**Slide Deck: Lecture 6 - API Gateway Pattern & Security**

**1. Title Slide: The Front Door**

- **Title:** Software Architecture: Lecture 6
- **Subtitle:** API Gateway Pattern & Security
- **Focus:** Implementing the crucial **API Gateway** to manage traffic and enforce our **Security ASR**.

---

**2. Recap: The Need for a Gateway**

- **Current State:** Clients would need to know the address (IP/Port) for *every* Microservice (Product on 5001, Order on 5002, etc.). This is complex and insecure.
- **Security ASR:** Requires centralized authentication and authorization checks.
- **Gateway Role:** The single entry point that handles client requests, performs cross-cutting concerns (like security), and routes the request to the correct backend service.

---

**3. Knowledge Base: API Gateway Pattern**

- **Definition:** The API Gateway acts as a **reverse proxy** and a facade for the microservices.
- **Key Functions:**
    1. **Routing/Service Discovery:** Directs requests (e.g., /api/products → Product Service, /api/orders → Order Service).
    2. **Authentication/Security:** Verifies tokens before forwarding the request.
    3. **Protocol Translation:** (Advanced) Translates REST to GraphQL or vice versa.
    4. **Rate Limiting/Monitoring:** Controls traffic volume and collects metrics.

---

**4. Architectural Drivers: Security ASR**

- **The Problem Solved:** Placing security logic (authentication, token validation) within the Gateway ensures it's enforced **before** the request reaches the business service.
- **Benefit:** Backend services (like the Product Service) are simpler, focusing only on business logic, and don't need to re-implement security checks.

---

**5. Activity 1: Project Setup and Dependencies**

- **Goal:** Create the Gateway project and install the HTTP client library necessary for routing.
- **Installation Steps:**
    1. mkdir api_gateway
    2. cd api_gateway
    3. python -m venv venv
    4. source venv/bin/activate
    5. pip install Flask requests (The requests library is used to make internal calls).
    6. touch gateway.py

---

**6. Knowledge Base: Reverse Proxy Implementation**

- **Mechanism:** When the Gateway receives an HTTP request, it creates a brand new HTTP request (using the requests library) and sends it internally to the backend service. It then takes the backend's response and sends it back to the original client.
- **Configuration:** The Gateway must know the URL/Port of every backend service it routes to.

---

**7. Code Step 1: Initial Setup and Service Configuration**

- **Goal:** Define the Gateway's port (5000) and the target service URL (Product Service, 5001).
- **File: gateway.py (Setup)**

Python

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

import requests
```

app = Flask(__name__)

GATEWAY_PORT = 5000

PRODUCT_SERVICE_URL = 'http://127.0.0.1:5001' # Target backend

---

**8. Knowledge Base: Implementing the Security Check Stub**

- **Concept:** We simulate token validation by checking for a specific string (Bearer valid-token) in the Authorization header.
- **Benefit:** This function is executed **once** at the Gateway, protecting all subsequent services.

---

**9. Code Step 2: The Security Logic**

- **Goal:** Implement a reusable function to check the incoming request's authorization header.
- **File: gateway.py (Security Stub)**

Python

```
def validate_token(auth_header):
    """Simulates checking an Authorization token."""
    if not auth_header or not auth_header.startswith("Bearer "):
        return False, "Authorization header missing or malformed"

    token = auth_header.split("Bearer ")[-1]

    # Simple whitelist check (Proof of Concept)
    if token in ("valid-admin-token", "valid-user-token"):
        return True, None
    else:
```

return False, "Invalid or expired token"

---

**10. Knowledge Base: The Central Routing Function**

- **Challenge:** The Gateway needs to handle all HTTP methods (GET, POST, PUT, DELETE) and preserve the path information (/api/products/123).
- **Flask Feature:** Using <path:path> in the route definition allows Flask to capture the rest of the URL path dynamically.

---

**11. Code Step 3: The Combined Gateway Route**

- **Goal:** Define the route and implement the security → routing flow.
- **File: gateway.py (Routing Logic - Part 1)**

Python

```
@app.route('/api/products', defaults={'path': ''}, methods=['GET', 'POST', 'PUT', 'DELETE'])
@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 right here (meeting the Security ASR)
        return jsonify({"error": "Unauthorized", "details": error_msg}), 401

    # ... Routing continues in the next slide ...
```

---

**12. Code Step 4: Internal Request Forwarding**

- **Goal:** Construct the target URL and forward the request using the requests library.

- **File: gateway.py (Routing Logic - Part 2)**

Python

```
    # --- 2. FORWARDING ---
    # Construct the full backend URL, including path and query string
    target_url = f'{PRODUCT_SERVICE_URL}/api/products/{path}?{request.query_string.decode("utf-8")}'

    try:
        # Forward the request, preserving method, data (body), and headers
        response = requests.request(
            method=request.method,
            url=target_url,
            headers={k: v for k, v in request.headers if k.lower() != 'host'}, # Preserve headers
            data=request.get_data(), # Preserve body content
            timeout=5
        )

        # ... Response handling in the next slide ...
```

---

**13. Code Step 5: Response Handling**

- **Goal:** Return the response received from the backend service to the original client.
- **File: gateway.py (Routing Logic - Part 3)**

Python

```
        # --- 3. RESPONSE HANDLING ---
        # Create a response object from the backend's status code and content
        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:
        # --- 4. FAILURE HANDLING ---
        # Critical for Availability: If the backend is unreachable
        return jsonify({"error": "Service Unavailable", "details": "Backend service failed to respond."}), 503
```

---

**14. Activity 2: Project Integration Checklist**

- **Prerequisite 1:** The **Product Service** (Lecture 5) must be running on **port 5001**.
- **Prerequisite 2:** The **API Gateway** (Lecture 6) must be running on **port 5000**.
- **Test Path:** All client traffic will now go through http://127.0.0.1:5000/api/products/...

---

**15. Verification (Hands-On): Test 1 - Unauthorized Access**

- **Goal:** Verify the **Security ASR** is met (Gateway blocks bad requests).
- **Action:** Attempt to view products without any header.
- **Tool:** cURL or Postman.
- **Command:** curl -X GET http://127.0.0.1:5000/api/products
- **Expected Result:** HTTP **401 Unauthorized** (The request was blocked by the Gateway's security stub).

---

**16. Verification (Hands-On): Test 2 - Successful Routing**

- **Goal:** Verify routing and token validation success.
- **Action:** View products with a valid 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 authenticated and routed to the backend).

---

**17. Verification (Hands-On): Test 3 - Failure Handling**

- **Goal:** Verify graceful degradation (Availability).
- **Action: Stop the Product Service** (terminate the process running on port 5001).
- **Action:** Repeat the successful request (from Test 2).
- **Command:** curl -X GET -H "Authorization: Bearer valid-user-token" http://127.0.0.1:5000/api/products
- **Expected Result:** HTTP **503 Service Unavailable** (The Gateway intercepted the network connection error and returned a friendly error, fulfilling an aspect of Availability).

---

**18. Architectural Review: Gateway Trade-Offs**

- **Pro:** Centralized security, unified entry point, improved fault tolerance (503 handling). Meets **Security ASR**.
- **Con: Single Point of Failure (SPOF):** If the Gateway itself fails, the entire system is down. Requires high-availability setup (load balancing and redundancy for the Gateway).
- **Trade-off:** We traded the complexity of having *multiple* client entry points for the operational necessity of managing *one* highly critical entry point.

---

**19. Summary of Deliverables for Submission**

- **Document 1:** Code snippet of the validate_token function (Security Stub).
- **Document 2:** Code snippet of the requests.request() call in the routing logic.

- **Document 3:** Output logs/screenshots showing the three critical test results: **401 Unauthorized**, **200 OK**, and **503 Service Unavailable**.

---

**20. Q&A and Next Steps**

- **Questions?** (Review header forwarding, data transfer).
- **Pre-work:** Research **Message Brokers** (RabbitMQ/Kafka) and the **Publisher-Subscriber** pattern.
- **Next Lecture: Lecture 7: Event-Driven Architecture (EDA) & Decoupling**. We tackle the **Fault Isolation ASR** using asynchronous communication.