that “update” or “change” request and should submit again next time it is asked.