DB-based email service

Team:

Annam Saivardhan, 200050008

Cheerla Vinay Kumar, 200050027

Dendukuri Sandeep Varma, 200050032

Dwarapudi Harshavardhan, 200050038

Rahul Dathathreya G, 180050034

Overview:

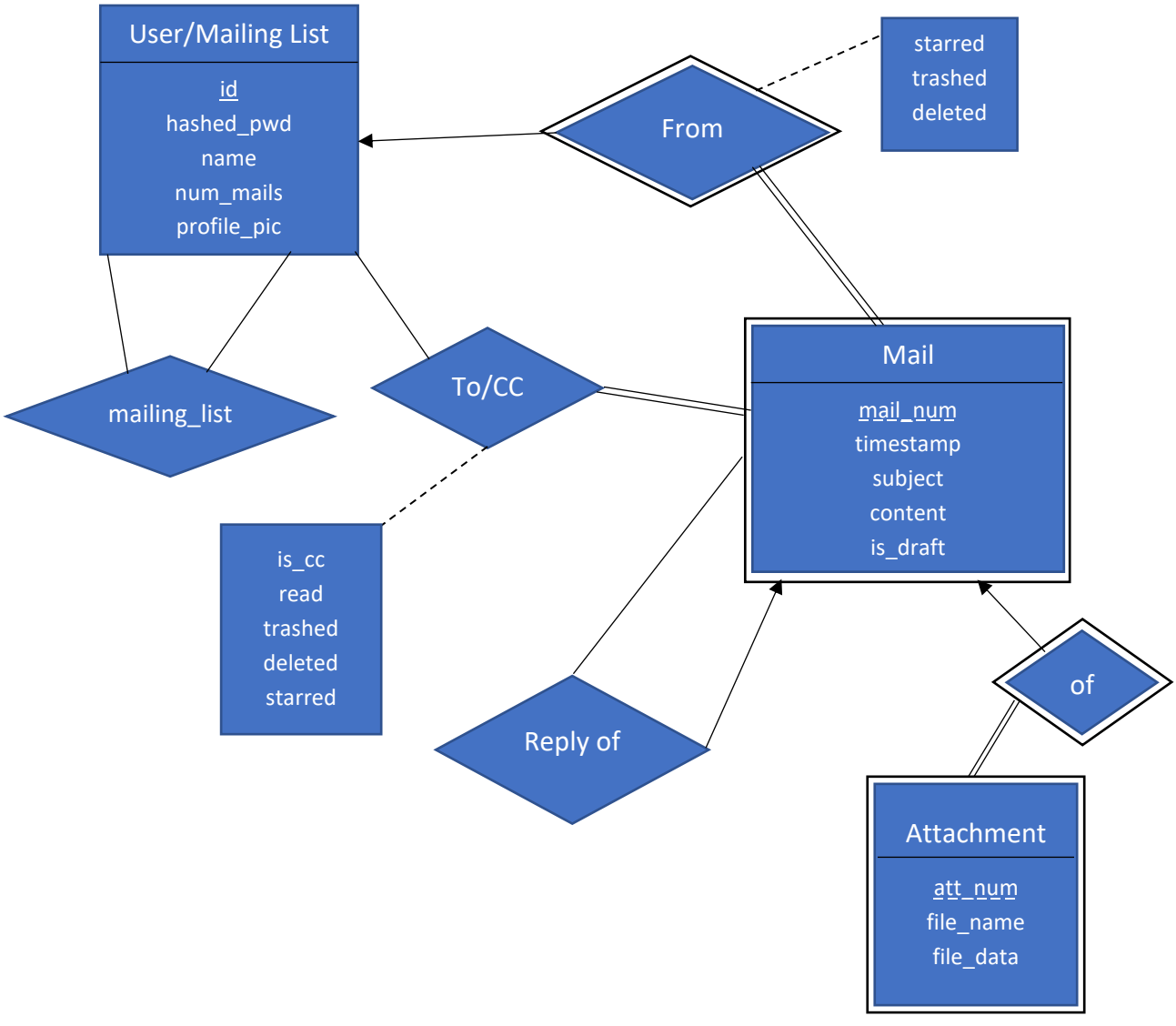
We create a website which serves the purpose of a generic email service.

Users:

An enterprise, university, or community

Proposed database usage design (how data is stored):

ER Design



Relational Design/Schema

From the above ER design, the following relational schema can be inferred to be optimal:

The relation mail_admin wasn't shown in the above ER design as it doesn't relate with any other relations and is only for administration purposes.

Primary key for each relation is marked in bold.

1) mail_admin (**id**, hashed_pwd)
2) mail_user (**id**, hashed_pwd, name, num_mails, profile_pic)
    a. check condition on num_mails (>=0)
3) mail (**sender_id, mail_num**, time, subject, content, is_draft, starred, trashed, deleted)
    a. sender_id references id in user
    b. is_draft, starred, trashed, and deleted are by default set to false
4) recipient (**sender_id, mail_num, id**, is_cc, read, starred, trashed, deleted)
    a. (sender_id, mail_num) references (sender_id, mail_num) in mail
    b. id references id in mail_user
    c. is_cc, read, starred, trashed, and deleted are by default set to false
5) mailing_list (**id, list_id**)
    a. id references id in mail_user
    b. list_id references id in mail_user
6) reply (**id, mail_num**, p_id, p_mail_num)
    a. (id, mail_num) references (sender_id, mail_num) in mail
    b. (p_id, p_mail_num) references (sender_id, mail_num) in mail
7) attachment (**sender_id, mail_num, att_num**, file_name, file_data)
    a. (sender_id, mail_num) references (sender_id, mail_num) in mail
    b. check condition on att_num (>=0)

Points to note:

- Scheduled mails are also stored in the mail table but with timestamp in future
- The attribute trashed in the mail relation denotes whether the mail is in sent box or trash box of sender whereas the same attribute into relation denotes the same but of recipient. The attributed deleted denotes deleted from trashed too.

Application Backend:

Based on Express, Node JS. Connected to PostgreSQL database.

Instead of directly using node-postgres functionality, we have created functions which are similar to those in python (psycopg2 functions): execute, executemany, consistent_execute.

execute: Arguments to this function are the list of queries, and the corresponding list of parameters. This function executes each query with the corresponding parameters tuple in params. We are using pg pool which can simultaneously handle large no. of requests.

executemany: Arguments to this function are a query, and a list of parameters. This function executes a single query with each tuple in the parameters list.

consistent_execute: This has the same functionality as execute but if output is inconsistent, it re-executes. We had to implement this because of some bugs in node-postgres pool. When there are large number of parallel queries or when different APIs are called simultaneously, it might sometimes end up returning the result of one client to another. But there is no problem in execution. Problem is only in returning query results.

So, in some of the places like viewing mail, and viewing parent mail, etc we use consistent execute instead of execute. It is because whenever these APIs are called, there is high chance that many other APIs are also called. But consistent execute is not required in APIs like fetching all mails in mailbox as it is generally not in combo with other APIs.

Following are the APIs provided by the backend to the frontend general users:

1) /login – logging in the user and returning status.
2) /signup – creates a new user.
3) /logout - destroys the session and logs out the user
4) /check_login – checks if a user is already logged in. Though all the other APIs check this first and return corresponding status when the user is not logged in, an additional API is created for this to serve the same purpose but when a user tries to open a wrong link (I mean wrong or non existing endpoint in frontend). Then, If he is logged in, redirect him to the home page (inbox page), else to the login page.
5) /mail/:box – this API serves the purpose of returning the list of mails to be displayed in the current mailbox (mailbox refers to different endpoints like inbox and sent). It would only send the necessary info to display list of mails like subject, time, read, starred.
   “box” here has 6 possibilities – inbox, starred, sent, drafts, trash, scheduled.
6) /get_mail/:box - displays a particular mail on which the user clicked, using the mail_num, and sender_id stored in the body of the request. Note that sender_id and mail_num together form the primary key of mail. Also, we send “box” as suffix because it is required to know whether we need a mail sent by the user or received by the user. It also sends the attachments in the mail.
7) /get_parent_mail – For the purpose of mail threading i.e., show the parent of a mail recursively, this API is provided. With the primary key of a mail, it fetches the mail to which this mail is a reply if such mail exists.

8) /compose/:num/:p_id/:p_mn – This API is used while loading the compose page.
   Case 1: num = 0 – To write a new mail
      Case 1a (when user clicks on compose button): p_id = 0, p_mn = 0 – It is in a new thread i.e., not a reply to any mail
      Case 1b (when user clicks on reply button for a mail): Else, it is a reply to the mail (p_id,p_mn). So, it loads all the recipients of the parent mail into the cc of the new mail and the sender of theparent mail as the to of the new mail. It also puts the subject of the parentmail as the subject of the new mail but with added prefix '"Re: ". With theseinfo filled, compose page is loaded for the user, he is free to further change these already filled items too.

   Case 2 (when user clicks on edit or send draft button): num != 0 – To edit an already saved draft. Here p_id, p_mn have no significance.
9) /check_ids – this API is used to verify the existence of IDs added by the user in the To and CC fields of compose page.
10) /send_mail/:num – When the user clicks on send or save as draft or schedule mail, this is the API which is used. It gets all the required info from the frontend along with attachments.
11) /modify – Suppose we star a mail or we mark a mail as unread. We might also move a mail to trash or restore it from trash. We might want to move a scheduled mail to drafts. For the backend these are just small changes without any insert, select or removal of tuples from database tables. So all these small changes are done using the modify API. Within body, along with primary key of a mail, we send the modification in the form of a json within json. For example, mod: {s: true} would mark a mail as starred.
12) /new_mailing_list (for admin only) - Create a mailing list
13) /add_to_mailing_list (for admin only) - Add a user to a mailing list

Application Frontend:

Based on React. It would be an email website which would have most of the features in any common email like viewing inbox, sent, drafts, starred, scheduled, bin pages and compose, reply, forward, move to trash, star, move to draft, schedule mail, etc.

The main endpoints in the frontend are: (along with login and signup pages)

The endpoint 'MailPage' contains the UI and necessary implementation to show all mail pages such as inbox, sent, starred, draft, scheduled, trash, etc and to display selected mail in a threaded view.

The endpoint 'ComposePage' includes composing new mails, replying and editing drafts.

The endpoint default is used when the path (in the link) doesn't correspond to any other endpoint. This just verifies the credentials of user from backend and redirects accordingly.

Our Learnings:

While designing ER, we felt we realised many things which we didn't learn in ER chapter and understood why first ER model is made and then converted to relational model.

While doing frontend, we learnt a lot about react hooks, useStates and useEffects while trying to make our frontend light and quick.

While doing backend, we learnt many small things how javascript is different from other languages.