Gateway Service Integration Guide

> How to integrate your microservice with the Mission Engadi Gateway

Table of Contents

Overview

The Gateway Service acts as the single entry point for all Mission Engadi microservices. It provides:

- **Unified routing**: All requests go through the gateway
- **Authentication**: Centralized JWT validation
- **Rate limiting**: Protect your service from overload
- **Health monitoring**: Automatic health checks
- **Request tracing**: End-to-end request tracking
- **Logging**: Centralized request/response logging

Architecture Diagram

```
Client → Gateway Service → Your Microservice
          ├─ Routing
          ├─ Auth
          ├─ Rate Limiting
          ├─ Health Checks
          └─ Logging
```

Quick Start

1. Deploy Your Service

Ensure your service is accessible from the gateway:

```yaml
# docker-compose.yml
services:
  your-service:
    build: .
    ports:
      - "8003:8003"
    networks:
      - mission-engadi

networks:
  mission-engadi:
    external: true
```

2. Register with Gateway

Create a route configuration:

```
curl -X POST http://gateway:8000/api/v1/routes \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <admin-token>" \
  -d '{
    "path": "/api/v1/your-service/*",
    "target_service": "your-service",
    "target_url": "http://your-service:8003",
    "methods": ["GET", "POST", "PUT", "DELETE"],
    "auth_required": true,
    "rate_limit": 100,
    "is_active": true
  }'
```

3. Test Your Integration

```
# Request goes through gateway
curl http://gateway:8000/api/v1/your-service/health

# Gateway forwards to: http://your-service:8003/api/v1/your-service/health
```

Registering Your Service

Route Configuration

Path Patterns

Supports wildcard matching:

```
# Exact match
"/api/v1/auth/login"  # Only matches this exact path

# Wildcard match
"/api/v1/content/*"   # Matches /api/v1/content/items, /api/v1/content/items/123, etc.

# Multiple wildcards
"/api/v1/*/items/*"   # Matches /api/v1/content/items/123, /api/v1/media/items/456
```

HTTP Methods

Specify allowed methods:

```
{
  "methods": ["GET", "POST"],  // Only GET and POST allowed
  // or
  "methods": ["*"],            // All methods allowed
}
```

Authentication

Control authentication requirements:

```
{
  "auth_required": true,   // Requires JWT token
  "auth_required": false,  // Public endpoint
}
```

Using the API

Create Route (Admin Only)

```python
import requests

response = requests.post(
    "http://gateway:8000/api/v1/routes",
    headers={
        "Authorization": f"Bearer {admin_token}",
        "Content-Type": "application/json"
    },
    json={
        "path": "/api/v1/media/*",
        "target_service": "media-service",
        "target_url": "http://media-service:8004",
        "methods": ["GET", "POST", "PUT", "DELETE"],
        "auth_required": True,
        "rate_limit": 50,
        "is_active": True
    }
)

route = response.json()
print(f"Route created with ID: {route['id']}")
```

Update Route

```python
route_id = 123

response = requests.put(
    f"http://gateway:8000/api/v1/routes/{route_id}",
    headers={"Authorization": f"Bearer {admin_token}"},
    json={
        "rate_limit": 100,  # Increase rate limit
        "is_active": True
    }
)
```

Delete Route

```
response = requests.delete(
    f"http://gateway:8000/api/v1/routes/{route_id}",
    headers={"Authorization": f"Bearer {admin_token}"}
)
```

Authentication

How It Works

1. Client sends request with JWT token to gateway
2. Gateway validates token (if `auth_required: true`)
3. Gateway forwards request to your service
4. Your service receives validated request

Accessing User Information

The gateway adds authentication headers:

```
# In your service
from fastapi import Request

@app.get("/api/v1/your-service/profile")
async def get_profile(request: Request):
    # Gateway adds these headers after validation
    user_id = request.headers.get("X-Gateway-User-ID")
    user_email = request.headers.get("X-Gateway-User-Email")
    user_roles = request.headers.get("X-Gateway-User-Roles").split(",")

    return {
        "user_id": user_id,
        "email": user_email,
        "roles": user_roles
    }
```

Headers Added by Gateway

```
X-Gateway-User-ID: user_123
X-Gateway-User-Email: user@example.com
X-Gateway-User-Roles: user,editor
X-Gateway-Request-ID: abc123-def456
X-Forwarded-For: 192.168.1.100
```

Public Endpoints

For public endpoints, set `auth_required: false`:

```
{
  "path": "/api/v1/public/*",
  "auth_required": false
}
```

Rate Limiting

Default Rate Limits

Set at route level:

```
{
  "path": "/api/v1/your-service/*",
  "rate_limit": 100  // 100 requests per minute
}
```

Custom Rate Limit Rules

Create specific rules for endpoints:

```
# Strict rate limit for login endpoint
requests.post(
    "http://gateway:8000/api/v1/rate-limits",
    headers={"Authorization": f"Bearer {admin_token}"},
    json={
        "path": "/api/v1/your-service/login",
        "limit": 5,      # 5 requests
        "window": 60,    # per 60 seconds
        "is_active": True
    }
)

# Generous limit for read endpoints
requests.post(
    "http://gateway:8000/api/v1/rate-limits",
    headers={"Authorization": f"Bearer {admin_token}"},
    json={
        "path": "/api/v1/your-service/items",
        "limit": 1000,
        "window": 60,
        "is_active": True
    }
)
```

Rate Limit Response

When rate limit is exceeded:

```
{
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Too many requests",
    "retry_after": 45
  }
}
```

Headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1640390400
Retry-After: 45
```

Health Checks

Implementing Health Endpoint

Your service must implement a health check endpoint:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/api/v1/health")
async def health_check():
    # Check database, Redis, etc.
    return {
        "status": "healthy",
        "service": "your-service",
        "version": "1.0.0",
        "checks": {
            "database": "connected",
            "redis": "connected"
        }
    }
```

Health Check Intervals

Gateway checks service health:
- **Interval**: Every 30 seconds
- **Timeout**: 5 seconds
- **Failure threshold**: 3 consecutive failures

Circuit Breaker

When your service is unhealthy:

1. Gateway marks service as “unhealthy”
2. Requests receive `503 Service Unavailable`
3. Gateway continues health checks
4. Service auto-recovers when healthy

Error Handling

Error Response Format

Your service should return consistent error responses:

```python
from fastapi import HTTPException

@app.get("/api/v1/your-service/items/{item_id}")
async def get_item(item_id: int):
    item = await get_item_from_db(item_id)

    if not item:
        raise HTTPException(
            status_code=404,
            detail={
                "code": "ITEM_NOT_FOUND",
                "message": f"Item {item_id} not found",
                "details": {"item_id": item_id}
            }
        )

    return item
```

Gateway Error Responses

502 Bad Gateway

Your service returned an invalid response

503 Service Unavailable

Your service is marked unhealthy

504 Gateway Timeout

Your service didn't respond within timeout (30 seconds)

Request Tracing

Trace ID Propagation

Gateway adds trace ID to all requests:

```python
from fastapi import Request
import logging

@app.get("/api/v1/your-service/items")
async def get_items(request: Request):
    # Get trace ID from gateway
    trace_id = request.headers.get("X-Gateway-Request-ID")

    # Add to logs
    logging.info(
        "Fetching items",
        extra={"trace_id": trace_id}
    )

    # Include in responses for debugging
    return {
        "items": [...],
        "trace_id": trace_id
    }
```

Structured Logging

Use structured logging for better observability:

```
import structlog

logger = structlog.get_logger()

@app.post("/api/v1/your-service/items")
async def create_item(item: ItemCreate, request: Request):
    trace_id = request.headers.get("X-Gateway-Request-ID")

    logger.info(
        "item.create",
        trace_id=trace_id,
        user_id=request.headers.get("X-Gateway-User-ID"),
        item_title=item.title
    )

    new_item = await create_item_in_db(item)

    logger.info(
        "item.created",
        trace_id=trace_id,
        item_id=new_item.id
    )

    return new_item
```

Best Practices

1. Design for Gateway Integration

✅ **Do:**
- Use standard HTTP status codes
- Return consistent error formats
- Implement health check endpoint
- Support idempotency where applicable
- Use structured logging

❌ **Don't:**
- Implement your own authentication (use gateway's)
- Perform rate limiting (gateway handles it)
- Log raw request/response bodies (sensitive data)

2. Path Design

```
# Good: Clear, RESTful paths
/api/v1/content/items
/api/v1/content/items/{id}
/api/v1/content/categories

# Bad: Inconsistent, unclear
/getItems
/api/items-list
/content_api/v1/get_item_by_id
```

3. Versioning

Include version in path:

```
/api/v1/your-service/items  # Version 1
/api/v2/your-service/items  # Version 2 (breaking changes)
```

4. Performance

- Keep responses under 5MB
- Respond to health checks in <100ms
- Set appropriate timeouts
- Use async/await for I/O operations

5. Security

- Never return sensitive data in errors
- Validate all input (even from gateway)
- Use HTTPS in production
- Rotate secrets regularly

Examples

Complete Service Integration

```python
# your_service/main.py
from fastapi import FastAPI, Request, HTTPException
import structlog

app = FastAPI(title="Your Service")
logger = structlog.get_logger()

# Health check
@app.get("/api/v1/health")
async def health():
    return {"status": "healthy", "service": "your-service"}

# Protected endpoint
@app.get("/api/v1/your-service/items")
async def get_items(request: Request):
    # Extract user info from gateway headers
    user_id = request.headers.get("X-Gateway-User-ID")
    trace_id = request.headers.get("X-Gateway-Request-ID")

    logger.info("items.list", trace_id=trace_id, user_id=user_id)

    # Your business logic
    items = await fetch_items_from_db()

    return {
        "items": items,
        "trace_id": trace_id
    }

# Error handling
@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    trace_id = request.headers.get("X-Gateway-Request-ID")

    logger.error(
        "unhandled.error",
        trace_id=trace_id,
        error=str(exc),
        path=request.url.path
    )

    return {
        "error": {
            "code": "INTERNAL_ERROR",
            "message": "An internal error occurred",
            "trace_id": trace_id
        }
    }
```

Register with Gateway

```python
# scripts/register_with_gateway.py
import requests
import os

GATEWAY_URL = os.getenv("GATEWAY_URL", "http://gateway:8000")
ADMIN_TOKEN = os.getenv("ADMIN_TOKEN")

def register_service():
    response = requests.post(
        f"{GATEWAY_URL}/api/v1/routes",
        headers={
            "Authorization": f"Bearer {ADMIN_TOKEN}",
            "Content-Type": "application/json"
        },
        json={
            "path": "/api/v1/your-service/*",
            "target_service": "your-service",
            "target_url": "http://your-service:8003",
            "methods": ["GET", "POST", "PUT", "DELETE"],
            "auth_required": True,
            "rate_limit": 100,
            "is_active": True
        }
    )

    response.raise_for_status()
    route = response.json()
    print(f"✅ Service registered with route ID: {route['id']}")
    return route

if __name__ == "__main__":
    register_service()
```

Docker Compose Integration

```yaml
# docker-compose.yml
version: '3.8'

services:
  gateway:
    image: mission-engadi/gateway:latest
    ports:
      - "8000:8000"
    networks:
      - mission-engadi
    environment:
      - DATABASE_URL=postgresql://...

  your-service:
    build: .
    ports:
      - "8003:8003"
    networks:
      - mission-engadi
    environment:
      - GATEWAY_URL=http://gateway:8000
    depends_on:
      - gateway

networks:
  mission-engadi:
    driver: bridge
```

Troubleshooting

Service Not Receiving Requests

1. Check route configuration:
   `bash`
   `curl http://gateway:8000/api/v1/routes`

2. Verify service is reachable:
   `bash`
   `curl http://your-service:8003/api/v1/health`

3. Check gateway logs:
   `bash`
   `docker logs gateway-service`

Rate Limiting Issues

1. Check rate limit rules:
   `bash`
   `curl http://gateway:8000/api/v1/rate-limits`

2. Check rate limit status:
   `bash`

```
   curl "http://gateway:8000/api/v1/rate-limits/check?path=/api/v1/your-service/items&client_id=user123"
```

Authentication Failures

1. Verify route requires auth:
   ```
   json
      {"auth_required": true}
   ```
2. Check token validity:
   ```
   bash
      jwt decode <your-token>
   ```
3. Ensure token is in header:
   ```
   bash
      curl -H "Authorization: Bearer <token>" ...
   ```

Support

- **Documentation**: docs.engadi.org (https://docs.engadi.org)
- **Issues**: GitHub Issues (https://github.com/mission-engadi/gateway_service/issues)
- **Email**: support@engadi.org