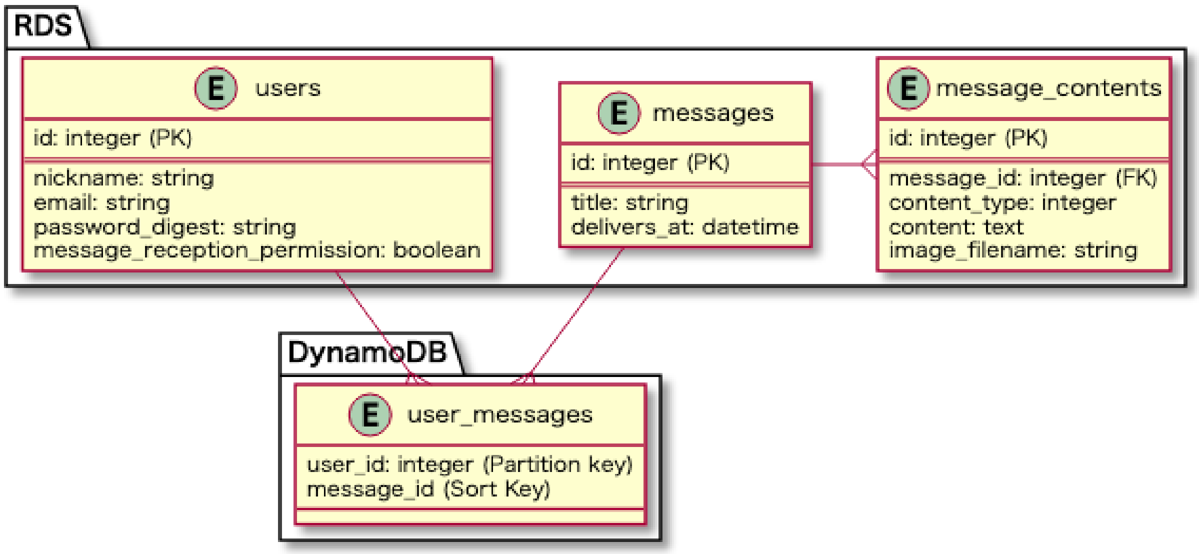
EXISTING TABLE



user_messages [adding following fields to the table]
message_sent_status: string [This field will update the status of messages i.e successfully delivered or not] {Values can be set at New, Pending, Sent, Failed}
message_sent_notification: string [This field is to add any notification in case of failure message]

New Table to be added : {Mainly for future purpose like sending individual message to a user or group}

message_group [creating new group for future mapping]
id: integer(PK)
name: string [This is to give group name for identification]
category_id: string [Can be referenced for categorization when we will have many group records : for example type of cuisine, type of message, send notices, special offers, campaign information etc.
|| Subsequent Sub Category will be map for further classification like indian, japanese, thai etc (type of cuisine) or promotional, transactional (type of message)]

user_group [For mapping of users to a specific group
user_id: integer(Partition Key)
group_id: Sort Key

message_group [For mapping of messages to a specific group
user_id: integer(Partition Key)
group_id: Sort Key

[Please note : I have seen in the Kurashiru website that we can search recipes by category. We will integrate it for mapping to groups. If the feature is not extensible then we create the following table]
message_group_category [New category for group mapping]
id: integer(PK)
name: string [Category name]

message_group_subcategory [New sub category associated with each group category]
id: integer(PK)
name: string [Sub Category name]
category_id : integer (Foreign Key)

MANAGEMENT SCREEN FLOW

CREATE NEW MESSAGE SCREEN :

Management can create new messages through this screen. The user interface will have the standard input screen where management can either upload images and/or add message content. In case of images we would be using **AMAZON S3 BUCKET SERVICE (Cloud service to store data)** to store the image and will update the s3 image url in our message table.

Future Scalability : Currently since we are sending messages to all the users who are eligible, we can add the schedule time feature here.
When we extend the feature to send messages to user or group, then we would have a mapping here with group/user which will identify generic messages or targeted messages.

LIST MESSAGES SCREEN :

Management can view all the list of messages created through this screen.

Future Scalability : (a) Management can edit the user/group to which the message is to be send and can also modify the schedule time provided the current time is less than the scheduled message time. (b) can download the logs/status messages for the messages that are sent to user.

CREATE CATEGORY/SUBCATEGORY/GROUP SCREEN : [Future Enhancement]

Management can create new group, category and subcategory for messages. Management can further map the user to each group, can update category and subcategory for the groups.

MESSAGING SERVICES [eg RabbitMQ / Kafka]

PRODUCER :

The producer daemon/cron will be responsible for preparing the message to be sent to the users. It will perform the following steps :

1. The producer will check the database for any messages that are scheduled to be sent.
2. It will first select all user that are eligible for receiving messages (i.e ***message_reception_permission*** is set to true) [*Future Enhancement: In case of group mapping, the pool of users will be selected from the group and then checked whether they are eligible for receiving messages.*]
3. Finally create a new record in the ***user_messages*** table.

CONSUMER :

The consumer daemon/cron will be responsible for sending the messages to the user. It will check all the records from the user_messages table whose ***message_sent_status*** is New and then send it to the user. On successful completion, the ***message_sent_status*** will be updated to Sent, and in case of failure ***message_sent_status*** will be updated to Fail and the corresponding **message_sent_notification** will be updated with the Error Message.

API SERVICES [The code can be written in existing Rails structure or we can create a complete new repository]

User_Unsubscribe_Messages_API :

This API will be invoked when a user opts for no messages from kurashiru messages function. It will update the ***users*** table and update ***message_reception_permission*** to false.
If we have an existing API system in Kurashiru, we will add the services to the existing module. Else in case of a new development we will create a new api module and create the services there.

Message_Not_Sent_Failure_API [Failure Detection API]:

This API will be invoked when the messages are not sent to a user. We can define a group of user or mail IDs who can receive such mails to debug the issue where messages are not sent to a user. This scenario may arise if the Messaging Service [RabbitMQ or Kafka] is not working or maybe the email of the user is not correct.

Backup_Messages_Service_API :

This API should be developed in case the Messaging service is not working properly or is down for some reason. In this case we can use backup messaging services {eg AMAZON SNS} which will be used to send the messages to the user as a backup.

TECHNOLOGY / SERVICES USED

Amazon S3 : It is reliable, low cost, easily migrated and very simple to use. The API's are very easy to integrate into an existing system.

RabbitMQ / Kafka : Both the services can cater to large data. It is assumed that we have 50 million users in the database and it can be easily handled by both the services.

API's : Creating them in Ruby on Rails or any backend language like PHP and then deploying it on the existing AWS Server.

POINTS TO NOTE

- **We should try to use the existing Kurashiru functions to avoid code rewriting.**
- **We should use indexes in our database as we are dealing with large datasets and should focus on efficiency and performance.**
- **Logging to be done to check all failure scenarios (Ruby Logger).**
- **Server side alerts for checking Daemon and Cron running status. Fail-safe to be added to avoid functionality stop.**