HW2

# Homework 2

## Intro

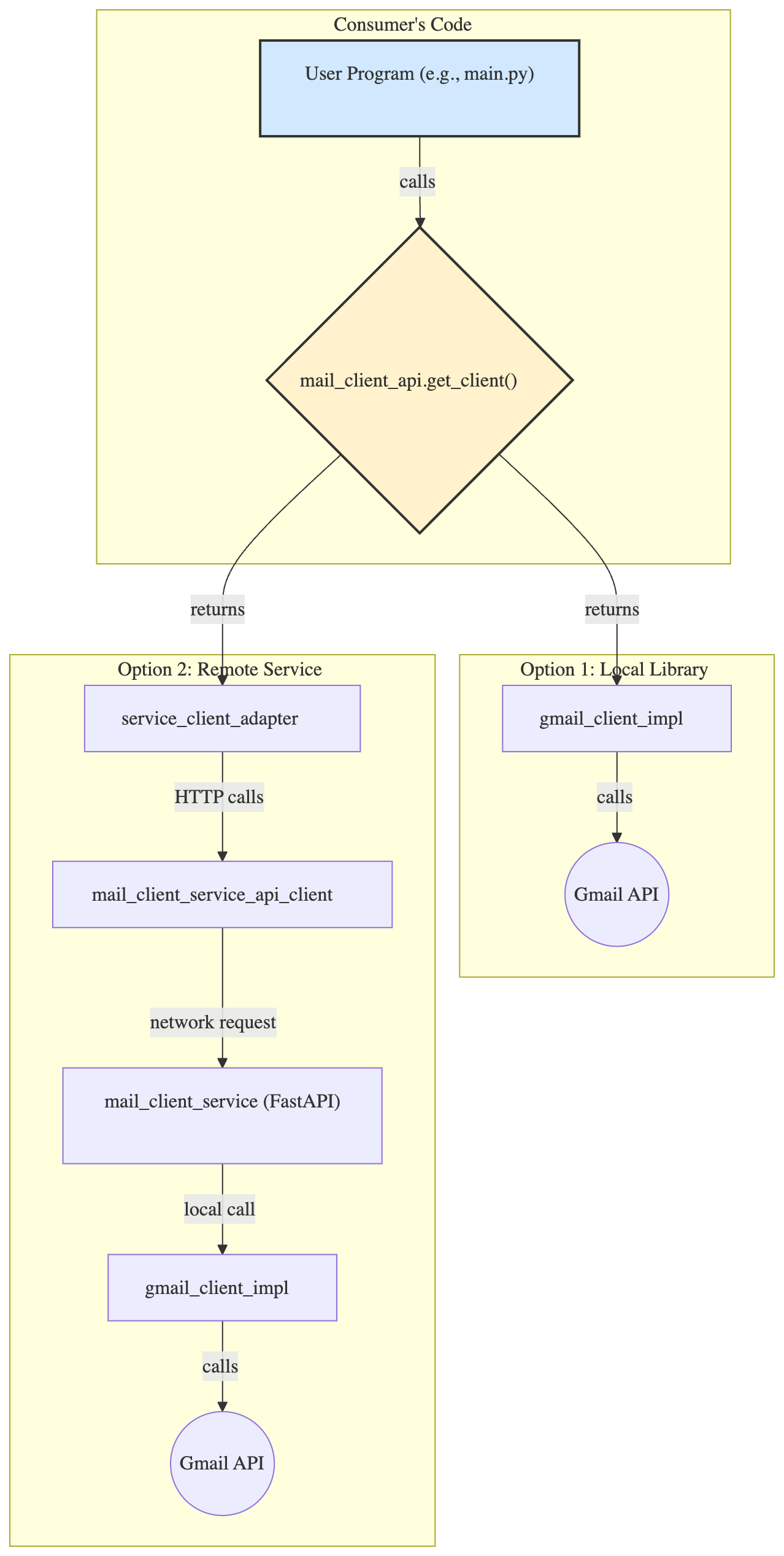
Congratulations on successfully implementing your Gmail service adapter in Homework 1! You navigated the foundational steps of turning a local **component** into a discoverable **service** that remains locally usable via the **Adapter Pattern**.

To recap the components we focused on was:

1. **Interface API (Contract):** Defining the abstract interface for what your package does.
2. **Implementation:** Building the core logic that wraps an external API (like Gmail).
3. **FastAPI Service (Deployment Unit):** Exposing your core logic over HTTP endpoints.
4. **Auto-generated Client:** Using tools like openapi-python-client to create a thin, type-safe network client from your service's OpenAPI spec.
5. **Service Client Adapter:** Wrapping the auto-generated client with the original Interface API, hiding the network details from the end-user.

This architecture ensures that whether the code is running as a local library or as a remote service is merely a matter of geography—the consumer's code doesn't have to change.

In this next assignment, you will apply this entire cycle to a **new problem domain** and, most importantly, **elevate your professional development practices and system architecture.**



## Remarks and Increased Expectations

We saw a lot of great work in Homework 1. For Homework 2, we will be stricter about the development process. Please take note of the following:

### Commits and Pull Requests

Many PRs contained a large number of small, iterative commits. While committing frequently is good practice during development, it makes the final review process difficult.

For HW2, your workflow should be structured as follows:

1. Create a main branch for the assignment (e.g., hw2).
2. For each significant piece of work (e.g., designing the API, implementing the chat logic, setting up authentication), create a new **feature branch** off your hw2 branch (e.g., hw2-api-design).
3. You can have as many commits as you want on your feature branch.
4. When the feature is complete, **squash and merge** the feature branch into your hw2 branch. This condenses all the intermediate work into a single, meaningful commit on the main assignment branch. For example, 30 commits on hw2-api-design can become one commit on hw2 titled "feat: Design and implement the core service API".
5. Your final PR to the class repository should be from your hw2 branch, containing a clean history of these squashed feature commits.

Also, remember these big no-nos:

* **Do not push directly to main**. All work must go through PRs.
* **Always rebase** to incorporate changes made by TAs. This avoids messy merge commits and keeps your contribution history clean.

### MyPy and Ruff

Your code must pass mypy and ruff checks. These are not hassles; they are essential tools for writing safe, maintainable Python.

* **Mypy:** Do not ignore Mypy errors with type: ignore unless you have a very specific, documented reason. An error from Mypy indicates a potential issue with your type safety or configuration.
* **Ruff:** A few noqa comments are acceptable for niche cases (especially in tests), but they must be explained. Your source code should not be littered with ignored rules.
* **Workflow:** Run these tools **consistently as you code**, not just at the end. This helps catch issues early and keeps your codebase clean.

### CircleCI

Set up your CircleCI configuration **as early as possible**. Don't wait until the end of the assignment. This allows you to see if your changes are breaking the build throughout your development process.

### Authentication

In HW1, the method for handling authentication was a simple workaround to get things running locally. For HW2, we expect a more robust, professional implementation.

Your new service must implement a proper **OAuth 2.0 flow**. When a user needs to authenticate, your service should:

1. Redirect the user to an external provider (e.g., Google).
2. Handle the callback from the provider.
3. Receive the credentials (e.g., access token, refresh token).
4. Securely **store these credentials in a database** associated with the user.

Subsequent requests from that user will use the stored credentials to interact with the underlying API (e.g., Slack, OpenAI, etc.).

## The Assignment

### Objective

The primary goal of this assignment is to design, implement, and deploy a complete, standalone microservice from scratch.

You will select a service category, define its API, build its core functionality, and make it publicly accessible.

This project will build upon the skills you developed in HW1 and challenge you to think about system architecture and user interaction.

Throughout this entire sprint, you must adhere to the guidelines talked about in class regarding testing and dev workflow.

### Instructions

#### 1. Service Selection

Each team will choose **one** of the following service categories to implement. We'll randomly release a sign-up sheet sometime in the next 24 hours for fairness.

* **Chat Service** (e.g., wrapping the Slack, Discord, or WhatsApp API)
* **Mail Service** (e.g., wrapping the Outlook or another mail provider's API [that's obviously not gmail])
* **AI Service** (e.g., providing an interface to an AI model like OpenAI's GPT or Anthropic's Claude)
* **Issue Tracker** (e.g., wrapping the Jira, Trello, or Asana API)
* **Custom** (custom ideas must be approved by TAs)

#### 2. Build the Core Components and Service

Follow these steps to build your application, mirroring the structure of Homework 1.

**A. The Abstract API ([your\_service]\_api)**

* **Action:** Define the abstract contract (ABC) for your new service. This is your mail\_client\_api equivalent.
* **Goal:** Create a clean, minimal interface that defines *what* your service does, not *how*. This package should have no dependencies on the external service's libraries.

**B. The Concrete Implementation ([your\_service]\_impl)**

* **Action:** Create the concrete implementation that wraps the external API (e.g., Slack, Jira). This corresponds to gmail\_client\_impl.
* **Key Requirement:** This component must implement the new **OAuth 2.0 flow** as described in the "Remarks" section. It will handle redirecting the user, processing callbacks, and securely storing credentials.
* **Goal:** This package contains the core business logic and interacts directly with the third-party service.

**C. The FastAPI Service ([your\_service]\_service)**

* **Action:** Build a FastAPI service that exposes your concrete implementation over HTTP endpoints. This service will be your main deployment unit and corresponds to the mail\_client\_service from HW1.
* **Functionality:**
  + It should import and use your concrete implementation from Step B.
  + It must expose endpoints for the OAuth 2.0 flow.
  + It should provide endpoints for all the core functions defined in your abstract API.

**D. The Auto-Generated Client ([your\_service]\_service\_api\_client)**

* **Action:** Once your FastAPI service is running locally, use a tool like openapi-python-client to generate a client library from its OpenAPI specification (/openapi.json).
* **Goal:** This package provides a type-safe way to make network requests to your service, just like in HW1.

**E. The Service Client Adapter ([your\_service]\_adapter)**

* **Action:** Finally, create an adapter that implements your original abstract API from Step A. However, instead of performing the logic directly, this adapter will use the auto-generated client from Step D to make network calls to your running service.
* **Goal:** This component makes your remote service usable through the exact same contract as your local library, achieving the "Adapter Pattern" and location transparency.

#### 3. Deployment

@Kamen Are we making students deploy this?

### Extra Credit

@Kamen what can we add?

## Timeline and Submission

This is a two-week assignment with the following key deadlines:

| Milestone | Due Date & Time | Duration |
| --- | --- | --- |
| **HW2 Released** | Wednesday (10/22) | — |
| **First Run Submission** | Tuesday (10/27) @ Midnight | (6 days) |
| **Peer Review + TA Reviews** | Friday (10/31) @ Midnight | (3 days) |
| **Final Submission** | Wednesday (11/5) @ Midnight | (5 days) |

## Checklist on Motions

Submitting work that fails these checks is considered an incomplete submission.

* **uv:** uv is your package manager. No requirements.txt or pip. pyproject.toml only.
* **Code Quality and Static Analysis:** Use ruff (ruff check) and mypy (mypy src tests). Make sure it all passes.
* **Testing and Coverage:**
  + Does your submission include comprehensive unit, integration, and E2E tests?
  + Does your code meet or exceed the coverage percentage defined in your root pyproject.toml? A failing coverage check is a failing build.
* **Continuous Integration:**
  + Have you pushed your latest commits to your team's remote feature branch on GitHub?
  + Is your CircleCI build for that branch passing?
  + Is your CircleCI project public?
* **Documentation:**
  + Have you updated the documentation in the root README.md and in the README.md file for each component you created or modified?
  + Does your mkdocs documentation build correctly and reflect the current state of your project?
* **PRs and Commits:**
  + Does your PR have a descriptive title and a clear summary of the changes?
  + Is your commit history meaningful and concise, following the squash-merge workflow?