

Shared Functionality using Middleware

In your WSGI and ASGI applications

Amit Saha

<https://echorand.me>



Agenda

Middleware
in Computing

WSGI
Middleware

ASGI
Middleware

Slides and Resources

<https://echorand.me/talks/>

Origin of “middleware”

Usage in computing as early as 1968

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 \(PDF\)](#)

“..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.

Key relevant ideas from back then

- “Glue”
- “interface between hardware and software”
- IETF workshop in 2000: *middleware as those services found above the transport layer .. but below the application environment..*

Today's working definition

.. 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.

PEP 333 – Python Web Server Gateway Interface v1.0

Middleware for WSGI applications

A Flask Application

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

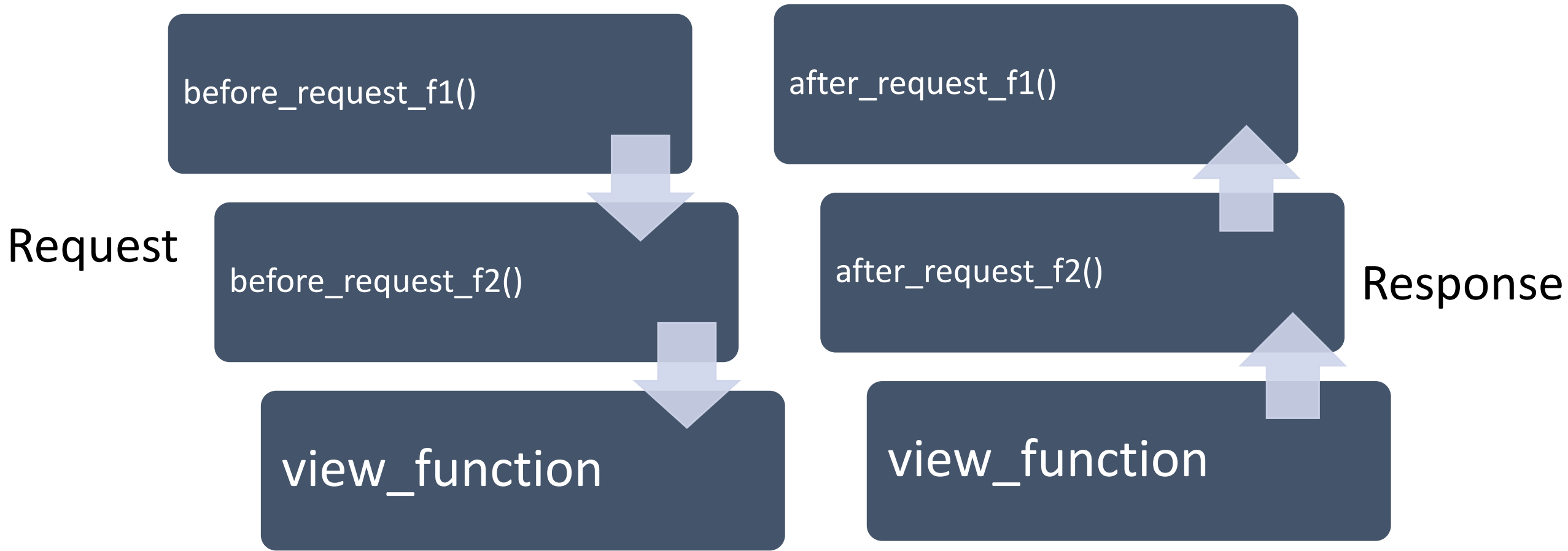
@bp.route("/")
def index():
    return render_template("blog/index.html", posts=posts)
```


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
```

Multiple middleware in Flask



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
```

Django middleware – function based

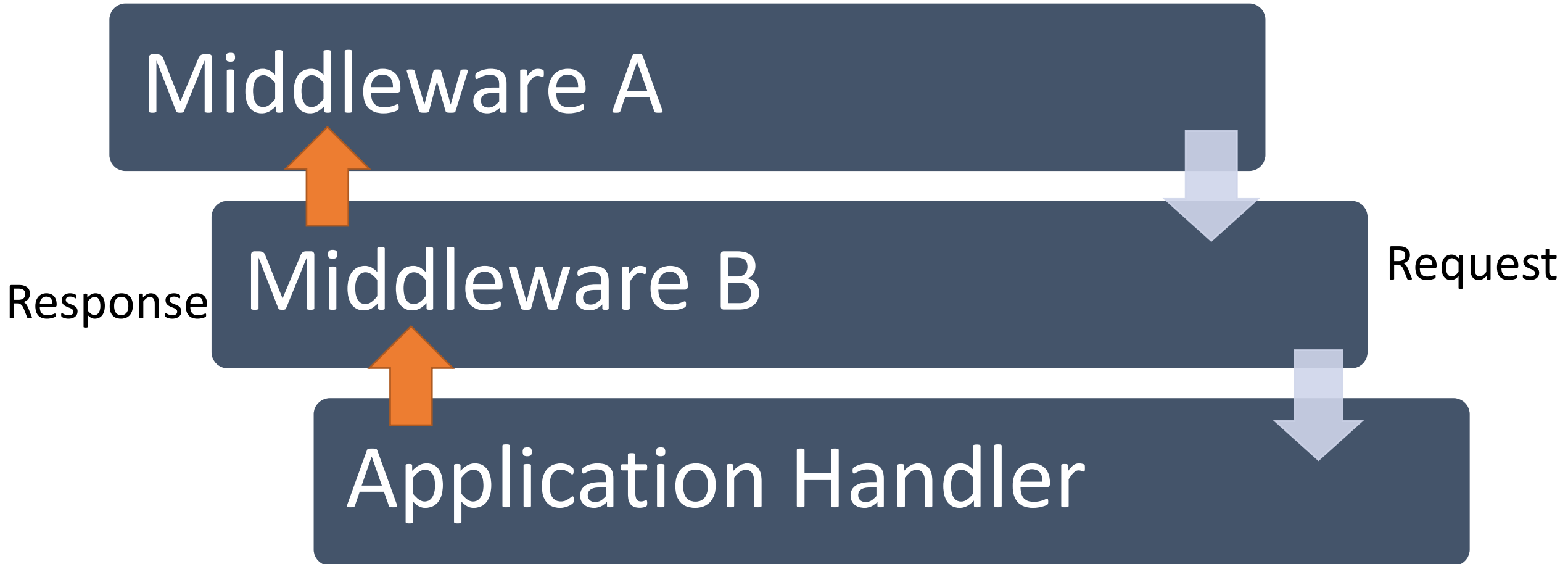
```
def latency_reporter(get_response):  
    def middleware(request):  
        request_begin = time.time()  
        response = get_response(request)  
        print("Latency:f{time.time()-request_begin} seconds")  
        return response  
    return middleware
```

Activate middleware

```
# settings.py
```

```
MIDDLEWARE = [  
    'polls.latency_reporter.latency_reporter',  
    'polls.my_exc_handler.ExHandlingMiddleware'  
]
```

Multiple middleware in Django



Recap

WSGI Frameworks define their own mechanism to define middleware

You define custom code to run before and after request processing

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

OpenTelemetry WSGI Middleware

```
class OpenTelemetryMiddleware:

    def __init__(self, wsgi):
        self.wsgi = wsgi
        # other things

    def __call__(self, environ, start_response):
        try:
            iterable = self.wsgi(environ, start_response)
        except Exception as ex:
            # do other stuff
```

Flask + WSGI Middleware

```
from opentelemetry.instrumentation.wsgi \
    import OpenTelemetryMiddleware
```

```
app = Flask(__name__)
```

```
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)
```

Django + WSGI Middleware

```
# wsgi.py
```

```
from opentelemetry.instrumentation.wsgi \
    import OpenTelemetryMiddleware
```


```
application = get_wsgi_application()
```

```
application = OpenTelemetryMiddleware(application)
```


Pause

Middleware for wrapping another WSGI application

You have a Django application,
and moving some views to a Flask
backend



Write a middleware to
include the Django app inside
your Flask app

Wrapper Middleware

```
class FlaskAppWrapper(object):  
  
    def __call__(self, environ, start_response):  
        data = flask_app.app.wsgi_app(  
            environ, start_response  
        )  
        if not self.status.startswith('404'):  
            return data  
        return self.wsgi_app(environ, start_response)
```

Wrapping Django Application

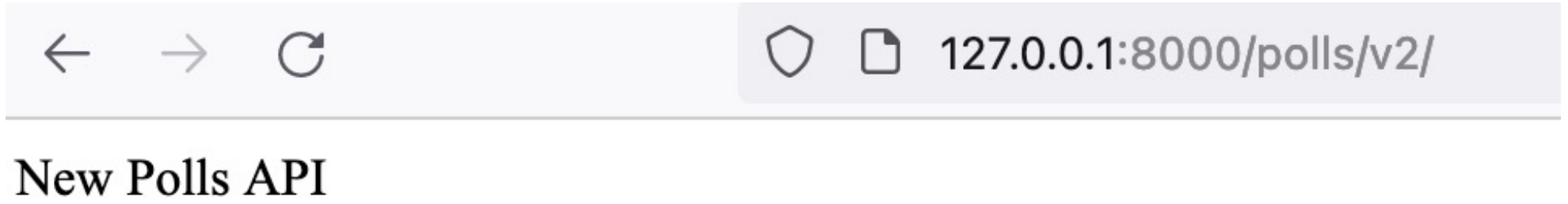
```
# wsgi.py
```

```
application = get_wsgi_application()
```

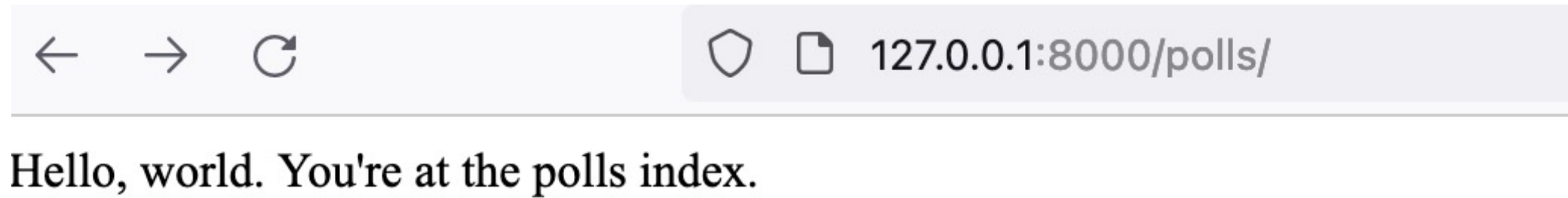
```
application = FlaskAppWrapper(application)
```

Wrapping a WSGI application

Flask



Django



Recap

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

WSGI middleware is framework agnostic

Use the wrapping technique to include WSGI middleware

Middleware for ASGI applications

An ASGI HTTP application

```
async def simple_handler(scope, receive, send):  
  
    request = await receive()  
  
    await send({  
        'type': 'http.response.start',  
        'status': 200,  
        'headers': [...]  
    })  
  
    await send({  
        'type': 'http.response.body',  
        'body': b'Hello, world!',  
    })
```


An ASGI Middleware

```
class RateLimiter(object):  
  
    def __init__(self, asgi_app):  
        self.asgi_app = asgi_app  
  
    async def __call__(self, scope, receive, send):  
        if self.allowed(scope):  
            await self.asgi_app(scope, receive, send)  
        else:  
            # Send custom response
```

ASGI application with middleware

```
app = RateLimiter(simple_handler)
```

```
$ gunicorn app:app --worker-class uvicorn.workers.UvicornWorker
```

```
2022-04-26 13:04:14 +1000] [74357] [INFO] Starting gunicorn 20.1.0
[2022-04-26 13:04:14 +1000] [74357] [INFO] Listening at: http://0.0.0.0:8000 (74357)
[2022-04-26 13:04:14 +1000] [74357] [INFO] Using worker: uvicorn.workers.UvicornWorker
```

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. 0.001107931137084961s.

websocket:/ws: Got request.

..

websocket: /ws: Finished request. 28.716175079345703s.

FastAPI/Starlette specific approach

```
@app.middleware("http")
async def exc_handler(request: Request, call_next):
    try:
        return await call_next(request)
    except Exception as e:
        return HTMLResponse(
            'An error occurred!',
            status_code=500,
            headers={'x-exception-handled': 'application'}
        )
```

Framework Implementation

```
# starlette/middleware/base.py

def middleware(self, middleware_type):

    def decorator(func):
        self.add_middleware(BaseHTTPMiddleware, dispatch=func)
        return func

    return decorator
```


Pause

Integrate a WSGI application

```
from fastapi.middleware.wsgi \
    import WSGIMiddleware
```

```
app = FastAPI()
```

```
app.mount("/v1", WSGIMiddleware(flask_app))
```

```
$ curl localhost:8000/v1/  
Hello world from Flask!
```

```
$ curl localhost:8000/v2/  
{"message": "Hello from FastAPI"}
```

WSGIMiddleware Implementation

```
# starlette/middleware/wsgi.py
```

```
class WSGIMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        responder = WSGIResponder(self.app, scope)
        await responder(receive, send)
```

Recap

ASGI middleware is framework agnostic

FastAPI has helper methods to add ASGI middleware

WSGIMiddleware allows you to forward requests to an WSGI application

Key takeaways

01

Can be defined generally as an WSGI or ASGI application or be framework specific

02

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

03

Enables decoupling and sharing of non-functional requirements

Exercise!

Check out the source code of the built-in and community middleware in Flask, Django and FastAPI!

Thanks!

- <https://echorand.me>
- mail@echorand.me
- Check out my books!
 - Practical Go: <https://practicalgobook.net>
 - Doing Math with Python: <https://doingmathwithpython.github.io>

