08 Middleware in Web Apps - KirkYagami X NikhilSharma

Understanding Middleware in Web Applications

Introduction to Middleware

Middleware represents a crucial architectural pattern in modern web development that acts as the connecting layer between different components of a web application. Think of middleware as a series of processing layers that a request must pass through before reaching its intended destination, and then the response must travel back through on its return journey.

In essence, middleware functions are pieces of code that have access to:

- The request object
- The response object
- The next middleware function in the application's request-response cycle

This positioning gives middleware tremendous power to:

- Examine incoming requests
- Modify requests and responses
- Execute code
- End the request-response cycle
- Call the next middleware in the stack

The Middleware Concept: A Real-World Analogy

Imagine a postal service delivering mail (requests) to various departments in a large office building. Before reaching the intended recipient (your route handler), each piece of mail passes through several checkpoints:

1. The front desk logs all incoming mail (logging middleware)
2. A security officer scans for prohibited items (authentication middleware)
3. A sorter adds routing information (request processing middleware)
4. A translator helps if the mail is in a foreign language (parsing middleware)

Each checkpoint can:

- Process the mail and send it forward
- Return the mail to sender with a note (error response)
- Add something to the mail (enhance the request)
- Take something out of the mail (filter the request)

Why Use Middleware?

Middleware provides several significant benefits:

1. **Separation of Concerns**: Each middleware focuses on a specific functionality, making your code more modular and easier to maintain.
2. **Code Reusability**: Middleware can be applied across different routes or applications without duplication.

3. **Cross-Cutting Concerns**: Middleware is perfect for handling aspects that cut across your application like logging, authentication, or error handling.
4. **Request/Response Modification**: Middleware can transform requests or responses before they reach their final destination.
5. **Conditional Processing**: You can apply middleware conditionally based on routes or request properties.

Common Use Cases for Middleware

Here are situations where middleware shines:

1. **Authentication & Authorization**: Verifying user credentials and permissions before allowing access to protected routes.
2. **Logging**: Recording information about each request for monitoring and debugging.
3. **Error Handling**: Catching errors and providing appropriate responses.
4. **Request Parsing**: Processing request bodies in various formats (JSON, form data, etc.).
5. **CORS (Cross-Origin Resource Sharing)**: Managing cross-origin requests securely.
6. **Rate Limiting**: Controlling how many requests a client can make in a given timeframe.
7. **Response Compression**: Compressing responses for faster transmission.
8. **Session Management**: Handling user sessions across requests.

Middleware in FastAPI

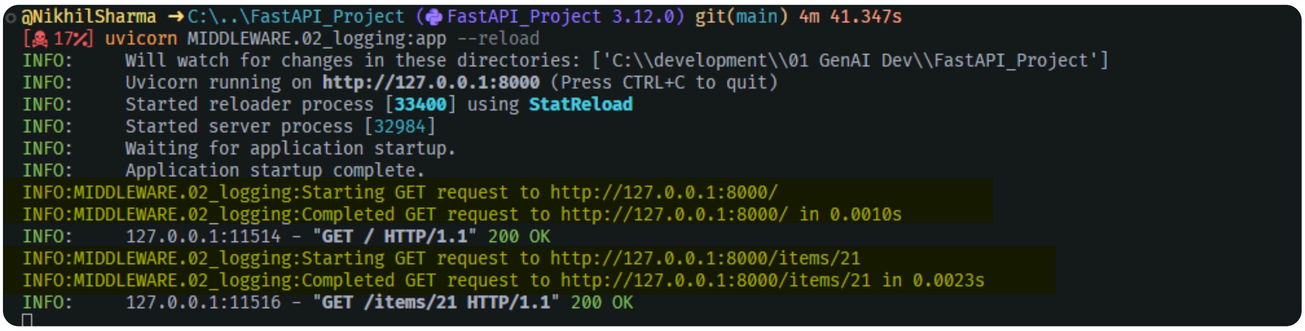
FastAPI uses a system of "dependencies" that functions similarly to middleware. However, it also supports more traditional middleware approaches.

FastAPI Middleware Example

Here's a basic logging middleware in FastAPI:

```python
from fastapi import FastAPI, Request
import time
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

@app.middleware("http")
async def log_requests(request: Request, call_next):
    # Log the request start
    start_time = time.time()
    method = request.method
    url = request.url

    logger.info(f"Starting {method} request to {url}")

    # Process the request
    response = await call_next(request)
    # Log the request completion
    process_time = time.time() - start_time
    logger.info(f"Completed {method} request to {url} in {process_time:.4f}s")

    return response


"""
The decorator `@app.middleware("http")` specifies that the function it decorates
will act as an **HTTP middleware** . This means the function will intercept every
HTTP request and response that passes through the FastAPI application.
"""

@app.get("/")
async def root():
    return {"message": "Hello World"}

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

This middleware logs each request's method and URL when it starts, processes the request, and then logs the completion time when finished.

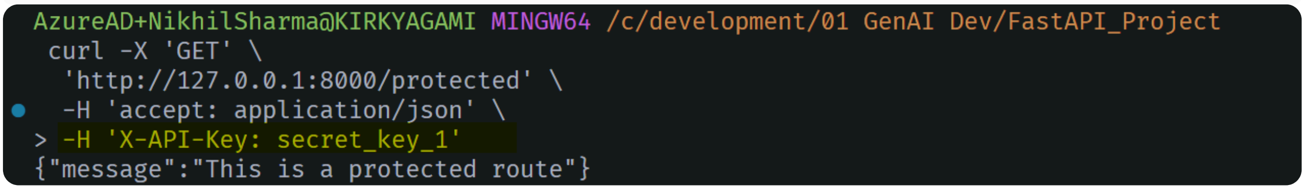


```
@NikhilSharma ➜C:\..\FastAPI_Project (FastAPI_Project 3.12.0) git(main) 4m 41.347s
[17%] uvicorn MIDDLEWARE.02_logging:app --reload
INFO:     Will watch for changes in these directories: ['C:\\development\\01 GenAI Dev\\FastAPI_Project']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [33400] using StatReload
INFO:     Started server process [32984]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:MIDDLEWARE.02_logging:Starting GET request to http://127.0.0.1:8000/
INFO:MIDDLEWARE.02_logging:Completed GET request to http://127.0.0.1:8000/ in 0.0010s
INFO:     127.0.0.1:11514 - "GET / HTTP/1.1" 200 OK
INFO:MIDDLEWARE.02_logging:Starting GET request to http://127.0.0.1:8000/items/21
INFO:MIDDLEWARE.02_logging:Completed GET request to http://127.0.0.1:8000/items/21 in 0.0023s
INFO:     127.0.0.1:11516 - "GET /items/21 HTTP/1.1" 200 OK
```

Authentication Middleware in FastAPI

Let's create a simple token-based authentication middleware:

```
from fastapi import FastAPI, Request, HTTPException
from fastapi.responses import JSONResponse

app = FastAPI()

# In a real application, you would store tokens more securely
VALID_API_KEYS = ["secret_key_1", "secret_key_2"]

@app.middleware("http")
async def authenticate_request(request: Request, call_next):
    # Exclude authentication for specific paths
    if request.url.path == "/docs" or request.url.path == "/redoc" or
request.url.path == "/openapi.json":
        return await call_next(request)

    # Check for API key in header
    api_key = request.headers.get("X-API-Key")
    if not api_key or api_key not in VALID_API_KEYS:
        return JSONResponse(
            status_code=401,
            content={"detail": "Invalid or missing API key"}
        )

    # If authenticated, proceed with the request
    response = await call_next(request)
    return response

@app.get("/protected")
async def protected_route():
    return {"message": "This is a protected route"}

@app.get("/public")
async def public_route():
    # Even though we named this "public", our middleware will still require
authentication
    # since it applies to all routes
    return {"message": "This is a public route"}
```



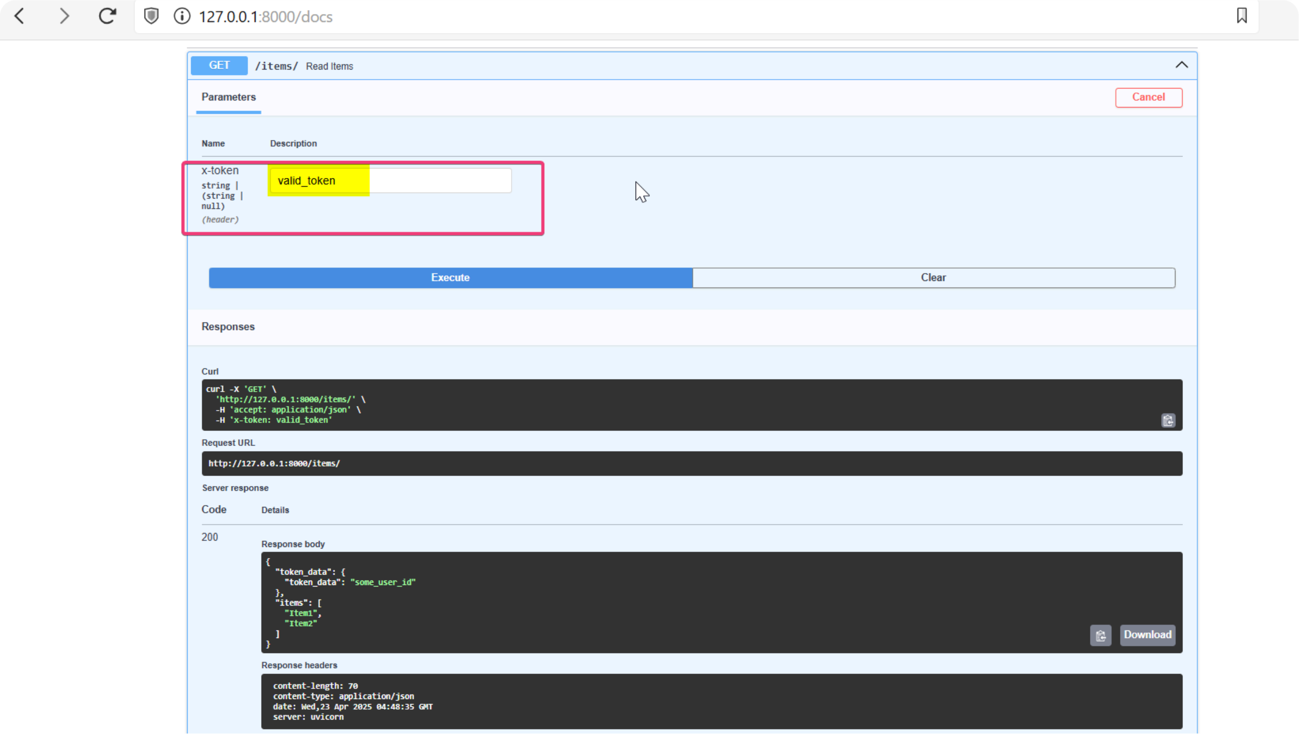
```
AzureAD+NikhilSharma@KIRKYAGAMI MINGW64 /c/development/01 GenAI Dev/FastAPI_Project
 curl -X 'GET' \
  'http://127.0.0.1:8000/protected' \
  -H 'accept: application/json' \
> -H 'X-API-Key: secret_key_1'
{"message":"This is a protected route"}
```

Using Dependencies in FastAPI as Middleware

FastAPI's dependency injection system can also achieve middleware-like functionality with more flexibility:

```
from fastapi import FastAPI, Depends, HTTPException, Request, Header
from typing import Optional

app = FastAPI()

async def verify_token(x_token: Optional[str] = Header(None)):
    """This function acts like middleware but is a dependency"""
    if not x_token:
        raise HTTPException(status_code=400, detail="X-Token header missing")
    if x_token != "valid_token":
        raise HTTPException(status_code=401, detail="Invalid X-Token")
    # We can return data that will be passed to the endpoint
    return {"token_data": "some_user_id"}

async def log_request(request: Request):
    """This dependency logs request information"""
    print(f"Request to {request.url.path}")
    # This doesn't return anything, just performs an action

@app.get("/items/", dependencies=[Depends(log_request)])
```

```python
async def read_items(token_data: dict = Depends(verify_token)):
    # token_data contains what verify_token returns
    return {"token_data": token_data, "items": ["Item1", "Item2"]}

@app.get("/status/")
async def get_status():
    # This endpoint doesn't use the verify_token dependency
    return {"status": "ok"}
```



Name	Description
x-token string | (string | null) *(header)*	valid_token

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/items/' \
  -H 'accept: application/json' \
  -H 'x-token: valid_token'
```

Request URL

```
http://127.0.0.1:8000/items/
```

Server response

Code	Details
200	Response body

```
{
  "token_data": {
    "token_data": "some_user_id"
  },
  "items": [
    "Item1",
    "Item2"
  ]
}
```

Response headers

```
content-length: 70
content-type: application/json
date: Wed,23 Apr 2025 04:48:35 GMT
server: uvicorn
```

Middleware in Flask

Flask has a more traditional middleware system using decorators, request hooks, and WSGI middleware.

Flask Middleware with Decorators

```python
from flask import Flask, request, g
from functools import wraps
import time

app = Flask(__name__)

# Authentication middleware as a decorator
def require_api_key(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        api_key = request.headers.get('X-API-Key')
        if not api_key or api_key != 'valid_key':
            return {'error': 'Invalid or missing API key'}, 401
        return f(*args, **kwargs)
    return decorated_function

# Logging middleware as a decorator
def log_request(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # Log before request
        start_time = time.time()
        app.logger.info(f"Request started: {request.method} {request.path}")

        # Process request
        response = f(*args, **kwargs)

        # Log after request
        duration = time.time() - start_time
        app.logger.info(f"Request completed in {duration:.4f}s")

        return response
    return decorated_function

@app.route('/protected')
@require_api_key
@log_request
def protected_route():
    return {'message': 'This is a protected route'}

@app.route('/public')
@log_request  # Only logging middleware, no auth required
def public_route():
    return {'message': 'This is a public route'}

if __name__ == '__main__':
    app.run(debug=True)
```

Flask Request Hooks

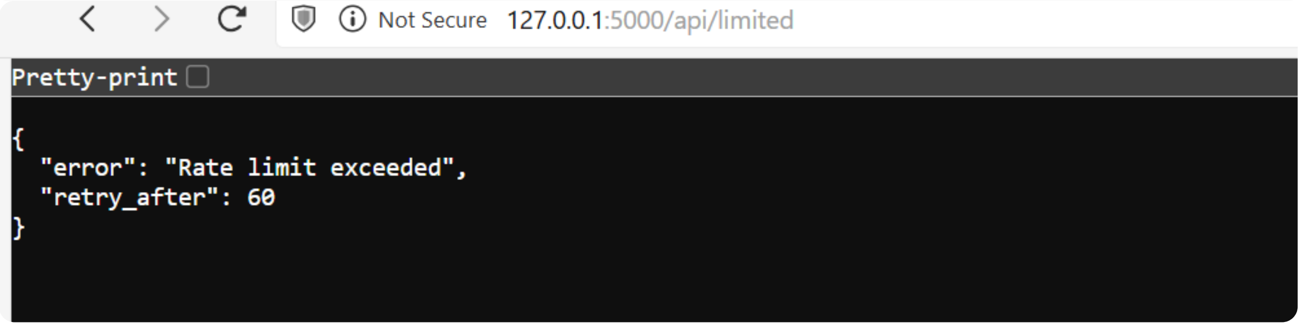
Flask provides request hooks that act as global middleware:

```python
from flask import Flask, request, g
import time

app = Flask(__name__)

@app.before_request
def before_request():
    """This runs before every request"""
    g.start_time = time.time()
    app.logger.info(f"Processing request: {request.method} {request.path}")

    # You can also verify authentication here
    if request.path.startswith('/admin') and request.headers.get('X-Admin-Key') != 
'admin_secret':
        return {'error': 'Unauthorized access to admin area'}, 403

@app.after_request
def after_request(response):
    """This runs after every request"""
    if hasattr(g, 'start_time'):
        duration = time.time() - g.start_time
        app.logger.info(f"Request processed in {duration:.4f}s")

    # You can modify the response
    response.headers['X-Powered-By'] = 'Flask'
    return response

@app.teardown_request
def teardown_request(exception=None):
    """This runs after every request, even if an exception occurred"""
    # Clean up resources
    if exception:
        app.logger.error(f"Request failed with exception: {exception}")

@app.route('/')
def home():
    return {'message': 'Welcome home'}

@app.route('/admin')
def admin():
    return {'message': 'Admin area'}

if __name__ == '__main__':
    app.run(debug=True)
```

Middleware for Specific Tasks

Rate Limiting Middleware

```python
from flask import Flask, request, g, jsonify
from functools import wraps
import time
from collections import defaultdict

app = Flask(__name__)

# Simple in-memory rate limiter
class RateLimiter:
    def __init__(self, limit=100, window=60):
        self.limit = limit  # Max requests
        self.window = window  # Time window in seconds
        self.clients = defaultdict(list)  # client_id -> list of timestamps

    def is_rate_limited(self, client_id):
        now = time.time()

        # Remove timestamps outside current window
        self.clients[client_id] = [ts for ts in self.clients[client_id] if now - ts < self.window]
        # Check if the client is over the limit
        if len(self.clients[client_id]) >= self.limit:
            return True

        # Record this request
        self.clients[client_id].append(now)
        return False

# Create a limiter with 5 requests per minute
limiter = RateLimiter(limit=5, window=60)

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

        # Check if client is rate limited
        if limiter.is_rate_limited(client_id):
            return jsonify({
                'error': 'Rate limit exceeded',
                'retry_after': 60  # Seconds to wait
            }), 429

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

@app.route('/api/limited')
@rate_limit
def limited_endpoint():
    return {'message': 'This endpoint is rate limited'}

@app.route('/api/unlimited')
def unlimited_endpoint():
    return {'message': 'This endpoint is not rate limited'}

if __name__ == '__main__':
    app.run(debug=True)
```



```
{
  "error": "Rate limit exceeded",
  "retry_after": 60
}
```

Request Validation Middleware

```
from fastapi import FastAPI, Request, HTTPException
from fastapi.responses import JSONResponse
import json

app = FastAPI()

@app.middleware("http")
async def validate_request_body(request: Request, call_next):
    # Only validate POST/PUT/PATCH requests
    if request.method in ["POST", "PUT", "PATCH"]:
        content_type = request.headers.get("Content-Type", "")

        # Validate JSON requests
        if "application/json" in content_type:
            try:
                # Try to parse the body as JSON
                body = await request.body()
                if body:
                    json.loads(body)
            except json.JSONDecodeError:
                return JSONResponse(
                    status_code=400,
                    content={"detail": "Invalid JSON format in request body"}
                )

    # Process the request normally
    return await call_next(request)

@app.post("/items/")
async def create_item(request: Request):
    # We can be confident here that if the request has a body, it's valid JSON
    body = await request.json()
    return {"received": body}

@app.get("/status/")
async def get_status():
    # This GET endpoint isn't affected by the validation middleware
    return {"status": "ok"}
```

Middleware Anti-patterns and Pitfalls

1. **Too Many Middleware:** Each middleware adds overhead. Use only what you need.
2. **Poorly Ordered Middleware:** Incorrect ordering can cause bugs that are hard to debug.
3. **Heavy Processing in Middleware:** Middleware should be lightweight. Move heavy processing to background tasks.
4. **No Error Handling:** Always handle exceptions in middleware to prevent request failures.
5. **Middleware State Leakage:** Be careful with shared state in middleware to avoid unexpected behaviors.

Conclusion: Choosing the Right Middleware Approach

The middleware approach you choose should match your application's requirements:

1. **For Simple Applications**:
   - Flask decorators or FastAPI dependencies offer simplicity and readability.
2. **For Larger Applications**:
   - ASGI/WSGI middleware provides better separation of concerns.
   - Framework-specific middleware offers performance benefits.
3. **For Enterprise Applications**:
   - Consider middleware that can be shared across multiple services.
   - Focus on observability, security, and maintainability.

Remember, middleware is a powerful pattern but should be used judiciously. Each layer adds complexity and potential performance costs. Choose middleware that provides clear value to your application architecture and user experience.