Implementing Shared Functionality using Middleware

In your WSGI, ASGI and gRPC applications

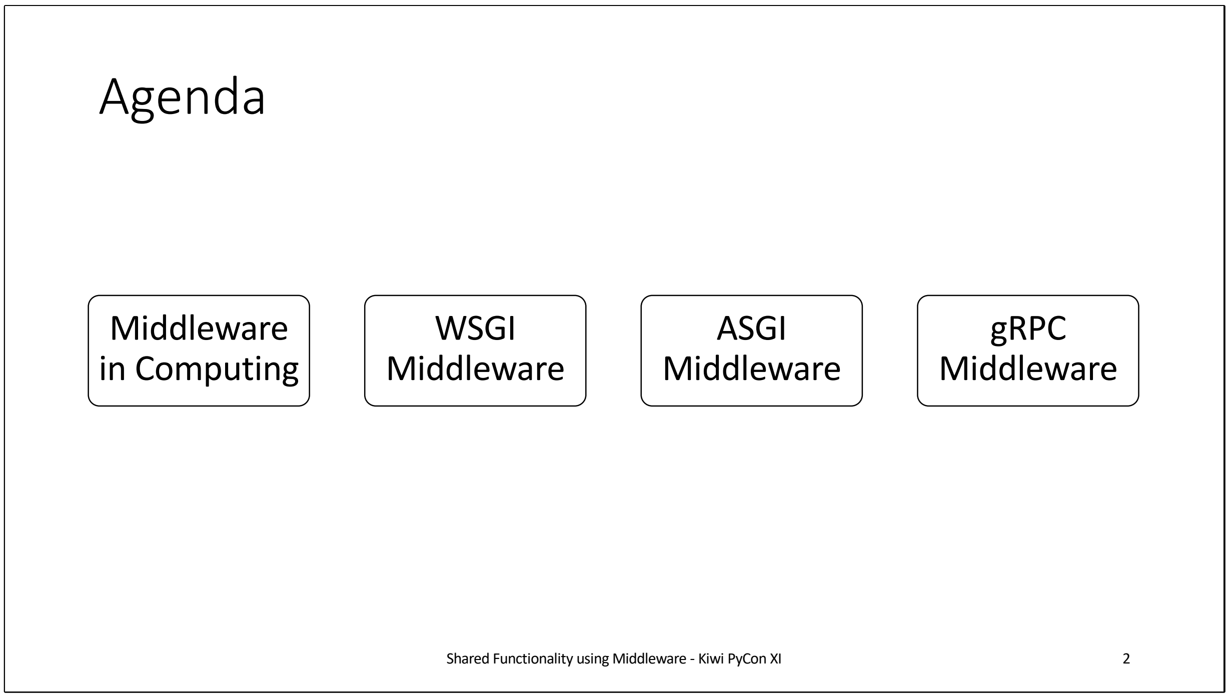
Amit Saha

https://echorand.me

kiwipycon

Hello everyone, welcome to my talk, “implementing shared functionality using middleware”. My name is Amit Saha and I work as a software engineer at Atlassian, in Sydney, Australia. I am very excited to be presenting to you – my first kiwipycon, would have loved to be there in person, but a certain virus didn’t want me to.

Like all of you, I *love* figuring out how things work and that brings me to the agenda for my presentation today.



Agenda

Middleware in Computing

WSGI Middleware

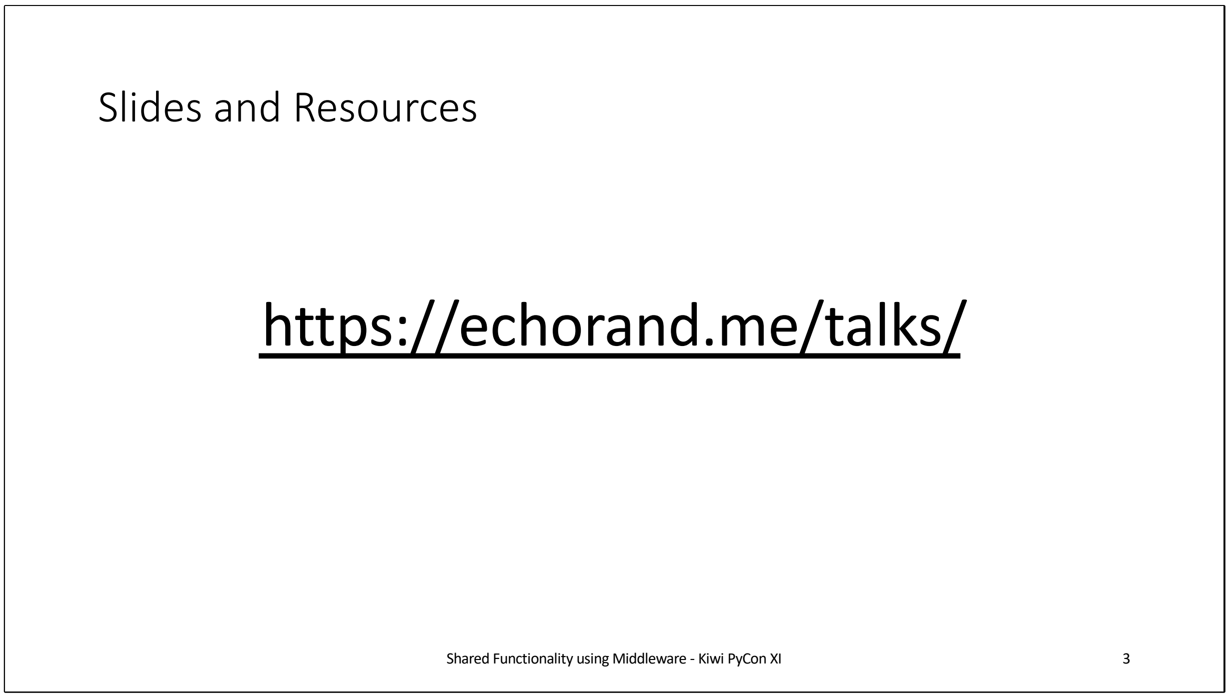
ASGI Middleware

gRPC Middleware

Shared Functionality using Middleware - Kiwi PyCon XI 2

My goal in this talk is to take you through an exploration how middleware works in the context of web and gRPC applications today. Preparing this talk was an insightful journey for me, full of ah ha moments – and I will hopefully reproduce it with high fidelity given the constraints of time.

We start with a very brief history of the origin of term middleware in computing, then we learn how middleware works in WSGI applications, followed by how middleware works in ASGI applications and finally gRPC applications.

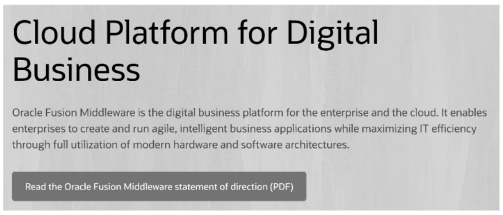


Slides and Resources

https://echorand.me/talks/

Let’s follow our curiosity!

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)

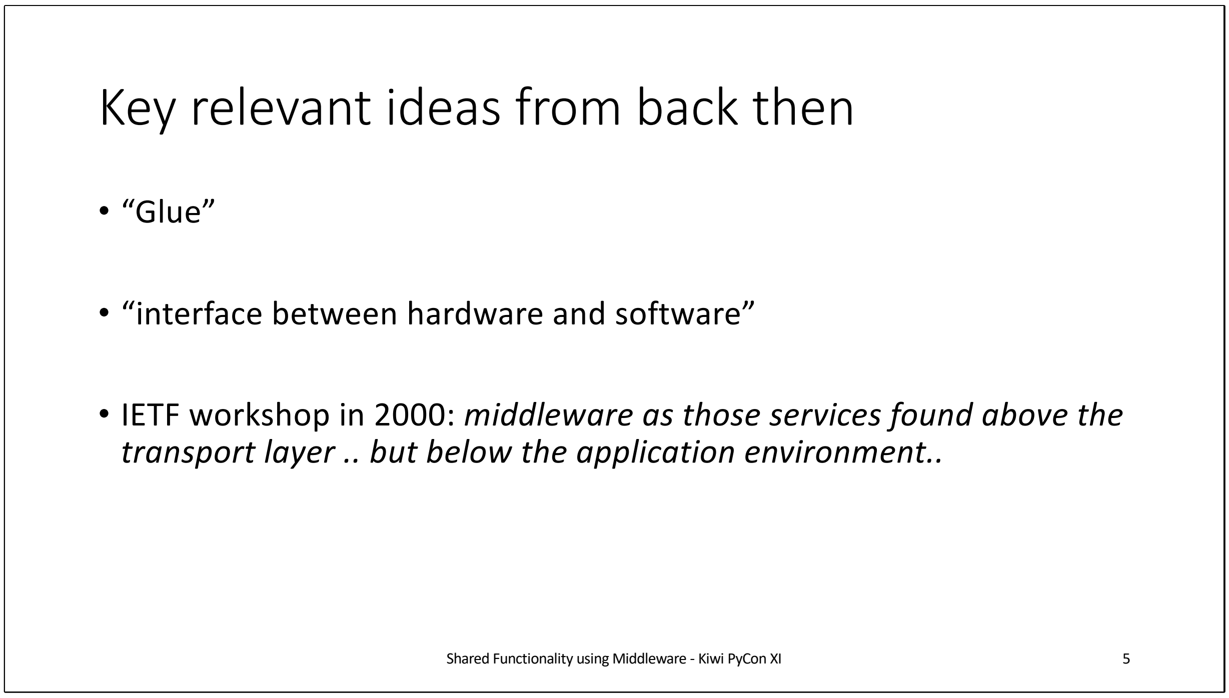
Usage in computing as early as 1968

“..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.

Depending on the time you have spent in the software world, you may have heard of the term middleware outside of web applications. I remember getting free CDs for BEA Weblogic circa, early 2000s, didn’t do anything with them as far as I recall. The usage of the term can be found in computing as early as 1968. A book on the topic, describes middleware as the “-” in client-server or the –to- in peer-to-peer.

Of course, the term middleware is still in use today in that context, such as a product by Oracle, called Oracle Fusion Middleware

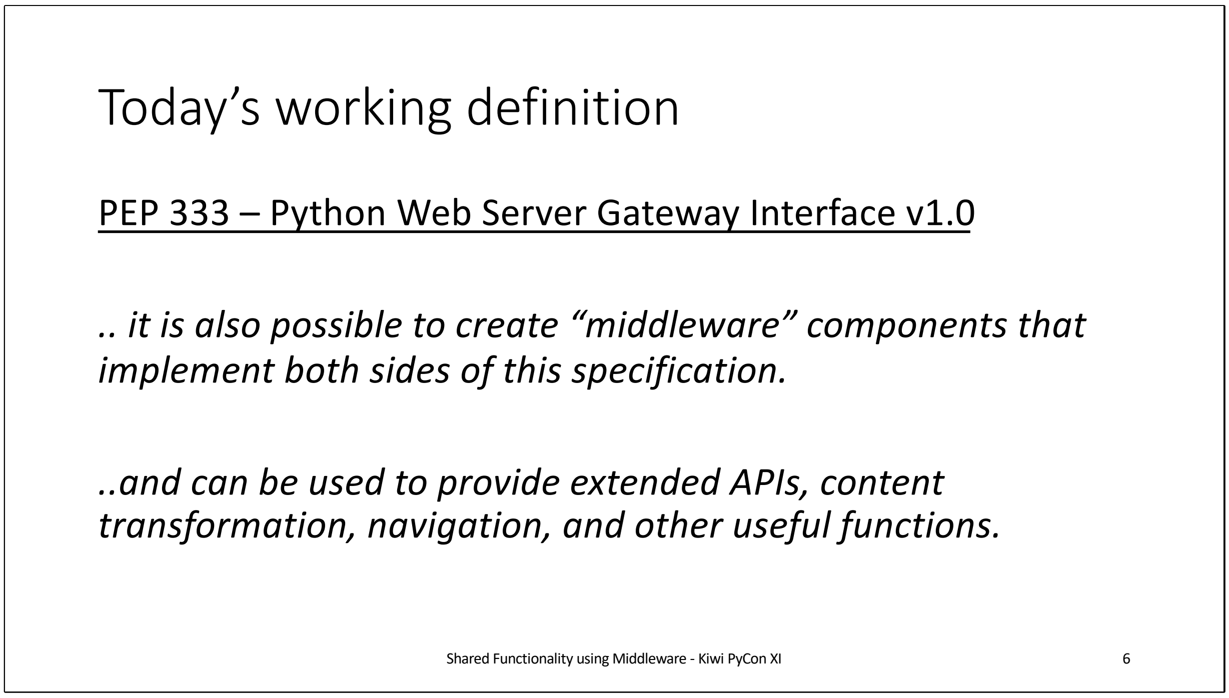


Key relevant ideas from back then

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

Shared Functionality using Middleware - Kiwi PyCon XI 5

Although the usage of the term in the previous slide is quite far from what we will be discussing today, some key ideas from back then are relevant today I think. Sure, we are not going to talk about hardware today, but there is something that seems quite applicable to be when I discovered that middleware was referred to as "glue", "an interface between hardward and software" or "services sitting above the transport layer and below the application environment".



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.

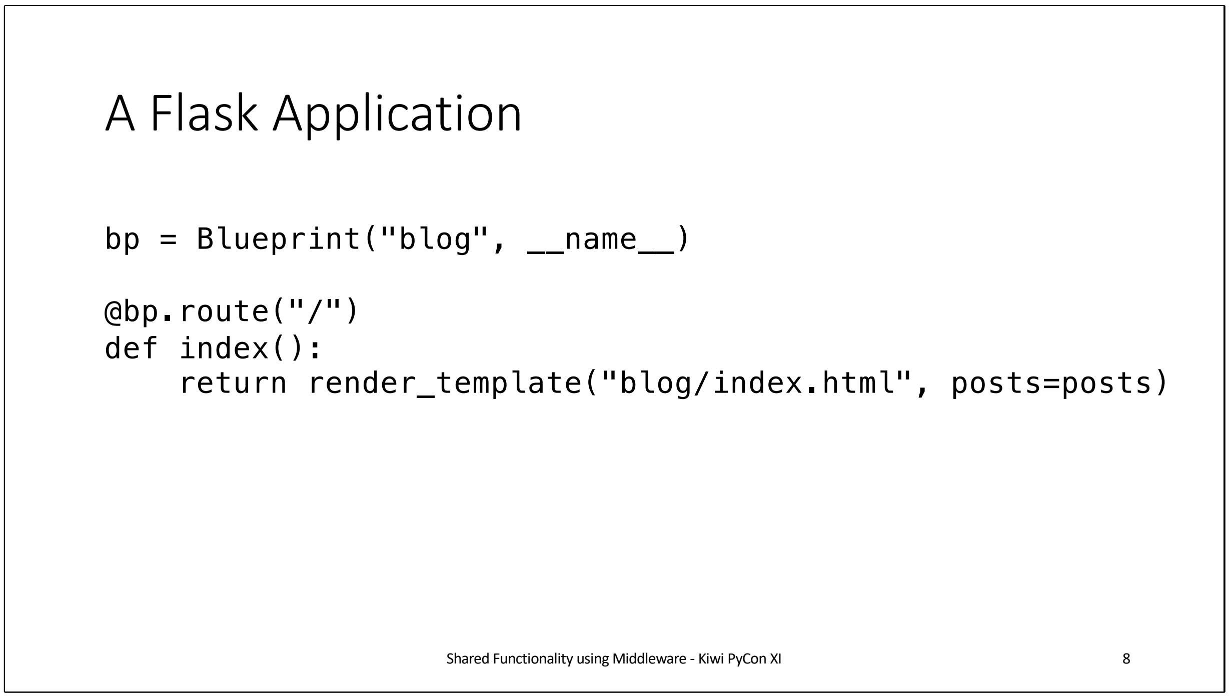
..and can be used to provide extended APIs, content transformation, navigation, and other useful functions.

Consider a web application consists of one or more API endpoints or views:
Middleware is code you write to implement functionality that is common across one or more endpoints or views
-Count of page views
-Error handling
-Caching, and many others

-The WSGI specification ..

-This leads us to our second topic of the talk, middleware for WSGI applications.

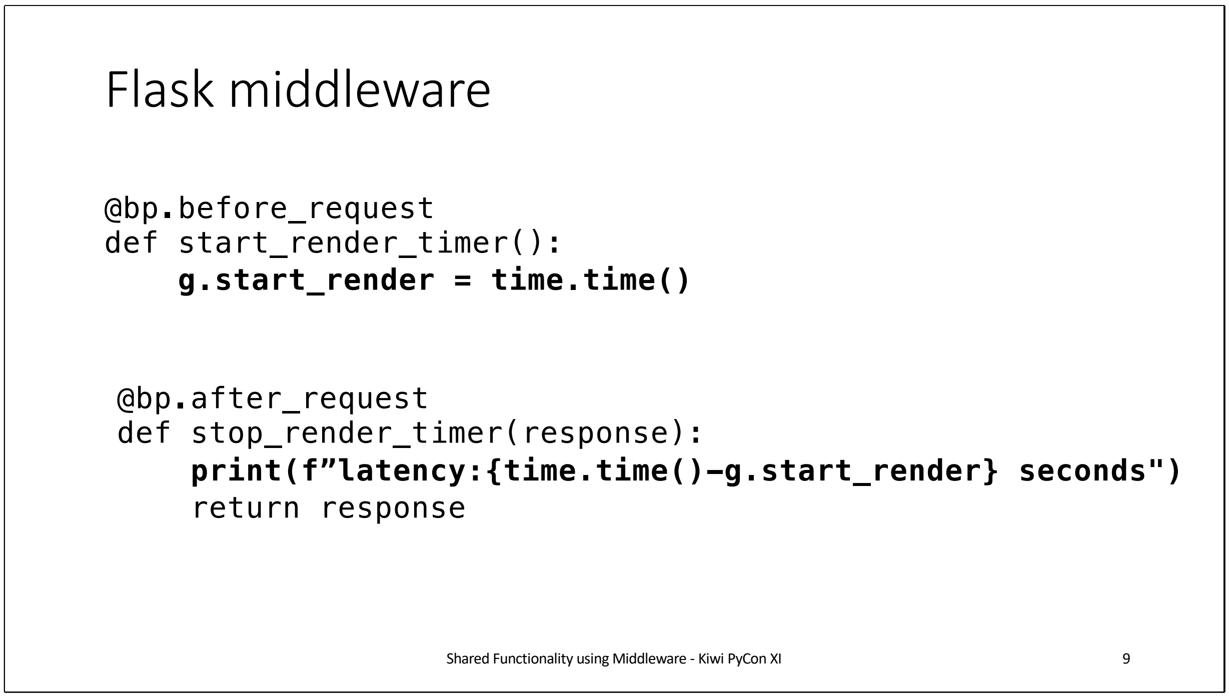
Middleware for WSGI applications



A Flask Application

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

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

Consider a Flask application. Here I create a Blueprint, and then define a route. Of course, I have omittied a lot of the code that’s not relevant here. Simple. We will refer to the index() function as the view function, the one handling the application requests.

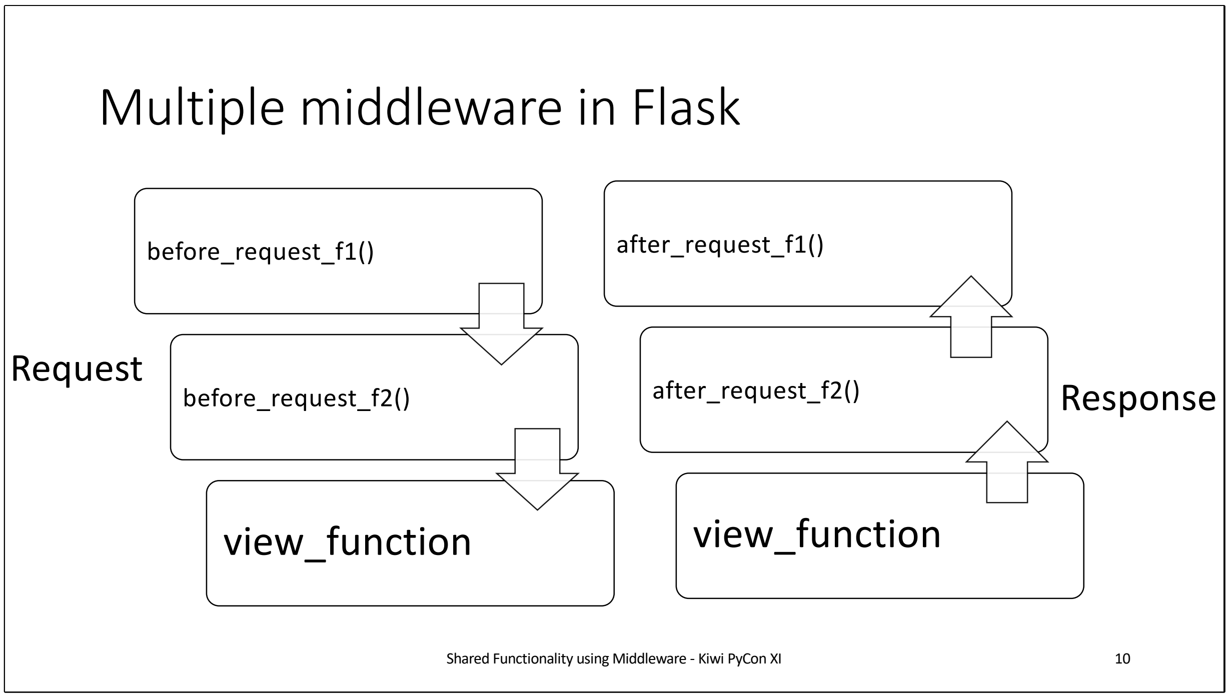


Flask middleware

```
@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
```

This is how we define middleware for a Flask application, using the before_request and after_request decorators. When a function is decorated with the before_request decorator, it is called before a request is processed by the view function if one is defined. The after_request function is called after the request has been processed by the view function.

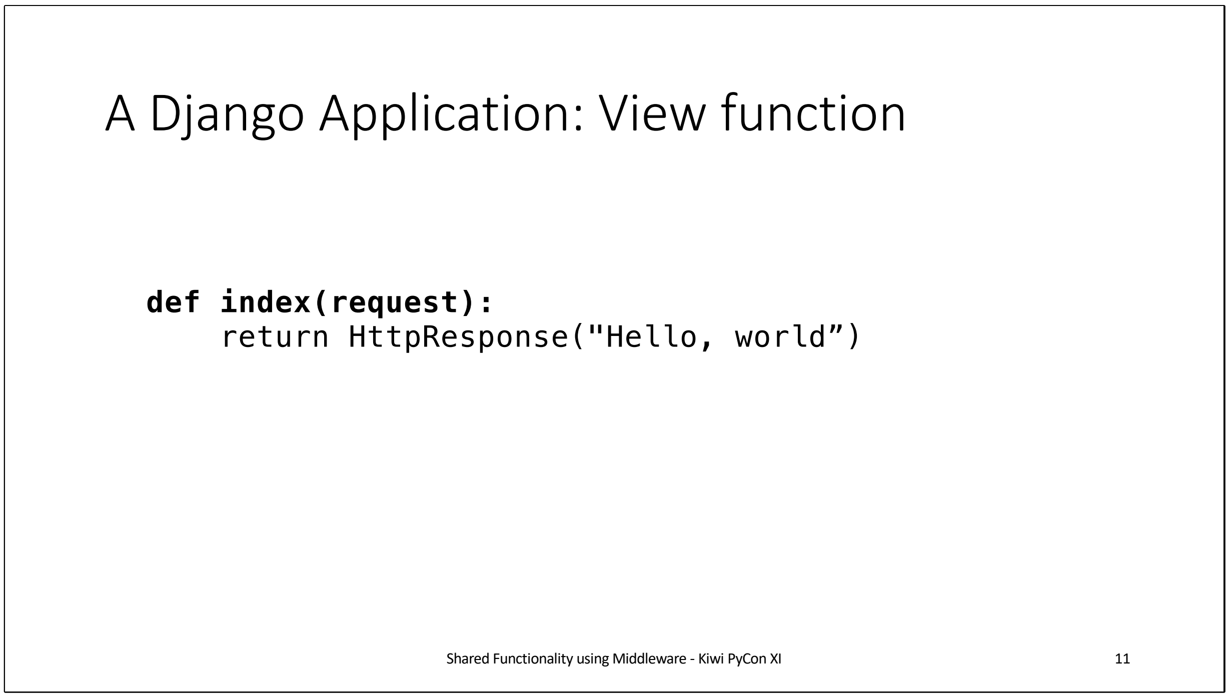
The same applies to non-Blueprint applications, by the way in case you are wondering.



Multiple middleware in Flask

before_request_f1()
before_request_f2()
view_function
Request
after_request_f1()
after_request_f2()
view_function
Response

The ordering specified middleware is significant. For Flask, the middleware defined via before_request are called in order. The middleware defined via after_request are called in reverse_order.

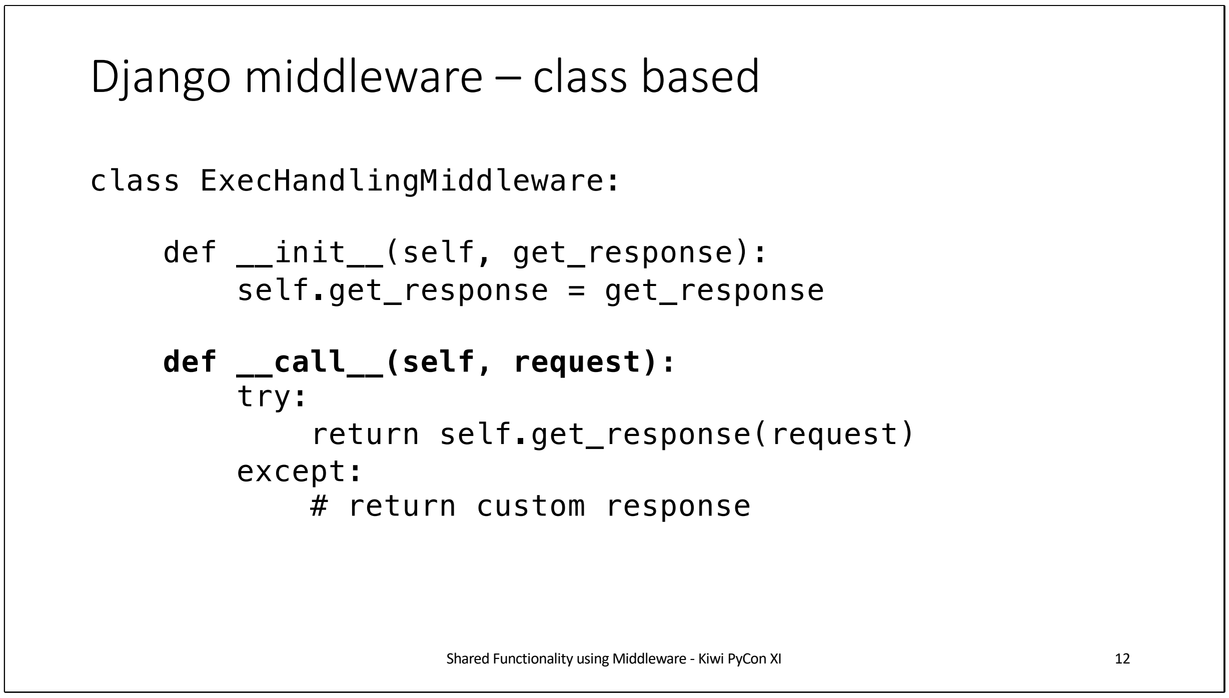


A Django Application: View function

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

Shared Functionality using Middleware - Kiwi PyCon XI 11

Let’s consider a Django view function. To define a middleware in Django, we can use a define a class or a function.



Django middleware – class based

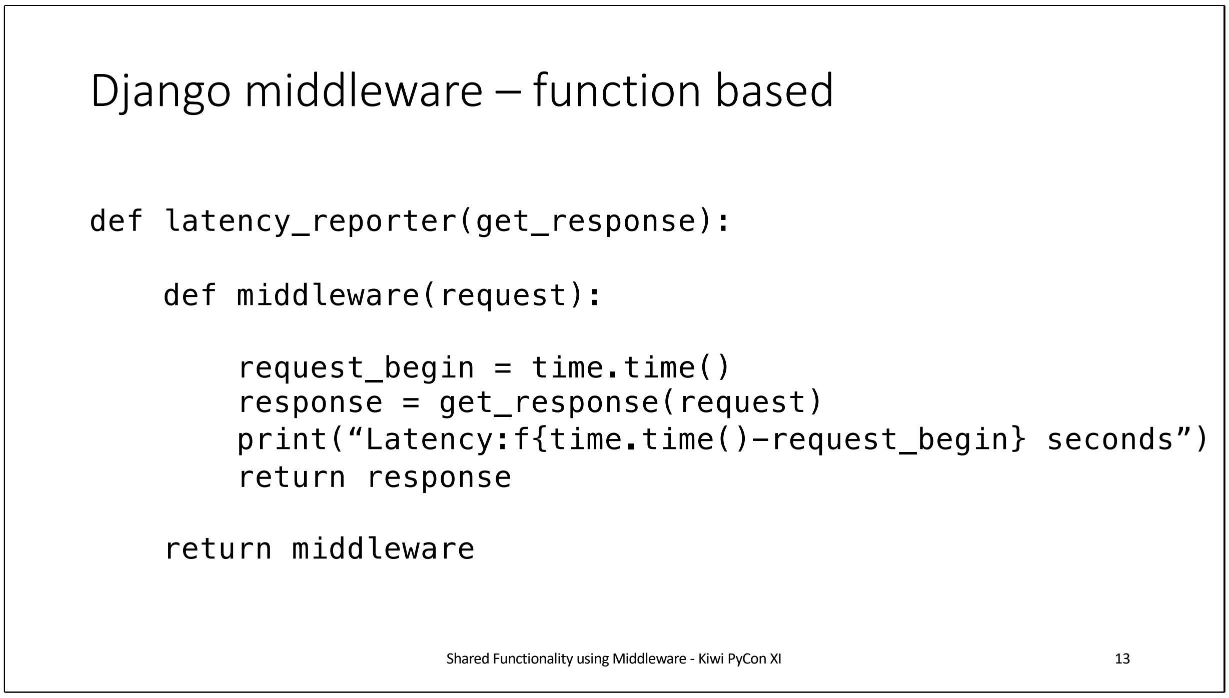
```
class ExecHandlingMiddleware:

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

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

Shared Functionality using Middleware - Kiwi PyCon XI 12

A class based Middleware in Django looks as follows.

The key is the special __call__ method which takes one argument the current request being handled, this is exactly how the view function looks as well in the previous slide.

The get_response attribute points to the view function that will be called or another middleware to be called next.

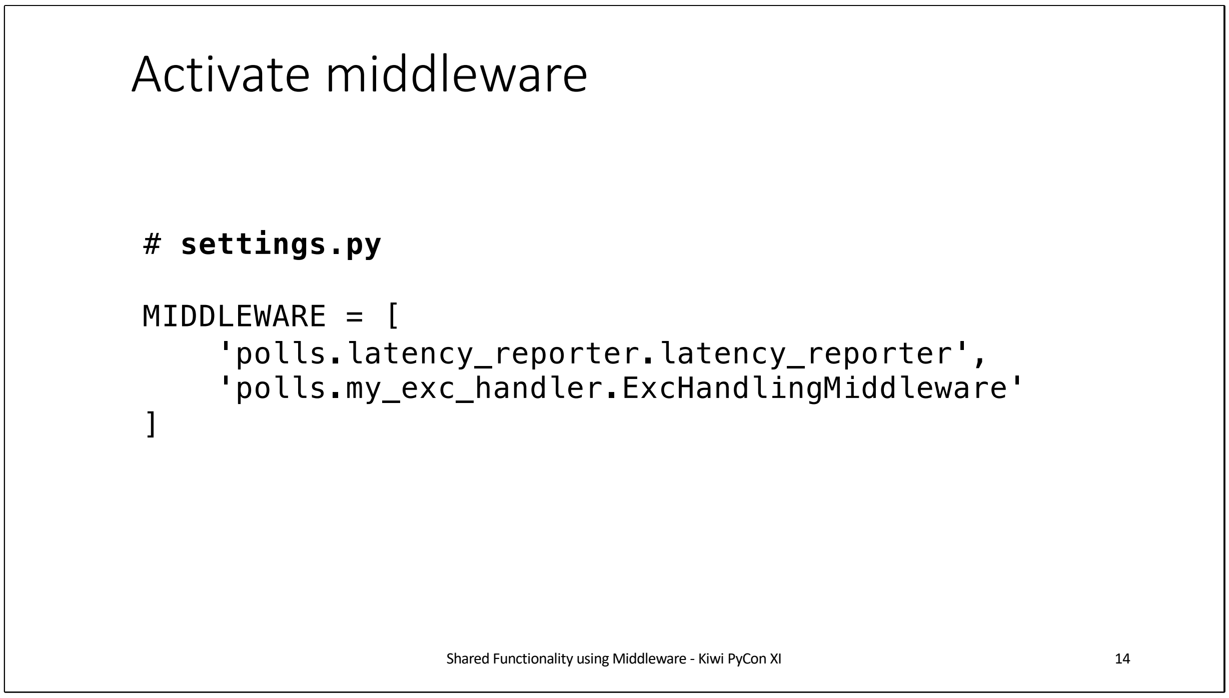


Django middleware – function based

```
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
```

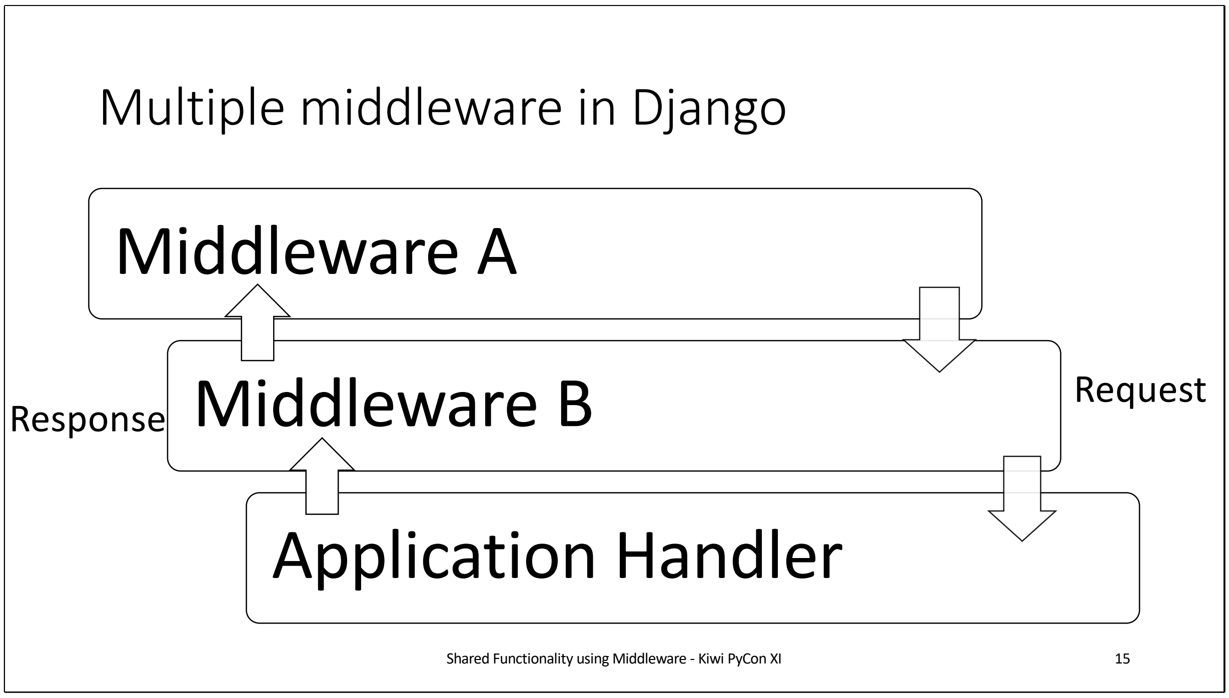
Shared Functionality using Middleware - Kiwi PyCon XI 13

A function based middleware returns another function – a closure as shown in the example.


Activate middleware

```
# settings.py

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

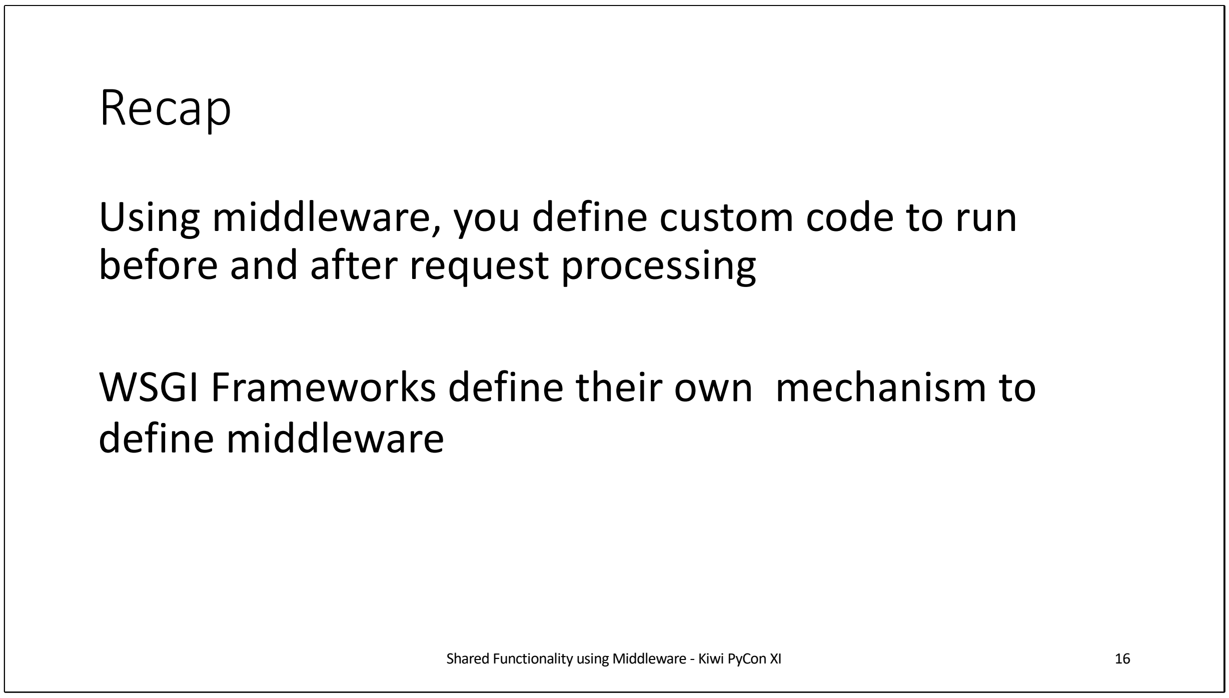
Then, we activate it by including them via their path in settings.py. The ordering here matters, as does in the case of Flask applications.



Multiple middleware in Django

For Django, the ordering of the elements in the list in the previous slide matters. For an incoming request, the first middleware is called before the next middleware before the view function is called. Once a response has been generated, the middleware defined are invoked in the reverse order.

One key difference betweek Flask and Django middleware is that in Flask, your middleware may not be invoked for both request and the response to the request where as in Django, a middleware is always invoked when the request is being sent to the view function and when the response is being sent back.

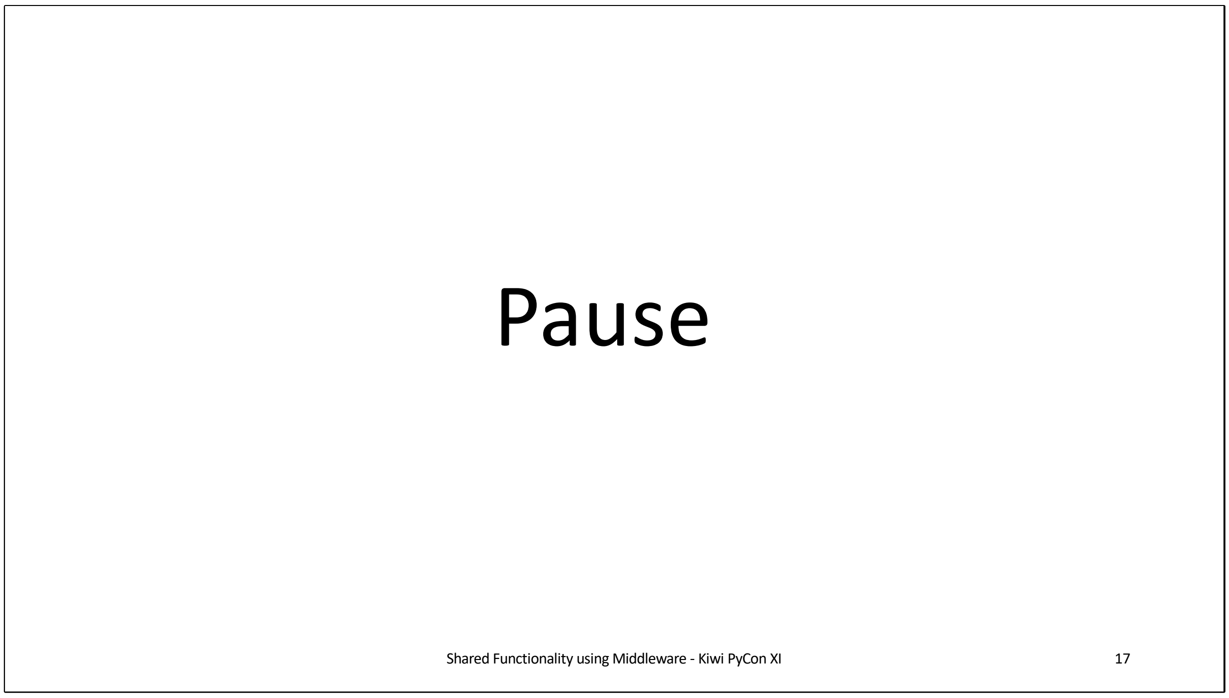


Recap

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

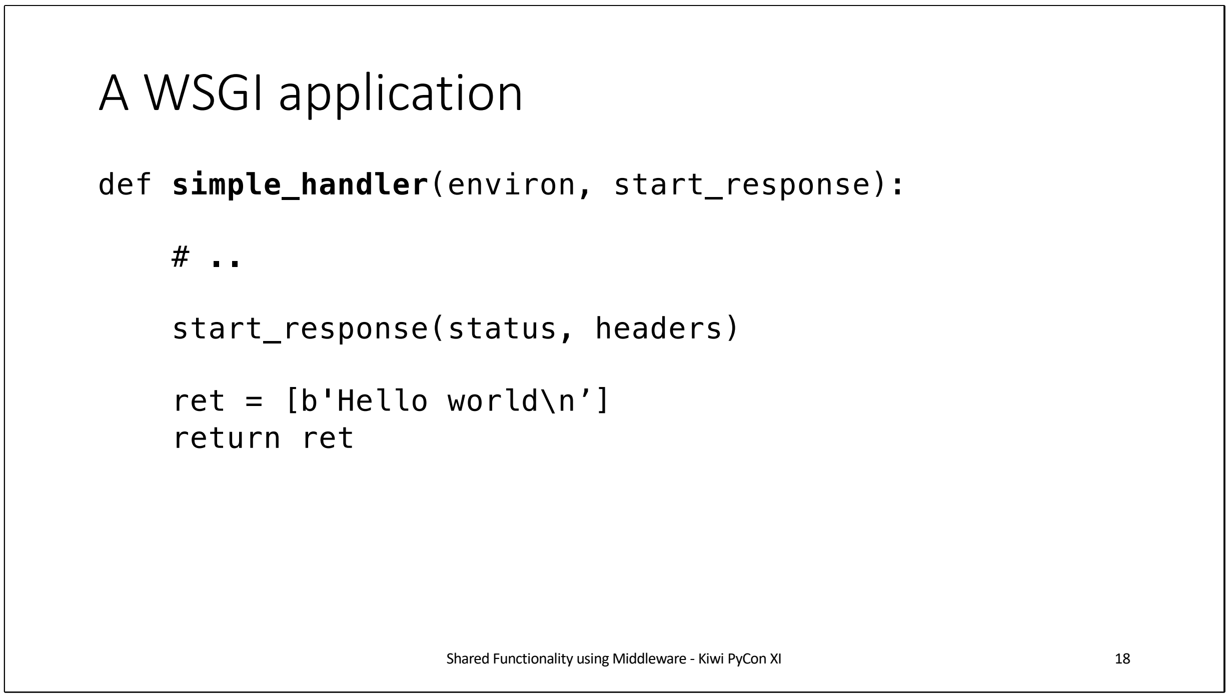
WSGI Frameworks define their own mechanism to define middleware

How the middleware works is a framework specific detail which usually you can find out by looking up the framework's source code. I would have loved to include it today, but time constraints!



Pause

As we have just seen, the way we write a middleware for Flask is different from Django. Could we write middleware in a framework independent way? It turns out we can. These are WSGI frameworks, which means they define WSGI applications..and there lies the answer as we shall see next.

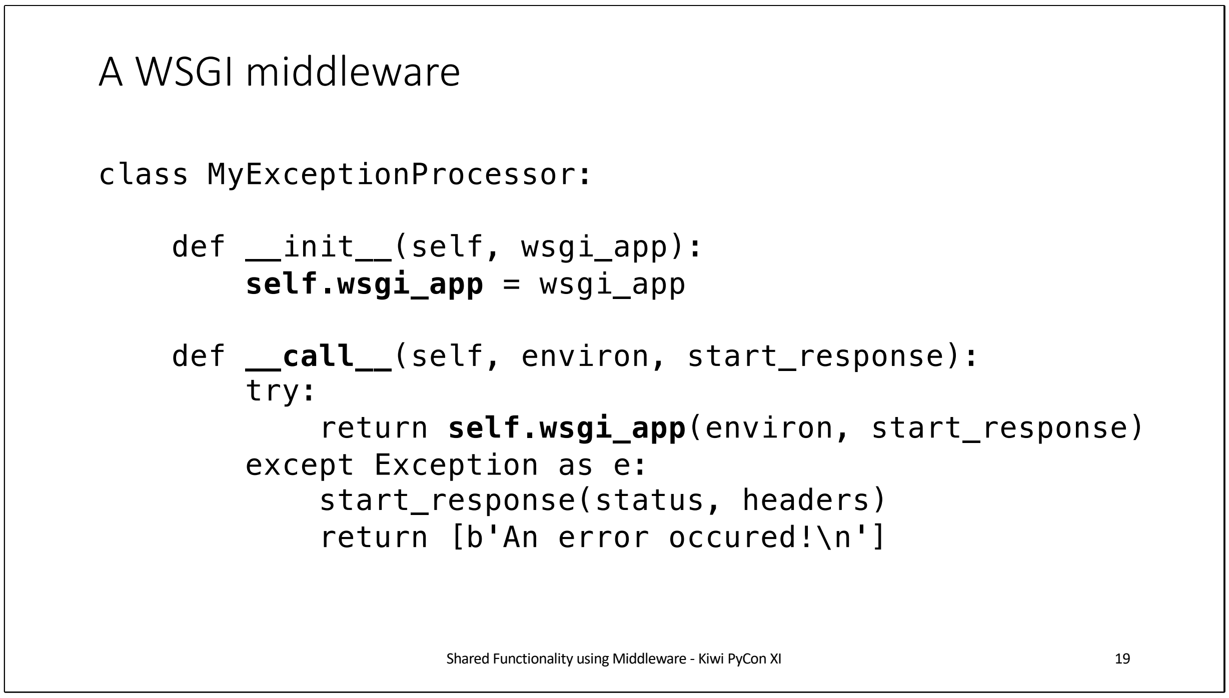

A WSGI application

```
def simple_handler(environ, start_response):

    # ..

    start_response(status, headers)

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

I want to start by taking a step back. Irrespective of which WSGI framework you are using, somewhere hidden underneath there is a function like this defined which kickstarts everything in your application. This is an WSGI application.

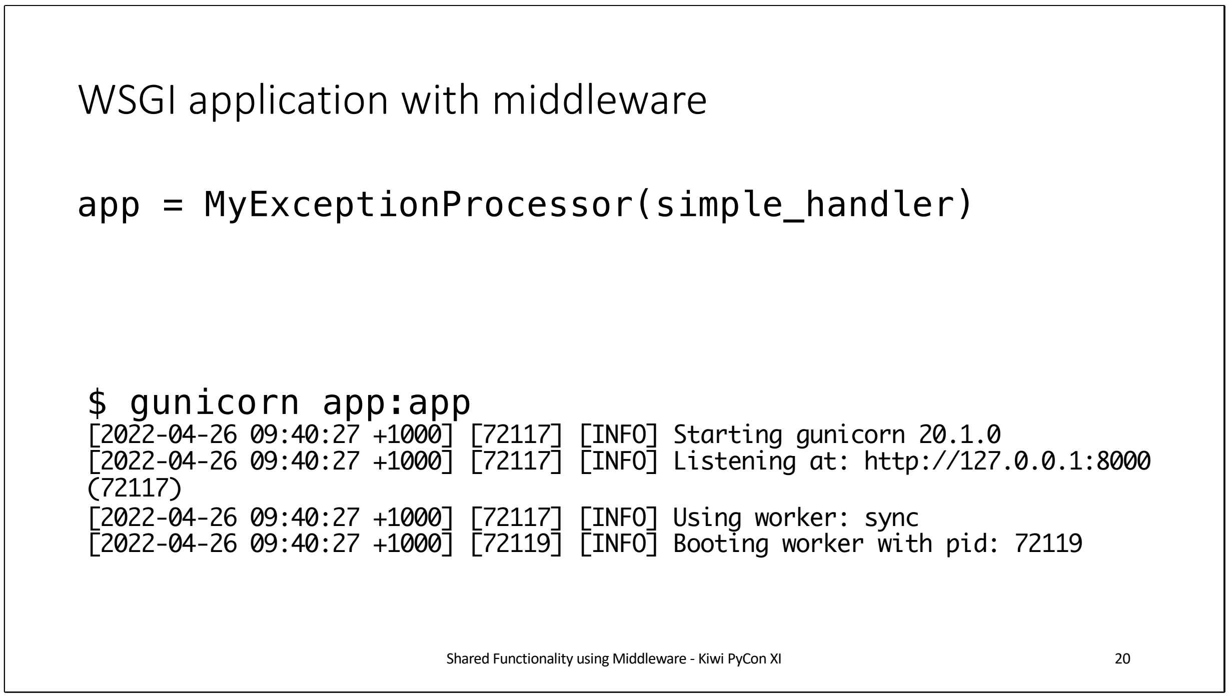
And this function signature is going to be the key for us to better understand WSGI middleware. You can ignore everything for this part of this talk, except for the function signature.

environ is a dictionary containing various key value pairs describing the current request. And start_response is a function that is used to send back a response to the client.



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']
```

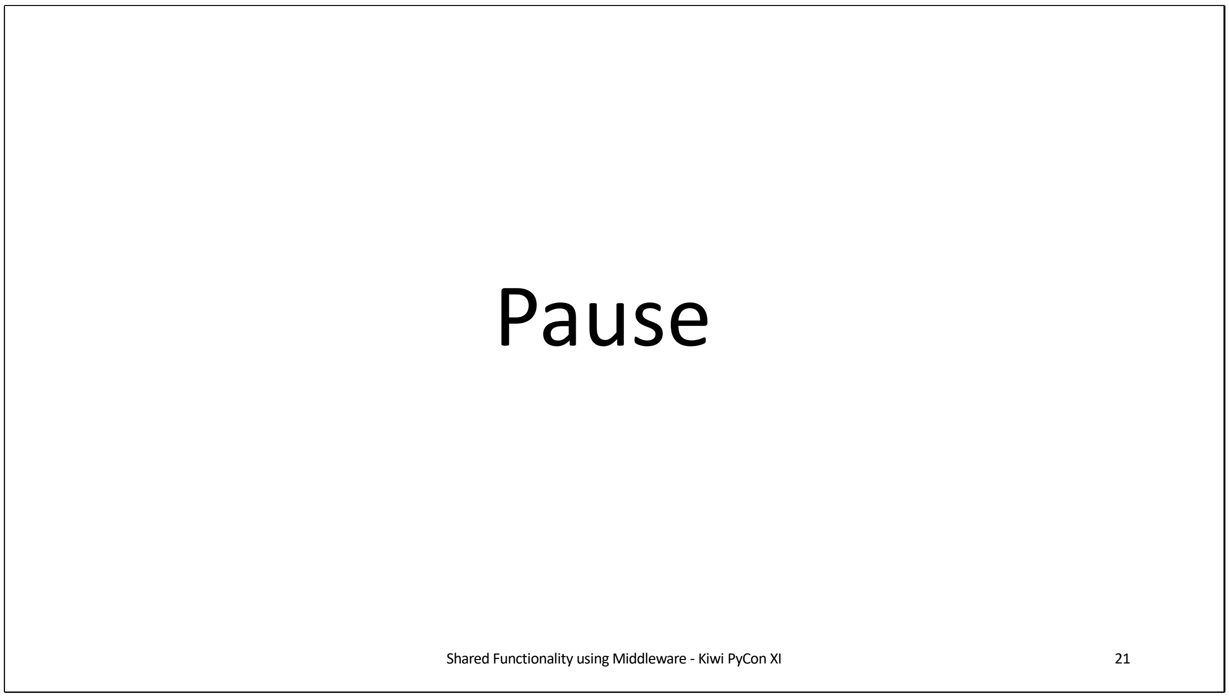


WSGI application with middleware

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

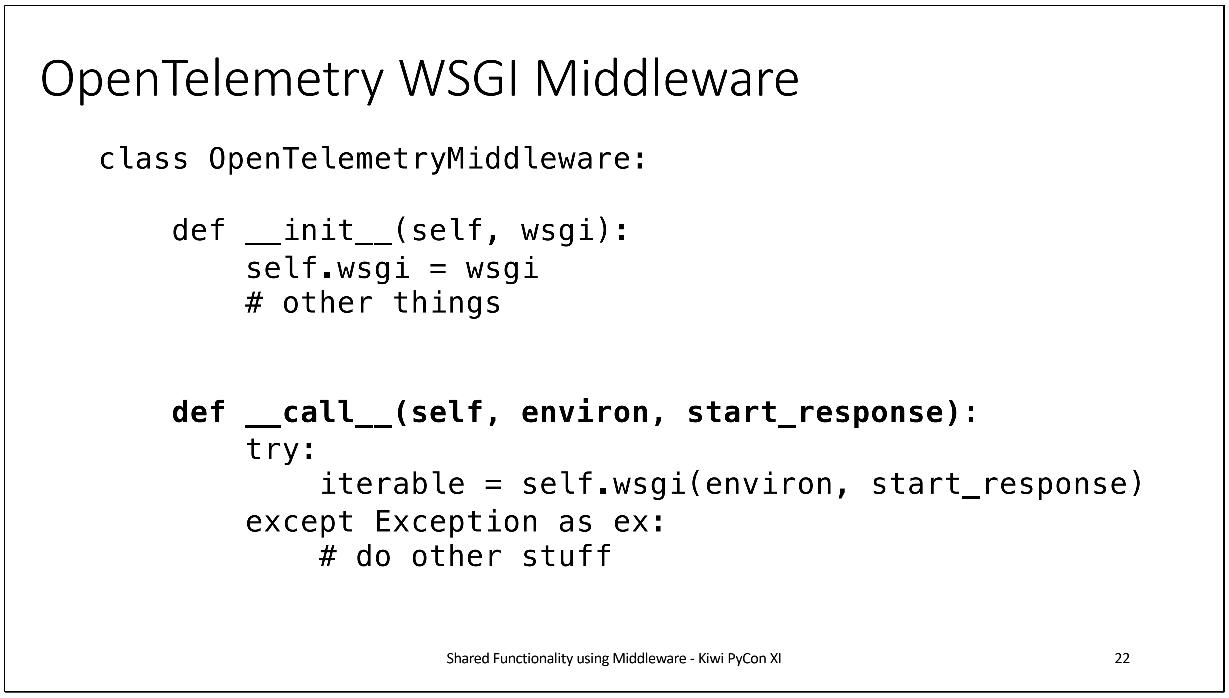
```
$ gunicorn app:app
[2022-04-26 09:40:27 +1000] [72117] [INFO] Starting gunicorn 20.1.0
[2022-04-26 09:40:27 +1000] [72117] [INFO] Listening at: http://127.0.0.1:8000
(72117)
[2022-04-26 09:40:27 +1000] [72117] [INFO] Using worker: sync
[2022-04-26 09:40:27 +1000] [72119] [INFO] Booting worker with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

I am going to refer to this as the wrapping technique of defining middleware for WSGI applications



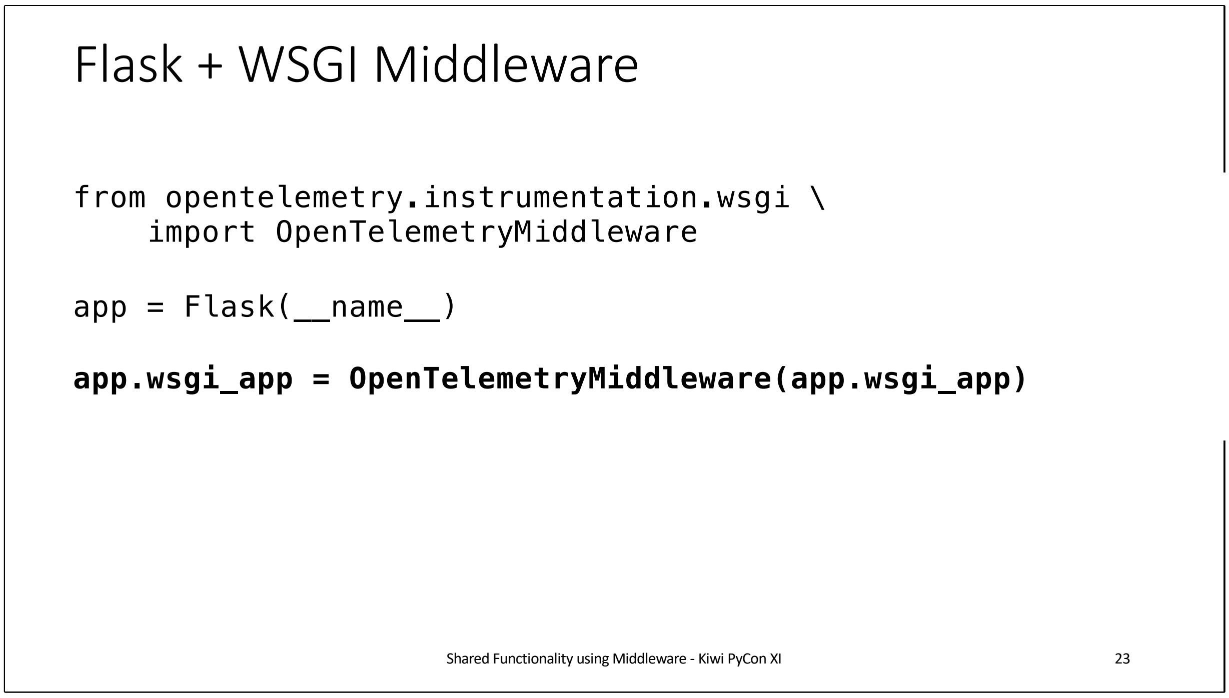
Pause

Frameworks implement their own mechanisms to define middleware, WSGI middleware = Another WSGI application, So that means if we implement a functionality using WSGI middleware, they are framework independent, let’s see how they work.



OpenTelemetry WSGI Middleware

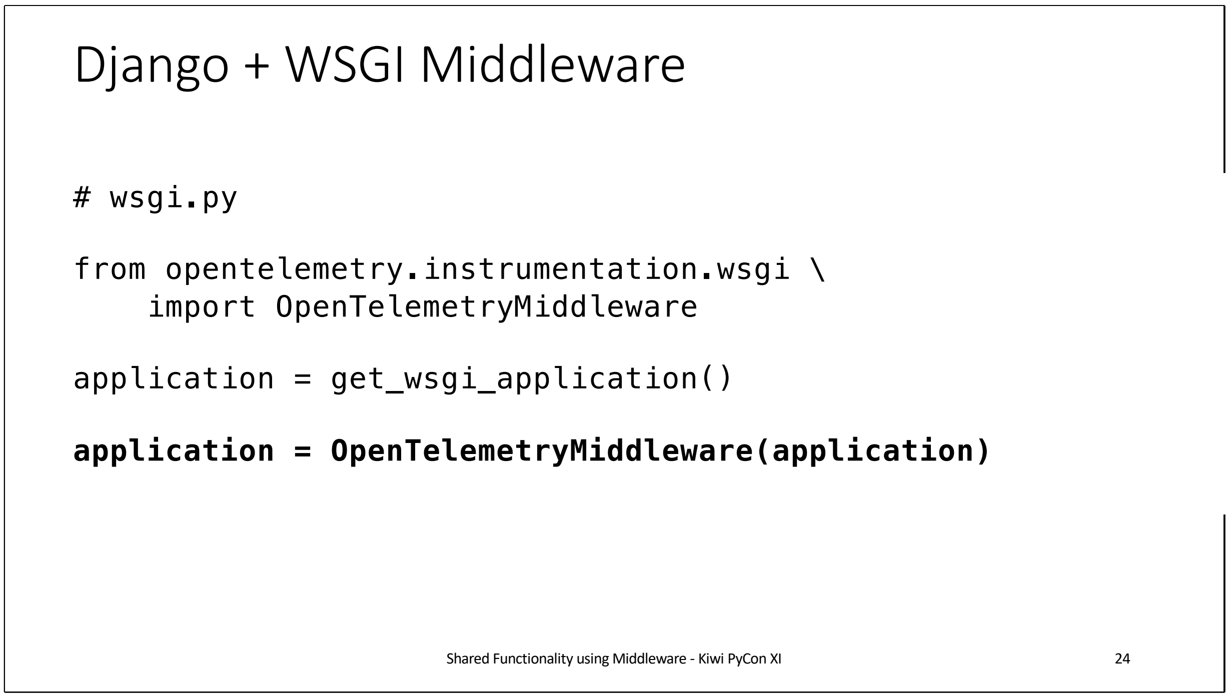
```
class OpenTelemetryMiddleware:

    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)
```

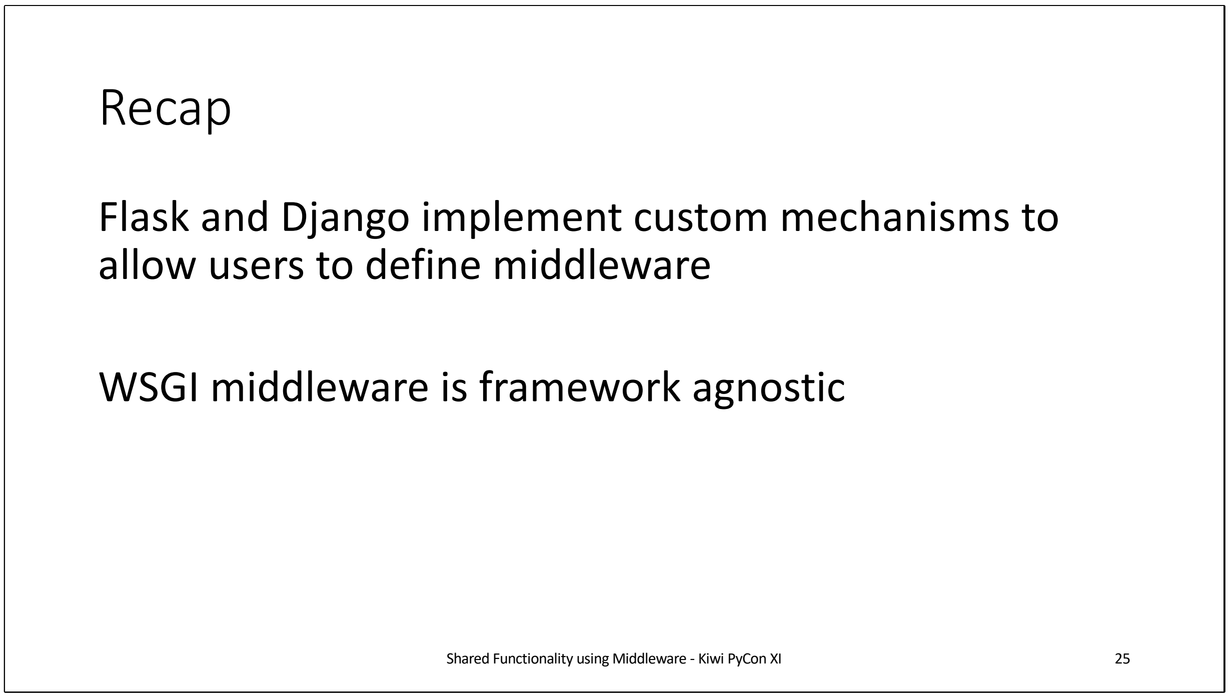
```
def wsgi_app(self, environ, start_response) -> t.Any:
```



Django + WSGI Middleware

```python
# wsgi.py

from opentelemetry.instrumentation.wsgi \
    import OpenTelemetryMiddleware

application = get_wsgi_application()

application = OpenTelemetryMiddleware(application)
```

```python
def wsgi_app(self, environ, start_response) -> t.Any:
```

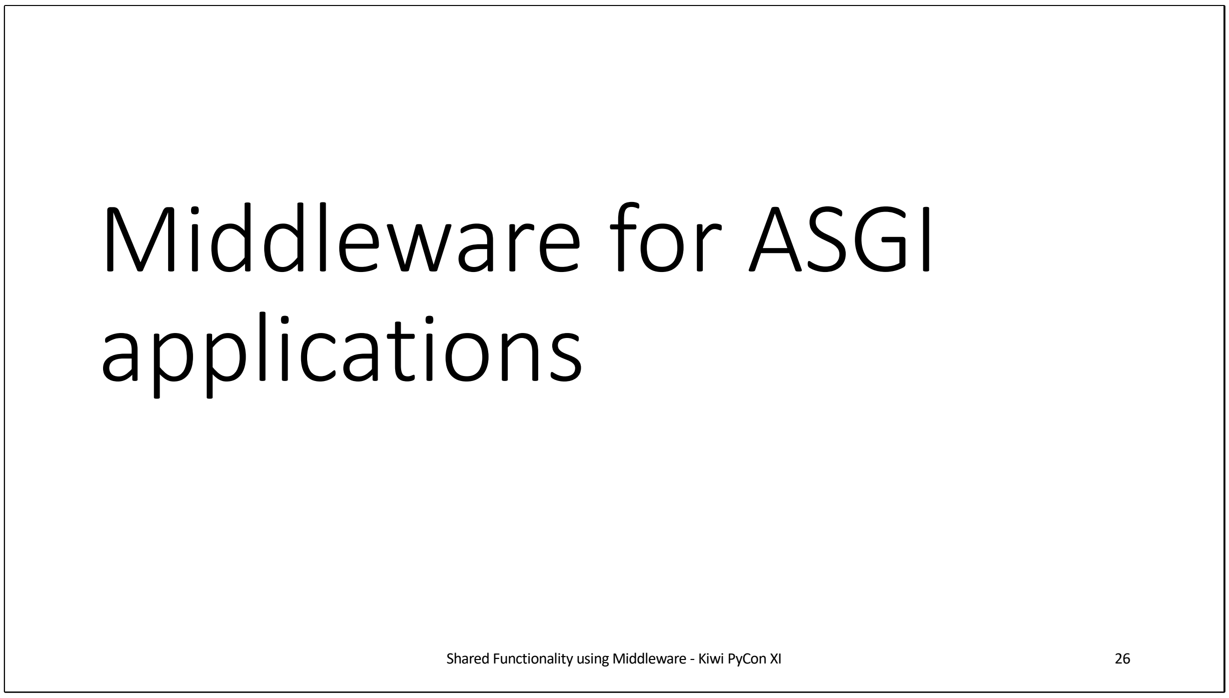


Recap

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

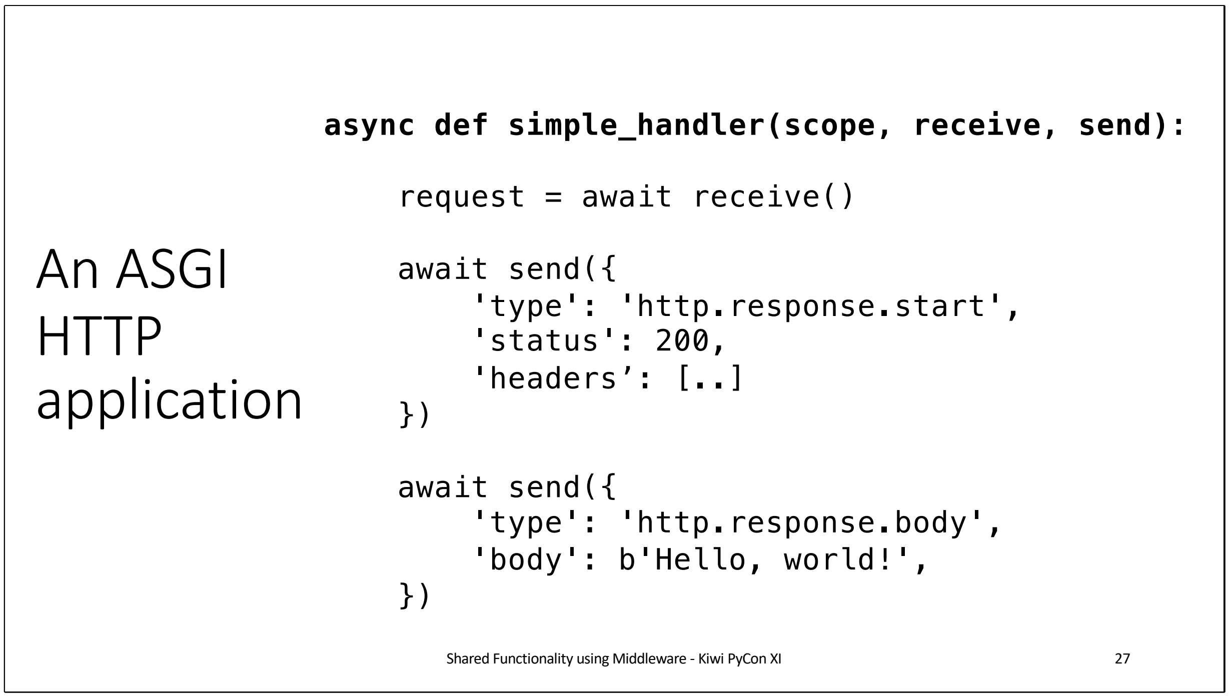
WSGI middleware is framework agnostic

WSGI frameworks such as Django and Flask define custom mechanisms for defining middleware. You can also however write a middleware as an WSGI application and then wrap your WSGI application inside it. That middleware you wrote is framework independent. However, I haven’t fully explored if there are potential problems or limitations with that approach.



Middleware for ASGI applications

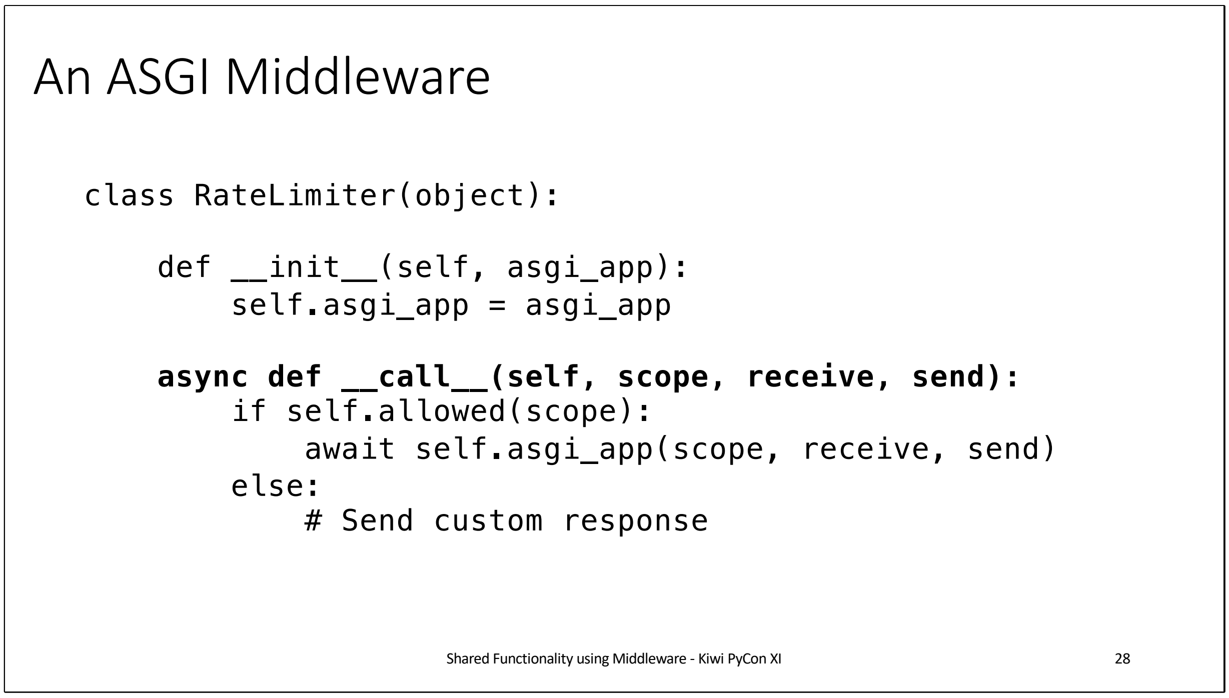
Next, we are going to look at middleware for ASGI applications.



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!',
    })
```

Once again, remember this is the foundation for FastAPI or Starlette or any ASGI frameworks, so I want you to keep the function signature in mind here.



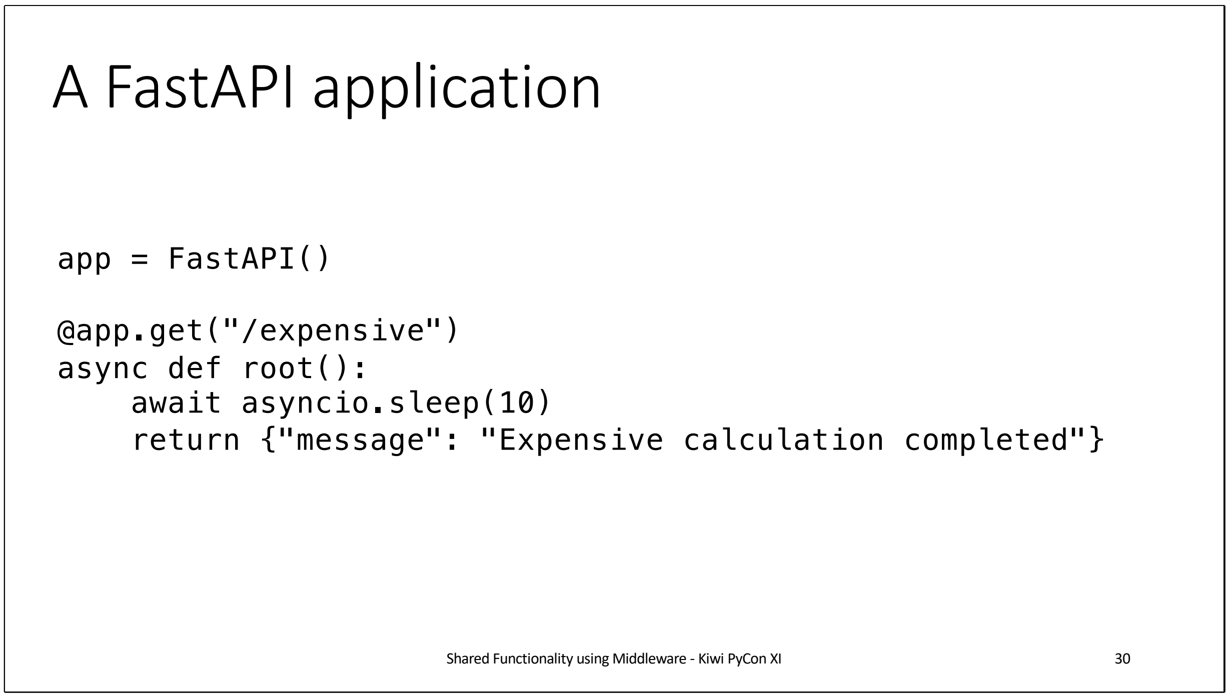
An ASGI Middleware

```
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

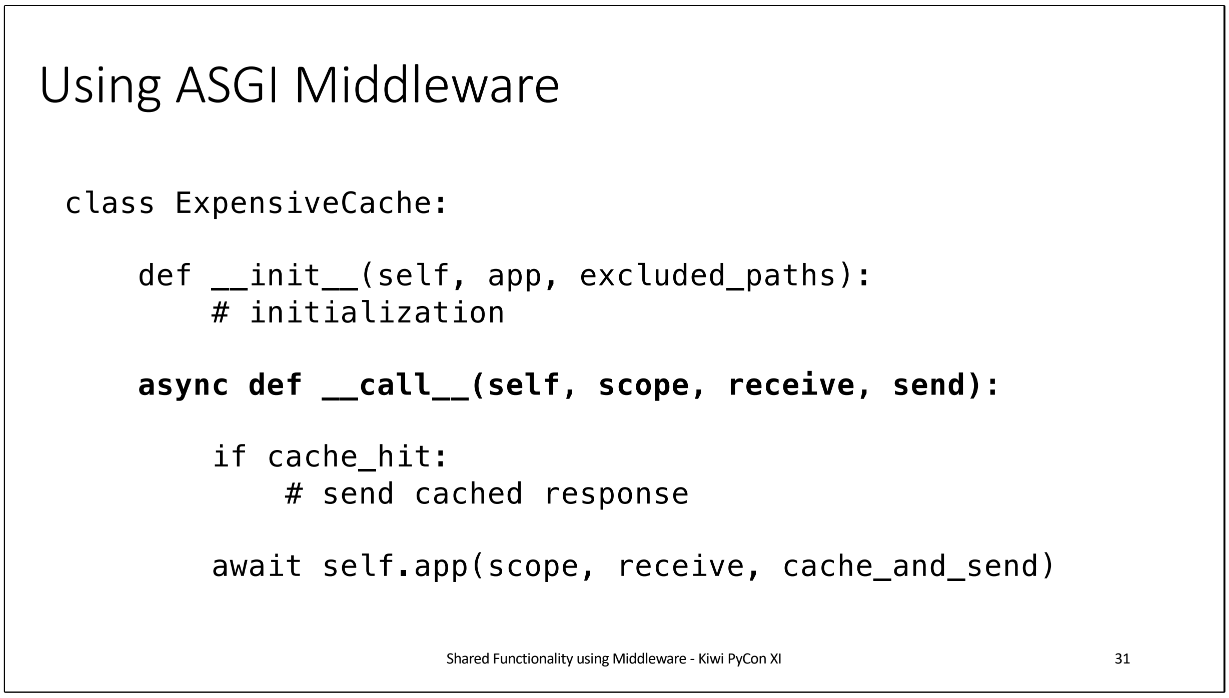
```
app = RateLimiter(simple_handler)
```

```
$ 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
```



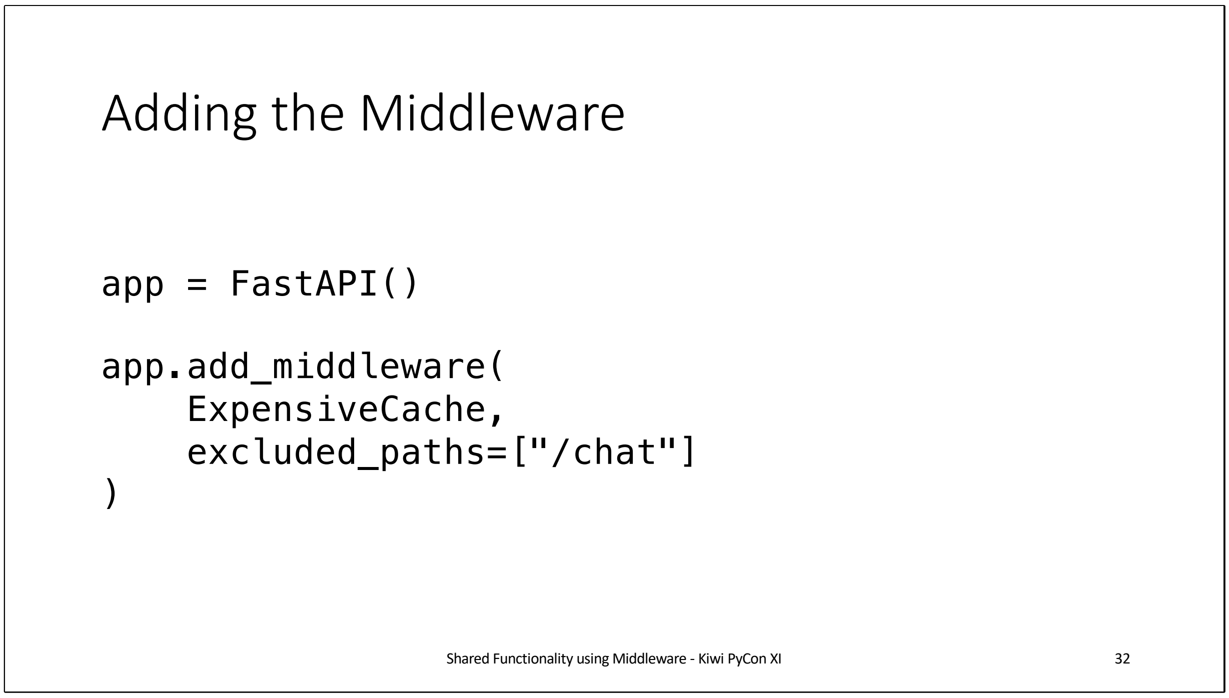
A FastAPI application

```
app = FastAPI()

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



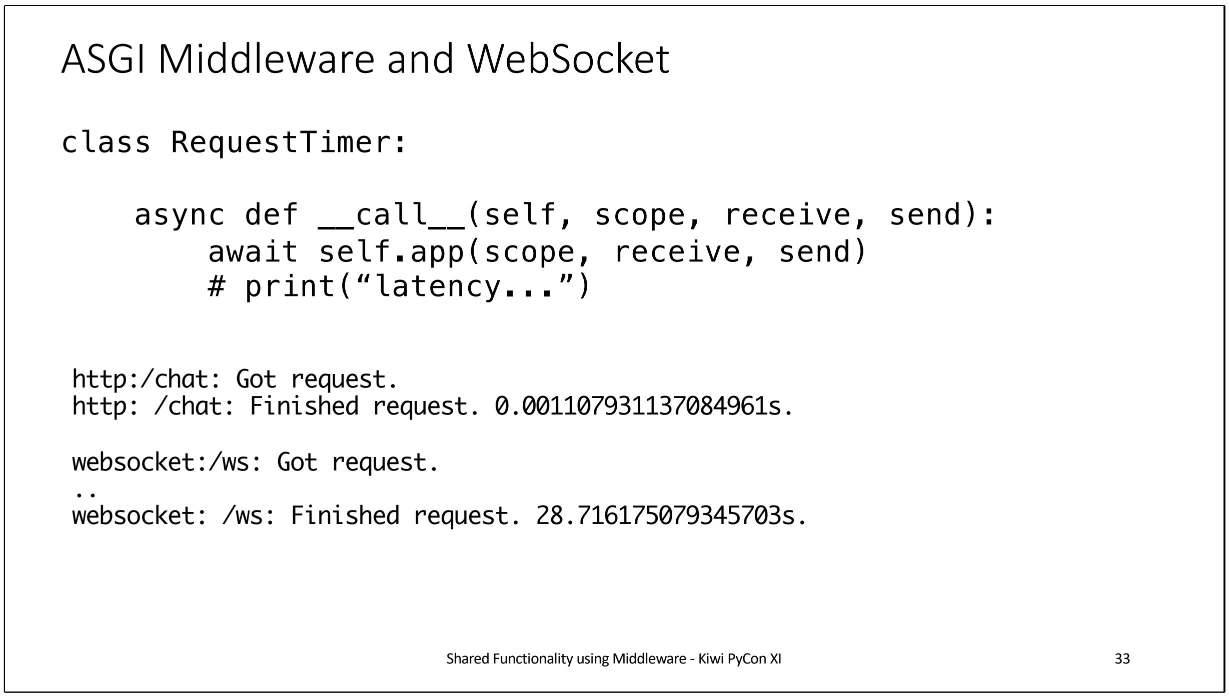
Using ASGI Middleware

```
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"]
)
```

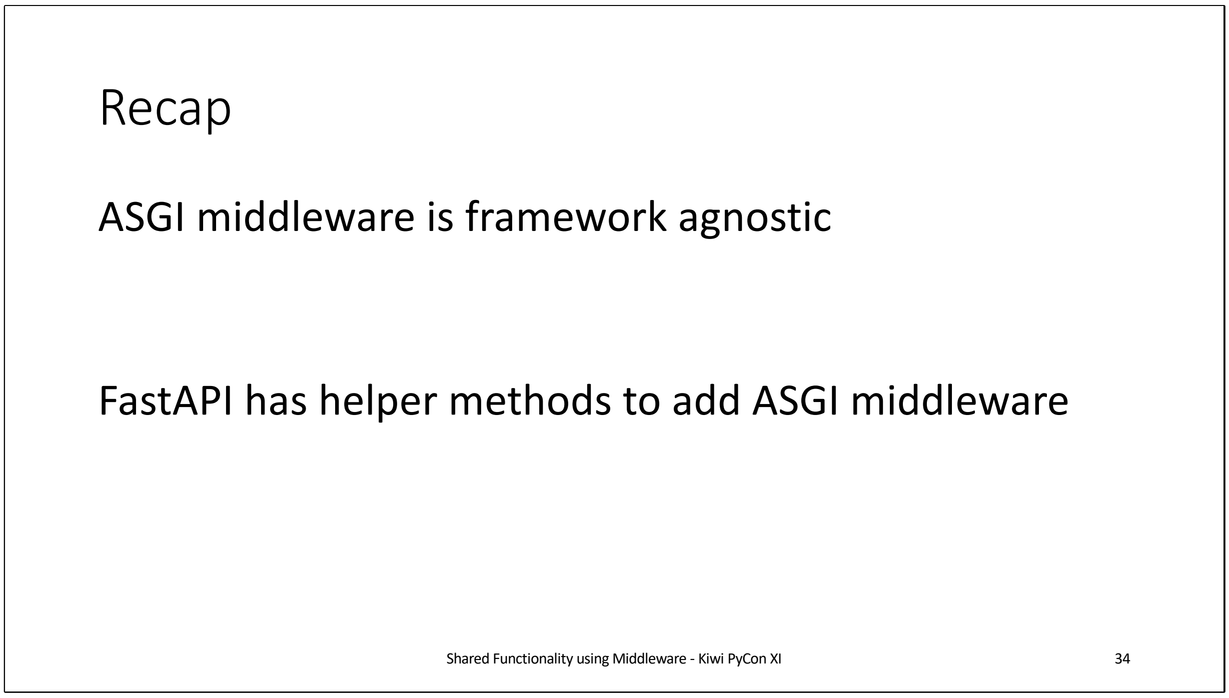

ASGI Middleware and WebSocket

```
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. 0.0011079311370849 61s.

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

Shared Functionality using Middleware - Kiwi PyCon XI   33

Scope lasts for the lifetime of the connection

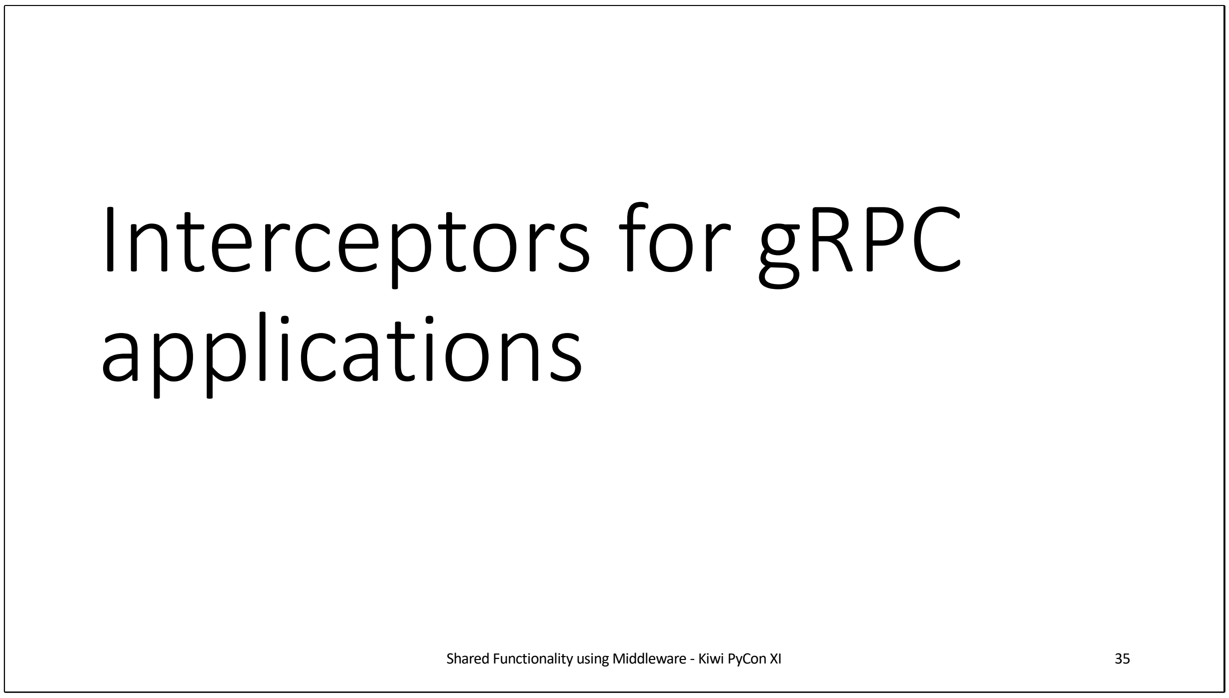


Recap

ASGI middleware is framework agnostic

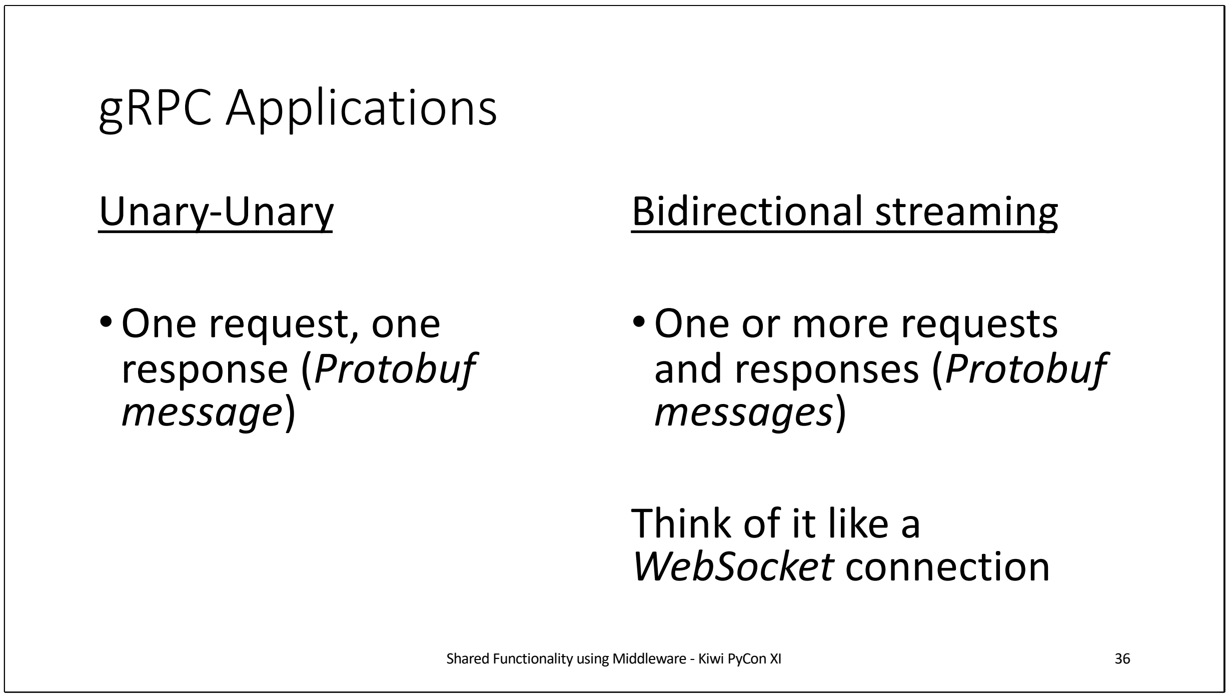
FastAPI has helper methods to add ASGI middleware

ASGI middleware is an ASGI application, ASGI middleware is framework independent. An ASGI middleware supports HTTP and WebSockets.



Interceptors for gRPC applications

gRPC community uses the term “intereptors” to refer to what is traditionally referred to as middleware in the web community.



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

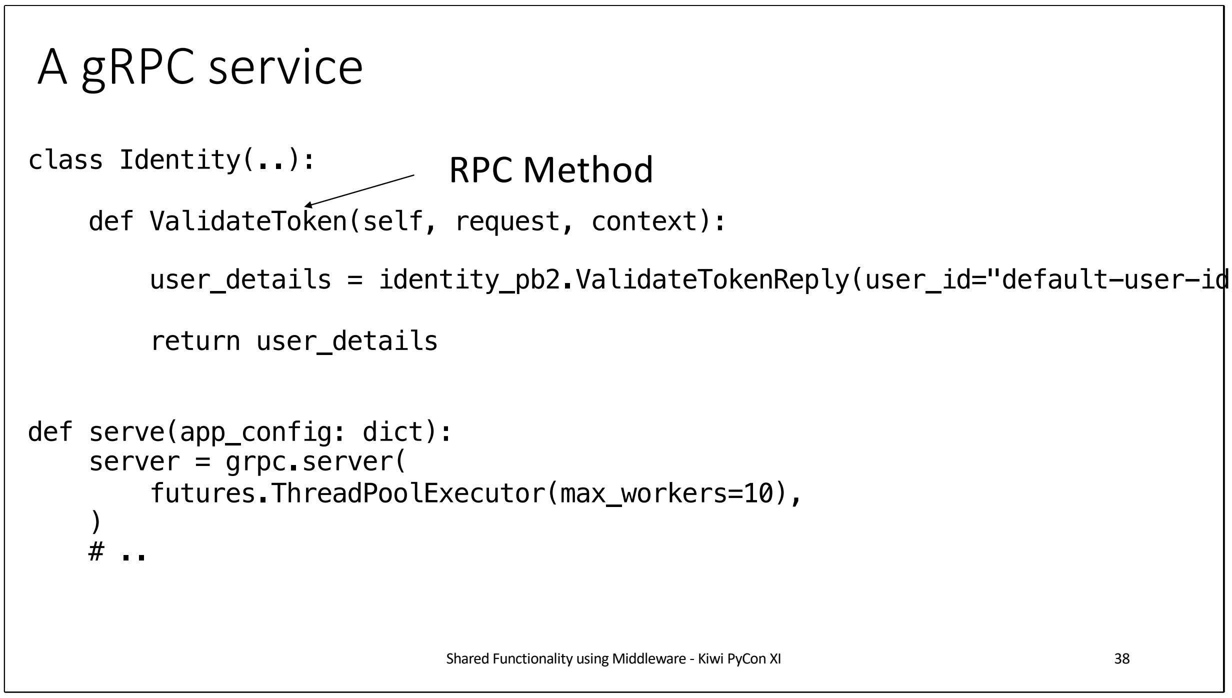
We are going to look at two patterns of implementation of gRPC applications today in the context of implementing interceptors. Unary-Unary, where the client sends one request – a protobuf message and waits for a reply from the server.

The second category of applications we will look at implements the bidirectional streaming pattern. A client can send one or more requests and the server can send one or more replies, like a websocket connection from the HTTP world.

Unary-Unary gRPC Applications

Shared Functionality using Middleware - Kiwi PyCon XI 37

First, let’s look at a unary-unary gRPC application pattern and how we can implement intereceptors – both server and client side.

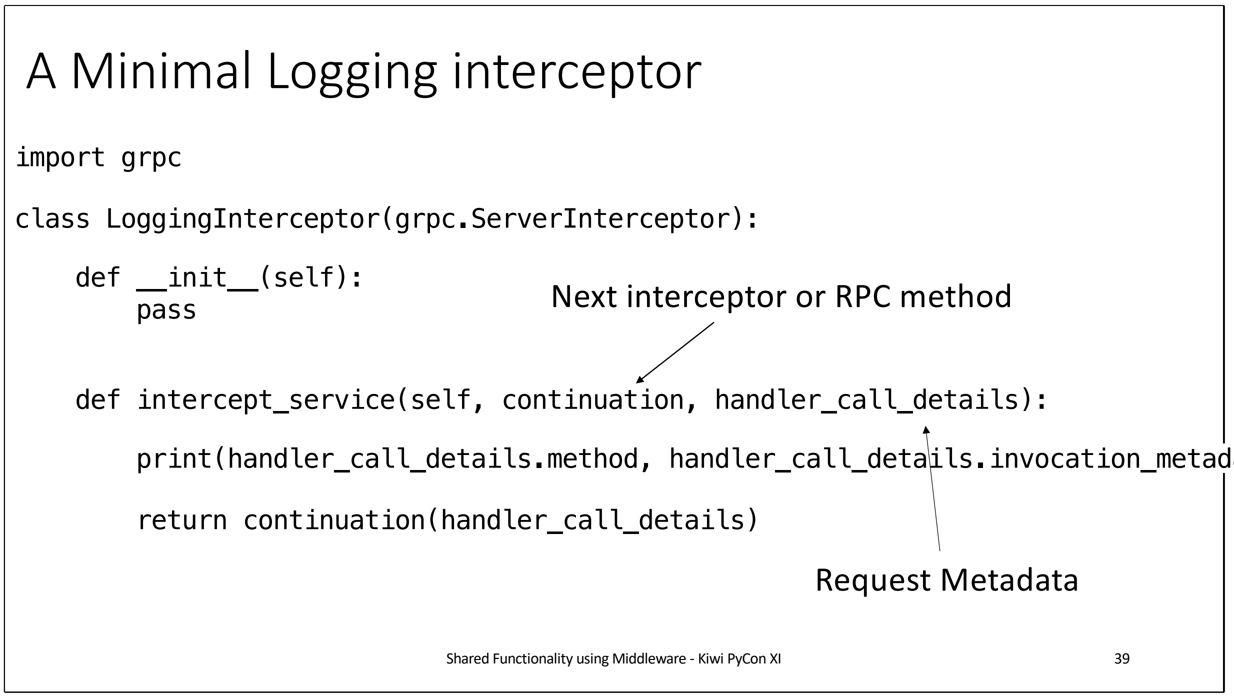


A gRPC service

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),
    )
    # ..
```

Shared Functionality using Middleware - Kiwi PyCon XI 38

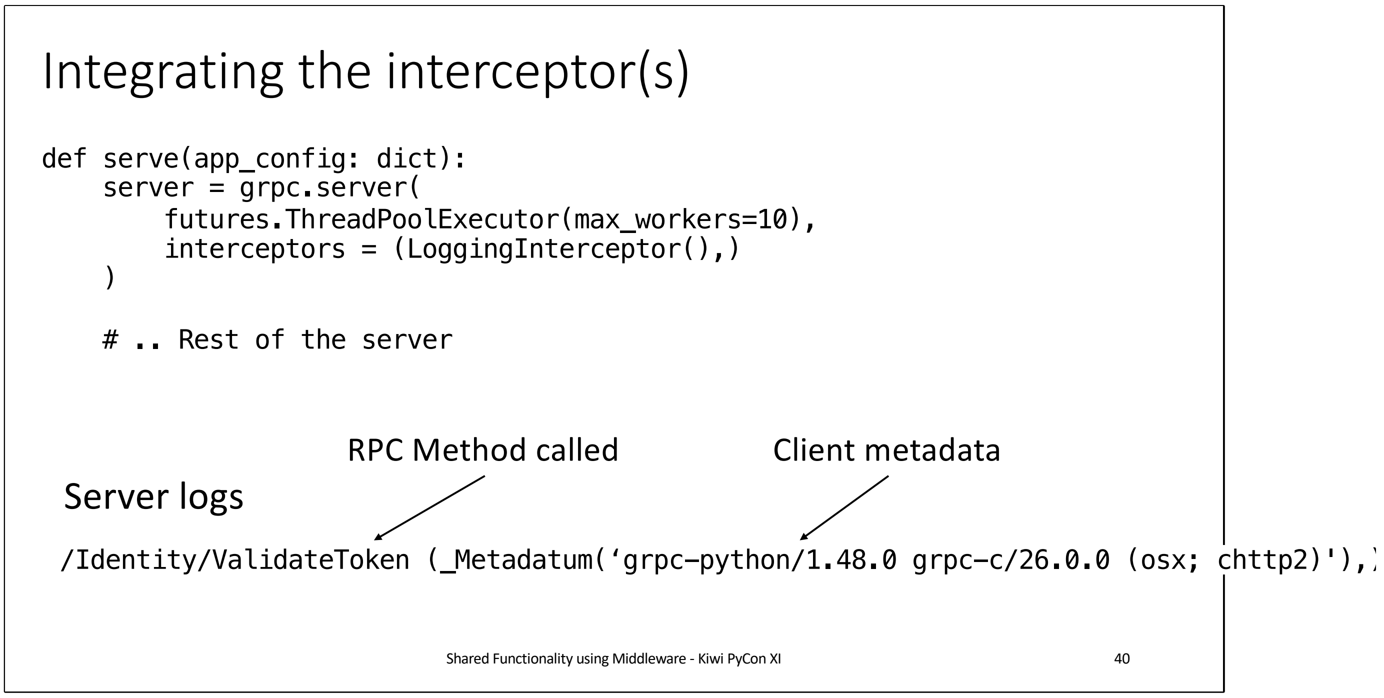


A Minimal Logging interceptor

```
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

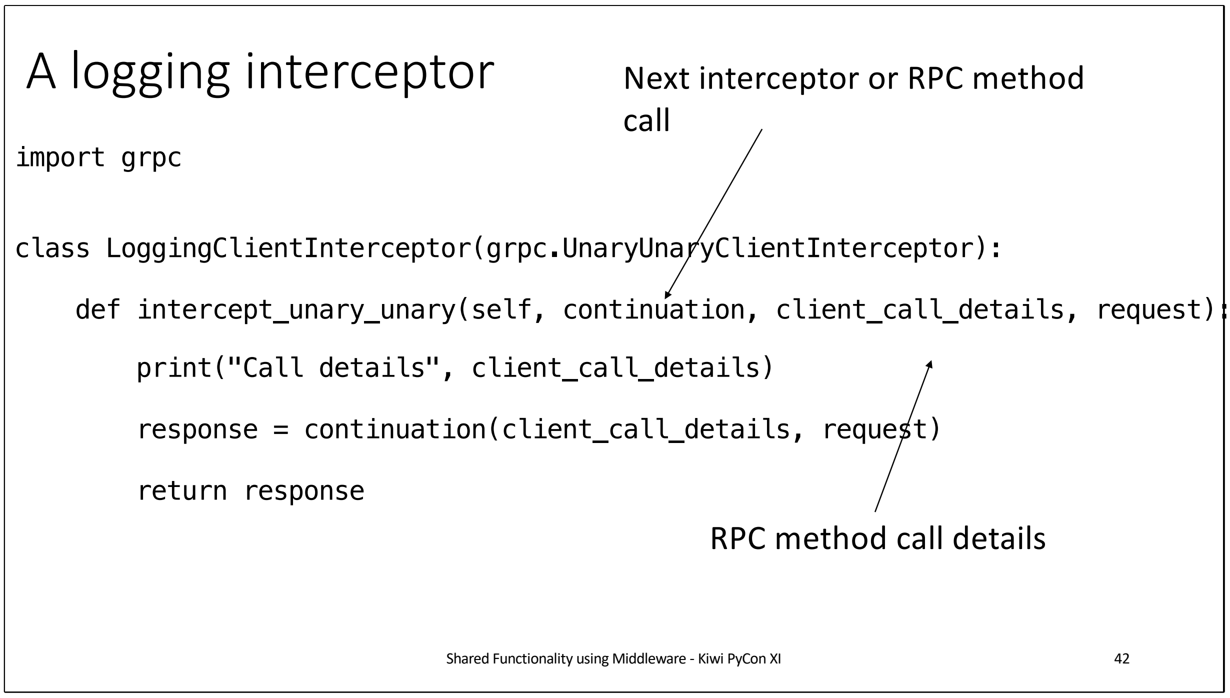
```
/Identity/ValidateToken (_Metadatum(‘grpc-python/1.48.0 grpc-c/26.0.0 (osx; chttp2)'),
```

This is how we integrate the interceptor – we add a keyword argument, interceptors which is a tuple of the interceptors we want to add.

Then a call to the server will be logged..

A gRPC Client

```
# imports

def run():
    channel_credentials = grpc.ssl_channel_credentials(server_cert)

    with grpc.secure_channel('localhost:50051', channel_credentials) as channel:
        stub = identity_pb2_grpc.IdentityStub(channel)

        response = stub.ValidateToken(
            identity_pb2.ValidateTokenRequest(token="a-token")
        )

    # rest of the client
```



A logging interceptor

Next interceptor or RPC method call

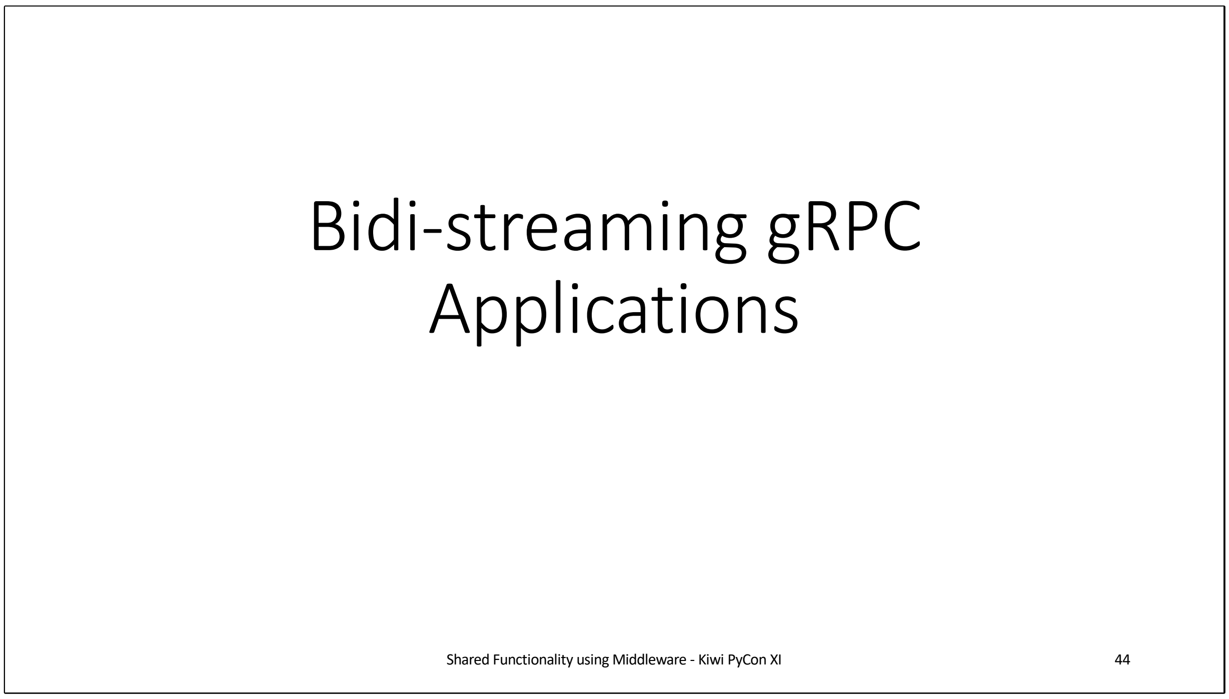
```
import grpc

class LoggingClientInterceptor(grpc.UnaryUnaryClientInterceptor):
    def intercept_unary_unary(self, continuation, client_call_details, request):
        print("Call details", client_call_details)
        response = continuation(client_call_details, request)
        return response
```

RPC method call details

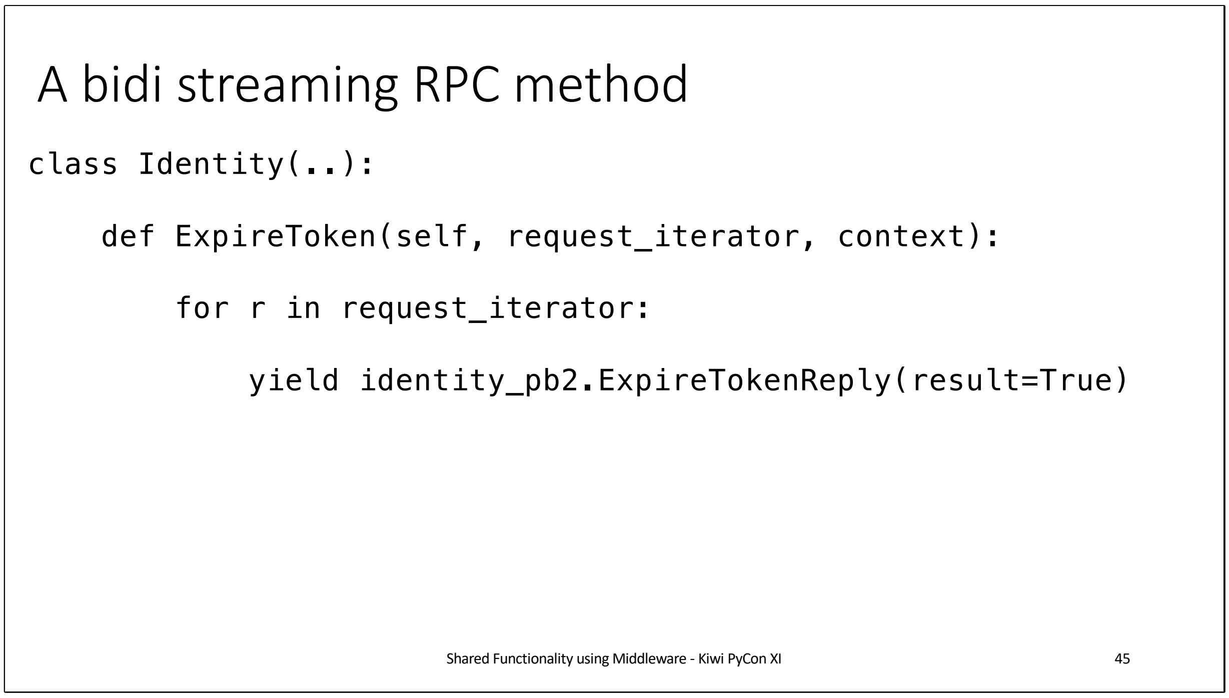
Shared Functionality using Middleware - Kiwi PyCon XI 42

Integrating the interceptor(s)

```
def run():
    with grpc.secure_channel('localhost:50051', channel_credentials) as
channel:
        intercepted_channel = grpc.intercept_channel(
                channel,
                LoggingClientInterceptor()
        )
        stub = identity_pb2_grpc.IdentityStub(intercepted_channel)

        # .. Rest of client
```



Bidi-streaming gRPC Applications

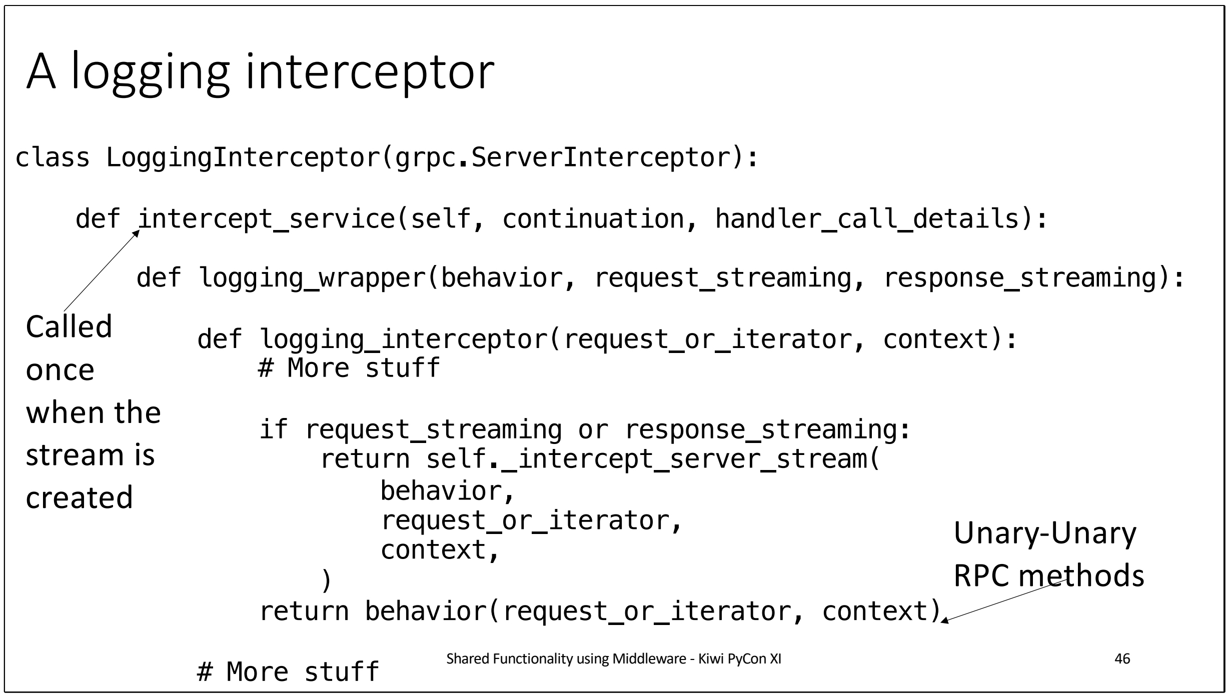


A bidi streaming RPC method

```
class Identity(..):

    def ExpireToken(self, request_iterator, context):

        for r in request_iterator:

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

Shared Functionality using Middleware - Kiwi PyCon XI 45

Request iterator, response iterator

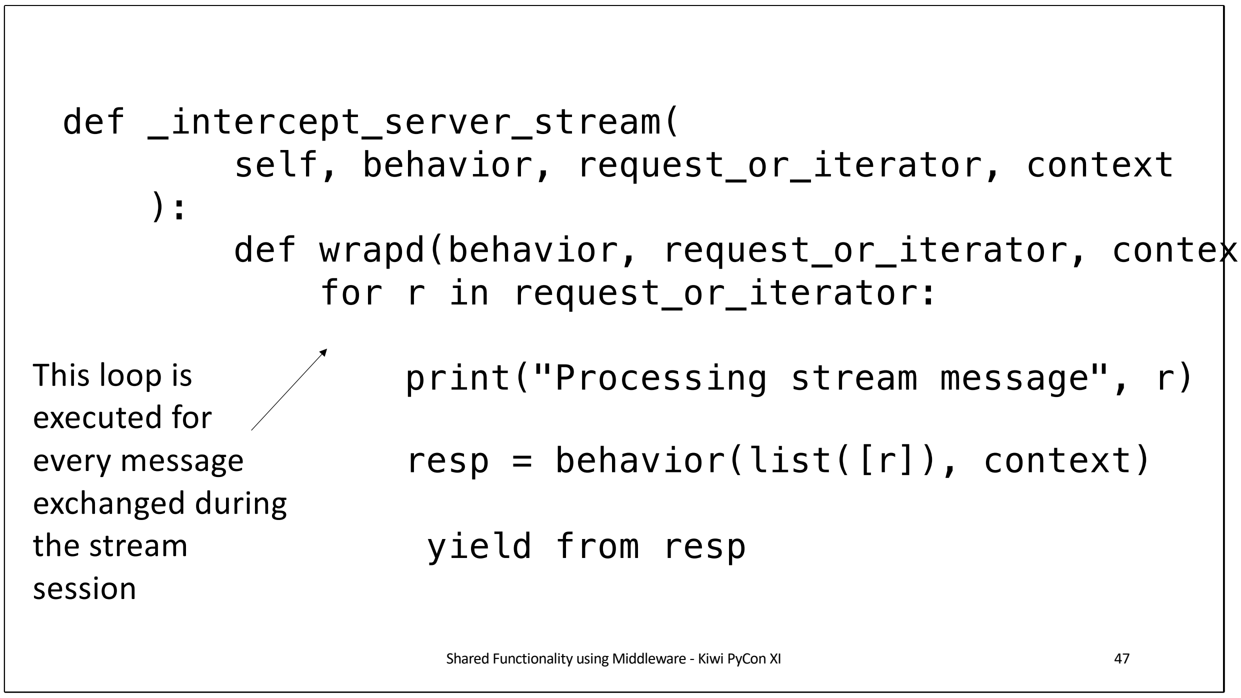


A logging interceptor

```
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

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, request):

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

Logging client-side interceptor

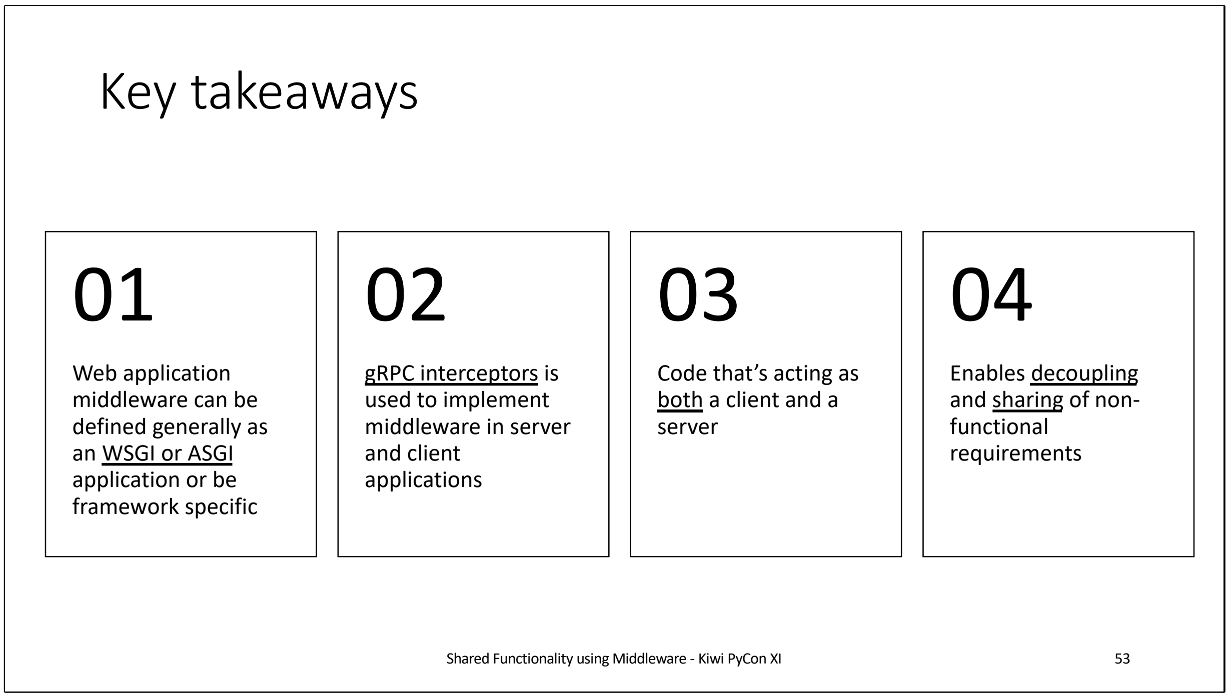
```
def intercept_stream_stream(
    self,continuation, client_call_details, request_iterator
):

    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_duration))
```

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

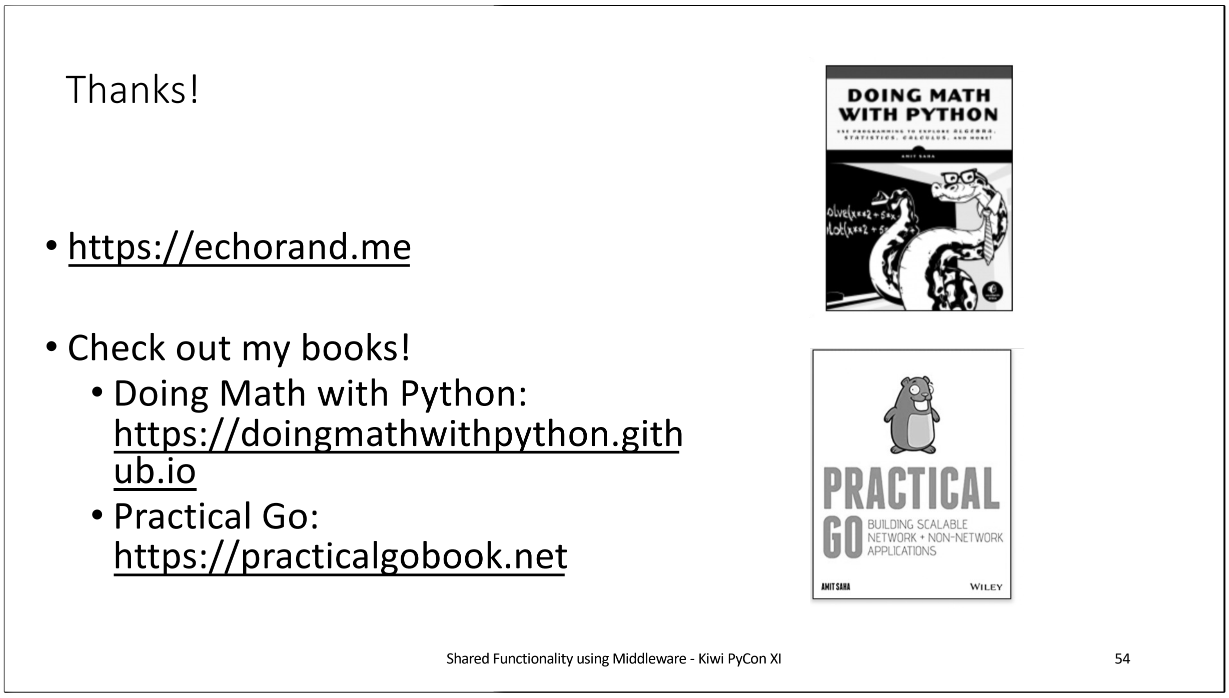
Code that's acting as both a client and a server

04

Enables decoupling and sharing of non-functional requirements

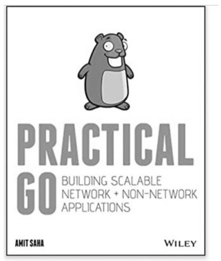
Key takeaways
Client for another middleware or your application
Server for an external request or another middleware

Migration between application frameworks/versions WSGI or ASGI



Thanks!

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



Shared Functionality using Middleware - Kiwi PyCon XI 54

I hope you have gained insights into the internals of middleware as they apply to WSGI, ASGI and gRPC applications and go away with a few new neural pathways formed in your brain. Thank you for coming to my talk. I hope you will check out my books and enjoy the rest of your conference!