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”

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

When a request is coming in, this function will be executed before the view function

```
@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
```

When a response is going out, this function will be executed before the response is sent to the client

Result of middleware integration

```
$ FLASK_APP=flaskr poetry run flask run --port=5001

Page rendered in: 0.00033783912658691406 seconds
```

Our `@after_request` middleware

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):
        # Execute any custom code here before
        # processing the request
        try:
            response = self.get_response(request)
            # Execute any custom code here before
            # sending the response
        except:
            # return custom response
```

Activate middleware

```
# settings.py

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

Result of middleware integration

```
$ python manage.py runserver

Got exception: division by zero when processing
<WSGIRequest: GET '/polls/’>
```

Our `ExecHandlingMiddleware` middleware

Client sees:

`An exception occured!`

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
```

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

WSGI middleware is framework agnostic

Open source package example:

OpenTelemetry WSGI Instrumentation

This library provides a WSGI middleware that can be used on any WSGI framework (such as Django / Flask / Web.py) to track requests timing through OpenTelemetry.

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):
        self.app = app
        # other initialization

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

        if cache_hit:
            # send cached response

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

Your Fast API application

Calls the FastAPI application

Sending the cached response

```
if cached_response:
            await send({
                'type': 'http.response.start',
                'status': 200,
                'headers': [
                        [b'content-type', b'application/json'],
                        [b'x-cached-data', b'yes']
                ],
            })
            await send({
                'type': 'http.response.body',
                'body': cached_response,
            })
            return
```

Adding the Middleware

```
app = FastAPI()

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

Result of middleware integration

```
$ gunicorn app:app # ..other arguments

http:/expensive: Got request.
http: /expensive: Finished request. 10.002439737319946s.

http:/expensive: Got request.
http: /expensive: Finished request. 0.00028800964355468 75s.
```

First request

Second request

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. 0.00110793113708496 1s.

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

Recap

FastAPI, like Flask and Django defines helper methods to add ASGI middleware

ASGI middleware is framework agnostic

ASGI middleware works for both HTTP and WebSocket connections

Interceptors for gRPC applications

gRPC Applications

<u>Unary-Unary</u>

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

<u>Bidirectional streaming</u>

- 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-id

        return user_details


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

A logging interceptor

```
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_metadata
        )

        return continuation(handler_call_details)
```

Next interceptor or RPC method

Request Metadata

Integrating the interceptor(s)

```
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; c
```

Bidirectional 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, context):
            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

Create an iterator for a single request

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
```

Key takeaways

01

Enables decoupling and sharing of non-functional requirements

02

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

03

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

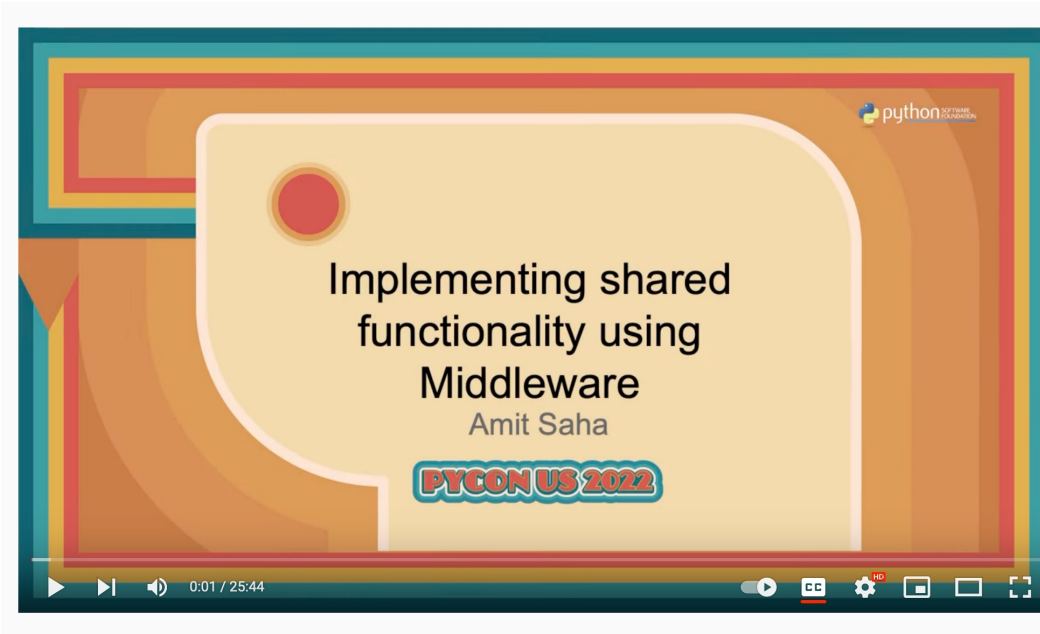
04

Implementing gRPC interceptors varies based on the pattern of communication

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