CPSC 313: Distributed and Cloud Computing

Spring 2022

Lab 4: Message based chat – MVP2

Now that we have begun our foray into distributed services using a messaging queue, we can start to make the distributed chat more realistic. For now, we are calling each chat channel/queue a "room".

We are adding persistence, caching, and basic queue management. The basic idea is that for each room, we have an internal cached object that allows us to simplify the use of the room while all the details of handling both RabbitMQ and MongoDB interaction is implemented as part of the ChatRoom class.

Some notes about rooms and the ChatRoom class:

- ChatRoom(deque) – inherits deque from the collections module
- Constructor:
  - Takes room_name: str, member_list: list, owner_alias: str, room_type: int, create_new: bool
- Each room has a room_name, passed to the constructor
  - This name is the default queue_name for consuming if using messaging queues
- Each room has a int "room_type" designating whether it is private or public group or not
  - Constants: ROOM_TYPE_PUBLIC = 100, ROOM_TYPE_PRIVATE = 200
  - The difference between public and private rooms is whether or not you must register for the room and be approved.
    - Put another way, private rooms are whitelisted
  - In both cases there can be many-to-many groups
- Each room also has an owner, specified by owner_alias: str
- Each room has a member list. For a private queue it is this list that whitelists members and a user must be added without just trying to get messages.
  - You must have 'add_group_member' and 'remove_group_member' methods
- We'll need to know if a room should be saved/persisted, so we need a bool "dirty" flag
- The room should have a create_time and modify_time
- Each room has persist and restore methods that save and restore from mongodb.
  - Each room has its own collection, the name of which is the name of the room
  - You should have a document in the collection for the room metadata,
  - You should have a document in the collection for each message in the room
- **Maybe:** Each room should have a __retrieve_messages private method that gets messages from rabbitmq.
- Other methods:
  - get_messages(num_messages: int, return_objects: bool) -> list of ChatMessage
  - send_message(message: str, mess_props: MessageProperties) -> bool
  - find_message(message_text: str) -> ChatMessage
  - get() -> ChatMessage – gets the next message in the deque from the right
  - put(message: ChatMessage) -> None – puts message into the (left of the) queue

You should have a class for message properties: MessageProperties() that holds the following properties:

- mess_type: int - message type – either sent or received.
  - You should have constants for the type
- room_name: str - room to which the message belongs
- to_user: str – destination for message. Can be group exchange
- from_user: str – alias that is the sender of the message
- sent_time: datetime – timestamp for when the message was sent
- rec_time: datetime – timestamp for when the message was received
- sequence_number: int – number of the message in the sequence of messages

You should have a class called ChatMessage():

- property message: str - the actual chat message
- property mess_id: id returned by MongoDB after saving
- property mess_props: MessageProperties – the message properties
- property dirty: bool – flag for if an instance needs to be saved

I find it useful to have a RoomList class. You can either inherit list or create an internal private list that holds the list of rooms. Properties and Methods you should have:

- Name: the list name.
- Constructor: just takes name: str for the name of the list
- Either inherit from list or create an internal __rooms: list property
- add: add a new RoomChat instance to the list
- remove: remove a RoomChat instance from the list
- find: find a RoomChat instance by name
- find_by_member: return all chatrooms that have the alias as a member
- find_by_owner: return all chatrooms that have the alias as the owner
- persist and restore, saving the list metadata and restoring the list metadata
  - After saving or restoring the metadata, iterate through all rooms in the list calling their persist or restore methods

In your program, if we're using RabbitMQ you will be connecting to RabbitMQ (RMQ) on a globally available server. That server is:

IP: 35.236.51.203 port: 5672

You'll be connecting to a globally available MongoDB instance:

IP: 34.94.157.136 Port: 27017

Your API should be roughly the same, though we will use the concepts of room instead of queues.

The beginning of this process is simply to create an API that exposes the following endpoints/methods:

- Send: accepts a message and a sends it to RMQ

    - POST http://<url>:<port>/message/?room_name=<name>&message=<your message>&from_alias=<alias>&to_alias=<alias>
- Messages: returns the list of messages currently in the queue
    - GET http://<url>:<port>/messages/?room_name=<your room>&messages_to_get=<int>
    - This method returns a list of messages:
        - ["test message 1", "test message 2", "another message", "and so on"]

You should have a test file room_test.py that has good tests for your ChatRoom and RoomList classes.

If you are using messaging, you will be using the "pika" (pip install pika) library to connect with RMQ. For a client-side HTTP library you should probably use either "requests" or "xhttp". There are others, and you may use whichever library works best for you.

You will create a single python code file named "room_chat_api.py" that implements the API. You should also have a python code file named "room_chat_test.py" that tests your API. Your third file will be "rmq.py" and will contain the implementation for RMQ interaction.

- Both unittest and pytest are good test frameworks for python. I tend to use both.
- To run the tests, you can use appropriate commands in your powershell or terminal
- However, I strongly recommend you use VSCode to run your tests. It's easy to integrate either pytest or unittest (or both) into VSCode to run your tests.
- You should be getting used to using postman (postman.com) to help you manually test your API's