Homework 1 - OSPD F'25

### Assignment 1: Building a Mail Client API Server

#### Making Our Components Usable

In Homework 0, you built the core logic of our mail client as a set of Python packages. However, these components can only be used by other Python code that imports them directly. There is currently no way for an end-user, a web browser, or another application to interact with the system over a network.

This assignment is about creating a **service** that exposes the functionality of your mail client to the outside world via a REST API. This is the critical step that transforms your Python library into a usable, interactive application.

#### What is a Service?

In this course, we make a crucial distinction between a **component** and a **service**:

* A **Component** is a unit of **code organization and packaging**. It is a self-contained, installable Python package (e.g., a wheel) that exposes its functionality through a public API of classes and functions. Other Python code interacts with it directly via import statements. Your gmail\_client\_impl is a component; it's a library that another part of a Python application can use.
* A **Service** is a unit of **deployment and runtime execution**. It is an independently running process that exposes its functionality over a network protocol, typically HTTP. You do not import a service; you communicate with it by making network requests to specific endpoints (e.g., GET /messages). The service owns its own process and memory space.

The key difference is the boundary of interaction:  
**Component Boundary:** A function call within the same process (my\_client.get\_messages()).  
**Service Boundary:** A network call between different processes (requests.get("http://localhost:8000/messages")).

#### Learning Goals

* Understand the difference between a library/component and a network service.
* Design and implement a clean RESTful API using FastAPI.
* Containerize an application using Docker.
* Write unit and integration tests for a web service, mocking the underlying components.

#### Build a FastAPI Service

Your team will add a new component to your workspace: a FastAPI server that acts as a web-accessible interface for your mail client.

Create a new, self-contained Python package named **mail\_client\_service** inside the src/ directory.

**Requirements:**

1. **pyproject.toml:** The new component must have its own pyproject.toml file.
2. **Workspace Integration:**
   * Add the new component to the [tool.uv.workspace].members list in your root pyproject.toml.
   * The mail\_client\_service will depend on your gmail\_client\_impl and gmail\_message\_impl packages. These should be declared as workspace dependencies.
3. **FastAPI Application:**
   * Inside your new component, create a FastAPI application.
   * This application should use your existing mail\_client\_api.get\_client() factory to get a client instance. **Do not re-implement any logic.** Your service should only be a thin wrapper around the components already built.
4. **API Endpoints:** Your service must expose (at a minimum) the following RESTful endpoints:
   * GET /messages: Fetches a list of message summaries.
   * GET /messages/{message\_id}: Fetches the full detail of a single message.
   * DELETE /messages/{message\_id}: Deletes a message.
   * ... other ideas?
5. **Testing:**
   * **Unit Tests (src/mail\_client\_service/tests/):** Write unit tests for your API endpoints. These tests **must mock the mail\_client\_api.Client**. You are testing your FastAPI logic (request handling, response codes, data serialization), not your Gmail client.
   * **Integration Tests (tests/integration/):** Add a new integration test that starts the FastAPI server and makes a real HTTP request to it, verifying that the service correctly calls the underlying GmailClient.
   * Some sort of E2E test maybe?

#### Extra Credit Opportunities

For teams who complete the core assignment, you can extend your project in the following ways. These are not required but offer a chance to explore more advanced topics.

1. **Containerize the Service (Docker):**
   * Create a Dockerfile in the root of your project.
   * The Dockerfile should correctly build and run your FastAPI service.
   * Provide instructions in your README.md on how to build and run the Docker container.
2. **Deploy the Service Online:**
   * Deploy your containerized service to a cloud platform (e.g., AWS, Google Cloud Run, fly.io, etc.).
   * Provide the public URL to your running service in your final submission.
   * **Note:** This is a significant challenge that involves managing cloud resources and secrets.
3. **Design an Advanced API for AI Integration (MCP):**
   * Think about the "intelligent assistant" goal. What API endpoints would an AI agent need?
   * Design and implement new endpoints that go beyond basic CRUD, such as:
     + POST /messages/summarize: An endpoint that takes a list of message IDs and returns an AI-generated summary.
     + GET /insights: An endpoint that analyzes the last 20 messages and returns key topics or sentiments.

#### Timeline and Review Process

idk yet