MAIL SERVICE WEB-application

Elevate communication and productivity with Gmail Server—an innovative hub for seamless email experiences

A fully designed usable web application using Spring and VUE JS

Github link:

```
https://github.com/mahfouz72/mail-service
```

Contributors

1.Aly El-Din Mohamed El Sayed **21010835**

2.Mohamed Mahfouz Mohamed **21011210**

3.Esmail Mahmoud Hassan **21010272**

4.Youssif Khaled Ahmed **21011655**

Running process

The attached Pom.xml, Main.Js, vue.config.js, and index.html files serve the purpose of instantiating the required libraries for our project.
After Setting up the project using the Uploaded Source Code:

You should Use the Commands “npm install primevue”.

- It is advised to Modify the Pom.xml file to match the versions used on your machine.

Running front end

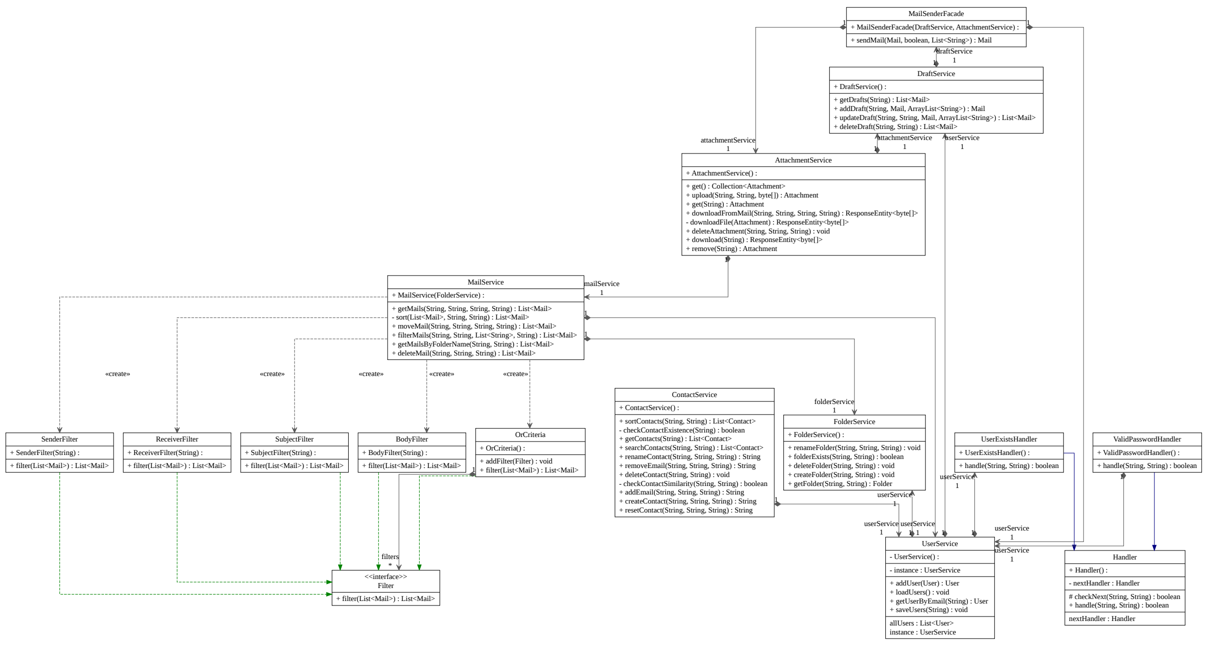
using npm run serve command and make sure that it didn’t open in the same port with spring boot you can modify the running portin thee config file

Running back end

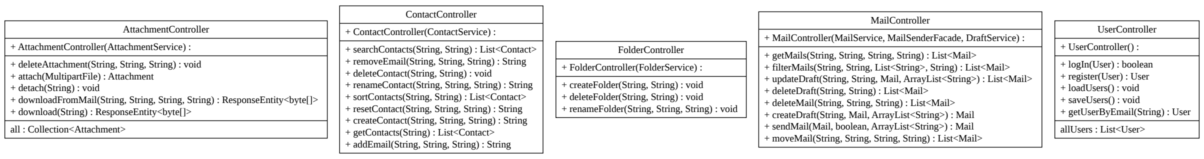
open backend folders in any ide and run using the normal run button or with maven

UML design

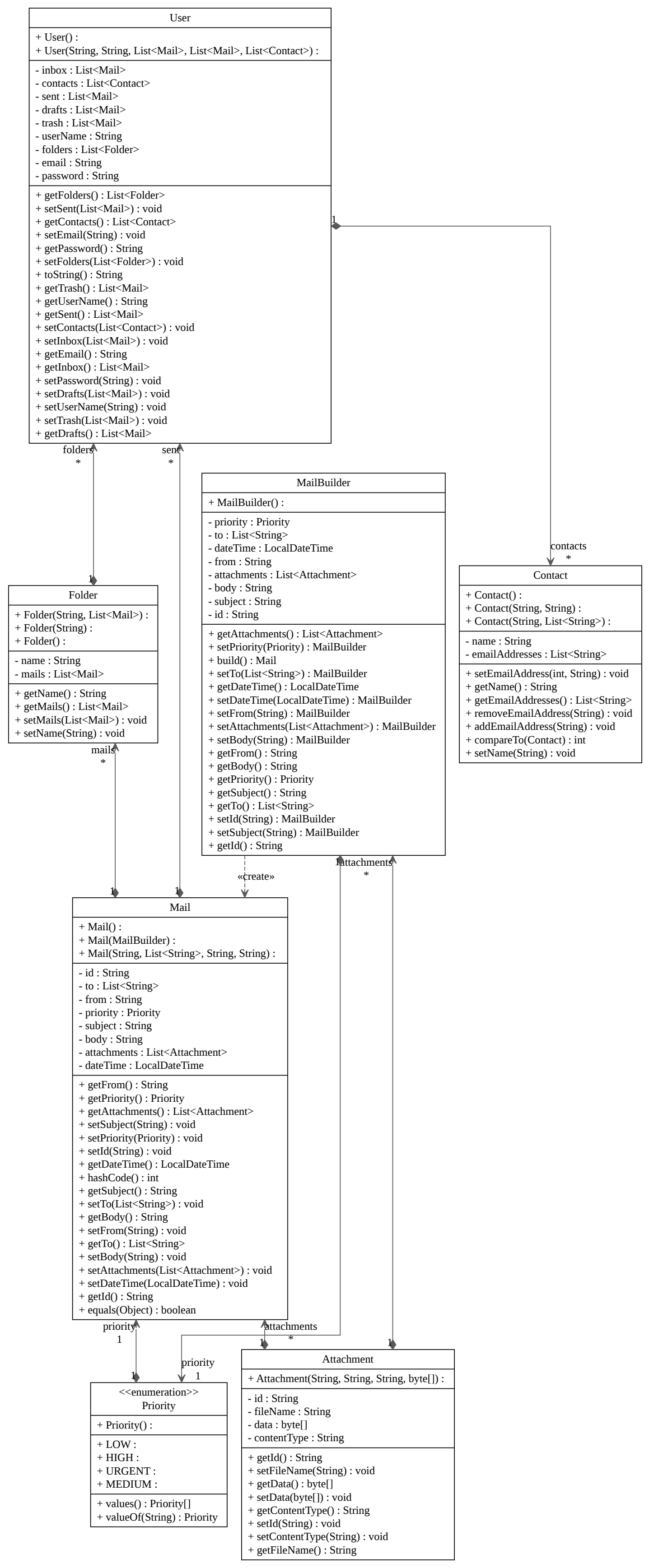
Service package


MailSenderFacade
+ MailSenderFacade(DraftService, AttachmentService) :
+ sendMail(Mail, boolean, List<String>) : Mail
draftService
DraftService
+ DraftService() :
+ getDrafts(String) : List<Mail>
+ addDraft(String, Mail, ArrayList<String>) : Mail
+ updateDraft(String, String, Mail, ArrayList<String>) : List<Mail>
+ deleteDraft(String, String) : List<Mail>
attachmentService
userService
AttachmentService
+ AttachmentService() :
+ get() : Collection<Attachment>
+ upload(String, String, byte[]) : Attachment
+ get(String) : Attachment
+ downloadFromMail(String, String, String, String) : ResponseEntity<byte[]>
- downloadFile(Attachment) : ResponseEntity<byte[]>
+ deleteAttachment(String, String, String) : void
+ download(String) : ResponseEntity<byte[]>
+ remove(String) : Attachment
mailService
MailService
+ MailService(FolderService) :
+ getMails(String, String, String, String) : List<Mail>
- sort(List<Mail>, String, String) : List<Mail>
+ moveMail(String, String, String, String) : List<Mail>
+ filterMails(String, String, List<String>, String) : List<Mail>
+ getMailsByFolderName(String, String) : List<Mail>
+ deleteMail(String, String, String) : List<Mail>
«create»
SenderFilter
+ SenderFilter(String) :
+ filter(List<Mail>) : List<Mail>
ReceiverFilter
+ ReceiverFilter(String) :
+ filter(List<Mail>) : List<Mail>
SubjectFilter
+ SubjectFilter(String) :
+ filter(List<Mail>) : List<Mail>
BodyFilter
+ BodyFilter(String) :
+ filter(List<Mail>) : List<Mail>
OrCriteria
+ OrCriteria() :
+ addFilter(Filter) : void
+ filter(List<Mail>) : List<Mail>
filters
<<interface>>
Filter
+ filter(List<Mail>) : List<Mail>
ContactService
+ ContactService() :
+ sortContacts(String, String) : List<Contact>
- checkContactExistence(String) : boolean
+ getContacts(String) : List<Contact>
+ searchContacts(String, String) : List<Contact>
+ renameContact(String, String, String) : String
+ removeEmail(String, String, String) : String
+ deleteContact(String, String) : void
- checkContactSimilarity(String, String) : boolean
+ addEmail(String, String, String) : String
+ createContact(String, String, String) : String
+ resetContact(String, String, String) : String
folderService
FolderService
+ FolderService() :
+ renameFolder(String, String, String) : void
+ folderExists(String, String) : boolean
+ deleteFolder(String, String) : void
+ createFolder(String, String) : void
+ getFolder(String, String) : Folder
UserExistsHandler
+ UserExistsHandler() :
+ handle(String, String) : boolean
ValidPasswordHandler
+ ValidPasswordHandler() :
+ handle(String, String) : boolean
UserService
- UserService() :
- instance : UserService
+ addUser(User) : User
+ loadUsers() : void
+ getUserByEmail(String) : User
+ saveUsers(String) : void
allUsers : List<User>
instance : UserService
Handler
+ Handler() :
- nextHandler : Handler
# checkNext(String, String) : boolean
+ handle(String, String) : boolean
nextHandler : Handler

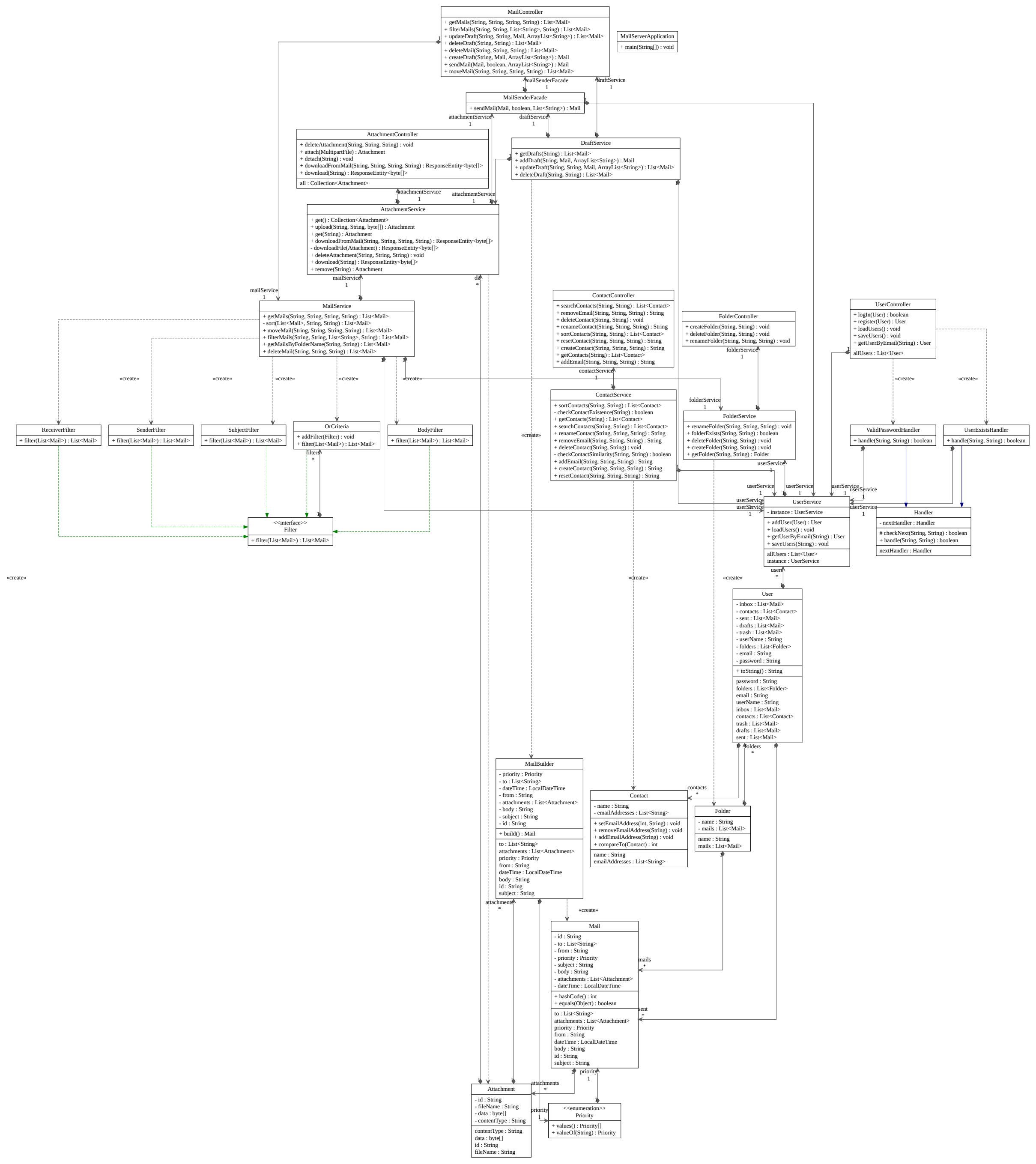
Controller package


AttachmentController
+ AttachmentController(AttachmentService) :
+ deleteAttachment(String, String, String) : void
+ attach(MultipartFile) : Attachment
+ detach(String) : void
+ downloadFromMail(String, String, String, String) : ResponseEntity<byte[]>
+ download(String) : ResponseEntity<byte[]>
all : Collection<Attachment>
ContactController
+ ContactController(ContactService) :
+ searchContacts(String, String) : List<Contact>
+ removeEmail(String, String, String) : String
+ deleteContact(String, String) : void
+ renameContact(String, String, String) : String
+ sortContacts(String, String) : List<Contact>
+ resetContact(String, String, String) : String
+ createContact(String, String, String) : String
+ getContacts(String) : List<Contact>
+ addEmail(String, String, String) : String
FolderController
+ FolderController(FolderService) :
+ createFolder(String, String) : void
+ deleteFolder(String, String) : void
+ renameFolder(String, String, String) : void
MailController
+ MailController(MailService, MailSenderFacade, DraftService) :
+ getMails(String, String, String, String) : List<Mail>
+ filterMails(String, String, List<String>, String) : List<Mail>
+ updateDraft(String, String, Mail, ArrayList<String>) : List<Mail>
+ deleteDraft(String, String) : List<Mail>
+ deleteMail(String, String, String) : List<Mail>
+ createDraft(String, Mail, ArrayList<String>) : Mail
+ sendMail(Mail, boolean, ArrayList<String>) : Mail
+ moveMail(String, String, String, String) : List<Mail>
UserController
+ UserController() :
+ logIn(User) : boolean
+ register(User) : User
+ loadUsers() : void
+ saveUsers() : void
+ getUserByEmail(String) : User
allUsers : List<User>

Model package


User
+ User() :
+ User(String, String, List<Mail>, List<Mail>, List<Contact>) :
- inbox : List<Mail>
- contacts : List<Contact>
- sent : List<Mail>
- drafts : List<Mail>
- trash : List<Mail>
- userName : String
- folders : List<Folder>
- email : String
- password : String
+ getFolders() : List<Folder>
+ setSent(List<Mail>) : void
+ getContacts() : List<Contact>
+ setEmail(String) : void
+ getPassword() : String
+ setFolders(List<Folder>) : void
+ toString() : String
+ getTrash() : List<Mail>
+ getUserName() : String
+ getSent() : List<Mail>
+ setContacts(List<Contact>) : void
+ setInbox(List<Mail>) : void
+ getEmail() : String
+ getInbox() : List<Mail>
+ setPassword(String) : void
+ setDrafts(List<Mail>) : void
+ setUserName(String) : void
+ setTrash(List<Mail>) : void
+ getDrafts() : List<Mail>
1
folders
*
sent
*
MailBuilder
+ MailBuilder() :
- priority : Priority
- to : List<String>
- dateTime : LocalDateTime
- from : String
- attachments : List<Attachment>
- body : String
- subject : String
- id : String
+ getAttachments() : List<Attachment>
+ setPriority(Priority) : MailBuilder
+ build() : Mail
+ setTo(List<String>) : MailBuilder
+ getDateTime() : LocalDateTime
+ setDateTime(LocalDateTime) : MailBuilder
+ setFrom(String) : MailBuilder
+ setAttachments(List<Attachment>) : MailBuilder
+ setBody(String) : MailBuilder
+ getFrom() : String
+ getBody() : String
+ getPriority() : Priority
+ getSubject() : String
+ getTo() : List<String>
+ setId(String) : MailBuilder
+ setSubject(String) : MailBuilder
+ getId() : String
contacts
*
Contact
+ Contact() :
+ Contact(String, String) :
+ Contact(String, List<String>) :
- name : String
- emailAddresses : List<String>
+ setEmailAddress(int, String) : void
+ getName() : String
+ getEmailAddresses() : List<String>
+ removeEmailAddress(String) : void
+ addEmailAddress(String) : void
+ compareTo(Contact) : int
+ setName(String) : void
1
Folder
+ Folder(String, List<Mail>) :
+ Folder(String) :
+ Folder() :
- name : String
- mails : List<Mail>
+ getName() : String
+ getMails() : List<Mail>
+ setMails(List<Mail>) : void
+ setName(String) : void
mails
*
1
attachments
*
«create»
1
1
Mail
+ Mail() :
+ Mail(MailBuilder) :
+ Mail(String, List<String>, String, String) :
- id : String
- to : List<String>
- from : String
- priority : Priority
- subject : String
- body : String
- attachments : List<Attachment>
- dateTime : LocalDateTime
+ getFrom() : String
+ getPriority() : Priority
+ getAttachments() : List<Attachment>
+ setSubject(String) : void
+ setPriority(Priority) : void
+ setId(String) : void
+ getDateTime() : LocalDateTime
+ hashCode() : int
+ getSubject() : String
+ setTo(List<String>) : void
+ getBody() : String
+ setFrom(String) : void
+ getTo() : List<String>
+ setBody(String) : void
+ setAttachments(List<Attachment>) : void
+ setDateTime(LocalDateTime) : void
+ getId() : String
+ equals(Object) : boolean
priority
1
attachments
*
1
1
priority
1
1
<<enumeration>>
Priority
+ Priority() :
+ LOW :
+ HIGH :
+ URGENT :
+ MEDIUM :
+ values() : Priority[]
+ valueOf(String) : Priority
Attachment
+ Attachment(String, String, String, byte[]) :
- id : String
- fileName : String
- data : byte[]
- contentType : String
+ getId() : String
+ setFileName(String) : void
+ getData() : byte[]
+ setData(byte[]) : void
+ getContentType() : String
+ setId(String) : void
+ setContentType(String) : void
+ getFileName() : String

Full UML



Design patterns

Facade

Usage

The Façade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.

How It’s used Mail-service?

The facade encapsulates the logic of **sending emails**, **handling drafts**, **and managing attachments**. This encapsulation promotes a clean separation of concerns, as the complexity of these operations is hidden behind a simplified interface provided by the facade.

Consequences of using Facade pattern

1.**Simplified Interface:**

- The `MailSenderFacade` provides a simplified method `sendMail` that abstracts away the complexity of dealing with various services (UserService, DraftService, AttachmentService) and their interactions.

2.**Encapsulation:**

- The facade encapsulates the interactions with `DraftService`, `AttachmentService`, and `UserService`. Clients only need to interact with the `MailSenderFacade`, which hides the details of these services.

3.**Reduced Dependency:**

- Clients depend only on the `MailSenderFacade`, not on the individual services. This reduces the coupling between clients and the underlying subsystem components, making it easier to modify or extend the system without affecting clients.

4.**Centralized Code:**

- The code related to sending mail, managing drafts, handling attachments, and interacting with users is centralized within the facade. This makes the codebase more organized and easier to maintain.

5.**Promotes Good Design Practices:**

- Encapsulation and abstraction promoted by the facade pattern adhere to good design practices, such as the separation of concerns and the single responsibility principle.

Filter

Usage

Filter pattern is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations.

How It's used Mail-service?

Used to filter mails according to some ciriteria.

```
public interface Filter {
    List<Mail> filter(List<Mail> mails);
}
```

Consequences of using Filter pattern?

1.**Modularity and Extensibility:**

- The Filter Design Pattern provides a modular way to encapsulate individual filtering criteria (e.g., sender, receiver).
- New filtering criteria can be added easily by implementing the `Filter` interface without modifying existing filters or the code that applies filters.

2.**Avoiding Code Duplication:**

- By using a common interface (`Filter`), the filtering logic is encapsulated and can be reused across different types of filters.
- This avoids code duplication and promotes a cleaner, more maintainable codebase.

3.**Separation of Concerns:**

- Each filter has a single responsibility: applying a specific filtering criterion.

- Separating filtering concerns into distinct classes adheres to the single responsibility principle and makes the codebase more maintainable.

Singleton

Usage

the Singleton design pattern is a creational design pattern that ensures a class has only one instance and provides a global point to this instance. This pattern is useful when exactly one object is needed to coordinate actions across the system.

How It’s used in Mail-service application?

The `UserService` class in the provided code is designed as a Singleton, following the Singleton pattern. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. In the context of your code, the `UserService` has a single instance (`instance`) that is shared across the entire application.

```
public static synchronized UserService getInstance() {
    if(instance == null)
        instance = new UserService();
    return instance;
}
```

Consequences of using Singleton pattern?

1. **Resource Efficiency:**

- Creating a single instance and reusing it can be resource-efficient, especially if the class involves heavy initialization or resource-intensive operations.

2.**Global Access:**

- The Singleton pattern provides a global point of access to the instance. This can be advantageous when a single instance needs to be shared and accessed throughout the application.

Chain of responsibility

Usage

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

How It’s used in Mail-service application?

There should be a base Handler class that declares the common interface for all concrete handlers. It might include methods like handle and setNext (to set the next handler in the chain).
Concrete Handler Classes (ValidPasswordHandler and UserExistsHandler):

Consequences of using Chain of responsibility pattern?

create a flexible and modular way to process authentication-related requests

Strategy

Usage

It's worth noting that the Strategy pattern is more general and can involve multiple methods, whereas `Comparable` is specifically focused on providing a single method for comparison.

How It’s used in Mail-service application?

The `Comparable` interface allows a class to define a single method (`compareTo`) that determines how instances of that class should be compared to each other, and this is somewhat akin to the Strategy pattern.

In the case of `Comparable`, the strategy is the specific comparison logic implemented in the `compareTo` method. Other classes or components that need to compare instances of the class can do so using this standardized strategy.

Builder

Usage

The Builder design pattern is a creational pattern that is used to construct a complex object step by step. It allows you to create different representations of an object by separating the construction process from the actual representation. This pattern is particularly useful when an object has a large number of optional parameters or configuration settings.

- The `Mail` object is a complex object with multiple attributes (e.g., id, from, to, subject, attachments, body, dateTime, priority).
- Constructing this object directly using a constructor with numerous parameters can lead to code that is difficult to read and maintain, especially when dealing with optional parameters.

How It’s used in Mail-service application?

- When updating a draft, not all attributes of the original `Mail` object need to be modified. Some attributes may remain the same, while others need to be updated.
- The Builder pattern allows for selective modification of specific attributes by providing a fluent interface to set only the necessary attributes.

```
Mail updatedDraft = new MailBuilder()
                    .setId(id)
                    .setFrom(newDraft.getFrom())
                    .setTo(newDraft.getTo())
                    .setSubject(newDraft.getSubject())
                    .setAttachments(draft.getAttachments())
                    .setBody(newDraft.getBody())
                    .setDateTime(LocalDateTime.now().withNano
                    .setPriority(newDraft.getPriority())
                    .build();
```

Design decisions

- **Contacts**: They are designed to be users that the user could save them in contacts list, and

  each contact must have name it’s available that the contact have multi email addresses in condition that the email address is owned by a registered user in creating or reset operations

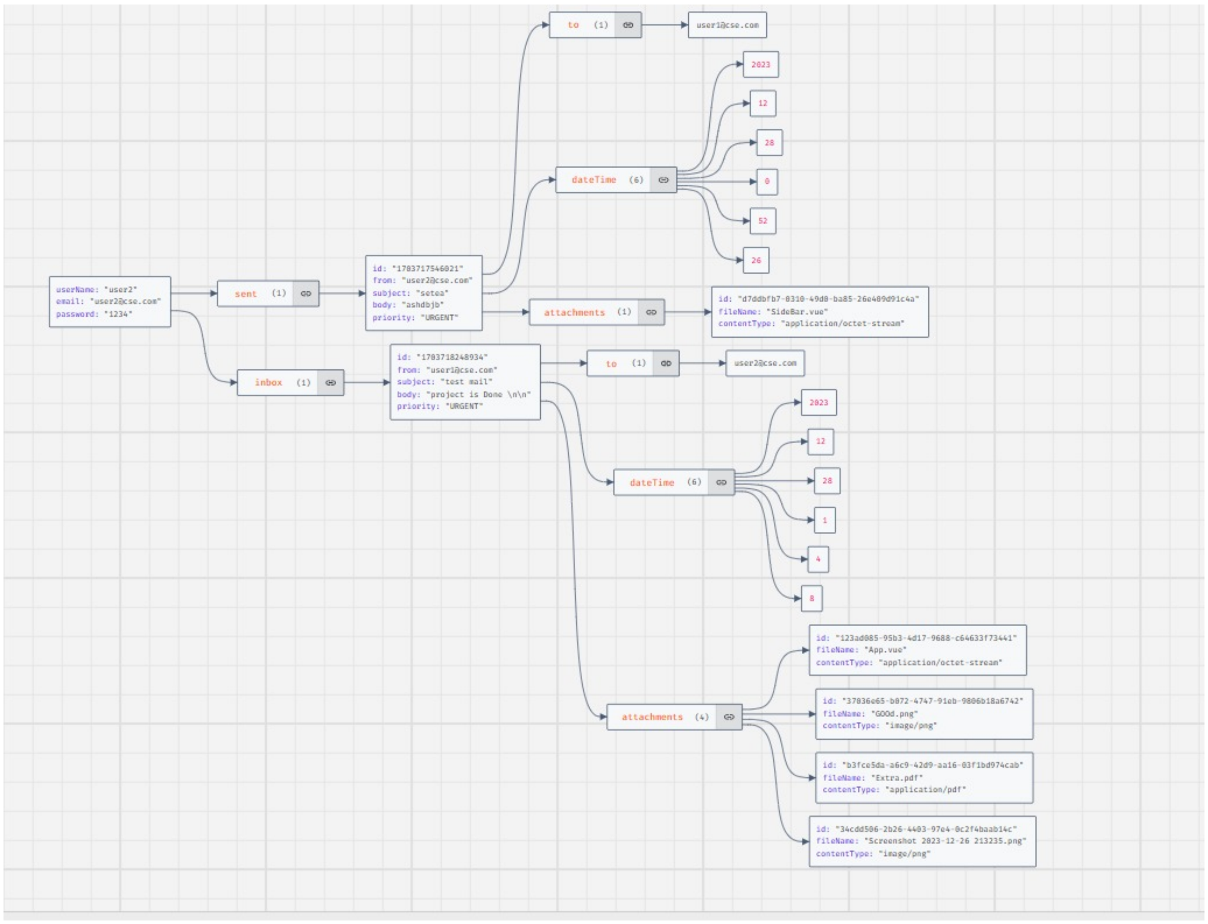
  Attributes: Name, List of email addresses

  Methods: Create, rename, reset email, delete

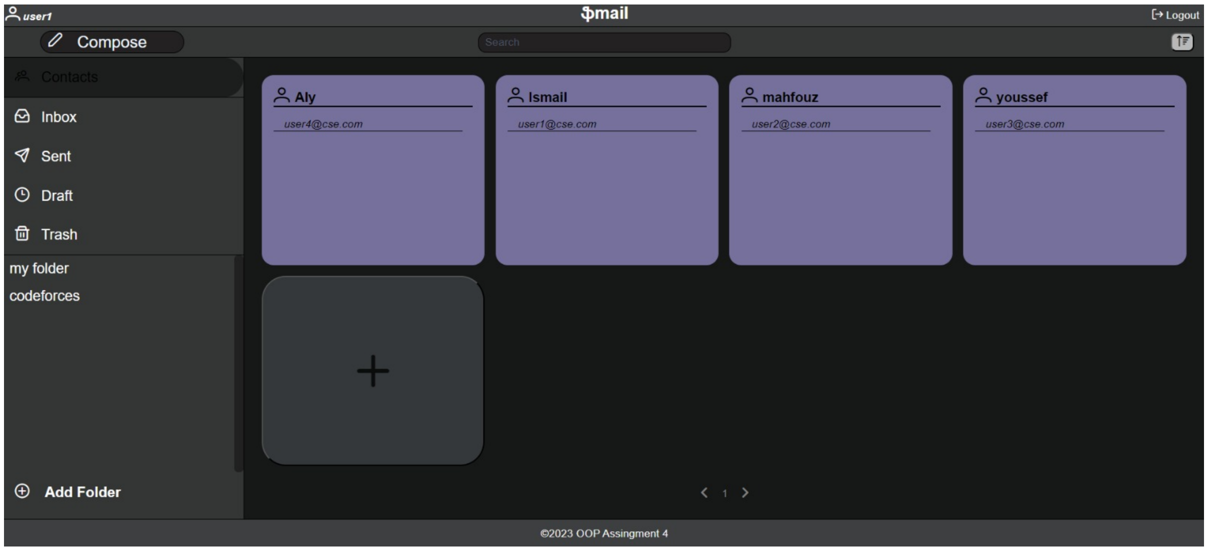
- Drafts: They are designed to be a List of emails that each user owns
- Each user will be saved in a separate file using JSON files
- you can’t move mail to not existing folder
- moving mails will not remove it from its current folder
- sorting and filtering will be done in the current folder only

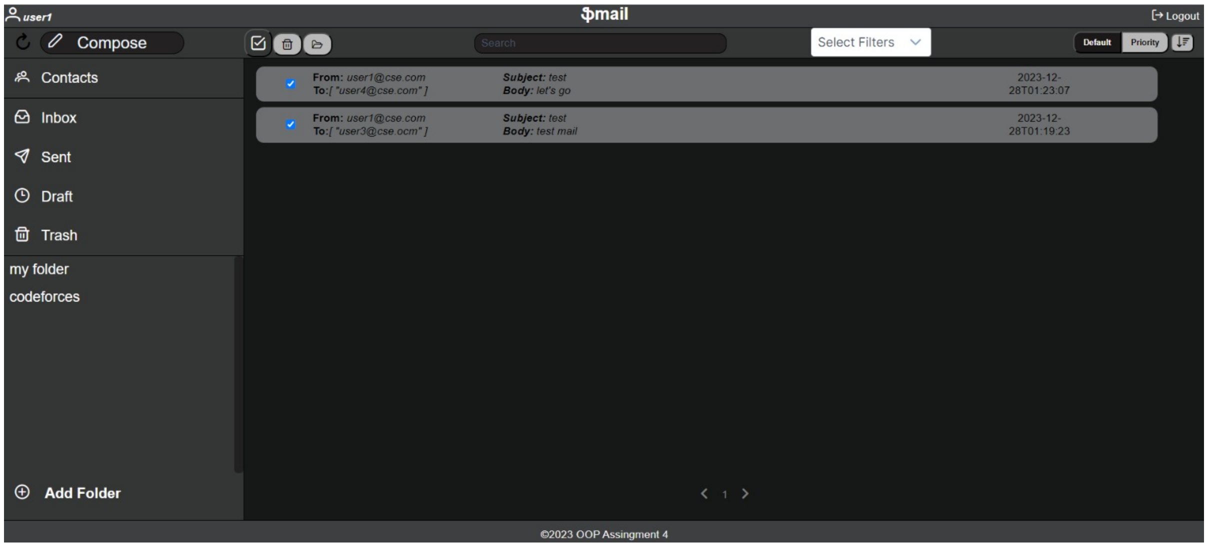
---

JSON SChema


userName: "user2"
email: "user2@cse.com"
password: "1234"
sent (1)
id: "1703717546021"
from: "user2@cse.com"
subject: "setea"
body: "ashdbjb"
priority: "URGENT"
to (1)
user1@cse.com
dateTime (6)
2023
12
28
0
52
26
attachments (1)
id: "d7ddbfb7-0310-49d0-ba85-26e489d91c4a"
fileName: "SideBar.vue"
contentType: "application/octet-stream"
inbox (1)
id: "1703718248934"
from: "user1@cse.com"
subject: "test mail"
body: "project is Done \n\n"
priority: "URGENT"
to (1)
user2@cse.com
dateTime (6)
2023
12
28
1
4
8
attachments (4)
id: "123ad085-95b3-4d17-9688-c64633f73441"
fileName: "App.vue"
contentType: "application/octet-stream"
id: "37036e65-b072-4747-91eb-9806b18a6742"
fileName: "GOOd.png"
contentType: "image/png"
id: "b3fce5da-a6c9-42d9-aa16-03f1bd974cab"
fileName: "Extra.pdf"
contentType: "application/pdf"
id: "34cdd506-2b26-4403-97e4-0c2f4baab14c"
fileName: "Screenshot 2023-12-26 213235.png"
contentType: "image/png"

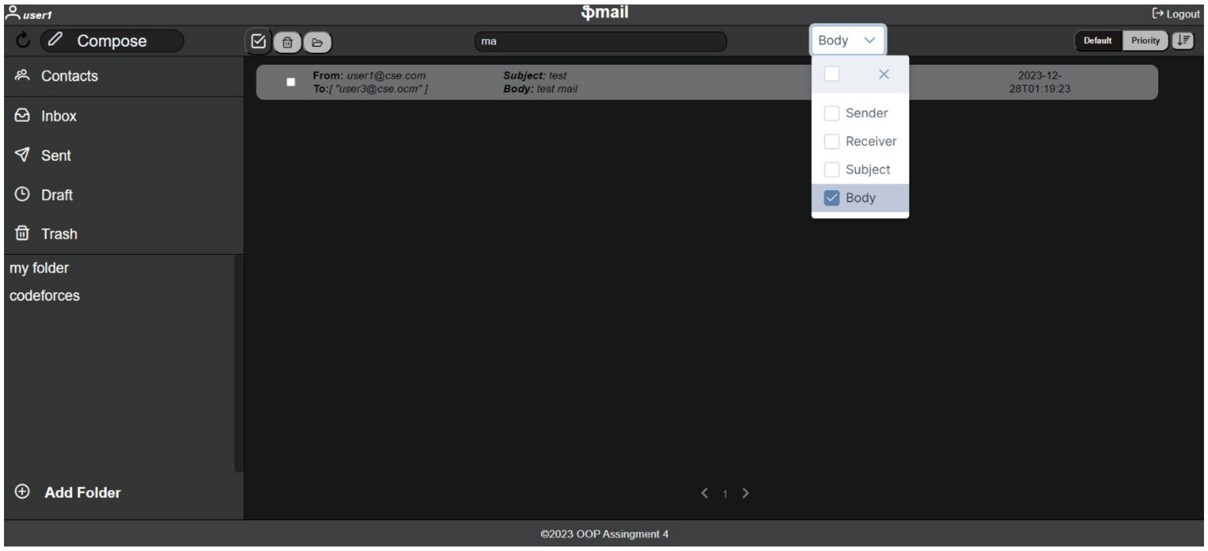
Snap shots


user1
$mail
Logout
Compose
Search
Contacts
Inbox
Sent
Draft
Trash
my folder
codeforces
Add Folder
Aly
user4@cse.com
Ismail
user1@cse.com
mahfouz
user2@cse.com
youssef
user3@cse.com
+
1
©2023 OOP Assingment 4

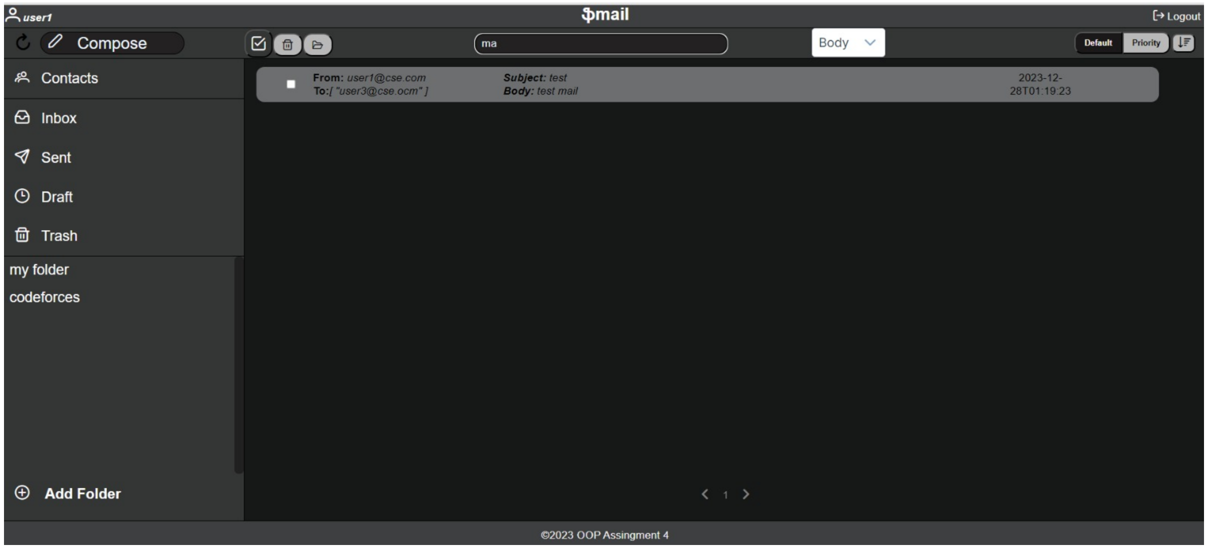


From: user1@cse.com
To: ["user4@cse.com"]
Subject: test
Body: let's go
2023-12-28T01:23:07

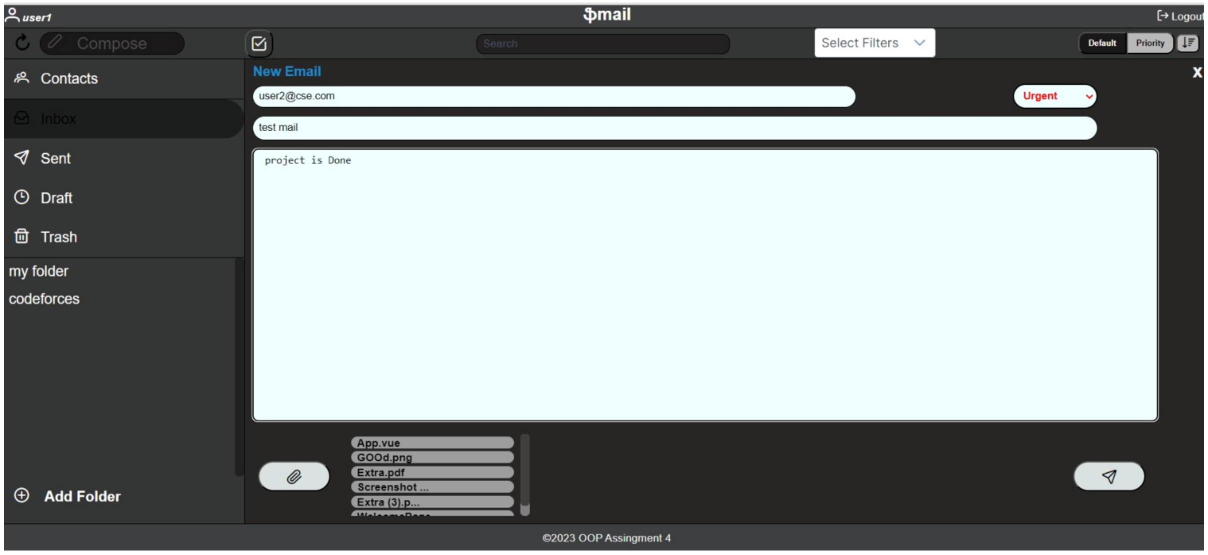
From: user1@cse.com
To: ["user3@cse.ocm"]
Subject: test
Body: test mail
2023-12-28T01:19:23



From: user1@cse.com
To: ["user3@cse.ocm"]
Subject: test
Body: test mail
2023-12-28T01:19:23



From: user1@cse.com
To: ["user3@cse.ocm"]
Subject: test
Body: test mail
2023-12-28T01:19:23

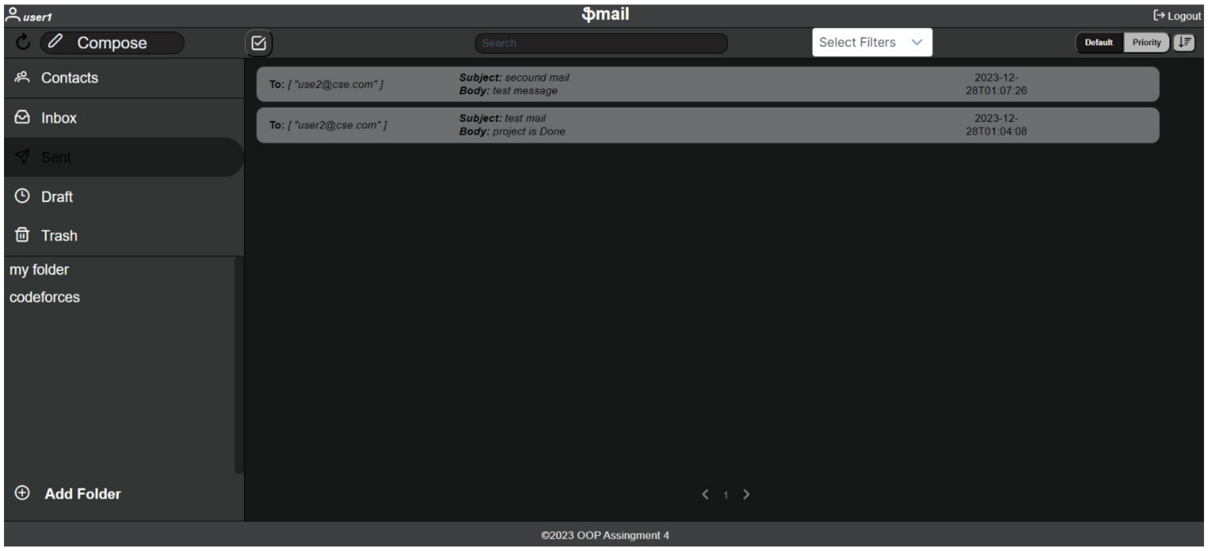


user2@cse.com

test mail

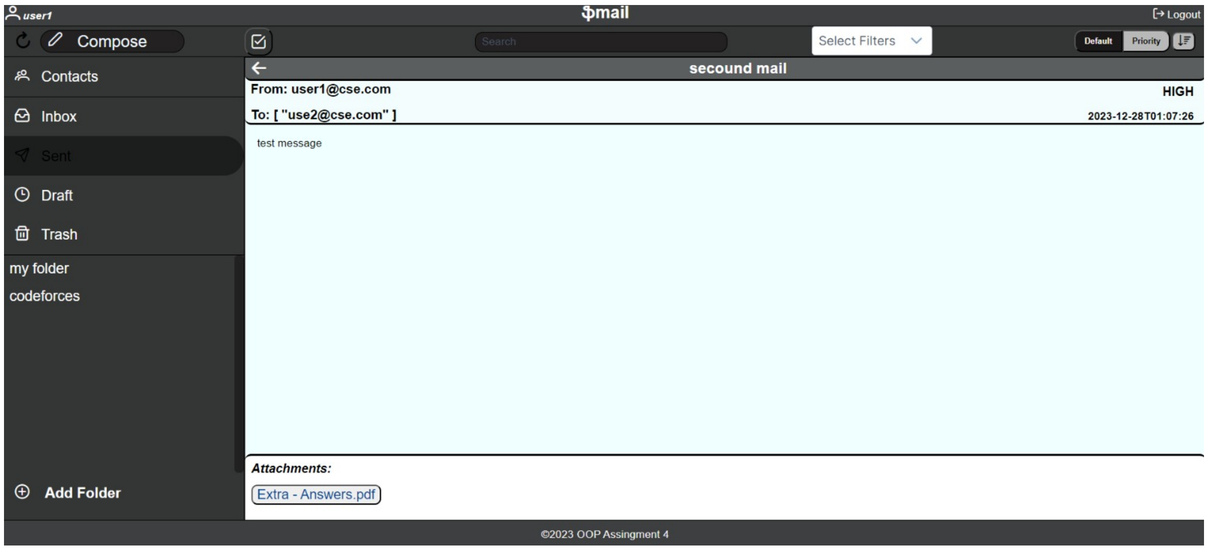
```
project is Done
```

App.vue
GOOd.png
Extra.pdf
Screenshot ...
Extra (3).p...



To: *["use2@cse.com"]* ***Subject:*** *secound mail* ***Body:*** *test message* 2023-12-28T01:07:26

To: *["user2@cse.com"]* ***Subject:*** *test mail* ***Body:*** *project is Done* 2023-12-28T01:04:08



secound mail

From: user1@cse.com **HIGH**

To: ["use2@cse.com"] **2023-12-28T01:07:26**

test message

Attachments:

Extra - Answers.pdf

Mail Server Application User Guide

Introduction

Welcome to the Mail Server Application! This guide will walk you through the functionalities and usage of the different components provided in the application.

Table of Contents

Attachments

1. Get All Attachments

- **Endpoint:** `GET /attachments`
- **Description:** Retrieve a collection of all attachments.

2. Attach File

- **Endpoint:** `POST /attach`
- **Description:** Upload an attachment file.
- **Request:** Multipart form-data with the file data.
- **Response:** Details of the uploaded attachment.

3. Detach Attachment

- **Endpoint:** `DELETE /detach/{id}`
- **Description:** Remove an attachment by its ID.
- **Request:** Path variable with the attachment ID.

4. Download Attachment

- **Endpoint:** `GET /download/{id}`
- **Description:** Download an attachment by its ID.
- **Request:** Path variable with the attachment ID.
- **Response:** Attachment file for download.

5. View Attachment from Email

- **Endpoint:** `GET /view/{email}/{mailId}/{folderName}/{attachmentId}`
- **Description:** View an attachment from a specific email.
- **Request:** Path variables with email, mail ID, folder name, and attachment ID.
- **Response:** Attachment file for viewing.

6. Delete Attachment from Email

- **Endpoint:** `DELETE /deleteAttachment/{email}/{mailId}/{attachmentId}`
- **Description:** Delete an attachment from a specific email.
- **Request:** Path variables with email, mail ID, and attachment ID.

Contacts

1. Create Contact

- **Endpoint:** `POST /contact/create/{userEmailAddress}/{contactName}/{contactEmailAddress}`
- **Description:** Create a new contact for a user.
- **Request:** Path variables with user email, contact name, and contact email.
- **Response:** Confirmation message.

2. Rename Contact

- **Endpoint:** `POST /contact/rename/{userEmailAddress}/{contactName}/{contactEmailAddress}`
- **Description:** Rename an existing contact for a user.
- **Request:** Path variables with user email, old contact name, new contact name, and contact email.
- **Response:** Confirmation message.

3. Reset Contact Email

- **Endpoint:** `POST /contact/reset/{userEmailAddress}/{contactEmailAddress}/{newContactEmailAddress}`
- **Description:** Reset the email address of an existing contact for a user.
- **Request:** Path variables with user email, contact email, and new contact email.
- **Response:** Confirmation message.

4. Add Email to Contact

- **Endpoint:** `POST /contact/add/{userEmailAddress}/{contactEmailAddress}/{newContactEmailAddress}`
- **Description:** Add an additional email to an existing contact for a user.
- **Request:** Path variables with user email, contact email, and new contact email.
- **Response:** Confirmation message.

5. Remove Email from Contact

- **Endpoint:** `DELETE /contact/remove/{userEmailAddress}/{contactEmailAddress}/{removedContactEmailAddress}`
- **Description:** Remove an email from an existing contact for a user.
- **Request:** Path variables with user email, contact email, and removed contact email.
- **Response:** Confirmation message.

6. Delete Contact

- **Endpoint:** `DELETE /contact/delete/{userEmailAddress}/{contactEmailAddress}`
- **Description:** Delete an existing contact for a user.
- **Request:** Path variables with user email and contact email.

7. Sort Contacts

- **Endpoint:** `GET /contact/sort/{userEmailAddress}/{order}`
- **Description:** Sort contacts for a user based on the specified order (ascending or descending).
- **Request:** Path variables with user email and sort order.
- **Response:** Sorted list of contacts.

8. Search Contacts

- **Endpoint:** `GET /contact/search/{userEmailAddress}/{contactName}`
- **Description:** Search for contacts for a user based on the provided contact name.
- **Request:** Path variables with user email and contact name.
- **Response:** List of contacts matching the search criteria.

9. Get All Contacts

- **Endpoint:** `GET /contact/get/{userEmailAddress}`
- **Description:** Retrieve all contacts for a user.
- **Request:** Path variable with user email.
- **Response:** List of all contacts for the user.

Folders

1. Create Folder

- **Endpoint:** `POST /folder/{email}/{folderName}`
- **Description:** Create a new folder for a user.

- **Request:** Path variables with user email and folder name.

2. Delete Folder

- **Endpoint:** `DELETE /folder/{email}/{folderName}`
- **Description:** Delete an existing folder for a user.
- **Request:** Path variables with user email and folder name.

3. Rename Folder

- **Endpoint:** `POST /folder/{email}/{oldName}/{newName}`
- **Description:** Rename an existing folder for a user.
- **Request:** Path variables with user email, old folder name, and new folder name.

Mails

1. Get Mails

- **Endpoint:** `GET /{email}/{folderName}`
- **Description:** Retrieve emails for a user from a specific folder.
- **Request:** Path variables with user email and folder name.
- **Optional Query Parameters:** Sorting criteria and sorting order.

2. Filter Mails

- **Endpoint:** `GET /{email}/{folderName}/filter`
- **Description:** Filter emails for a user from a specific folder based on criteria.
- **Request:** Path variables with user email and folder name.
- **Query Parameters:** List of filter criteria and filter value.

3. Send Mail

- **Endpoint:** `POST /compose`
- **Description:** Send a new email or compose a draft.

- **Request:** JSON body with email details and optional parameters for draft.

4. Move Mail

- **Endpoint:** `POST /{email}/{fromFolder}/{toFolder}/{id}`
- **Description:** Move an email from one folder to another.
- **Request:** Path variables with user email, source folder, destination folder, and email ID.

5. Delete Mail

- **Endpoint:** `DELETE /{email}/{folderName}/{id}`
- **Description:** Delete an email from a specific folder for a user.
- **Request:** Path variables with user email, folder name, and email ID.

6. Create Draft

- **Endpoint:** `POST /createDraft/{email}`
- **Description:** Create a draft email for a user.
- **Request:** Path variable with user email and JSON body with draft details.