Email Client UI (Outlook-like)

□ Table of Contents

Table of Contents

Clarify the Problem and Requirements

□ Back to Top

Problem Understanding

□ Back to Top

Design a comprehensive web-based email client interface that provides rich email management capabilities, similar to Microsoft Outlook, Gmail, or Thunderbird. The system must handle large volumes of emails efficiently, support advanced organizational features, provide real-time synchronization, and deliver an intuitive user experience across different devices while maintaining security and performance standards.

Functional Requirements

□ Back to Top

- **Email Management**: Send, receive, reply, forward, delete emails with rich text composition
- **Folder Organization**: Hierarchical folder structure, custom folders, smart folders, drag-and-drop
- **Search & Filtering**: Advanced search, saved searches, filters, sorting options
- **Email Threading**: Conversation grouping, thread management, collapsed/expanded views
- **Label System**: Color-coded labels, multiple labels per email, label-based filtering
- **Attachment Handling**: Upload, download, preview attachments, virus scanning
- **Offline Support**: Offline reading, compose drafts, sync when online
- **Multi-account Support**: Multiple email accounts, unified inbox, account switching

Non-Functional Requirements

☐ Back to Top

- **Performance**: <2s initial load, <500ms email open time, smooth scrolling with 10K+ emails
- **Scalability**: Handle mailboxes with 100K+ emails, efficient virtualization
- **Reliability**: 99.9% uptime, robust error handling, automatic recovery
- **Security**: End-to-end encryption, secure authentication, malware protection
- **Cross-platform**: Responsive design for desktop, tablet, mobile browsers
- **Accessibility**: WCAG 2.1 AA compliance, keyboard navigation, screen reader support
- **Real-time Updates**: Live email synchronization, push notifications
- **Memory Efficiency**: Optimal memory usage, efficient caching, no memory leaks

Key Assumptions

☐ Back to Top

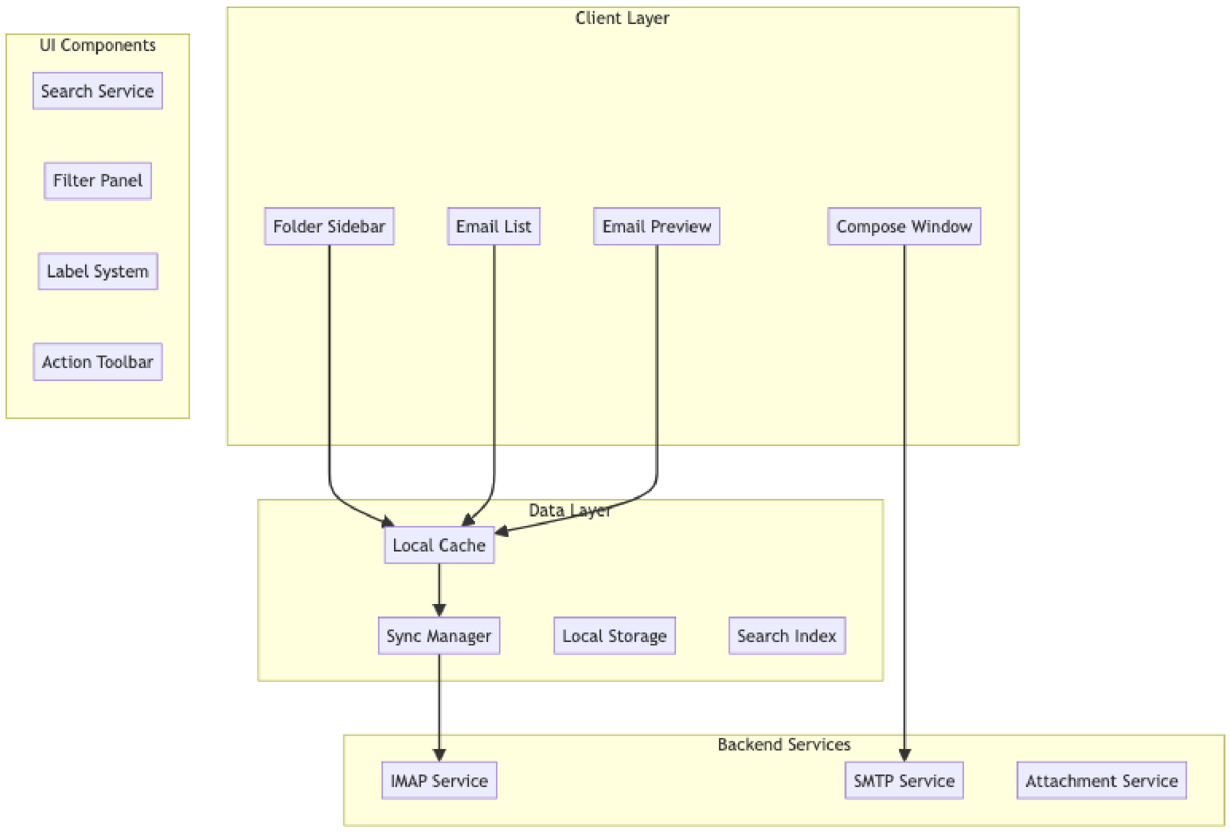
- Average mailbox size: 10K-50K emails, power users up to 100K+ emails
- Email volume: 50-200 emails per day for typical users
- Attachment sizes: Average 1-5MB, maximum 25MB per email
- Device usage: 70% desktop, 20% mobile, 10% tablet
- Network conditions: Variable from mobile to high-speed connections
- User behavior: Heavy email readers, moderate composers, organizers
- Email protocols: IMAP/POP3 for receiving, SMTP for sending
- Storage: Local caching with server synchronization

High-Level Design (HLD)

☐ Back to Top

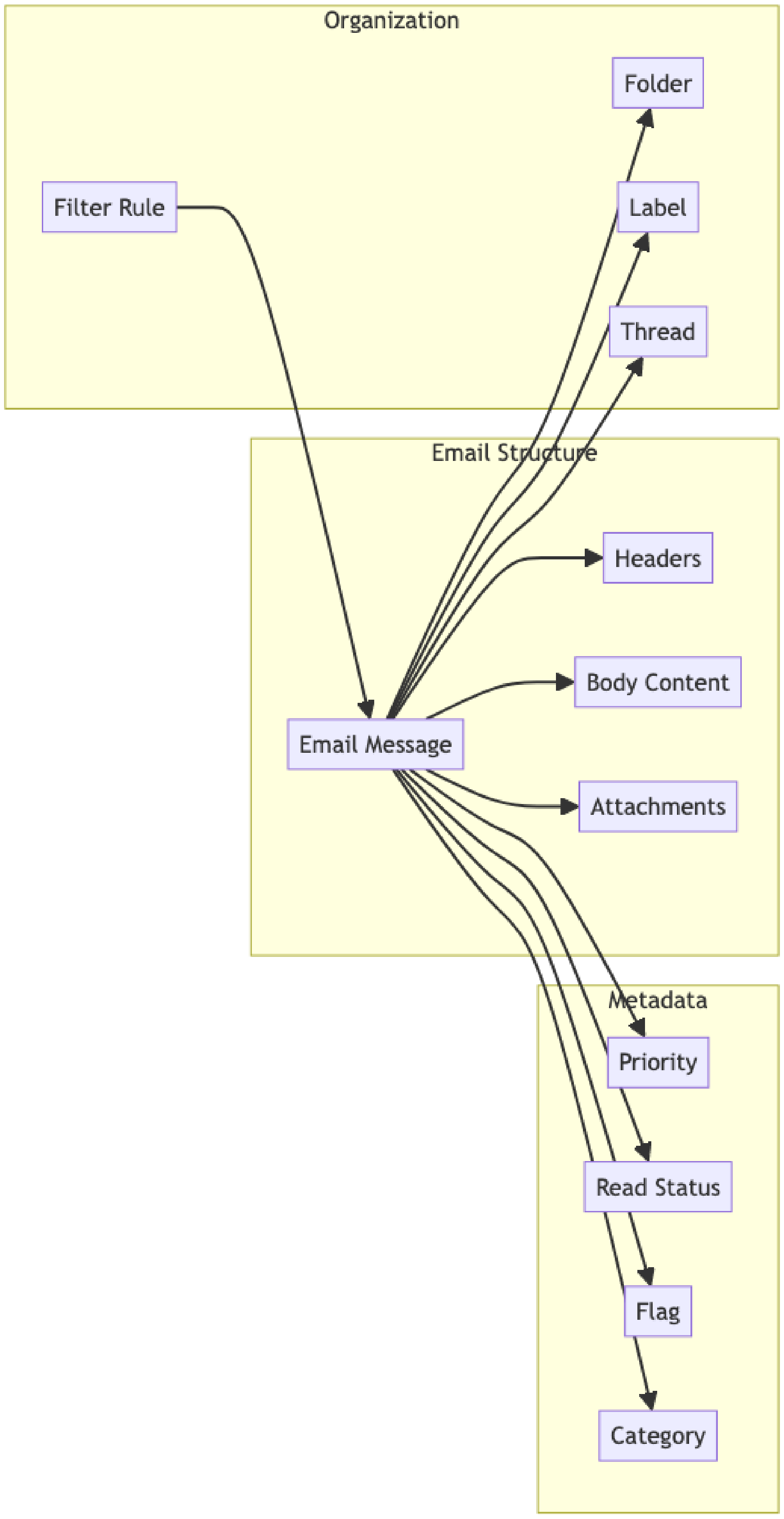
System Architecture Overview

☐ Back to Top



Email Data Model

☐ Back to Top

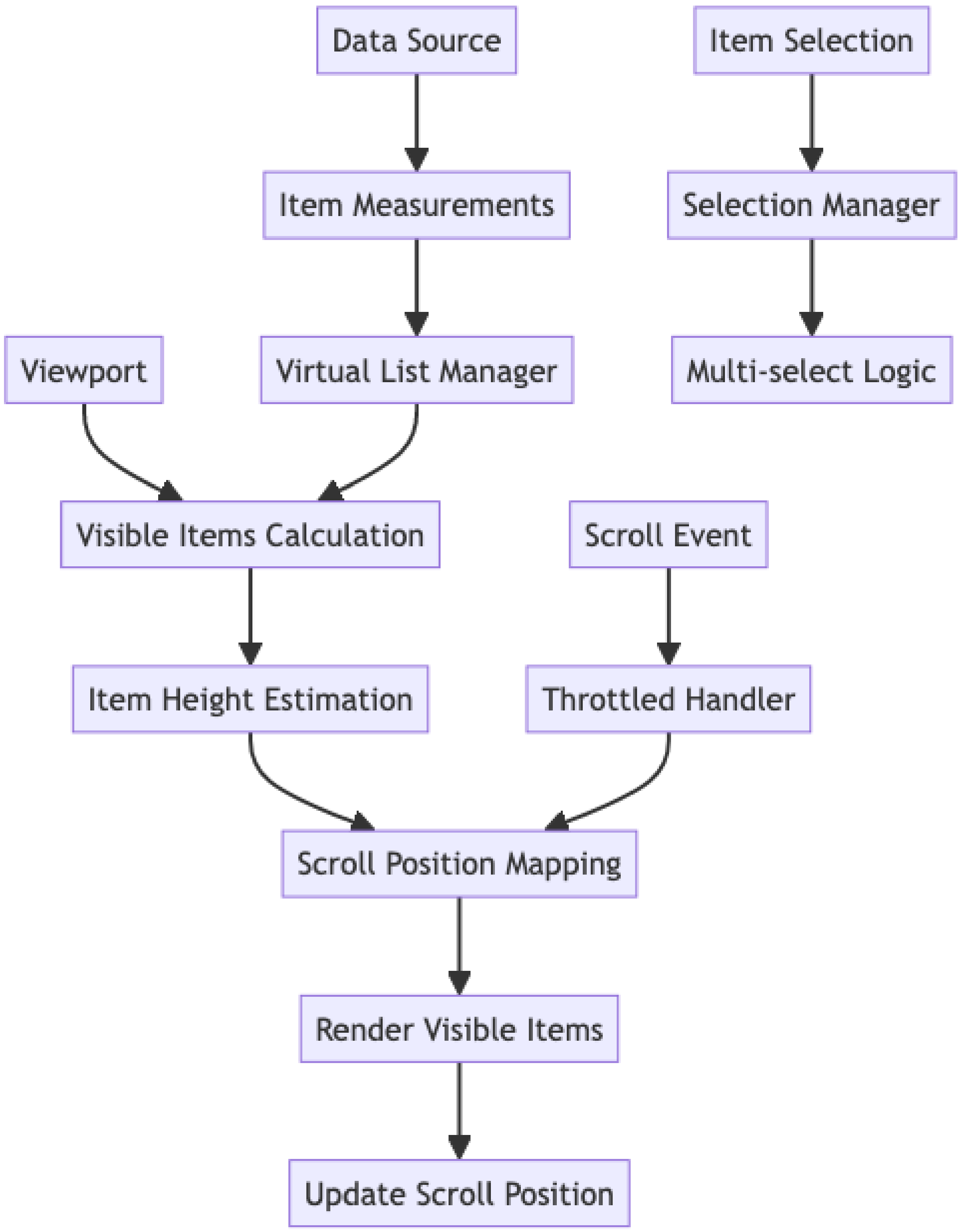

Organization
Folder
Filter Rule
Label
Thread
Email Structure
Headers
Body Content
Email Message
Attachments
Metadata
Priority
Read Status
Flag
Category

Low-Level Design (LLD)

☐ Back to Top

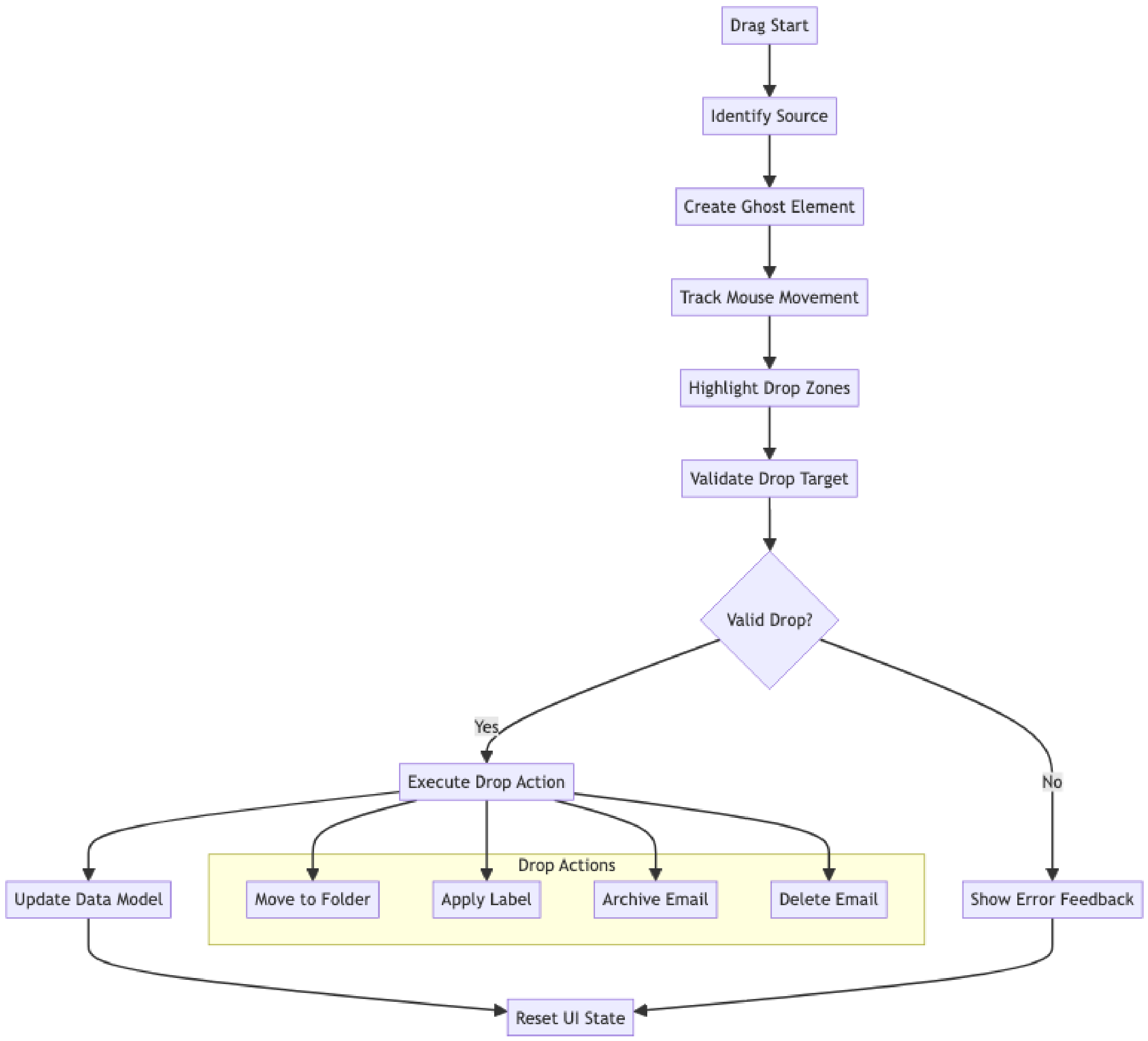
Email List Virtualization

☐ Back to Top


Data Source
Item Selection
Item Measurements
Selection Manager
Viewport
Virtual List Manager
Multi-select Logic
Visible Items Calculation
Scroll Event
Item Height Estimation
Throttled Handler
Scroll Position Mapping
Render Visible Items
Update Scroll Position

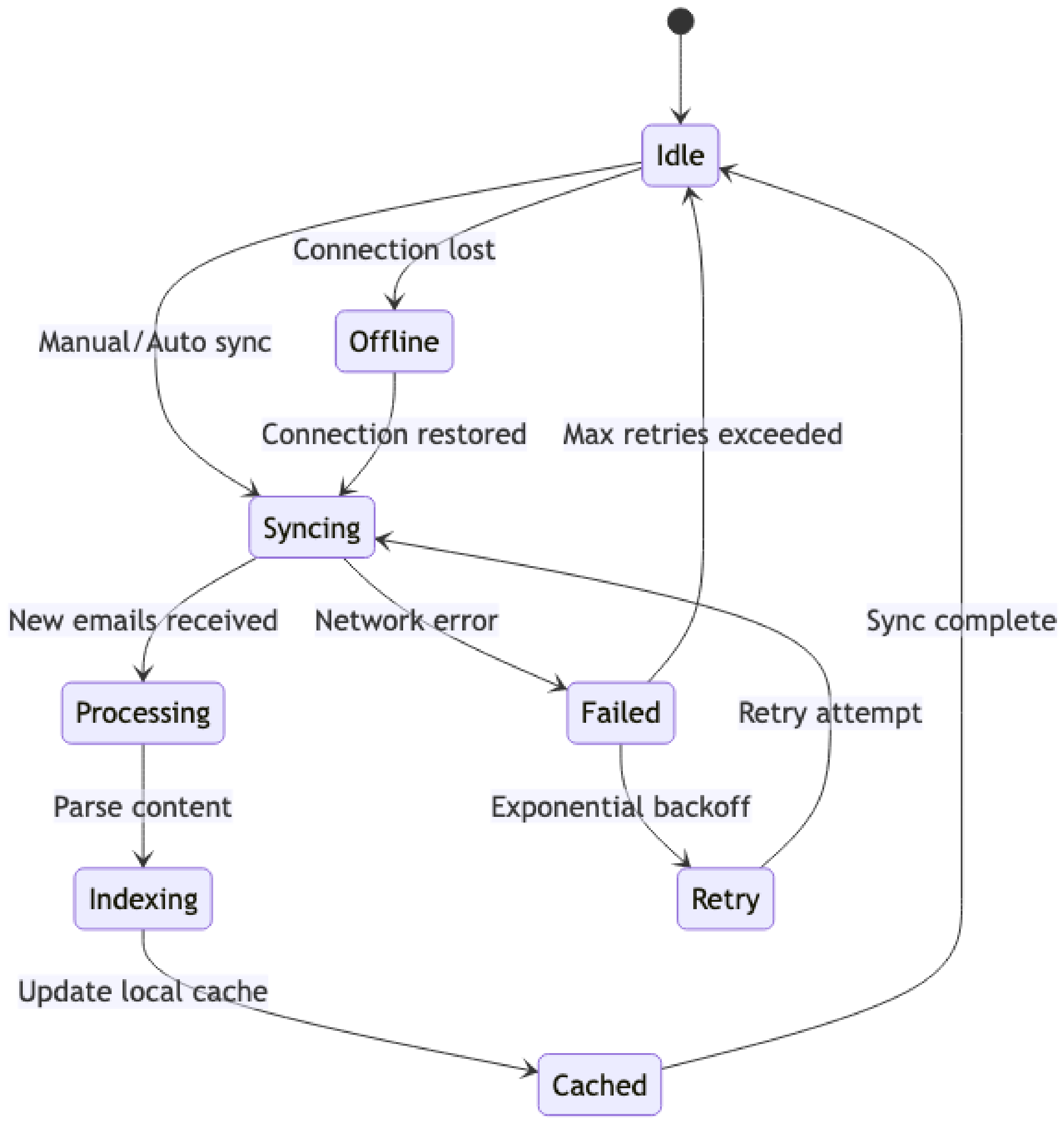
Drag and Drop System

☐ Back to Top



Email Synchronization State Machine

☐ Back to Top



Core Algorithms

☐ Back to Top

---

1. Virtual List Rendering Algorithm

☐ Back to Top

Purpose: Efficiently render large email lists without performance degradation.

Key Components:

```
VirtualList = {
  itemHeight: number | function,
  containerHeight: number,
  scrollTop: number,
  totalItems: number,
  overscan: number
}
```

Rendering Algorithm:

```
function calculateVisibleRange(virtualList):
  startIndex = Math.floor(scrollTop / itemHeight)
  endIndex = Math.min(
    startIndex + Math.ceil(containerHeight / itemHeight) + overscan,
    totalItems - 1
  )

  return {
    start: Math.max(0, startIndex - overscan),
    end: endIndex
  }
```

Dynamic Height Handling: - Measure rendered items to get actual heights - Cache height measurements for performance - Recalculate scroll position when heights change - Implement binary search for position lookup

2. Email Threading Algorithm

□ Back to Top

Purpose: Group related emails into conversation threads.

Threading Strategy:

```
function buildThreads(emails):
  threads = new Map()

  for email in emails:
    threadId = extractThreadId(email)
    if not threadId:
      threadId = generateThreadId(email.subject, email.references)
    if not threads.has(threadId):
      threads.set(threadId, new Thread())

    thread = threads.get(threadId)
    thread.addEmail(email)
    thread.sortByDate()

  return Array.from(threads.values())
```

Subject Normalization: - Remove “Re:”, “Fwd:”, and language variants - Normalize whitespace and case - Handle subject line truncation - Account for automated subject modifications

3. Intelligent Search Algorithm

□ Back to Top

Multi-field Search Strategy:

```
SearchQuery = {
  text: string,
  sender: string,
  subject: string,
  dateRange: DateRange,
  hasAttachment: boolean,
  labels: Label[],
  folder: Folder
}
```

Ranking Algorithm:

```
function calculateRelevanceScore(email, query):
  score = 0

  // Text content matching
  score += textMatch(email.body, query.text) * 0.4
  score += textMatch(email.subject, query.text) * 0.3
  score += textMatch(email.sender, query.text) * 0.2

  // Exact field matches
  if query.sender and email.sender.includes(query.sender):
    score += 0.5

  // Recency boost
  daysSince = (now - email.date) / (24 * 60 * 60 * 1000)
  score += Math.max(0, (30 - daysSince) / 30) * 0.1
  return score
```

Full-text Search Optimization: - Implement inverted index for fast text search - Use stemming and tokenization - Support fuzzy matching for typos - Cache popular search queries

4. Label Management System

☐ Back to Top

Hierarchical Label Structure:

```
Label = {
  id: string,
  name: string,
  color: string,
  parent?: Label,
  children: Label[],
  isSystem: boolean
}
```

Label Application Algorithm:

```
function applyLabel(emails, label):
  batch = new LabelBatch()

  for email in emails:
    if not email.labels.includes(label):
      batch.add(email.id, label.id)
      email.labels.push(label)

      // Apply parent labels automatically
      parent = label.parent
      while parent and not email.labels.includes(parent):
        batch.add(email.id, parent.id)
        email.labels.push(parent)
        parent = parent.parent

  return batch.execute()
```

5. Folder Synchronization Algorithm

☐ Back to Top

Incremental Sync Strategy:

```
function syncFolder(folder):
  lastSync = getLastSyncTime(folder)

  // Get server state
  serverState = await fetchFolderState(folder, lastSync)
  localState = getLocalFolderState(folder)

  // Calculate differences
  diff = calculateDiff(serverState, localState)

  // Apply changes
  for change in diff.additions:
    await downloadEmail(change.messageId)

  for change in diff.deletions:
    await deleteLocalEmail(change.messageId)

  for change in diff.modifications:
    await updateLocalEmail(change.messageId, change.flags)

  setLastSyncTime(folder, now)
```

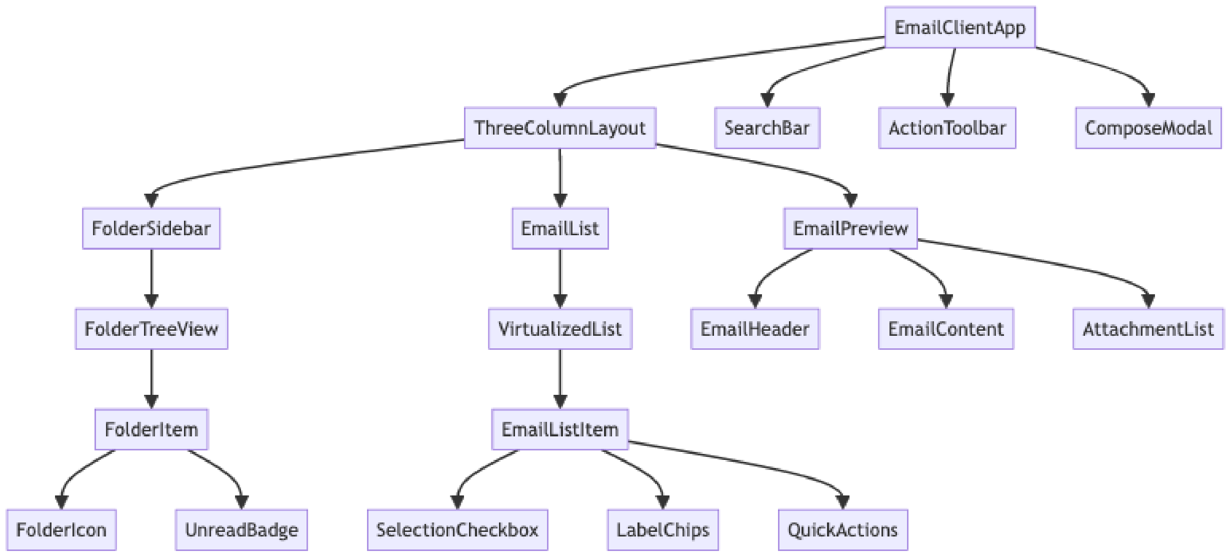
Conflict Resolution: - Server state takes precedence for flags - Local deletions are preserved unless email exists on server - Handle concurrent modifications gracefully - Implement rollback for failed operations

Component Architecture

☐ Back to Top

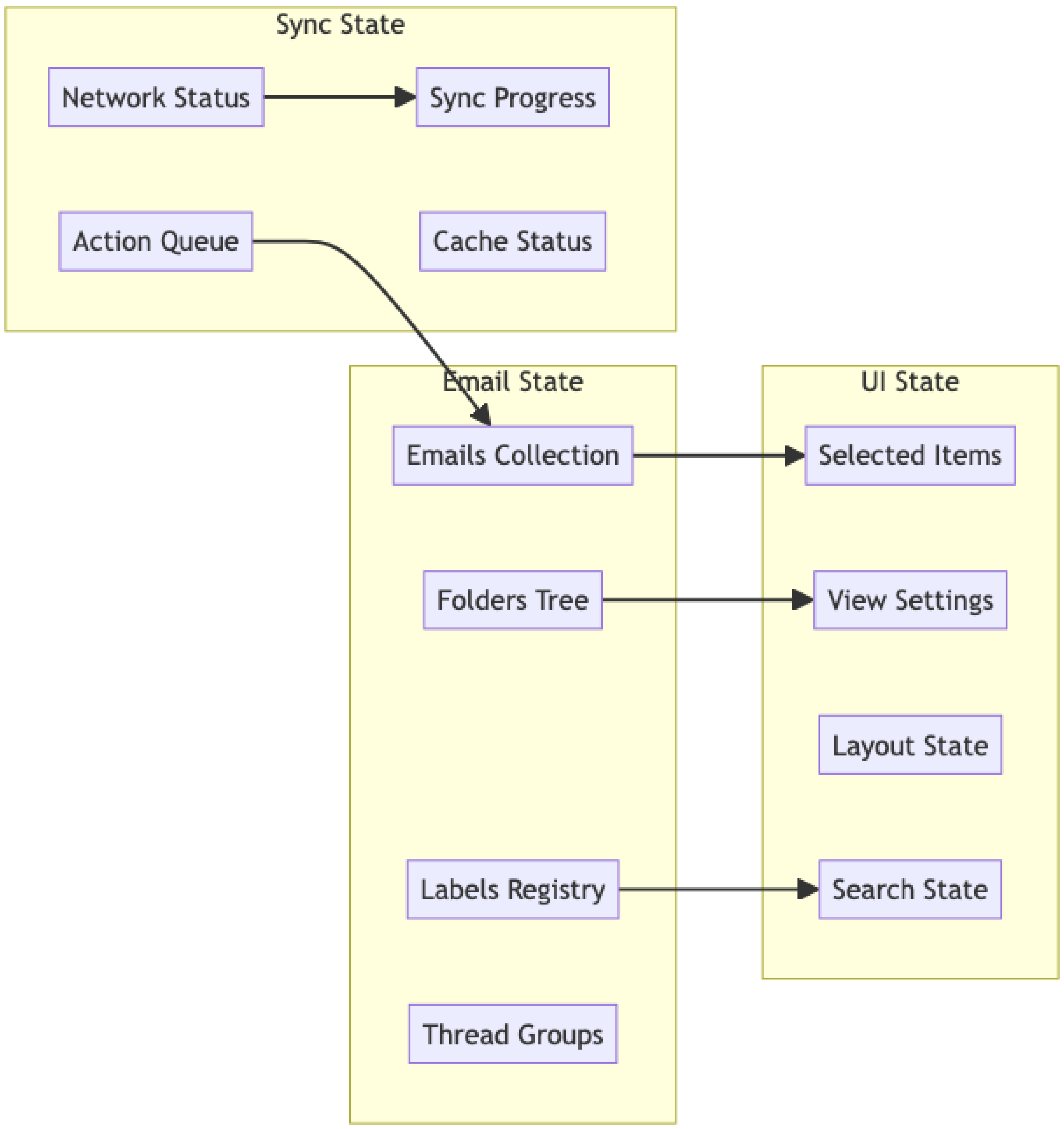
Email Client Component Hierarchy

☐ Back to Top



State Management Architecture

☐ Back to Top



React Component Implementation ☐ Back to Top

EmailClientApp.jsx

```
import React, { useState, useEffect, useCallback } from 'react';
import { EmailProvider } from './EmailContext';
import ThreeColumnLayout from './ThreeColumnLayout';
import SearchBar from './SearchBar';
import ActionToolbar from './ActionToolbar';
import ComposeModal from './ComposeModal';
import { useEmailSync } from './hooks/useEmailSync';

const EmailClientApp = () => {
  const [emails, setEmails] = useState([]);
  const [folders, setFolders] = useState([]);
  const [selectedEmails, setSelectedEmails] = useState([]);
  const [currentFolder, setCurrentFolder] = useState('inbox');
  const [selectedEmail, setSelectedEmail] = useState(null);
  const [searchQuery, setSearchQuery] = useState('');
  const [isComposing, setIsComposing] = useState(false);
  const [viewSettings, setViewSettings] = useState({
    density: 'comfortable',
    groupByThread: true,
    showPreview: true
  });

  const { syncStatus, syncEmails, isOnline } = useEmailSync();

  useEffect(() => {
    loadInitialData();
  }, []);

  useEffect(() => {
    if (currentFolder) {
      loadFolderEmails(currentFolder);
    }
  }, [currentFolder, searchQuery]);

  const loadInitialData = async () => {
    try {
      const [foldersData, emailsData] = await Promise.all([
        fetch('/api/folders').then(r => r.json()),
        fetch('/api/emails?folder=inbox&limit=50').then(r => r.json())
      ]);

      setFolders(foldersData.folders);
      setEmails(emailsData.emails);
    } catch (error) {
      console.error('Failed to load initial data:', error);
    }
  };

  const loadFolderEmails = async (folderId) => {
```

```
  try {
    const params = new URLSearchParams({
      folder: folderId,
      limit: 100,
      ...(searchQuery && { search: searchQuery })
    });

    const response = await fetch(`/api/emails?${params}`);
    const data = await response.json();
    setEmails(data.emails);
  } catch (error) {
    console.error('Failed to load folder emails:', error);
  }
};

const handleEmailSelect = useCallback((emailId) => {
  const email = emails.find(e => e.id === emailId);
  setSelectedEmail(email);

  // Mark as read if unread
  if (email && !email.isRead) {
    markAsRead(emailId);
  }
}, [emails]);

const handleEmailsSelection = useCallback((emailIds) => {
  setSelectedEmails(emailIds);
}, []);

const markAsRead = useCallback(async (emailId) => {
  setEmails(prev => prev.map(email =>
    email.id === emailId ? { ...email, isRead: true } : email
  ));

  try {
    await fetch(`/api/emails/${emailId}/read`, { method: 'POST' });
  } catch (error) {
    console.error('Failed to mark as read:', error);
    // Revert on error
    setEmails(prev => prev.map(email =>
      email.id === emailId ? { ...email, isRead: false } : email
    ));
  }
}, []);
```

```
const archiveEmails = useCallback(async (emailIds) => {
  const emailsToArchive = emailIds || selectedEmails;

  setEmails(prev => prev.filter(email => !emailsToArchive.includes(email.id)));
  setSelectedEmails([]);

  try {
    await fetch('/api/emails/archive', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ emailIds: emailsToArchive })
    });
  } catch (error) {
    console.error('Failed to archive emails:', error);
    loadFolderEmails(currentFolder);
  }
}, [selectedEmails, currentFolder]);

const deleteEmails = useCallback(async (emailIds) => {
  const emailsToDelete = emailIds || selectedEmails;

  setEmails(prev => prev.filter(email => !emailsToDelete.includes(email.id)));
  setSelectedEmails([]);

  try {
    await fetch('/api/emails/delete', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ emailIds: emailsToDelete })
    });
  } catch (error) {
    console.error('Failed to delete emails:', error);
    loadFolderEmails(currentFolder);
  }
}, [selectedEmails, currentFolder]);

const sendEmail = useCallback(async (emailData) => {
  try {
    const response = await fetch('/api/emails/send', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(emailData)
    });

    if (response.ok) {
      setIsComposing(false);
      // Refresh sent folder if currently viewing
      if (currentFolder === 'sent') {
        loadFolderEmails(currentFolder);
      }
    }
  } catch (error) {
    console.error('Failed to send email:', error);
    throw error;
  }
}, [currentFolder]);

const handleSearch = useCallback((query) => {
  setSearchQuery(query);
}, []);

const handleRefresh = useCallback(() => {
  syncEmails();
  loadFolderEmails(currentFolder);
}, [currentFolder, syncEmails]);

return (
  <EmailProvider value={{
    emails,
    folders,
    selectedEmails,
    selectedEmail,
    currentFolder,
    searchQuery,
    viewSettings,
    syncStatus,
    isOnline,
    onEmailSelect: handleEmailSelect,
    onEmailsSelection: handleEmailsSelection,
    onFolderChange: setCurrentFolder,
    onSearch: handleSearch,
    onArchive: archiveEmails,
    onDelete: deleteEmails,
    onMarkAsRead: markAsRead,
    onViewSettingsChange: setViewSettings
  }}>
    <div className="email-client-app">
      <header className="email-header">
        <SearchBar onSearch={handleSearch} />
        <ActionToolbar
```

```
            selectedCount={selectedEmails.length}
            onRefresh={handleRefresh}
            onCompose={() => setIsComposing(true)}
            onArchive={() => archiveEmails()}
            onDelete={() => deleteEmails()}
          />
        </header>

        <main className="email-main">
          <ThreeColumnLayout />
        </main>

        {isComposing && (
          <ComposeModal
            onClose={() => setIsComposing(false)}
            onSend={sendEmail}
          />
        )}
      </div>
    </EmailProvider>
  );
};

export default EmailClientApp;
```

ThreeColumnLayout.jsx

```
import React, { useContext } from 'react';
import { EmailContext } from './EmailContext';
import FolderSidebar from './FolderSidebar';
import EmailList from './EmailList';
import EmailPreview from './EmailPreview';

const ThreeColumnLayout = () => {
  const { viewSettings, selectedEmail } = useContext(EmailContext);

  return (
    <div className="three-column-layout">
      <div className="folder-sidebar-column">
        <FolderSidebar />
      </div>

      <div className="email-list-column">
        <EmailList />
      </div>
      {viewSettings.showPreview && (
        <div className="email-preview-column">
          {selectedEmail ? (
            <EmailPreview email={selectedEmail} />
          ) : (
            <div className="no-email-selected">
              <div className="placeholder-content">
                <h3>No email selected</h3>
                <p>Choose an email from the list to preview it here</p>
              </div>
            </div>
          )}
        </div>
      )}
    </div>
  );
};

export default ThreeColumnLayout;
```

EmailList.jsx

```
import React, { useContext, useMemo } from 'react';
import { FixedSizeList as VirtualList } from 'react-window';
import { EmailContext } from './EmailContext';
import EmailListItem from './EmailListItem';

const EmailList = () => {
  const {
    emails,
    selectedEmails,
    searchQuery,
    viewSettings,
    onEmailsSelection
  } = useContext(EmailContext);

  const filteredEmails = useMemo(() => {
    if (!searchQuery) return emails;

    return emails.filter(email =>
      email.subject.toLowerCase().includes(searchQuery.toLowerCase()) ||
      email.sender.toLowerCase().includes(searchQuery.toLowerCase()) ||
      email.body.toLowerCase().includes(searchQuery.toLowerCase())
    );
  }, [emails, searchQuery]);
```

```
const groupedEmails = useMemo(() => {
  if (!viewSettings.groupByThread) return filteredEmails;

  const threads = new Map();
  filteredEmails.forEach(email => {
    const threadId = email.threadId || email.id;
    if (!threads.has(threadId)) {
      threads.set(threadId, []);
    }
    threads.get(threadId).push(email);
  });

  return Array.from(threads.values()).map(thread => thread[0]);
}, [filteredEmails, viewSettings.groupByThread]);

const handleSelectAll = () => {
  const allIds = groupedEmails.map(email => email.id);
  const isAllSelected = allIds.every(id => selectedEmails.includes(id));

  if (isAllSelected) {
    onEmailsSelection([]);
  } else {
    onEmailsSelection(allIds);
  }
};

const handleItemSelection = (emailId, isSelected) => {
  if (isSelected) {
    onEmailsSelection([...selectedEmails, emailId]);
  } else {
    onEmailsSelection(selectedEmails.filter(id => id !== emailId));
  }
};

const itemHeight = viewSettings.density === 'compact' ? 60 :
                 viewSettings.density === 'comfortable' ? 80 : 100;

const ItemRenderer = ({ index, style }) => (
  <div style={style}>
    <EmailListItem
      email={groupedEmails[index]}
      isSelected={selectedEmails.includes(groupedEmails[index].id)}
      onSelect={handleItemSelection}
      density={viewSettings.density}
    />
  </div>
);

return (
  <div className="email-list">
    <div className="list-header">
      <div className="select-all-section">
        <input
          type="checkbox"
          checked={groupedEmails.length > 0 &&
                   groupedEmails.every(email => selectedEmails.includes(email.id))}
          onChange={handleSelectAll}
          className="select-all-checkbox"
        />
        <span className="email-count">
          {groupedEmails.length} emails
        </span>
      </div>

      <div className="list-actions">
        <button className="sort-button">
          Sort by Date ↓
        </button>
      </div>
    </div>

    <div className="list-content">
      {groupedEmails.length === 0 ? (
        <div className="empty-list">
          <div className="empty-state">
            <h3>No emails found</h3>
            <p>Your {searchQuery ? 'search' : 'folder'} is empty</p>
          </div>
        </div>
      ) : (
        <VirtualList
          height={600}
          itemCount={groupedEmails.length}
          itemSize={itemHeight}
          overscanCount={5}
        >
          {ItemRenderer}
        </VirtualList>
      )}
    </div>
  </div>
);
};

export default EmailList;
```

EmailListItem.jsx

```
import React, { useContext } from 'react';
import { EmailContext } from './EmailContext';

const EmailListItem = ({ email, isSelected, onSelect, density }) => {
  const { onEmailSelect, selectedEmail } = useContext(EmailContext);

  const handleClick = () => {
    onEmailSelect(email.id);
  };

  const handleCheckboxChange = (e) => {
    e.stopPropagation();
    onSelect(email.id, e.target.checked);
  };

  const formatDate = (dateString) => {
    const date = new Date(dateString);
    const now = new Date();
    const isToday = date.toDateString() === now.toDateString();

    if (isToday) {
      return date.toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' });
    } else {
      return date.toLocaleDateString();
    }
  };

  const truncateText = (text, maxLength) => {
    return text.length > maxLength ? text.slice(0, maxLength) + '...' : text;
  };

  const isActive = selectedEmail?.id === email.id;

  return (
    <div
      className={`email-list-item ${density} ${isActive ? 'active' : ''} ${!email.isRead
      onClick={handleClick}
    >
```

```
<div className="item-checkbox">
  <input
    type="checkbox"
    checked={isSelected}
    onChange={handleCheckboxChange}
    onClick={(e) => e.stopPropagation()}
  />
</div>

<div className="item-content">
  <div className="item-header">
    <div className="sender-info">
      <span className="sender-name">
        {email.sender}
      </span>
      {!email.isRead && <span className="unread-indicator" />}
    </div>

    <div className="email-meta">
      <span className="date">{formatDate(email.date)}</span>
      {email.attachments?.length > 0 && (
        <span className="attachment-icon"> </span>
      )}
      {email.priority === 'high' && (
        <span className="priority-high"> </span>
      )}
    </div>
  </div>

  <div className="item-body">
    <div className="subject">
      {truncateText(email.subject, 60)}
    </div>

    {density !== 'compact' && (
      <div className="preview">
        {truncateText(email.body || '', 100)}
      </div>
    )}
  </div>

  {email.labels && email.labels.length > 0 && (
    <div className="item-labels">
      {email.labels.slice(0, 3).map(label => (
        <span key={label.id} className={`label-chip ${label.color}`}>
```

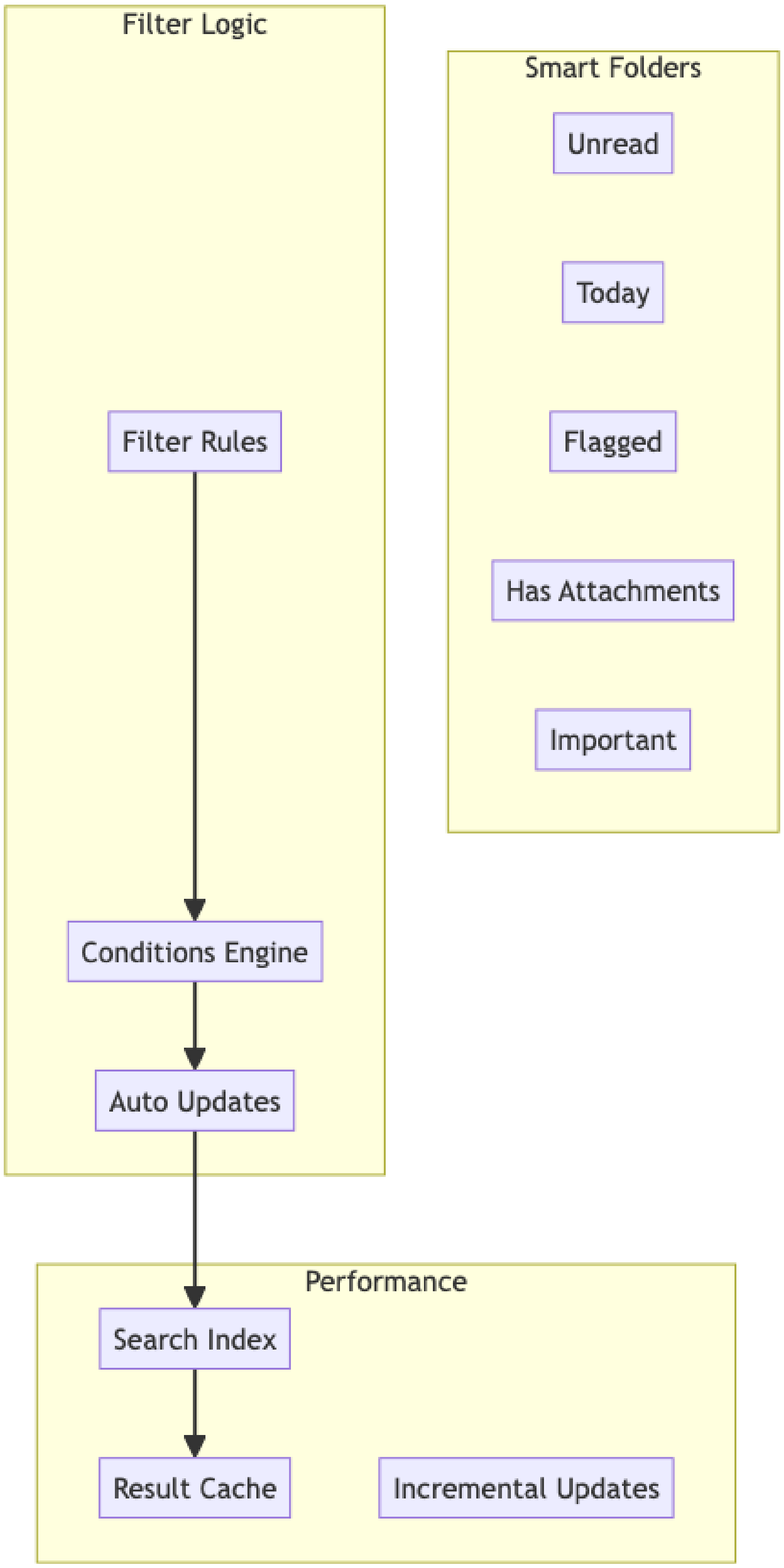
```
                {label.name}
              </span>
            ))}
          </div>
        )}
      </div>
    </div>
  );
};

export default EmailListItem;
```

Advanced Features

☐ Back to Top

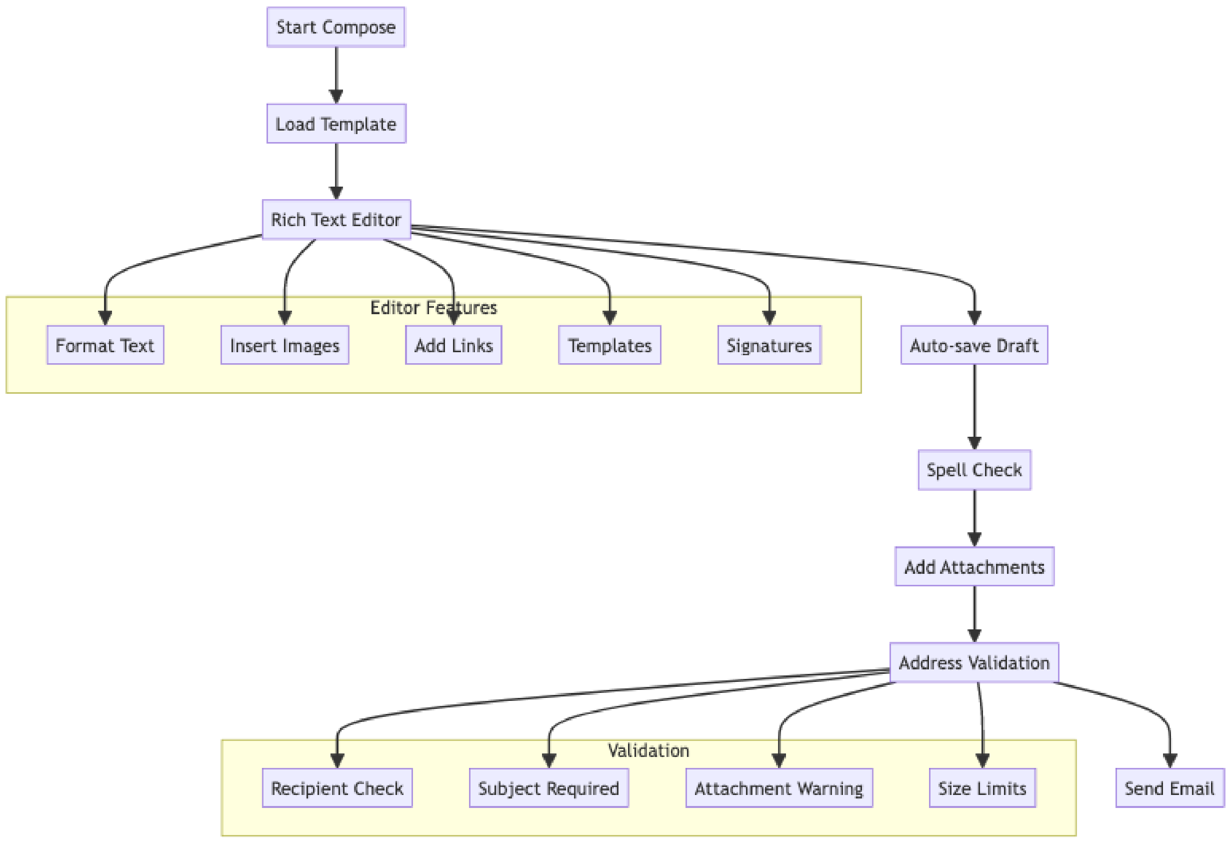
Smart Folder System

☐ Back to Top


Filter Logic
Filter Rules
Conditions Engine
Auto Updates
Smart Folders
Unread
Today
Flagged
Has Attachments
Important
Performance
Search Index
Result Cache
Incremental Updates

Email Composition Features

☐ Back to Top



TypeScript Interfaces & Component Props

☐ Back to Top

Core Data Interfaces

```
interface Email {
  id: string;
  messageId: string;
  threadId?: string;
  subject: string;
  from: EmailAddress;
  to: EmailAddress[];
  cc?: EmailAddress[];
  bcc?: EmailAddress[];
  date: Date;
  body: EmailBody;
  attachments: Attachment[];
  labels: Label[];
  isRead: boolean;
  isStarred: boolean;
  isDraft: boolean;
  priority: 'low' | 'normal' | 'high';
}

interface EmailAddress {
  email: string;
  name?: string;
}

interface EmailBody {
  text?: string;
  html?: string;
  preview: string;
  hasExternalImages: boolean;
}

interface Folder {
  id: string;
  name: string;
  type: 'inbox' | 'sent' | 'drafts' | 'spam' | 'trash' | 'custom';
  parentId?: string;
  unreadCount: number;
  totalCount: number;
  children?: Folder[];
}

interface Label {
  id: string;
  name: string;
  color: string;
  isSystem: boolean;
}
```

Component Props Interfaces

```
interface EmailListProps {
  emails: Email[];
  selectedEmails: string[];
  currentFolder: string;
  onEmailSelect: (emailId: string) => void;
  onEmailsSelection: (emailIds: string[]) => void;
  onEmailAction: (action: string, emailIds: string[]) => void;
  density: 'compact' | 'comfortable' | 'spacious';
  groupByThread: boolean;
}

interface EmailViewerProps {
  email: Email;
  onReply: () => void;
  onReplyAll: () => void;
  onForward: () => void;
  onDelete: () => void;
  onArchive: () => void;
  onMarkAsRead: (isRead: boolean) => void;
  showExternalImages: boolean;
}

interface FolderSidebarProps {
  folders: Folder[];
  selectedFolder: string;
  onFolderSelect: (folderId: string) => void;
  onFolderCreate: (name: string, parentId?: string) => void;
  onFolderRename: (folderId: string, newName: string) => void;
  collapsedFolders: string[];
}

interface ComposeWindowProps {
  initialTo?: string[];
  initialSubject?: string;
  initialBody?: string;
  draftId?: string;
  onSend: (email: ComposeData) => void;
  onSaveDraft: (email: ComposeData) => void;
  onClose: () => void;
  attachments: File[];
}
```

API Reference

☐ Back to Top

Email Management

- `GET /api/emails` - Get emails with filtering, pagination, and search
- `GET /api/emails/:id` - Fetch specific email with full content and attachments
- `PUT /api/emails/:id/read` - Mark email as read or unread
- `PUT /api/emails/:id/star` - Star or unstar email for quick access
- `DELETE /api/emails/:id` - Move email to trash or permanently delete

Folder Operations

- `GET /api/folders` - Get folder hierarchy with unread counts
- `POST /api/folders` - Create new custom folder with permissions
- `PUT /api/folders/:id` - Rename folder or update folder settings
- `DELETE /api/folders/:id` - Delete custom folder (move emails to parent)
- `POST /api/emails/move` - Move emails between folders in batch

Email Composition

- `POST /api/emails/send` - Send new email with attachments and scheduling
- `POST /api/emails/draft` - Save email as draft with auto-save functionality
- `GET /api/emails/drafts` - Get saved drafts with pagination
- `PUT /api/emails/drafts/:id` - Update existing draft content
- `POST /api/emails/schedule` - Schedule email for future delivery

Search & Filtering

- `GET /api/search/emails` - Advanced email search with multiple criteria
- `GET /api/search/autocomplete` - Get search suggestions and autocomplete
- `POST /api/search/save` - Save search query for quick access
- `GET /api/filters` - Get email filtering rules and conditions
- `POST /api/filters` - Create automatic email filtering rule

Attachments & Media

- `POST /api/attachments/upload` - Upload file attachments with virus scanning
- `GET /api/attachments/:id` - Download attachment with access control
- `GET /api/attachments/:id/preview` - Get attachment preview or thumbnail
- `DELETE /api/attachments/:id` - Remove attachment from draft email
- `POST /api/attachments/scan` - Scan attachment for malware before sending

Labels & Organization

- `GET /api/labels` - Get all labels with usage statistics
- `POST /api/labels` - Create new custom label with color coding
- `PUT /api/labels/:id` - Update label name, color, or visibility
- `DELETE /api/labels/:id` - Delete label and remove from all emails
- `POST /api/emails/:id/labels` - Add or remove labels from email

Threading & Conversations

- `GET /api/threads/:id` - Get complete email thread with all messages
- `PUT /api/threads/:id/read` - Mark entire thread as read or unread
- `POST /api/threads/:id/mute` - Mute thread to reduce notifications
- `DELETE /api/threads/:id` - Delete entire conversation thread
- `GET /api/threads/:id/participants` - Get all participants in thread

Performance Optimizations

☐ Back to Top

Memory Management

☐ Back to Top

Email Data Lifecycle:

```
EmailCache = {
  active: LRU<EmailId, Email>,
  headers: Map<EmailId, EmailHeaders>,
  bodies: WeakMap<EmailId, EmailBody>,
  attachments: Map<AttachmentId, Blob>
}
```

Optimization Strategies: - Lazy load email bodies and attachments - Implement LRU cache for recently accessed emails - Use weak references for large binary data - Compress cached email content - Unload off-screen email data

Network Optimization

☐ Back to Top

Batch Operations:

```
BatchManager = {
  markAsRead: EmailId[],
  markAsUnread: EmailId[],
  delete: EmailId[],
  move: { emailIds: EmailId[], folderId: string }[],
  addLabel: { emailIds: EmailId[], labelId: string }[]
```

}

Connection Management: - Pool IMAP connections efficiently - Implement exponential backoff for retries - Use compression for large data transfers - Prioritize user-initiated actions - Cache frequently accessed folder structures

Search Performance

□ Back to Top

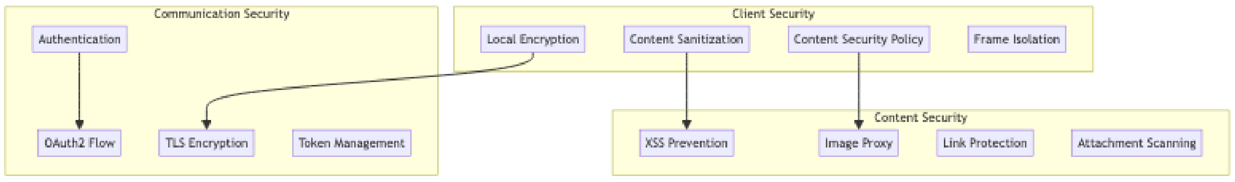
Indexing Strategy: - Build inverted index for email content - Update index incrementally for new emails - Use bloom filters for negative lookups - Implement faceted search for quick filters - Store index in IndexedDB for persistence

Security Considerations

□ Back to Top

Data Protection

□ Back to Top



Email Content Security

□ Back to Top

HTML Sanitization: - Strip dangerous HTML tags and attributes - Block external resource loading - Implement image proxy for tracking protection - Sanitize CSS styles and inline scripts - Use Content Security Policy headers

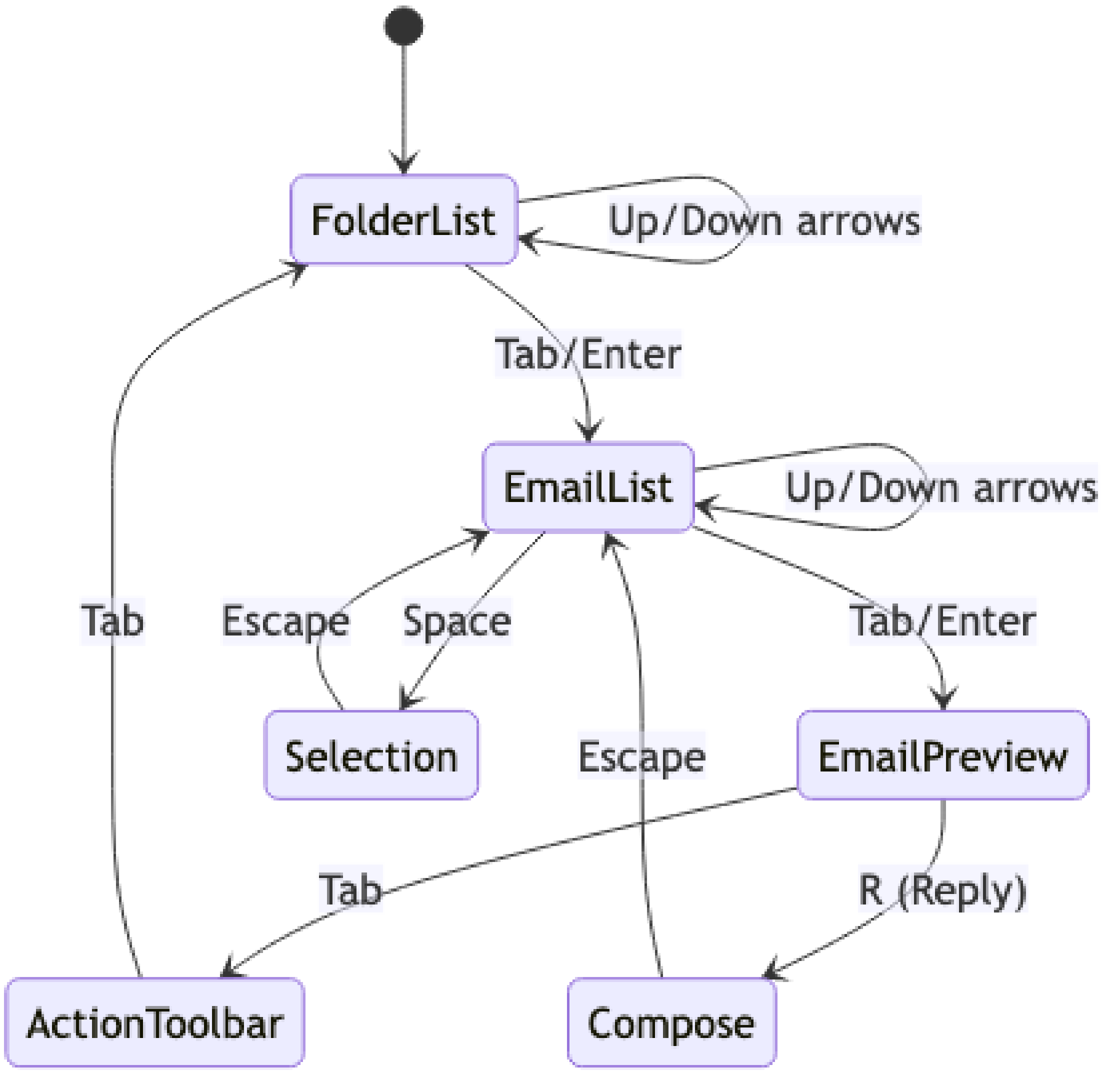
Attachment Security: - Scan attachments for malware - Block executable file types - Implement download warnings - Use sandbox for attachment preview - Virus scanning integration

Accessibility Implementation

 Back to Top

Keyboard Navigation

 Back to Top



Accessibility Features: - Full keyboard navigation support - Screen reader announcements for email content - High contrast mode support - Focus indicators for all interactive elements - ARIA labels for complex UI components

Screen Reader Support

□ Back to Top

Email Announcement Pattern:

```
"Email 1 of 50, from John Doe, subject: Meeting Tomorrow,
received 2 hours ago, unread, has attachment,
press Enter to open, Space to select"
```

Navigation Landmarks: - Main navigation region for folders - Search region for email search - Content region for email list and preview - Complementary region for quick actions

Testing Strategy

□ Back to Top

Unit Testing Focus Areas

□ Back to Top

Core Algorithm Testing: - Virtual list rendering accuracy - Email threading correctness - Search ranking algorithms - Synchronization logic

Component Testing: - Email list interactions - Drag and drop functionality - Folder navigation - Compose window features

Integration Testing

□ Back to Top

Email Flow Testing: - Complete email workflow from compose to send - Multi-folder synchronization - Search across different email sources - Label and filter application

Performance Testing: - Large email list rendering - Search performance with large datasets - Memory usage patterns - Network efficiency

End-to-End Testing

□ Back to Top

User Workflow Testing: - Email management scenarios - Cross-device synchronization - Offline functionality - Email composition and sending

Trade-offs and Considerations

□ Back to Top

Performance vs Features

□ Back to Top

- **Real-time sync**: Battery usage vs instant updates
- **Rich preview**: Loading time vs email content richness
- **Search capability**: Index size vs search speed
- **Offline support**: Storage usage vs offline functionality

Security vs Usability

□ Back to Top

- **Content blocking**: Security vs email display fidelity
- **Attachment handling**: Safety vs user convenience
- **External images**: Privacy vs email formatting
- **Link protection**: Security vs click-through experience

Scalability Considerations

□ Back to Top

- **Email volume**: Performance with large mailboxes
- **Attachment size**: Storage and transfer limits
- **Search index**: Index size vs search capability
- **Synchronization**: Bandwidth usage vs sync frequency

This email client system provides a comprehensive foundation for modern email management with advanced features like intelligent threading, robust search, and efficient synchronization while maintaining high performance and accessibility standards.