import os import json import time import logging from datetime import datetime, timedelta from functools import wraps from flask import Flask, request, jsonify, Response from flask_cors import CORS from flask_jwt_extended import JWTManager, create_access_token, get_jwt_identity, jwt_required from flasgger import Swagger from pymongo import MongoClient import redis import requests from prometheus_client import Counter, Histogram, generate_latest, CONTENT_TYPE_LATEST from dotenv import load_dotenv

Load environment variables

load_dotenv()

Configure logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s') logger = logging.getLogger(**name**)

Initialize Flask app

app = Flask(**name**) CORS(app)

Configure JWT

app.config["JWT_SECRET_KEY"] = os.getenv("JWT_SECRET", "change_this_to_a_secure_random_string") app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1) jwt = JWTManager(app)

Configure Swagger

swagger_config = { "headers": [], "specs": [{ "endpoint": "apispec", "route": "/apispec.json", "rule_filter": lambda rule: True, "model_filter": lambda tag: True, }], "static_url_path": "/flasgger_static", "swagger_ui": True, "specs_route": "/docs/" } swagger = Swagger(app, config=swagger_config)

Connect to MongoDB

mongo_uri = os.getenv("MONGODB_URI", "mongodb://localhost:27017/amgf") mongo_client = MongoClient(mongo_uri) db = mongo_client.amgf users_collection = db.users api_keys_collection = db.api_keys metrics_collection = db.metrics

Connect to Redis

redis_uri = os.getenv("REDIS_URI", "redis://localhost:6379") redis_client = redis.from_url(redis_uri)

Service endpoints

OPPORTUNITY_SCANNER_URL = os.getenv(“OPPORTUNITY_SCANNER_URL”, “http://opportunity-scanner:5000”) ROI_PREDICTOR_URL = os.getenv(“ROI_PREDICTOR_URL”, “http://roi-predictor:5000”) RESOURCE_ALLOCATOR_URL = os.getenv(“RESOURCE_ALLOCATOR_URL”, “http://resource-allocator:5000”) DIGITAL_SERVICE_EXECUTOR_URL = os.getenv(“DIGITAL_SERVICE_EXECUTOR_URL”, “http://digital-service-executor:5000”) CONTENT_ARBITRAGE_SYSTEM_URL = os.getenv(“CONTENT_ARBITRAGE_SYSTEM_URL”, “http://content-arbitrage-system:5000”) MICRO_ARBITRAGE_ENGINE_URL = os.getenv(“MICRO_ARBITRAGE_ENGINE_URL”, “http://micro-arbitrage-engine:5000”) PERFORMANCE_TRACKER_URL = os.getenv(“PERFORMANCE_TRACKER_URL”, “http://performance-tracker:5000”)

Prometheus metrics

REQUEST_COUNT = Counter(‘request_count’, ‘App Request Count’, [‘method’, ‘endpoint’, ‘status’]) REQUEST_LATENCY = Histogram(‘request_latency_seconds’, ‘Request latency’, [‘method’, ‘endpoint’])

Rate limiting

RATE_LIMIT = int(os.getenv(“API_RATE_LIMIT”, “100”)) # requests per minute RATE_LIMIT_WINDOW = 60 # seconds

def rate_limit(f): @wraps(f) def decorated_function(*args, **kwargs): # Get client IP client_ip = request.remote_addr

```
    # Get current timestamp
    now = int(time.time())

    # Create rate limit key
    key = f"rate_limit:{client_ip}:{now // RATE_LIMIT_WINDOW}"

    # Increment request count
    current = redis_client.incr(key)

    # Set expiration if first request
    if current == 1:
        redis_client.expire(key, RATE_LIMIT_WINDOW)

    # Check if rate limit exceeded
    if current > RATE_LIMIT:
        return jsonify({
            "status": "error",
            "message": "Rate limit exceeded",
            "timestamp": datetime.utcnow().isoformat()
        }), 429

    return f(*args, **kwargs)
return decorated_function
```

def api_key_required(f): @wraps(f) def decorated_function(*args, **kwargs): # Get API key from header api_key = request.headers.get(“X-API-Key”)

```
    if not api_key:
        return jsonify({
            "status": "error",
            "message": "API key required",
            "timestamp": datetime.utcnow().isoformat()
        }), 401

    # Check if API key exists
    key_doc = api_keys_collection.find_one({"key": api_key})

    if not key_doc:
        return jsonify({
            "status": "error",
            "message": "Invalid API key",
            "timestamp": datetime.utcnow().isoformat()
        }), 401

    # Check if API key is active
    if not key_doc.get("active", False):
        return jsonify({
            "status": "error",
            "message": "API key is inactive",
            "timestamp": datetime.utcnow().isoformat()
        }), 401

    # Store user ID in request
    request.user_id = key_doc.get("user_id")

    return f(*args, **kwargs)
return decorated_function
```

@app.before_request def before_request(): request.start_time = time.time()

@app.after_request def after_request(response): # Record request latency latency = time.time() - request.start_time REQUEST_LATENCY.labels(request.method, request.path).observe(latency)

```
# Record request count
REQUEST_COUNT.labels(request.method, request.path,
response.status_code).inc()

return response
```

API Routes

@app.route(‘/health’, methods=[‘GET’]) def health_check(): “”” Health check endpoint — responses: 200: description: System is healthy “”” return jsonify({“status”: “healthy”, “timestamp”: datetime.utcnow().isoformat()})

@app.route(‘/metrics’, methods=[‘GET’]) def metrics(): “”” Prometheus metrics endpoint — responses: 200: description: Prometheus metrics “”” return Response(generate_latest(), mimetype=CONTENT_TYPE_LATEST)

@app.route(‘/auth/login’, methods=[‘POST’]) def login(): “”” User login endpoint — parameters: - name: body in: body required: true schema: type: object properties: username: type: string password: type: string responses: 200: description: Login successful 401: description: Invalid credentials “”” username = request.json.get(“username”) password = request.json.get(“password”)

```
if not username or not password:
    return jsonify({
        "status": "error",
        "message": "Username and password required",
        "timestamp": datetime.utcnow().isoformat()
    }), 400

# In a real implementation, this would check against a hashed password
user = users_collection.find_one({"username": username})

if not user or user.get("password") != password:
    return jsonify({
        "status": "error",
        "message": "Invalid credentials",
        "timestamp": datetime.utcnow().isoformat()
    }), 401

# Create access token
access_token = create_access_token(identity=str(user["_id"]))

return jsonify({
    "status": "success",
    "access_token": access_token,
    "user_id": str(user["_id"]),
    "timestamp": datetime.utcnow().isoformat()
})
```

@app.route('/auth/api-key', methods=['POST']) @jwt_required() def create_api_key(): """ Create API key endpoint — parameters: - name: Authorization in: header type: string required: true description: Bearer token responses: 200: description: API key created 401: description: Unauthorized """ user_id = get_jwt_identity()

```
# Generate API key
import secrets
api_key = secrets.token_hex(16)

# Store API key
api_keys_collection.insert_one({
    "key": api_key,
    "user_id": user_id,
    "active": True,
    "created_at": datetime.utcnow().isoformat()
})

return jsonify({
    "status": "success",
    "api_key": api_key,
    "timestamp": datetime.utcnow().isoformat()
})
```

Opportunity Scanner Routes

@app.route('/opportunities', methods=['GET']) @rate_limit @api_key_required def get_opportunities(): """ Get opportunities endpoint — parameters: - name: X-API-Key in: header type: string required: true - name: platform in: query type: string required: false - name: category in: query type: string required: false - name: min_value in: query type: number required: false - name: max_value in: query type: number required: false - name: max_implementation_time in: query type: number required: false - name: competition_level in: query type: string required: false - name: limit in: query type: integer required: false responses: 200: description: Opportunities retrieved 500: description: Error retrieving opportunities """ try: # Forward request to Opportunity Scanner response = requests.get(f'{OPPORTUNITY_SCANNER_URL}/opportunities', params=request.args)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error getting opportunities: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/opportunities/scan’, methods=[‘POST’]) @rate_limit @api_key_required def trigger_scan(): “”” Trigger opportunity scan endpoint — parameters: - name: X-API-Key in: header type: string required: true responses: 200: description: Scan triggered 500: description: Error triggering scan “”” try: # Forward request to Opportunity Scanner response = requests.post(f”{OPPORTUNITY_SCANNER_URL}/scan”)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error triggering scan: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

ROI Predictor Routes

@app.route(‘/predict’, methods=[‘POST’]) @rate_limit @api_key_required def predict_opportunity(): “”” Predict ROI for opportunity endpoint — parameters: - name: X-API-Key in: header type: string required: true - name: body in: body required: true schema: type: object responses: 200: description: Prediction successful 500: description: Error making prediction “”” try: # Forward request to ROI Predictor response = requests.post(f”{ROI_PREDICTOR_URL}/predict”, json=request.json)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error predicting ROI: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/predict/batch’, methods=[‘POST’]) @rate_limit @api_key_required def predict_batch(): “”” Predict ROI for multiple opportunities endpoint — parameters: - name: X-API-Key in: header type: string required: true - name: body in: body required: true schema: type: array items: type: object responses: 200: description: Batch prediction successful 500: description: Error making batch prediction “”” try: # Forward request to ROI Predictor response = requests.post(f”{ROI_PREDICTOR_URL}/predict/batch”, json=request.json)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error batch predicting ROI: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

Resource Allocator Routes

@app.route(‘/allocate’, methods=[‘POST’]) @rate_limit @api_key_required def trigger_allocation(): “”” Trigger resource allocation endpoint — parameters: - name: X-API-Key in: header type: string required: true responses: 200: description: Allocation triggered 500: description: Error triggering allocation “”” try: # Forward request to Resource Allocator response =

requests.post(f”{RESOURCE_ALLOCATOR_URL}/allocate”)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error triggering allocation: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/allocation’, methods=[‘GET’]) @rate_limit @api_key_required def get_allocation(): “”” Get current resource allocation endpoint — parameters: - name: X-API-Key in: header type: string required: true responses: 200: description: Allocation retrieved 500: description: Error retrieving allocation “”” try: # Forward request to Resource Allocator response = requests.get(f”{RESOURCE_ALLOCATOR_URL}/allocation”)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error getting allocation: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/allocation/history’, methods=[‘GET’]) @rate_limit @api_key_required def get_allocation_history(): “”” Get allocation history endpoint — parameters: - name: X-API-Key in: header type: string required: true - name: limit in: query type: integer required: false responses: 200: description: Allocation history retrieved 500: description: Error retrieving allocation history “”” try: # Forward request to Resource Allocator response = requests.get(f”{RESOURCE_ALLOCATOR_URL}/allocation/history”, params=request.args)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error getting allocation history: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/reinvest’, methods=[‘POST’]) @rate_limit @api_key_required def reinvest_profits(): “”” Reinvest profits endpoint — parameters: - name: X-API-Key in: header type: string required: true - name: body in: body required: true schema: type: object properties: profits: type: number responses: 200: description: Profits reinvested 500: description: Error reinvesting profits “”” try: # Forward request to Resource Allocator response = requests.post(f”{RESOURCE_ALLOCATOR_URL}/reinvest”, json=request.json)

```
    return jsonify(response.json()), response.status_code
except Exception as e:
    logger.error(f"Error reinvesting profits: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

System Routes

@app.route(‘/system/status’, methods=[‘GET’]) @rate_limit @api_key_required def get_system_status(): “”” Get system status endpoint — parameters: - name: X-API-Key in: header type: string required: true responses: 200: description: System status retrieved 500: description: Error retrieving system status “”” try: # Check status of all services services = [{“name”: “opportunity_scanner”, “url”: f”{OPPORTUNITY_SCANNER_URL}/health”},

{“name”: “roi_predictor”, “url”: f”{ROI_PREDICTOR_URL}/health”}, {“name”: “resource_allocator”, “url”: f”{RESOURCE_ALLOCATOR_URL}/health”}]

```
    status = {}
    for service in services:
        try:
            response = requests.get(service["url"], timeout=2)
            status[service["name"]] = {
                "status": "healthy" if response.status_code == 200 else
"unhealthy",
                "response_time": response.elapsed.total_seconds()
            }
        except Exception as e:
            status[service["name"]] = {
                "status": "unavailable",
                "error": str(e)
            }

    # Get system metrics
    metrics = {}
    for metric_doc in metrics_collection.find():
        metric_name = metric_doc.get("metric")
        if metric_name:
            # Remove MongoDB ID
            if "_id" in metric_doc:
                del metric_doc["_id"]
            metrics[metric_name] = metric_doc

    return jsonify({
        "status": "success",
        "services": status,
        "metrics": metrics,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error getting system status: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/system/run’, methods=[‘POST’]) @rate_limit @api_key_required def run_system(): “”” Run full system cycle endpoint — parameters: - name: X-API-Key in: header type: string required: true responses: 200: description: System cycle started 500: description: Error starting system cycle “”” try: # 1. Trigger opportunity scan scan_response = requests.post(f”{OPPORTUNITY_SCANNER_URL}/scan”) if scan_response.status_code != 200: return jsonify({ “status”: “error”, “message”: “Error triggering opportunity scan”, “scan_response”: scan_response.json(), “timestamp”: datetime.utcnow().isoformat() }), 500

```
    # 2. Get opportunities
    opportunities_response = requests.get(f"
{OPPORTUNITY_SCANNER_URL}/opportunities")
    if opportunities_response.status_code != 200:
        return jsonify({
            "status": "error",
            "message": "Error getting opportunities",
            "opportunities_response": opportunities_response.json(),
            "timestamp": datetime.utcnow().isoformat()
        }), 500

    opportunities = opportunities_response.json().get("opportunities", [])

    # 3. Predict ROI for opportunities
    predictions_response = requests.post(
        f"{ROI_PREDICTOR_URL}/predict/batch",
        json=opportunities
    )
    if predictions_response.status_code != 200:
        return jsonify({
            "status": "error",
            "message": "Error predicting ROI",
            "predictions_response": predictions_response.json(),
            "timestamp": datetime.utcnow().isoformat()
        }), 500

    # 4. Allocate resources
    allocation_response = requests.post(f"
{RESOURCE_ALLOCATOR_URL}/allocate")
    if allocation_response.status_code != 200:
        return jsonify({
            "status": "error",
            "message": "Error allocating resources",
            "allocation_response": allocation_response.json(),
            "timestamp": datetime.utcnow().isoformat()
        }), 500

    return jsonify({
        "status": "success",
        "message": "System cycle started",
        "scan_result": scan_response.json(),
        "opportunities_count": len(opportunities),
        "allocation_result": allocation_response.json(),
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error running system cycle: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

if **name** == '**main**': # Start Flask app app.run(host='0.0.0.0', port=8080)