

To implement this modular email management system, we will break the project down into several stages to ensure it's both functional and scalable. I'll outline the high-level steps with some example code snippets and suggestions for each core feature.

## 1. Account Registration and Authentication

### a. Registration and Login

- **API Integration:** You would use mail.tm's API for mailbox creation. Since you'll likely be handling multiple accounts, consider securely storing access tokens.

Example Backend (Node.js/Express):

js



```
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const User = require('./models/User'); // Assume a MongoDB schema for Users
const router = express.Router();

// User Registration Route
router.post('/register', async (req, res) => {
  const { username, email, password } = req.body;

  const hashedPassword = await bcrypt.hash(password, 10);

  const newUser = new User({
    username,
    email,
    password: hashedPassword,
  });

  await newUser.save();
  res.status(201).send("User created!");
});

// User Login Route
router.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });

  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).send("Invalid credentials");
  }

  const token = jwt.sign({ userId: user._id }, 'your_jwt_secret');
  res.status(200).send({ token });
});
```

### b. JWT Authentication

For handling user sessions, JWT (JSON Web Tokens) is a common approach. It allows the front-end to securely make authenticated API requests without needing to maintain a traditional session.

## 2. Real-Time Notifications

You can use **WebSockets** or **Server-Sent Events (SSE)** for real-time notifications when a new email arrives.

### Example using WebSockets (Socket.io):

- **Backend:** Set up a WebSocket server to push notifications to the front end when an email is received.

js



```
const http = require('http');
const socketIo = require('socket.io');

const server = http.createServer();
const io = socketIo(server);

io.on('connection', (socket) => {
  console.log('User connected');

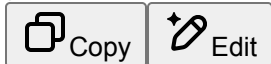
  // Example: Notify client when an email arrives
  socket.emit('new-email', { subject: 'Welcome!', sender: 'example@mail.com' });

  socket.on('disconnect', () => {
    console.log('User disconnected');
  });
});

server.listen(3001, () => {
  console.log('Server listening on port 3001');
});
```

- **Frontend:** On the front-end, you'd listen for these updates using Socket.io.

js



```
const socket = io('http://localhost:3001');

socket.on('new-email', (emailData) => {
  console.log('New email received:', emailData);
  // Display notification or update the UI
});
```

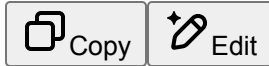
## 3. Email Storage and Management (Beyond 20 Emails)

Since mail.tm has a default email retention limit, you can store emails locally or in a database (e.g., MongoDB or PostgreSQL).

### Example using Local Storage for Email Data (Frontend):

1. **Email Fetching and Local Storage:** Fetch emails via mail.tm API, and store them in the browser's Local Storage or a database.

js



```
// Function to fetch emails from mail.tm API and store in Local Storage
async function fetchEmails() {
  const response = await fetch('https://api.mail.tm/messages', {
    method: 'GET',
    headers: { 'Authorization': 'Bearer ' + userToken }
  });
  const emails = await response.json();
  localStorage.setItem('emails', JSON.stringify(emails));
  renderEmails(emails);
}

// Render Emails
function renderEmails(emails) {
  const emailContainer = document.getElementById('email-list');
  emailContainer.innerHTML = '';
  emails.forEach(email => {
    const emailItem = document.createElement('div');
    emailItem.innerText = email.subject;
    emailContainer.appendChild(emailItem);
  });
}
```

2. **Email Deletion:** You can provide functionality to delete emails locally.

js



```
function deleteEmail(emailId) {
  let emails = JSON.parse(localStorage.getItem('emails'));
  emails = emails.filter(email => email.id !== emailId);
  localStorage.setItem('emails', JSON.stringify(emails));
  renderEmails(emails);
}
```

## 4. Email Filtering

### Custom Filters (Keyword-Based Categorization)

You can allow users to create custom filters based on keywords (e.g., filtering by "adjustments" in subject).

- **Frontend:** Allow users to input their filters and apply them.

js



```
function applyFilter(keyword) {
  let emails = JSON.parse(localStorage.getItem('emails'));
  let filteredEmails = emails.filter(email => email.subject.includes(keyword));
}
```

```
renderEmails(filteredEmails);  
}
```

- **Backend:** You can also implement this filtering logic on the server-side to enhance performance for large datasets.

## 5. Multi-Account Management

For multi-account support, you can store user tokens and email information locally or in your backend (preferably with token expiry for added security).

### Example of Switching Accounts (Frontend):

- **UI:** You can create a dropdown or sidebar to allow users to switch between accounts. For each account, a unique token should be saved.

js



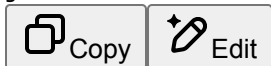
```
// Store user account info in localStorage  
function switchAccount(newAccountToken) {  
  localStorage.setItem('userToken', newAccountToken);  
  fetchEmails();  
}
```

## 6. UI/UX Implementation (React Example)

For the front-end, React is a great option for a modern, responsive UI.

- **Dark Mode Toggle:**
  - You can use React's state to toggle dark mode, storing the preference in localStorage.

js



```
const [darkMode, setDarkMode] = useState(localStorage.getItem('theme') === 'dark');  
  
useEffect(() => {  
  document.body.className = darkMode ? 'dark-mode' : '';  
  localStorage.setItem('theme', darkMode ? 'dark' : 'light');  
}, [darkMode]);  
  
return (  
  <button onClick={() => setDarkMode(!darkMode)}>Toggle Dark Mode</button>  
);
```

- **Email List Display:** You can display emails using simple cards or table structures, and integrate sorting and pagination as necessary.

## 7. Security Considerations

- **JWT Tokens:** Use JWT for authentication and store them in HttpOnly cookies for added security.

- **API Rate Limiting:** Use middleware like `express-rate-limit` in your API server to protect against abuse.

## 8. Exporting Emails (EML/MBOX)

Allow users to export emails in EML or MBOX formats for backup or external use.

js



```
// Example: Convert email to EML format
function convertToEML(email) {
  const emlContent = `
    From: ${email.sender}
    To: ${email.receiver}
    Subject: ${email.subject}
    Date: ${email.date}

    ${email.body}
  `;
  return emlContent;
}
```

## 9. Testing & Debugging

- **UI/UX:** Ensure that the dark mode and account switching persist across sessions.
- **Backend:** Test the authentication flow thoroughly (JWT expiry, token validation).
- **Real-time notifications:** Test WebSocket connection stability across different network conditions.

---

### Next Steps:

1. **Set up the initial backend with JWT authentication.**
2. **Integrate mail.tm API for fetching, storing, and managing emails.**
3. **Build frontend UI using React (or preferred JS framework),** including components for displaying emails, dark mode, and account management.
4. **Implement real-time notifications via WebSocket/SSE.**
5. **Develop email filtering and categorization features.**
6. **Ensure that email data persists beyond the 20-email limit (local storage or database).**

This is a high-level guide, but it should give you a solid foundation to start building out the app in stages. Let me know if you want to dive deeper into any part of the process!