PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

API Gateway Pattern Implementation

Course: Software Architecture

Course Class: CSE703110-1-2-25(N02)

Group 7:

1. Le Thi Kieu Trang **ID: 23010502**

2. Quach Huu Nam **ID: 23012358**

3. Trieu Tien Quynh **ID: 23010648**

Hanoi, January 7, 2026

Lab 6 – API Gateway Pattern Implementation

1. Abstract

In Microservices architecture, allowing the Client (Frontend) to communicate directly with individual microservices (Direct Client-to-Microservice Communication) poses significant security risks, complicates CORS configuration, and exposes the internal network infrastructure.

Lab 6 Objectives:

Deploy an API Gateway to serve as the Single Entry Point for the entire system.

Implement a Reverse Proxy mechanism: Route requests from the Client to the correct target service (Student, Course, Identity, etc.).

Establish Security Checks: Perform centralized Authentication at the Gateway before permitting requests to enter the system.

2. Technology & Tool Installation

We will use Python/Flask along with the requests library to make internal HTTP calls to the backend services.

Tool	Purpose	Installation/Setup Guide
Python 3.x	Core programming language.	Ensure Python 3 is installed.
Flask	Lightweight web framework for handling incoming requests.	pip install Flask
requests	Python library for making HTTP requests (used to call backend services).	pip install requests
Student Service	The backend service the Gateway will route to.	Ensure app.py is running on port 5001.

3. Activity Practice 1: Project Setup and Configuration

Goal: Create the Gateway project structure and define service locations.

Step-by-Step Instructions:

1. Create Gateway Directory:

```
# Navigate to your microservices directory

mkdir api_gateway

cd api_gateway

touch app.py
```

2. Define Service Configuration: In a real application, these would be loaded from environment variables, but we will define them in code for simplicity.
   File: app.py (Configuration)

```
# Define the base URLs for the backend services

# Ensure the Student Service is running on port 5001

STUDENT_SERVICE_URL = 'http://127.0.0.1:5001/api/students'

# Define the port the Gateway itself will run on

GATEWAY_PORT = 5000
```

4. Activity Practice 2: Security and Routing Implementation

Goal: Implement the logic for token validation (stub) and request forwarding.

Step-by-Step Instructions:

1. Implement Security Stub: Create functions to simulate checking an authorization token found in the request header.
   File: app.py (Add to existing content)

```
from flask import Flask, request, jsonify, make_response

import requests
```

```
app = Flask(__name__)

def validate_token(auth_header):
    """Simulates checking an Authorization token."""
    if not auth_header:
        return False, "Authorization header missing"
    token = auth_header.split("Bearer ")[-1]
    # Simple security logic: Only 'valid-admin-token' and 'valid-user-token' are accepted
    if token in ("valid-admin-token", "valid-user-token"):
        return True, None
    else:
        return False, "Invalid or expired token"
def is_admin_token(auth_header):
    """Checks if the token belongs to an admin user."""
    if auth_header and "valid-admin-token" in auth_header:
        return True
    return False
```

2. Implement the Routing Logic: Create an endpoint on the Gateway that accepts requests for /api/students and forwards them to the backend service.
   File: app.py (Add route handler)

```
@app.route('/api/students', defaults={'path': ''}, methods=['GET', 'POST'])
@app.route('/api/students/<path:path>', methods=['GET', 'POST', 'PUT', 'DELETE'])
def route_student_service(path):
    # 1. SECURITY CHECK (Cross-Cutting Concern)
```

```
    auth_header = request.headers.get('Authorization')
    is_valid, error_msg = validate_token(auth_header)
    if not is_valid:
        # Block unauthorized requests at the Gateway
        return jsonify({"error": "Unauthorized", "details": error_msg}), 401
    # Admin check for POST/PUT/DELETE operations
    if request.method in ['POST', 'PUT', 'DELETE'] and not is_admin_token(auth_header):
        return jsonify({"error": "Forbidden", "details": "Only Admins can modify students"}),
403


    # 2. ROUTING LOGIC
    suffix = f"/{path}" if path else ""
    query_string = f"?{request.query_string.decode('utf-8')}" if request.query_string else ""
    target_url = f"{STUDENT_SERVICE_URL}{suffix}{query_string}"
    try:
        # Forward the request to the Student Service
        response = requests.request(
            method=request.method,
            url=target_url,
            headers={k: v for k, v in request.headers.items() if k.lower() != 'host'},
            data=request.get_data(),
            timeout=5
        )
```

```
        # 3. RESPONSE HANDLING
        gateway_response = make_response(response.content, response.status_code)
        # Copy headers from backend response, filtering out hop-by-hop headers
        for key, value in response.headers.items():
            if key.lower() not in ['content-length', 'transfer-encoding', 'connection']:
                gateway_response.headers[key] = value
        return gateway_response
    except requests.exceptions.RequestException as e:
        # Handle connection errors (e.g., if the backend service is down)
        return jsonify({"error": "Service Unavailable", "details": str(e)}), 503
if __name__ == '__main__':
    print(f"API Gateway starting on port {GATEWAY_PORT}...")
    app.run(port=GATEWAY_PORT, debug=True)
```

4. Activity Practice 3: Testing the Gateway

Goal: Verify that the Gateway correctly handles security and routing.

Prerequisites:

- Ensure the Student Service is running on port 5001.
- Start the API Gateway on port 5000.

Test Cases (Using cURL or Postman):

1. Test Unauthorized Access:
    - Action: Attempt to access the student list without a token.
    - Command: curl -X GET http://127.0.0.1:5000/api/students

- Expected Result: HTTP 401 Unauthorized (Blocked by Gateway).

2. Test Authorized Access (Success):
    - Action: Access students with a valid user token.
    - Command: curl -X GET -H "Authorization: Bearer valid-user-token" http://127.0.0.1:5000/api/students
    - Expected Result: HTTP 200 OK and the student list JSON.
3. Test Forbidden Access (RBAC):
    - Action: Attempt to create a student (POST) using a regular user token.
    - Command: curl -X POST -H "Authorization: Bearer valid-user-token" -H "Content-Type: application/json" -d '{"student_id": "S999", "student_name": "New Student"}' http://127.0.0.1:5000/api/students
    - Expected Result: HTTP 403 Forbidden (Blocked by Gateway's Admin check).
4. Test Service Failure Handling:
    - Action: Stop the Student Service (port 5001) and attempt the authorized GET again.

Expected Result: HTTP 503 Service Unavailable (Handled by Gateway's exception block).

5. Conclusion

5.1. Key Achievements

Successfully constructed and deployed the API Gateway running on port 5000.

The Frontend system now connects to a single unified endpoint (localhost:5000), resulting in a leaner and more maintainable JavaScript codebase.

Achieved centralized security: The Gateway acts as the "gatekeeper," significantly reducing the processing load by eliminating redundant security checks at each individual microservice.

5.2. Limitations & Future Improvements

Limitations: The Gateway currently employs a static routing mechanism (URLs are hard-coded in the SERVICES variable). Consequently, if a sub-service changes its IP or Port, the Gateway requires manual code updates and a system restart.

Future Improvements: Integrate Service Discovery (such as Consul or Eureka) in upcoming labs to enable the Gateway to automatically detect active services without requiring manual configuration.