Software Development Intern Assignment

About Us:

ReachInbox is transforming cold outreach with our revolutionary AI-driven platform. Our all-in-one solution empowers businesses to effortlessly find, enrich, and engage high-intent leads through multi-channel outreach on Twitter, LinkedIn, email, and phone. With just a single prompt, ReachInbox springs into action, prospecting and verifying leads, crafting personalized sequences, and notifying businesses of responsive prospects. Imagine being part of an AI-powered growth team that consistently generates top-tier leads. ReachInbox is more than a tool; it's your growth partner.

We are looking for passionate and innovative individuals to join our team and help us continue to redefine the future of lead generation and business growth.

🚀 ReachInbox Hiring Assignment – Full-stack Email Job Scheduler

🎯 Problem Statement

At ReachInbox, a huge part of our system is **reliable scheduling and sending of emails at scale**.

Your task is to build a **production-grade email scheduler service + dashboard** that:

- Accepts **email send requests** via APIs
- Schedules them to be sent at a **specific time**
- Uses **BullMQ + Redis** as a persistent job scheduler (**no cron jobs**)
- Sends emails using fake SMTP via **Ethereal Email**
- Survives **server restarts** without restarting from scratch or losing jobs
- Exposes a **frontend dashboard** to:
    - Schedule new emails
    - View scheduled emails
    - View sent emails

Think of it as a tiny slice of what ReachInbox does under the hood.

🧪 Tech Requirements

You **must** use:

Backend

- Language: **TypeScript**
- Framework: **Express.js**
- Queue: **BullMQ** (backed by **Redis**)
- Database: **MySQL or PostgreSQL** (ORM or query builder is your choice)
- SMTP: **Ethereal Email** (fake SMTP for testing)

Frontend

- **React.js or Next.js**
- **Tailwind CSS** (or any modern CSS library)
- **TypeScript** strongly preferred

Infra

- Redis and DB can be run via **Docker** (recommended but not mandatory)

We care about how you **structure a real backend**, **wire queues correctly**, and **build a clean frontend**.

🖥️ Backend Requirements

1️⃣ Core Scheduler Behavior

Your backend must:

- Accept email scheduling requests via API
- Store them in a **relational DB** (MySQL/Postgres)
- Schedule them using **BullMQ delayed jobs** (or a custom Redis/DB-based scheduler) – **no cron**
- Send emails from multiple senders via **Ethereal Email (SMTP)**
- Persist state so that:
    - If the server restarts, **future emails are still sent at the correct time**
    - Emails are **not duplicated** or restarted from scratch

2️⃣ Throughput, Rate Limiting & Concurrency (Required)

Your scheduler should behave like a **real-world email system under load**.

You **must** support:

✅ Worker Concurrency

- Configure your BullMQ worker(s) with a **configurable concurrency** level .
- Implementation must be safe when multiple jobs run in parallel:

✅ Delay Between Each Email

- There must be a **minimum delay between individual email sends** (to mimic provider throttling).
- You can:
  - Use **BullMQ's limiter** options, **or**
  - Add a custom delay in the worker logic.
- Document in the README what delay you chose (e.g. "min 2 seconds between sends").

✅ Emails Per Hour (Rate Limiting)

- Implement a **rate limit** on the number of emails sent per hour:
  - Either **global** (e.g. `MAX_EMAILS_PER_HOUR=200`), or
  - **Per-sender / per-tenant** (e.g. `MAX_EMAILS_PER_HOUR_PER_SENDER`), you will have to support multiple senders.
- The **limit values must be configurable** via env/config (no hardcoding).
- Rate-limiting logic must be **safe across multiple workers / instances**:
  - Use **Redis or DB** backed counters (e.g. keyed by `hour_window + sender`),
  - Do **not** rely only on in-memory counts.
- When the hourly limit is reached:
  - **Do not drop** or permanently fail jobs.
  - Jobs should be **delayed / rescheduled** into the **next available hour window** while preserving order as much as possible.
- Explain in the README how you enforce this:
  - e.g. BullMQ limiter, Redis counters, custom logic, trade-offs, etc.

✅ Behavior Under Load

Your design should clearly define behavior when:

- **1000+ emails** are scheduled for roughly the same time.
- The **rate limit** would be exceeded.

You don't need to actually send thousands via Ethereal, but your logic should handle it.

3️⃣ Hard Constraints (Important)

These are **non-negotiable**:

- ❌ **Do NOT use cron jobs**
    - No OS-level cron (`crontab`, etc.)
    - No Node cron libraries (`node-cron`, `agenda`, etc.)
- ✅ Scheduling must be done using:
    - **BullMQ delayed jobs**, or
    - A **custom scheduler** that uses Redis/DB to track next run times — but still **not cron**.
- ✅ The system **must be persistent**:
    - After a restart:
        - Future scheduled emails **still send at the right time**
        - Emails are **not re-sent** or restarted from Day 1
- ❌ Same email queues should not be sent more than once. Maintain idempotency.

🎨 Frontend Requirements

You will build a **frontend that matches the provided Figma** as closely as possible and talks to your backend APIs.

Figma Link

1️⃣ Google Login (Required)

- Implement **real Google OAuth login** (no mock).
- After login, redirect the user to the **dashboard**.
- Show in the top header:
    - User's **name**
    - **Email**
    - **Avatar**
- Provide a simple **Logout** option.

2️⃣ Main Dashboard

After login, show the main UI with:

- **Top header** (user info + logout).
- Tabs/sections:
    - **Scheduled Emails**
    - **Sent Emails**

- A primary **“Compose New Email”** button.

Layout and styling should closely follow the **Figma design**.

3️⃣ Compose New Email

User must be able to:

- Enter:
  - **Subject**
  - **Body**
- **Upload** a CSV/text file of email leads.
  - Parse and show the **number of email addresses** detected.
- Set:
  - **Start time** (when scheduling begins)
  - **Delay between emails**
  - **Hourly limit**
- Click **Schedule** to send data to the **backend schedule API**.

This can be a **modal** or **separate page**, depending on the Figma.

4️⃣ Scheduled Emails

Show a clean table/list with:

- **Email**
- **Subject**
- **Scheduled time**
- **Status**

Include:

- **Loading states**
- **Empty state** when there are no scheduled emails

5️⃣ Sent Emails

Show a table/list with:

- **Email**
- **Subject**
- **Sent time**
- **Status** (`sent` / `failed`)

Include:

- **Loading states**
- **Empty state** when there are no sent emails

6️⃣ Frontend Code Quality

We expect:

- Clean **folder structure**
- Reusable **UI components** (buttons, inputs, tables, modals, etc.)
- **DRY** code (avoid duplication)
- Proper **TypeScript** usage:
    - Types/interfaces for API responses & props
- Good UX:
    - Loading indicators
    - Empty states
    - Error handling (basic messages/toasts)

📦 Submission Guidelines

1. **Create a private GitHub repository** (monorepo or separate folders for backend & frontend is fine).
2. **Grant access** to user: `Mitrajit`
3. Add a **README** that includes:
    - How to run **backend** (Express, Redis, DB, BullMQ worker)
    - How to run **frontend**
    - How to set up **Ethereal Email** and env variables
    - Architecture overview:
        - How scheduling works
        - How persistence on restart is handled
        - How rate limiting & concurrency are implemented
    - List of **features implemented**, mapped to:
        - Backend: scheduler, persistence, rate limiting, concurrency
        - Frontend: login, dashboard, compose, tables, etc.
4. Add a **short demo video (max 5 minutes)**:
    - Show creating scheduled emails (from frontend or Postman)
    - Show the dashboard with **Scheduled** and **Sent** emails
    - Show a **restart scenario**:
        - Stop server → start again → future emails still send
    - (Bonus) Briefly demonstrate how **rate limiting / delay** behaves under load
5. Note any **assumptions, shortcuts, or trade-offs** you made.

<aside> 💡 **Fill this form with relevant links and details - Assignment Submission - https://forms.gle/SRv2FoT913n2B2yg6

</aside>

Deadline for Task Completion:

You have a maximum of 48 hours to complete the task. Receiving this assignment means you're already ahead of many candidates. Good luck!

Note: Do not submit a plagiarized assignment. All GitHub code will be thoroughly reviewed, and any evidence of plagiarism will result in the assignment being rejected.

Thank you!