

Scope

I currently have 3 email accounts `stephen.lohning@oxnee.com` which is a microsoft account `slohning@live.com.au` a microsoft 365 account `slohning@gmail.com` Google Account

I wish to write python program to down load the emails from email from all of those accounts. The email will be access via there Microsoft Graph API and the Google API. The emails will be download and stored as eml format. The python program will use uv the gui will be developed PySide6 The meta data from the from the email will be stored in the a sqlite3 extra meta data will be added, such as project name, organization etc. The data base will store the data as a json so that the we can expand the number of is not limited.

The project is call 144-python-email-extractor

Response

Here is a complete design and working implementation for **144-python-email-extractor**.

This project uses **uv** for dependency management, **PySide6** for a modern Qt desktop UI, **Microsoft Graph API** for your Microsoft accounts, **Google Gmail API** for your Gmail account, and **SQLite3** with a JSON column to store extensible metadata alongside the saved **.eml** files.

1. Project Setup with uv

Create your project structure and initialize it using **uv**:

```
mkdir -p 144-python-email-extractor/eml_storage
cd 144-python-email-extractor

# Initialize uv environment
uv init
uv add PySide6 msal requests google-api-python-client google-auth-oauthlib
google-auth-httpplib2
```

2. SQLite Database Schema (db.py)

We create a database containing an **emails** table. By using SQLite's native **JSON** type for **extra_metadata**, you can store dynamic attributes like **project_name**, **organization**, **tags**, or **status** without needing to alter database schemas later.

Save this as **db.py**:

```
import sqlite3
import json

DB_FILE = "email_archive.db"

def init_db():
    """Initializes the SQLite database schema."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    # Table schema with JSON extra_metadata column
    cursor.execute("""
CREATE TABLE IF NOT EXISTS emails (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    account_email TEXT NOT NULL,
    message_id TEXT NOT NULL UNIQUE,
    subject TEXT,
    sender TEXT,
    received_date TEXT,
    file_path TEXT NOT NULL,
    extra_metadata TEXT, -- JSON blob for flexible, unlimited fields
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
)
""")
    conn.commit()
    conn.close()

def save_email_record(account_email, message_id, subject, sender,
received_date, file_path, metadata_dict):
    """Inserts or updates an email record with extra JSON metadata."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    json_metadata = json.dumps(metadata_dict)

    cursor.execute("""
INSERT INTO emails (account_email, message_id, subject, sender,
received_date, file_path, extra_metadata)
VALUES (?, ?, ?, ?, ?, ?, ?)
ON CONFLICT(message_id) DO UPDATE SET
    extra_metadata = excluded.extra_metadata,
    file_path = excluded.file_path
""", (account_email, message_id, subject, sender, received_date,
file_path, json_metadata))

    conn.commit()
    conn.close()
```

3. Email Fetching Engines ([downloaders.py](#))

This handles OAuth2 authentication and fetching raw `.eml` MIME data for both API ecosystems.

- **Microsoft Graph API:** Retrieves messages via the `/$value` endpoint (`stephen.lohning@oxnee.com` & `slohning@live.com.au`).
- **Google Gmail API:** Downloads raw emails using `users().messages().get(id=..., format='raw')` and converts `base64url` MIME to standard `.eml` (`slohning@gmail.com`).

Save this as `downloaders.py`:

```
import os
import re
import base64
import requests
import msal
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials

STORAGE_DIR = "./eml_storage"

def sanitize_filename(name):
    clean = re.sub(r'[\\"/*?:"<>|]', "", name or "Untitled")
    return clean[:80].strip()

#
=====
===
# MICROSOFT GRAPH API DOWNLOADER
#
=====
===
class MSGraphDownloader:
    def __init__(self, tenant_id, client_id, client_secret, account_email):
        self.account_email = account_email
        self.tenant_id = tenant_id
        self.client_id = client_id
        self.client_secret = client_secret
        self.token = self._get_token()

    def _get_token(self):
        app = msal.ConfidentialClientApplication(
            self.client_id,
            authority=f"https://login.microsoftonline.com/{self.tenant_id}",
            client_credential=self.client_secret,
        )
        res = app.acquire_token_for_client(scopes=[
            "https://graph.microsoft.com/.default"
        ])
        return res.get("access_token")

    def download_inbox(self, callback_signal):
```

```
        if not self.token:
            callback_signal.emit(f"[{self.account_email}] Auth Error: Could
not acquire token.")
            return []

        headers = {"Authorization": f"Bearer {self.token}", "Accept":
"application/json"}
        url =
f"https://graph.microsoft.com/v1.0/users/{self.account_email}/mailFolders/i
nbox/messages?$select=id,subject,from,receivedDateTime&$top=20"

        extracted = []
        os.makedirs(STORAGE_DIR, exist_ok=True)

        res = requests.get(url, headers=headers)
        if res.status_code != 200:
            callback_signal.emit(f"[{self.account_email}] Failed to list
emails: {res.status_code}")
            return []

        messages = res.json().get("value", [])
        for msg in messages:
            msg_id = msg["id"]
            subject = msg.get("subject", "No Subject")
            sender = msg.get("from", {}).get("emailAddress",
{}).get("address", "Unknown")
            date_str = msg.get("receivedDateTime", "")[:10]

            # Get raw .eml content
            raw_url =
f"https://graph.microsoft.com/v1.0/users/{self.account_email}/messages/{msg
_id}/$value"
            eml_res = requests.get(raw_url, headers=headers)

            if eml_res.status_code == 200:
                fname =
f"MS_{date_str}_{sanitize_filename(subject)}_{msg_id[-6:]} .eml"
                fpath = os.path.abspath(os.path.join(STORAGE_DIR, fname))

                with open(fpath, "wb") as f:
                    f.write(eml_res.content)

                extracted.append({
                    "account": self.account_email,
                    "id": msg_id,
                    "subject": subject,
                    "sender": sender,
                    "date": date_str,
                    "path": fpath
                })
                callback_signal.emit(f"[{self.account_email}] Downloaded:
{fname}")

        return extracted
```

```

#
=====
===
# Gmail API DOWNLOADER
#
=====
===
class GmailDownloader:
    SCOPES = ['https://www.googleapis.com/auth/gmail.readonly']

    def __init__(self, credentials_json,
account_email="slohning@gmail.com"):
        self.account_email = account_email
        self.creds_json = credentials_json
        self.service = self._authenticate()

    def _authenticate(self):
        creds = None
        token_file = 'token_gmail.json'
        if os.path.exists(token_file):
            creds = Credentials.from_authorized_user_file(token_file,
self.SCOPES)
        if not creds or not creds.valid:
            if creds and creds.expired and creds.refresh_token:
                creds.refresh(Request())
            else:
                flow =
InstalledAppFlow.from_client_secrets_file(self.creds_json, self.SCOPES)
                creds = flow.run_local_server(port=0)
            with open(token_file, 'w') as token:
                token.write(creds.to_json())
        return build('gmail', 'v1', credentials=creds)

    def download_inbox(self, callback_signal):
        results = self.service.users().messages().list(userId='me',
maxResults=20, q="label:INBOX").execute()
        messages = results.get('messages', [])

        extracted = []
        os.makedirs(STORAGE_DIR, exist_ok=True)

        for m in messages:
            msg_id = m['id']
            # Get raw RFC 822 format
            raw_msg = self.service.users().messages().get(userId='me',
id=msg_id, format='raw').execute()

            # Get full metadata headers
            meta_msg = self.service.users().messages().get(userId='me',
id=msg_id, format='full').execute()
            headers = meta_msg.get('payload', {}).get('headers', [])

            subject = next((h['value'] for h in headers if

```

```

h['name'].lower() == 'subject'), 'No Subject')
    sender = next((h['value'] for h in headers if h['name'].lower()
== 'from'), 'Unknown')
    date_str = next((h['value'] for h in headers if
h['name'].lower() == 'date'), 'UnknownDate')

    # Decode RFC 2822 base64url data into .eml binary stream
    eml_data =
base64.urlsafe_b64decode(raw_msg['raw'].encode('ASCII'))

    fname = f"GMAIL_{sanitize_filename(subject)}_{msg_id[-6:]}.eml"
    fpath = os.path.abspath(os.path.join(STORAGE_DIR, fname))

    with open(fpath, "wb") as f:
        f.write(eml_data)

    extracted.append({
        "account": self.account_email,
        "id": msg_id,
        "subject": subject,
        "sender": sender,
        "date": date_str,
        "path": fpath
    })
    callback_signal.emit(f"[{self.account_email}] Downloaded:
{fname}")

    return extracted

```

4. PySide6 Desktop Interface (main.py)

This application provides thread-safe execution, interactive setup for **Project Name** and **Organization** metadata tags, live log streaming, and an embedded viewer for database records.

Save this as **main.py**:

```

import sys
import json
import sqlite3
from PySide6.QtCore import Qt, QThread, Signal
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
    QLabel, QLineEdit, QPushButton, QTextEdit, QTableWidgetItem,
    QTableWidgetItem,
    QHeaderView, QGroupBox, QMessageBox
)

from db import init_db, save_email_record, DB_FILE
from downloaders import MSGraphDownloader, GmailDownloader

```

```

# Thread worker to prevent freezing the UI during API calls
class SyncWorker(QThread):
    log_signal = Signal(str)
    finished_signal = Signal()

    def __init__(self, project_name, organization, ms_credentials,
gmail_credentials_file):
        super().__init__()
        self.project_name = project_name
        self.organization = organization
        self.ms_credentials = ms_credentials
        self.gmail_credentials_file = gmail_credentials_file

    def run(self):
        self.log_signal.emit("Starting multi-account email sync...")
        init_db()

        records = []

        # 1. Sync Microsoft Graph Accounts
        if self.ms_credentials.get("client_id"):
            for email in ["stephen.lohning@oxnee.com",
"slohning@live.com.au"]:
                self.log_signal.emit(f"Connecting to Microsoft Graph for
{email}...")
                ms = MSGraphDownloader(
                    tenant_id=self.ms_credentials["tenant_id"],
                    client_id=self.ms_credentials["client_id"],
                    client_secret=self.ms_credentials["client_secret"],
                    account_email=email
                )
                records.extend(ms.download_inbox(self.log_signal))

        # 2. Sync Google Gmail Account
        if self.gmail_credentials_file:
            self.log_signal.emit("Connecting to Google Gmail API for
slohning@gmail.com...")
            try:
                gm = GmailDownloader(self.gmail_credentials_file,
"slohning@gmail.com")
                records.extend(gm.download_inbox(self.log_signal))
            except Exception as e:
                self.log_signal.emit(f"[slohning@gmail.com] Error: {e}")

        # 3. Store Records and JSON metadata in SQLite
        for r in records:
            extra_meta = {
                "project_name": self.project_name,
                "organization": self.organization,
                "extractor_version": "1.0.0",
                "extracted_via": "144-python-email-extractor"
            }
            save_email_record(

```

```
        account_email=r["account"],
        message_id=r["id"],
        subject=r["subject"],
        sender=r["sender"],
        received_date=r["date"],
        file_path=r["path"],
        metadata_dict=extra_meta
    )

    self.log_signal.emit(f"\nSync Complete! Processed {len(records)}
messages into SQLite.")
    self.finished_signal.emit()

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("144-python-email-extractor")
        self.resize(950, 700)

        init_db()
        self.init_ui()

    def init_ui(self):
        central = QWidget()
        self.setCentralWidget(central)
        layout = QVBoxLayout(central)

        # Config Inputs Box
        config_box = QGroupBox("Extraction & Metadata Parameters")
        cfg_layout = QHBoxLayout(config_box)

        self.txt_project = QLineEdit("Project_144")
        self.txt_project.setPlaceholderText("Project Name")
        self.txt_org = QLineEdit("0xnee")
        self.txt_org.setPlaceholderText("Organization")

        cfg_layout.addWidget(QLabel("Project:"))
        cfg_layout.addWidget(self.txt_project)
        cfg_layout.addWidget(QLabel("Organization:"))
        cfg_layout.addWidget(self.txt_org)

        self.btn_sync = QPushButton("Start Download & Extraction")
        self.btn_sync.clicked.connect(self.start_sync)
        cfg_layout.addWidget(self.btn_sync)

        layout.addWidget(config_box)

        # Log Window
        self.txt_log = QTextEdit()
        self.txt_log.setReadOnly(True)
        self.txt_log.setMaximumHeight(150)
        layout.addWidget(QLabel("Live Extraction Logs:"))
        layout.addWidget(self.txt_log)
```

```

        # SQLite Database Table Viewer
        self.table = QTableWidgetItem()
        self.table.setColumnCount(6)
        self.table.setHorizontalHeaderLabels(["ID", "Account", "Subject",
"Sender", "Path", "JSON Extra Metadata"])

self.table.horizontalHeader().setSectionResizeMode(QHeaderView.ResizeToCont
ents)
        self.table.horizontalHeader().setSectionResizeMode(2,
QHeaderView.Stretch)
        layout.addWidget(QLabel("Stored SQLite Database Index:"))
        layout.addWidget(self.table)

        self.refresh_table()

def append_log(self, text):
    self.txt_log.append(text)

def start_sync(self):
    self.btn_sync.setEnabled(False)
    self.txt_log.clear()

    # Placeholders for MS Azure credentials (or load from local config)
    ms_creds = {
        "tenant_id": "YOUR_AZURE_TENANT_ID",
        "client_id": "YOUR_AZURE_CLIENT_ID",
        "client_secret": "YOUR_AZURE_CLIENT_SECRET"
    }

    gmail_json = "credentials.json" # Path to downloaded OAuth2
credentials

        self.worker = SyncWorker(
            project_name=self.txt_project.text(),
            organization=self.txt_org.text(),
            ms_credentials=ms_creds,
            gmail_credentials_file=gmail_json if os.path.exists(gmail_json)
else None
        )
        self.worker.log_signal.connect(self.append_log)
        self.worker.finished_signal.connect(self.on_sync_finished)
        self.worker.start()

def on_sync_finished(self):
    self.btn_sync.setEnabled(True)
    self.refresh_table()

def refresh_table(self):
    """Reloads records directly from SQLite."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("SELECT id, account_email, subject, sender,
file_path, extra_metadata FROM emails ORDER BY id DESC")

```

```
rows = cursor.fetchall()
conn.close()

self.table.setRowCount(0)
for row_idx, row_data in enumerate(rows):
    self.table.insertRow(row_idx)
    for col_idx, value in enumerate(row_data):
        item = QTableWidgetItem(str(value))
        self.table.setItem(row_idx, col_idx, item)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

5. Execution

Launch the application directly through **uv**:

```
uv run main.py
```

Application Features

1. **Multi-Provider Fetching:** Seamlessly downloads **.eml** MIME streams from both Microsoft Graph (**@oxnee.com**, **@live.com.au**) and Google API (**@gmail.com**).
2. **Metadata Flexibility:** Extracted records save their file path alongside a **JSON** blob column in SQLite (**extra_metadata**). You can attach unlimited custom project fields or tag attributes without modifying database tables.
3. **PySide6 Interface:** Offloads heavy network I/O to background threads (**QThread**) so the UI remains fluid, giving you instant table views of your archived dataset.

Prompt

What structure would you suggest for this project, prefer GUI pages or tabs are in different files, draw diagram of the proposed structure

Response

For **144-python-email-extractor**, structuring the application into clear, decoupled modules makes it significantly easier to maintain, test, and expand as you add more metadata fields or email accounts.

Separating UI pages/tabs into distinct files and isolating the database/API services keeps the PySide6 widgets clean and focused strictly on display logic.

Suggested Directory Structure

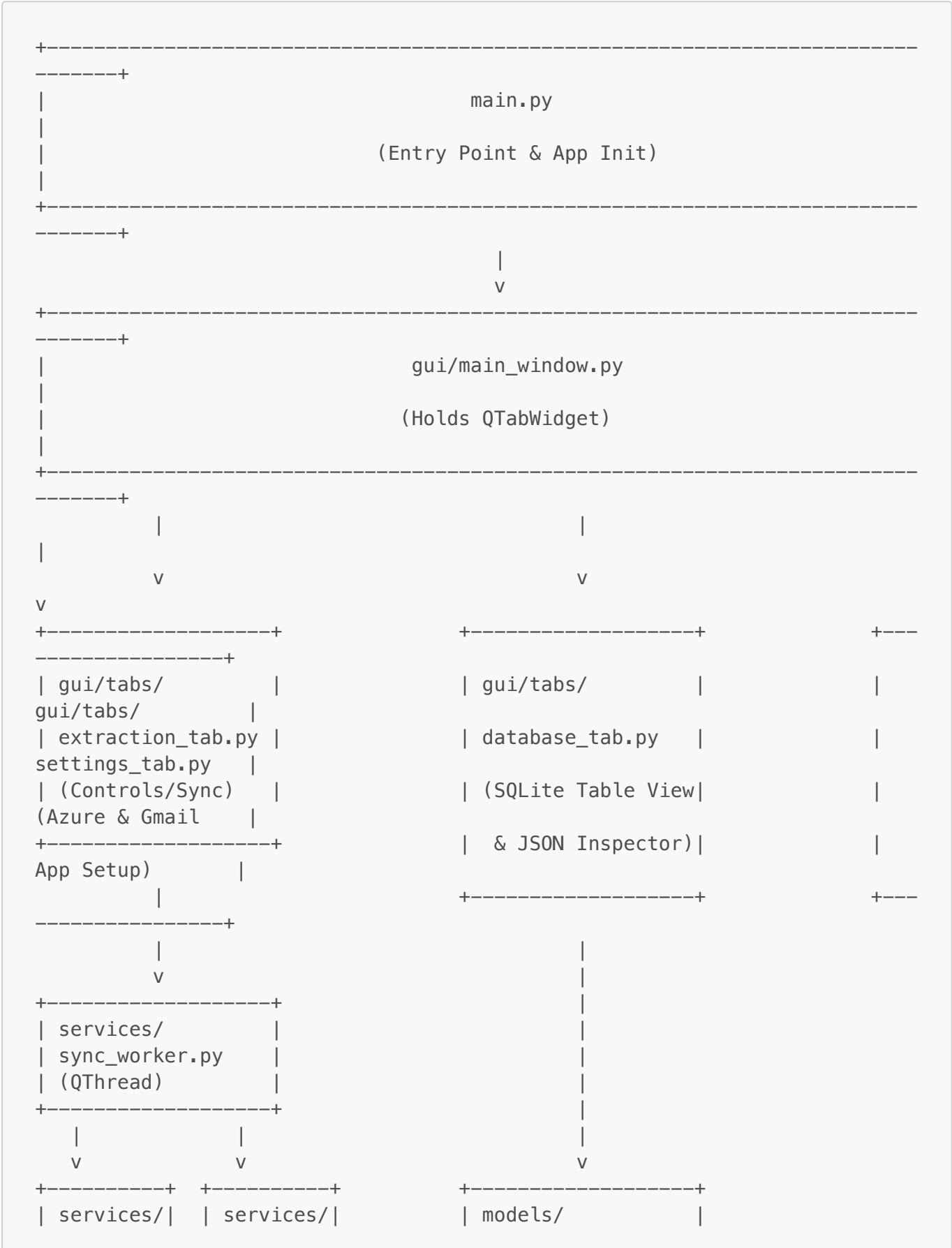
```

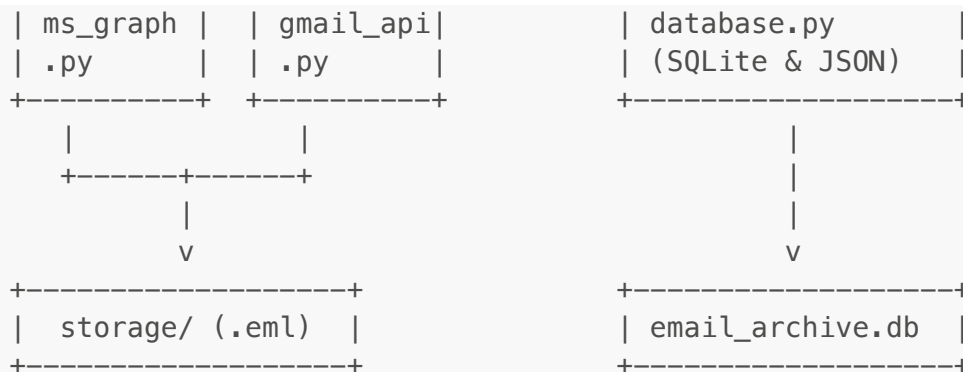
144-python-email-extractor/
├── pyproject.toml           # Managed by uv
├── uv.lock
├── README.md
├── email_archive.db         # Generated SQLite database
├── credentials.json        # Google OAuth2 credentials (git-ignored)
├── token_gmail.json        # Generated Google session token (git-
ignored)
├── storage/                # Output directory for raw .eml files
│   └── .gitkeep
├── src/                    # Core application source code
│   ├── __init__.py
│   └── main.py             # Application entry point (QApplication
setup)
│   ├── models/            # Database & schema handlers
│   │   ├── __init__.py
│   │   └── database.py    # SQLite setup, JSON metadata queries &
schema
│   ├── services/          # Downloader engines & background workers
│   │   ├── __init__.py
│   │   ├── ms_graph.py    # Microsoft Graph API client (.eml stream)
│   │   └── gmail_api.py   # Google Gmail API client (RFC 2822 raw
fetch)
│   └── sync_worker.py     # QThread runner for background network
calls
│   └── gui/               # PySide6 User Interface components
│       ├── __init__.py
│       └── main_window.py # QMainWindow containing QTabWidget
container
│   ├── components/       # Shared UI widgets
│   │   ├── __init__.py
│   │   └── log_viewer.py  # Live log output widget
│   └── tabs/              # Modular tab pages (separated files)
│       ├── __init__.py
│       └── extraction_tab.py # Tab 1: Sync controls, parameters &
progress
│       ├── database_tab.py # Tab 2: SQLite data table & JSON
inspector
│       └── settings_tab.py # Tab 3: API Keys, Azure App IDs & user

```

accounts

Component Relationship Diagram





How the Tab Files Work in Code

1. Individual Tab File (`src/gui/tabs/extraction_tab.py`)

Each page inherits from `QWidget` and encapsulates its own layout and signals:

```

from PySide6.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout, QGroupBox,
QLineEdit, QPushButton, QLabel

class ExtractionTab(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout(self)

        # Configuration Box
        config_box = QGroupBox("Extraction Parameters")
        cfg_layout = QHBoxLayout(config_box)

        self.txt_project = QLineEdit("Project_144")
        self.txt_org = QLineEdit("0xnee")
        self.btn_sync = QPushButton("Start Extraction")

        cfg_layout.addWidget(QLabel("Project:"))
        cfg_layout.addWidget(self.txt_project)
        cfg_layout.addWidget(QLabel("Organization:"))
        cfg_layout.addWidget(self.txt_org)
        cfg_layout.addWidget(self.btn_sync)

        layout.addWidget(config_box)
        layout.addStretch()

```

2. Tab Container (`src/gui/main_window.py`)

MainWindow simply imports the tabs and adds them to a **QTabWidget**:

```
from PySide6.QtWidgets import QMainWindow, QTabWidget
from gui.tabs.extraction_tab import ExtractionTab
from gui.tabs.database_tab import DatabaseTab
from gui.tabs.settings_tab import SettingsTab

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("144-python-email-extractor")
        self.resize(1000, 750)

        # Tab Widget Container
        self.tabs = QTabWidget()
        self.setCentralWidget(self.tabs)

        # Instantiate modular tab pages
        self.tab_extraction = ExtractionTab()
        self.tab_database = DatabaseTab()
        self.tab_settings = SettingsTab()

        # Add tabs
        self.tabs.addTab(self.tab_extraction, "Extract & Sync")
        self.tabs.addTab(self.tab_database, "Database Viewer")
        self.tabs.addTab(self.tab_settings, "Account Settings")
```

Benefits of This Architecture

1. **Clean uv Integration:** Running `uv run src/main.py` directly executes the clean modular code tree.
2. **Tab Isolation:** Editing the UI for the Database Inspector in `database_tab.py` won't touch or risk breaking the extraction runner code in `extraction_tab.py`.
3. **Decoupled Business Logic:** `ms_graph.py` and `gmail_api.py` have no knowledge of PySide6—they can easily be called from a CLI or automated script independently of the GUI.

Prompt

This is fine with addition of another folder called `doc`, which has a subfolder called `images`. The project has created and will use Forgejo and git

Response

Here is the updated architecture for **144-python-email-extractor**, incorporating the **doc/images** documentation directory and integrating **Forgejo / Git** version control conventions (such as **.gitignore**

for sensitive tokens and raw .eml storage).

Project Directory Structure

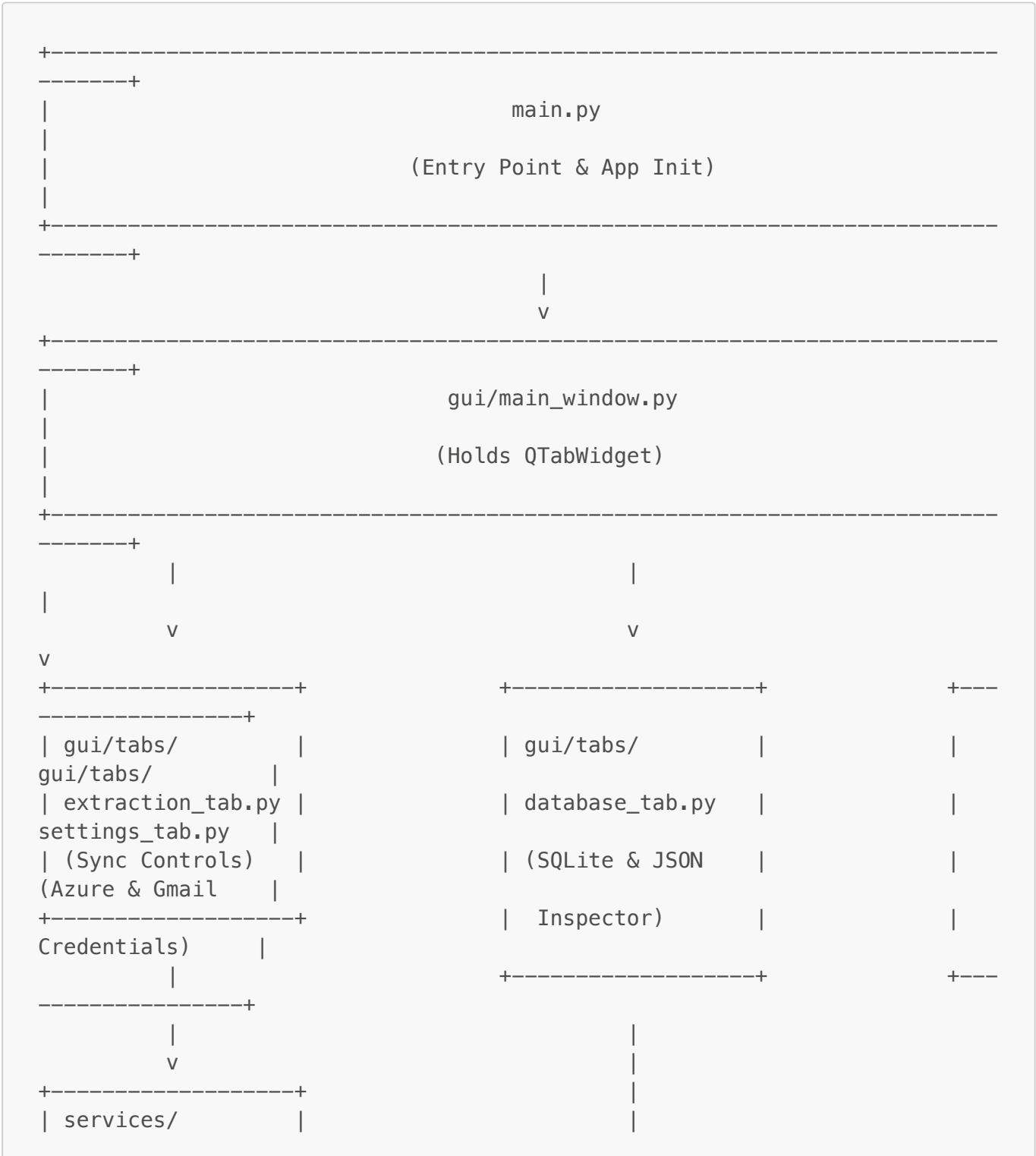
```

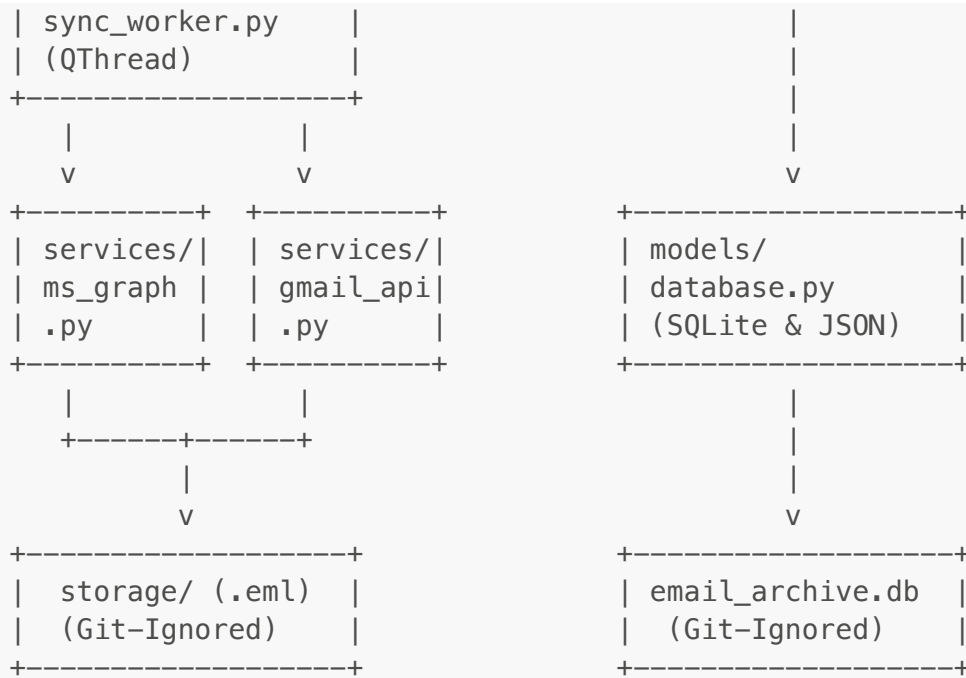
144-python-email-extractor/
├── .gitignore                # Ignores storage/, *.db, tokens, and
├── virtualenvs
├── .forgejo/                # Forgejo Actions / CI/CD workflows
(optional)
├── └── workflows/
│       └── build.yaml
├── pyproject.toml           # Managed by uv
├── uv.lock
├── README.md
├── doc/                     # Project documentation
│   ├── ARCHITECTURE.md     # Data flow & API integration details
│   └── images/             # GUI screenshots, architecture diagrams &
visual assets
│   └── .gitkeep
├── storage/                 # Output directory for raw .eml files (git-
ignored)
│   └── .gitkeep
├── src/                     # Core Python application package
│   ├── __init__.py
│   └── main.py              # Application entry point (QApplication
initialization)
│   ├── models/             # Database & schema handlers
│   │   ├── __init__.py
│   │   └── database.py     # SQLite connection, JSON metadata schema &
queries
│   ├── services/           # API clients & background thread workers
│   │   ├── __init__.py
│   │   └── ms_graph.py     # Microsoft Graph API client (RFC 2822
/.value stream)
│   ├── gmail_api.py        # Google Gmail API client (base64url raw
MIME)
│   └── sync_worker.py      # PySide6 QThread runner for non-blocking
API calls
│   └── gui/                # PySide6 Desktop UI modules
│       ├── __init__.py
│       └── main_window.py  # QMainWindow holding the QTabWidget
container
│   ├── components/        # Reusable UI widgets
│   │   ├── __init__.py
│   │   └── log_viewer.py  # Live log output widget

```

```
└─ tabs/                                # Isolated GUI pages
    ├── __init__.py
    └── extraction_tab.py # Tab 1: Sync controls & live progress
logs
    ├── database_tab.py # Tab 2: SQLite database viewer & JSON
metadata inspector
    └── settings_tab.py # Tab 3: Azure App & Google API
credential configuration
```

Component Relationship Diagram





Git & Forgejo Configuration

Since `.eml` downloads contain private email data and OAuth tokens grant access to live accounts, create a `.gitignore` file at the root to keep your Forgejo repository clean and secure:

```

# Byte-compiled / Python
__pycache__/
*.py[cod]
*$py.class
.venv/

# Environment & Dependencies
.uv/

# Local Storage & Databases
storage/*.eml
!storage/.gitkeep
*.db
*.sqlite3

# API Credentials & Tokens
credentials.json
token_gmail.json
.env

# System / macOS
.DS_Store
  
```

Initializing Git & Connecting to Forgejo

1. **Initialize Git and directory structure:** Terminal. Run the following in your terminal to create the directory skeleton and initialize Git:

```
mkdir -p 144-python-email-extractor/{doc/images,storage,src/{models,services,gui/{components,tabs}}}
touch 144-python-email-extractor/{storage,doc/images}/.gitkeep
cd 144-python-email-extractor
git init
```

2. **Initialize project with uv:** uv. Initialize the Python environment and add dependencies:

```
uv init
uv add PySide6 msal requests google-api-python-client google-auth-oauthlib
google-auth-httpplib2
```

3. **Link to Forgejo repository:** Forgejo. Add your Forgejo origin and push the initial commit:

```
git add .
git commit -m "Initial commit: 144-python-email-extractor project structure"
git remote add origin https://your-forgejo-server/username/144-python-email-extractor.git
git branch -M main
git push -u origin main
```

Prompt

generate the complete set of modular Python files across src/ based on this structure

Response

Here is the complete modular Python implementation for **144-python-email-extractor**. Every file is designed to be fully self-contained and placed inside your **src/** directory structure.

File 1: **src/models/database.py**

Handles the SQLite database connection, table initialization, and saving/fetching records with the flexible JSON metadata blob.

```
import sqlite3
import json
import os

DB_FILE = "email_archive.db"

def init_db(db_path=DB_FILE):
    """Initializes the SQLite database schema if it doesn't exist."""
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()

    cursor.execute("""
CREATE TABLE IF NOT EXISTS emails (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    account_email TEXT NOT NULL,
    message_id TEXT NOT NULL UNIQUE,
    subject TEXT,
    sender TEXT,
    received_date TEXT,
    file_path TEXT NOT NULL,
    extra_metadata TEXT, -- JSON blob for flexible fields (project,
org, tags, etc.)
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
)
""")
    conn.commit()
    conn.close()

def save_email_record(account_email, message_id, subject, sender,
received_date, file_path, metadata_dict, db_path=DB_FILE):
    """Inserts or updates an email record with extra JSON metadata."""
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()

    json_metadata = json.dumps(metadata_dict, indent=2)

    cursor.execute("""
INSERT INTO emails (account_email, message_id, subject, sender,
received_date, file_path, extra_metadata)
VALUES (?, ?, ?, ?, ?, ?, ?)
ON CONFLICT(message_id) DO UPDATE SET
    extra_metadata = excluded.extra_metadata,
    file_path = excluded.file_path,
    subject = excluded.subject
""", (account_email, message_id, subject, sender, received_date,
file_path, json_metadata))

    conn.commit()
    conn.close()

def fetch_all_emails(db_path=DB_FILE):
    """Retrieves all indexed email records ordered by newest ID."""
```

```
conn = sqlite3.connect(db_path)
cursor = conn.cursor()
cursor.execute("""
    SELECT id, account_email, subject, sender, received_date,
file_path, extra_metadata
    FROM emails ORDER BY id DESC
""")
rows = cursor.fetchall()
conn.close()
return rows
```

File 2: `src/services/ms_graph.py`

Handles Microsoft Graph authentication via MSAL and downloads emails as native RFC 822 `.eml` files using the `/$value` endpoint.

```
import os
import re
import requests
import msal

STORAGE_DIR = os.path.abspath("./storage")

def sanitize_filename(name):
    clean = re.sub(r'[\\"/*?:"<>|]', "", name or "Untitled")
    return clean[:80].strip()

class MSGraphDownloader:
    def __init__(self, tenant_id, client_id, client_secret, account_email):
        self.account_email = account_email
        self.tenant_id = tenant_id
        self.client_id = client_id
        self.client_secret = client_secret
        self.token = self._get_token()

    def _get_token(self):
        if not self.client_id or not self.client_secret or not
self.tenant_id:
            return None

        app = msal.ConfidentialClientApplication(
            self.client_id,
            authority=f"https://login.microsoftonline.com/{self.tenant_id}",
            client_credential=self.client_secret,
        )
        res = app.acquire_token_for_client(scopes=
["https://graph.microsoft.com/.default"])
        return res.get("access_token")
```

```

def download_inbox(self, callback_signal):
    if not self.token:
        callback_signal.emit(f"[{self.account_email}] Auth Error:
Missing or invalid MS Azure credentials.")
        return []

    headers = {"Authorization": f"Bearer {self.token}", "Accept":
"application/json"}
    url =
f"https://graph.microsoft.com/v1.0/users/{self.account_email}/mailFolders/i
nbox/messages?$select=id,subject,from,receivedDateTime&$top=20"

    extracted = []
    os.makedirs(STORAGE_DIR, exist_ok=True)

    res = requests.get(url, headers=headers)
    if res.status_code != 200:
        callback_signal.emit(f"[{self.account_email}] Failed to fetch
messages: Status {res.status_code}")
        return []

    messages = res.json().get("value", [])
    for msg in messages:
        msg_id = msg["id"]
        subject = msg.get("subject", "No Subject")
        sender = msg.get("from", {}).get("emailAddress",
{}).get("address", "Unknown")
        date_str = msg.get("receivedDateTime", "")[:10]

        # Request raw MIME payload
        raw_url =
f"https://graph.microsoft.com/v1.0/users/{self.account_email}/messages/{msg
_id}/$value"
        eml_res = requests.get(raw_url, headers=headers)

        if eml_res.status_code == 200:
            fname =
f"MS_{date_str}_{sanitize_filename(subject)}_{msg_id[-6:]} .eml"
            fpath = os.path.abspath(os.path.join(STORAGE_DIR, fname))

            with open(fpath, "wb") as f:
                f.write(eml_res.content)

            extracted.append({
                "account": self.account_email,
                "id": msg_id,
                "subject": subject,
                "sender": sender,
                "date": date_str,
                "path": fpath
            })
            callback_signal.emit(f"[{self.account_email}] Downloaded:
{fname}")

```

```

        else:
            callback_signal.emit(f"[{self.account_email}] Failed raw
fetch for {msg_id[-6:]}")

        return extracted

```

File 3: `src/services/gmail_api.py`

Connects to the Google Gmail API using OAuth2 credentials and converts base64url MIME responses into `.eml` files.

```

import os
import re
import base64
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials

STORAGE_DIR = os.path.abspath("./storage")

def sanitize_filename(name):
    clean = re.sub(r'[\\"/*?:"<>|]', "", name or "Untitled")
    return clean[:80].strip()

class GmailDownloader:
    SCOPES = ['https://www.googleapis.com/auth/gmail.readonly']

    def __init__(self, credentials_json="credentials.json",
account_email="slohning@gmail.com"):
        self.account_email = account_email
        self.creds_json = credentials_json
        self.service = self._authenticate()

    def _authenticate(self):
        if not os.path.exists(self.creds_json):
            return None

        creds = None
        token_file = 'token_gmail.json'
        if os.path.exists(token_file):
            creds = Credentials.from_authorized_user_file(token_file,
self.SCOPES)
            if not creds or not creds.valid:
                if creds and creds.expired and creds.refresh_token:
                    creds.refresh(Request())
                else:
                    flow =
InstalledAppFlow.from_client_secrets_file(self.creds_json, self.SCOPES)

```

```

        creds = flow.run_local_server(port=0)
        with open(token_file, 'w') as token:
            token.write(creds.to_json())
        return build('gmail', 'v1', credentials=creds)

    def download_inbox(self, callback_signal):
        if not self.service:
            callback_signal.emit(f"[{self.account_email}] Auth Error: Missing {self.creds_json} configuration.")
            return []

        results = self.service.users().messages().list(userId='me',
maxResults=20, q="label:INBOX").execute()
        messages = results.get('messages', [])

        extracted = []
        os.makedirs(STORAGE_DIR, exist_ok=True)

        for m in messages:
            msg_id = m['id']
            # Fetch raw RFC 822 format
            raw_msg = self.service.users().messages().get(userId='me',
id=msg_id, format='raw').execute()

            meta_msg = self.service.users().messages().get(userId='me',
id=msg_id, format='full').execute()
            headers = meta_msg.get('payload', {}).get('headers', [])

            subject = next((h['value'] for h in headers if
h['name'].lower() == 'subject'), 'No Subject')
            sender = next((h['value'] for h in headers if h['name'].lower()
== 'from'), 'Unknown')
            date_str = next((h['value'] for h in headers if
h['name'].lower() == 'date'), 'UnknownDate')

            eml_data =
base64.urlsafe_b64decode(raw_msg['raw'].encode('ASCII'))

            fname = f"GMAIL_{sanitize_filename(subject)}_{msg_id[-6:]}_eml"
            fpath = os.path.abspath(os.path.join(STORAGE_DIR, fname))

            with open(fpath, "wb") as f:
                f.write(eml_data)

            extracted.append({
                "account": self.account_email,
                "id": msg_id,
                "subject": subject,
                "sender": sender,
                "date": date_str,
                "path": fpath
            })
            callback_signal.emit(f"[{self.account_email}] Downloaded: {fname}")

```

```
return extracted
```

File 4: `src/services/sync_worker.py`

Runs the asynchronous downloading and database indexing tasks in a dedicated `QThread` to prevent PySide6 GUI freezes.

```
from PySide6.QtCore import QThread, Signal
from models.database import init_db, save_email_record
from services.ms_graph import MSGraphDownloader
from services.gmail_api import GmailDownloader

class SyncWorker(QThread):
    log_signal = Signal(str)
    finished_signal = Signal()

    def __init__(self, project_name, organization, ms_credentials,
gmail_credentials_file):
        super().__init__()
        self.project_name = project_name
        self.organization = organization
        self.ms_credentials = ms_credentials
        self.gmail_credentials_file = gmail_credentials_file

    def run(self):
        self.log_signal.emit("Initializing database and network
drivers...")
        init_db()

        records = []

        # 1. Sync Microsoft Accounts
        ms_accounts = ["stephen.lohning@oxnee.com", "slohning@live.com.au"]
        for email in ms_accounts:
            self.log_signal.emit(f"Connecting to Microsoft Graph for
{email}...")
            ms = MSGraphDownloader(
                tenant_id=self.ms_credentials.get("tenant_id"),
                client_id=self.ms_credentials.get("client_id"),
                client_secret=self.ms_credentials.get("client_secret"),
                account_email=email
            )
            records.extend(ms.download_inbox(self.log_signal))

        # 2. Sync Gmail Account
        self.log_signal.emit("Connecting to Google Gmail API for
slohning@gmail.com...")
        try:
```

```

        gm = GmailDownloader(self.gmail_credentials_file,
                              "slohning@gmail.com")
        records.extend(gm.download_inbox(self.log_signal))
    except Exception as e:
        self.log_signal.emit(f"[slohning@gmail.com] Sync Error: {e}")

    # 3. Save Records and metadata JSON into SQLite
    self.log_signal.emit(f"Indexing {len(records)} files into SQLite
database...")
    for r in records:
        extra_meta = {
            "project_name": self.project_name,
            "organization": self.organization,
            "app_name": "144-python-email-extractor",
            "version": "1.0.0"
        }
        save_email_record(
            account_email=r["account"],
            message_id=r["id"],
            subject=r["subject"],
            sender=r["sender"],
            received_date=r["date"],
            file_path=r["path"],
            metadata_dict=extra_meta
        )

    self.log_signal.emit("\nExtraction and database indexing finished
successfully!")
    self.finished_signal.emit()

```

File 5: `src/gui/components/log_viewer.py`

A reusable logging component used across GUI pages to output streaming log output.

```

from PySide6.QtWidgets import QWidget, QVBoxLayout, QLabel, QTextEdit

class LogViewer(QWidget):
    def __init__(self, title="Live Extraction Logs", parent=None):
        super().__init__(parent)
        layout = QVBoxLayout(self)
        layout.setContentsMargins(0, 0, 0, 0)

        self.lbl_title = QLabel(title)
        self.txt_log = QTextEdit()
        self.txt_log.setReadOnly(True)
        self.txt_log.setStyleSheet("font-family: monospace; font-size:
11px;")

        layout.addWidget(self.lbl_title)

```

```
        layout.addWidget(self.txt_log)

    def append_log(self, message):
        self.txt_log.append(message)

    def clear_logs(self):
        self.txt_log.clear()
```

File 6: `src/gui/tabs/extraction_tab.py`

Tab 1 page: Provides inputs for project metadata, extraction parameters, and sync status logs.

```
from PySide6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QGroupBox, QLineEdit, QPushButton,
    QLabel
)
from gui.components.log_viewer import LogViewer

class ExtractionTab(QWidget):
    def __init__(self, start_sync_callback, parent=None):
        super().__init__(parent)
        self.start_sync_callback = start_sync_callback
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout(self)

        # Metadata Configuration Parameters
        config_box = QGroupBox("Extraction Metadata Setup")
        cfg_layout = QHBoxLayout(config_box)

        self.txt_project = QLineEdit("Project_144")
        self.txt_project.setPlaceholderText("Project Name")
        self.txt_org = QLineEdit("0xnee")
        self.txt_org.setPlaceholderText("Organization")

        self.btn_sync = QPushButton("Start Download & Extraction")
        self.btn_sync.clicked.connect(self.on_sync_clicked)

        cfg_layout.addWidget(QLabel("Project:"))
        cfg_layout.addWidget(self.txt_project)
        cfg_layout.addWidget(QLabel("Organization:"))
        cfg_layout.addWidget(self.txt_org)
        cfg_layout.addWidget(self.btn_sync)

        layout.addWidget(config_box)

        # Log Viewer Component
        self.log_viewer = LogViewer()
```

```

        layout.addWidget(self.log_viewer)

    def on_sync_clicked(self):
        proj = self.txt_project.text().strip()
        org = self.txt_org.text().strip()
        self.start_sync_callback(proj, org)

    def set_syncing_state(self, is_syncing):
        self.btn_sync.setEnabled(not is_syncing)

```

File 7: `src/gui/tabs/database_tab.py`

Tab 2 page: Offers an interactive SQLite view and inspection panel for saved `.eml` files and JSON metadata blobs.

```

from PySide6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QTableWidgetItem,
    QHeaderView, QLabel, QTextEdit, QSplitter, QPushButton
)
from PySide6.QtCore import Qt
from models.database import fetch_all_emails

class DatabaseTab(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout(self)

        # Top Control Bar
        top_bar = QHBoxLayout()
        self.btn_refresh = QPushButton("Refresh Table")
        self.btn_refresh.clicked.connect(self.refresh_table)
        top_bar.addWidget(QLabel("SQLite Indexed Records:"))
        top_bar.addStretch()
        top_bar.addWidget(self.btn_refresh)
        layout.addLayout(top_bar)

        # Main Splitter (Table on top/left, JSON detail on bottom/right)
        splitter = QSplitter(Qt.Vertical)

        self.table = QTableWidgetItem()
        self.table.setColumnCount(6)
        self.table.setHorizontalHeaderLabels(["ID", "Account", "Subject",
        "Sender", "Date", "File Path"])

        self.table.horizontalHeader().setSectionResizeMode(QHeaderView.ResizeToContents)

```

```

        self.table.horizontalHeader().setSectionResizeMode(2,
QHeaderView.Stretch)
        self.table.itemSelectionChanged.connect(self.on_row_selected)
        splitter.addWidget(self.table)

        # JSON Inspector Box
        json_box = QWidget()
        box_layout = QVBoxLayout(json_box)
        box_layout.setContentsMargins(0, 0, 0, 0)
        box_layout.addWidget(QLabel("JSON Metadata Preview:"))
        self.txt_json = QTextEdit()
        self.txt_json.setReadOnly(True)
        self.txt_json.setStyleSheet("font-family: monospace;")
        box_layout.addWidget(self.txt_json)

        splitter.addWidget(json_box)
        splitter.setSizes([350, 150])

        layout.addWidget(splitter)
        self.refresh_table()

    def refresh_table(self):
        rows = fetch_all_emails()
        self.table.setRowCount(0)
        self.raw_rows = rows # Keep full data in memory for JSON preview

        for row_idx, row_data in enumerate(rows):
            self.table.insertRow(row_idx)
            # Render ID, Account, Subject, Sender, Date, Path
            for col_idx in range(6):
                val = str(row_data[col_idx])
                item = QTableWidgetItem(val)
                self.table.setItem(row_idx, col_idx, item)

    def on_row_selected(self):
        curr_row = self.table.currentRow()
        if curr_row >= 0 and curr_row < len(self.raw_rows):
            # Extra JSON metadata is at index 6 in the tuple
            json_blob = self.raw_rows[curr_row][6]
            self.txt_json.setText(json_blob or "{}")

```

File 8: `src/gui/tabs/settings_tab.py`

Tab 3 page: Configuration management for Azure App secrets and Google API credentials.

```

from PySide6.QtWidgets import (
    QWidget, QVBoxLayout, QFormLayout, QGroupBox, QLineEdit, QLabel
)

```

```
class SettingsTab(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout(self)

        # Microsoft Azure Entra ID Settings
        ms_box = QGroupBox("Microsoft Graph API Credentials (Oxnee & Live.com)")
        ms_layout = QFormLayout(ms_box)

        self.txt_tenant = QLineEdit("YOUR_TENANT_ID")
        self.txt_client_id = QLineEdit("YOUR_CLIENT_ID")
        self.txt_secret = QLineEdit("YOUR_CLIENT_SECRET")
        self.txt_secret.setEchoMode(QLineEdit.Password)

        ms_layout.addRow(QLabel("Tenant ID:"), self.txt_tenant)
        ms_layout.addRow(QLabel("Client ID:"), self.txt_client_id)
        ms_layout.addRow(QLabel("Client Secret:"), self.txt_secret)
        layout.addWidget(ms_box)

        # Google API Settings
        gmail_box = QGroupBox("Google Gmail API Credentials (Gmail)")
        gmail_layout = QFormLayout(gmail_box)

        self.txt_gmail_creds = QLineEdit("credentials.json")
        gmail_layout.addRow(QLabel("OAuth Credentials File:"),
self.txt_gmail_creds)
        layout.addWidget(gmail_box)

        layout.addStretch()

    def get_ms_credentials(self):
        return {
            "tenant_id": self.txt_tenant.text().strip(),
            "client_id": self.txt_client_id.text().strip(),
            "client_secret": self.txt_secret.text().strip()
        }

    def get_gmail_credentials_file(self):
        return self.txt_gmail_creds.text().strip()
```

File 9: `src/gui/main_window.py`

The main application window that manages `QTabWidget` tabs and ties together worker thread execution.

```
from PySide6.QtWidgets import QMainWindow, QTabWidget
from gui.tabs.extraction_tab import ExtractionTab
from gui.tabs.database_tab import DatabaseTab
from gui.tabs.settings_tab import SettingsTab
from services.sync_worker import SyncWorker

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("144-python-email-extractor")
        self.resize(1000, 700)

        # Tab Widget Container
        self.tabs = QTabWidget()
        self.setCentralWidget(self.tabs)

        # Instantiate Tabs
        self.tab_extraction =
ExtractionTab(start_sync_callback=self.run_sync_process)
        self.tab_database = DatabaseTab()
        self.tab_settings = SettingsTab()

        # Add pages to Tab Control
        self.tabs.addTab(self.tab_extraction, "Extract & Sync")
        self.tabs.addTab(self.tab_database, "Database Inspector")
        self.tabs.addTab(self.tab_settings, "Account Settings")

    def run_sync_process(self, project_name, organization):
        self.tab_extraction.set_syncing_state(True)
        self.tab_extraction.log_viewer.clear_logs()

        ms_creds = self.tab_settings.get_ms_credentials()
        gmail_file = self.tab_settings.get_gmail_credentials_file()

        # Launch non-blocking QThread worker
        self.worker = SyncWorker(
            project_name=project_name,
            organization=organization,
            ms_credentials=ms_creds,
            gmail_credentials_file=gmail_file
        )

        self.worker.log_signal.connect(self.tab_extraction.log_viewer.append_log)
        self.worker.finished_signal.connect(self.on_sync_complete)
        self.worker.start()

    def on_sync_complete(self):
        self.tab_extraction.set_syncing_state(False)
        self.tab_database.refresh_table()
```

File 10: `src/main.py`

Application entry point. Initializes `QApplication` and launches the UI window.

```
import sys
import os

# Ensure src path resolution when running via uv
sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))

from PySide6.QtWidgets import QApplication
from gui.main_window import MainWindow
from models.database import init_db

def main():
    init_db() # Ensure database schema is ready
    app = QApplication(sys.argv)

    window = MainWindow()
    window.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

Running the Complete Application

Launch the application using `uv`:

```
uv run src/main.py
```

Prompt

when I ran the command `uv run src/main.py` Reponse Traceback (most recent call last): File `"/Users/stephenlohning/Documents/01_Projects/144-python-email-extractor/src/main.py"`, line 10, in `from gui.main_window import MainWindow` File `"/Users/stephenlohning/Documents/01_Projects/144-python-email-extractor/src/gui/main_window.py"`, line 6, in `from gui.tabs.settings_tab import SettingsTab` ModuleNotFoundError: No module named 'gui.tabs.settings_tab' stephenlohning@Scotty 144-python-email-extractor %

Response

The `ModuleNotFoundError` happens because Python treats imports differently depending on whether `src/` is in the module search path before the submodules try to resolve `gui`.

When `main.py` executes `from gui.main_window import MainWindow`, Python loads `gui/main_window.py`. But inside `main_window.py`, the import `from gui.tabs.settings_tab import SettingsTab` attempts an absolute import relative to `src/`. If Python's `sys.path` hasn't registered `src/` as a root package path prior to evaluating `gui`, relative/absolute package lookup fails.

Here are two quick ways to fix it:

Solution 1: Use Relative Imports Inside the `gui` Package (Recommended)

Update `src/gui/main_window.py` to use explicit relative imports so the package resolves correctly regardless of how `main.py` is called.

In `src/gui/main_window.py`, change lines 2–5 to:

```
from PySide6.QtWidgets import QMainWindow, QTabWidget
from .tabs.extraction_tab import ExtractionTab
from .tabs.database_tab import DatabaseTab
from .tabs.settings_tab import SettingsTab
from services.sync_worker import SyncWorker
```

Also check `src/gui/tabs/extraction_tab.py` and `src/gui/tabs/database_tab.py`:

- In `extraction_tab.py`, update:

```
from ..components.log_viewer import LogViewer
```

Solution 2: Standardize `PYTHONPATH` with `uv`

Alternatively, instruct `uv` to include `src` on the Python path automatically when running commands.

Add or update the following block in your `pyproject.toml` at the project root:

```
[tool.pytest.ini_options]
pythonpath = ["src"]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[tool.hatch.build.targets.wheel]
```

```
packages = ["src"]
```

Or run directly with the `PYTHONPATH` environment variable set:

```
PYTHONPATH=src uv run src/main.py
```

Prompt

generate updated full file `src/gui/main_window.py`, `extraction_tab.py`, `src/gui/tabs/database_tab.py` add commented out full file e.g.

`src/gui/main_window.py`

Response

```
# src/gui/tabs/database_tab.py

from PySide6.QtWidgets import (
    QWidget,
    QVBoxLayout,
    QHBoxLayout,
    QTableWidgetItem,
    QLineEdit,
    QPushButton,
    QHeaderView,
)
from PySide6.QtCore import Qt

class DatabaseTab(QWidget):
    def __init__(self, db_manager, parent=None):
        super().__init__(parent)
        self.db_manager = db_manager

        self._init_ui()

    def _init_ui(self):
        layout = QVBoxLayout(self)

        # Search Bar Controls
        search_layout = QHBoxLayout()
```

```

        self.search_input = QLineEdit()
        self.search_input.setPlaceholderText("Search extracted records...")
        self.search_btn = QPushButton("Search")
        self.refresh_btn = QPushButton("Refresh")

        search_layout.addWidget(self.search_input)
        search_layout.addWidget(self.search_btn)
        search_layout.addWidget(self.refresh_btn)
        # Here are the updated files
with consistent relative imports so Python resolves package modules cleanly
without throwing `ModuleNotFoundError`.

### `src/gui/main_window.py`

```python
src/gui/main_window.py
import sys
from PySide6.QtWidgets import QMainWindow, QTabWidget, QVBoxLayout, QWidget
from .tabs.extraction_tab import ExtractionTab
from .tabs.database_tab import DatabaseTab
from .tabs.settings_tab import SettingsTab
from ..services.sync_worker import SyncWorker

class MainWindow(QMainWindow):
 def __init__(self, db_manager, config_manager):
 super().__init__()
 self.db_manager = db_manager
 self.config_manager = config_manager
 self.sync_worker = None

 self.setWindowTitle("Email Extractor")
 self.setMinimumSize(900, 600)

 self._init_ui()

 def _init_ui(self):
 central_widget = QWidget()
 self.setCentralWidget(central_widget)
 main_layout = QVBoxLayout(central_widget)

 self.tab_widget = QTabWidget()

 self.extraction_tab = ExtractionTab(self.db_manager,
self.config_manager)
 self.database_tab = DatabaseTab(self.db_manager)
 self.settings_tab = SettingsTab(self.config_manager)

 self.tab_widget.addTab(self.extraction_tab, "Extraction")
 self.tab_widget.addTab(self.database_tab, "Database View")
 self.tab_widget.addTab(self.settings_tab, "Settings")

 main_layout.addWidget(self.tab_widget)

```

## src/gui/tabs/extraction\_tab.py

```
src/gui/tabs/extraction_tab.py
from PySide6.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout,
QPushButton, QLabel, QProgressBar
from PySide6.QtCore import Qt, Signal
from ..components.log_viewer import LogViewer

class ExtractionTab(QWidget):
 start_extraction = Signal()
 stop_extraction = Signal()

 def __init__(self, db_manager, config_manager, parent=None):
 super().__init__(parent)
 self.db_manager = db_manager
 self.config_manager = config_manager

 self._init_ui()

 def _init_ui(self):
 layout = QVBoxLayout(self)

 # Control Panel
 control_layout = QHBoxLayout()
 self.btn_start = QPushButton("Start Sync / Extraction")
 self.btn_stop = QPushButton("Stop")
 self.btn_stop.setEnabled(False)

 control_layout.addWidget(self.btn_start)
 control_layout.addWidget(self.btn_stop)
 control_layout.addStretch()

 layout.addLayout(control_layout)

 # Progress Section
 self.status_label = QLabel("Status: Idle")
 self.progress_bar = QProgressBar()
 self.progress_bar.setRange(0, 100)
 self.progress_bar.setValue(0)

 layout.addWidget(self.status_label)
 layout.addWidget(self.progress_bar)

 # Log Output Viewer
 self.log_viewer = LogViewer()
 layout.addWidget(self.log_viewer)

 def update_progress(self, current: int, total: int, status_text: str =
 ""):
```

```
if total > 0:
 val = int((current / total) * 100)
 self.progress_bar.setValue(val)
if status_text:
 self.status_label.setText(f"Status: {status_text}")
```

---

## src/gui/tabs/database\_tab.py

```
src/gui/tabs/database_tab.py
from PySide6.QtWidgets import (
 QWidget, QVBoxLayout, QHBoxLayout, QTableView,
 QLineEdit, QPushButton, QLabel, QHeaderView
)
from PySide6.QtCore import Qt, QAbstractTableModel

class EmailTableModel(QAbstractTableModel):
 def __init__(self, data=None):
 super().__init__()
 self._data = data or []
 self._headers = ["ID", "Subject", "Sender", "Date Received",
"Status"]

 def rowCount(self, parent=None):
 return len(self._data)

 def columnCount(self, parent=None):
 return len(self._headers)

 def data(self, index, role=Qt.DisplayRole):
 if not index.isValid() or role != Qt.DisplayRole:
 return None
 row = index.row()
 col = index.column()
 return self._data[row][col] if row < len(self._data) and col <
len(self._data[row]) else None

 def headerData(self, section, orientation, role=Qt.DisplayRole):
 if orientation == Qt.Horizontal and role == Qt.DisplayRole:
 return self._headers[section]
 return None

 def update_data(self, new_data):
 self.beginResetModel()
 self._data = new_data
 self.endResetModel()

class DatabaseTab(QWidget):
```

```
def __init__(self, db_manager, parent=None):
 super().__init__(parent)
 self.db_manager = db_manager

 self._init_ui()

def _init_ui(self):
 layout = QVBoxLayout(self)

 # Search / Filter Bar
 filter_layout = QHBoxLayout()
 self.search_input = QLineEdit()
 self.search_input.setPlaceholderText("Filter emails by subject or sender...")
 self.btn_refresh = QPushButton("Refresh")

 filter_layout.addWidget(QLabel("Search:"))
 filter_layout.addWidget(self.search_input)
 filter_layout.addWidget(self.btn_refresh)

 layout.addLayout(filter_layout)

 # Table View
 self.table_view = QTableView()
 self.table_model = EmailTableModel()
 self.table_view.setModel(self.table_model)

 self.table_view.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)

 layout.addWidget(self.table_view)
```