Mail Web App

2025

Faculty Of Engineering

Alexandria University
Programming 2

Name	ID
Esraa Abdelhay Mohamed	23010276
Nour Ahmed Mahmoud	23010913
Karim Mohamed Basim	23010673
Ahmed Hassan Etman	23010235

Application Setup and Execution Guide

The application is built as a full-stack solution with separate frontend and backend components. Both must be running concurrently for the application to function.

Component	Technology	Required Tools
Backend	Spring Boot (Java)	Java Development Kit (JDK) Maven
Frontend	Angular	Node.js npm Angular CLI
Database	MySQL	MySQL Server MySQL Workbench

Step-by-Step Execution

- **Start the Backend (Spring Boot)**
  - **Navigate to the backend directory:**
    - cd C:\Users\.....\MailBackend
  - **Build the project using Maven:**
    - mvn clean install.
  - **Run the Spring Boot application:**
    - mvn spring-boot:run

NOTE: The backend REST API server will start, typically running on `http://localhost:8080`

- **Start the Frontend (Angular)**
  - **Open a new terminal window and navigate to the frontend directory:**
    - cd C:\Users\.....\MailFrontend
  - **Install the necessary Node Package Manager (npm) dependencies:**
    - npm install
  - **Start the Angular development server:**
    - ng serve --open

NOTE: The frontend application will compile and automatically open in your default browser at `http://localhost:4200`.

The Angular application uses its services to make HTTP requests to the running Spring Boot backend.

Chain of Responsibility Pattern in Authentication System

Architecture Components

1. Handler Interface (`IHandler`)

Defines the contract with two methods: `setNext()` to link handlers and `handle()` to process requests.

2. Base Handler (`ParentHandler`)

Implements the chain mechanism. If a next handler exists, it delegates the request; otherwise, it marks authentication as successful (`reqState = true`). This serves as the foundation for all concrete handlers.

3. Concrete Handlers

Four specialized validators extending **`ParentHandler`**:

- **`UniqueEmail`**: Ensures email doesn't exist (registration)
- **`EmailExists`**: Ensures email exists (login)
- **`LongEnough`**: Validates password length (≥8 characters)
- **`CorrectPassword`**: Verifies password matches stored value

Each handler follows a consistent pattern: validate → pass to next handler if valid → set error state and terminate if invalid.

4. Chain Factory (`ChainFactory`)

Assembles validation chains based on context using the Factory pattern:

Registration Chain: `uniqueEmail → longEnough → chainEnd`

Login Chain: `emailExists → correctPassword → chainEnd`

Execution Flow

When **`handle(userDTO)`** is called on the chain head:

1. First handler performs its validation
2. **Success**: Calls `super.handle()` to pass to next handler
3. **Failure**: Sets error message in `userDTO` and returns `false`, terminating the chain
4. **Chain end**: When no next handler exists, sets success state and returns `true`

Pattern Benefits

- **Single Responsibility**: Each handler has one validation task.
- **Open/Closed Principle**: New validators can be added without modifying existing code—simply create a new handler class and add it to the chain.
- **Dynamic Configuration**: Different chains for different use cases (registration vs. login) assembled at runtime.
- **Early Termination**: Chain stops at first failure, improving performance by skipping unnecessary validations.
- **Flexibility**: Validation order and composition can be changed easily in the factory without affecting handler implementations.

Chain of Responsibility Pattern in Searching Mails

- **Define the Abstract Template**: The `SearchHandler` class serves as the abstract foundation, defining the mandatory `handle()` and `shouldHandle()` methods that every specific search filter must implement.
- **Establish the Linkage**: It provides a `setNext()` method, which allows individual handlers to be linked together into a sequential chain.
- **Execute the "Conditional Passing" Logic**: The `processChain()` method manages the flow of the search request through the following internal sequence:
    1. **Validation**: It first calls `shouldHandle(criteria)` to determine if the current handler is relevant to the user's search input.
    2. **Action**: If relevant, it executes the `handle()` method to append specific database constraints (predicates).
    3. **Delegation**: Regardless of whether the current handler acted, it checks if a `nextHandler` exists.
    4. **Recursion**: If a subsequent handler is found, it recursively calls `processChain()` to ensure every field in the criteria object is evaluated by the rest of the chain.
- **Decoupled Logic**: Each handler is independent. For example, the **`AttachmentSearchHandler`** only cares if `hasAttachment` is specified, while the `DateRangeSearchHandler` isolatedly manages temporal logic.

Filter Design Pattern in Searching Mails

The search system integrates the CoR pattern with the **Filter Pattern**.

- **The Criteria Object**: The `SearchCriteriaDTO` acts as the "Criteria" container. It carries all possible filter parameters (query, folder, priority, dates, etc.) through the chain.
- **Dynamic Predicate Building**: Instead of a single massive "if-else" block or a complex SQL string, the **Filter Pattern** is distributed across the handlers. Each handler "filters" the data by adding a `Predicate` to a shared `List<Predicate>`.
- **Abstraction of Data Access**: The pattern allows the `EmailSearchService` to remain agnostic of the underlying database schema. It simply passes the criteria to the `emailSearchRepository`, which uses the chain to generate the final `Specification` or `Predicate` set.

Synergy Between the Patterns

The combination of these two patterns results in a highly flexible architecture:

- **Recursive Composition**: The CoR ensures that every field in the `SearchCriteria` is checked. If a user provides both a "Subject" and a "Date Range," the `SubjectSearchHandler` and `DateRangeSearchHandler` will both contribute predicates to the final query.
- **Boolean Logic Handling**:
    - **Global Search** uses an **OR** logic internally (searching email, subject, and body simultaneously).
    - **The Chain** effectively applies **AND** logic between different handlers, as all added predicates must be satisfied in the final repository call.
- **Performance via Early Exit**: While most handlers pass the request forward, the `shouldHandle` check ensures that the system doesn't waste resources performing expensive Joins (like the `Subquery` in `ToSearchHandler`) if those search fields are empty.

Strategy Design Pattern in Sorting Mails

Architecture Components

Strategy Interface (`SortStrategy.java`): Defines the standard contract for all sorting behaviors. It mandates two methods: `getSort()`, which returns a Spring Data `Sort` object, and `getStrategyName()`, used for identification.

Concrete Strategies: Each class implements a specific sorting logic, isolating the rules for different fields:

- **Field-Specific Sorting**: `SortBySubjectStrategy` and `SortBySenderStrategy` sort alphabetically by their respective fields and use `sentAt` as a secondary tie-breaker.
- **Status-Based Sorting**: `SortByUnreadStrategy` prioritizes unread messages.
- **Priority-Based Sorting**: `SortByPriorityStrategy` orders mail by its importance level.
- **Temporal Sorting**: Multiple strategies handle dates, such as `SortByDateDescStrategy` (for standard folders) and `SortByUpdatedDateDesc` (for Drafts).

Strategy Factory (`SortStrategyFactory.java`): Acts as the orchestrator that manages all available strategies. It uses **Dependency Injection** to collect all concrete strategy components and stores them in a map for quick retrieval.

Dynamic Execution Flow

The power of this pattern is most evident in how the `SortStrategyFactory` selects a strategy based on the application's state:

1. **Contextual Logic**: The factory doesn't just return a strategy by name; it considers the `folderName`. If a user requests to sort by "Date" while in the **Drafts** folder, the factory automatically redirects the request to use `UPDATED_DATE_DESC` instead of the standard `DATE_DESC`.
2. **Runtime Retrieval**: The `EmailSearchService` calls the factory with a string identifier (e.g., "PRIORITY"). The factory retrieves the corresponding object from its internal map.
3. **Default Fallback**: If an invalid or null strategy name is provided, the factory defaults to `DATE_DESC`, ensuring the system never fails to return a sorted list.

Integration with Pagination

The sorting strategies are integrated directly into the pagination flow to ensure consistent results across pages:

- **Pageable Construction**: The `EmailSearchService` uses the `Sort` object from the chosen strategy to create a `PageRequest`.
- **Database Execution**: The repository uses this `Pageable` object to perform limit/offset queries in the correct order.

Pattern Benefits

- **Open/Closed Principle**: New sorting criteria (e.g., sorting by "Size") can be added by creating a single new class implementing `SortStrategy` and adding it to the factory, without touching existing sorting logic.
- **Elimination of Conditional Complexity**: It replaces large `switch` or `if-else` blocks in the service layer with a clean, polymorphic call to `getSort()`.
- **Separation of Concerns**: The `EmailSearchService` is responsible for searching, while the `SortStrategy` implementations are solely responsible for defining the order of results.
- **Reusability**: These strategies can be reused across different parts of the application, such as standard folder views and advanced search results.

Strategy Design Pattern in Sorting Mails

Strategy Interface: (`SendStrategy.java`) A common interface defines the contract for different sending behaviors. This allows different implementations to be used interchangeably without changing the calling code.

Concrete Strategies:

- **SingleReceiverSend**: This strategy is used when the email has only one recipient.
  It:
  - Determines the single receiver (TO, CC, or BCC)
  - Saves the mail in the receiver's inbox
  - Saves the mail once in the sender's Sent folder

- **MultipleReceiverSend**: This strategy handles emails with multiple recipients.
  It:
  - Processes TO, CC, and BCC recipients using a queue
  - Ensures the email is saved once in the sender's Sent folder
  - Prevents duplicate inbox entries for the same recipient
  - Prevents duplicate (email, receiverType) database records

Strategy Selection and Usage:

The appropriate strategy is selected at runtime based on the email request and executed through the common interface.

This ensures that the email-sending service is:

- **Open for extension** (new strategies can be added easily)
- **Closed for modification** (existing logic remains unchanged)
- Clean, readable, and easier to test

Factory Pattern (Chain Creation)

The Factory pattern is implemented in `ChainFactory` to create different validation chains based on the authentication operation type.

How it Works:

The `getChain(ChainType)` method constructs the appropriate chain:

- Takes a `ChainType` enum (Login or Register)
- Assembles handlers in the correct order
- Returns the head of the chain for execution

This ensures that:

- Chain construction logic is centralized
- Controllers don't need to know chain composition
- New authentication flows can be added easily

Builder Design Pattern

The **Builder Design Pattern** was used to construct complex domain entities in a clear, readable, and maintainable way. Several entities in the mail system contain many optional or context-dependent fields, making traditional constructors difficult to read and error-prone.

Motivation

Entities such as `Mail`, `UserMail`, and `MailReceiver`:

- Have multiple attributes
- Are created in different contexts (draft, sent mail, inbox mail, receiver mapping)
- Do not always require all fields to be set at creation time

Using large constructors or setters would:

- Reduce code readability
- Increase the risk of incorrect object initialization
- Make the code harder to maintain as the model evolves

The Builder Pattern solves this by allowing objects to be constructed step by step with explicit intent.

Builder Implementation

The builder pattern is implemented using Lombok's **`@Builder`** annotation on the entity classes.

Key Design Approaches

The following significant design decisions were made to structure the application for scalability, maintainability, and security:

1. Full-Stack Separation

Choosing a separate Angular frontend and Spring Boot backend allows for:

- Independent development and deployment
- Technology flexibility (can replace frontend or backend independently)
- Clear API contracts through REST endpoints
- Scalability (frontend and backend can scale separately)

2. Layered Architecture

The backend follows a strict layered architecture:

- **Controller Layer:** Handles HTTP requests and responses
- **Service Layer:** Contains business logic
- **Repository Layer:** Manages data persistence
- **Entity Layer:** Represents database structure

This separation ensures maintainability and testability.

3. Chain of Responsibility for Authentication

The Chain of Responsibility pattern was chosen for authentication because:

- **Flexibility:** Easy to add or modify validation rules
- **Clarity:** Each validation concern is isolated
- **Extensibility:** New authentication flows can be added without modifying existing code
- **Maintainability:** Changes to one validation don't affect others

4. Pagination Support

- Improve performance by loading data in chunks
- Reduce network bandwidth usage
- Enhance user experience with faster page loads
- Support large mailboxes efficiently

The `MailPageDTO` includes metadata like total pages, current page, and element counts to support frontend pagination controls.

5. System vs. Custom Folders

The folder system distinguishes between:

- **System Folders** (Inbox, Sent, Trash, Drafts): Cannot be deleted or renamed
- **Custom Folders:** User-created, fully manageable

This design provides familiar email organization while allowing customization.

6. Normalized Database Design

The database schema uses proper normalization with:

- Separate tables for Users, Mail, Folders, Contacts
- Junction tables (UserMail, MailReceiver) for many-to-many relationships
- Foreign key relationships to maintain referential integrity
- Cascade operations for automatic cleanup

7. CORS Configuration

The application includes a comprehensive CORS filter to:

- Allow frontend-backend communication from different origins
- Support all necessary HTTP methods (GET, POST, PUT, DELETE)
- Handle credentials and custom headers
- Provide secure cross-origin resource sharing

8. Use of Enums for Type Safety

Enums are used throughout the application for:

- **Priority:** LOW, NORMAL, HIGH, CRITICAL
- **ReceiverType:** TO, CC, BCC
- **ChainType:** Login, Register

This provides compile-time type safety and prevents invalid values.

9. Lazy Loading Strategy

JPA entities use `FetchType.LAZY` for associations to:

- Improve performance by loading related data only when needed
- Reduce memory consumption
- Prevent unnecessary database queries

Database Schema Overview

The application uses a relational database with the following main entities:

Core Tables

Users :

- Stores User Account Information
- One-to-many relationships with folders, contacts and mails

user_id	email	full_name	folders	contacts	mails

Mails:

- Contains email content
- References sender (User) as foreign key

mail_id	sender_id	subject	body	sent_at	is_draft	is_deleted	user

UserMails:

- Junction table linking users to received emails
- Enables multiple users to receive the same email with individual read status and folder placement

user_mail_id	user_id	mail_id	folder_id	is_sent	is_read	importance	is_archived

Folders:

- Organizes email into categories
- Each user has their own set of folders

folder_id	user_id	folder_name	is_system_folder

Contacts:

- Stores user contact information
- Supports multiple email addresses per contact through contactEmails Table

contact_id	user_id	contact_name	primary_email

Attachments:

- Manages email file attachments

attachment_id	mail_id	file_name	storage_path	file_type	file_size

MailReceivers

- Tracks all recipients of an email

mail_receiver_id	mail_id	receiver_id	receiver_type

MailFilters

- Tracks all filters of a user

mail_filter_id	user_id	from_address	subject	hasAttachment	priority	folder_id

---

Application Features

User Authentication

- User registration with validation (unique email, password length)
- Secure login with credential verification
- Session management
- Error messaging for failed authentication attempts

Email Management

- View paginated inbox and other folders
- Sort emails by date (descending by default)
- Mark emails as read/unread
- Organize emails by priority level
- Support for email attachments

Folder System

- Pre-configured system folders (Inbox, Sent, Trash, Drafts)
- Create custom folders
- Rename custom folders
- Delete custom folders
- System folder protection (cannot be modified)

Contact Management

- Store contact information
- Support multiple email addresses per contact
- Quick access to frequently used contacts

Email Composition

- Compose new emails, Specify multiple recipients (To, CC, BCC)
- Add attachments
- Set priority levels
- Save drafts: Drafts are saved automatically as the user composes the message. In addition, a Save Draft button is provided to give users explicit reassurance that their progress has been preserved.

Pagination

- Configurable page size (default 10 emails per page)
- Navigation controls (first, previous, next, last), Page number information, Total element count.

Priority Inbox Logic

- **Data Structure:** The Priority mode utilizes a PriorityQueue in Java to sort emails based on importance levels.
- **Algorithm:** A custom comparator prioritizes emails with higher importance values (supporting 4 distinct levels); if importance is equal, it sorts by the most recent sentAt date.

Automated Trash Management

- **Scheduling:** The EmailCleanupService uses the **@Scheduled** annotation to run a daily cleanup at 2:00 AM.
- **Logic:** The service identifies all UserMail entries in the "Trash" folder where the current date exceeds the movedAt date by 30 days and removes them permanently from the database.

Attachment Handling

- **Storage:** Files are saved to a dedicated uploads/ directory on the server disk.
- **Security:** To prevent naming collisions, filenames are prefixed with a unique UUID before storage.
- **Metadata:** Information such as file size and type is stored in the database and linked to the corresponding Mail entity.

Search Functionality

The system provides a comprehensive search feature designed to help users efficiently locate emails within large mailboxes. Two levels of search are supported:

Quick Search

Quick Search allows users to rapidly find emails using a single search bar. As the user types and presses enter, the system searches across all key email attributes such as:

- Sender
- Subject
- Body
- Date

Advanced Search

Advanced Search offers more precise control for users who need refined results. It allows filtering emails based on multiple criteria, including:

- Sender (From)
- Subject keywords
- Priority level
- Presence of attachments
- Date range
- Folder or all mail

Email Filtering and Automatic Folder Routing

The system includes a Create Filter feature that enables users to automatically organize incoming emails. Users can define custom rules based on specific email attributes, such as:

- Sender (From)
- Subject
- Priority
- Presence of attachments

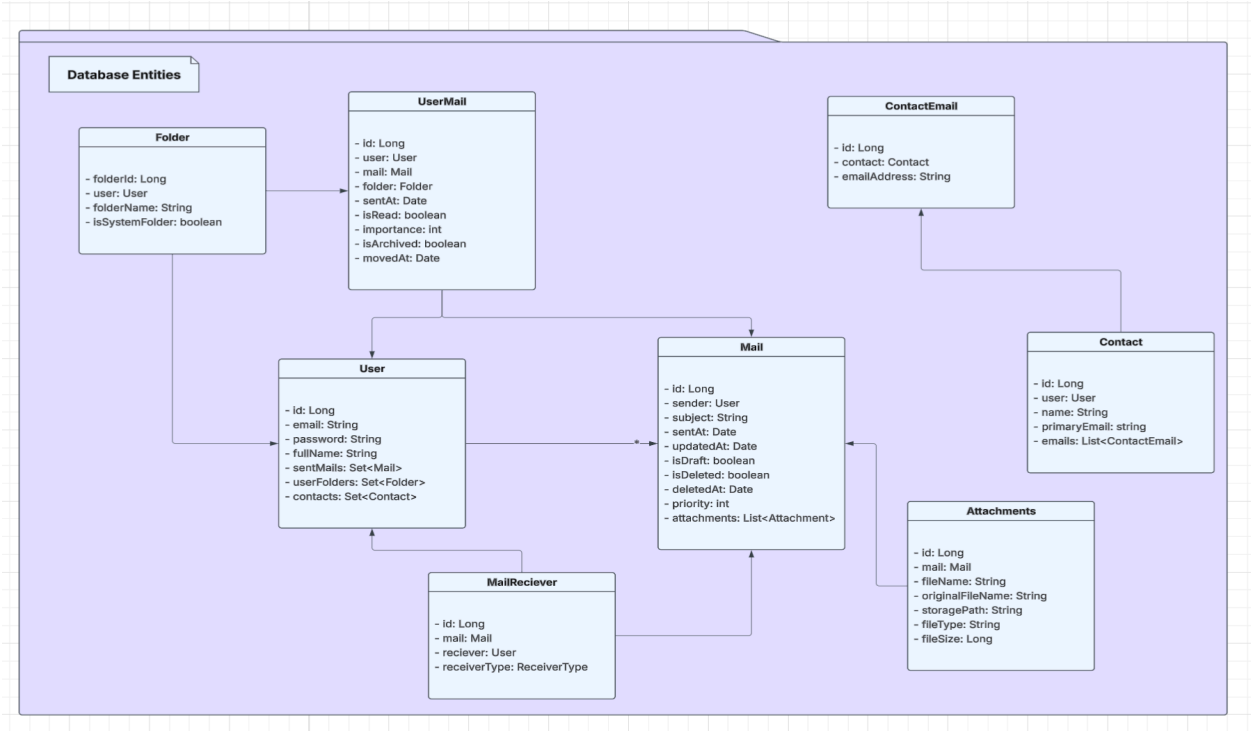
Once a filter is created, emails that match the defined conditions are automatically directed to a folder specified by the user. This reduces manual sorting, improves inbox organization, and enhances overall productivity.

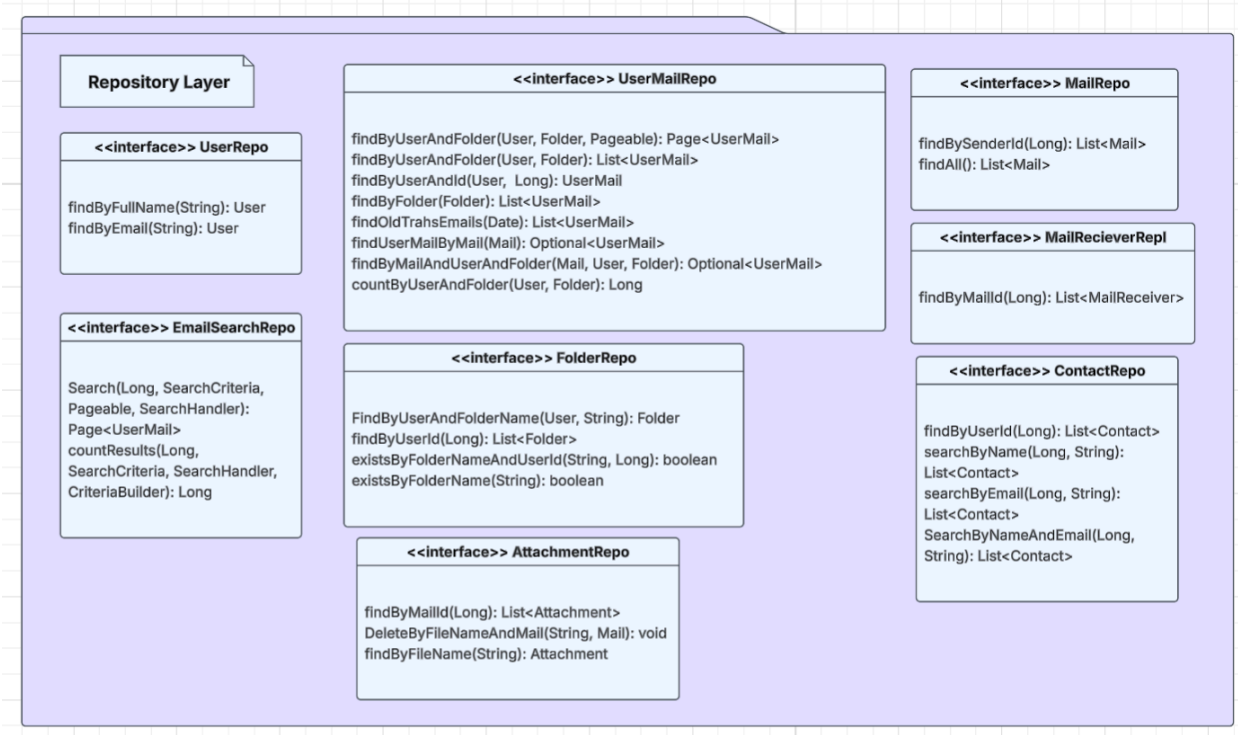
AI-Enhanced features:

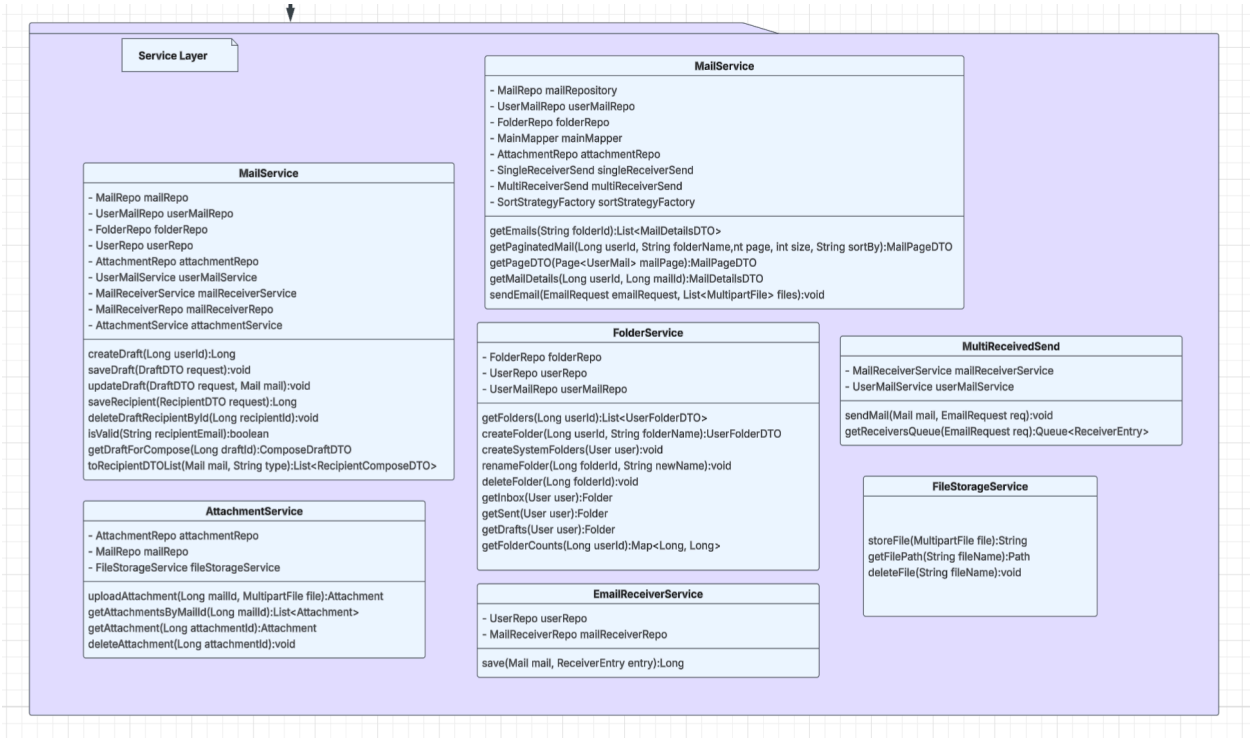
Voice-Based Email Search: Enables users to search for emails using spoken commands (e.g., sender, subject, or keywords), improving accessibility and allowing faster, hands-free interaction.

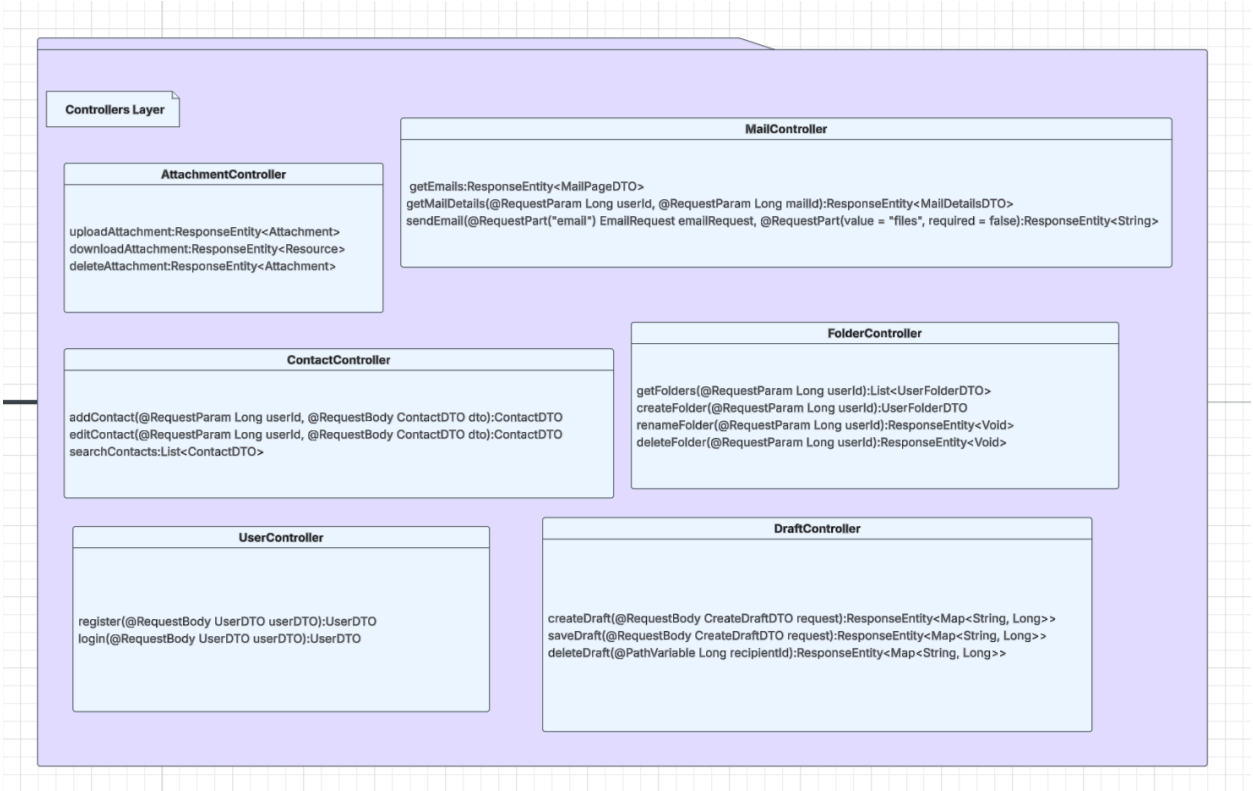
AI Email Summarization: Automatically generates short summaries of long emails, helping users quickly grasp key information and reduce time spent reading.

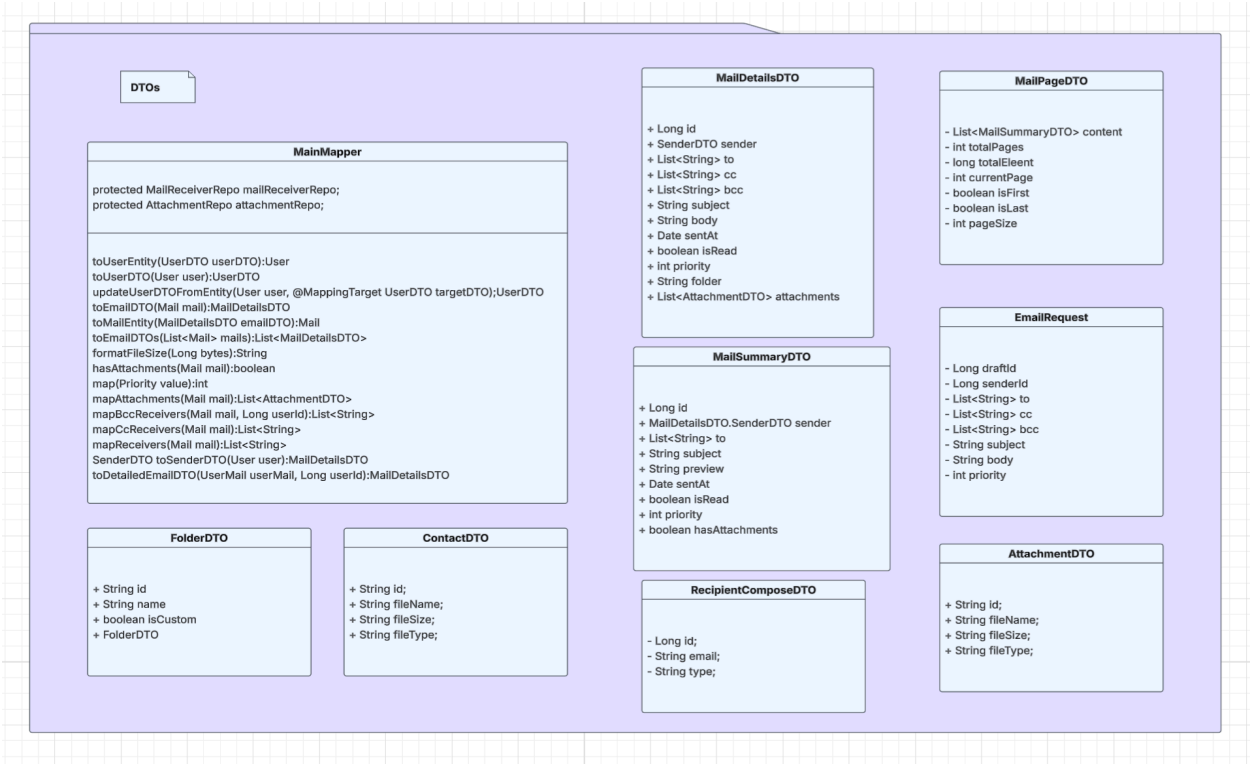
UML Diagram


Database Entities
Folder
- folderId: Long
- user: User
- folderName: String
- isSystemFolder: boolean
UserMail
- id: Long
- user: User
- mail: Mail
- folder: Folder
- sentAt: Date
- isRead: boolean
- importance: int
- isArchived: boolean
- movedAt: Date
ContactEmail
- id: Long
- contact: Contact
- emailAddress: String
User
- id: Long
- email: String
- password: String
- fullName: String
- sentMails: Set<Mail>
- userFolders: Set<Folder>
- contacts: Set<Contact>
Mail
- id: Long
- sender: User
- subject: String
- sentAt: Date
- updatedAt: Date
- isDraft: boolean
- isDeleted: boolean
- deletedAt: Date
- priority: int
- attachments: List<Attachment>
Contact
- id: Long
- user: User
- name: String
- primaryEmail: string
- emails: List<ContactEmail>
Attachments
- id: Long
- mail: Mail
- fileName: String
- originalFileName: String
- storagePath: String
- fileType: String
- fileSize: Long
MailReciever
- id: Long
- mail: Mail
- reciever: User
- receiverType: ReceiverType

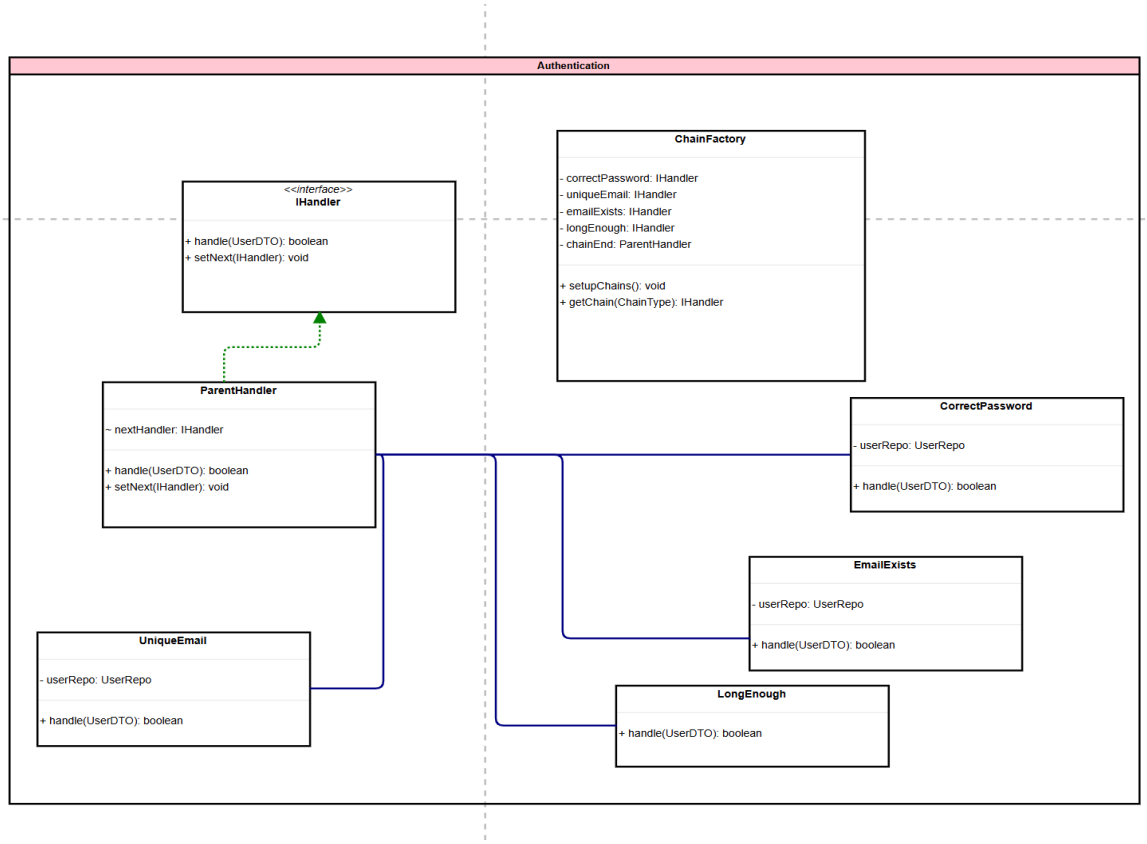

Repository Layer
<<interface>> UserRepo
findByFullName(String): User
findByEmail(String): User
<<interface>> EmailSearchRepo
Search(Long, SearchCriteria, Pageable, SearchHandler): Page<UserMail>
countResults(Long, SearchCriteria, SearchHandler, CriteriaBuilder): Long
<<interface>> UserMailRepo
findByUserAndFolder(User, Folder, Pageable): Page<UserMail>
findByUserAndFolder(User, Folder): List<UserMail>
findByUserAndId(User, Long): UserMail
findByFolder(Folder): List<UserMail>
findOldTrahsEmails(Date): List<UserMail>
findUserMailByMail(Mail): Optional<UserMail>
findByMailAndUserAndFolder(Mail, User, Folder): Optional<UserMail>
countByUserAndFolder(User, Folder): Long
<<interface>> FolderRepo
FindByUserAndFolderName(User, String): Folder
findByUserId(Long): List<Folder>
existsByFolderNameAndUserId(String, Long): boolean
existsByFolderName(String): boolean
<<interface>> AttachmentRepo
findByMailId(Long): List<Attachment>
DeleteByFileNameAndMail(String, Mail): void
findByFileName(String): Attachment
<<interface>> MailRepo
findBySenderId(Long): List<Mail>
findAll(): List<Mail>
<<interface>> MailRecieverRepl
findByMailId(Long): List<MailReceiver>
<<interface>> ContactRepo
findByUserId(Long): List<Contact>
searchByName(Long, String): List<Contact>
searchByEmail(Long, String): List<Contact>
SearchByNameAndEmail(Long, String): List<Contact>


Service Layer
MailService
- MailRepo mailRepository
- UserMailRepo userMailRepo
- FolderRepo folderRepo
- MainMapper mainMapper
- AttachmentRepo attachmentRepo
- SingleReceiverSend singleReceiverSend
- MultiReceiverSend multiReceiverSend
- SortStrategyFactory sortStrategyFactory
getEmails(String folderId):List<MailDetailsDTO>
getPaginatedMail(Long userId, String folderName,nt page, int size, String sortBy):MailPageDTO
getPageDTO(Page<UserMail> mailPage):MailPageDTO
getMailDetails(Long userId, Long mailId):MailDetailsDTO
sendEmail(EmailRequest emailRequest, List<MultipartFile> files):void
MailService
- MailRepo mailRepo
- UserMailRepo userMailRepo
- FolderRepo folderRepo
- UserRepo userRepo
- AttachmentRepo attachmentRepo
- UserMailService userMailService
- MailReceiverService mailReceiverService
- MailReceiverRepo mailReceiverRepo
- AttachmentService attachmentService
createDraft(Long userId):Long
saveDraft(DraftDTO request):void
updateDraft(DraftDTO request, Mail mail):void
saveRecipient(RecipientDTO request):Long
deleteDraftRecipientById(Long recipientId):void
isValid(String recipientEmail):boolean
getDraftForCompose(Long draftId):ComposeDraftDTO
toRecipientDTOList(Mail mail, String type):List<RecipientComposeDTO>
FolderService
- FolderRepo folderRepo
- UserRepo userRepo
- UserMailRepo userMailRepo
getFolders(Long userId):List<UserFolderDTO>
createFolder(Long userId, String folderName):UserFolderDTO
createSystemFolders(User user):void
renameFolder(Long folderId, String newName):void
deleteFolder(Long folderId):void
getInbox(User user):Folder
getSent(User user):Folder
getDrafts(User user):Folder
getFolderCounts(Long userId):Map<Long, Long>
MultiReceivedSend
- MailReceiverService mailReceiverService
- UserMailService userMailService
sendMail(Mail mail, EmailRequest req):void
getReceiversQueue(EmailRequest req):Queue<ReceiverEntry>
AttachmentService
- AttachmentRepo attachmentRepo
- MailRepo mailRepo
- FileStorageService fileStorageService
uploadAttachment(Long mailId, MultipartFile file):Attachment
getAttachmentsByMailId(Long mailId):List<Attachment>
getAttachment(Long attachmentId):Attachment
deleteAttachment(Long attachmentId):void
EmailReceiverService
- UserRepo userRepo
- MailReceiverRepo mailReceiverRepo
save(Mail mail, ReceiverEntry entry):Long
FileStorageService
storeFile(MultipartFile file):String
getFilePath(String fileName):Path
deleteFile(String fileName):void

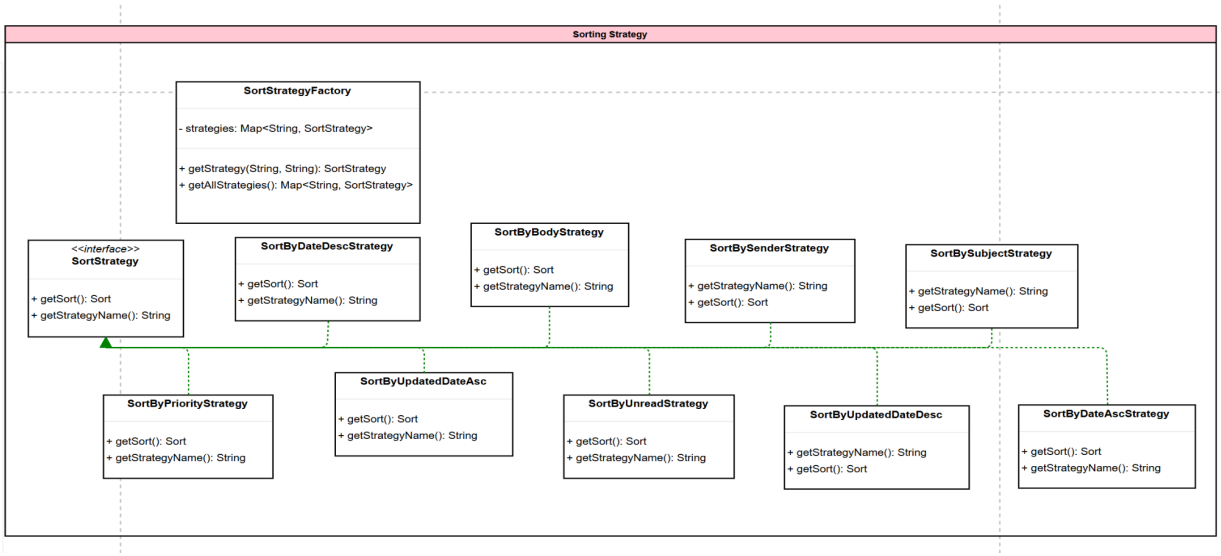

Controllers Layer
MailController
getEmails:ResponseEntity<MailPageDTO>
getMailDetails(@RequestParam Long userId, @RequestParam Long mailId):ResponseEntity<MailDetailsDTO>
sendEmail(@RequestPart("email") EmailRequest emailRequest, @RequestPart(value = "files", required = false):ResponseEntity<String>
AttachmentController
uploadAttachment:ResponseEntity<Attachment>
downloadAttachment:ResponseEntity<Resource>
deleteAttachment:ResponseEntity<Attachment>
FolderController
getFolders(@RequestParam Long userId):List<UserFolderDTO>
createFolder(@RequestParam Long userId):UserFolderDTO
renameFolder(@RequestParam Long userId):ResponseEntity<Void>
deleteFolder(@RequestParam Long userId):ResponseEntity<Void>
ContactController
addContact(@RequestParam Long userId, @RequestBody ContactDTO dto):ContactDTO
editContact(@RequestParam Long userId, @RequestBody ContactDTO dto):ContactDTO
searchContacts:List<ContactDTO>
UserController
register(@RequestBody UserDTO userDTO):UserDTO
login(@RequestBody UserDTO userDTO):UserDTO
DraftController
createDraft(@RequestBody CreateDraftDTO request):ResponseEntity<Map<String, Long>>
saveDraft(@RequestBody CreateDraftDTO request):ResponseEntity<Map<String, Long>>
deleteDraft(@PathVariable Long recipientId):ResponseEntity<Map<String, Long>>


DTOs
MainMapper
protected MailReceiverRepo mailReceiverRepo;
protected AttachmentRepo attachmentRepo;
toUserEntity(UserDTO userDTO):User
toUserDTO(User user):UserDTO
updateUserDTOFromEntity(User user, @MappingTarget UserDTO targetDTO);UserDTO
toEmailDTO(Mail mail):MailDetailsDTO
toMailEntity(MailDetailsDTO emailDTO):Mail
toEmailDTOs(List<Mail> mails):List<MailDetailsDTO>
formatFileSize(Long bytes):String
hasAttachments(Mail mail):boolean
map(Priority value):int
mapAttachments(Mail mail):List<AttachmentDTO>
mapBccReceivers(Mail mail, Long userId):List<String>
mapCcReceivers(Mail mail):List<String>
mapReceivers(Mail mail):List<String>
SenderDTO toSenderDTO(User user):MailDetailsDTO
toDetailedEmailDTO(UserMail userMail, Long userId):MailDetailsDTO
FolderDTO
+ String id
+ String name
+ boolean isCustom
+ FolderDTO
ContactDTO
+ String id;
+ String fileName;
+ String fileSize;
+ String fileType;
MailDetailsDTO
+ Long id
+ SenderDTO sender
+ List<String> to
+ List<String> cc
+ List<String> bcc
+ String subject
+ String body
+ Date sentAt
+ boolean isRead
+ int priority
+ String folder
+ List<AttachmentDTO> attachments
MailSummaryDTO
+ Long id
+ MailDetailsDTO.SenderDTO sender
+ List<String> to
+ String subject
+ String preview
+ Date sentAt
+ boolean isRead
+ int priority
+ boolean hasAttachments
RecipientComposeDTO
- Long id;
- String email;
- String type;
MailPageDTO
- List<MailSummaryDTO> content
- int totalPages
- long totalEleent
- int currentPage
- boolean isFirst
- boolean isLast
- int pageSize
EmailRequest
- Long draftId
- Long senderId
- List<String> to
- List<String> cc
- List<String> bcc
- String subject
- String body
- int priority
AttachmentDTO
+ String id;
+ String fileName;
+ String fileSize;
+ String fileType;

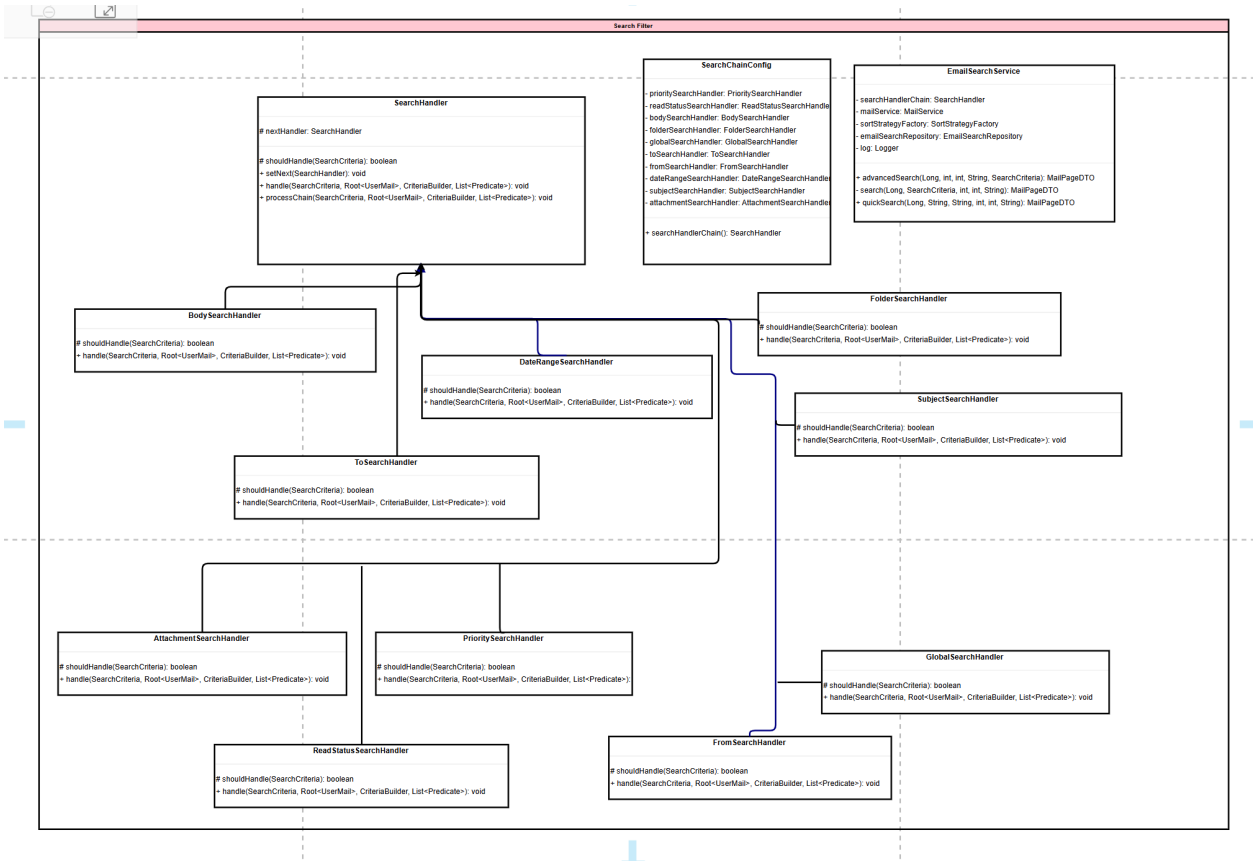
User Authentication (Filter/ Chain Of Responsibiliy)



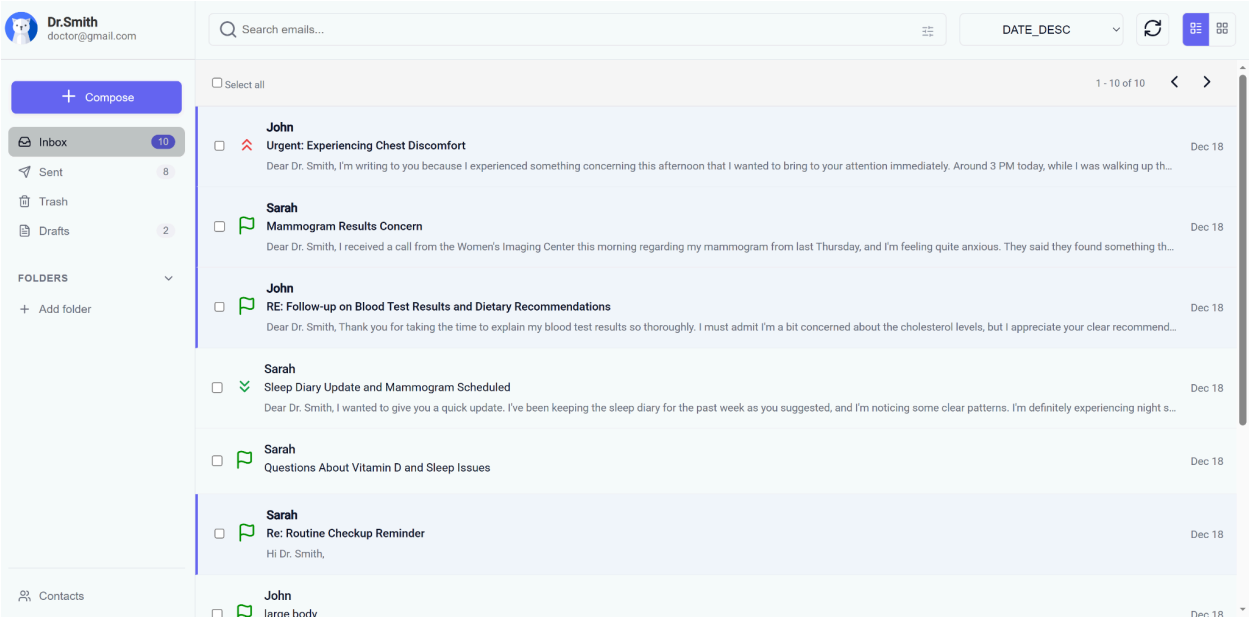
Sorting Emails (Strategy)

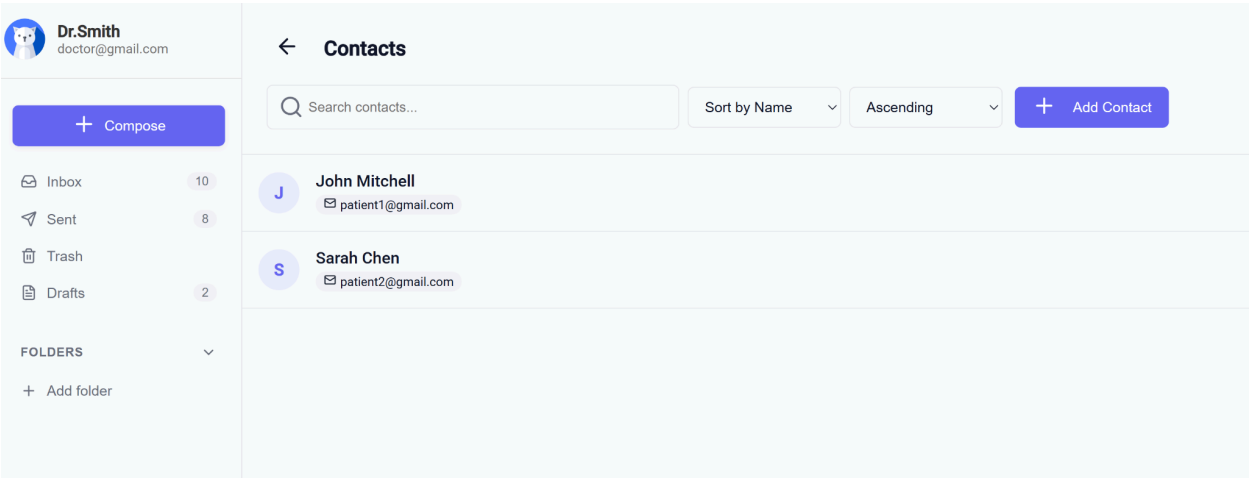


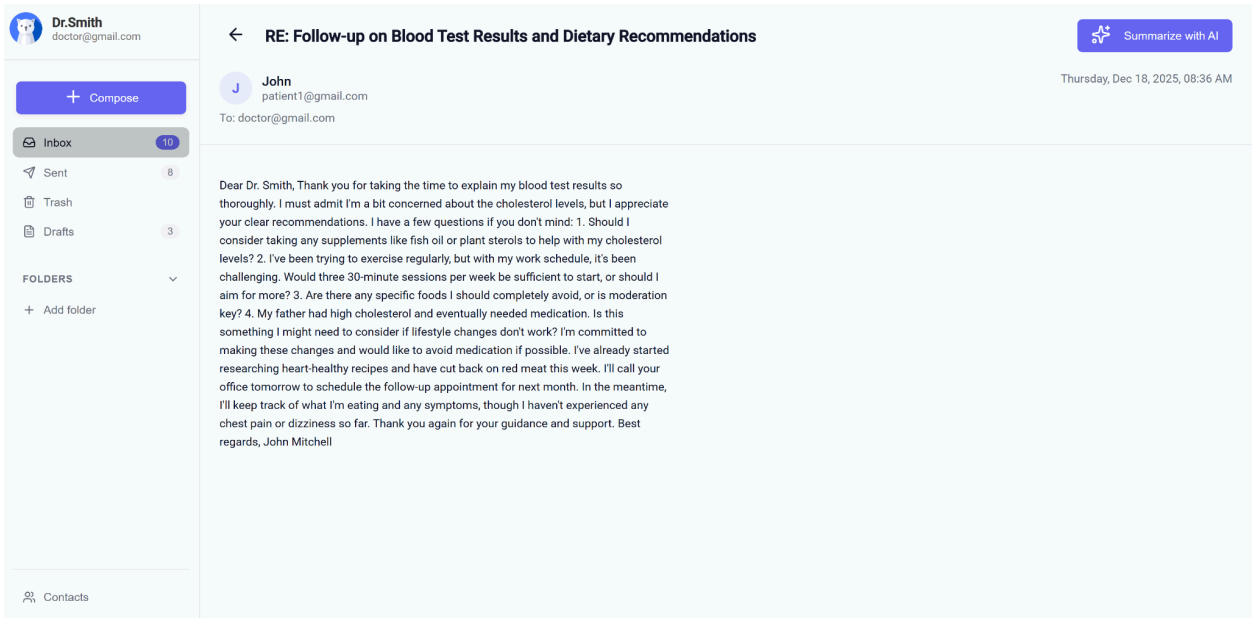
Searching Emails (Filter/ Chain Of Responsibility)

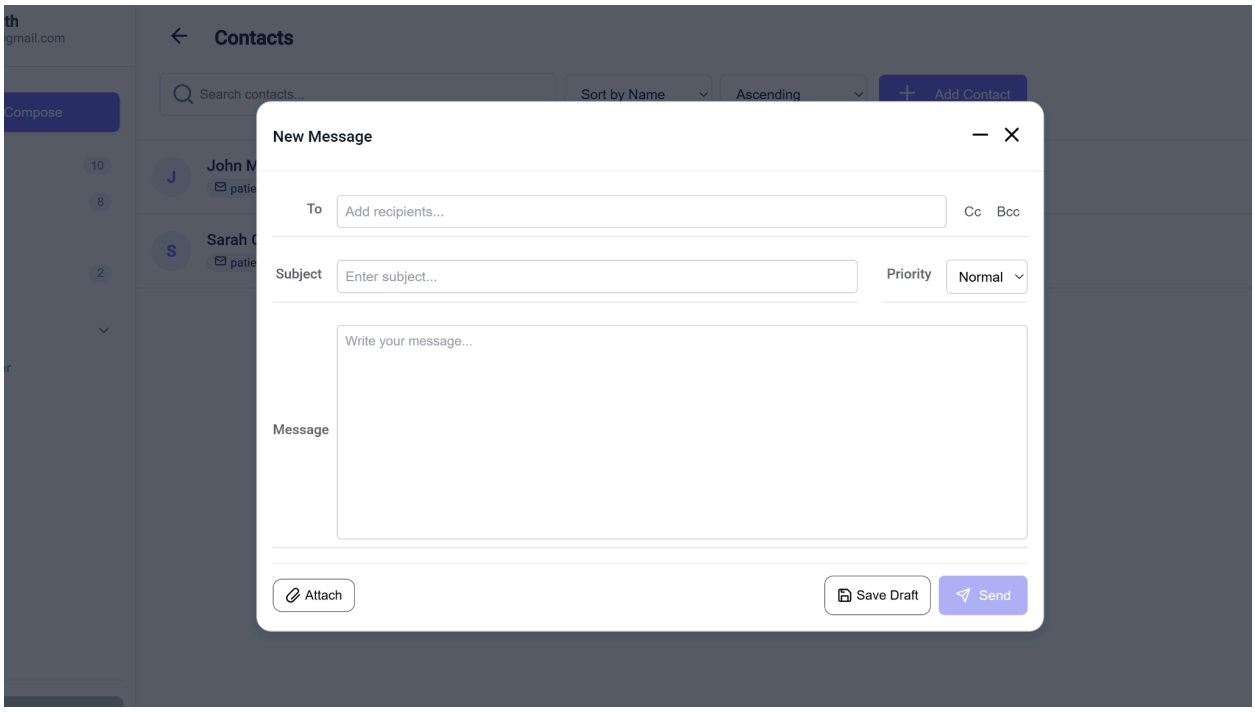


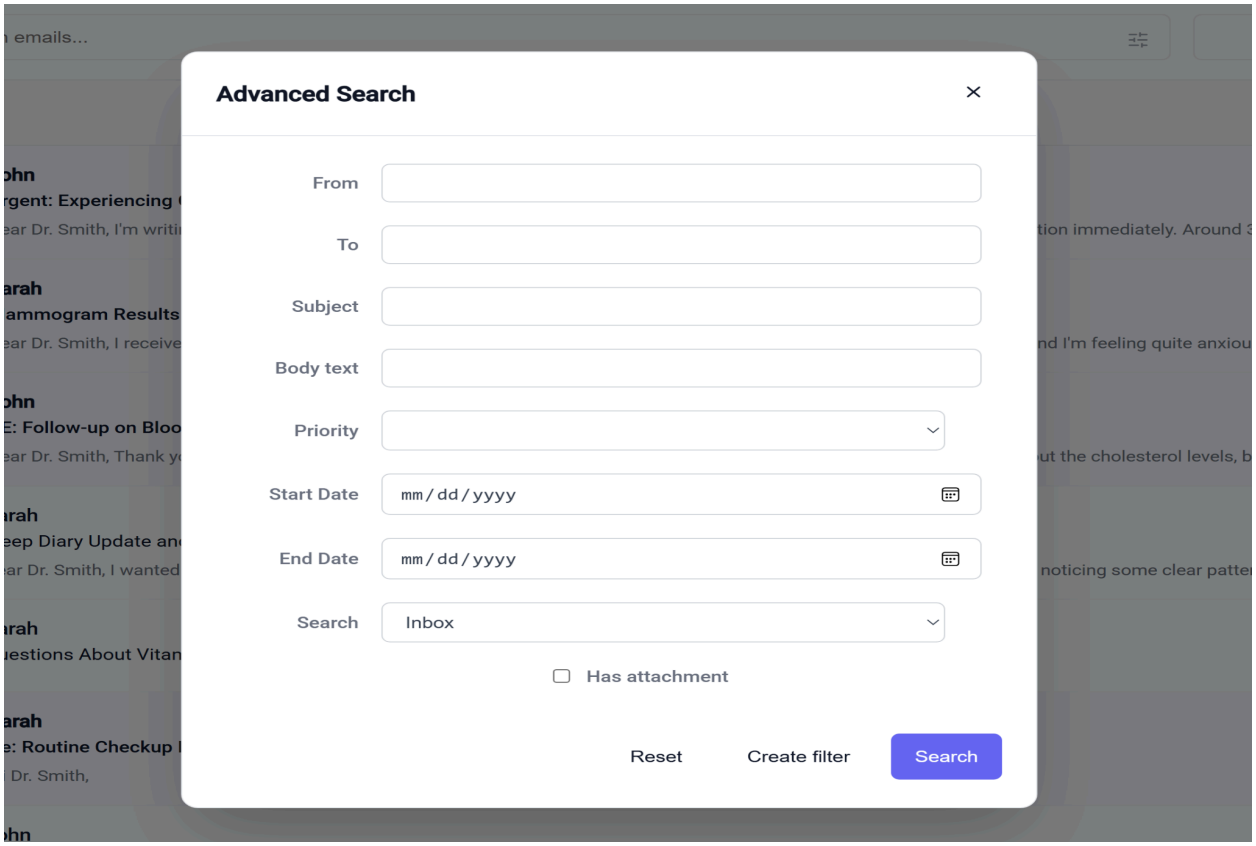
System Screenshots

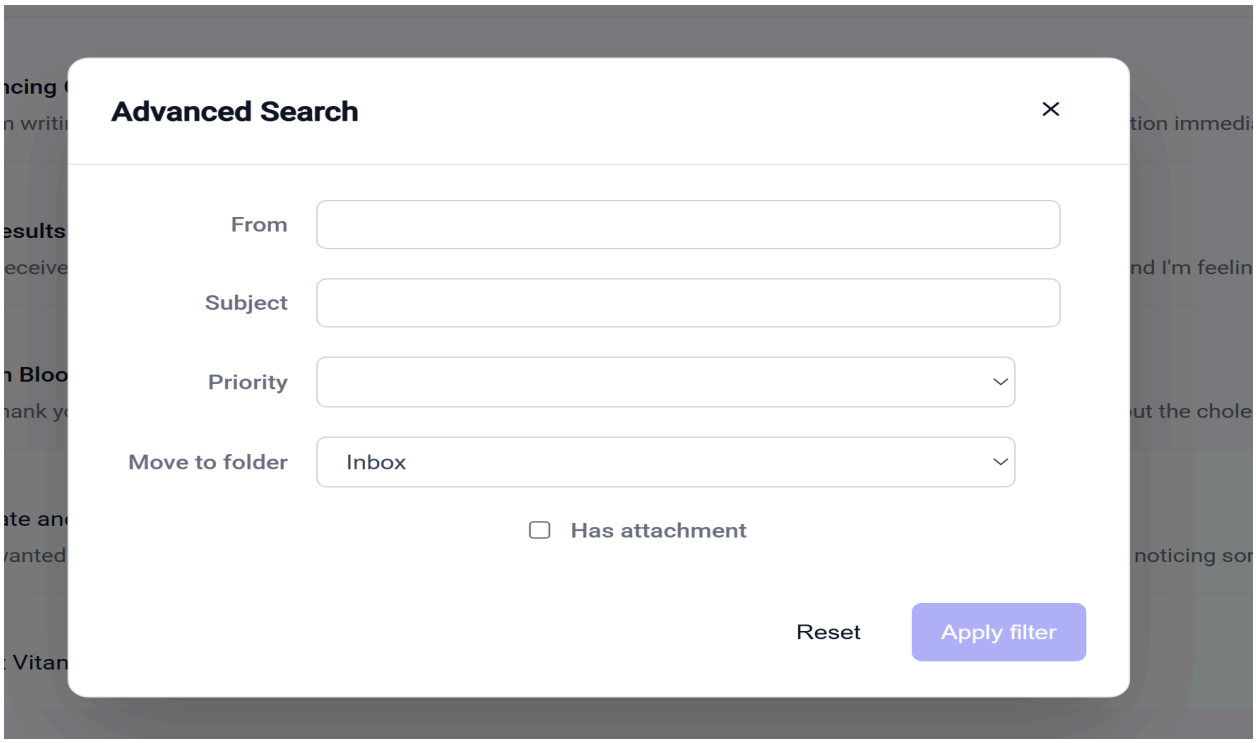





Dr.Smith
doctor@gmail.com
Compose
Inbox 10
Sent 8
Trash
Drafts 3
FOLDERS
Add folder
Contacts
RE: Follow-up on Blood Test Results and Dietary Recommendations
Summarize with AI
John
patient1@gmail.com
Thursday, Dec 18, 2025, 08:36 AM
To: doctor@gmail.com
Dear Dr. Smith, Thank you for taking the time to explain my blood test results so thoroughly. I must admit I'm a bit concerned about the cholesterol levels, but I appreciate your clear recommendations. I have a few questions if you don't mind: 1. Should I consider taking any supplements like fish oil or plant sterols to help with my cholesterol levels? 2. I've been trying to exercise regularly, but with my work schedule, it's been challenging. Would three 30-minute sessions per week be sufficient to start, or should I aim for more? 3. Are there any specific foods I should completely avoid, or is moderation key? 4. My father had high cholesterol and eventually needed medication. Is this something I might need to consider if lifestyle changes don't work? I'm committed to making these changes and would like to avoid medication if possible. I've already started researching heart-healthy recipes and have cut back on red meat this week. I'll call your office tomorrow to schedule the follow-up appointment for next month. In the meantime, I'll keep track of what I'm eating and any symptoms, though I haven't experienced any chest pain or dizziness so far. Thank you again for your guidance and support. Best regards, John Mitchell


Contacts
Search contacts...
Sort by Name
Ascending
Add Contact
New Message
To
Add recipients...
Cc
Bcc
Subject
Enter subject...
Priority
Normal
Message
Write your message...
Attach
Save Draft
Send


Advanced Search
From
To
Subject
Body text
Priority
Start Date
mm / dd / yyyy
End Date
mm / dd / yyyy
Search
Inbox
Has attachment
Reset
Create filter
Search


Advanced Search
From
Subject
Priority
Move to folder
Inbox
Has attachment
Reset
Apply filter

Priority vs Default view