Implementing Shared Functionality using Middleware

In your WSGI, ASGI and gRPC applications

Amit Saha

https://echorand.me

kiwipycon

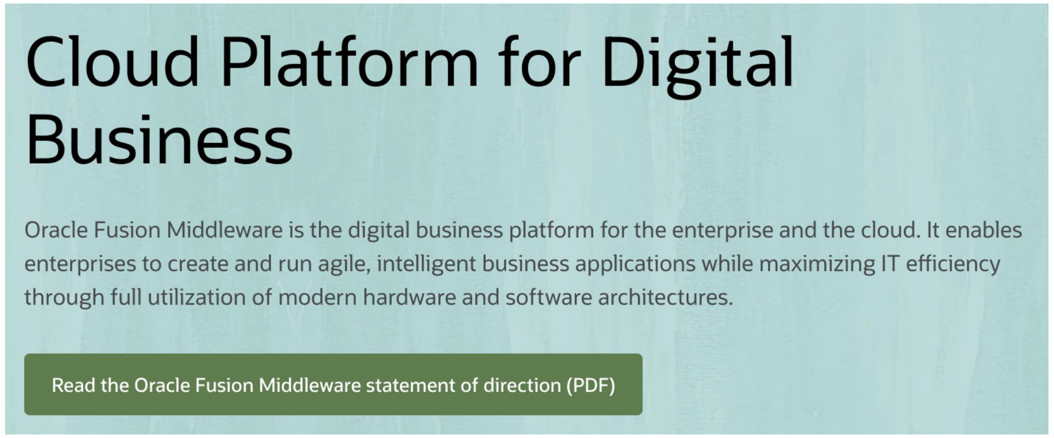
Agenda

- Middleware in Computing
- WSGI Middleware
- ASGI Middleware
- gRPC Middleware

Slides and Resources

https://echorand.me/talks/

Origin of “middleware”


Cloud Platform for Digital Business

Oracle Fusion Middleware is the digital business platform for the enterprise and the cloud. It enables enterprises to create and run agile, intelligent business applications while maximizing IT efficiency through full utilization of modern hardware and software architectures.

Read the Oracle Fusion Middleware statement of direction (PDF)

Usage in computing as early as 1968

“..middleware can be described as the dash ("-") in client-server, or the -to- in peer-to-peer.” –

Etzkorn, L. H. (2017). Introduction to Middleware: Web Services, Object Components, and Cloud Computing. CRC Press.

Today’s working definition

PEP 333 – Python Web Server Gateway Interface v1.0

.. it is also possible to create “middleware” components that implement both sides of this specification.

..and can be used to provide extended APIs, content transformation, navigation, and other useful functions.

Middleware for WSGI applications

A Flask Application

```
bp = Blueprint("blog", __name__)

@bp.route("/")
def index():
    return render_template("blog/index.html", posts=posts)
```

Flask middleware

```
@bp.before_request
def start_render_timer():
    g.start_render = time.time()


 @bp.after_request
 def stop_render_timer(response):
     print(f”latency:{time.time()-g.start_render} seconds")
     return response
```

A Django Application: View function

```
def index(request):
    return HttpResponse("Hello, world”)
```

Django middleware – class based

```
class ExecHandlingMiddleware:

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        try:
            return self.get_response(request)
        except:
            # return custom response
```

Activate middleware

```
# settings.py

MIDDLEWARE = [
    'polls.my_exc_handler.ExcHandlingMiddleware'
]
```

Recap

Using middleware, you define custom code to run before and after request processing

WSGI Frameworks define their own mechanism to define middleware

Pause

A WSGI application

```
def simple_handler(environ, start_response):

    # ..

    start_response(status, headers)

    ret = [b'Hello world\n’]
    return ret
```

A WSGI middleware

```
class MyExceptionProcessor:

    def __init__(self, wsgi_app):
        self.wsgi_app = wsgi_app

    def __call__(self, environ, start_response):
        try:
            return self.wsgi_app(environ, start_response)
        except Exception as e:
            start_response(status, headers)
            return [b'An error occured!\n']
```

WSGI application with middleware

```
app = MyExceptionProcessor(simple_handler)
```

```
$ gunicorn app:app
[2022-04-26 09:40:27 +1000] [72117] [INFO] Starting gunicorn 20.1.0
[2022-04-26 09:40:27 +1000] [72117] [INFO] Listening at: http://127.0.0.1:8000
(72117)
[2022-04-26 09:40:27 +1000] [72117] [INFO] Using worker: sync
[2022-04-26 09:40:27 +1000] [72119] [INFO] Booting worker with pid: 72119
```

Pause

Flask + WSGI Middleware

```
# import MyExceptionProcessor

app = Flask(__name__)

app.wsgi_app = MyExceptionProcessor(app.wsgi_app)
```

Django + WSGI Middleware

```
# wsgi.py

# import MyExceptionprocessor

application = get_wsgi_application()

application = MyExceptionProcessor(application)
```

Recap

Flask and Django implement custom mechanisms to allow users to define middleware

WSGI middleware is framework agnostic

Middleware for ASGI applications

A FastAPI application

```
app = FastAPI()

@app.get("/expensive")
async def root():
    await asyncio.sleep(10)
    return {"message": "Expensive calculation completed"}
```

Using ASGI Middleware

```
class ExpensiveCache:

    def __init__(self, app, excluded_paths):
        # initialization

    async def __call__(self, scope, receive, send):

        if cache_hit:
            # send cached response

        await self.app(scope, receive, cache_and_send)
```

Adding the Middleware

```
app = FastAPI()

app.add_middleware(
    ExpensiveCache,
    excluded_paths=["/chat"]
)
```

ASGI Middleware and WebSocket

```
class RequestTimer:

    async def __call__(self, scope, receive, send):
        await self.app(scope, receive, send)
        # print(“latency...”)
```

```
http:/chat: Got request.
http: /chat: Finished request. 0.0011079311370849615.

websocket:/ws: Got request.
..
websocket: /ws: Finished request. 28.716175079345703s.
```

Recap

ASGI middleware is framework agnostic

FastAPI has helper methods to add ASGI middleware

Interceptors for gRPC applications

gRPC Applications

Unary-Unary

- One request, one response (*Protobuf message*)

Bidirectional streaming

- One or more requests and responses (*Protobuf messages*)

Think of it like a *WebSocket* connection

Unary-Unary gRPC Applications

A gRPC service

RPC Method

```
class Identity(..):

    def ValidateToken(self, request, context):

        user_details = identity_pb2.ValidateTokenReply(user_id="default-user-ic

        return user_details


def serve(app_config: dict):
    server = grpc.server(
        futures.ThreadPoolExecutor(max_workers=10),
    )
    # ..
```

A Minimal Logging interceptor

```python
import grpc

class LoggingInterceptor(grpc.ServerInterceptor):

    def __init__(self):
        pass


    def intercept_service(self, continuation, handler_call_details):

        print(handler_call_details.method, handler_call_details.invocation_metad

        return continuation(handler_call_details)
```

Next interceptor or RPC method

Request Metadata

Integrating the interceptor(s)

```python
def serve(app_config: dict):
    server = grpc.server(
        futures.ThreadPoolExecutor(max_workers=10),
        interceptors = (LoggingInterceptor(),)
    )

    # .. Rest of the server
```

RPC Method called

Client metadata

Server logs

```
/Identity/ValidateToken (_Metadatum('grpc-python/1.48.0 grpc-c/26.0.0 (osx;
```

Bidi-streaming gRPC Applications

A bidi streaming RPC method

```
class Identity(..):

    def ExpireToken(self, request_iterator, context):

        for r in request_iterator:

            yield identity_pb2.ExpireTokenReply(result=True)
```

A logging interceptor

```
class LoggingInterceptor(grpc.ServerInterceptor):

    def intercept_service(self, continuation, handler_call_details):

        def logging_wrapper(behavior, request_streaming, response_streaming):

            def logging_interceptor(request_or_iterator, context):
                # More stuff

                if request_streaming or response_streaming:
                    return self._intercept_server_stream(
                        behavior,
                        request_or_iterator,
                        context,
                    )
                return behavior(request_or_iterator, context)

            # More stuff
```

Called once when the stream is created

Unary-Unary RPC methods

```
def _intercept_server_stream(
        self, behavior, request_or_iterator, context
    ):
        def wrapd(behavior, request_or_iterator, conte
            for r in request_or_iterator:

                print("Processing stream message", r)

                resp = behavior(list([r]), context)

                 yield from resp
```

This loop is executed for every message exchanged during the stream session

Server logs

```
/Identity/ExpireToken (_Metadatum(key='user-agent’, ..(osx; chttp2)'),)

Processing stream message token: "a-token"

Processing stream message token: "b-token"

Processing stream message token: "c-token"

Stream duration: 3.0171940326690674 seconds
```

Logging client-side interceptor

```
class LoggingClientInterceptor(grpc.UnaryUnaryClientInterceptor,
                               grpc.StreamStreamClientInterceptor):


    # def intercept_unary_unary(self, continuation, client_call_details, reques

    # def intercept_stream_stream(self, continuation, client_call_details,
request_iterator)
```

Logging client-side interceptor

```
def intercept_stream_stream(
    self,continuation, client_call_details, request_iterator
):

    response_it = continuation(
        client_call_details,
        self._intercept_request_stream_msg(request_iterator)
    )

    yield from self._intercept_response_stream_msg(response_it

    stream_duration = time.time() - self.stream_started

    print("Stream duration: {0} seconds".format(stream_duratio
```

Logging client-side interceptor

```
def _intercept_request_stream_msg(self, request_iterator):
        for r in request_iterator:
            print("Streaming request")
            yield r


def _intercept_response_stream_msg(self, response_iterator):
        for r in response_iterator:
            print("Streaming response")
            yield r
```

Client-side logs

```
Call details _ClientCallDetails(method='/Identity/ExpireToken’.

Streaming request
Streaming response

Streaming request
Streaming response

Streaming request
Streaming response

3.0214710235595703
```

Key takeaways

01

Web application middleware can be defined generally as an WSGI or ASGI application or be framework specific

02

gRPC interceptors is used to implement middleware in server and client applications

03

Code that’s acting as both a client and a server

04

Enables decoupling and sharing of non-functional requirements

Check out my PyCon US 2022 Talk!

Using middleware to:

- Migrate between WSGI frameworks

- Migrate between WSGI and ASGI frameworks

- More!

Thanks!

https://echorand.me

- Check out my books!
  - Doing Math with Python: https://doingmathwithpython.github.io
  - Practical Go: https://practicalgobook.net

DOING MATH WITH PYTHON
USE PROGRAMMING TO EXPLORE ALGEBRA, STATISTICS, CALCULUS, AND MORE!
AMIT SAHA