**CHAT SYSTEM WORK FLOW AND IMPLEMENTATION DETAILS**

**Requirement:-**

* Protocol used for instant message broadcasting is Stomp V1.2
* Message Broker used for reliable message delivery is RabbitMQ
* Database used for chat and associated data storage is MonogDB
* Backend used is Spring MVC websocket with Stomp subprotocol support

**Algorithem:-**

Following is the step by step algorithem to create the chat system in client side:

* Create a Stomp Configuration object with all the necessary details like heartbeat settings, connection timeout settings sockjs factory settings and all the necessary field values
* Create a Stomp Client object with the configuration object as parameter in the above step
* Create and maintain a flag variable with initial value true which tells whether the **onConnect()** callback of stomp client is triggered by activate function of the client object or because of a reconnection
* Activate the connection with **activate()** function of the client object.
* Once connected manipulate the **onConnect()** callback as following if the global flag is true(first-connection):  
   🡪create a ClientObject with empty waiting queue, empty user cache, empty chat cache.  
   🡪call **init()** which will send an http request to the server and fetch all the chat information of the user and populate the client object attributes(id, cache etc).  
   🡪render the UI.  
   🡪call **createSubscriptions()** which will subscribe to the default destination of the user and then to all the chat destinations.  
   🡪iterate through all the messages that are received from the initial http request and for each of them call **sendDeliveryAck()** to notify the sender of the message that the message is delivered to the user by sending a message to their corresponding default destination.  
   🡪update the global flag to false.
* Once Connected manipulate the **onConnect()** callback as following if the global flag is false(re-connection):

🡪call **reinit()** which will send an http request to server and get all the updated information of all the chats that the user is already connected to like any new messages arrived during the reconnection process or other users online status etc and update the local cache.

🡪render the UI according to the updated information of the existing chats along with new chat.

🡪 call **createSubscriptions()** which will subscribe to the default destination of the user and then to all the chat destinations.

🡪iterate through all the messages that are received from the earlier http request and for each of them call **sendDeliveryAck()** to notify the sender of the message that the message is delivered to the user by sending a message to their corresponding default destination.

🡪dequeue all the messages from the waiting queue and resend them and enqueue all of them once again.

🡪if a chat window is open right now and if any new messages are fetched from the earlier http request corresponding to the focused chat then don’t perform step 4 for those messages instead call **sendSeenAck()** for those messages to the senders to notify that the message is seen by the user and then render those messages to the chat window.

* When subscribing to the default destination following happens:  
   🡪server creates a non-durable, auto-delete queue with a random name.  
   🡪binds the queue with the name as the **defaultDestination.** 🡪creates a subscription of the queue with a randomly generated subscription-id.  
   🡪changes the online status of the current user to true in the database.  
   🡪returns the subscription-id to the client for further use.
* When subscribing to the chat destination following happens:  
   🡪server creates a non-durable, auto-delete queue with a random name.  
   🡪binds the queue with the name as the **chatDestination.** 🡪creates a subscription of the queue with a randomly generated subscription-id.  
   🡪returns the subscription-id to the client for further use.
* When message arrives to the default destination following happens when msg **type** is delivery:  
   🡪get the details of the msgId and chatId and update the msg cache by calling **ClientObject.getChatCache().getChat(chatId).getMsg(msgId).getReceiver().setStatus().** 🡪get the details of the userId, profilePictureLink, isActive, destination and update the user cache by calling **ClientObject.getUserCache().setUser().**  
   🡪render changes in the UI.
* When message arrives to the default destination following happens when msg **type** is seen:  
   🡪 get the details of the msgId and chatId attributes and update the msg cache by calling **ClientObject.getChatCache().getChat(chatId).getMsg(msgId).getReceiver().setStatus().** 🡪 get the details of the userId, profilePictureLink, isActive, destination and update the user cache by calling **ClientObject.getUserCache().setUser().** 🡪 render changes in the UI.
* When message arrives to the default destination following happens when msg **type** is new-chat:  
   🡪get the new chatId from the chat-id field and call **ClientObject.getChatCache().getChat(chatId).** This will get the chat information from the server and populate the cache with the response.  
   🡪create subscription to that chat destination.  
   🡪get all the new msgs and for each of them call **sendDeliveryAck().**  
   🡪render changes in UI.
* When message arrives to the specific chat destination following happens when a message arrives:  
   🡪immedietly acknowledge the message.  
   🡪get the message object from the msg attribute and chatId from the chat-id header and call the function **ClientObject.getChatCache().getChat(chatId).setMsg().** 🡪render changes in UI.  
   🡪if any chat window is open right now and the message received belongs to that chat then render the msg immedietly in the chat window and then call **sendSeenAck()** or else send **sendDeliveryAck().**
* when trying to send a msg following will happen sequentially when called **ClientObject.send()**:  
   🡪 a new MessageObject will be created from the raw text String.  
   🡪save the msg to the cache.  
   🡪save the msg to the waiting queue.  
   🡪publish the msg with a receipt header with a value equals to msgId.  
   🡪listen for the receipt frame for the particular msg(i.e where receipt-id header of receipt frame equals to msgId).  
   🡪render the UI.
* When a receipt is received following happens:  
   🡪get the receipt-id from the receipt which is nothing but the msgId of the msg that the receipt is for.  
   🡪delete the msg from the waiting queue corresponding to the msgId.  
   🡪update the cache to set the delivery mode of the message to 0(sent).  
   🡪render the UI.