Mail Server Project Report

Course: CSE 223: Programming 2
Faculty: Faculty of Engineering, Alexandria University
Department: Computer and Systems Engineering
Assignment: #4 - Web-based Mail Program
Team Members:

- Roaa Mostafa Mohamed (23010402)
- Salma Karem Mahfouz (23010462)
- Abdelrahman Ibrahim Mohamed (23010506)
- Zuhair Qussai Jarrar (23011929)

---

1. Steps to Run the Code:

Backend (Spring Boot)

1. **Prerequisites:**

- Java 17+ installed.
- Maven installed.
- H2 database (in-memory or file-based as in project).

2. **Steps:**

```
# Clone the repository
git clone <repo-url>

# Move into project folder
cd MailServer/backend

# Install dependencies and build
mvn clean install

# Run the backend server
mvn spring-boot:run
```

- The backend server runs by default on http://localhost:8080/.

3. **Configurations**

- "application.properties" contains database paths and server settings.

- Default attachment directory: `"C:\Users\<username>\Downloads\Server"` (adjust path as needed).

Frontend (Angular)

1. **Prerequisites:**

- Node.js 18+ installed.
- Angular CLI installed globally:

```
npm install -g @angular/cli
```

2. **Steps:**

```
# Navigate to frontend folder
cd MailServer_frontend

# Install dependencies
npm install

# Run frontend server
ng serve
```

- The frontend runs by default on http://localhost:4200/.

3. **Interaction with Backend:**

- Angular services handle API calls to the Spring Boot backend (`http://localhost:8080/`).

---

2. REST API Endpoints:

Authentication (/api/auth)

• **POST** `/api/auth/signIn` → Sign in user
Body: **UserSigninDTO**

• **POST** `/api/auth/signUp` → Register new user
Body: **UserSignupDTO**

• **GET** `/api/auth/get/all` → Get all users
Body: **None**

Mail Filters (/api/filters)

- **GET** `/api/filters/by-user/{userId}` → Get all filters for user
  Body: **None**

- **GET** `/api/filters/{filterId}` → Get filter
  Body: **None**

- **POST** `/api/filters` → Create filter
  Body: **MailFilterDto**

- **PUT** `/api/filters/{filterId}` → Update filter
  Body: **MailFilterDto**

- **DELETE** `/api/filters/{filterId}` → Delete filter
  Body: **None**

Mails (/api/mails)

- **POST** `/api/mails` → Compose mail
  Body: **MailDto**

- **GET** `/api/mails/valid/{email}` → Check valid email
  Body: **None**

- **GET** `/api/mails/{folderId}/{mailId}` → Get mail details
  Body: **None**

- **DELETE** `/api/mails/{folderId}` → Delete mails
  Body: **None**

- **PATCH** `/api/mails/{mailId}/read-status` → Mark mail as read
  Body: **None**

- **POST** `/api/mails/filter` → Filter mails
  Body: **MailSearchRequestDto**

- **GET** `/api/mails` → Get all mails
  Body: **None**

- **GET** `/api/mails/search` → Search mails
  Body: **None**

- **PATCH** `/api/mails/{toFolderId}/{fromFolderId}` → Move mails
  Body: **MoveMailDto**

- **GET** `/api/mails/sort` → Sort mails
  Body: **None**

- **PATCH** `/api/mails` → Undo
  Body: **List<String>**

- **DELETE** `/api/mails` → Delete forever
  Body: **List<String>**

Folders (/api/folders)

- **GET** `/api/folders/{userId}` → Get folders for user
  Body: **None**

- **DELETE** `/api/folders/{folderId}/{userId}` → Delete folder
  Body: **None**

- **POST** `/api/folders` → Create folder
  Body: **FolderDto**

- **PUT** `/api/folders/{folderId}` → Rename folder
  Body: **None**

- **GET** `/api/folders?type=custom&userId=` → Get custom folders
  Body: **None**

Drafts (/api/drafts)

- **POST** `/api/drafts` → Save draft
  Body: **MailDto**

- **GET** `/api/drafts/{userId}` → Get drafts
  Body: **None**

- **PUT** `/api/drafts/{mailId}` → Update draft
  Body: **MailDto**

- **GET** `/api/drafts/{mailId}/snapshots` → Get snapshots
  Body: **None**

- **POST** `/api/drafts/{mailId}/send` → Send draft
  Body: **None**

- **DELETE** `/api/drafts` → Delete drafts forever
  Body: **List<String>**

• **PUT** `/api/drafts/{draftId}/{snapshotId}` → Force snapshot
  Body: **None**

Contacts (/api/contacts)

• **POST** `/api/contacts/{userId}` → Create contact
  Body: **ContactDto**

• **PUT** `/api/contacts/{contactId}` → Edit contact
  Body: **ContactDto**

• **DELETE** `/api/contacts/{contactId}` → Delete contact
  Body: **None**

• **DELETE** `/api/contacts` → Delete multiple contacts
  Body: **List<String>**

• **GET** `/api/contacts/{userId}` → Get contacts
  Body: **None**

• **PATCH** `/api/contacts/{contactId}` → Toggle star
  Body: **None**

Attachments (/api/attachments)

• **POST** `/api/attachments` → Add attachment
  Body: **MultipartFile + List<String>**

• **DELETE** `/api/attachments/delete/{id}` → Delete attachment
  Body: **None**

• **GET** `/api/attachments/download/{id}` → Download attachment
  Body: **None**

AI (/ai)

• **POST** `/ai/body` → Generate email body
  Body: **Map<String, String>** (`prompt`, `type`, `sender`)

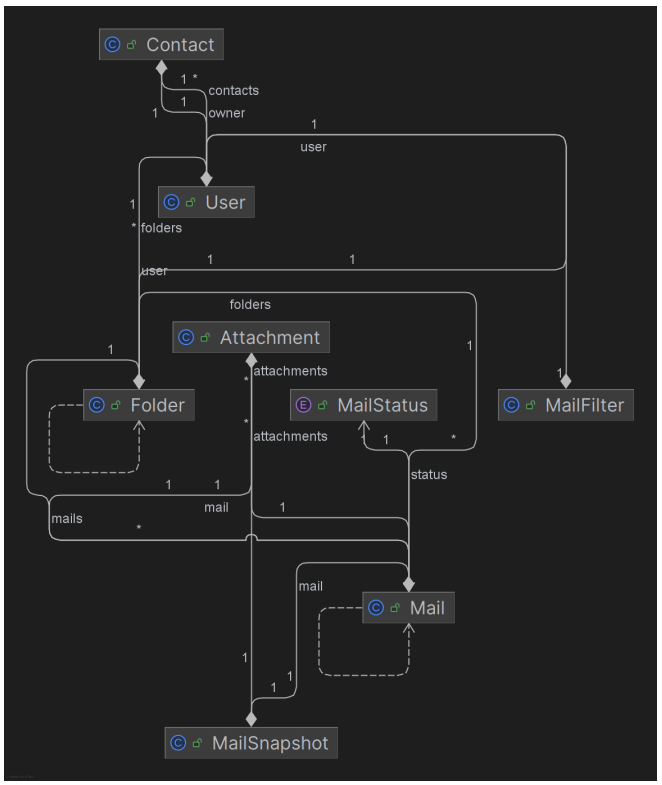
---

3. UML Diagram:

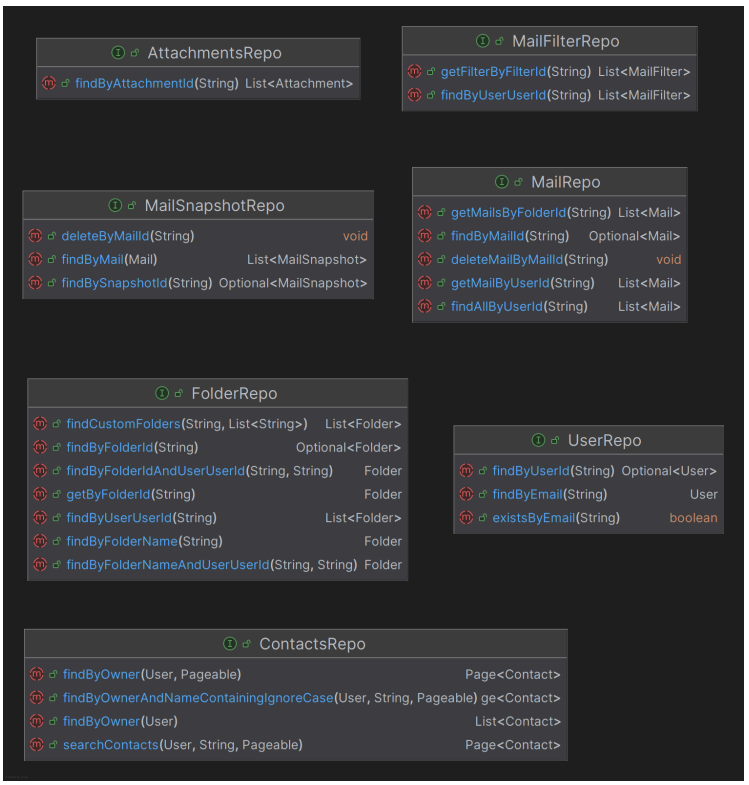
1. **Main Packages:**

- `model` → Entities: `Mail`, `Folder`, `Attachment`, `User`, `...`
- `repo` → Spring Data JPA repositories
- `services` → Business logic (MailService, FolderService, AttachmentService, DraftService)
- `controllers` → REST controllers (MailController, FolderController, DraftController)
- `mappers` → Convert Entities ↔ DTOs
- `DTOs`

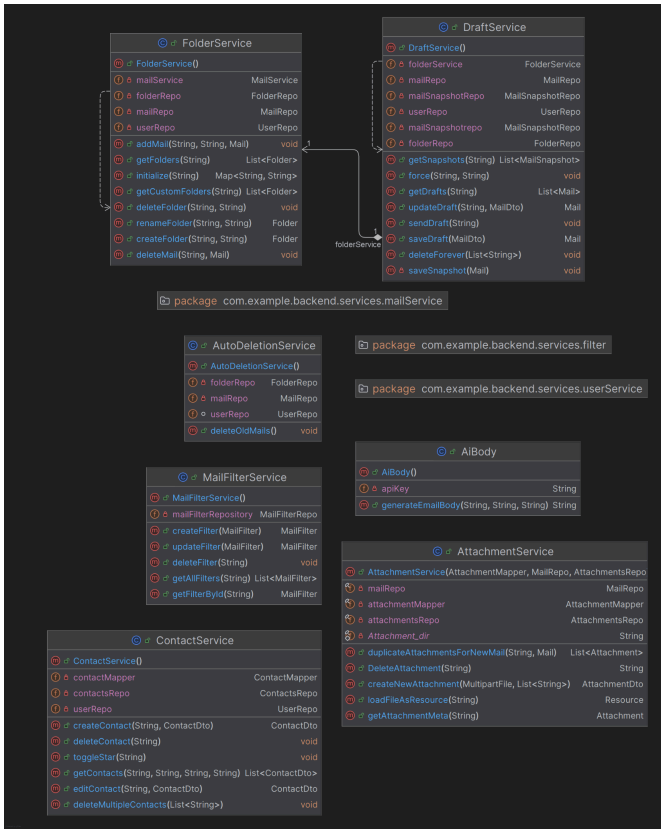
1.Models:



2.Repositories:

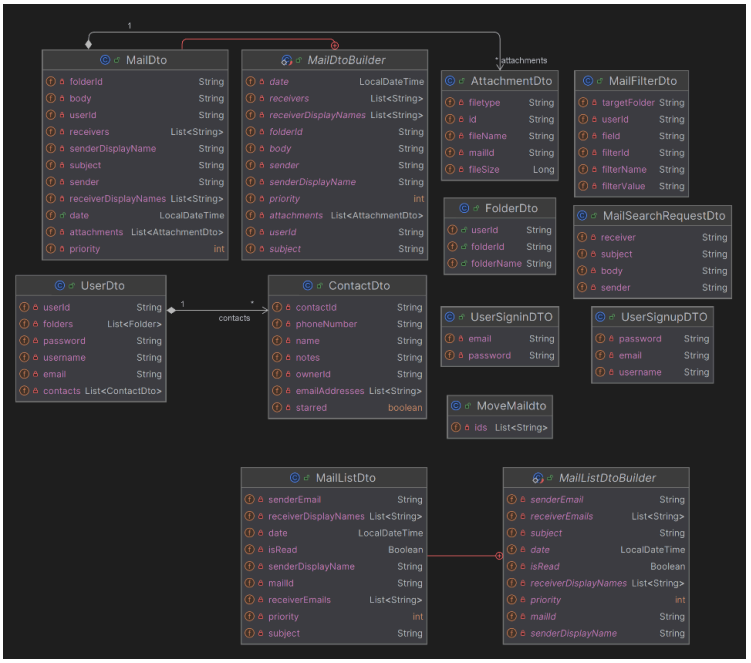


3. Services:

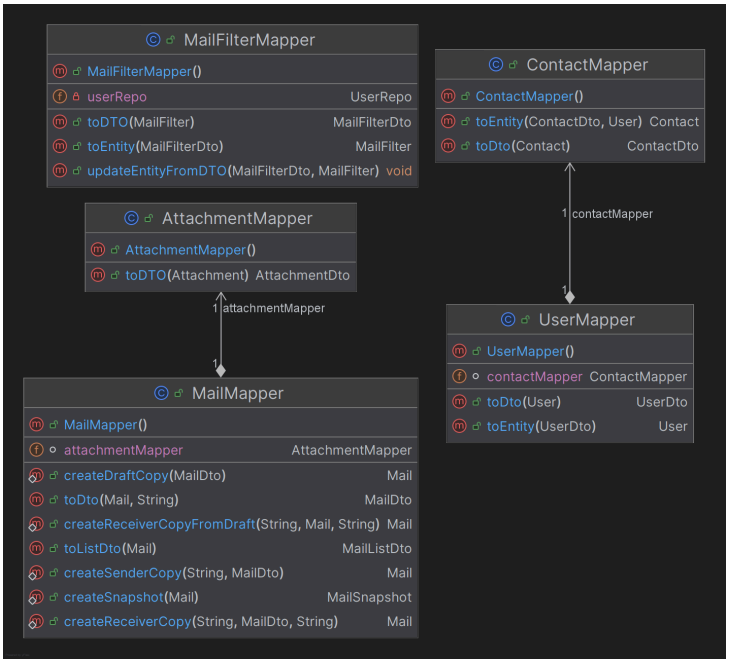


4. Controllers:

5. DTOs:

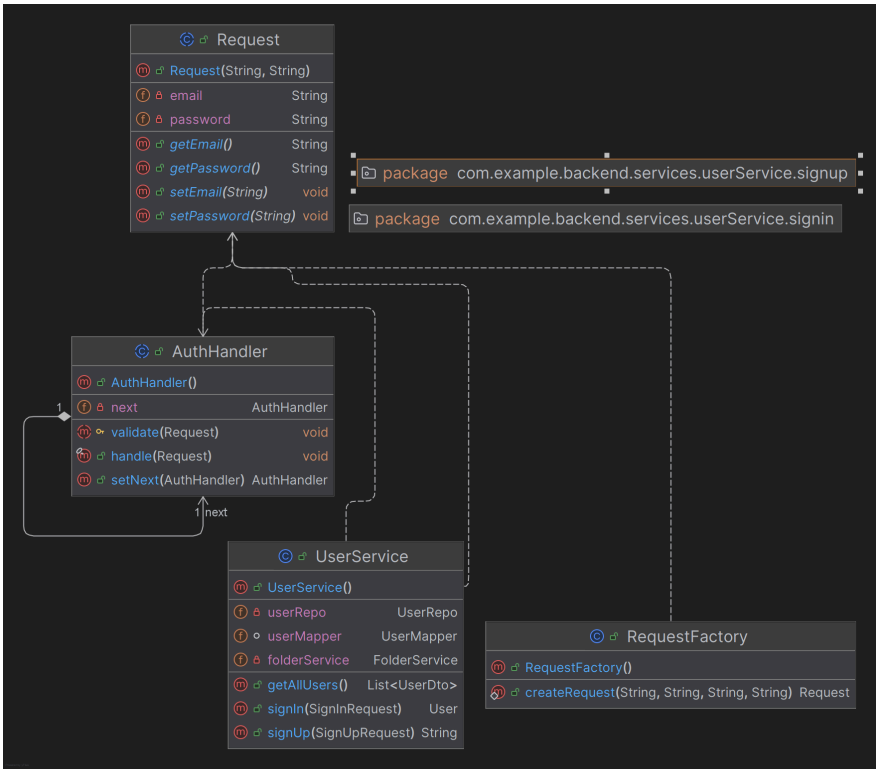


6. Mappers:

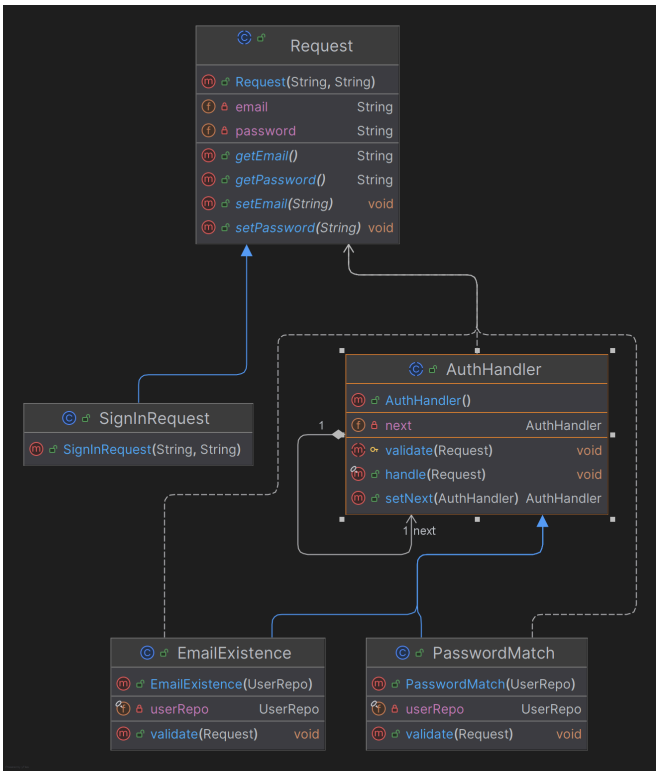


Services Details:

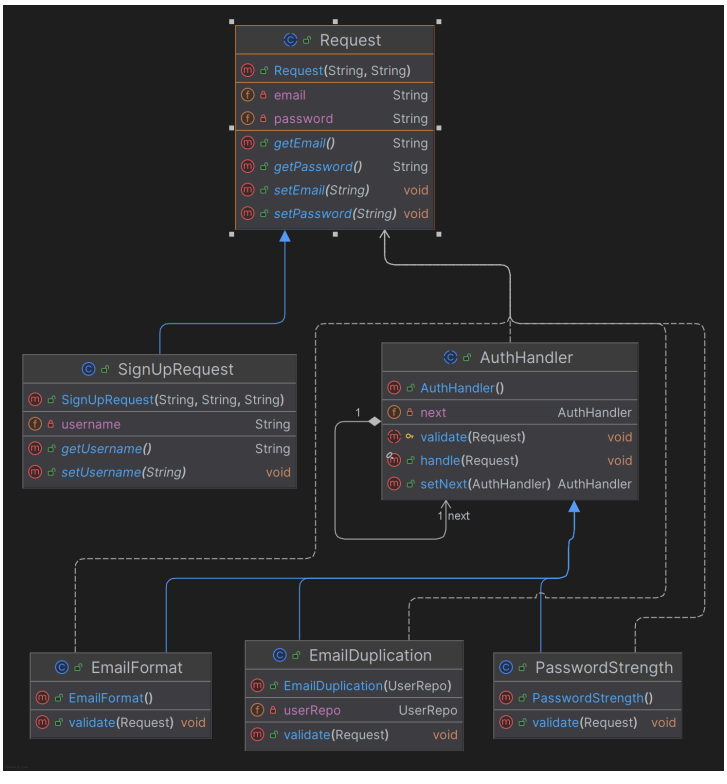
1. UserService:



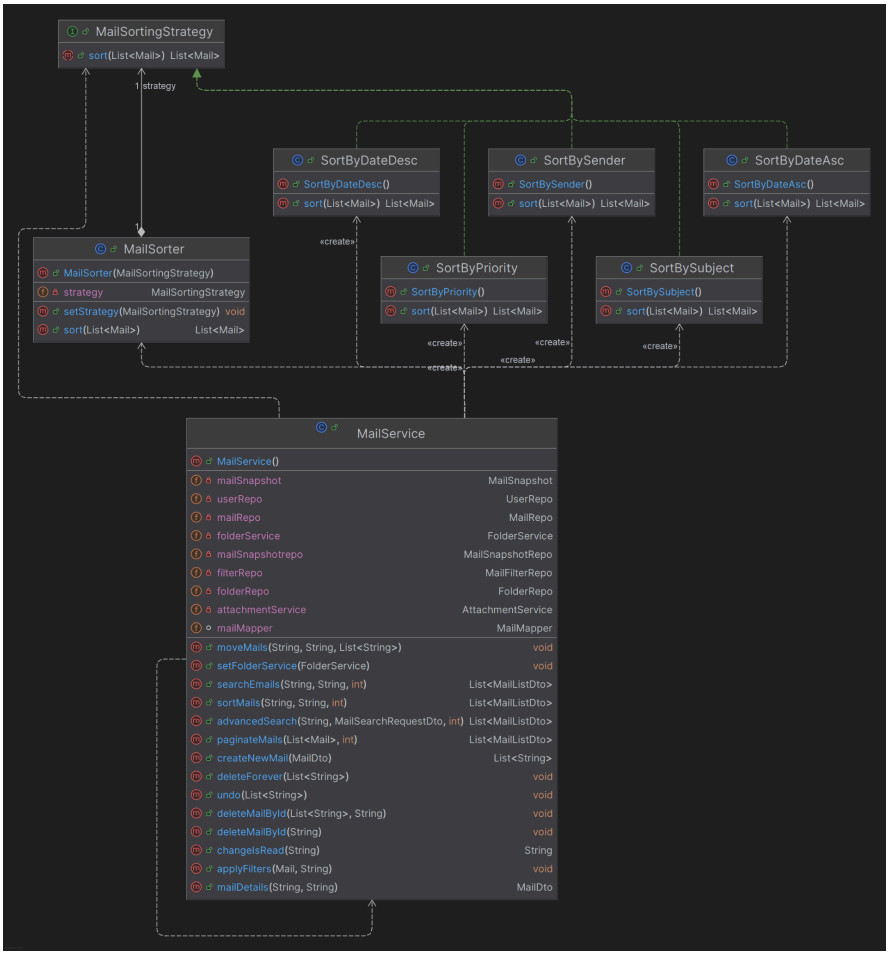
a. Sign in:



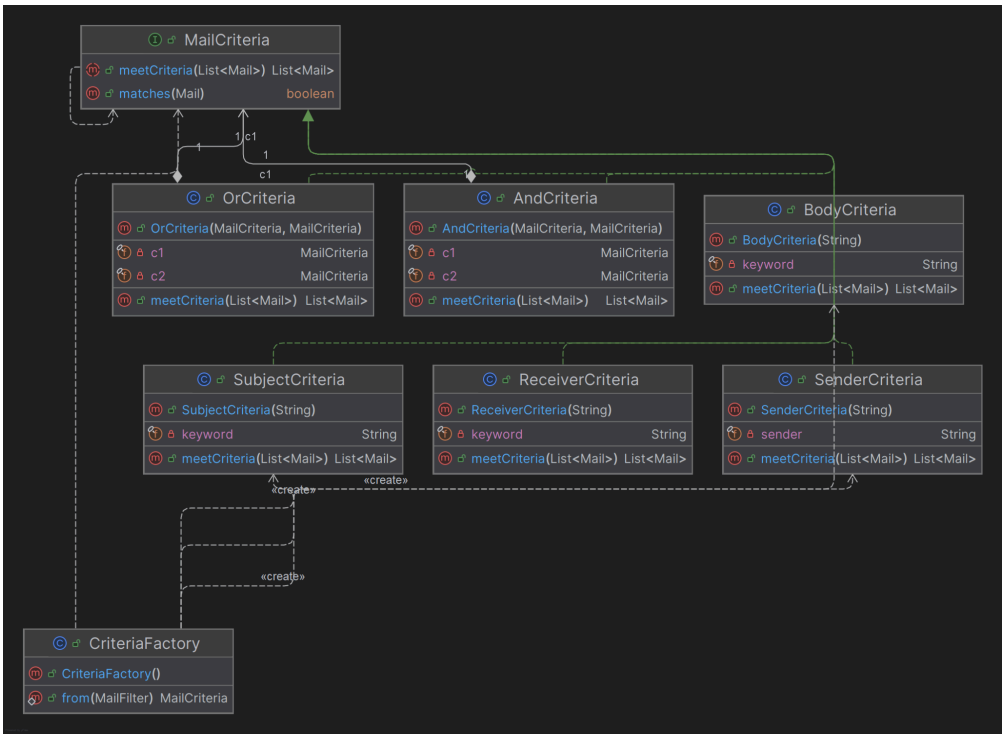
b. Signup:



2. MailService:



3. Filter:



4. Applied Design Patterns:

1. Factory Pattern:

- **Pattern Type:** Creational
- **Purpose:**

- The Factory Pattern is used to create objects without exposing the instantiation logic to the client and provides a common interface for creating different types of objects.

- **Implementation Details:**

a) RequestFactory:

- **Benefits:**

1. Centralizes object creation logic.
2. Makes it easy to add new request types without modifying client code without violating the open-closed principle.
3. Encapsulates the decision-making process for which Request subclass to instantiate.

b) CriteriaFactory:

- **Benefits:**

1. Creates appropriate filter criteria objects based on the filter field
2. Used in automatic mail filtering when new emails arrive
3. Simplifies the filter application logic in `MailService.applyFilters()`

2. Strategy Pattern:

- **Pattern Type:** Behavioral
- **Purpose:**

- The Strategy Pattern defines a family of algorithms (sorting algorithms in this case), encapsulates each one, and makes them interchangeable. This allows the sorting algorithm to vary independently from clients that use it.

- **How It Works:**

a) Sorting Strategy Interface:

- The MailSortingStrategy interface defines a single method: sort, which takes a list of Mail objects and returns them sorted according to the implementing strategy's logic. This interface is the contract that all sorting strategies must fulfill.

b) Concrete Strategy Implementations:

- Each concrete strategy class implements the sorting logic for one specific sorting criterion:
  - **SortByDateDesc**: Sorts emails from newest to oldest by comparing their date timestamps (default sorting).
  - **SortByDateAsc**: Sorts emails from oldest to newest, useful when users want to review their email history chronologically.

- ❖ **SortByPriority**: Sorts emails by their priority level (1=Urgent, 2=High, 3=Normal, 4=Low), ensuring important emails appear first.
- ❖ **SortBySender**: Sorts emails alphabetically by sender's email address or display name, helpful when looking for all emails from a specific person.
- ❖ **SortBySubject**: Sorts emails alphabetically by subject line, useful for grouping related conversation threads.

- Each strategy uses Java streams and Comparator to perform the sorting. They handle edge cases like null values using Comparator.nullsLast() or similar utilities.

c) Context Class (MailSorter):

- The MailSorter class acts as the context that uses a strategy. It holds a reference to a MailSortingStrategy and delegates the actual sorting work to that strategy. The strategy can be set at construction time or changed later using setStrategy(), allowing runtime flexibility.
- When the sort() method is called on MailSorter, it simply forwards the request to the current strategy's sort() method. This separation means the MailSorter doesn't need to know anything about how sorting actually works.

d) Integration with MailService:

- In the MailService.sortMails() method, when a request comes in with a sortBy parameter (like "priority" or "date_asc"), the service uses a switch statement to select the appropriate strategy class. It then creates a MailSorter with that strategy and calls sort() on the mail list.
- The sorted results are then paginated and converted to DTOs before being returned to the client.

- **Benefits:**

- Each sorting algorithm is isolated in its own class with clear responsibility.
- Adding new sorting options (like by attachment count or unread status) requires only creating a new strategy class.
- The sorting logic can be tested independently for each strategy.
- Users can dynamically switch between sorting methods without page reloads.
- The MailService doesn't contain complex conditional logic for different sorting types.

3. Chain of Responsibility Pattern:

- **Pattern Type:** Behavioral
- **Purpose:**

- The Chain of Responsibility pattern allows passing requests along a chain of handlers during the sign in/sign up process. Each handler decides either to process the request or to pass it to the next handler in the chain.

- **How It Works:**

a) Abstract Handler Structure:

- The AuthHandler is an abstract base class that provides the chain mechanism. It contains a reference to the next handler in the chain and a setNext() method that both sets the next handler and returns it, allowing fluent chaining syntax.

- The handle() method is marked final to prevent subclasses from breaking the chain mechanism. It calls the abstract validate() method (which subclasses implement), then automatically forwards the request to the next handler (if chain is broken).

b) Concrete Handlers:

1. ***Sign Up Handlers:***

❖ **Password Strength Checker:** The PasswordStrength handler implements the validate() method to check password requirements: Minimum 8 characters length, Contains at least one uppercase letter, Contains at least one lowercase letter, Contains at least one digit and Contains at least one special character.

❖ **Email Duplication Checker:** The EmailDuplication handler implements the validate() method to prevent duplicate accounts. It checks if an email already exists in the database during signup and throws an exception if the email is already registered.

❖ **Email Format Validator:** The EmailFormat handler implements the validate() method to check basic email validity. It verifies that the email contains an "@" symbol, throwing an exception if the format is invalid.

2. ***Sign In Handlers:***

❖ **Email Existence Checker:** The EmailExistence handler implements the validate() method to verify that a user account exists. It checks the database using the email from the request and throws an exception if no user is found with that email address.

❖ **Password Match Validator:** The PasswordMatch handler implements the validate() method to verify login credentials. It retrieves the user by email from the database and compares the stored password with the provided password, throwing an exception if they don't match.

- **Chain Setup and Usage:**

- During user registration, the authentication service sets up a chain of validators. First comes EmailValidator (which checks email format), then PasswordStrength.
- If any validator throws an exception, the chain stops immediately and the error is returned to the user. Only if all validators pass does the registration proceed.

- **Benefits**:

- Validation logic is decoupled from the authentication service
- Each validator has a single, clear responsibility
- Easy to add or remove validation rules without touching other code
- The order of validation can be changed by simply reordering the chain
- Validators can be reused across different authentication flows

4. Filter/Criteria Pattern:

- **Purpose:**

- The Filter Pattern (also known as Criteria Pattern) allows combining multiple filtering criteria dynamically to filter a collection of objects. It's used for both advanced mail searching and automatic filtering and redirecting rules.

- **How It Works:**

a) Criteria Interface Design:

- The MailCriteria interface defines the contract for all filter implementations. It has one primary method: meetCriteria(), which takes a list of Mail objects and returns a filtered list containing only mails that satisfy the criteria.
- The interface also provides a default matches() method that checks if a single mail meets the criteria. This is useful for automatic filtering where we need to test each incoming email against filter rules.

b) Concrete Criteria Implementations:

- Each **concrete** criteria class encapsulates the logic for filtering based on one attribute:
    - **SenderCriteria**: Checks if the sender's email address or display name contains the specified keyword. It handles case-insensitive matching by converting both the keyword and email to lowercase. The filtering uses Java streams to iterate through all emails and select only those where the sender email contains the keyword. It safely handles null values to prevent NullPointerException.
    - **SubjectCriteria**: Similar to SenderCriteria but examines the subject field instead.
    - **BodyCriteria**: Searches within the email body text. This criteria performs substring matching on the text content. Users might search for specific phrases or keywords mentioned in email conversations.
    - **ReceiverCriteria**: More complex than sender criteria because emails can have multiple receivers. It checks if any of the receiver email addresses or display names match the keyword. This uses Java streams with anyMatch() to check if at least one receiver matches. It's useful for finding all emails sent to a particular person or distribution list.

c) Composite Criteria: AndCriteria:

- The AndCriteria class implements the **Composite** pattern within the Filter pattern. It takes two MailCriteria objects and combines them with AND logic. When meetCriteria() is called, it first applies the first criteria to filter the list, then applies the second criteria to the already-filtered results.

d) Advanced Search Flow:

- When a user performs an advanced search, the MailService.advancedSearch() method receives a MailSearchRequestDto containing optional sender, receiver, subject, and body search terms.
- The service builds a **composite criteria** dynamically:
    1. It starts with criteria = null
    2. If sender is specified, it creates a SenderCriteria
    3. If receiver is specified, it creates a ReceiverCriteria. If criteria already exists, it wraps both in an AndCriteria
    4. It continues this pattern for subject and body

5. Finally, it retrieves all mails from the folder and applies the composite criteria
- This approach allows any combination of search terms. Users can search by just sender, or sender + subject, or all four fields together.

e) Automatic Filter Application:

- User-defined filters use the same criteria system. When a user creates a filter rule like "move emails from newsletter@company.com to Newsletter folder," this is stored as a MailFilter entity in the database with field="sender" and filterValue="newsletter@company.com".
- When a new email arrives, the applyFilters() method:
    1. Retrieves all filter rules for the recipient user
    2. For each filter, uses CriteriaFactory to create the appropriate criteria object
    3. Tests if the email matches using criteria.matches(mail)
    4. If it matches and the email isn't already in multiple folders, automatically moves it to the target folder
    5. Stops after the first match (First matched filter only applies)
- **Benefits:**
- **Composability**: Multiple criteria combine using AndCriteria
- **Reusability**: Same criteria classes for both searching and automatic filtering
- **Single Responsibility**: Each criteria class handles one type of filtering
- **Extensibility**: New filter types added without modifying existing code
- **Expressiveness**: Complex filter combinations built from simple pieces
- **Decoupling**: Filtering logic separated from service and controller code

5. Builder Pattern:

- **Pattern Type:** Creational
- **Purpose:** The purpose of the builder in general to build like a complex object rather than using the constructors to build the object and in our project using the builder annotation in spring boot ,so we can build complex objects using it and it supports the idea of reusability .
- **The examples in the project : MailDto, MailEntity, MailSnapshotEntity** inside the mail factory to build all these objects to make easier to set the attributes and as an example here : `Mail.builder()`

```
        .userId(userId)
        .senderEmail(dto.getSender())
        .receiverEmails(new ArrayList<>(dto.getReceivers()))
        .priority(dto.getPriority())
        .subject(dto.getSubject())
        .body(dto.getBody())
        .status(MailStatus.SENT)
        .isRead(true)
        .date(Timestamp.valueOf(LocalDateTime.now()))
        .build();
```

Using the annotation `@Builder` above all these classes using the library

`import lombok.*;` **and here is another example for the MailDto and MailSnapshotEntity**

```
MailSnapshot.builder()

        .userId(draft.getUserId())

        .attachments(draft.getAttachments() != null ? new
ArrayList<>(draft.getAttachments()) : new ArrayList<>())

        .mail(draft)

        .priority(draft.getPriority())

        .subject(draft.getSubject())

        .body(draft.getBody())

        .receiverEmails(new ArrayList<>(draft.getReceiverEmails()))

        .savedAt(Timestamp.valueOf(LocalDateTime.now()))

        .build();
```

```
MailDto.builder()

        .userId(mail.getUserId())

        .folderId(folderId)

        .receivers(mail.getReceiverEmails())

        .sender(mail.getSenderEmail())

        .senderDisplayName(mail.getSenderDisplayName())

        .subject(mail.getSubject())

        .body(mail.getBody())

        .priority(mail.getPriority())

        .date(mail.getDate().toLocalDateTime())

        .attachments(attachments)

        .build() ;
```

6. Snapshot (memento):

- **Pattern type: behavioural design pattern.**
- **Purpose:** It support the feature of restoring previous state for an object that is to be created within the process through :
- ▪ **Memento**: contains the state of an object that can be restored later.
- ▪ **Originator**: creates the Memento objects and stores its current state inside them.
- ▪ **Caretaker**: is responsible for keeping the Memento and restoring the object's state when needed.

How it works : In our project , the purpose of that feature in our project to have the states of a draft to be edited and in that case we can restore the draft through that pattern to any state we want to choose ,therefore, once the draft is created , it takes a snapshot for each edit and that's how it works in the code:

```
saveSnapshot(draft);
```

▪ Through this function we can save the state of each edit and it is considered to be the Originator and it is responsible for creating the mailSnapshots of that draft through another design pattern which is builder annotation ,and in our case the Memento is the mailSnapshot , and the Caretaker is the draftservice.

7. Singleton:

- **Pattern type:** Creational
- **Purpose:**
  - The Singleton pattern restricts the instantiation of a class to a single instance and provides a global point of access to that instance. This ensures that all objects across the application use the same shared resource, preventing the overhead and inconsistency of creating multiple objects for the same task.
- **In this Project**
  - the Service Layer (eg. MailService), Controller Layer (eg. MailController) and Repository Layer (eg. MailRepo) are implemented as Singletons.
- **Benefits:**
  - Resource Efficiency: We avoid the memory overhead of creating a new `MailService` or `MailController` object for every single HTTP request.
  - State Consistency: It ensures that all parts of the application interacting with the database go through the exact same service instance, maintaining transactional integrity.
  - Controller Reuse: When multiple users send requests simultaneously (e.g., calling `/api/mail/send`), the web server reuses the same existing Controller instance to handle all requests on different threads, rather than instantiating a new controller for every user action. This drastically improves performance and scalability.

8. Facade:

- **Pattern Type:** Structural
- **Purpose:**

- The Facade pattern provides a simplified, unified interface to a set of complex subsystems, making it easier for clients to interact with them without needing to know their internal details or dependencies. It reduces coupling between the client and subsystems and improves code maintainability.

- **In this project:**

  Each service acts as a facade for the controller providing a simple unified interface to perform complex operations.

  The service hides the internal workflow involving multiple subsystems (like `Repositories, Folder management, Attachments and Filters`), so the controller can perform tasks without dealing with the underlying complexity.

Example 1: MailService as a Facade

What it hides:

- Creating separate copies of a mail for sender and each receiver
- Linking attachments to new mails
- Applying filters/ multiple filtering
- Adding mails to appropriate folders
- Finding mails in a folder
- Moving them to Trash folder
- Setting **deletedAt** timestamp
- Paginating results

While The controller only calls:

```
createNewMail()/ deleteMailById(List<String> ids, String folderId)/
advancedSearch() / searchEmails()
```

Example 2: AttachmentService as a Facade

What it hides:

- Storing attachments physically in `Attachment_dir` with unique file names
- Creating attachment metadata (`Attachment` entity) in DB
- Handling multiple attachments per mail
- Deleting attachments both physically and in DB (`Files.deleteIfExists`)
- Duplicating attachments for new mail (`duplicateAttachmentsForNewMail`)
- Loading attachments as resources for download (`UrlResource`)

- Returning attachment metadata to controllers

While the controller only calls:

`createNewAttachment(file, mailId)/ DeleteAttachment(attachmentId)/ loadFileAsResource(attachmentId / getAttachmentMeta(attachmentId)`

Example 3: FolderService as a Facade

What it hides:

- Creating new folders and linking them to users (`createFolder`)
- Initializing default folders for a user (Inbox, Drafts, Sent, Trash) (`initialize`)
- Deleting a folder (`deleteFolder`) and:
- Moving mails inside it to Inbox or Sent depending on whether the user is sender or receiver
- Updating previous folder information of mails
- Maintaining folder-mail links and saving changes in DB
- Adding and deleting mails inside folders (`addMail`, `deleteMail`)
- Renaming folders (`renameFolder`)
- Retrieving all folders for a user or only custom folders (`getFolders`, `getCustomFolders`)
- Hiding all `FolderRepo` and `UserRepo` interactions and DB persistence logic

While the controller only calls indirectly via MailService:

`createFolder(FolderDto folderDto)` / `deleteFolder(folderId, userId) / renameFolder(folderId, newName) / getFolders(userId)` / `getCustomFolders(userId)`

9. Mapper:

- **Pattern Type:** Structural
- **Purpose:**
  - The Mapper Design Pattern is used to separate the internal data representation of an object from how it is presented or transferred to other layers.

    Here it maps between two objects:

    - **Entity objects** (e.g., `User`, `Mail`) — your database models
    - **DTOs (Data Transfer Objects)** — objects used to send/receive data from controllers or APIs
- **In this project:**
    - **AttachmentMapper:**

      Converts an `Attachment` entity to a `DTO` for the controller, including `mailId` if associated, hiding the entity's internal details.

    - **User Mapper:**

      Converts `User` entities ↔ DTOs, including nested contacts and folders, so controllers/services don't handle entity relationships directly.

    - **Contact Mapper:**

      Converts between `Contact` entity and DTO, handling owner mapping and collections like email addresses, so controllers/services don't deal with entity details.

    - **Mail Filter Mapper:**

      Converts `MailFilter` entities ↔ DTOs, fetches and maps `User`, updates entities from DTOs, so controllers/services only work with simplified DTOs.

---

5. Design Decisions:

1. **Folder Organization:**
    a. Default folders: Inbox, Sent, Trash, Drafts
    b. Default folders can not be deleted to preserve core mail functionality.
    c. User-created folders stored separately.
    d. Custom folders allow users to organize emails based on personal preferences.
    e. Each folder is linked to a specific user to ensure data privacy.

f. Deleting a custom folder doesn't delete its emails, moving them to sent or inbox folders.

2. **Attachment Storage:**
   a. Stored in `Attachment_dir` in file system.
   b. Metadata stored in DB (`Attachment` entity).
   c. Supports multiple attachments per mail.
   d. Files are stored in `mail-specific-subfolders` for organized storage.
   e. Unique filenames are generated using `UUIDs` to prevent collisions.
   f. Deleting an attachment removes it `from both the database and the file system`, ensuring no orphan files.
   g. Attachments are linked to a **specific mail** via a `ManyToOne` relationship.
   h. Attachments can be **downloaded as resources** with correct MIME type and filename.
3. **Pagination:**
   a. Emails sent to frontend in pages of 10 emails.
   b. Reduces frontend load, improves performance.
   c. Handled in `MailService.paginateMails()` by slicing the sorted or filtered list.
   d. Supports `Dynamic page navigation` (pages are not preloaded).
4. **Searching & Filtering:**
   a. Backend uses Criteria pattern (`SenderCriteria`, `ReceiverCriteria`) for flexible filtering.
   b. **Advanced search** supports multiple fields: `Sender, Receiver, Subject, Body.`
   c. Multiple criteria can be combined using `AndCriteria.`
   d. Filters are applied per folder and per user.
   e. Stored user-defined filters (`MailFilter`) can automatically **move mails to target folders** based on matching criteria.
   f. Supports keyword search across `Sender, Receiver, Subject, Body.`

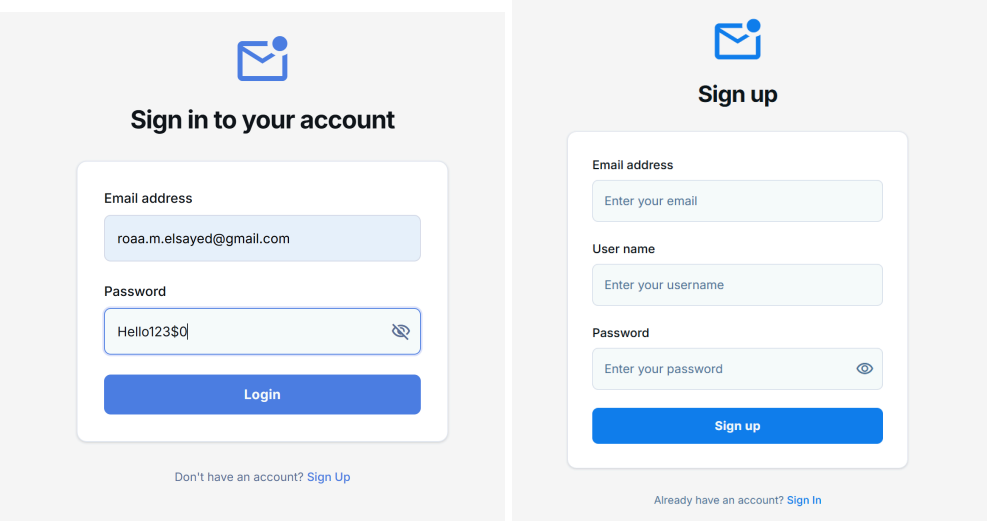
5. **Mail Organization:**
   a. Composing a mail creates `independent copies for:`
      `Sender: (saved in sent folder, marked as read)`
      `Each Receiver: (saved in inbox folder, marked as unread)`
   b. Receiver copies maintain **attachments and display names** of sender.
   c. Each copy is **independent**, so actions (delete, read, move) **do not affect other users' copies**.

d. Supports **attachment duplication** for each recipient while keeping the **physical file shared**.
e. Mails can be moved between folders, deleted, or restored without affecting other copies.

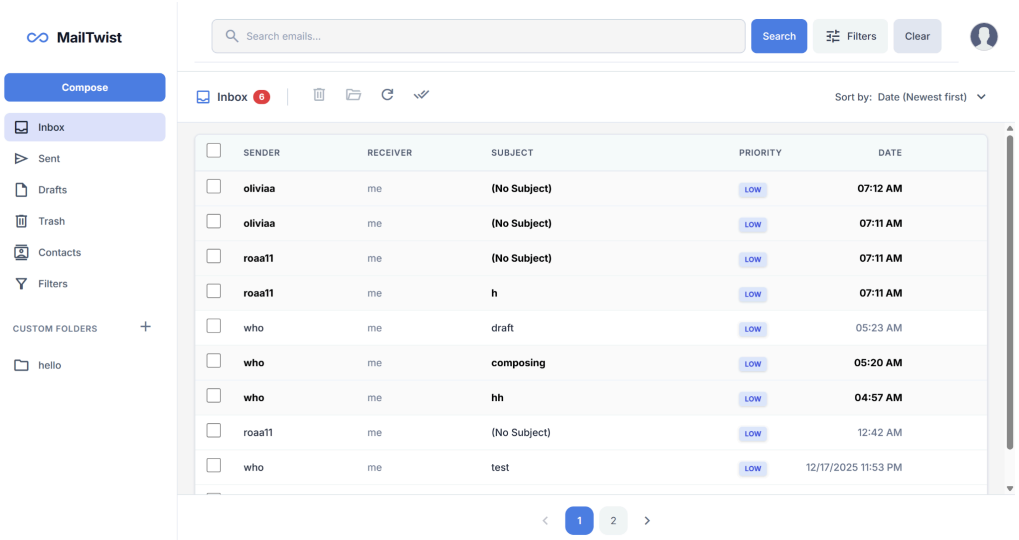
---

5. User Guide / Snapshots:

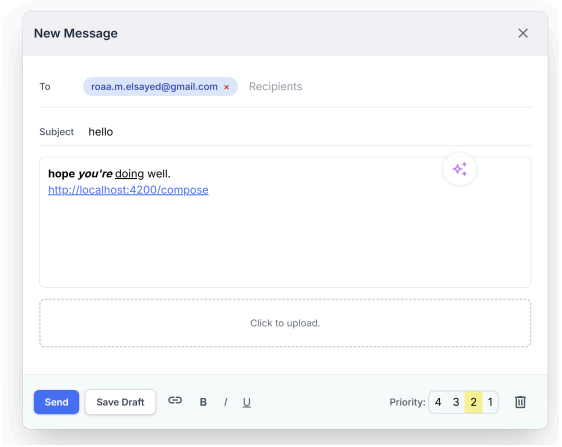
1. **Sign in Page/Sign up Page: Enter email and password or you can register to have another account.(Once you sign in, you will face the inbox page as in (2))**

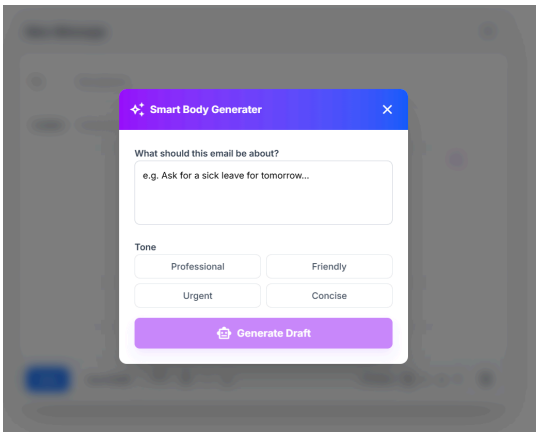


2. **Inbox: Displays emails with pagination and sorting options.You can navigate.**

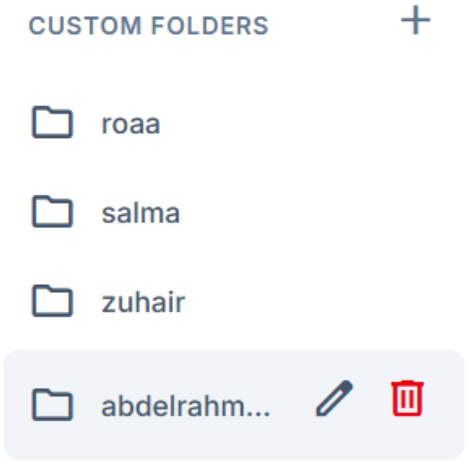


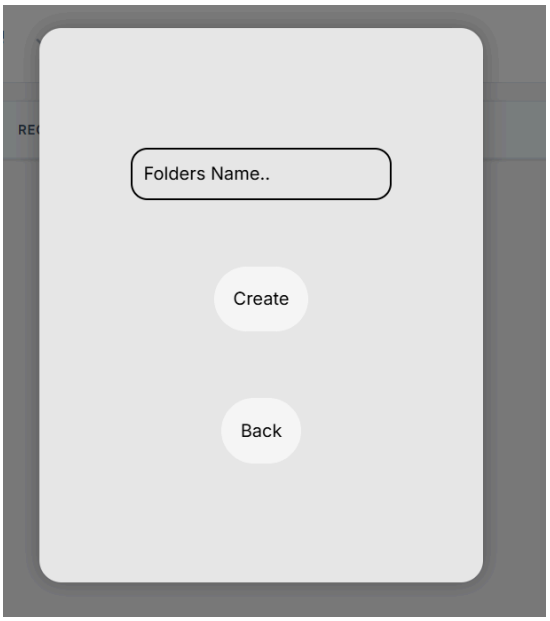
3. **Compose: Create new email, attach files. You can choose the priority for the mail to be sent , save it as a draft to edit it whenever you want , enter the receivers that you want to send to.You can use the AI to help you, too !**



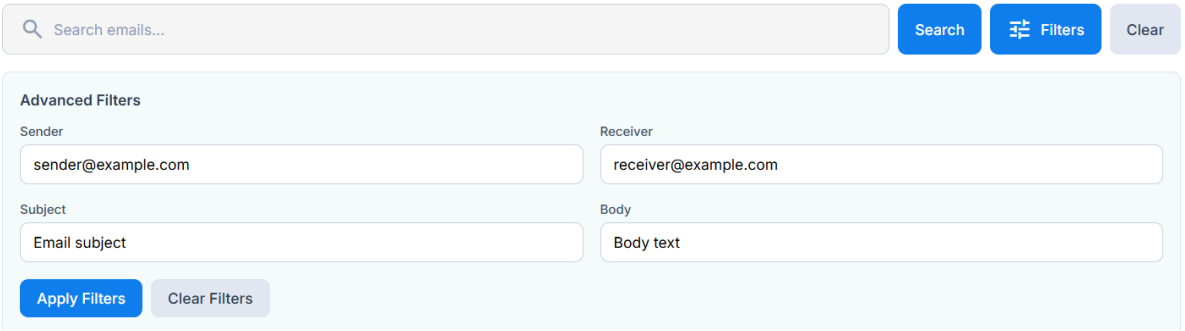


4. **Folders: Create, rename, delete folders; move emails. You can add custom folders which suit you from the icon in each main folder ,and you can delete them permanently. They have the same options that you have in your inbox, sent…**

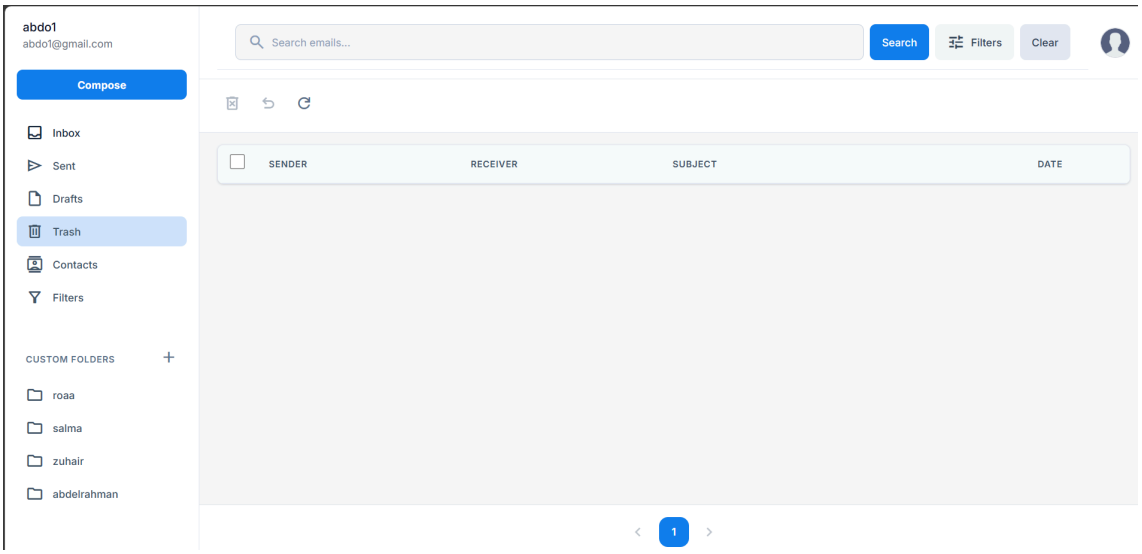




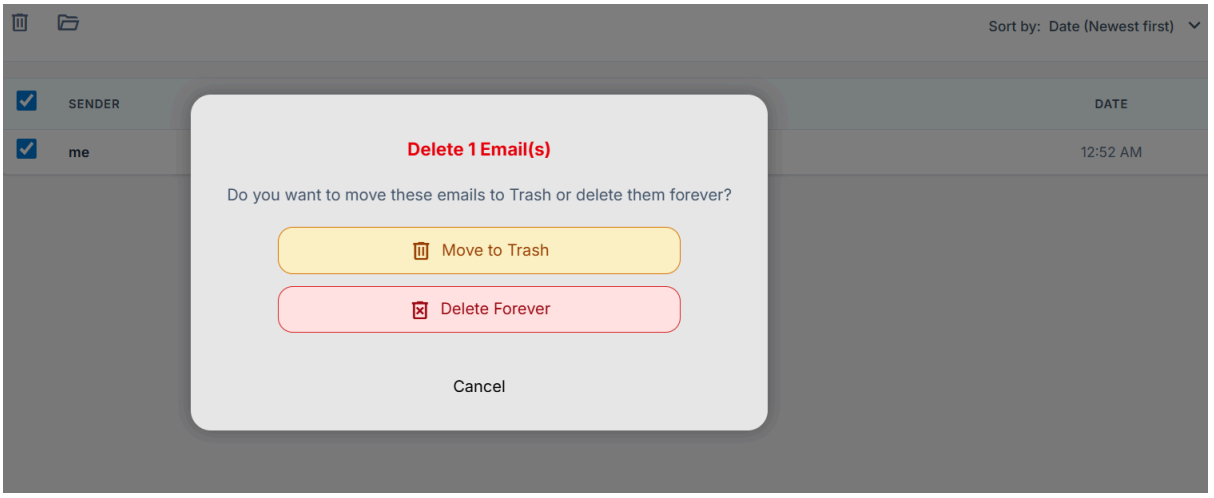
5. **Search Bar: Search by sender, receiver, or subject. You can also use the filter option.**



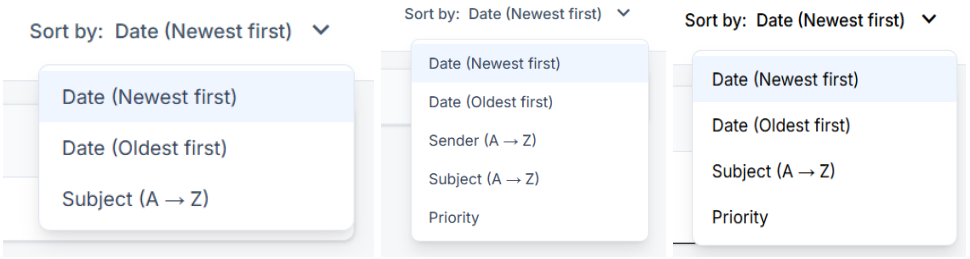
6. **Trash: Auto-deletes emails after 30 days. You can undo the mails that you have moved to the trash or delete them forever.**



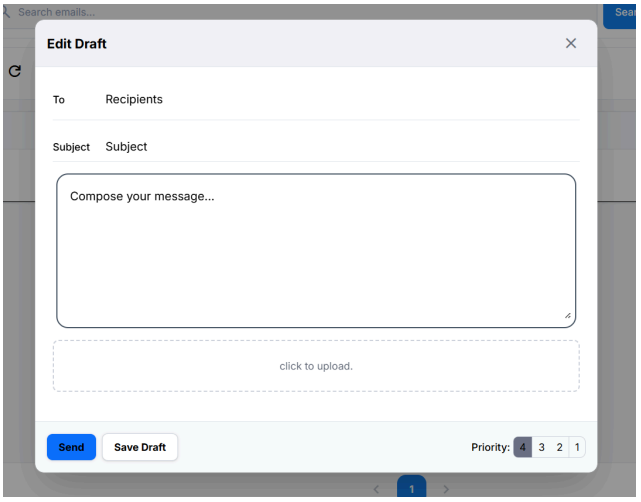
7. **Deletion options from any folder (except the drafts)**



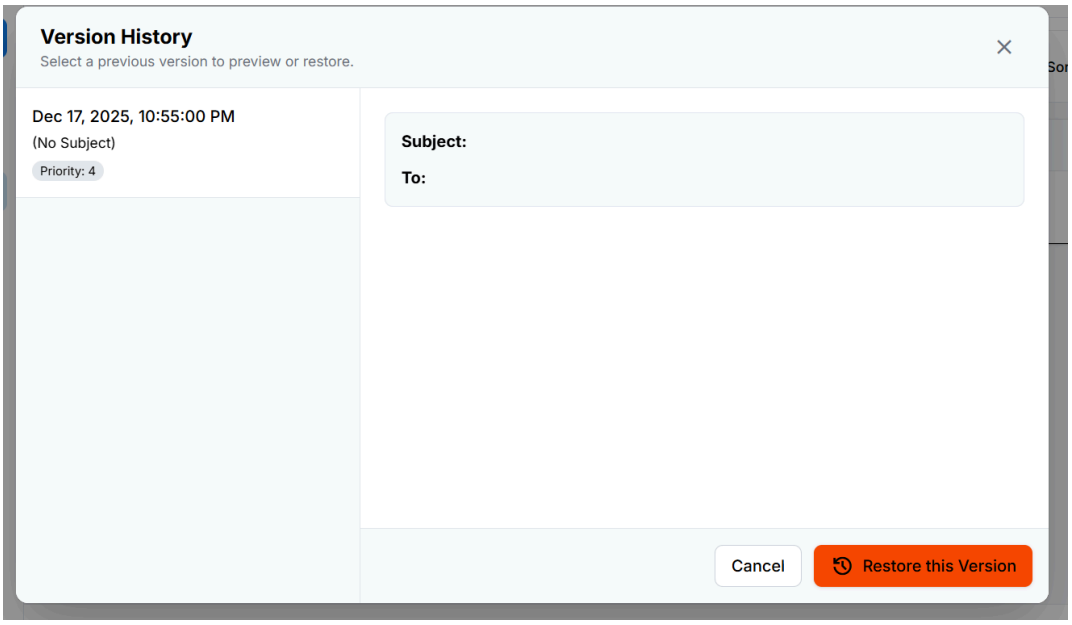
8. **Sort options.Differs from folder to another**



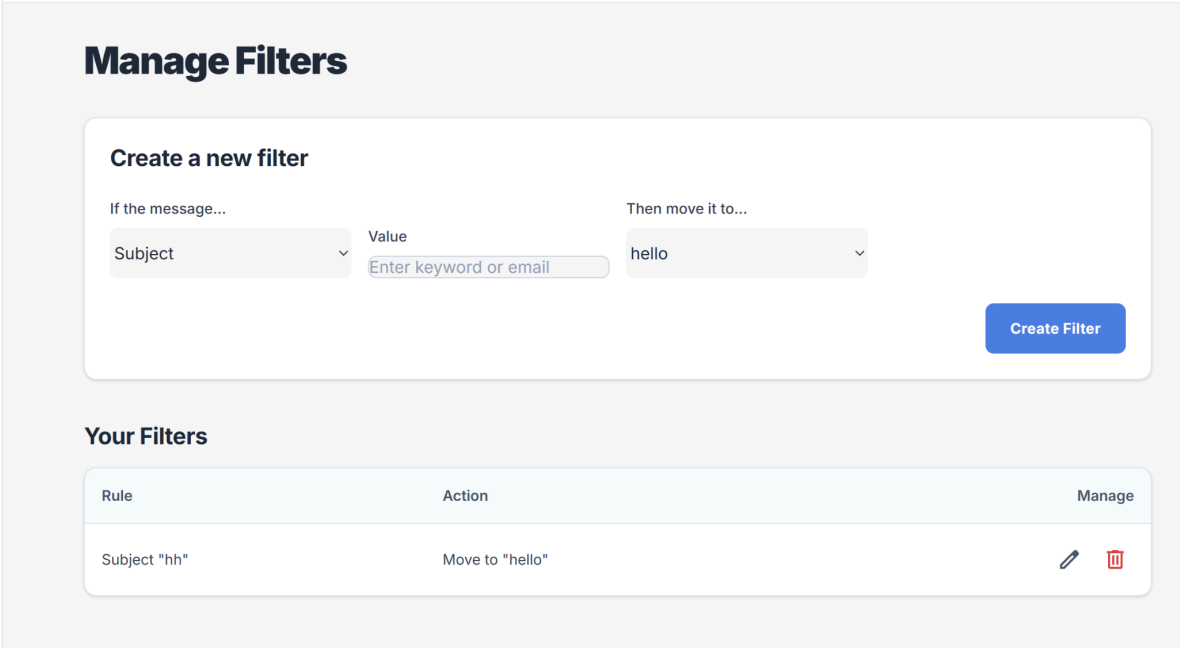
9. **Drafts: Save and edit unfinished emails.**



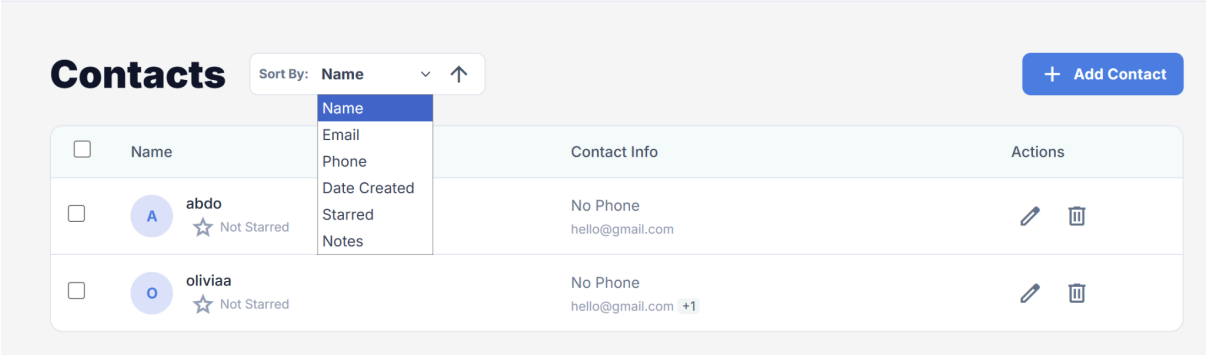
10. **Snapshots , where you can see the history of the drafts and you can restore any version you see**

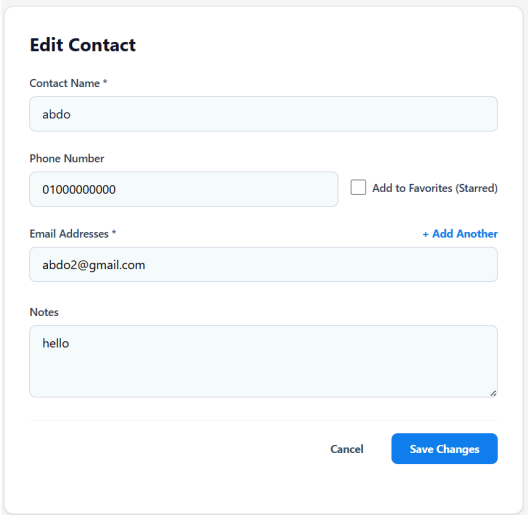


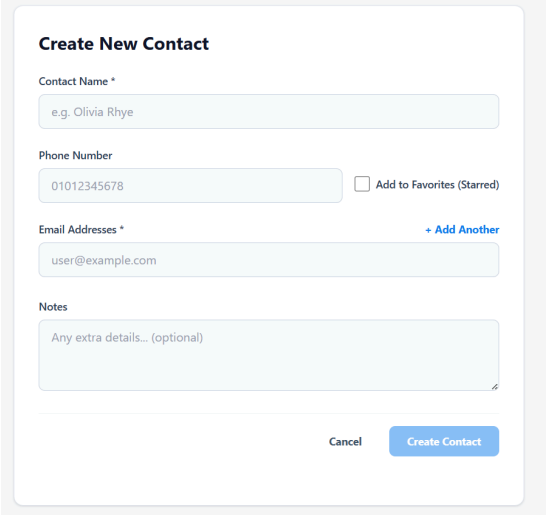
11. Here you can manage you special filter to direct them to a specific custom folder



12. You can have your own contacts, too. You can save the contacts, make them starred, write notes, and even search for a contact if needed or ,sort them. You can edit the contact.







13. **You can easily from here to sign out or add another account**