Lab 6: Introducing the API Gateway Pattern

This lab focuses on implementing the **API Gateway Pattern**, a crucial component in Microservices Architecture. The Gateway acts as the single-entry point for all client requests, routing them to the correct backend services (like the Product Service from Lab 5) and handling cross-cutting concerns like security.

Objectives

1. Understand the role of the **API Gateway** in a microservices environment.
2. Implement a simple reverse proxy/router using **Flask** and the requests library.
3. Configure the Gateway to route requests to the **Product Service**.
4. Implement a basic **Security Check stub** (e.g., simulated token validation) on the Gateway layer.

Technology & Tool Installation

We will use **Python/Flask** again, 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.	Run: pip install Flask
requests	Python library for making HTTP requests (used to call backend services).	Run: pip install requests
Product Service (from Lab 5)	The backend service the Gateway will route to.	Ensure the product_service application is running on http://127.0.0.1:5001.

Activity Practice 1: Project Setup and Dependencies

Goal: Create the Gateway project and install the necessary libraries.

Step-by-Step Instructions & Coding Guide

1. **Create Gateway Directory:**

Bash

```
# Ensure you are outside the product_service directory
mkdir api_gateway
cd api_gateway
python -m venv venv
source venv/bin/activate
pip install Flask requests
touch gateway.py
```

2. **Define Service Configuration:** In a real application, this would be loaded from a configuration file, but we will define the URL here for simplicity.

File: gateway.py (Configuration)

Python

```
# Define the base URLs for the backend services
# Ensure the Product Service (Lab 5) is running on port 5001
PRODUCT_SERVICE_URL = 'http://127.0.0.1:5001/api/products'
# Define the port the Gateway itself will run on
GATEWAY_PORT = 5000
```

Activity Practice 2: Security and Routing Implementation

Goal: Implement the logic for token validation (stub) and route mapping.

Step-by-Step Instructions & Coding Guide

1. **Implement Security Stub:** Create a function to simulate checking an authorization token found in the request header.

File: gateway.py (Add to existing content)

Python

```
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 products and forwards them to the backend service.

File: gateway.py (Add route)

Python

```
@app.route('/api/products', defaults={'path': ''}, methods=['GET', 'POST'])
@app.route('/api/products/<path:path>', methods=['GET', 'POST', 'PUT', 'DELETE'])
def route_product_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 products"}), 403

    # 2. ROUTING LOGIC
    # Construct the full backend URL, including any path or query parameters
    target_url =
f'{PRODUCT_SERVICE_URL.split("/api/products")[0]}/api/products/{path}?{request.query_string
.decode("utf-8")}'

    try:
        # Forward the request to the Product Service
        response = requests.request(
            method=request.method,
            url=target_url,
```

```
            headers={k: v for k, v in request.headers if k.lower() != 'host'}, # Forward headers, exclude 'Host'
            data=request.get_data(),
            timeout=5
        )

        # 3. RESPONSE HANDLING
        # Create a response object from the backend response
        gateway_response = make_response(response.content, response.status_code)
        # Copy all headers from backend response to the gateway response
        for key, value in response.headers.items():
            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": f"Product Service failed to respond: {e}"}), 503

if __name__ == '__main__':
    print(f"API Gateway starting on port {GATEWAY_PORT}...")
    app.run(port=GATEWAY_PORT, debug=True)
```

Activity Practice 3: Testing the Gateway

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

Prerequisites

- Ensure the **Product Service** (product_service/app.py) is running on **port 5001**.
- Start the **API Gateway** (api_gateway/gateway.py) on **port 5000**.

Step-by-Step Instructions (Using cURL or Postman)

1. **Test Unauthorized Access:**
    - **Action:** Attempt to access the product list without an authorization token.
    - **Command:** curl -X GET http://127.0.0.1:5000/api/products
    - **Expected Result:** HTTP **401 Unauthorized** (Blocked by the Gateway's security check).
2. **Test Authorized Access (Success):**
    - **Action:** Access the product list with a valid user token.
    - **Command:** curl -X GET -H "Authorization: Bearer valid-user-token" http://127.0.0.1:5000/api/products
    - **Expected Result:** HTTP **200 OK** and the product list JSON (Request successfully routed to the Product Service).
3. **Test Forbidden Access (Admin Check):**
    - **Action:** Attempt to create a new product (POST) using a regular user token. (Note: The Product Service itself may not have the POST implemented, but the **Gateway should block it first**).
    - **Command:** curl -X POST -H "Authorization: Bearer valid-user-token" -H "Content-Type: application/json" -d '{"name": "Test", "price": 1}' http://127.0.0.1:5000/api/products
    - **Expected Result:** HTTP **403 Forbidden** (Blocked by the Gateway's admin check).
4. **Test Service Failure Handling:**
    - **Action:** Shut down the Product Service (stop the process running on port 5001).
    - **Action:** Attempt the authorized GET request again.
    - **Command:** curl -X GET -H "Authorization: Bearer valid-user-token" http://127.0.0.1:5000/api/products
    - **Expected Result:** HTTP **503 Service Unavailable** (Handled by the Gateway's try...except block).

The API Gateway is now operational, providing a unified access point and handling crucial cross-cutting concerns for your ShopSphere Microservices Architecture.