Project Documentation: Email Summarizer

1. Introduction

This document provides comprehensive technical documentation for the Email Summarizer project. It details the system architecture, components, installation procedures, configuration guidelines, API specifications, and troubleshooting information.

2. System Architecture

The system employs a client-server architecture consisting of a backend service and a frontend user interface.

2.1. Backend

The backend is implemented in Python, utilizing the **FastAPI** framework for building the RESTful API and WebSocket communication. Key FastAPI features employed include:

- **Asynchronous Operations**: Leveraging `async` and `await` for non-blocking I/O, crucial for handling email fetching and API requests efficiently.
- **Routing**: Defining API endpoints (`@app.get`, `@app.post`, `@app.websocket`) to handle different functionalities like fetching emails, registering devices, and real-time updates.
- **Data Validation**: Using Pydantic models to define request and response structures, ensuring data integrity.
- **Dependency Injection**: Utilizing FastAPI's dependency injection system (e.g., `Depends`) for managing resources like the Gmail service connection.
- **Background Tasks**: Employing `BackgroundTasks` for operations like sending notifications without blocking the main request flow.
- **WebSocket Support**: Providing a `/ws` endpoint for real-time communication with the frontend.

It encompasses modules responsible for email retrieval, processing, natural language processing (NLP) tasks, and notification dispatch.

2.2. Frontend

The frontend is developed using React. It provides an interactive user interface for users to manage email accounts, view summaries, and configure notification preferences.

6. Core Components

6.1 Summarization (Pegasus Model)

The project utilizes the `google/pegasus-xsum` model for generating concise summaries of emails. The model is loaded and managed within `backend/summarizer.py`.

Initialization:

The model and tokenizer are initialized when needed using the `initialize_model` function. This ensures the potentially large model is only loaded into memory once.

```python
# backend/summarizer.py (Simplified Initialization)
from transformers import PegasusForConditionalGeneration, PegasusTokenizer

tokenizer = None
model = None

def initialize_model():
    global tokenizer, model
    if tokenizer is None or model is None:
        model_name = "google/pegasus-xsum"
        print(f"Initializing Pegasus model ({model_name})... This might take a moment.")
        tokenizer = PegasusTokenizer.from_pretrained(model_name)
        model = PegasusForConditionalGeneration.from_pretrained(model_name)
        print("Pegasus model initialized.")
```

Usage:

The `summarize_email` function takes email details (subject, sender, snippet, body), ensures the model is initialized, formats the input text, tokenizes it, and then generates the summary using `model.generate()`.

```python
# backend/summarizer.py (Simplified Summarization)
def summarize_email(subject, sender, snippet, body):
    initialize_model()  # Ensure model is initialized

    full_text = f"Summarize this email casually:\nSubject: {subject}\nFrom: {sender}\nSnippet: {snippet}\n\n{body}"

    tokens = tokenizer(full_text, truncation=True, padding="longest", return_tensors="pt", max_length=512)
    summary_ids = model.generate(
        tokens["input_ids"],
        max_length=150, # Adjust as needed
        min_length=40,
        num_beams=4,
        early_stopping=True
    )
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary
```

6.2 Google Authentication (OAuth 2.0)

Authentication with the Gmail API is handled via OAuth 2.0 using the `google-auth-oauthlib` and `google-api-python-client` libraries. The core logic resides in `backend/auth.py`.

Process:

1. **Credentials File (`credentials.json`):** You must obtain OAuth 2.0 Client ID credentials from the Google Cloud Console and save the downloaded JSON file as `credentials.json` in the project root directory.
2. **Token File (`token.json`):** On the first run, the application will:
    - Check if `token.json` exists in the root directory.
    - If not, or if the token is invalid/expired and cannot be refreshed, it uses `credentials.json` to initiate the OAuth 2.0 flow.
    - It starts a local web server and opens your browser, prompting you to log in to your Google account and grant permission.
    - Upon successful authorization, it receives an authorization code, exchanges it for an access token and a refresh token, and saves these in `token.json` in the project root.
3. **Subsequent Runs:** The application loads the credentials from `token.json`. If the access token is expired, it uses the refresh token (if available) to obtain a new access token without requiring user interaction.
4. **Service Object:** A Gmail API service object is built using the obtained credentials, allowing the application to make authenticated requests.

Code Snippet (`backend/auth.py` - Simplified Flow):

```
# backend/auth.py (Simplified Authentication Flow)
import os
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build

SCOPES = ['https://www.googleapis.com/auth/gmail.readonly']
# Paths are resolved relative to the backend directory
CREDS_PATH = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'credentials.json'))
TOKEN_PATH = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'token.json'))

def get_gmail_service():
    creds = None
    if os.path.exists(TOKEN_PATH):
        creds = Credentials.from_authorized_user_file(TOKEN_PATH, SCOPES)

    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            try:
                creds.refresh(Request())
            except Exception as e:
                print(f"Error refreshing token: {e}. Re-authenticating.")
                if os.path.exists(TOKEN_PATH):
                    os.remove(TOKEN_PATH) # Remove invalid token
                creds = None # Force re-auth
        else:
            if not os.path.exists(CREDS_PATH):
                raise FileNotFoundError(f"Credentials file not found at
{CREDS_PATH}")
            flow = InstalledAppFlow.from_client_secrets_file(CREDS_PATH, SCOPES)
            creds = flow.run_local_server(port=0)

        # Save the credentials for the next run
        with open(TOKEN_PATH, 'w') as token_file:
            token_file.write(creds.to_json())

    service = build('gmail', 'v1', credentials=creds)
    return service
```

Frontend Interaction:

The frontend (`frontend/src/App.jsx`) does not directly handle the Google Authentication process. It assumes the backend is already authenticated and running. It communicates with the backend via:

- **WebSocket (`ws://localhost:8000/ws`):** For real-time updates on new emails and summaries.
- **HTTP API (`http://localhost:8000/emails/{emailId}`):** To fetch the full details of a selected email on demand.

There is no specific code in the frontend for *initiating* the Google login or managing `token.json`. This is entirely handled by the backend when it starts up or needs to refresh credentials.

6.3 Backend API (FastAPI)

4. Installation

4.1. Prerequisites

- Python 3.9+
- Node.js and npm (for frontend)
- Git

4.2. Backend Setup

1. Clone the repository:

   ```
   git clone <repository-url>
   cd summarized-email-notifier
   ```

2. Navigate to the backend directory:

   ```
   cd backend
   ```

3. Create and activate a virtual environment:
   - On macOS/Linux:

     ```
     python3 -m venv .venv
     source .venv/bin/activate
     ```

   - On Windows (Command Prompt/PowerShell):

     ```
     python -m venv .venv
     .\.venv\Scripts\activate
     ```

4. Install Python dependencies:

   ```
   pip install -r requirements.txt
   ```

4.3. Frontend Setup

1. Navigate to the frontend directory (from the project root):

```
cd frontend
```

2. Install Node.js dependencies:

```
npm install
```

5. Configuration

1. **Backend**:
   - Navigate to the `backend` directory.
   - Copy the `.env.example` file to `.env`:
     - On macOS/Linux: `cp .env.example .env`
     - On Windows: `copy .env.example .env`
   - Populate the `.env` file with necessary credentials and settings. Example structure:

```
# .env file in backend directory

# Gmail API Credentials (obtain from Google Cloud Console)
# Store the 'credentials.json' file obtained from Google Cloud in the project root
GMAIL_CREDENTIALS_PATH=../credentials.json
# The token.json file will be generated automatically on first run
GMAIL_TOKEN_PATH=../token.json

# Storage Option ('local' or 'firestore')
STORAGE_OPTION=local

# Firebase (only needed if STORAGE_OPTION=firestore)
FIREBASE_SERVICE_ACCOUNT_KEY_PATH=./emailsummarizer-e34f2-firebase-adminsdk-fbsvc-xxxxxxxxxx.json # Replace with your actual key file name

# Optional: FCM Server Key for Push Notifications (if using notifier.py)
# FCM_SERVER_KEY=YOUR_FCM_SERVER_KEY
```

2. **Frontend**: Configuration variables (e.g., backend API endpoint) are typically set in `frontend/src/config.js` or similar, or via environment variables during the build process. Ensure the backend API URL is correctly pointing to where the backend server is running (e.g., `http://localhost:8000`).

6. Usage

1. **Start Backend Server**:
    - Ensure you are in the `backend` directory and the virtual environment is activated.
    - Run the FastAPI server:

      ```
      uvicorn main:app --host 0.0.0.0 --port 8000 --reload
      ```

      (Note: `--reload` is for development; remove it for production. `--host 0.0.0.0` makes it accessible on your network.)

2. **Start Frontend Development Server**:
    - Open a *new* terminal window/tab.
    - Navigate to the `frontend` directory.
    - Run the Vite development server:

      ```
      npm run dev
      ```

3. Access the application via the URL provided by the frontend development server (usually `http://localhost:5173` or similar, check the terminal output).

7. API Reference

Detailed API endpoint specifications, including request/response formats and authentication requirements, are provided below. The API uses standard HTTP methods and returns JSON responses.

7.1. Authentication

The current implementation does not enforce strict authentication for simplicity during development. In a production environment, mechanisms like OAuth2 with Google Sign-In or API Keys should be implemented to secure the endpoints.

7.2. Endpoints

- `GET /emails`: Retrieves recent emails with summaries and classifications.
    - Query Parameters:
        - `max_results` (int, optional, default: 10): Maximum number of emails to retrieve.
    - Response Body: `List[EmailResponse]`

```
[
  {
    "id": "email_id_1",
    "subject": "Meeting Reminder",
    "sender": "colleague@example.com",
    "snippet": "Just a reminder about our meeting...",
    "summary": "Reminder for the meeting scheduled today.",
    "category": "Work",
    "importance": 3,
    "icon": "📅"
  }
  // ... more emails
]
```

- `GET /emails/important`: Retrieves only emails classified as important.
    - Query Parameters:
        - `max_results` (int, optional, default: 10): Maximum number of important emails to retrieve.
    - Response Body: `List[EmailResponse]` (Same format as `/emails`)
- `POST /check-emails`: Triggers a background task to check for new emails, process them (summarize, classify, extract events), and store the results. Sends updates via WebSocket.
    - Request Body: None
    - Response Body:

```
{
  "status": "success",
  "message": "Email check initiated. Updates will be sent via WebSocket."
}
```

- `GET /events`: Extracts and retrieves upcoming events from recent emails.
  - Query Parameters:
    - `days_ahead` (int, optional, default: 7): How many days into the future to look for events.
    - `max_emails` (int, optional, default: 20): How many recent emails to scan for events.
  - Response Body: `List[EventResponse]`

```
[
  {
    "event_type": "Meeting",
    "description": "Project Sync-up",
    "date_str": "2024-08-15T10:00:00",
    "is_today": false,
    "is_tomorrow": true,
    "days_until": 1,
    "formatted_date": "Tomorrow at 10:00 AM"
  }
  // ... more events
]
```

- `POST /register-device`: Registers a device token for push notifications (requires FCM setup).
  - Request Body: `DeviceRegistration`

```
{
  "device_token": "your_fcm_device_token",
  "user_id": "optional_user_identifier",
  "device_name": "optional_device_name"
}
```

  - Response Body:

```
{
  "status": "success",
  "message": "Device registered successfully"
}
```

- `WS /ws`: WebSocket endpoint for real-time updates (e.g., new email summaries).
    - Client sends: No specific messages required from client after connection.
    - Server sends: JSON messages containing updates, e.g.:

      ```
      {"type": "new_summary", "data": { ... EmailResponse ... }}
      ```

      ```
      {"type": "processing_update", "message": "Checked 5 new emails."}
      ```

(Note: The `/api/summarize` and `/api/emails/sync` endpoints mentioned previously might be deprecated or replaced by the `/check-emails` workflow and direct fetching via `/emails` in the current implementation based on `api.py` and `main.py` analysis. The documentation reflects the endpoints found in the code.)

8. Troubleshooting

This section addresses common operational issues and their respective solutions.

- **Issue**: Backend server fails to start.
    - **Solution**: Verify Python dependencies are installed correctly and the `.env` file is properly configured.
- **Issue**: Frontend cannot connect to the backend.
    - **Solution**: Ensure the backend server is running and the frontend is configured with the correct API endpoint URL.

(Add more common issues and solutions)

9. Contributing

Please refer to the `CONTRIBUTING.md` file (if available) for guidelines on contributing to the project development, including code style, pull request procedures, and issue reporting.