ONLINE COMMUNICATION USING PHOTOGRAPHS

This document denotes an architecture design for a online website on which many users annotate the photos with circles and then discussed on these annotations using the chat threads

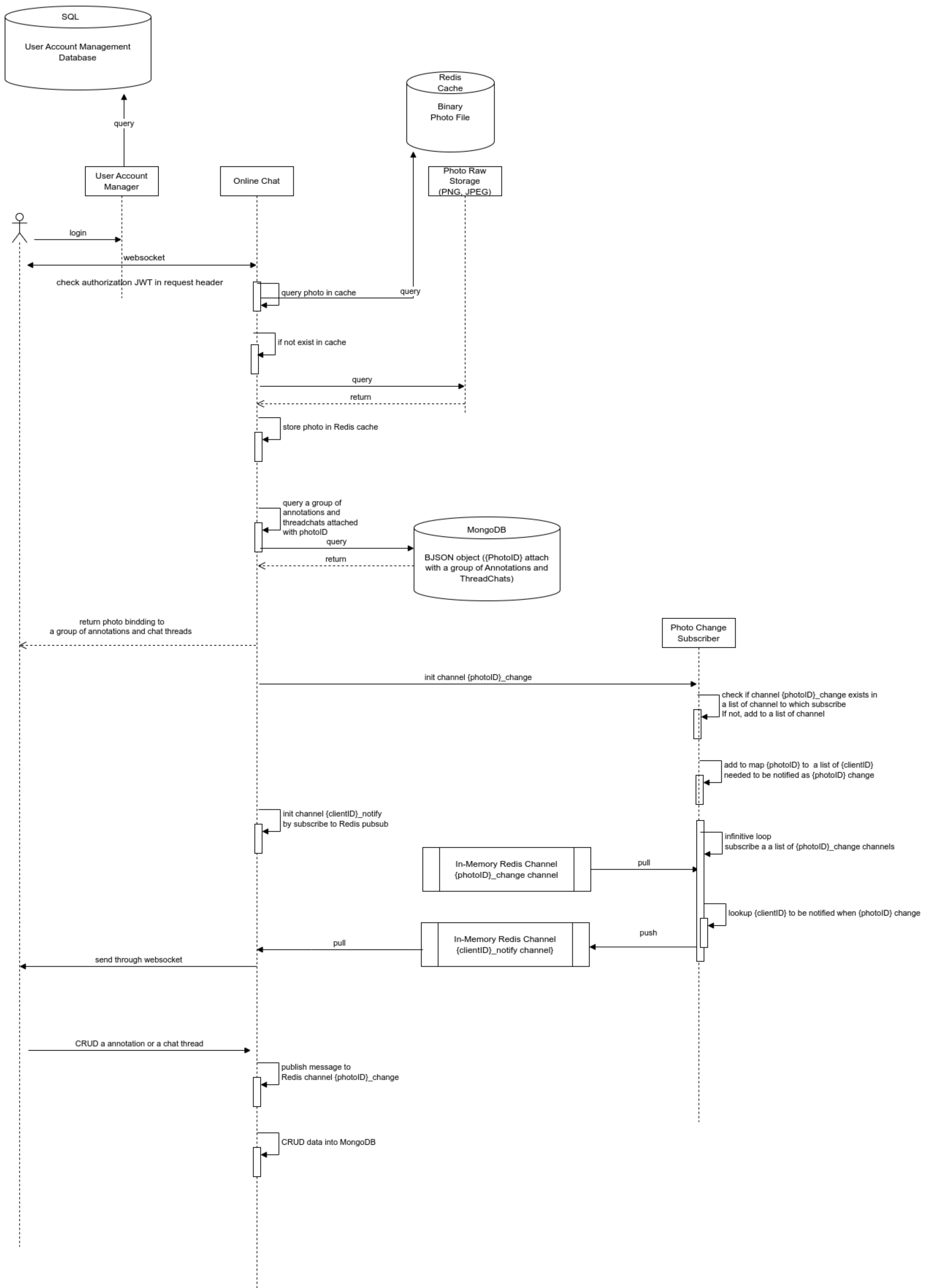
A list of main considered functions:

1. Only authenticated and authorization users can open and write on the photos which uploaded to the private storage
2. After opening the photo on the website, users want to see real-time updates from others who write on the same photo.
3. Many (up to many hundreds of) users read and write on the shared photos simultaneously

Brief proposal for any main non-functions:

1. Security:
   - authenticate users with registered username/password
   - authorize users with role-based access
   - a limited access time to private photos by JWT token which need to refresh after a few hours
   - The internal server has a service account to which the raw storage service grants that can download private photos.
2. Coding techniques are approached in this app:
   - Retrieving from database and disk is slow and expensive, use in-memory cache instead.
   - Sticky sessions in a gateway context direct all requests from a specific client to the same server for the duration of their session. It is to keep reuse already-established bidirectional websocket connection between client and server
   - Reactor pattern (another name call asynchronous response) (example here is simply in-memory publisher/subscriber) for handling multiple of simultaneous requests
   - Pooling connection queues between server and database is to prevent from creating a new expensive connection to database
   - Pooling thread of handing a list of live connection from client and server, if one of them is close, idle thread is used for a new connection
   - Enabled Health-check, Circuit breaker reroute requests away from unhealthy server instances. Retry for timeout request, Fallback uses a temporarable accommodation. Rate-Limiter controls the flow of requests to prevent overload and protect resources from excess use.
   - Hacking your code that intentionally makes 1% incoming request fail and see how other services or websites react against this issue

The main flow of a online communication through a photo by a group of users


SQL
User Account Management Database
query
Redis Cache
Binary Photo File
User Account Manager
Online Chat
Photo Raw Storage (PNG, JPEG)
login
websocket
check authorization JWT in request header
query photo in cache
query
if not exist in cache
query
return
store photo in Redis cache
query a group of annotations and threadchats attached with photoID
query
return
MongoDB
BJSON object ({PhotoID} attach with a group of Annotations and ThreadChats)
return photo bindding to a group of annotations and chat threads
Photo Change Subscriber
init channel {photoID}_change
check if channel {photoID}_change exists in a list of channel to which subscribe If not, add to a list of channel
add to map {photoID} to a list of {clientID} needed to be notified as {photoID} change
init channel {clientID}_notify by subscribe to Redis pubsub
infinitive loop subscribe a a list of {photoID}_change channels
In-Memory Redis Channel {photoID}_change channel
pull
lookup {clientID} to be notified when {photoID} change
push
In-Memory Redis Channel {clientID}_notify channel
pull
send through websocket
CRUD a annotation or a chat thread
publish message to Redis channel {photoID}_change
CRUD data into MongoDB

Brief description in the above diagram:

- [ ] Server checks the validation of JWT token in the client request header for security and then a websocket between client and server is made that helps the server send live changes of an opened photo to client.
- [ ] If the original content of the photo is not found in cache, get it from a disk storage. The binded annotations and chat threads are stored in tree-like BSON format in MongoDB also returns to make a full presentation of photo for a user
- [ ] The changes made by the user are published to in-memory channel {photoID}_change, and then the Photo Change Subscriber services will receive and broadcast these changes through channel {clientID}_notify to servers which keep a list of connections to related users. Like above talk, server actively send to client by websocket
- [ ] The annotations and chat threads are store in MongoDB for later retrieval because in-memory channel don't store messages

Tech Debts:

1. How to enable folder access permission in the Raw File Storage
2. If the data size of annotations and thread chats of a photo is small, prefer to store in Redis than in MongoDB for swiftly retrieving
3. If a business allows infinite nested reply messages in one message, should we change MongoDB, instead Recursive Query in SQL for persistent storage and LinkedList in Redis cache?
4. In-memory Redis publisher/subscriber sometimes lost messages
5. Server keeps a list of Redis channels in RAM. Therefore, if the server crashes, the list is lost. Maybe store a list of active channels in cache
6. If the connection between client and server interrupt inadvertently during a session, are some annotation and chat threads lost if restart a new session