Below is a step-by-step outline for designing and implementing your dashboard (portal) to handle device alerts from Outlook emails, parse them, store them in a database, and present them in a user-friendly interface for your ONCALL team. The solution emphasizes modularity, scalability, and a pleasant, “Intel-style” user experience.

**1. High-Level Architecture**

1. **Alert Retrieval Layer**
   * **Microsoft Graph API** (or alternative methods like Exchange Web Services, or an IMAP-based approach) for fetching emails from the Outlook mailbox.
   * A **Background Service** or **Cron Job** that periodically polls the mailbox.
2. **Alert Parsing & Normalization**
   * A **Parsing Service** that transforms different email formats into a standardized alert structure.
   * This logic can be done within the same service that retrieves the emails or as a separate microservice.
3. **Data Storage**
   * A **Relational Database** (SQL—e.g., PostgreSQL, MySQL, or MSSQL) to store:
     + Alerts (raw data, parsed content, timestamps, etc.)
     + Device information
     + Alert resolution status, comments, and ONCALL user details
     + Historical data for each device
4. **API / Backend Layer**
   * A **RESTful API** (or GraphQL, if you prefer) that the frontend will consume.
   * Could be built with frameworks like **Node.js (Express/NestJS)**, **Python (Flask/Django/FastAPI)**, or **.NET**—whatever best fits your environment.
5. **Frontend (Web Portal)**
   * A **Single-Page Application (SPA)** using React, Angular, or Vue.
   * Secure user login/role management (with an ONCALL role).
   * Dashboard pages to display active alerts, device details, and alert history.
   * A consistent design system (“Intel style”: clean, modern, minimal, professional) for a smooth user experience.
6. **Authentication & Authorization**
   * Use OAuth2 / OpenID Connect (e.g., via Microsoft identity platform or a third-party identity provider).
   * Restrict certain actions (e.g., resolving alerts) to ONCALL users or admins.

**2. Data Flow Diagram**

+-------------------+

| Outlook Mail |

| (Mailbox) |

+---------+---------+

|

v

[1] Background Service

(Fetch emails via Graph)

|

v

Parsing/Normalization

(Extract device info, alert

severity, date/time, etc.)

|

v

SQL Database

(Alerts, Devices, History)

^

|

+-----------------------------------+

| [2] Backend API (REST or GraphQL)|

| ( CRUD for alerts & devices ) |

+------------+----------------------+

|

v

[3] Frontend Portal

(React / Angular / Vue SPA)

1. **Retrieve**: The background service fetches emails periodically from the Outlook mailbox using Graph API.
2. **Parse**: The service parses the emails, identifies alert details, and saves or updates records in the SQL database.
3. **Display**: The frontend portal calls the backend API to display alerts, allow alert resolution, and show device history.

**3. Detailed Implementation Steps**

**3.1. Alert Retrieval using Microsoft Graph**

1. **Register an Azure App**:
   * Create an application in Azure Active Directory to authorize access to Microsoft 365 mailbox data.
   * Configure the required permissions (Mail.Read, Mail.ReadWrite, etc.).
   * Generate client secret or certificate.
2. **Service / Microservice for Email Fetching**:
   * Use a library/SDK for Microsoft Graph (depending on language):
     + **JavaScript/TypeScript**: @microsoft/microsoft-graph-client
     + **Python**: azure-identity + msgraph-core
     + **.NET**: Microsoft.Graph NuGet package
   * Implement polling logic (e.g., run every X minutes, or subscribe to push notifications if you prefer real-time).
   * Filter the mailbox for unread or new alert emails (using search queries or message rules).
3. **Cron/Job Scheduling**:
   * Host this service on a server or container that periodically runs the fetch operation.
   * Alternatively, use Azure Functions or AWS Lambda with a scheduled trigger.

**3.2. Parsing & Normalization of Alerts**

1. **Define a Standardized Alert Schema** in your code (e.g., Alert class or data model) that includes:
   * alert\_id
   * device\_id
   * device\_name
   * alert\_type or category (e.g., “Temperature Alert”, “Network Down”, etc.)
   * severity (low, medium, high)
   * timestamp (when alert was triggered)
   * message or description
   * status (new, acknowledged, resolved)
   * resolution\_comment
   * resolved\_by
   * resolved\_at
2. **Parsing Logic**:
   * Since each device might generate different email structures, create **parsing templates** or **regular expressions** to extract data.
   * For example, if the subject line contains [AlertType] [DeviceID] [Timestamp], you can parse accordingly.
3. **Error Handling**:
   * Log malformed alerts or emails that don’t match known patterns.
   * Possibly store them in a “quarantine” table to be handled manually or updated as you add new parsing rules.

**3.3. Storing Alerts in a SQL Database**

1. **Database Schema** (simplified):
   * **Devices** table:
   * CREATE TABLE devices (
   * id SERIAL PRIMARY KEY,
   * device\_name VARCHAR(100),
   * ... (any extra metadata)
   * );
   * **Alerts** table:
   * CREATE TABLE alerts (
   * id SERIAL PRIMARY KEY,
   * device\_id INT REFERENCES devices(id),
   * alert\_type VARCHAR(100),
   * severity VARCHAR(20),
   * timestamp TIMESTAMP,
   * message TEXT,
   * status VARCHAR(20) DEFAULT 'new',
   * resolution\_comment TEXT,
   * resolved\_by VARCHAR(50),
   * resolved\_at TIMESTAMP,
   * created\_at TIMESTAMP DEFAULT NOW()
   * );
   * Optionally, a **Users** table to manage ONCALL and Admin roles.
2. **ORM Integration**:
   * Using an ORM (e.g., **SQLAlchemy** for Python, **Entity Framework** for .NET, or **TypeORM** for Node) can simplify CRUD operations and migrations.

**3.4. Backend API (REST or GraphQL)**

1. **Endpoints**:
   * GET /alerts – Retrieve a list of current alerts (filter by status).
   * GET /alerts/:id – Retrieve details for a specific alert.
   * POST /alerts/resolve/:id – Resolve a specific alert (with user comment).
   * GET /devices/:id/alerts – Get alert history for a specific device.
   * GET /oncall – (Optional) Return the current ONCALL user or schedule.
2. **Implementation**:
   * **Python/Flask or FastAPI**: Define routes, use the ORM for DB operations.
   * **Node/Express or NestJS**: Define controllers/services; use TypeORM or Sequelize.
   * **.NET Web API**: Controllers + Entity Framework.
3. **Security**:
   * Implement **JWT** or token-based authentication (or integrate with Azure AD for corporate SSO).
   * Restrict certain endpoints to ONCALL or Admin roles (e.g., resolving alerts).

**3.5. Frontend (UI/UX)**

1. **Framework & Libraries**:
   * **React** with a UI library like **Material-UI (MUI)** or **Chakra UI** for a modern, “Intel-style” look.
   * **Angular** with Angular Material.
   * **Vue** with Vuetify.
2. **Design & Layout**:
   * **Dashboard Page**:
     + Show active alerts in a table or card layout.
     + Include sorting/filtering by severity, device, date, etc.
   * **Alert Detail Modal/Page**:
     + When an alert is clicked, open a detailed view.
     + Include a “Resolve” button (if user has permission).
   * **Device History Page**:
     + List all past alerts for a device, their status, and resolution comments.
   * **Navigation**:
     + A top-level or side-level menu to switch between “Active Alerts,” “All Devices,” or “ONCALL tools.”
3. **Styling Considerations**:
   * Maintain a consistent color scheme (light background, subtle accent colors).
   * Keep surfaces “soft” with minimal borders and subtle shadows.
   * Use consistent typography and spacing.
   * Add corporate or brand logos if required.
   * “Intel style” often implies a professional, enterprise theme—focus on clean lines, minimal clutter, and clarity of data presentation.

**3.6. ONCALL Workflow**

1. **Identifying ONCALL Users**:
   * Integrate with your scheduling system or a simple “ONCALL” flag in the Users table.
   * The system can display or highlight who is currently on call.
2. **Alert Acknowledgment & Resolution**:
   * A user with ONCALL role sees a “Resolve” button on each alert.
   * They can add a resolution comment.
   * The API updates the alert’s status to resolved, records resolved\_by and resolved\_at.
3. **Device History & Commenting**:
   * An alerts feed for each device in a separate tab or modal.
   * Show all previously resolved alerts and their comments for context.

**4. Additional Implementation Considerations**

1. **Scaling & Performance**
   * If the volume of emails is high, consider using a message queue (RabbitMQ, Azure Service Bus, or AWS SQS) for asynchronous alert parsing.
   * Cache frequently accessed data (e.g., device details) in Redis if needed.
2. **Error Handling & Logging**
   * Implement robust logging (e.g., using Winston (Node), Serilog (.NET), or Python’s logging) to track email parsing failures or DB write errors.
   * Store logs in a centralized system (Splunk, ELK stack, Azure Monitor, etc.).
3. **Notifications**
   * Optionally add a system to push notifications (e.g., Slack, Teams, or email) when new critical alerts arrive or remain unresolved for too long.
4. **DevOps & Deployment**
   * Use Docker containers for consistent runtime environments.
   * Consider CI/CD pipelines (GitHub Actions, Azure DevOps, Jenkins) to automate testing and deployment.
   * Deploy the backend (API) and the parsing service to your chosen cloud environment (Azure App Service, AWS ECS/EKS, etc.).
   * Serve the frontend via static hosting (Azure Blob Storage, AWS S3, or Netlify) if it’s an SPA.
5. **Security & Compliance**
   * Ensure data in transit is encrypted (HTTPS/TLS).
   * Secure database credentials and secrets in a vault (Azure Key Vault, AWS Secrets Manager).
   * Follow least-privilege principles when granting mailbox and user permissions.

**5. First Steps to Begin the Project**

1. **Set Up the Repository & Project Structure**
   * Create a mono-repo or separate repos for “backend,” “parsing service,” and “frontend.”
2. **Prototype the Parsing**
   * Start by fetching sample alert emails from the mailbox using the Graph API in a local script.
   * Write a basic parser for a couple of known alert formats.
3. **Design the Database Schema**
   * Draft the tables for devices, alerts, users, and any additional references you need.
   * Create migrations if using an ORM.
4. **Implement the Backend CRUD**
   * Build minimal endpoints for listing alerts and inserting them.
5. **Proof-of-Concept Frontend**
   * Set up React/Angular with a minimal UI showing a table of alerts.
   * Connect it to your backend endpoints.
6. **Refine Workflow & UI**
   * Add the “Resolve Alert” functionality and device history view.
   * Implement user authentication and role-based access.
7. **Iterate & Enhance**
   * Once the core alert functionality is working, add visual polish to the UI.
   * Incorporate more complex parsing rules and error handling.

**Conclusion**

By following this architecture, you’ll have a structured, modular approach to receiving alerts from Outlook, parsing them, storing them in a SQL database, and displaying them through a user-friendly “Intel-style” portal. The key is to **keep the parsing rules flexible**, **maintain a clean schema**, and **design a frontend that’s both visually appealing and easy to navigate** for ONCALL users.

With this foundation, you can scale as more devices or alert types come online, and you can continually refine the user experience (e.g., advanced filtering, notification enhancements, integration with chat tools, etc.) to ensure the ONCALL team handles device alerts efficiently and consistently. Good luck with your project!

-------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------

Below is a more in-depth, example-driven guide to help you implement your dashboard for device alerts. It includes:

1. **Concrete Examples** of each step (retrieving and parsing emails, storing data in a database, building a backend, and creating a UI).
2. **Expanded Database Schema** ideas to accommodate on-call schedules and additional metadata.
3. **VictorOps On-Call Scheduling** integration example (API-based).
4. **Microsoft Teams Notification** example (via Incoming Webhooks).

**1. Expanded Architecture Recap**

1. **Email Retrieval & Parsing**
   * Poll emails from Outlook using Microsoft Graph API.
   * Normalize/parse email data into a standardized Alert structure.
2. **Database Storage** (SQL)
   * Store **Alerts**, **Devices**, **Users**, **On-call Schedules**, and **Resolution History**.
3. **Backend Service**
   * Provides REST/GraphQL endpoints for the frontend.
   * Integrates with VictorOps (or other) for retrieving on-call schedules.
   * Triggers notifications (e.g., Teams) on important events.
4. **Frontend Portal**
   * A Single-Page Application (SPA) built with React/Angular/Vue.
   * Displays active alerts, alert details, and device alert histories.
   * Provides resolution workflow and an at-a-glance view of who’s on-call.

**2. Step-by-Step Examples**

**Step A: Retrieving Emails via Microsoft Graph (Python Example)**

import os

import time

import datetime

import requests

from msgraph.core import GraphClient

from azure.identity import ClientSecretCredential

TENANT\_ID = os.environ.get("AZURE\_TENANT\_ID")

CLIENT\_ID = os.environ.get("AZURE\_CLIENT\_ID")

CLIENT\_SECRET = os.environ.get("AZURE\_CLIENT\_SECRET")

MAILBOX\_USER = "your-alerts-mailbox@yourdomain.com"

def get\_graph\_client():

credential = ClientSecretCredential(

tenant\_id=TENANT\_ID,

client\_id=CLIENT\_ID,

client\_secret=CLIENT\_SECRET

)

client = GraphClient(credential=credential)

return client

def fetch\_alert\_emails():

client = get\_graph\_client()

# Example: search for unread emails in the Inbox.

endpoint = f"/users/{MAILBOX\_USER}/mailFolders/Inbox/messages?$filter=isRead eq false"

response = client.get(endpoint)

mail\_data = response.json()

if "value" in mail\_data:

for message in mail\_data["value"]:

subject = message.get("subject", "")

body = message.get("body", {}).get("content", "")

sender = message.get("from", {}).get("emailAddress", {}).get("address", "")

# Example parse step

parse\_and\_store\_alert(subject, body, sender, message["id"])

# Mark email as read to avoid re-processing

mark\_email\_as\_read(client, message["id"], MAILBOX\_USER)

def mark\_email\_as\_read(client, message\_id, mailbox):

endpoint = f"/users/{mailbox}/messages/{message\_id}"

data = {"isRead": True}

client.patch(endpoint, json=data)

def parse\_and\_store\_alert(subject, body, sender, message\_id):

"""

Here you define your parsing logic to extract the device ID, type of alert, severity, etc.

For demonstration, let's assume the subject is like:

"[ALERT] Device: {device\_id} Severity: {severity} Timestamp: {timestamp}"

"""

import re

pattern = r"\[ALERT\] Device:\s\*(\S+)\s+Severity:\s\*(\S+)\s+Timestamp:\s\*(\S+)"

match = re.search(pattern, subject)

if match:

device\_id = match.group(1)

severity = match.group(2)

timestamp\_str = match.group(3)

timestamp = datetime.datetime.strptime(timestamp\_str, "%Y-%m-%d\_%H:%M:%S")

# Construct the alert record

alert\_record = {

"device\_id": device\_id,

"severity": severity,

"alert\_timestamp": timestamp,

"raw\_message": body,

"subject": subject,

"sender": sender,

}

# Now store in DB

store\_alert\_in\_db(alert\_record)

def store\_alert\_in\_db(alert\_record):

# This is where you'd use your ORM or direct SQL to insert into the alerts table.

# Pseudo-code example (SQLAlchemy):

from models import db, Alert

new\_alert = Alert(

device\_id=alert\_record["device\_id"],

severity=alert\_record["severity"],

timestamp=alert\_record["alert\_timestamp"],

raw\_message=alert\_record["raw\_message"],

subject=alert\_record["subject"],

sender=alert\_record["sender"]

)

db.session.add(new\_alert)

db.session.commit()

if \_\_name\_\_ == "\_\_main\_\_":

# Could be scheduled with cron or run in a loop for a PoC

while True:

fetch\_alert\_emails()

time.sleep(60) # Wait a minute before checking again

In this example:

* We retrieve unread emails from the **Inbox** of a dedicated alerts mailbox.
* We parse the subject line using a **regular expression**.
* We store alert data in the database through an ORM (SQLAlchemy).
* We then mark the email as **read** to avoid reprocessing.

**Step B: Expanded Database Schema**

Here’s a more comprehensive schema that handles **users**, **devices**, **alerts**, **alert history**, and **on-call schedules**. You can combine certain fields if they’re not needed separately.

-- USERS TABLE

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

username VARCHAR(100) NOT NULL,

email VARCHAR(200) NOT NULL,

role VARCHAR(50) DEFAULT 'user', -- e.g. "admin", "oncall", "viewer"

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- DEVICES TABLE

CREATE TABLE IF NOT EXISTS devices (

id SERIAL PRIMARY KEY,

device\_name VARCHAR(100) NOT NULL,

location VARCHAR(100), -- optional location info

metadata JSONB, -- optional for extra device properties

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- ALERTS TABLE

CREATE TABLE IF NOT EXISTS alerts (

id SERIAL PRIMARY KEY,

device\_id INT NOT NULL REFERENCES devices(id) ON DELETE CASCADE,

severity VARCHAR(20), -- e.g. "Critical", "High", "Medium", "Low"

subject VARCHAR(200), -- or short title

raw\_message TEXT,

sender VARCHAR(200),

status VARCHAR(20) DEFAULT 'new', -- e.g. "new", "acknowledged", "resolved"

timestamp TIMESTAMP NOT NULL, -- when the alert happened (from the device/email)

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- ALERT RESOLUTION HISTORY / COMMENTS

CREATE TABLE IF NOT EXISTS alert\_resolutions (

id SERIAL PRIMARY KEY,

alert\_id INT NOT NULL REFERENCES alerts(id) ON DELETE CASCADE,

resolved\_by INT REFERENCES users(id) ON DELETE SET NULL,

resolution\_comment TEXT,

resolved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- ONCALL SCHEDULE TABLE (OPTIONAL, for storing oncall data if not using VictorOps directly)

CREATE TABLE IF NOT EXISTS oncall\_schedule (

id SERIAL PRIMARY KEY,

user\_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,

schedule\_start TIMESTAMP NOT NULL,

schedule\_end TIMESTAMP NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Key Points**:

* **devices** holds information about each device.
* **alerts** references a device\_id and stores the current status.
* **alert\_resolutions** can store multiple records for each alert if needed (though you can also store a single resolution in the alerts table if you only need one).
* **oncall\_schedule** is optional if you want to store on-call times locally. If you plan to rely on an external system like VictorOps, you might skip or only partially use this table.

**Step C: Integrating VictorOps (On-Call Scheduling)**

If your organization uses **VictorOps** (now known as Splunk On-Call) for on-call scheduling, you can fetch the current on-call user from the VictorOps API. Below is **example pseudo-code** in Python. (Adjust the actual endpoints and parameters according to the latest [Splunk On-Call API docs](https://docs.splunk.com/Documentation/VictorOps)):

import requests

VICTOROPS\_API\_ID = "YOUR\_VICTOROPS\_API\_ID"

VICTOROPS\_API\_KEY = "YOUR\_VICTOROPS\_API\_KEY"

def get\_current\_oncall():

"""

Example of fetching the current on-call user from a specific rotation or team.

The actual endpoint might differ. Check official Splunk On-Call (VictorOps) docs.

"""

headers = {

"X-VO-Api-Id": VICTOROPS\_API\_ID,

"X-VO-Api-Key": VICTOROPS\_API\_KEY,

"Accept": "application/json"

}

url = "https://api.victorops.com/api-public/v1/oncall/current"

response = requests.get(url, headers=headers)

data = response.json()

# Example structure:

# {

# "teamsOnCall": [

# {

# "team": {

# "slug": "team-slug",

# "name": "YourTeam"

# },

# "onCallNow": [

# {

# "userName": "oncall\_username",

# "email": "oncall@yourdomain.com"

# }

# ]

# }

# ]

# }

oncall\_info = []

for t in data.get("teamsOnCall", []):

team\_name = t["team"].get("name")

for person in t["onCallNow"]:

oncall\_user = person["userName"]

oncall\_email = person["email"]

oncall\_info.append({

"team": team\_name,

"username": oncall\_user,

"email": oncall\_email

})

return oncall\_info

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

current\_oncall = get\_current\_oncall()

print("Current on-call staff:", current\_oncall)

**What to do with this data**:

* You could sync the on-call user(s) to your **Users** table if they don’t exist.
* Display the on-call status in your dashboard header, so everyone knows who to contact.
* Combine it with your local oncall\_schedule if you also store offline schedules.

**Step D: Push Notifications to Microsoft Teams**

If you want to post a message to a Microsoft Teams channel (e.g., for critical alerts), you can use an **Incoming Webhook**. First, set up a webhook in the Teams channel (click “Connectors” → “Incoming Webhook”), then use the generated URL to post JSON payloads.

**Example Python Code**:

import requests

# Replace with your channel's incoming webhook URL

TEAMS\_WEBHOOK\_URL = "https://outlook.office.com/webhook/abc123/IncomingWebhook/..."

def send\_teams\_alert(alert\_data):

"""

alert\_data might contain:

- device\_id

- severity

- subject

- link to the dashboard alert page

"""

message\_title = f"New Alert from Device {alert\_data['device\_id']}"

severity = alert\_data['severity']

subject = alert\_data['subject']

dashboard\_link = alert\_data.get('dashboard\_link', '#')

payload = {

"@type": "MessageCard",

"@context": "https://schema.org/extensions",

"themeColor": "FF0000" if severity.lower() == "critical" else "0076D7",

"summary": subject,

"sections": [

{

"activityTitle": message\_title,

"text": f"\*\*Severity\*\*: {severity}\n\*\*Subject\*\*: {subject}\n[View Alert]({dashboard\_link})",

}

]

}

response = requests.post(

TEAMS\_WEBHOOK\_URL,

json=payload

)

if response.status\_code != 200:

print(f"Error sending Teams message: {response.text}")

else:

print("Teams alert sent successfully.")

**Usage**:

def store\_alert\_in\_db(alert\_record):

# ... (same as before)

# After storing the alert, we can push a Teams notification (for high/critical severity, for example).

if alert\_record["severity"].lower() in ["critical", "high"]:

alert\_record["dashboard\_link"] = f"https://my-alert-portal.com/alerts/{some\_alert\_id}"

send\_teams\_alert(alert\_record)

**Important**:

* Each Team’s channel has its own webhook URL.
* The content of the payload must follow the **MessageCard** format for Teams or an Adaptive Card if you want a richer layout.

**Step E: Example Backend Endpoints (Express.js or Flask)**

Below is a minimal example using **Flask (Python)** that exposes REST endpoints. This reuses the same Alert SQLAlchemy model from above.

from flask import Flask, request, jsonify

from models import db, Alert, Device, AlertResolution

from datetime import datetime

app = Flask(\_\_name\_\_)

@app.route("/alerts", methods=["GET"])

def get\_alerts():

status = request.args.get("status", None)

query = Alert.query

if status:

query = query.filter\_by(status=status)

alerts = query.all()

# Convert to JSON

results = []

for a in alerts:

results.append({

"id": a.id,

"device\_id": a.device\_id,

"severity": a.severity,

"status": a.status,

"timestamp": a.timestamp.isoformat(),

"subject": a.subject

})

return jsonify(results)

@app.route("/alerts/<int:alert\_id>", methods=["GET"])

def get\_alert(alert\_id):

alert = Alert.query.get\_or\_404(alert\_id)

return jsonify({

"id": alert.id,

"device\_id": alert.device\_id,

"severity": alert.severity,

"status": alert.status,

"timestamp": alert.timestamp.isoformat(),

"subject": alert.subject,

"raw\_message": alert.raw\_message,

"sender": alert.sender

})

@app.route("/alerts/<int:alert\_id>/resolve", methods=["POST"])

def resolve\_alert(alert\_id):

data = request.json

alert = Alert.query.get\_or\_404(alert\_id)

if alert.status == "resolved":

return jsonify({"message": "Alert is already resolved."}), 400

# Mark as resolved

alert.status = "resolved"

db.session.add(alert)

# Record resolution comment in alert\_resolutions

resolution = AlertResolution(

alert\_id=alert.id,

resolved\_by=data.get("resolved\_by"),

resolution\_comment=data.get("comment", ""),

resolved\_at=datetime.utcnow()

)

db.session.add(resolution)

db.session.commit()

return jsonify({"message": "Alert resolved successfully."})

@app.route("/devices/<int:device\_id>/alerts", methods=["GET"])

def get\_device\_alert\_history(device\_id):

alerts = Alert.query.filter\_by(device\_id=device\_id).order\_by(Alert.timestamp.desc()).all()

results = []

for a in alerts:

resolutions = AlertResolution.query.filter\_by(alert\_id=a.id).all()

resolution\_comments = [{"comment": r.resolution\_comment, "resolved\_at": r.resolved\_at.isoformat()}

for r in resolutions]

results.append({

"alert\_id": a.id,

"severity": a.severity,

"status": a.status,

"timestamp": a.timestamp.isoformat(),

"subject": a.subject,

"resolution\_comments": resolution\_comments

})

return jsonify(results)

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize DB, run app, etc.

db.init\_app(app)

app.run(debug=True)

**Step F: Frontend Example (React)**

A simple snippet using React to display alerts:

// AlertsTable.js

import React, { useEffect, useState } from 'react';

function AlertsTable() {

const [alerts, setAlerts] = useState([]);

useEffect(() => {

fetch('/alerts?status=new')

.then(resp => resp.json())

.then(data => setAlerts(data));

}, []);

return (

<div style={{ padding: '1rem' }}>

<h2>Active Alerts</h2>

<table style={{ width: '100%', borderCollapse: 'collapse' }}>

<thead>

<tr>

<th>ID</th>

<th>Device ID</th>

<th>Severity</th>

<th>Subject</th>

<th>Status</th>

</tr>

</thead>

<tbody>

{alerts.map(a => (

<tr key={a.id}>

<td>{a.id}</td>

<td>{a.device\_id}</td>

<td>{a.severity}</td>

<td>{a.subject}</td>

<td>{a.status}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

export default AlertsTable;

**Styling**:

* Use a design system like **Material UI** or **Chakra UI** for a cleaner “Intel-style” design.
* Add a detail view or a modal for each alert’s raw message and resolution button.

**3. Putting It All Together**

1. **Automate** the **Email Polling** process (Step A) to run on a schedule.
2. **Parse** the content and **store** alerts in the **DB** (Step B).
3. Provide a **REST API** for your portal (Steps D & E).
4. Integrate with **VictorOps** to fetch on-call info and display it in the UI or automatically assign “resolved\_by” (Step C).
5. Notify relevant channels (Teams, Slack, SMS) for critical alerts (Step D).
6. Build a **user-friendly frontend** with a neat UI library (Step F).

**4. Next Steps and Recommendations**

1. **Refine Parsing Logic**:
   * Use advanced pattern matching for various email formats.
   * Log unrecognized email formats to a “quarantine” for manual review.
2. **Add Authentication / RBAC**:
   * Integrate with Azure AD (if you want Single Sign-On) or use JWT-based auth.
   * Restrict “resolve” functionality to on-call or admin roles.
3. **Add Alert Escalations**:
   * If an alert stays unresolved for X minutes, automatically escalate to the next on-call or send additional notifications.
4. **Dashboard Enhancements**:
   * Include filtering by severity, device, and time ranges.
   * Provide real-time updates via WebSockets or long polling.
5. **Deploy**:
   * Containerize (Docker) the app.
   * Use CI/CD pipelines (GitHub Actions, Jenkins, Azure DevOps, etc.) to test and deploy to your environment of choice (AWS, Azure, on-prem, etc.).

**Final Thoughts**

By following these examples and expanding your schema to include on-call schedules (or integrating with VictorOps), you’ll have a robust system that:

* **Retrieves** alerts from Outlook (via Graph API).
* **Parses & Normalizes** them to a consistent DB schema.
* **Displays** them in a polished, modern UI where the ONCALL user can **resolve** issues.
* **Maintains** a full audit/history of alerts for each device.
* **Notifies** relevant stakeholders (via Teams or other channels) for critical alerts.

This approach will scale as you add more devices and more complex alert types. You can continually refine the frontend design to have that “soft, Intel-style” professional look while ensuring the ONCALL workflow is efficient and transparent. Good luck with your project!