Here is an issue I have. I am using Connexion v3 with opentracing/opentelemetry libs in python. I am building a backend which also uses other libs like celery, sqlalchemy, … so a request could traverse all of them. Underlining, we use Flask as web framework for connexion. We were using connexion v2 and used to store request context info like user account_id in Flask.g and access them anywhere in all the libs to do end to end tracing. Now with v3, this can't be done because Flask.g cannot be accessed from a connexion v3 context. Please help design that solution end to end.

Designing End-to-End Tracing with Connexion v3, OpenTelemetry, and Flask

Problem Analysis

You're migrating from Connexion v2 to v3 and losing access to Flask's `g` object for storing request context across your stack (Celery, SQLAlchemy, etc.). Here's a comprehensive solution using OpenTelemetry context propagation.

Recommended Solution

1. Use OpenTelemetry Context for Propagation

Instead of Flask.g, leverage OpenTelemetry's context propagation which works across asynchronous boundaries and process boundaries (important for Celery).

python

```
from opentelemetry import trace
from opentelemetry.context import Context, attach,
get_current
```

```python
def store_in_context(key, value):
    # Create a new context with the key-value pair
    new_context = get_current().set_value(key, value)
    attach(new_context)

def get_from_context(key):
    return get_current().get(key, None)
```

2. Middleware for Context Initialization

Create a Connexion middleware to initialize the context:

```python
from connexion.middleware import MiddlewarePosition
from connexion import AsyncApp

app = AsyncApp(__name__)

class ContextMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        # Initialize context here
        ctx = Context()
        attach(ctx)
        
        async def modified_send(event):
            if event['type'] == 'http.response.start':
                # Cleanup if needed
                pass
            await send(event)

        await self.app(scope, receive, modified_send)

app.add_middleware(ContextMiddleware, 
position=MiddlewarePosition.BEFORE_EXCEPTION)
```

3. Custom Accessor Functions

Create utility functions to access context data:

```python
def get_account_id():
    return get_from_context('account_id')
```

```python
def set_account_id(account_id):
    store_in_context('account_id', account_id)
```

4. Integrating with Celery

For Celery tasks, you'll need to propagate the context:

python

```python
from opentelemetry.propagate import inject, extract

@app.task
def background_task(headers, *args, **kwargs):
    # Extract context from headers
    context = extract(headers)
    attach(context)
    
    # Now you can access the context
    account_id = get_account_id()
    # ... rest of task
```

When calling the task:

python

```python
headers = {}
inject(headers)  # Inject current context into headers
background_task.delay(headers, *args, **kwargs)
```

5. SQLAlchemy Integration

For SQL queries, add context information to SQL comments:

python

```python
from sqlalchemy.event import listen
from sqlalchemy.engine import Engine

def before_cursor_execute(conn, cursor, statement, 
parameters, context, executemany):
    account_id = get_account_id()
    if account_id:
        statement = f"-- account_id: {account_id}
\n{statement}"

listen(Engine, 'before_cursor_execute', 
before_cursor_execute)
```

6. OpenTelemetry Instrumentation

Configure OpenTelemetry to include your custom context:

python

```python
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter

provider = TracerProvider()
processor = BatchSpanProcessor(OTLPSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

# Add custom attributes to spans
def add_custom_attributes(span, account_id):
    if account_id:
        span.set_attribute("account.id", account_id)
```

7. Connexion v3 Integration

In your operation handlers, set the context:

python

```python
async def some_operation(account_id):
    set_account_id(account_id)
    # ... rest of handler
```

8. Error Handling

Ensure context is properly cleaned up:

python

```python
from contextlib import asynccontextmanager

@asynccontextmanager
async def request_context(account_id):
    try:
        set_account_id(account_id)
        yield
    finally:
        # Clear context if needed
        pass
```

Migration Considerations

1. **Gradual Migration:** Implement the new system alongside the old one during transition
2. **Testing:** Verify context propagation works across all components
3. **Documentation:** Document the new context access patterns for your team

Benefits

1. **Standardized:** Uses OpenTelemetry's standard context propagation
2. **Distributed:** Works across service boundaries
3. **Future-proof:** Not tied to Flask-specific implementations
4. **Observable:** Integrates with your existing tracing infrastructure

This approach gives you similar functionality to Flask.g but with better support for distributed systems and async operations.

The solution also use python opentracing lib and I need to keep using it

Hybrid Tracing Solution with OpenTracing and OpenTelemetry in Connexion v3

Since you need to maintain compatibility with the OpenTracing library while migrating to Connexion v3, here's a solution that bridges both worlds:

1. Context Propagation Bridge

Create a bridge between OpenTracing and OpenTelemetry contexts:

python

```python
from opentracing import tracer as opentracing_tracer
from opentelemetry import trace
from opentelemetry.context import Context, attach, get_current
from opentelemetry.trace.propagation.tracecontext import TraceContextTextMapPropagator

class TracingBridge:
    @staticmethod
    def get_current_context():
        """Get current context from both systems"""
        otel_ctx = get_current()
        
        # Try to get OpenTracing active span
        ot_span = opentracing_tracer.active_span
        if ot_span:
            # Convert OT span to OTEL context if needed
            carrier = {}
            opentracing_tracer.inject(ot_span.context, opentracing.Format.TEXT_MAP, carrier)
            otel_ctx = TraceContextTextMapPropagator().extract(carrier)
        
        return otel_ctx

    @staticmethod
    def set_account_id(account_id):
        """Store account ID in both systems"""
        # Store in OTEL context
        ctx = get_current().set_value('account_id', account_id)
        attach(ctx)
        
        # Also store in OpenTracing active span
        span = opentracing_tracer.active_span
        if span:
```

```
            span.set_tag('account.id', account_id)

    @staticmethod
    def get_account_id():
        """Retrieve account ID from either system"""
        # First try OTEL context
        account_id = get_current().get('account_id', None)
        if account_id:
            return account_id
            
        # Fall back to OpenTracing span
        span = opentracing_tracer.active_span
        if span:
            return span.get_tag('account.id')
        
        return None
```

2. Connexion Middleware Integration

Create middleware that works with both tracing systems:

python

```python
from connexion.middleware import MiddlewarePosition
from connexion import AsyncApp
import opentracing
from opentracing.propagation import Format

app = AsyncApp(__name__)

class HybridTracingMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        # Extract context from headers for both systems
        headers = {k.decode(): v.decode() for k, v in scope.get('headers', [])}
        
        # OpenTracing extraction
        ot_span_ctx = opentracing_tracer.extract(
            Format.HTTP_HEADERS,
            headers
        )
        
        # OpenTelemetry extraction
        otel_ctx = 
```

```python
TraceContextTextMapPropagator().extract(headers)
        attach(otel_ctx)
        
        # Start span in both systems
        with opentracing_tracer.start_active_span(
            scope['path'],
            child_of=ot_span_ctx
        ) as scope_ot, trace.use_span(
            trace.get_current_span(),
            otel_ctx
        ):
            # Store both contexts
            TracingBridge.set_account_id(headers.get('x-
account-id'))
            
            async def modified_send(event):
                if event['type'] == 'http.response.start':
                    # Cleanup if needed
                    pass
                await send(event)

            await self.app(scope, receive, modified_send)

app.add_middleware(HybridTracingMiddleware, 
position=MiddlewarePosition.BEFORE_EXCEPTION)
```

3. Celery Integration with Both Systems

python

```python
from celery import Celery
from opentracing.propagation import Format
import opentracing

app = Celery()

@app.task
def background_task(headers, *args, **kwargs):
    # OpenTracing extraction
    ot_span_ctx = opentracing_tracer.extract(
        Format.TEXT_MAP,
        headers
    )
    
    # OpenTelemetry extraction
    otel_ctx = 
TraceContextTextMapPropagator().extract(headers)
```

```
    attach(otel_ctx)
    
    with opentracing_tracer.start_active_span(
        'background_task',
        child_of=ot_span_ctx
    ), trace.use_span(
        trace.get_current_span(),
        otel_ctx
    ):
        account_id = TracingBridge.get_account_id()
        # ... rest of task
```

4. SQLAlchemy Instrumentation for Both

python

```
from sqlalchemy.event import listen
from sqlalchemy.engine import Engine

def before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    account_id = TracingBridge.get_account_id()
    if account_id:
        # Add to SQL comment
        statement = f"-- account_id: {account_id}\n{statement}"
        
        # Add as span tag in both systems
        ot_span = opentracing_tracer.active_span
        if ot_span:
            ot_span.set_tag('sql.account_id', account_id)
            
        otel_span = trace.get_current_span()
        if otel_span:
            otel_span.set_attribute('sql.account_id', account_id)

listen(Engine, 'before_cursor_execute', before_cursor_execute)
```

5. Unified Access Pattern

Replace all Flask.g accesses with:

python

```
# Instead of from flask import g
from your_tracing_bridge import TracingBridge
```

```
# Set values
TracingBridge.set_account_id(account_id)

# Get values
account_id = TracingBridge.get_account_id()
```

6. Migration Strategy

1. **Phase 1:** Implement the bridge alongside existing Flask.g usage
2. **Phase 2:** Gradually replace Flask.g references with TracingBridge
3. **Phase 3:** Remove Flask.g dependencies once all components are migrated
4. **Phase 4:** (Optional) Fully migrate to OpenTelemetry if desired

Benefits of This Approach

1. **Backward Compatibility:** Maintains existing OpenTracing instrumentation
2. **Future-Readiness:** Prepares for OpenTelemetry migration
3. **Consistent Context:** Single API for accessing tracing context
4. **Distributed Tracing:** Works across process boundaries (Celery)
5. **Minimal Disruption:** Gradual migration path

This solution gives you a clean migration path while maintaining all existing OpenTracing functionality during your transition to Connexion v3.

Where does this comes from opentracing_tracer

You're absolutely right to question where `opentracing_tracer` comes from - I should have made that explicit. In the OpenTracing Python implementation, there's a global tracer instance that you typically configure early in your application setup.

Clarifying the `opentracing_tracer` Reference

The `opentracing_tracer` in my examples refers to the global tracer instance from the OpenTracing package. Here's how it's properly initialized:

python

```python
from opentracing import tracer as opentracing_tracer
from opentracing import set_global_tracer

# Typically you would initialize it during app startup:
def init_tracing():
    # Example with Jaeger client:
    from jaeger_client import Config
    
    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name='your-service'
    )
    tracer = config.initialize_tracer()
    set_global_tracer(tracer)
```

Complete Initialization Example

Here's how you would properly set up both OpenTracing and OpenTelemetry:

python

```python
# tracing_setup.py
from opentracing import set_global_tracer, tracer as opentracing_tracer
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from jaeger_client import Config  # or your OpenTracing tracer of choice

def configure_tracing():
    # 1. Configure OpenTracing (legacy)
    config = Config(
        config={
            'sampler': {'type': 'const', 'param': 1},
            'logging': True,
        },
        service_name='your-service'
    )
    ot_tracer = config.initialize_tracer()
    set_global_tracer(ot_tracer)
    
    # 2. Configure OpenTelemetry (modern)
    provider = TracerProvider()
    processor = BatchSpanProcessor(OTLPSpanExporter())
    provider.add_span_processor(processor)
    trace.set_tracer_provider(provider)
    
    return ot_tracer

# Then in your application startup:
from tracing_setup import configure_tracing
opentracing_tracer = configure_tracing()  # This is what you'd use in the examples
```

Why This Works

1. `opentracing_tracer` is the global tracer instance after you call `set_global_tracer()`
2. The `from opentracing import tracer` import gives you access to this global instance
3. This pattern matches how OpenTracing is typically used

in Python applications

Important Notes

1. **Initialization Order:** Configure your tracer before any spans are created
2. **Thread Safety:** The global tracer is thread-safe
3. **Dependencies:** You'll need the `jaeger-client` package (or your preferred OpenTracing tracer implementation) installed
4. **Compatibility:** This works alongside OpenTelemetry as shown in the hybrid solution

Where You'd Use This

In the previous examples, you would:

1. Add the initialization code to your application startup
2. Then all the middleware and other components can safely use `opentracing_tracer`
3. The imports would look like:

python

```python
from opentracing import tracer as opentracing_tracer
from opentelemetry import trace
```

This maintains clean access to both tracing systems while keeping the initialization centralized.

so the following impacts the global tracer ?   provider = TracerProvider()
    processor = BatchSpanProcessor(OTLPSpanExporter())
    provider.add_span_processor(processor)
    trace.set_tracer_provider(provider)

Yes, calling `trace.set_tracer_provider(provider)` *does* impact the global tracer in OpenTelemetry, but it's important to clarify how this interacts with OpenTracing's global tracer since you're using both systems.

OpenTelemetry vs. OpenTracing Global Tracers

1. OpenTelemetry's Global Tracer

When you do:

python

```python
from opentelemetry import trace
provider = TracerProvider()
trace.set_tracer_provider(provider)
```

This:

- Establishes OpenTelemetry's global tracer provider
- Affects all `trace.get_tracer()` calls
- Is completely separate from OpenTracing's global tracer

2. OpenTracing's Global Tracer

When you do:

python

```python
from opentracing import set_global_tracer
set_global_tracer(your_tracer)
```

This:

- Sets OpenTracing's global tracer
- Affects all `opentracing.tracer` references
- Is completely separate from OpenTelemetry's tracer

Key Impacts in Your Hybrid System

1. **No Direct Conflict:**
    - These are parallel global states that don't interfere with each other

- OpenTelemetry configuration won't affect OpenTracing's tracer and vice versa

2. **Automatic Context Propagation:**
   - The OpenTelemetry tracer will automatically handle context propagation
   - The OpenTracing tracer will use its own propagation mechanism
   - Our bridge code connects these two propagation systems
3. **Performance Considerations:**
   - Both tracers will be active simultaneously
   - Each will process spans independently
   - You might see "double spans" unless you implement filtering

Recommended Configuration Pattern

python

```python
def configure_tracing(enable_opentracing=True, enable_otel=True):
    tracers = {}
    
    if enable_opentracing:
        # OpenTracing config
        from jaeger_client import Config
        ot_config = Config(config={...}, service_name='your-service')
        ot_tracer = ot_config.initialize_tracer()
        set_global_tracer(ot_tracer)
        tracers['opentracing'] = ot_tracer
    
    if enable_otel:
        # OpenTelemetry config
        from opentelemetry.sdk.trace import TracerProvider
        from opentelemetry.sdk.trace.export import BatchSpanProcessor
        provider = TracerProvider()
        processor = BatchSpanProcessor(OTLPSpanExporter())
        provider.add_span_processor(processor)
        trace.set_tracer_provider(provider)
        tracers['opentelemetry'] = provider
    
    return tracers
```

Important Considerations

1. **Span Duplication**:
   - You'll likely generate two spans for each operation (one per system)
   - Solution: Use the bridge to make them share context but only record in one system
2. **Resource Usage**:
   - Running both systems increases overhead
   - Consider this temporary during migration
3. **Sampling Coordination**:
   - Ensure both systems use compatible sampling decisions
   - Example config:

python

```python
# Coordinated sampling
sampling_rate = 0.5  # 50% of requests
ot_config = Config(config={'sampler': {'type': 'probabilistic', 'param': sampling_rate}})
provider = TracerProvider(sampler=ParentBased(ProbabilisticSampler(sampling_rate)))
```

4. **Shutdown Handling**:
   - Remember to properly shutdown both tracers
5. python

```python
def shutdown_tracing(tracers):
    if 'opentracing' in tracers:
        tracers['opentracing'].close()  # Jaeger-specific
    if 'opentelemetry' in tracers:
        tracers['opentelemetry'].shutdown()
```

This approach gives you full control over both tracing systems while maintaining their independence. The global configuration of one doesn't affect the other, but our bridge code allows them to work together when needed.

We won't be using both at the same time.

Clean OpenTracing Solution for Connexion v3 Without OpenTelemetry

Since you'll only be using OpenTracing (not both systems simultaneously), here's a streamlined solution that maintains your existing OpenTracing instrumentation while solving the Connexion v3 context propagation challenge.

Core Solution Components

1. Context Management Middleware

python

```python
from connexion import AsyncApp
from connexion.middleware import MiddlewarePosition
import opentracing
from opentracing.propagation import Format
from opentracing import tracer as opentracing_tracer

app = AsyncApp(__name__)

class OpenTracingContextMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        headers = {k.decode(): v.decode() for k, v in
```

```
scope.get('headers', [])}

        # Extract OpenTracing context
        span_ctx = 
opentracing_tracer.extract(Format.HTTP_HEADERS, headers)

        # Start span
        with opentracing_tracer.start_active_span(
            scope['path'],
            child_of=span_ctx
        ) as scope:
            # Store request context in the span
            account_id = headers.get('x-account-id')
            if account_id:
                scope.span.set_tag('account.id', 
account_id)

            async def modified_send(event):
                if event['type'] == 'http.response.start':
                    # Cleanup if needed
                    pass
                await send(event)

            await self.app(scope, receive, modified_send)

app.add_middleware(OpenTracingContextMiddleware, 
position=MiddlewarePosition.BEFORE_EXCEPTION)
```

2. Global Context Accessor Functions

python

```python
from opentracing import tracer as opentracing_tracer

def get_request_context(key):
    """Get context value from active span"""
    span = opentracing_tracer.active_span
    if span:
        return span.get_tag(key)
    return None

def set_request_context(key, value):
    """Set context value in active span"""
    span = opentracing_tracer.active_span
    if span:
        span.set_tag(key, value)

# Convenience methods for account_id
```

```python
def get_account_id():
    return get_request_context('account.id')

def set_account_id(account_id):
    set_request_context('account.id', account_id)
```

3. Celery Integration

python

```python
from celery import Celery
from opentracing.propagation import Format
import opentracing

app = Celery()

@app.task
def background_task(headers, *args, **kwargs):
    # Extract context from headers
    span_ctx = opentracing_tracer.extract(Format.TEXT_MAP, headers)

    with opentracing_tracer.start_active_span(
        'background_task',
        child_of=span_ctx
    ):
        # Access context from span
        account_id = get_account_id()
        # ... rest of task

# When calling the task:
def delay_background_task(*args, **kwargs):
    headers = {}
    span = opentracing_tracer.active_span
    if span:
        opentracing_tracer.inject(span.context, Format.TEXT_MAP, headers)
    background_task.delay(headers, *args, **kwargs)
```

4. SQLAlchemy Instrumentation

python

```python
from sqlalchemy.event import listen
from sqlalchemy.engine import Engine

def before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
```

```
    account_id = get_account_id()
    if account_id:
        # Add to SQL comment
        statement = f"-- account_id: {account_id}
\n{statement}"
        
        # Add as span tag
        span = opentracing_tracer.active_span
        if span:
            span.set_tag('sql.account_id', account_id)

listen(Engine, 'before_cursor_execute', 
before_cursor_execute)
```

Initialization Setup

python

```python
# tracing.py
from jaeger_client import Config
from opentracing import set_global_tracer, tracer as 
opentracing_tracer

def init_tracing(service_name):
    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name=service_name
    )
    tracer = config.initialize_tracer()
    set_global_tracer(tracer)
    return tracer

# In your application startup:
opentracing_tracer = init_tracing('your-service-name')
```

Key Benefits

1. **Simplified Architecture**: Uses only OpenTracing
2. **Consistent Context**: Maintains request context through span tags
3. **Backward Compatible**: Works with existing OpenTracing

instrumentation
4. **Distributed Tracing:** Properly propagates context to Celery tasks
5. **Minimal Changes:** Replaces Flask.g usage with span tags

Migration Steps

1. Replace all `Flask.g` references with the new accessor functions
2. Update your task calling code to inject headers
3. Ensure all middleware and instrumentation points use the span-based context
4. Remove any OpenTelemetry-related code if present

This solution provides a clean migration path from your current Flask.g-based implementation to a proper OpenTracing context propagation system in Connexion v3.

Here is some doc. As Connexion v3 is based on ASGI, I was planning to use some contextvar context manager. This library is a Python platform API for OpenTracing.

Required Reading

In order to understand the Python platform API, one must first be familiar with the OpenTracing project and terminology more specifically.

Status

In the current version, opentracing-python provides only the API and a basic no-op implementation that can be used by instrumentation libraries to collect and propagate distributed tracing context.

Future versions will include a reference implementation utilizing an abstract Recorder interface, as well as a Zipkin-compatible Tracer.

Usage

The work of instrumentation libraries generally consists of three steps:

When a service receives a new request (over HTTP or some other protocol), it uses OpenTracing's inject/extract API to continue an active trace, creating a Span object in the process. If the request does not contain an active trace, the service starts a new trace and a new root Span.
The service needs to store the current Span in some request-local storage, (called Span activation) where it can be retrieved from when a child Span must be created, e.g. in case of the service making an RPC to another service.
When making outbound calls to another service, the current Span must be retrieved from request-local storage, a child span must be created (e.g., by using the start_child_span() helper), and that child span must be embedded into the outbound request (e.g., using HTTP headers) via OpenTracing's inject/extract API.
Below are the code examples for the previously mentioned steps. Implementation of request-local storage needed for step 2 is specific to the service and/or frameworks / instrumentation libraries it is using, exposed as a ScopeManager child contained as Tracer.scope_manager. See details below.

Inbound request

Somewhere in your server's request handler code:

```
def handle_request(request):
    span = before_request(request, opentracing.tracer)
```

```
    # store span in some request-local storage using
Tracer.scope_manager,
    # using the returned `Scope` as Context Manager to ensure
    # `Span` will be cleared and (in this case) `Span.finish()` be
called.
    with tracer.scope_manager.activate(span, True) as scope:
        # actual business logic
        handle_request_for_real(request)


def before_request(request, tracer):
    span_context = tracer.extract(
        format=Format.HTTP_HEADERS,
        carrier=request.headers,
    )
    span = tracer.start_span(
        operation_name=request.operation,
        child_of(span_context))
    span.set_tag('http.url', request.full_url)

    remote_ip = request.remote_ip
    if remote_ip:
        span.set_tag(tags.PEER_HOST_IPV4, remote_ip)

    caller_name = request.caller_name
    if caller_name:
        span.set_tag(tags.PEER_SERVICE, caller_name)

    remote_port = request.remote_port
    if remote_port:
        span.set_tag(tags.PEER_PORT, remote_port)

    return span
```

Outbound request

Somewhere in your service that’s about to make an outgoing call:

```
from opentracing.ext import tags
from opentracing.propagation import Format
from opentracing_instrumentation import request_context

# create and serialize a child span and use it as context
manager
with before_http_request(
    request=out_request,
    current_span_extractor=request_context.get_current_span):

    # actual call
    return urllib2.urlopen(request)


def before_http_request(request, current_span_extractor):
    op = request.operation
    parent_span = current_span_extractor()
    outbound_span = opentracing.tracer.start_span(
        operation_name=op,
        child_of=parent_span
    )

    outbound_span.set_tag('http.url', request.full_url)
    service_name = request.service_name
    host, port = request.host_port
    if service_name:
        outbound_span.set_tag(tags.PEER_SERVICE,
service_name)
    if host:
        outbound_span.set_tag(tags.PEER_HOST_IPV4, host)
    if port:
        outbound_span.set_tag(tags.PEER_PORT, port)

    http_header_carrier = {}
    opentracing.tracer.inject(
        span_context=outbound_span,
        format=Format.HTTP_HEADERS,
        carrier=http_header_carrier)
```

```
    for key, value in http_header_carrier.iteritems():
        request.add_header(key, value)

    return outbound_span
```

Scope and within-process propagation

For getting/setting the current active Span in the used request-local storage, OpenTracing requires that every Tracer contains a ScopeManager that grants access to the active Span through a Scope. Any Span may be transferred to another task or thread, but not Scope.

```
# Access to the active span is straightforward.
scope = tracer.scope_manager.active()
if scope is not None:
    scope.span.set_tag('...', '...')
```

The common case starts a Scope that's automatically registered for intra-process propagation via ScopeManager.

Note that start_active_span('...') automatically finishes the span on Scope.close() (start_active_span('...', finish_on_close=False) does not finish it, in contrast).

```
# Manual activation of the Span.
span = tracer.start_span(operation_name='someWork')
with tracer.scope_manager.activate(span, True) as scope:
    # Do things.

# Automatic activation of the Span.
# finish_on_close is a required parameter.
with tracer.start_active_span('someWork',
finish_on_close=True) as scope:
    # Do things.

# Handling done through a try construct:
span = tracer.start_span(operation_name='someWork')
scope = tracer.scope_manager.activate(span, True)
```

```
try:
    # Do things.
except Exception as e:
    scope.set_tag('error', '...')
finally:
    scope.finish()
```

If there is a Scope, it will act as the parent to any newly started Span unless the programmer passes ignore_active_span=True at start_span()/start_active_span() time or specified parent context explicitly:

```
scope = tracer.start_active_span('someWork',
ignore_active_span=True)
```

Each service/framework ought to provide a specific ScopeManager implementation that relies on their own request-local storage (thread-local storage, or coroutine-based storage for asynchronous frameworks, for example).

Scope managers

This project includes a set of ScopeManager implementations under the opentracing.scope_managers submodule, which can be imported on demand:

```
from opentracing.scope_managers import
ThreadLocalScopeManager
```

There exist implementations for thread-local (the default), gevent, Tornado and asyncio:

```
from opentracing.scope_managers.gevent import
GeventScopeManager # requires gevent
from opentracing.scope_managers.tornado import
TornadoScopeManager # requires Tornado
from opentracing.scope_managers.asyncio import
AsyncioScopeManager # requires Python 3.4 or newer.
```

ASGI-Compatible OpenTracing Solution Using ContextVars for Connexion v3

Based on your requirements and the OpenTracing documentation, here's a complete solution using `contextvars` for request-local storage in an ASGI environment with Connexion v3.

Core Implementation

1. ContextVar-based Scope Manager

python

```python
from contextvars import ContextVar
from opentracing import Scope, ScopeManager, Span
import typing

class ContextVarsScopeManager(ScopeManager):
    def __init__(self):
        self._active_span: ContextVar[typing.Optional[Scope]] = ContextVar(
            'active_span', default=None
        )

    def activate(self, span: Span, finish_on_close: bool = True) -> Scope:
        scope = Scope(self, span, finish_on_close)
        self._active_span.set(scope)
        return scope

    @property
    def active(self) -> typing.Optional[Scope]:
        return self._active_span.get()
```

2. Tracer Initialization

python

```python
from jaeger_client import Config
from opentracing import Tracer
```

```python
def init_tracing(service_name: str) -> Tracer:
    config = Config(
        config={
            'sampler': {'type': 'const', 'param': 1},
            'logging': True,
        },
        service_name=service_name,
        scope_manager=ContextVarsScopeManager()  # Use our custom scope manager
    )
    return config.initialize_tracer()

# Initialize during app startup
tracer = init_tracing('your-service-name')
```

3. ASGI Middleware for Connexion v3

python

```python
from opentracing.propagation import Format
from opentracing import tags
from starlette.types import ASGIApp, Scope, Receive, Send

class OpenTracingMiddleware:
    def __init__(self, app: ASGIApp):
        self.app = app

    async def __call__(self, scope: Scope, receive: Receive, send: Send):
        if scope['type'] != 'http':
            return await self.app(scope, receive, send)

        headers = {k.decode(): v.decode() for k, v in scope['headers']}
        request_span = self._create_span_from_headers(headers, scope)

        with tracer.start_active_span(
            operation_name=f"{scope['method']} {scope['path']}",
            child_of=request_span,
            finish_on_close=True
        ) as scope_ctx:
            # Store additional context in the span
            self._enrich_span(scope_ctx.span, scope)

            async def modified_send(event):
```

```
                if event['type'] == 'http.response.start':
                    self._handle_response(scope_ctx.span, event)
                await send(event)

            await self.app(scope, receive, modified_send)

    def _create_span_from_headers(self, headers, scope):
        try:
            span_ctx = tracer.extract(Format.HTTP_HEADERS, headers)
            return span_ctx
        except Exception:
            return None

    def _enrich_span(self, span, scope):
        span.set_tag(tags.HTTP_METHOD, scope['method'])
        span.set_tag(tags.HTTP_URL, scope['path'])
        span.set_tag('component', 'asgi')
        
        # Store account_id if available
        account_id = headers.get('x-account-id')
        if account_id:
            span.set_tag('account.id', account_id)

    def _handle_response(self, span, event):
        span.set_tag(tags.HTTP_STATUS_CODE, event['status'])
        if event['status'] >= 400:
            span.set_tag(tags.ERROR, True)
```

4. Context Access Utilities

python

```
from opentracing import tracer as opentracing_tracer

def get_current_span():
    scope = opentracing_tracer.scope_manager.active
    return scope.span if scope else None

def get_request_context(key):
    span = get_current_span()
    return span.get_tag(key) if span else None

def set_request_context(key, value):
    span = get_current_span()
    if span:
```

```
        span.set_tag(key, value)

# Convenience methods
def get_account_id():
    return get_request_context('account.id')

def set_account_id(account_id):
    set_request_context('account.id', account_id)
```

5. Celery Integration

python

```python
from celery import Celery
from opentracing.propagation import Format

app = Celery()

@app.task
def background_task(headers, *args, **kwargs):
    span_ctx = tracer.extract(Format.TEXT_MAP, headers)
    
    with tracer.start_active_span(
        'background_task',
        child_of=span_ctx,
        finish_on_close=True
    ) as scope:
        # Access context from span
        account_id = get_account_id()
        # ... rest of task logic

def delay_background_task(*args, **kwargs):
    headers = {}
    span = get_current_span()
    if span:
        tracer.inject(span.context, Format.TEXT_MAP, 
headers)
    background_task.delay(headers, *args, **kwargs)
```

Application Setup

python

```python
from connexion import AsyncApp

app = AsyncApp(__name__)
app.add_middleware(OpenTracingMiddleware)
```

```
# Register your APIs
app.add_api('openapi.yaml')
```

Key Features

1. **ASGI-Compatible:** Works seamlessly with Connexion v3's ASGI foundation
2. **ContextVars-Based:** Properly handles request-local storage in async contexts
3. **Full OpenTracing Support:** Implements all required OpenTracing interfaces
4. **Distributed Tracing:** Proper context propagation to Celery tasks
5. **Consistent Context Access:** Simple API for getting/setting request context

Benefits Over Previous Solutions

1. **Proper Async Support:** Uses ContextVars instead of thread-locals
2. **Cleaner Integration:** Directly implements OpenTracing's ScopeManager
3. **More Robust:** Better handles edge cases in ASGI environments
4. **Future-Proof:** Ready for async Python and modern web frameworks

This implementation gives you a production-ready tracing solution that properly handles context propagation in Connexion v3 while maintaining all your existing OpenTracing instrumentation.

Looks like your contextvars implementation is not good. They have implemented one here opentracing/scope_managers/contextvars.py. # Copyright (c) The OpenTracing Authors.
#

```
from __future__ import absolute_import

from contextlib import contextmanager
from contextvars import ContextVar

from opentracing import Scope, ScopeManager
```

```
_SCOPE = ContextVar('scope')


class ContextVarsScopeManager(ScopeManager):
    """
    :class:`~opentracing.ScopeManager` implementation for
**asyncio**
    that stores the :class:`~opentracing.Scope` using
ContextVar.

    The scope manager provides
automatic :class:`~opentracing.Span` propagation
    from parent coroutines, tasks and scheduled in event loop
callbacks to
    their children.

    .. code-block:: python

        async def child_coroutine():
            # No need manual activation of parent span in child
coroutine.
            with tracer.start_active_span('child') as scope:
                ...

        async def parent_coroutine():
            with tracer.start_active_span('parent') as scope:
                ...
                await child_coroutine()
                ...

    """

    def activate(self, span, finish_on_close):
        """
        Make a :class:`~opentracing.Span` instance active.

        :param span: the :class:`~opentracing.Span` that should
become active.
```

```
        :param finish_on_close: whether *span* should
automatically be
            finished when :meth:`Scope.close()` is called.

        :return: a :class:`~opentracing.Scope` instance to control
the end
            of the active period for the :class:`~opentracing.Span`.
            It is a programming error to neglect to
call :meth:`Scope.close()`
            on the returned instance.
        """

        return self._set_scope(span, finish_on_close)

    @property
    def active(self):
        """
        Return the currently active :class:`~opentracing.Scope`
which
        can be used to access the currently
active :attr:`Scope.span`.

        :return: the :class:`~opentracing.Scope` that is active,
            or ``None`` if not available.
        """

        return self._get_scope()

    def _set_scope(self, span, finish_on_close):
        return _ContextVarsScope(self, span, finish_on_close)

    def _get_scope(self):
        return _SCOPE.get(None)


class _ContextVarsScope(Scope):
    def __init__(self, manager, span, finish_on_close):
        super(_ContextVarsScope, self).__init__(manager, span)
```

```
        self._finish_on_close = finish_on_close
        self._token = _SCOPE.set(self)

    def close(self):
        if self.manager.active is not self:
            return

        _SCOPE.reset(self._token)

        if self._finish_on_close:
            self.span.finish()


@contextmanager
def no_parent_scope():
    """
    Context manager that resets current Scope. Intended to
break span
    propagation to children coroutines, tasks or scheduled
callbacks.

    .. code-block:: python

        from opentracing.scope_managers.contextvars import
no_parent_scope

        def periodic()
            # `periodic` span will be children of root only at the first
time.
            with self.tracer.start_active_span('periodic'):
                # Now we break span propagation.
                with no_parent_scope():
                    self.loop.call_soon(periodic)

        with self.tracer.start_active_span('root'):
            self.loop.call_soon(periodic)
    """
    token = _SCOPE.set(None)
```

```
    try:
        yield
    finally:
        _SCOPE.reset(token)
```

Improved OpenTracing Solution Using Official ContextVarsScopeManager

Since OpenTracing already provides a well-tested `ContextVarsScopeManager`, we should use that instead of a custom implementation. Here's the complete solution for Connexion v3:

1. Tracer Initialization with Official Scope Manager

python

```python
from jaeger_client import Config
from opentracing.scope_managers.contextvars import ContextVarsScopeManager

def init_tracing(service_name: str):
    config = Config(
        config={
            'sampler': {'type': 'const', 'param': 1},
            'logging': True,
        },
        service_name=service_name,
        scope_manager=ContextVarsScopeManager()  # Use official implementation
    )
    return config.initialize_tracer()

tracer = init_tracing('your-service-name')
```

2. ASGI Middleware for Connexion v3

python

```python
from opentracing.propagation import Format
from opentracing import tags
from starlette.types import ASGIApp, Scope, Receive, Send

class OpenTracingMiddleware:
    def __init__(self, app: ASGIApp):
        self.app = app

    async def __call__(self, scope: Scope, receive: Receive, send: Send):
        if scope['type'] != 'http':
            return await self.app(scope, receive, send)

        headers = {k.decode(): v.decode() for k, v in scope['headers']}
        span_ctx = self._extract_span_context(headers)
        
        with tracer.start_active_span(
            f"{scope['method']} {scope['path']}",
            child_of=span_ctx,
            finish_on_close=True
        ) as scope:
            self._enrich_span(scope.span, scope, headers)
            
            async def modified_send(event):
                if event['type'] == 'http.response.start':
                    self._handle_response(scope.span, event)
                await send(event)

            await self.app(scope, receive, modified_send)

    def _extract_span_context(self, headers):
        try:
            return tracer.extract(Format.HTTP_HEADERS, headers)
        except Exception:
            return None

    def _enrich_span(self, span, scope, headers):
        span.set_tag(tags.HTTP_METHOD, scope['method'])
        span.set_tag(tags.HTTP_URL, scope['path'])
```

```
        span.set_tag('component', 'asgi')

        # Store account_id if available
        if account_id := headers.get('x-account-id'):
            span.set_tag('account.id', account_id)

    def _handle_response(self, span, event):
        span.set_tag(tags.HTTP_STATUS_CODE,
event['status'])
        if event['status'] >= 400:
            span.set_tag(tags.ERROR, True)
```

3. Context Access Utilities

python

```
from opentracing import tracer as opentracing_tracer

def get_current_span():
    scope = opentracing_tracer.scope_manager.active
    return scope.span if scope else None

def get_request_context(key):
    span = get_current_span()
    return span.get_tag(key) if span else None

def set_request_context(key, value):
    span = get_current_span()
    if span:
        span.set_tag(key, value)

# Convenience methods
def get_account_id():
    return get_request_context('account.id')

def set_account_id(account_id):
    set_request_context('account.id', account_id)
```

4. Celery Integration with Context Propagation

python

```
from celery import Celery
from opentracing.propagation import Format

app = Celery()
```

```
@app.task
def background_task(headers, *args, **kwargs):
    span_ctx = tracer.extract(Format.TEXT_MAP, headers)
    
    with tracer.start_active_span(
        'background_task',
        child_of=span_ctx,
        finish_on_close=True
    ) as scope:
        # Access context from span
        account_id = get_account_id()
        # ... rest of task logic

def delay_background_task(*args, **kwargs):
    headers = {}
    span = get_current_span()
    if span:
        tracer.inject(span.context, Format.TEXT_MAP, 
headers)
    background_task.delay(headers, *args, **kwargs)
```

5. Application Setup

python

```python
from connexion import AsyncApp

app = AsyncApp(__name__)
app.add_middleware(OpenTracingMiddleware)

# Register your APIs
app.add_api('openapi.yaml')
```

Key Improvements

1. **Official Scope Manager**: Uses the battle-tested `ContextVarsScopeManager` from OpenTracing
2. **Proper Context Handling**: Correctly manages context in async environments
3. **Cleaner Middleware**: More focused implementation with better separation of concerns
4. **Full OpenTracing Compliance**: Maintains all standard OpenTracing functionality

Usage Patterns

In Request Handlers

python

```
async def some_endpoint():
    account_id = get_account_id()  # Access stored context
    # ... handler logic
    set_request_context('new_key', 'value')  # Store new context
```

In Background Tasks

python

```
# Start task with context
delay_background_task(arg1, arg2)

# Inside task:
def background_task(headers, arg1, arg2):
    # Context automatically restored from headers
    account_id = get_account_id()
```

This solution provides a robust, production-ready implementation that properly handles context propagation in Connexion v3 while using the official OpenTracing contextvars implementation.

span does not have get_tag method

You're absolutely right to point that out - my mistake! The OpenTracing `Span` class doesn't actually have a `get_tag` method. Let me provide a corrected implementation that properly handles span context storage and retrieval.

Corrected Implementation

1. Context Storage Approach

Instead of trying to read tags back from spans (which isn't supported), we'll:

1. Store custom context in the span's `context` object
2. Or maintain a separate context dictionary in our scope manager

Here's the corrected solution:

2. Updated Context Utilities

python

```python
from opentracing import tracer as opentracing_tracer

def get_current_span():
    scope = opentracing_tracer.scope_manager.active
    return scope.span if scope else None

def get_request_context(key):
    span = get_current_span()
    if not span:
        return None
    
    # Option 1: If you can modify the span context class
    if hasattr(span, 'context') and hasattr(span.context, 
'custom_context'):
        return span.context.custom_context.get(key)
    
    # Option 2: Using span's baggage (built-in feature)
    return span.get_baggage_item(key)

def set_request_context(key, value):
    span = get_current_span()
    if not span:
        return
    
    # Option 1: If you can modify the span context class
    if hasattr(span, 'context') and hasattr(span.context, 
'custom_context'):
        if not hasattr(span.context, 'custom_context'):
            span.context.custom_context = {}
        span.context.custom_context[key] = value
    
```

```
    # Option 2: Using span's baggage (preferred standard approach)
    span.set_baggage_item(key, str(value))

# Convenience methods
def get_account_id():
    return get_request_context('account.id')

def set_account_id(account_id):
    set_request_context('account.id', account_id)
```

3. Recommended Approach: Using Baggage Items

The OpenTracing standard way to handle this is using baggage items:

python

```python
def get_request_context(key):
    span = get_current_span()
    return span.get_baggage_item(key) if span else None

def set_request_context(key, value):
    span = get_current_span()
    if span:
        span.set_baggage_item(key, str(value))
```

4. Updated Middleware to Use Baggage

python

```python
class OpenTracingMiddleware:
    # ... (previous methods remain the same)
    
    def _enrich_span(self, span, scope, headers):
        span.set_tag(tags.HTTP_METHOD, scope['method'])
        span.set_tag(tags.HTTP_URL, scope['path'])
        
        # Store account_id in baggage
        if account_id := headers.get('x-account-id'):
            span.set_baggage_item('account.id', account_id)
```

5. Accessing Context in Handlers

python

```python
async def some_endpoint():
    account_id = get_account_id()  # Retrieves from baggage
    if not account_id:
```

```
        raise Unauthorized()

    # Store additional context
    set_request_context('request_type', 'api_call')
```

Key Changes:

1. **Using Baggage Items**: The proper OpenTracing way to propagate context
    - `set_baggage_item()` to store values
    - `get_baggage_item()` to retrieve values
    - Automatically propagates to child spans
2. **Backward Compatibility**: Works with all OpenTracing implementations
3. **Type Safety**: Remember to convert non-string values when storing

Why This Works Better:

1. **Standard-Compliant**: Uses official OpenTracing baggage API
2. **Propagates Automatically**: Baggage items flow with the trace context
3. **No Hacks**: Doesn't rely on undocumented features
4. **Works Distributed**: Baggage propagates across service boundaries

Migration Notes:

1. Replace all `span.get_tag()` calls with `get_baggage_item()`
2. Replace `span.set_tag()` for context with `set_baggage_item()`
3. Remember baggage values must be strings (do conversion if needed)

This approach gives you a robust, standards-compliant way to handle request context in your Connexion v3 application while properly using the OpenTracing API.

is it ok to create a startlette request object like starlette.Request(scope, receive, send) for easy access to fields to enrich the trace ?

Yes, creating a Starlette `Request` object within your ASGI middleware is perfectly fine and can be quite beneficial for enriching your traces. Here's why and how to do it properly:

Recommended Approach

python

```
from starlette.requests import Request
from opentracing import tags

class OpenTracingMiddleware:
    def __init__(self, app: ASGIApp):
        self.app = app

    async def __call__(self, scope: Scope, receive: Receive, send: Send):
        if scope['type'] != 'http':
            return await self.app(scope, receive, send)

        request = Request(scope, receive, send)  # Create request object
        headers = dict(request.headers)

        span_ctx = tracer.extract(Format.HTTP_HEADERS, headers)

        with tracer.start_active_span(
            f"{request.method} {request.url.path}",
            child_of=span_ctx,
            finish_on_close=True
        ) as scope:
            self._enrich_span(scope.span, request)

            async def modified_send(event):
```

```python
                if event['type'] == 'http.response.start':
                    self._handle_response(scope.span, 
event)
                await send(event)

            await self.app(scope, receive, modified_send)

    def _enrich_span(self, span, request: Request):
        """Enrich span with request details"""
        span.set_tag(tags.HTTP_METHOD, request.method)
        span.set_tag(tags.HTTP_URL, str(request.url))

        # Client information
        if request.client:
            span.set_tag('client.ip', request.client.host)
            span.set_tag('client.port', 
request.client.port)

        # User agent
        if 'user-agent' in request.headers:
            span.set_tag('user_agent', 
request.headers['user-agent'])

        # Store important headers as baggage
        for header in ['x-account-id', 'x-request-id']:
            if header in request.headers:
                span.set_baggage_item(header.replace('x-', 
''), request.headers[header])
```

Benefits of Using Starlette Request

1. **Easier Access to Common Fields:**
   - `request.method` instead of `scope['method']`
   - `request.url` instead of manually constructing from `scope['path']` and `scope['scheme']`
   - `request.headers` as a proper dict rather than raw ASGI headers
2. **Additional Convenience Features:**
   - Built-in client info (`request.client`)
   - Proper URL handling (`request.url`)
   - Query parameter access (`request.query_params`)
3. **Cleaner Code:**
   - More readable than working with raw ASGI scope
   - Better type hints and IDE support

Performance Considerations

1. **Minimal Overhead**: The `Request` object is lightweight and designed for this purpose
2. **Single Instantiation**: We only create it once per request
3. **Lazy Evaluation**: Starlette's Request object lazily evaluates most properties

Advanced Usage

For even richer tracing, you can:

python

```
def _enrich_span(self, span, request: Request):
    # Authentication context
    if hasattr(request, 'user') and request.user:
        span.set_tag('user.id', request.user.id)
        span.set_baggage_item('user.id', 
str(request.user.id))
    
    # Query parameters
    if request.query_params:
        span.set_tag('query_params', 
dict(request.query_params))
    
    # Request body metadata (without loading full body)
    if 'content-length' in request.headers:
        span.set_tag('request.size', 
request.headers['content-length'])
```

Complete Middleware Example

Here's a production-ready version:

python

```
from starlette.requests import Request
from starlette.responses import Response
from opentracing import Format, tags

class OpenTracingMiddleware:
    def __init__(self, app: ASGIApp, tracer=None):
        self.app = app
        self.tracer = tracer or opentracing.global_tracer()
```

```
    async def __call__(self, scope: Scope, receive: 
Receive, send: Send):
        if scope['type'] not in ('http', 'websocket'):
            return await self.app(scope, receive, send)

        request = Request(scope, receive, send)
        
        try:
            span_ctx = 
self.tracer.extract(Format.HTTP_HEADERS, 
dict(request.headers))
        except Exception:
            span_ctx = None

        with self.tracer.start_active_span(
            self._get_operation_name(request),
            child_of=span_ctx,
            finish_on_close=True
        ) as scope:
            self._enrich_span(scope.span, request)
            
            async def wrapped_send(message):
                if message['type'] == 
'http.response.start':
                    self._handle_response(scope.span, 
message)
                await send(message)

            try:
                await self.app(scope, receive, 
wrapped_send)
            except Exception as exc:
                self._handle_exception(scope.span, exc)
                raise

    def _get_operation_name(self, request: Request) -> str:
        return f"{request.method} {request.url.path}"

    def _enrich_span(self, span, request: Request):
        span.set_tag(tags.HTTP_METHOD, request.method)
        span.set_tag(tags.HTTP_URL, str(request.url))
        span.set_tag('component', 'asgi')
        
        if request.client:
            span.set_tag(tags.PEER_HOST_IP, 
request.client.host)
            span.set_tag(tags.PEER_PORT, 
```

```
request.client.port)

        # Store account ID from headers if present
        if 'x-account-id' in request.headers:
            span.set_baggage_item('account.id', 
request.headers['x-account-id'])

    def _handle_response(self, span, message):
        span.set_tag(tags.HTTP_STATUS_CODE, 
message['status'])
        if message['status'] >= 400:
            span.set_tag(tags.ERROR, True)

    def _handle_exception(self, span, exc):
        span.set_tag(tags.ERROR, True)
        span.log_kv({
            'event': 'error',
            'error.object': exc,
            'message': str(exc),
            'stack': traceback.format_exc()
        })
```

This implementation provides:

1. Full request/response tracing
2. Error handling
3. Clean separation of concerns
4. Support for both HTTP and WebSockets
5. Proper baggage propagation

The Starlette `Request` object makes all this significantly cleaner and more maintainable than working with raw ASGI scope.

New chat