# Complete Setup and Deployment Guide

## Email Management System - Backend

---

# 1. Prerequisites

### Required Software

**Version Required:** JDK 17 or higher

**Version Required:** Maven 3.6.0 or higher

**Step 1: Open Project**

1. Launch IntelliJ IDEA
2. Click **"Open"**
3. Select your project folder
4. Wait for Maven to import dependencies (bottom right corner)

**Step 2: Run the Application**

1. Navigate to src/main/java/com/example/backend/BackendApplication.java
2. Right-click on the file
3. Select **"Run 'BackendApplication'"**

**OR**

Click the green play button  next to the main method

**Step 3: Check Console**

Look for the message:

Tomcat started on port(s): 8080 (http)

**Success!** Your application is running.

**Step 4:run the front**

Run ng serve in the terminal

# Design Patterns Implementation Documentation

## Table of Contents



---

## 1. Factory Pattern

### Pattern Description

The Factory Pattern is a creational design pattern that provides an interface for creating objects without specifying their exact classes. It encapsulates object creation logic and promotes loose coupling.

### Implementation Details

#### 1.1 ContactFactory

**Location:** `com.example.backend.Factory.ContactFactory`

**Purpose:** Centralized creation and configuration of Contact objects with automatic initialization of required fields.

**Key Features:**

- Generates unique UUIDs for each contact
- Automatically generates initials from contact names
- Assigns random avatar colors from a predefined palette
- Ensures at least one primary email and phone number when lists are not empty
- Handles null safety for email and phone lists

**Design Decisions:**

1. **Automatic Primary Selection:** When creating or updating contacts, if email/phone lists are provided without a primary designation, the factory automatically marks the first item as primary. This ensures data consistency and prevents UI errors.
2. **Initials Generation:** Implemented an algorithm that extracts the first letter of each word in a name and converts it to uppercase, providing a readable avatar fallback.
3. **Color Palette:** Defined a fixed set of six complementary colors (#3b82f6, #8b5cf6, #ec4899, #10b981, #f59e0b, #ef4444) to ensure visual consistency across the application.
4. **Separate Create and Update Methods:** The factory provides both `createContact()` and `updateContact()` methods. The create method generates new IDs and colors, while update preserves the existing color to maintain visual identity.

**Code Example:**

```
@Component
public class ContactFactory {
    private static final String[] avatarColors = {
        "#3b82f6", "#8b5cf6", "#ec4899",
        "#10b981", "#f59e0b", "#ef4444"
    };

    public Contact createContact(contactRequestDTO dto) {
        Contact contact = new Contact();
        contact.setId(UUID.randomUUID().toString());
        contact.setName(dto.getName());
        contact.setInitials(generateInitials(dto.getName()));
        contact.setColour(generateRandomColor());

        // Ensure primary designation
        ensurePrimaryEmail(contact.getEmail());
        ensurePrimaryPhone(contact.getPhone());

        return contact;
    }
}
```

**Usage in Service Layer:**

```
@Service
public class ContactService {
    @Autowired
    private ContactFactory contactFactory;

    public contactResponseDTO addContact(String user, contactRequestDTO dto) {
        Contact newContact = contactFactory.createContact(dto);
        contacts.add(newContact);
        return mapToDTO(newContact);
    }
}
```

### 1.2 FolderFactory

**Location:** `com.example.backend.Factory.FolderFactory`

**Purpose:** Manages creation and updates of custom folder entities.

**Key Features:**

- Generates unique folder IDs
- Initializes email count to zero for new folders
- Handles optional description fields with default empty strings
- Maintains folder metadata consistency

**Design Decisions:**

1. **Default Values:** Empty string for missing descriptions rather than null to prevent null pointer exceptions in the frontend.
2. **Immutable ID:** Once created, folder IDs are never changed, even during updates, ensuring referential integrity.

3. **Email Count Initialization:** New folders start with zero emails, which is then managed by the FolderService.

**Code Example:**

```
@Component
public class FolderFactory {
    public Folder createFolder(FolderRequestDTO dto) {
        Folder folder = new Folder();
        folder.setId(UUID.randomUUID().toString());
        folder.setName(dto.getName());
        folder.setDescription(dto.getDescription() != null
            ? dto.getDescription() : "");
        folder.setColor(dto.getColor());
        folder.setEmailCount(0);
        return folder;
    }
}
```

### 1.3 mailFactory

**Location:** `com.example.backend.Factory.mailFactory`

**Purpose:** Creates new mail objects with proper initialization and preview generation.

**Key Features:**

- Generates sequential integer IDs (persisted to file)
- Auto-generates email preview (first 100 characters of body)
- Sets default values for starred status
- Detects attachments automatically
- Records timestamp of creation

**Design Decisions:**

1. **ID Counter Persistence:** Uses file-based storage (`mail_counter.txt`) to maintain ID sequence across application restarts. Static initialization block loads the counter when the class is first loaded.
2. **Preview Generation:** Automatically creates a preview by taking the first 100 characters of the email body, providing a quick summary for list views.
3. **Attachment Detection:** Sets `hasAttachment` flag based on whether the attachments list is empty, allowing for quick filtering.
4. **Static Factory Method:** Uses a static `createNewMail()` method, making it accessible without dependency injection (though this could be improved).

**Code Example:**

```
public class mailFactory {
    private static int idCounter = 0;

    static {
        idCounter = loadCounterFromFile();
    }

    public static mail createNewMail(mailContentDTO mailContent) {
        mail mail = new mail();
        mail.setId(++idCounter);
```

```
        mail.setTimestamp(LocalDateTime.now());
        mail.setStarred(false);
        mail.setPreview(mailContent.getBody()
            .substring(0, Math.min(100, mailContent.getBody().length())));

        saveCounterToFile();
        return mail;
    }
}
```

---

# 2. Strategy Pattern

## Pattern Description

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it.

## Implementation Details

### 2.1 Contact Sorting Strategy

**Location:** `com.example.backend.StrategyPattern`

**Components:**

- `contactSortStrategy` (Interface)
- `sortByName` (Concrete Strategy)
- `sortByEmail` (Concrete Strategy)

**Purpose:** Provides flexible sorting options for contact lists.

**Design Decisions:**

1. **Interface-Based Design:** The `contactSortStrategy` interface defines a single `sort()` method, allowing any sorting algorithm to be plugged in.
2. **In-Place Sorting:** Strategies modify the list directly rather than returning a new list, improving memory efficiency.
3. **Case-Insensitive Comparison:** Name sorting uses `compareToIgnoreCase()` to provide user-friendly alphabetical ordering.
4. **Primary Email Selection:** Email sorting uses the first email in each contact's list, assuming it's the primary one.

**Code Example:**

```
public interface contactSortStrategy {
    void sort(List<Contact> contacts);
}

public class sortByName implements contactSortStrategy {
    @Override
    public void sort(List<Contact> contacts) {
        contacts.sort((a, b) ->
            a.getName().compareToIgnoreCase(b.getName()));
```

```
    }
}

public class sortByEmail implements contactSortStrategy {
    @Override
    public void sort(List<Contact> contacts) {
        contacts.sort((a, b) -> {
            String emailA = a.getEmail().get(0).getAddress().toLowerCase();
            String emailB = b.getEmail().get(0).getAddress().toLowerCase();
            return emailA.compareTo(emailB);
        });
    }
}
```

**Usage in ContactService:**

```
public class ContactService {
    private contactSortStrategy getStrategy(String sortBy) {
        if ("email".equalsIgnoreCase(sortBy)) {
            return new sortByEmail();
        }
        return new sortByName();
    }

    public PaginatedContactResponse getContacts(String user, String sortBy) {
        List<Contact> contacts = contactRepo.findAll(user);
        contactSortStrategy strategy = getStrategy(sortBy);
        strategy.sort(contacts);
        // ... pagination logic
    }
}
```

### 2.2 Email Sorting Strategy

**Location:** `com.example.backend.StrategyPattern`

**Components:**

- `EmailSortStrategy` (Interface)
- `SortByDateStrategy` (Concrete Strategy)
- `SortBySenderStrategy` (Concrete Strategy)
- `SortBySubjectStrategy` (Concrete Strategy)
- `SortByPriorityStrategy` (Concrete Strategy)
- `EmailSortContext` (Context/Registry)

**Purpose:** Provides comprehensive sorting options for email lists with both ascending and descending orders.

**Design Decisions:**

1. **Strategy Registry Pattern:** The `EmailSortContext` maintains a HashMap of all available strategies, allowing runtime strategy selection by name.
2. **Naming Convention:** Strategies use descriptive names like "date-desc", "sender-asc", enabling easy configuration from frontend.
3. **Bidirectional Sorting:** Each strategy supports both ascending and descending order through a boolean constructor parameter, reducing code duplication.

4. **Null Safety:** All strategies handle null values using `Comparator.nullsLast()`, ensuring stable sorting even with incomplete data.
5. **Default Fallback:** When an unknown strategy is requested, the context defaults to "date-desc", preventing runtime errors.

**Code Example:**

```
public interface EmailSortStrategy {
    void sort(List<mail> emails);
    String getStrategyName();
}

public class SortByDateStrategy implements EmailSortStrategy {
    private final boolean ascending;

    @Override
    public void sort(List<mail> emails) {
        Comparator<mail> comparator = Comparator.comparing(
            mail::getTimestamp,
            Comparator.nullsLast(Comparator.naturalOrder())
        );

        if (!ascending) {
            comparator = comparator.reversed();
        }

        Collections.sort(emails, comparator);
    }

    @Override
    public String getStrategyName() {
        return ascending ? "date-asc" : "date-desc";
    }
}

@Component
public class EmailSortContext {
    private final Map<String, EmailSortStrategy> strategies;

    public EmailSortContext() {
        this.strategies = new HashMap<>();
        registerDefaultStrategies();
    }

    private void registerDefaultStrategies() {
        registerStrategy(new SortByDateStrategy(true));
        registerStrategy(new SortByDateStrategy(false));
        registerStrategy(new SortBySenderStrategy(true));
        // ... other strategies
    }

    public List<mail> sortEmails(List<mail> emails, String strategyName) {
        EmailSortStrategy strategy = strategies.get(strategyName);

        if (strategy == null) {
            strategy = strategies.get("date-desc");
        }

        strategy.sort(emails);
        return emails;
    }
```

```
}
```

**Usage in mailService:**

```
@Service
public class mailService {
    @Autowired
    private EmailSortContext emailSortContext;

    public List<mail> getInboxEmails(String sort, FilterCriteriaDTO filters) {
        List<mail> emails = mailRepo.getInboxEmails();

        // Apply sorting if provided
        if (sort != null && !sort.isEmpty()) {
            emails = emailSortContext.sortEmails(emails, sort);
        }

        return emails;
    }
}
```

---

# 3. Chain of Responsibility Pattern

## Pattern Description

The Chain of Responsibility Pattern is a behavioral design pattern that passes requests along a chain of handlers. Each handler decides either to process the request or pass it to the next handler in the chain.

## Implementation Details

**Location:** `com.example.backend.FilterPattern`

**Components:**

- `EmailFilter` (Handler Interface)
- `AbstractEmailFilter` (Abstract Handler)
- Concrete Filters: `SearchFilter`, `DateRangeFilter`, `SenderFilter`, `PriorityFilter`, `AttachmentFilter`, `StarredFilter`, `SubjectFilter`, `BodyFilter`
- `EmailFilterService` (Chain Builder)

**Purpose:** Allows dynamic composition of email filtering criteria where multiple filters can be applied sequentially.

**Design Decisions:**

1. **Abstract Base Class:** `AbstractEmailFilter` provides common functionality for setting the next handler and passing filtered results along the chain, reducing code duplication.
2. **Null-Safe Processing:** Each filter checks if its criteria is null or empty before applying logic. If criteria is absent, it simply passes the unmodified list to the next handler.
3. **Immutable Chain Links:** Each filter maintains a reference to the next filter via `setNext()`, creating a linked chain structure.
4. **Stream-Based Filtering:** All filters use Java Streams for clean, functional filtering operations.

5. **Case-Insensitive Search:** Text-based filters (Search, Sender, Subject, Body) convert both search terms and email fields to lowercase for user-friendly matching.
6. **Flexible Chain Building:** The `EmailFilterService` builds the chain dynamically based on the provided `FilterCriteriaDTO`, allowing any combination of filters.

**Code Example:**

```
// Handler Interface
public interface EmailFilter {
    List<mail> apply(List<mail> emails);
    void setNext(EmailFilter next);
}

// Abstract Handler
public abstract class AbstractEmailFilter implements EmailFilter {
    protected EmailFilter next;

    @Override
    public void setNext(EmailFilter next) {
        this.next = next;
    }

    protected List<mail> passToNext(List<mail> emails) {
        if (next != null) {
            return next.apply(emails);
        }
        return emails;
    }
}

// Concrete Handler Example
public class SearchFilter extends AbstractEmailFilter {
    private final String searchTerm;

    @Override
    public List<mail> apply(List<mail> emails) {
        if (searchTerm == null || searchTerm.trim().isEmpty()) {
            return passToNext(emails);
        }

        String term = searchTerm.toLowerCase().trim();

        List<mail> filtered = emails.stream()
            .filter(email ->
                (email.getSubject() != null &&
                 email.getSubject().toLowerCase().contains(term)) ||
                (email.getBody() != null &&
                 email.getBody().toLowerCase().contains(term)) ||
                (email.getFrom() != null &&
                 email.getFrom().toLowerCase().contains(term))
            )
            .collect(Collectors.toList());

        return passToNext(filtered);
    }
}

// Other Concrete Handlers
public class DateRangeFilter extends AbstractEmailFilter {
    private final LocalDateTime dateFrom;
    private final LocalDateTime dateTo;
```

```
    @Override
    public List<mail> apply(List<mail> emails) {
        if (dateFrom == null && dateTo == null) {
            return passToNext(emails);
        }

        List<mail> filtered = emails.stream()
            .filter(email -> {
                LocalDateTime emailDate = email.getTimestamp();

                if (emailDate == null) return false;

                if (dateFrom != null && emailDate.isBefore(dateFrom)) {
                    return false;
                }

                if (dateTo != null) {
                    LocalDateTime endOfDay = dateTo
                        .withHour(23)
                        .withMinute(59)
                        .withSecond(59);
                    if (emailDate.isAfter(endOfDay)) {
                        return false;
                    }
                }

                return true;
            })
            .collect(Collectors.toList());

        return passToNext(filtered);
    }
}

public class PriorityFilter extends AbstractEmailFilter {
    private final List<Integer> priorities;

    @Override
    public List<mail> apply(List<mail> emails) {
        if (priorities == null || priorities.isEmpty()) {
            return passToNext(emails);
        }

        List<mail> filtered = emails.stream()
            .filter(email -> priorities.contains(email.getPriority()))
            .collect(Collectors.toList());

        return passToNext(filtered);
    }
}

public class AttachmentFilter extends AbstractEmailFilter {
    private final Boolean hasAttachment;

    @Override
    public List<mail> apply(List<mail> emails) {
        if (hasAttachment == null) {
            return passToNext(emails);
        }

        List<mail> filtered = emails.stream()
            .filter(email -> email.isHasAttachment() == hasAttachment)
```

```
            .collect(Collectors.toList());

        return passToNext(filtered);
    }
}

public class StarredFilter extends AbstractEmailFilter {
    private final Boolean isStarred;

    @Override
    public List<mail> apply(List<mail> emails) {
        if (isStarred == null) {
            return passToNext(emails);
        }

        List<mail> filtered = emails.stream()
            .filter(email -> email.isStarred() == isStarred)
            .collect(Collectors.toList());

        return passToNext(filtered);
    }
}
```

**Chain Builder (EmailFilterService):**

```
@Service
public class EmailFilterService {

    public List<mail> applyFilters(List<mail> emails, FilterCriteriaDTO criteria) {
        if (criteria == null || emails == null || emails.isEmpty()) {
            return emails;
        }

        EmailFilter chain = buildFilterChain(criteria);
        return chain.apply(emails);
    }

    private EmailFilter buildFilterChain(FilterCriteriaDTO criteria) {
        // Create all filters with criteria
        SearchFilter searchFilter = new SearchFilter(criteria.getSearchTerm());
        DateRangeFilter dateFilter = new DateRangeFilter(
            criteria.getDateFrom(),
            criteria.getDateTo()
        );
        SenderFilter senderFilter = new SenderFilter(criteria.getSender());
        PriorityFilter priorityFilter = new PriorityFilter(criteria.getPriority());
        AttachmentFilter attachmentFilter = new AttachmentFilter(
            criteria.getHasAttachment()
        );
        StarredFilter starredFilter = new StarredFilter(criteria.getIsStarred());
        SubjectFilter subjectFilter = new SubjectFilter(
            criteria.getSubjectContains()
        );
        BodyFilter bodyFilter = new BodyFilter(criteria.getBodyContains());

        // Build the chain
        searchFilter.setNext(dateFilter);
        dateFilter.setNext(senderFilter);
        senderFilter.setNext(priorityFilter);
        priorityFilter.setNext(attachmentFilter);
        attachmentFilter.setNext(starredFilter);
        starredFilter.setNext(subjectFilter);
```

```
        subjectFilter.setNext(bodyFilter);

        return searchFilter;
    }

    public boolean hasActiveFilters(FilterCriteriaDTO criteria) {
        if (criteria == null) return false;

        return (criteria.getSearchTerm() != null &&
                !criteria.getSearchTerm().trim().isEmpty()) ||
               criteria.getDateFrom() != null ||
               criteria.getDateTo() != null ||
               (criteria.getSender() != null &&
                !criteria.getSender().trim().isEmpty()) ||
               (criteria.getPriority() != null &&
                !criteria.getPriority().isEmpty()) ||
               criteria.getHasAttachment() != null ||
               criteria.getIsStarred() != null ||
               (criteria.getSubjectContains() != null &&
                !criteria.getSubjectContains().trim().isEmpty()) ||
               (criteria.getBodyContains() != null &&
                !criteria.getBodyContains().trim().isEmpty());
    }
}
```

**Usage in mailService:**

```
@Service
public class mailService {
    @Autowired
    private EmailFilterService emailFilterService;

    public List<mail> getInboxEmails(String sort, FilterCriteriaDTO filters) {
        List<mail> emails = mailRepo.getInboxEmails();

        // Apply filters if provided
        if (filters != null && emailFilterService.hasActiveFilters(filters)) {
            emails = emailFilterService.applyFilters(emails, filters);
        }

        // Apply sorting
        if (sort != null && !sort.isEmpty()) {
            emails = emailSortContext.sortEmails(emails, sort);
        }

        return emails;
    }
}
```

**Key Advantages:**

1. **Extensibility:** New filters can be added without modifying existing code
2. **Flexibility:** Filters can be arranged in any order
3. **Separation of Concerns:** Each filter handles one specific criterion
4. **Maintainability:** Each filter is a small, testable unit
5. **Dynamic Composition:** Chain is built at runtime based on user criteria

---

# 4. Command Pattern

## Pattern Description

The Command Pattern is a behavioral design pattern that encapsulates a request as an object, allowing parameterization of clients with different requests, queuing of requests, and support for undoable operations.

## Implementation Details

**Location:** `com.example.backend.service.ProfileCommandManager`

**Purpose:** Implements undo/redo functionality for user profile changes, maintaining a history of modifications.

**Components:**

- `ProfileCommandManager` (Invoker and Command Manager)
- `ProfileChange` (Command Object)

**Design Decisions:**

1. **Per-User History:** Each user has separate undo/redo stacks stored in ConcurrentHashMaps, ensuring thread safety and isolation between users.
2. **Field-Level Granularity:** Each command represents a single field change (jobTitle, phone, bio, profilePhoto), allowing fine-grained undo/redo.
3. **Stack-Based History:** Uses Stack data structures for LIFO (Last-In-First-Out) access, natural for undo/redo operations.
4. **Thread Safety:** Uses `synchronized` blocks and ConcurrentHashMap to handle concurrent access from multiple user sessions.
5. **History Cleanup:** Implements automatic cleanup of expired histories after 30 minutes of inactivity to prevent memory leaks.
6. **Size Limiting:** Caps history at 50 commands per user to prevent unbounded memory growth.
7. **Failure Recovery:** On undo/redo failure, the command is pushed back to its original stack, maintaining consistency.

**Code Example:**

```
@Service
public class ProfileCommandManager {
    private static final int MAX_HISTORY_SIZE = 50;
    private static final long HISTORY_TTL_MS = 30 * 60 * 1000; // 30 minutes

    private final Map<String, Stack<ProfileChange>> undoStacks =
        new ConcurrentHashMap<>();
    private final Map<String, Stack<ProfileChange>> redoStacks =
        new ConcurrentHashMap<>();
    private final Map<String, Long> lastAccessTime =
        new ConcurrentHashMap<>();

    // Command Object
    public static class ProfileChange {
        public String email;
        public String fieldName;
        public String oldValue;
        public String newValue;
```

```
    public ProfileChange(String email, String fieldName,
                         String oldValue, String newValue) {
        this.email = email;
        this.fieldName = fieldName;
        this.oldValue = oldValue;
        this.newValue = newValue;
    }
}

// Record a change (creates command)
public void recordChange(String email, String fieldName,
                         String oldValue, String newValue) {
    synchronized (this) {
        Stack<ProfileChange> undoStack = undoStacks
            .computeIfAbsent(email, k -> new Stack<>());
        Stack<ProfileChange> redoStack = redoStacks
            .computeIfAbsent(email, k -> new Stack<>());

        // Clear redo stack when new change is made
        redoStack.clear();

        // Create and store command
        ProfileChange change = new ProfileChange(
            email, fieldName, oldValue, newValue
        );
        undoStack.push(change);

        // Limit stack size
        if (undoStack.size() > MAX_HISTORY_SIZE) {
            undoStack.remove(0);
        }

        lastAccessTime.put(email, System.currentTimeMillis());
    }
}

// Execute undo command
public boolean undo(String email) {
    synchronized (this) {
        cleanupExpiredHistory(email);

        Stack<ProfileChange> undoStack = undoStacks.get(email);
        Stack<ProfileChange> redoStack = redoStacks
            .computeIfAbsent(email, k -> new Stack<>());

        if (undoStack == null || undoStack.isEmpty()) {
            return false;
        }

        ProfileChange change = undoStack.pop();

        try {
            // Execute undo: apply old value
            applyChange(change.email, change.fieldName, change.oldValue);

            // Move to redo stack
            redoStack.push(change);
            lastAccessTime.put(email, System.currentTimeMillis());

            return true;
        } catch (Exception e) {
            // Rollback on failure
```

```
            undoStack.push(change);
            return false;
        }
    }
}

// Execute redo command
public boolean redo(String email) {
    synchronized (this) {
        cleanupExpiredHistory(email);

        Stack<ProfileChange> undoStack = undoStacks
            .computeIfAbsent(email, k -> new Stack<>());
        Stack<ProfileChange> redoStack = redoStacks.get(email);

        if (redoStack == null || redoStack.isEmpty()) {
            return false;
        }

        ProfileChange change = redoStack.pop();

        try {
            // Execute redo: apply new value
            applyChange(change.email, change.fieldName, change.newValue);

            // Move back to undo stack
            undoStack.push(change);
            lastAccessTime.put(email, System.currentTimeMillis());

            return true;
        } catch (Exception e) {
            // Rollback on failure
            redoStack.push(change);
            return false;
        }
    }
}

// Command execution logic
private void applyChange(String email, String fieldName, String value)
    throws Exception {
    String path = "data/users/" + email + "/infoplus.json";

    InfoPlus infoPlus;
    if (Files.exists(Paths.get(path))) {
        String json = Files.readString(Paths.get(path));
        infoPlus = gson.fromJson(json, InfoPlus.class);
        if (infoPlus == null) {
            infoPlus = new InfoPlus();
        }
    } else {
        infoPlus = new InfoPlus();
    }

    // Apply field change
    switch (fieldName) {
        case "jobTitle":
            infoPlus.jobTitle = value != null ? value : "";
            break;
        case "phone":
            infoPlus.phone = value != null ? value : "";
            break;
        case "bio":
```

```
            infoPlus.bio = value != null ? value : "";
            break;
        case "profilePhoto":
            infoPlus.profilePhoto = value;
            break;
        default:
            throw new IllegalArgumentException("Unknown field: " + fieldName);
    }

    // Persist changes
    try (FileWriter fw = new FileWriter(path)) {
        gson.toJson(infoPlus, fw);
    }
}

// Query methods
public boolean canUndo(String email) {
    synchronized (this) {
        cleanupExpiredHistory(email);
        Stack<ProfileChange> stack = undoStacks.get(email);
        return stack != null && !stack.isEmpty();
    }
}

public boolean canRedo(String email) {
    synchronized (this) {
        cleanupExpiredHistory(email);
        Stack<ProfileChange> stack = redoStacks.get(email);
        return stack != null && !stack.isEmpty();
    }
}

public boolean hasChanges(String email) {
    synchronized (this) {
        cleanupExpiredHistory(email);
        Stack<ProfileChange> stack = undoStacks.get(email);
        return stack != null && !stack.isEmpty();
    }
}

public void clearHistory(String email) {
    synchronized (this) {
        undoStacks.remove(email);
        redoStacks.remove(email);
        lastAccessTime.remove(email);
    }
}

// Cleanup expired histories
private void cleanupExpiredHistory(String email) {
    Long lastAccess = lastAccessTime.get(email);
    if (lastAccess != null) {
        long elapsed = System.currentTimeMillis() - lastAccess;
        if (elapsed > HISTORY_TTL_MS) {
            undoStacks.remove(email);
            redoStacks.remove(email);
            lastAccessTime.remove(email);
        }
    }
}

public void cleanupAllExpiredHistories() {
    synchronized (this) {
```

```
            long currentTime = System.currentTimeMillis();
            List<String> expiredEmails = new ArrayList<>();

            lastAccessTime.forEach((email, lastAccess) -> {
                if (currentTime - lastAccess > HISTORY_TTL_MS) {
                    expiredEmails.add(email);
                }
            });

            expiredEmails.forEach(email -> {
                undoStacks.remove(email);
                redoStacks.remove(email);
                lastAccessTime.remove(email);
            });
        }
    }
}
```

**Usage Example:**

```
// Recording a change
profileCommandManager.recordChange(
    userEmail,
    "jobTitle",
    "Software Engineer",  // old value
    "Senior Engineer"     // new value
);

// Undoing a change
if (profileCommandManager.canUndo(userEmail)) {
    profileCommandManager.undo(userEmail);
}

// Redoing a change
if (profileCommandManager.canRedo(userEmail)) {
    profileCommandManager.redo(userEmail);
}
```

**Key Advantages:**

1. **Separation of Concerns:** Command objects separate the request from execution
2. **Undo/Redo Support:** Natural implementation of reversible operations
3. **History Management:** Commands can be stored and replayed
4. **Extensibility:** New command types can be added easily
5. **Thread Safety:** Concurrent access is properly handled

---

# 5. Proxy Pattern

## Pattern Description

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It acts as an intermediary that adds additional functionality without modifying the original object.

## Implementation Details

**Location:** `com.example.backend.Util.JsonFileManager`

**Type:** Smart Proxy (Virtual Proxy with added functionality)

**Purpose:** Acts as a centralized gateway for all file I/O operations, providing abstraction over the file system and adding cross-cutting concerns.

**Design Decisions:**

1. **Single Entry Point:** All file operations go through JsonFileManager, preventing direct file system access scattered throughout the codebase.
2. **Complexity Hiding:** Abstracts complex operations like GSON configuration, LocalDateTime serialization, and path construction.
3. **Generic Type Support:** Uses Java generics and reflection (TypeToken) to handle any DTO type, making it reusable across the application.
4. **Custom DateTime Handling:** Implements custom serializers/deserializers for LocalDateTime since JSON doesn't natively support it.
5. **Resource Management:** Uses try-with-resources to ensure proper file handle cleanup.
6. **Error Handling:** Catches and logs all IO exceptions, returning safe defaults (empty lists) rather than propagating exceptions.
7. **Lazy Initialization:** Creates parent directories only when needed using `mkdirs()`.
8. **User Folder Management:** Provides high-level methods for user folder creation and existence checks.

**Code Example:**

```
@Service
public class JsonFileManager {
    private static final DateTimeFormatter dateTimeFormatter =
        DateTimeFormatter.ISO_LOCAL_DATE_TIME;

    // GSON with custom LocalDateTime handling
    private static final Gson gson = new GsonBuilder()
        .setPrettyPrinting()
        .registerTypeAdapter(LocalDateTime.class,
            new JsonSerializer<LocalDateTime>() {
                @Override
                public JsonElement serialize(LocalDateTime src, Type typeOfSrc,
                                             JsonSerializationContext context) {
                    return new JsonPrimitive(src.format(dateTimeFormatter));
                }
            })
        .registerTypeAdapter(LocalDateTime.class,
            new JsonDeserializer<LocalDateTime>() {
                @Override
                public LocalDateTime deserialize(JsonElement json, Type typeOfT,
                                                 JsonDeserializationContext context) {
                    return LocalDateTime.parse(json.getAsString(),
                                               dateTimeFormatter);
                }
            })
        .create();

    private static final String basePath = "data/users/";

    // Generic read operation
    public <T> List<T> readListFromFile(String filePath, Type type) {
        File file = new File(filePath);
```

```
        if (!file.exists()) {
            return new ArrayList<>();
        }

        try (Reader reader = new FileReader(file)) {
            List<T> items = gson.fromJson(reader, type);
            return items != null ? items : new ArrayList<>();
        } catch (IOException e) {
            System.err.println("Error reading file: " + filePath);
            return new ArrayList<>();
        }
    }

    // Generic write operation
    public <T> boolean writeListToFile(String filePath, List<T> items) {
        try {
            File file = new File(filePath);
            file.getParentFile().mkdirs(); // Create directories if needed

            try (Writer writer = new FileWriter(file)) {
                gson.toJson(items, writer);
                return true;
            }
        } catch (IOException e) {
            System.err.println("Error writing to file: " + filePath);
            return false;
        }
    }

    // High-level user management
    public boolean createUserFolder(String email) {
        try {
            Path userPath = Paths.get(basePath + email);
            Files.createDirectories(userPath);

            // Create default files
            String[] folders = {
                "inbox.json", "sent.json", "trash.json",
                "draft.json", "contacts.json", "folders.json"
            };

            for (String folder : folders) {
                File file = new File(userPath.toString(), folder);
                if (!file.exists()) {
                    writeListToFile(file.getPath(), new ArrayList<>());
                }
            }
            return true;
        } catch (IOException e) {
            System.err.println("Error creating user folder: " + email);
            return false;
        }
    }

    // Utility methods
    public String getUserFolderPath(String email, String folder) {
        return basePath + email + "/" + folder + ".json";
    }

    public boolean userExists(String email) {
        File userFolder = new File(basePath + email);
        return userFolder.exists() && userFolder.isDirectory();
```

```
    }

    public boolean deleteFile(String filePath) {
        try {
            File file = new File(filePath);
            return file.exists() && file.delete();
        } catch (Exception e) {
            System.err.println("Error deleting file: " + filePath);
            return false;
        }
    }
}
```

**Usage Throughout Application:**

```
// In ContactService
@Service
public class ContactService {
    @Autowired
    private JsonFileManager jsonFileManager;

    public List<Contact> getContacts(String user) {
        // Auto-create user folder through proxy
        if (!jsonFileManager.userExists(user)) {
            jsonFileManager.createUserFolder(user);
        }

        String path = jsonFileManager.getUserFolderPath(user, "contacts");
        return jsonFileManager.readListFromFile(path, CONTACT_TYPE);
    }
}

// In mailService
@Service
public class mailService {
    private final JsonFileManager jsonFileManager;
    private static final Type MAIL_LIST_TYPE =
        new TypeToken<List<mail>>(){}.getType();

    public List<mail> getInboxEmails() {
        String inboxPath = BasePath + getLoggedInUser() + "/inbox.json";
        return jsonFileManager.readListFromFile(inboxPath, MAIL_LIST_TYPE);
    }
}
```

**Proxy Characteristics Demonstrated:**

1. **Access Control:** All file operations must go through JsonFileManager
2. **Lazy Initialization:** User folders created only when accessed
3. **Logging:** All operations are logged for debugging
4. **Error Handling:** Exceptions caught and handled centrally
5. **Resource Management:** Automatic cleanup via try-with-resources
6. **Caching Potential:** Could add caching layer without changing clients

**Key Advantages:**

1. **Single Responsibility:** All file I/O logic in one place
2. **Testability:** Easy to mock for unit tests
3. **Maintainability:** Changes to file structure require updates in one location

4. **Extensibility:** Can add features (caching, compression) without affecting clients
5. **Safety:** Prevents direct file manipulation throughout codebase

---

# 6. Singleton Pattern

## Pattern Description

The Singleton Pattern ensures a class has only one instance and provides a global point of access to it. It's commonly used for managing shared resources.

## Implementation Details

**Location:** Multiple service classes annotated with `@Service`

**Type:** Spring-managed Singleton

**Purpose:** Ensures single instance of services throughout application lifecycle, managing shared state and coordinating operations.

**Implementation Approach:** Spring Framework's Dependency Injection

Spring's `@Service`, `@Component`, and `@Repository` annotations automatically create singleton beans by default. The Spring IoC container manages the lifecycle and ensures only one instance exists.

**Services Implementing Singleton Pattern:**

### 6.1 ContactService

```
@Service
public class ContactService {
    @Autowired
    private contactRepo contactRepo;

    @Autowired
    private JsonFileManager jsonFileManager;

    @Autowired
    private ContactFactory contactFactory;

    // All methods use shared injected dependencies
}
```

### 6.2 mailService

```
@Service
public class mailService {
    @Autowired
    private EmailSortContext emailSortContext;

    @Autowired
    private EmailFilterService emailFilterService;

    @Autowired
```

```
    private mailRepo mailRepo;

    private final Object trashLock = new Object();
    private final Object folderLock = new Object();

    // Synchronized methods for thread-safe operations
}
```

### 6.3 FolderService

```
@Service
public class FolderService {
    @Autowired
    private FolderRepo folderRepo;

    @Autowired
    private JsonFileManager jsonFileManager;

    @Autowired
    private FolderFactory folderFactory;

    // Manages all folder operations through single instance
}
```

### 6.4 EmailFilterService

```
@Service
public class EmailFilterService {
    // Stateless service - safe for concurrent use

    public List<mail> applyFilters(List<mail> emails,
                                   FilterCriteriaDTO criteria) {
        // Builds filter chain on each call
    }
}
```

### 6.5 ProfileCommandManager

```
@Service
public class ProfileCommandManager {
    // Manages per-user state in thread-safe collections
    private final Map<String, Stack<ProfileChange>> undoStacks =
        new ConcurrentHashMap<>();
    private final Map<String, Stack<ProfileChange>> redoStacks =
        new ConcurrentHashMap<>();

    // Synchronized methods for thread safety
}
```

### 6.6 TrashCleanupService

```
@Service
public class TrashCleanupService {
    private final JsonFileManager jsonFileManager;

    @Scheduled(cron = "0 0 2 * * *")
    public void cleanupOldTrashEmails() {
        // Runs daily as singleton scheduled task
    }
}
```

### 6.7 EmailSortContext

```
@Component
public class EmailSortContext {
    private final Map<String, EmailSortStrategy> strategies;

    public EmailSortContext() {
        this.strategies = new HashMap<>();
        registerDefaultStrategies();
    }

    // Shared strategy registry across application
}
```

**Design Decisions:**

1. **Spring-Managed Lifecycle:** Leveraging Spring's IoC container eliminates need for manual singleton implementation (getInstance() methods, private constructors).
2. **Thread Safety Considerations:**
    - Stateless services (EmailFilterService, FolderService) are inherently thread-safe
    - Stateful services (ProfileCommandManager, mailService) use synchronization
    - Immutable shared data (EmailSortContext strategies) is safe
3. **Dependency Injection:** All singletons receive dependencies via `@Autowired`, promoting loose coupling and testability.
4. **Lazy vs Eager Initialization:** Spring creates beans lazily by default, but services are typically initialized on first use.
5. **Scope Declaration:** While `@Service` defaults to singleton, explicit scope can be declared:

```
@Service
@Scope("singleton") // Explicit, but default
public class ContactService { }
```

**Thread Safety Strategies:**

**Stateless Services (Thread-Safe):**

```
@Service
public class EmailFilterService {
    // No instance variables
    // Each method call is independent
    // Safe for concurrent access
}
```

**Synchronized Access:**

```
@Service
public class ProfileCommandManager {
    public void recordChange(...) {
        synchronized (this) {
            // Critical section
        }
    }
}
```

**Lock Objects:**

```
@Service
```

```
public class mailService {
    private final Object trashLock = new Object();

    public boolean moveEmail(...) {
        synchronized (folderLock) {
            // Folder operations
        }
    }
}
```

**Concurrent Collections:**

```
@Service
public class ProfileCommandManager {
    private final Map<String, Stack<ProfileChange>> undoStacks =
        new ConcurrentHashMap<>();
    // Thread-safe map
}
```

**Scheduled Tasks:**

```
@Service
public class TrashCleanupService {
    // Single instance runs scheduled tasks
    @Scheduled(cron = "0 0 2 * * *")
    public void cleanupOldTrashEmails() {
        // Executes once per day
    }
}
```

---

## Pattern Usage Overview

| Pattern | Location | Primary Purpose |
|---|---|---|
| **Factory** | ContactFactory, FolderFactory, mailFactory | Object creation and initialization |
| **Strategy** | Contact sorting, Email sorting | Flexible algorithm selection |
| **Chain of Responsibility** | Email filters | Sequential processing pipeline |
| **Command** | ProfileCommandManager | Undo/redo operations |
| **Proxy** | JsonFileManager | Controlled file system access |
| **Singleton** | All @Service classes | Single shared instance |

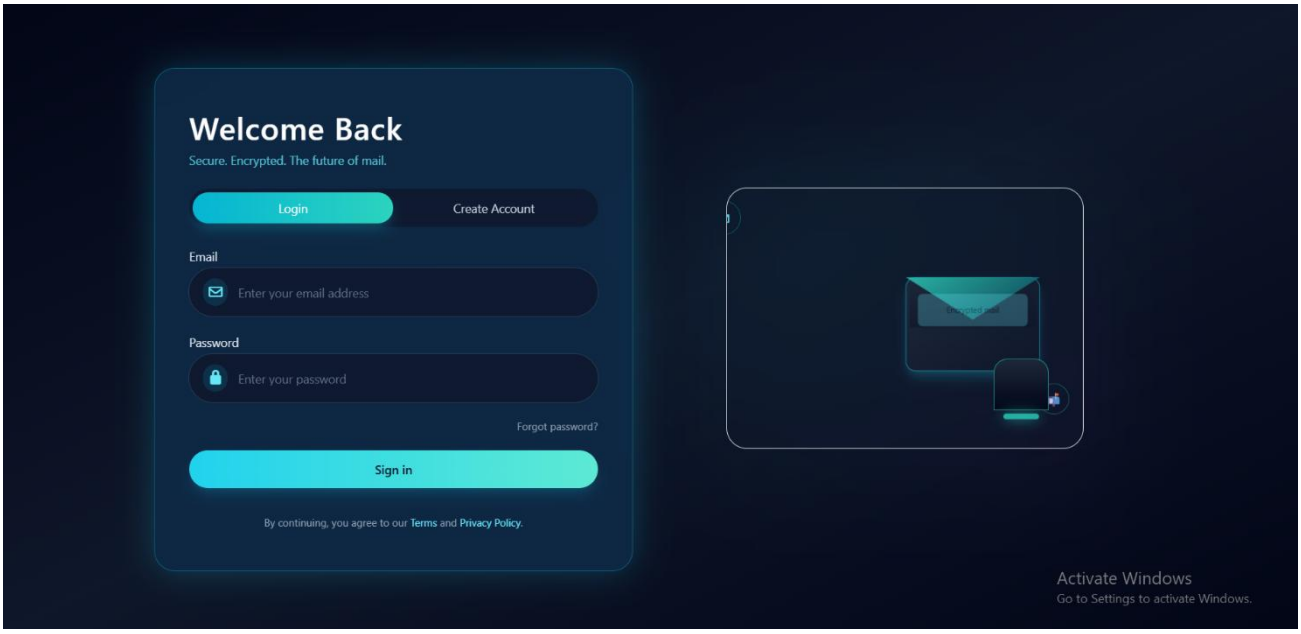
## Key Design Principles Applied

1. **Separation of Concerns:** Each pattern addresses a specific responsibility
2. **Open/Closed Principle:** Easy to extend without modifying existing code
3. **Dependency Inversion:** Depend on abstractions (interfaces) not concrete classes
4. **Single Responsibility:** Each class has one reason to change
5. **Composition Over Inheritance:** Favor object composition (Strategy, Chain)

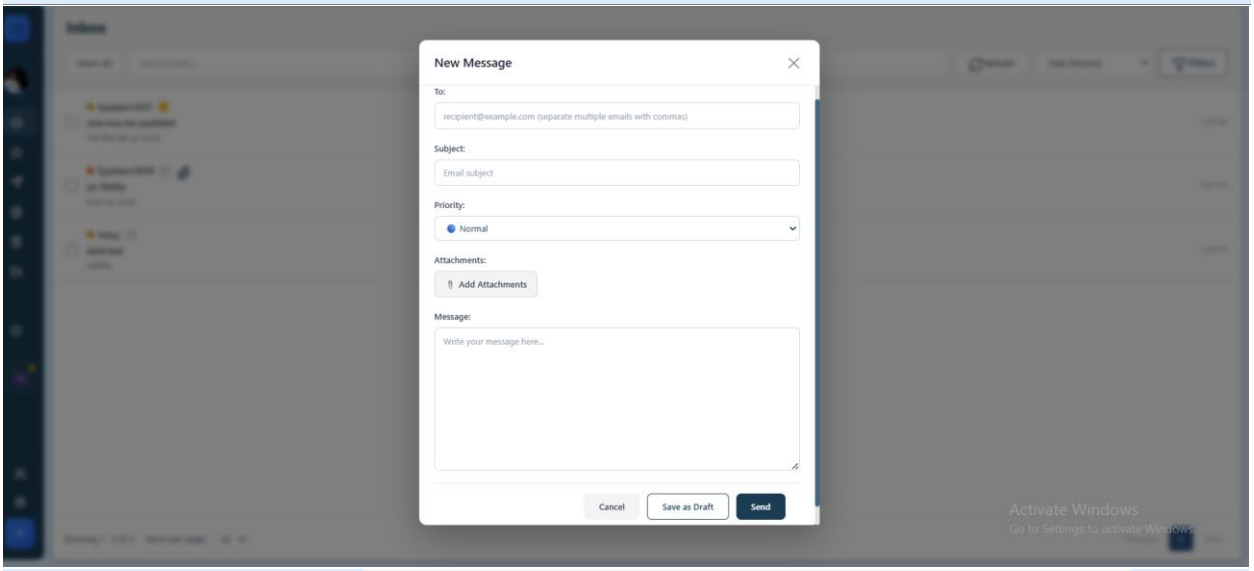
## Pattern Interactions

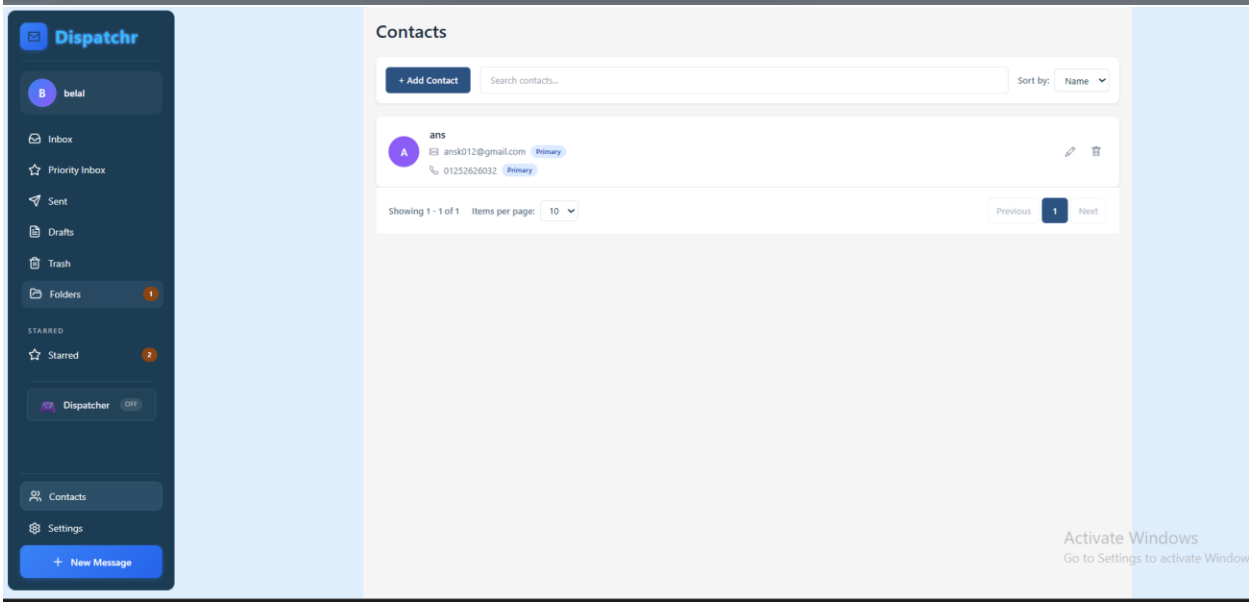
1. **Factory + Service:** Services use factories to create domain objects
2. **Strategy + Chain:** Emails are filtered (Chain) then sorted (Strategy)
3. **Proxy + Service:** All services use JsonFileManager proxy for persistence
4. **Singleton + Factory:** Singleton services contain factory dependencies
5. **Command + Service:** ProfileCommandManager implements command pattern as service

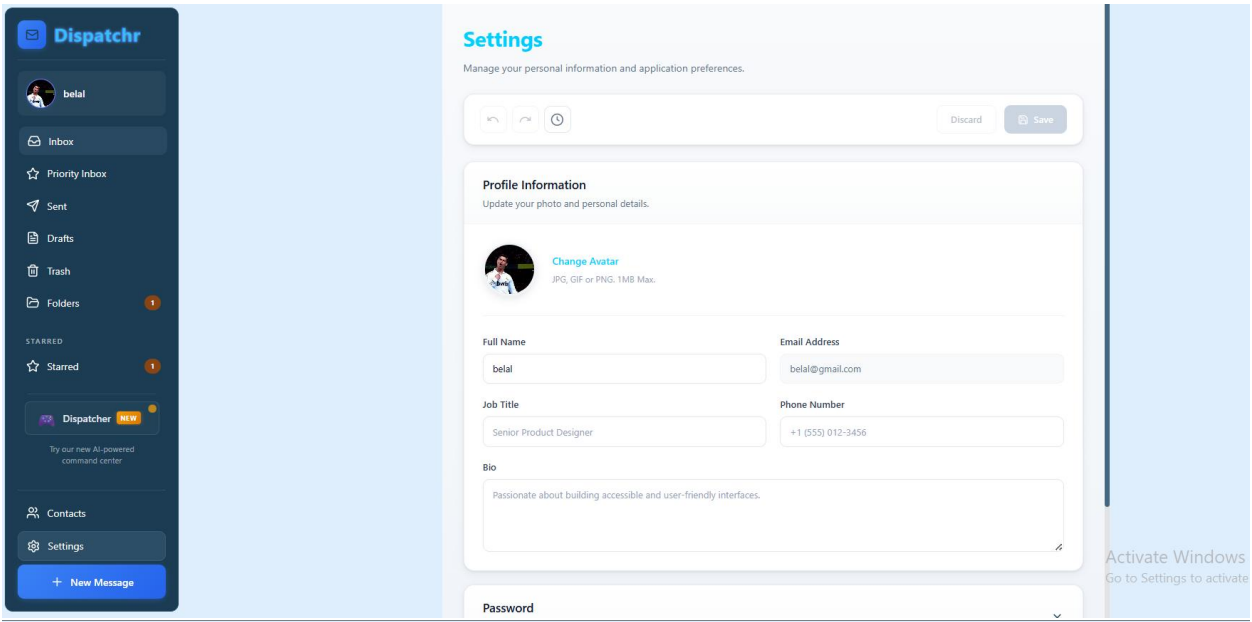
---

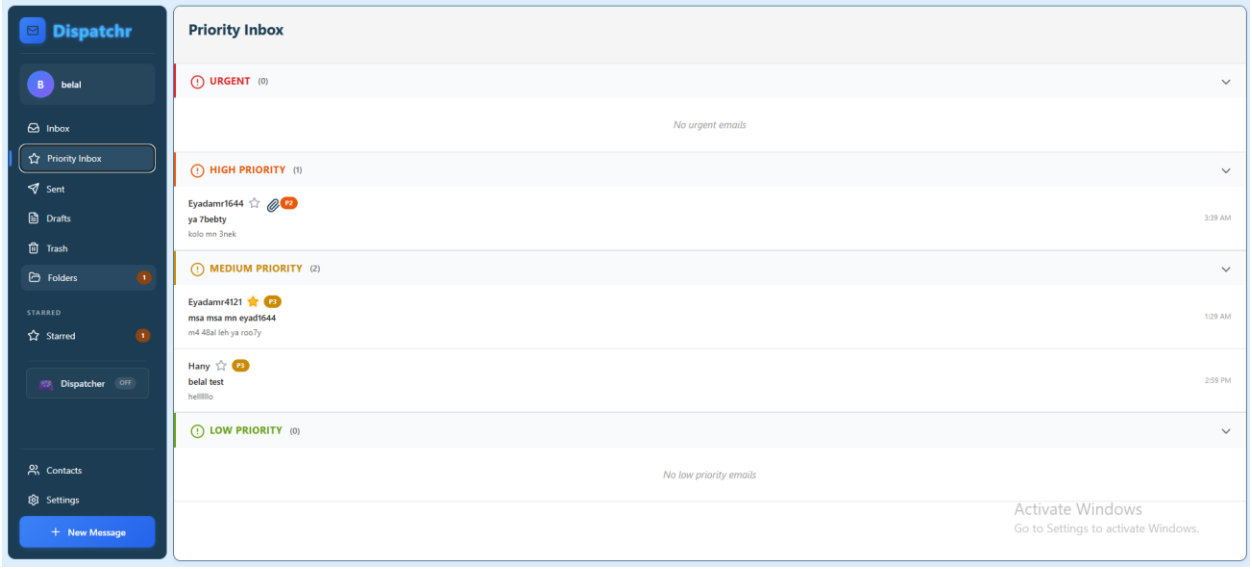



Custom Folders
Organize your emails with custom folders
+ New Folder
belal
1 emails
View All
Folder Tips
Drag and drop emails into folders to organize them
Set up automatic rules to sort incoming emails
Use colors to visually distinguish between folder types
Activate Windows
Go to Settings to activate Windows.


New Message
To:
recipient@example.com (separate multiple emails with commas)
Subject:
Email subject
Priority:
Normal
Attachments:
Add Attachments
Message:
Write your message here...
Cancel
Save as Draft
Send
Activate Windows
Go to Settings to activate Windows.


Dispatchr
belal
Inbox
Priority Inbox
Sent
Drafts
Trash
Folders
STARRED
Starred
Dispatcher
Contacts
Settings
New Message
Contacts
+ Add Contact
Search contacts...
Sort by:
Name
ans
ansk012@gmail.com
Primary
01252626032
Primary
Showing 1 - 1 of 1
Items per page:
10
Previous
1
Next
Activate Windows
Go to Settings to activate Windows.


Dispatchr
belal
Inbox
Priority Inbox
Sent
Drafts
Trash
Folders
STARRED
Starred
Dispatcher NEW
Try our new AI-powered command center
Contacts
Settings
New Message
Settings
Manage your personal information and application preferences.
Discard
Save
Profile Information
Update your photo and personal details.
Change Avatar
JPG, GIF or PNG. 1MB Max.
Full Name
belal
Email Address
belal@gmail.com
Job Title
Senior Product Designer
Phone Number
+1 (555) 012-3456
Bio
Passionate about building accessible and user-friendly interfaces.
Password
Activate Windows
Go to Settings to activate


Dispatchr
belal
Inbox
Priority Inbox
Sent
Drafts
Trash
Folders
STARRED
Starred
Dispatcher OFF
Contacts
Settings
New Message
Priority Inbox
URGENT (0)
No urgent emails
HIGH PRIORITY (1)
Eyadamr1644 P2
ya 7ebty
kolo mn 3nek
3:39 AM
MEDIUM PRIORITY (2)
Eyadamr4121 P3
msa msa mn eyad1644
m4 48al leh ya roo7y
1:29 AM
Hany P3
belal test
hellllllo
2:59 PM
LOW PRIORITY (0)
No low priority emails
Activate Windows
Go to Settings to activate Windows.

```java
/**
 * Compose and send a new email
 */
public void composeMail(mailContentDTO mailContent) throws UserNotFoundException {
    String currentUser = getLoggedInUser();
    String sentPath = BasePath + currentUser + "/sent.json";
    mail mail = mailFactory.createNewMail(mailContent);
    mail.setFrom(currentUser);

    // Add to sent folder
    Queue<String> recipientsQueue = mailContent.getRecipients();
    String receiver = recipientsQueue.peek();

    if (!(jsonFileManager.userExists(receiver))|| receiver.equals(currentUser)) {
        System.out.println("this email{ " + receiver + " } isn't found in our system");
        throw new UserNotFoundException("Email address " + receiver + " is not registered in our sy
    }

    List<mail> sentMails = jsonFileManager.readListFromFile(sentPath, MAIL_LIST_TYPE);
    sentMails.add(mail);
    jsonFileManager.writeListToFile(sentPath, sentMails);

    // Send to all receivers
    mail.setTo(to: null);
    String path = BasePath + receiver + "/inbox.json";
    List<mail> inboxMails = jsonFileManager.readListFromFile(path, MAIL_LIST_TYPE);
    inboxMails.add(mail);
    jsonFileManager.writeListToFile(path, inboxMails);
}
```

```java
public class sortByEmail implements contactSortStrategy {


    @Override
    public void sort(List<Contact> contacts)
    {
        //sort contacts by email
        contacts.sort((a,b)->
        {
            String emailA = a.getEmail().get(0).getAddress().toLowerCase();
            String emailB = b.getEmail().get(0).getAddress().toLowerCase();
            return emailA.compareTo(emailB);
        });
    }
}
```

```java
public List<mail> getInboxEmails() {  1 usage  Belal
    String inboxPath = BasePath + getLoggedInUser() + "/inbox.json";
    return jsonFileManager.readListFromFile(inboxPath, MAIL_LIST_TYPE);
}
public List<mail> getSentEmails() {  1 usage  Belal
    String sentPath = BasePath + getLoggedInUser() + "/sent.json";
    return jsonFileManager.readListFromFile(sentPath, MAIL_LIST_TYPE);
}

public List<mail> getTrashEmails() {  1 usage  Belal
    String trashPath = BasePath + getLoggedInUser() + "/trash.json";
    return jsonFileManager.readListFromFile(trashPath, MAIL_LIST_TYPE);
}
public List<mail> getDraftEmails() {  1 usage  Belal
    String draftPath = BasePath + getLoggedInUser() + "/draft.json";
    return jsonFileManager.readListFromFile(draftPath, MAIL_LIST_TYPE);
}
```

```java
@Data  Belal +1 *
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class mail implements Cloneable {
    private int id;
    private Queue<String> to;
    private String from;
    private String subject;
    private String body;
    private String preview;
    private boolean starred;
    private boolean hasAttachment;
    private LocalDateTime timestamp;
    private int priority;
    private List<attachementDTO> attachments;
    private LocalDateTime trashedAt;
    private String folder;
    private String customFolderId;
}
```

```java
    public static mail createNewMail(mailContentDTO mailContent) {  2 usages  Belal
        mail mail = new mail();
        mail.setId(++idCounter);
        mail.setSubject(mailContent.getSubject());
        mail.setBody(mailContent.getBody());
        mail.setTo(mailContent.getRecipients());
        mail.setTimestamp(LocalDateTime.now());
        mail.setStarred(false);
        mail.setHasAttachment(!(mailContent.getAttachements().isEmpty()));
        mail.setPriority(mailContent.getPiriority());
        mail.setPreview(mailContent.getBody().substring(0, Math.min(100, mailContent.getBody().length())));
        mail.setAttachments(mailContent.getAttachements());

        // Save the updated counter after creating new mail
        saveCounterToFile();

        return mail;
    }
}
```