Python Flask Manual: Structured Application with Microservice

Current Date: May 9, 2025

This manual will guide you through understanding, setting up, and running a Flask application based on the provided project structure. This structure involves a main application and a separate profile microservice.

0. Understanding the Project Structure

Before diving into the code, let's understand the purpose of each part of your project:

- `your_project_root/`: The main container for your entire project.
  - `.env`: Used to store environment-specific configurations like API keys, database URLs, or service endpoints. **Crucially, this file should be listed in your `.gitignore` file to prevent committing sensitive information to version control.**
  - `.gitignore`: Specifies intentionally untracked files that Git should ignore (e.g., `__pycache__/`, `*.pyc`, virtual environment folders, `.env` files).
  - `main_app/`: Contains the primary user-facing Flask application.
    - `run_main_app.py`: A simple script to initialize and run your main Flask application, often by calling an app factory.
    - `requirements.txt`: Lists all Python package dependencies for the `main_app` (e.g., Flask, requests).
    - `main_app_package/`: The actual Python package for your main application.
      - `__init__.py`: Marks the directory as a Python package. This is where you'll typically define your **application factory** (`create_app` function) and register blueprints.
      - `config.py`: Handles loading application configurations (e.g., from the `.env` file or environment variables).
      - `routes.py`: Defines the URL routes (e.g., `/`, `/login`, `/profile`) and the view functions that handle requests to these routes for the main application.
      - `auth.py`: Contains logic for user authentication (login, logout, registration, session management).
      - `services.py`: Includes functions that interact with other services, particularly your `profile_service`. This might involve making HTTP requests.
      - `models.py`: (Optional) Defines database models if your main application directly interacts with a database (e.g., using Flask-SQLAlchemy).
      - `static/`: Stores static assets like CSS, JavaScript, and images. Flask can serve these directly.

        - `css/`, `js/`, `images/`: Subdirectories for organizing static files.
    - `templates/`: Contains HTML templates (e.g., Jinja2 templates) that are rendered by your view functions.
        - `base.html`: A base template that other templates can inherit from, providing a common layout.
        - `header.html`, `footer.html`: Partial templates for common page elements, often included in `base.html` or other templates.
- `profile_service/`: A separate Flask application acting as a microservice, specifically for handling user profiles and profile pictures.
    - `run_profile_service.py`: Script to run the profile microservice.
    - `requirements.txt`: Python package dependencies for the `profile_service`.
    - `profile_pics/`: A directory where the `profile_service` might store uploaded profile images.

1. Prerequisites

- **Python:** Ensure Python (preferably version 3.7+) is installed.
- **pip:** The Python package installer (usually comes with Python).
- **Git:** For version control (essential for using `.gitignore`).

2. Setting up the Project Environment

It's highly recommended to use separate virtual environments for your `main_app` and `profile_service` to manage their dependencies independently.

Step 2.1: Create Project Root and Basic Files

1. Create the main project directory:

   Bash

   ```
   mkdir your_project_root
   cd your_project_root
   ```

2. Create the `.gitignore` file: `your_project_root/.gitignore`:

```
# Python
__pycache__/
*.pyc
*.pyo
*.pyd
*.egg-info/
dist/
build/
*.egg
*.whl
```

```
13.
14.  # Virtual environments
15.  venv/
16.  env/
17.  myflaskapp_env/ # Add your specific venv names if different
18.  profile_service_env/
19.
20.  # Environment variables
21.  .env
22.
23.  # IDE and OS specific
24.  .vscode/
25.  .idea/
26.  *.DS_Store
```

27. Create the (optional) `.env` file (example): `your_project_root/.env`:

```
28.  # Main App Configuration
29.  FLASK_APP_SECRET_KEY='your_strong_secret_key_here'
30.  FLASK_DEBUG=True
31.  PROFILE_SERVICE_URL='http://127.0.0.1:5001' # URL where
   profile_service will run
32.
33.  # Profile Service Configuration
34.  PROFILE_SERVICE_SECRET_KEY='another_strong_secret_key'
35.  PROFILE_SERVICE_DEBUG=True
```

Note: The actual loading of these variables will happen in `config.py`.

Step 2.2: Setting up the `main_app`

1. Navigate to where `main_app` will reside (it's directly under `your_project_root` in your structure):

   Bash

   ```
   # (If not already in your_project_root)
   # cd your_project_root
   mkdir main_app
   cd main_app
   ```

2. Create a virtual environment for `main_app`:

   Bash

   ```
   python -m venv main_app_env
   ```

3. Activate the `main_app` virtual environment:
   - Windows: `main_app_env\Scripts\activate`
   - macOS/Linux: `source main_app_env/bin/activate` Your prompt should change to indicate `(main_app_env)`.
4. Create `requirements.txt` for `main_app`: `main_app/requirements.txt`:

```
5. Flask>=2.0
```

```
6. python-dotenv  # For loading .env files
7. requests       # For calling the profile_service
8. # Add other dependencies like Flask-SQLAlchemy, Flask-WTF as needed
```

9. Install dependencies for `main_app`:

   Bash

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

10. Create the directory structure for `main_app_package`:

    Bash

    ```
    mkdir main_app_package
    cd main_app_package
    mkdir static
    mkdir static/css static/js static/images
    mkdir templates
    cd .. # Back to main_app directory
    ```

Step 2.3: Setting up the `profile_service`

1. Navigate out of `main_app` and create the `profile_service` directory:

   Bash

   ```
   cd .. # Back to your_project_root
   mkdir profile_service
   cd profile_service
   ```

2. Create a virtual environment for `profile_service`:

   Bash

   ```
   python -m venv profile_service_env
   ```

3. Activate the `profile_service` virtual environment:
   - Windows: `profile_service_env\Scripts\activate`
   - macOS/Linux: `source profile_service_env/bin/activate` Your prompt should change to indicate `(profile_service_env)`.
4. Create `requirements.txt` for `profile_service`:
   `profile_service/requirements.txt`:

```
5. Flask>=2.0
6. python-dotenv
7. # Add other dependencies if needed (e.g., Pillow for image processing)
```

8. Install dependencies for `profile_service`:

   Bash

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

9. Create the `profile_pics` directory:

   Bash

   ```
   mkdir profile_pics
   ```

10. Deactivate the `profile_service` environment for now if you're going back to work on `main_app`, or keep it active if you're writing its code next.

    Bash

    ```
    deactivate # Optional
    ```

3. Implementing the `main_app`

(Ensure `main_app_env` is activated for these steps)

Step 3.1: `main_app/main_app_package/config.py`

This file will load configurations. `main_app/main_app_package/config.py`:

Python

```
import os
from dotenv import load_dotenv

# Construct the path to the .env file located in the project root
# __file__ is 'your_project_root/main_app/main_app_package/config.py'
# os.path.dirname(__file__) is 'your_project_root/main_app/main_app_package/'
# os.path.dirname(os.path.dirname(os.path.dirname(__file__))) is
'your_project_root/'
dotenv_path =
os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))),
'.env')
load_dotenv(dotenv_path)

class Config:
    SECRET_KEY = os.environ.get('FLASK_APP_SECRET_KEY') or
'a_default_fallback_secret_key'
    DEBUG = os.environ.get('FLASK_DEBUG', 'False').lower() == 'true'
    PROFILE_SERVICE_URL = os.environ.get('PROFILE_SERVICE_URL')
    # Add other configurations like SQLALCHEMY_DATABASE_URI etc.
    # SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or
'sqlite:///site.db'
    # SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Step 3.2: `main_app/main_app_package/__init__.py` (App Factory)

This is where your Flask app instance is created and configured.
`main_app/main_app_package/__init__.py`:

Python
```
from flask import Flask
from .config import Config
# from flask_sqlalchemy import SQLAlchemy # Example if using SQLAlchemy
# from flask_login import LoginManager # Example if using Flask-Login

# db = SQLAlchemy() # Example
# login_manager = LoginManager() # Example
# login_manager.login_view = 'main_routes.login' # Example: redirect to login
if @login_required

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Initialize extensions (example)
    # db.init_app(app)
    # login_manager.init_app(app)

    # Import and register blueprints
    from .routes import main_bp  # Corrected to relative import for blueprint
    app.register_blueprint(main_bp)

    # If you have an auth blueprint:
    # from .auth import auth_bp
    # app.register_blueprint(auth_bp, url_prefix='/auth')

    # You can also register other blueprints here

    # Example context processor to make PROFILE_SERVICE_URL available in all
templates
    @app.context_processor
    def inject_profile_service_url():
        return
dict(PROFILE_SERVICE_URL=app.config.get('PROFILE_SERVICE_URL'))

    return app
```

Step 3.3: `main_app/main_app_package/routes.py`

Define your application's routes. `main_app/main_app_package/routes.py`:

Python
```
from flask import Blueprint, render_template, redirect, url_for, flash,
current_app, request
import requests # For calling the profile service
# from flask_login import login_required, current_user # Example

main_bp = Blueprint('main_routes', __name__) # 'main_routes' is the name of
the blueprint
```

```
@main_bp.route('/')
def home():
    return render_template('student_home.html', title='Home')

@main_bp.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Your login logic here (perhaps using main_app_package/auth.py)
        flash('Login functionality not yet implemented.', 'info')
        # return redirect(url_for('main_routes.home'))
    return render_template('login.html', title='Login')

@main_bp.route('/profile')
# @login_required # Example: protect this route
def profile_details():
    # Example: Fetch profile data from the profile_service
    # user_id = current_user.id # Assuming you have current_user from Flask-
Login
    user_id = "s12345678" # Hardcoded for example
    profile_service_url = current_app.config.get('PROFILE_SERVICE_URL')

    if not profile_service_url:
        flash('Profile service URL not configured.', 'danger')
        return render_template('profile_details.html', title='Profile',
profile_data=None, error=True)

    try:
        # The profile service should have an endpoint like /profile/<user_id>
        response = requests.get(f"{profile_service_url}/profile/{user_id}")
        response.raise_for_status() # Raise an exception for HTTP errors (4xx
or 5xx)
        profile_data = response.json()
    except requests.exceptions.RequestException as e:
        current_app.logger.error(f"Could not connect to profile service:
{e}")
        flash('Could not retrieve profile information at this time.',
'danger')
        profile_data = None
        error = True
    else:
        error = False

    return render_template('profile_details.html', title='Profile',
profile_data=profile_data, error=error)

# Add other routes as needed...
```

Step 3.4: `main_app/main_app_package/auth.py` (Placeholder)

main_app/main_app_package/auth.py:

Python

```
from flask import Blueprint, render_template, redirect, url_for, flash,
request
# from flask_login import login_user, logout_user, current_user
# from .models import User # Assuming you have a User model
# from . import db # Assuming you have db from __init__.py
```

```
auth_bp = Blueprint('auth', __name__, url_prefix='/auth')

@auth_bp.route('/register', methods=['GET', 'POST'])
def register():
    # Your registration logic
    return "Register Page (Not Implemented)"

@auth_bp.route('/logout')
def logout():
    # logout_user()
    flash('You have been logged out.', 'info')
    return redirect(url_for('main_routes.home'))

# Remember to register this blueprint in __init__.py if you use it.
```

If you use this, uncomment the registration in `__init__.py`.

Step 3.5: `main_app/main_app_package/services.py` (Illustrative) This file would contain more complex logic for interacting with microservices if simple `requests.get` in routes becomes too cumbersome. `main_app/main_app_package/services.py`:

Python
```
import requests
from flask import current_app, flash

def get_user_profile(user_id):
    profile_service_url = current_app.config.get('PROFILE_SERVICE_URL')
    if not profile_service_url:
        current_app.logger.error('Profile service URL not configured.')
        return None

    try:
        response = requests.get(f"{profile_service_url}/profile/{user_id}")
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        current_app.logger.error(f"Error fetching profile for {user_id}:
{e}")
        flash(f"Could not retrieve profile data for {user_id}.", 'warning')
        return None

# You would then call this function from your routes.py:
# from .services import get_user_profile
# profile_data = get_user_profile(user_id)
```

Step 3.6: `main_app/main_app_package/models.py` (Optional Placeholder)
`main_app/main_app_package/models.py`:

Python
```
# from . import db  # Assuming db = SQLAlchemy() from __init__.py
# from flask_login import UserMixin # For Flask-Login

# class User(UserMixin, db.Model):
```

```
#     id = db.Column(db.Integer, primary_key=True)
#     username = db.Column(db.String(20), unique=True, nullable=False)
#     email = db.Column(db.String(120), unique=True, nullable=False)
#     # Add other fields and relationships

#     def __repr__(self):
#         return f"User('{self.username}', '{self.email}')"
```

Step 3.7: Templates (`main_app/main_app_package/templates/`)

- `base.html`: `main_app/main_app_package/templates/base.html`:

  HTML

  ```
  <!DOCTYPE html>
  <html lang="en">
  <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-
  scale=1.0">
      <title>{% block title %}My Flask App{% endblock %}</title>
      <link rel="stylesheet" href="{{ url_for('static',
  filename='css/style.css') }}">
      {% block head_css %}{% endblock %}
  </head>
  <body>
      {% include 'header.html' %}

      <main>
          {% with messages = get_flashed_messages(with_categories=true)
  %}
              {% if messages %}
                  {% for category, message in messages %}
                      <div class="alert alert-{{ category }}">{{ message
  }}</div>
                  {% endfor %}
              {% endif %}
          {% endwith %}
          {% block content %}{% endblock %}
      </main>

      {% include 'footer.html' %}
      <script src="{{ url_for('static', filename='js/main.js')
  }}"></script>
      {% block scripts %}{% endblock %}
  </body>
  </html>
  ```

- `header.html`: `main_app/main_app_package/templates/header.html`:

  HTML

  ```
  <header>
      <nav>
  ```

```
        <a href="{{ url_for('main_routes.home') }}">Home</a>
        <a href="{{ url_for('main_routes.login') }}">Login</a>
        <a href="{{ url_for('main_routes.profile_details')
}}">Profile</a>
        </nav>
    <hr>
</header>
```

- `footer.html`: `main_app/main_app_package/templates/footer.html`:

  HTML

  ```
  <footer>
      <hr>
      <p>&copy; {{ "now"|date("%Y") }} My Application. All rights
  reserved.</p>
  </footer>
  ```

  (Note: `{{ "now"|date("%Y") }}` requires `datetime` to be in Jinja environment or passed via context processor. For simplicity, you might hardcode the year or pass it from Python.) For dynamic year, in `__init__.py`:

  Python

  ```
  # In create_app in __init__.py
  import datetime
  @app.context_processor
  def inject_now():
      return {'now': datetime.datetime.utcnow()}
  ```

- `login.html`: `main_app/main_app_package/templates/login.html`:

  HTML

  ```
  {% extends "base.html" %}

  {% block title %}Login - {{ super() }}{% endblock %}

  {% block content %}
  <h2>Login Page</h2>
  <form method="POST" action="{{ url_for('main_routes.login') }}">
      <div>
          <label for="username">Username:</label>
          <input type="text" id="username" name="username" required>
      </div>
      <div>
          <label for="password">Password:</label>
          <input type="password" id="password" name="password" required>
      </div>
      <button type="submit">Login</button>
  </form>
  {% endblock %}
  ```

- student_home.html: main_app/main_app_package/templates/student_home.html:

  HTML

  ```
  {% extends "base.html" %}

  {% block title %}Student Home - {{ super() }}{% endblock %}

  {% block content %}
  <h1>Welcome, Student!</h1>
  <p>This is your main dashboard.</p>
  <p>Check out your <a href="{{ url_for('main_routes.profile_details')
  }}">Profile</a>.</p>
  {% endblock %}
  ```

- profile_details.html:
  main_app/main_app_package/templates/profile_details.html:

  HTML

  ```
  {% extends "base.html" %}

  {% block title %}
      {% if profile_data and not error %}
          {{ profile_data.get('name', 'User') }}'s Profile
      {% else %}
          Profile Details
      {% endif %}
      - {{ super() }}
  {% endblock %}

  {% block head_css %}
      <link rel="stylesheet" href="{{ url_for('static',
  filename='css/profile_style.css') }}">
  {% endblock %}

  {% block content %}
  <h2>Profile Details</h2>
  {% if error %}
      <p class="error-message">Could not load profile information.</p>
  {% elif profile_data %}
      <p><strong>Name:</strong> {{ profile_data.get('name', 'N/A') }}</p>
      <p><strong>Email:</strong> {{ profile_data.get('email', 'N/A')
  }}</p>
      <p><strong>Student ID:</strong> {{ profile_data.get('student_id',
  'N/A') }}</p>
      {% if profile_data.get('profile_pic_url') %}
          <img src="{{ PROFILE_SERVICE_URL }}{{
  profile_data.profile_pic_url }}" alt="Profile Picture" width="150">
          {% else %}
          <img src="{{ url_for('static',
  filename='images/default_avatar.png') }}" alt="Default Avatar"
  width="150">
      {% endif %}
  ```

```
    {% else %}
    <p>No profile data available.</p>
{% endif %}
{% endblock %}
```

Step 3.8: Static Files (`main_app/main_app_package/static/`)

- `css/style.css`: `main_app/main_app_package/static/css/style.css`:

  CSS

  ```
  body { font-family: sans-serif; margin: 20px; }
  header, footer { text-align: center; margin-bottom: 20px; }
  nav a { margin: 0 10px; text-decoration: none; }
  .alert { padding: 10px; margin-bottom: 10px; border: 1px solid
  transparent; border-radius: 4px; }
  .alert-danger { color: #a94442; background-color: #f2dede; border-
  color: #ebccd1; }
  .alert-info { color: #31708f; background-color: #d9edf7; border-color:
  #bce8f1; }
  .error-message { color: red; font-weight: bold; }
  ```

- `css/profile_style.css`:
  `main_app/main_app_package/static/css/profile_style.css`:

  CSS

  ```
  /* Styles specific to the profile page */
  .profile-details img { border: 1px solid #ddd; border-radius: 4px;
  padding: 5px; }
  ```

- `js/main.js`: `main_app/main_app_package/static/js/main.js`:

  JavaScript

  ```
  console.log("Main App JavaScript Loaded!");
  // Add any global JavaScript here
  ```

- Place `usp_logo.png` and `default_avatar.png` in `main_app/main_app_package/static/images/`.

Step 3.9: `main_app/run_main_app.py`

This script runs your main application. `main_app/run_main_app.py`:

Python
```
from main_app_package import create_app # Ensure main_app_package is in
PYTHONPATH or adjust import
import os
```

```
app = create_app()

if __name__ == '__main__':
    # Debug will be set from Config, but port can be overridden here if
needed
    port = int(os.environ.get("MAIN_APP_PORT", 5000)) # Default to 5000 if
not set
    app.run(host='0.0.0.0', port=port)
```

4. Implementing the `profile_service`

(Ensure `profile_service_env` is activated for these steps if working in a new terminal session)

Step 4.1: `profile_service/config_profile.py` (Optional, but good practice) You might want a simple config for the microservice too. `profile_service/config_profile.py`:

Python
```
import os
from dotenv import load_dotenv

# Construct the path to the .env file located in the project root
dotenv_path = os.path.join(os.path.dirname(os.path.dirname(__file__)),
'.env')
load_dotenv(dotenv_path)

class ProfileServiceConfig:
    SECRET_KEY = os.environ.get('PROFILE_SERVICE_SECRET_KEY') or
'profile_service_secret'
    DEBUG = os.environ.get('PROFILE_SERVICE_DEBUG', 'False').lower() ==
'true'
    UPLOAD_FOLDER = os.path.join(os.path.dirname(__file__), 'profile_pics')
    # Ensure UPLOAD_FOLDER exists
    if not os.path.exists(UPLOAD_FOLDER):
        os.makedirs(UPLOAD_FOLDER)
```

Step 4.2: `profile_service/run_profile_service.py`

This is the Flask application for the profile microservice.
`profile_service/run_profile_service.py`:

Python
```
from flask import Flask, jsonify, request, send_from_directory
import os
# from config_profile import ProfileServiceConfig # If you created
config_profile.py

app = Flask(__name__)

# If using config_profile.py:
# app.config.from_object(ProfileServiceConfig)
# Otherwise, configure directly or load from .env here:
```

```
app.config['SECRET_KEY'] = os.environ.get('PROFILE_SERVICE_SECRET_KEY') or
'profile_service_secret'
app.config['DEBUG'] = os.environ.get('PROFILE_SERVICE_DEBUG',
'False').lower() == 'true'
app.config['UPLOAD_FOLDER'] = os.path.join(os.path.dirname(__file__),
'profile_pics')

# Ensure UPLOAD_FOLDER exists
if not os.path.exists(app.config['UPLOAD_FOLDER']):
    os.makedirs(app.config['UPLOAD_FOLDER'])

# Dummy data - in a real app, this would come from a database
PROFILES = {
    "s12345678": {
        "name": "Alice Wonderland",
        "email": "alice.w@example.com",
        "student_id": "s12345678",
        "bio": "Curiouser and curiouser.",
        "profile_pic_filename": "s12345678.jpg" # Example filename
    },
    "s87654321": {
        "name": "Bob The Builder",
        "email": "bob.b@example.com",
        "student_id": "s87654321",
        "bio": "Can we fix it? Yes, we can!",
        "profile_pic_filename": None
    }
}

@app.route('/profile/<string:user_id>', methods=['GET'])
def get_profile(user_id):
    profile = PROFILES.get(user_id)
    if profile:
        # Construct the URL for the profile picture if it exists
        pic_url = None
        if profile.get("profile_pic_filename"):
            # This endpoint will serve the image, see below
            pic_url = f"/profile_pics/{profile['profile_pic_filename']}"

        return jsonify({**profile, "profile_pic_url": pic_url})
    return jsonify({"error": "Profile not found"}), 404

@app.route('/profile_pics/<path:filename>')
def serve_profile_pic(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'], filename)

# Example: Endpoint to upload/update profile picture (very basic)
@app.route('/profile/<string:user_id>/upload_pic', methods=['POST'])
def upload_profile_picture(user_id):
    if user_id not in PROFILES:
        return jsonify({"error": "User profile not found"}), 404

    if 'profile_image' not in request.files:
        return jsonify({"error": "No profile image part in the request"}),
400

    file = request.files['profile_image']
```

```
    if file.filename == '':
        return jsonify({"error": "No image selected for uploading"}), 400

    if file: # Add checks for allowed file types
        # Use a secure filename and save it (e.g., based on user_id)
        filename = f"{user_id}.{file.filename.rsplit('.', 1)[1].lower()}" #
e.g., s12345678.jpg
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(file_path)
        PROFILES[user_id]['profile_pic_filename'] = filename
        return jsonify({"message": "Profile picture uploaded successfully",
"filename": filename}), 200

    return jsonify({"error": "Upload failed"}), 500


if __name__ == '__main__':
    port = int(os.environ.get("PROFILE_SERVICE_PORT", 5001)) # Default to
5001
    # Use a different port than the main app
    app.run(host='0.0.0.0', port=port, debug=app.config['DEBUG'])
```

You'll need to place an image like `s12345678.jpg` into `profile_service/profile_pics/` manually for the initial GET request to work with an image.

5. Running the Applications

You will need two separate terminal windows.

Terminal 1: Run the `profile_service`

1. Navigate to `your_project_root/profile_service/`.
2. Activate its virtual environment:
    - Windows: `profile_service_env\Scripts\activate`
    - macOS/Linux: `source profile_service_env/bin/activate`
3. Run the service (it's configured to run on port 5001 by default from the `.env` example or the script):

    Bash

    ```
    python run_profile_service.py
    ```

    You should see output indicating it's running on `http://127.0.0.1:5001/`.

Terminal 2: Run the `main_app`

1. Navigate to `your_project_root/main_app/`.
2. Activate its virtual environment:
    - Windows: `main_app_env\Scripts\activate`
    - macOS/Linux: `source main_app_env/bin/activate`

3. Set the `PYTHONPATH` if `run_main_app.py` has trouble finding `main_app_package`. Alternatively, you can install your package in editable mode (`pip install -e .` from within `main_app/` which would require a `setup.py`). For simplicity in development, `PYTHONPATH` can be easier:
    - macOS/Linux: `export PYTHONPATH=$PYTHONPATH:$(pwd)/..` (run from `main_app` dir, points to `your_project_root`)
    - Windows (cmd): `set PYTHONPATH=%PYTHONPATH%;%CD%\..`
    - Windows (PowerShell): `$env:PYTHONPATH += ";${pwd}\.."`
    - A more robust way for `run_main_app.py` is to adjust `sys.path` or ensure your project structure and how you run it makes `main_app_package` discoverable. One common way is to structure `run_main_app.py` to be outside `main_app_package` and add the parent directory of `main_app_package` to the path if needed, or run `main_app` as a module: `python -m main_app.run_main_app` (this might require adjustments to `run_main_app.py` or `__main__.py` in the package). For this specific structure, placing `run_main_app.py` inside `main_app` directory, and `main_app_package` being a sibling, the import `from main_app_package import create_app` in `run_main_app.py` should work if `main_app` is in Python's search path. Running `python run_main_app.py` from within the `main_app` directory usually makes the current directory part of `sys.path`.
4. Run the application (it's configured to run on port 5000 by default from the `.env` example or the script):

    Bash

    ```
    python run_main_app.py
    ```

    You should see output indicating it's running on `http://127.0.0.1:5000/`.

Accessing the Application:

- Open your web browser and go to `http://127.0.0.1:5000/` to see the `main_app`.
- Navigate to `http://127.0.0.1:5000/profile`. This page in `main_app` should attempt to fetch data from `profile_service` running on port 5001.
- You can test the `profile_service` directly by going to `http://127.0.0.1:5001/profile/s12345678` in your browser.

6. Further Development

- **Database Integration:** Add Flask-SQLAlchemy or another ORM to `main_app` and/or `profile_service` for persistent data storage.
- **Authentication:** Fully implement the `auth.py` logic using Flask-Login or JWTs.
- **Error Handling:** Improve error handling in both applications.
- **Testing:** Write unit and integration tests.
- **Deployment:** Consider tools like Docker, Gunicorn, Nginx for deploying your applications.

- **Microservice Communication:** For more robust communication than simple `requests`, explore message queues (e.g., RabbitMQ, Kafka) or gRPC.
- **Forms:** Use Flask-WTF for creating and validating forms in `main_app`.