From Flask to aiohttp

Skyscanner Engineering Sep 1, 2016 · 6 min read

*By **Manuel Miranda***

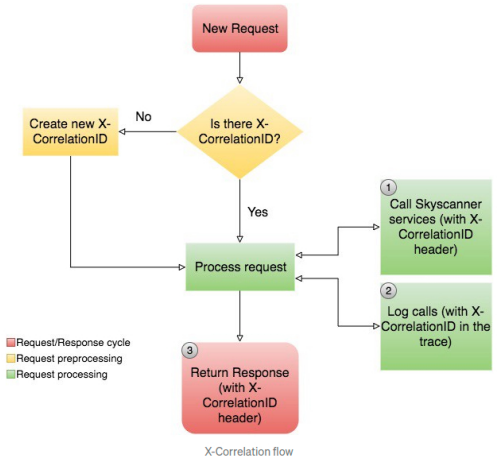
This post is about how to have a global context shared during the flow of a request in aiohttp.

> ***UPDATE August 2017:*** *After some more iterations I've ended up releasing aiotask-context. The package is an improved version of what I describe in the post because it uses a custom Task factory rather than modifying manually each Task to propagate context.*
> *Also note there is a PEP-550 being worked to support a similar functionality by default in Python*

Why?

In Skyscanner hotels we are developing a new service with Python 3 (h*ll yeah!), asyncio and aiohttp among other tools. As you can imagine, the company architecture is full of different micro services and tracking user journey through them can be really painful. That's why there is a guideline telling that all services should use something that allows us to track this journey between all services. This something is the *X-CorrelationID* header. So, to ensure proper traceability, what our service should do is:

1. All calls to Skyscanner services should send the X-CorrelationID header.
2. All log traces related to a request/response cycle should contain the X-CorrelationID.
3. Return the X-CorrelationID used in the Response.



X-Correlation flow

From the diagram above, you can see that we will be reusing the header in many places (calls to services, log calls, etc…). Knowing that you may realize that this header should be stored somewhere accessible from **everywhere** in our code.

If you've ever worked with aiohttp, you may have seen that there is no way of sharing state or storing global variables within the request/response cycle. The way of sharing the information of the request is by propagating the ClientRequest object throughout all the function calls. If you've ever worked with Django, it's the same pattern.

Obviously, we could do that and finish the post here, but this is totally against clean code practices, maintainability, DRY and some other best practices principles. So, the question is, how can we share this variable during all the request/response cycle without passing it explicitly?

At that point, I decided to do some research, ask other engineers about similar patterns, check code from other tools/frameworks, etc… After a while my brain looked more or less like that:

So yes, I came up with an interesting pattern which is how Flask is using threads to store local information like the request object. I won't go deep into how Flask works, but just to give an idea let's read a paragraph and see some code extracted from "how the context works" docs section (take your time):

> The method **request_context()** returns a new **RequestContext** object and uses it in combination with the with statement to bind the context. **Everything** that is called from the **same thread** from this point onwards until the end of the with statement **will have access to the request globals** (flask.request and others).

```
def wsgi_app(self, environ):
    with self.request_context(environ):
        try:
            response = self.full_dispatch_request()
        except Exception as e:
            response = self.make_response(self.handle_exception(e))
        return response(environ, start_response)
```

So, that piece of code means that, everything that gets executed inside the context manager, will have access to the **request** object. Awesome isn't it? by just executing from *flask import request* in **any** section of our code, if we are inside the context manager call, it will return us the request object belonging to the current request/response cycle!

Clean and simple right? After digging into that, my thought was, can we do that with aiohttp? The answer is yes, next section describes how we have implemented a similar behavior with aiohttp (only with the header).

How? The implementation

To recap the previous section: "We want a variable to be easily accessible from any part of our code during the request/response cycle without the need to pass it explicitly to all function calls".

Python coroutines are executed within the asyncio loop. This loop is the one in charge of picking Futures, Tasks, etc and executing them. Every time you use an *asyncio.ensure_future, await* and other asynchronous calls, the code is executed within a *Task* instance which is scheduled inside the loop. You can think about Tasks as small units to be processed sequentially by the loop.

This gives us an object where we can store this shared data throughout the cycle. Here some things to keep in mind:

- aiohttp request/response cycle is executed within a **single Task**.
- Every time a coroutine is called with the *await* or *yield from* syntax, the code is executed in the same **Task**.
- Other calls like *asyncio.ensure_future*, *asyncio.call_soon*, etc… create a new **Task** instance. If we want to share the context, we will have to do something there.

Seems we are onto something right? The object we want to work with is **Task**. After checking its API reference you can see there isn't a structure, function call or anything that allows us to store context information but, since we are in python, we can just do *task.context = {"X-CorrelationID": "1234"}*.

Integrating task context with aiohttp

If you've read the previous section, you know that we want to store the "X-CorrelationID" header to be easily accessible during all the request/response cycle to be able to use it during log calls, external services calls and return it in the response. To do that, we've coded this simple middleware:

```
import context

async def correlation_id_middleware(app, handler):

    async def middleware_handler(request):
        context.set("X-CorrelationID", request.headers.get("X-
CorrelationID", str(uuid.uuid4())))
        response = await handler(request)
        response.headers["X-CorrelationID"] = context.get("X-
CorrelationID")
        return response

    return middleware_handler
```

Note the *import context* line. The module is just a proxy to the *Task.context* attribute of the current Task being executed:

```
import asyncio

def get(key, default=None):
    try:
        return asyncio.Task.current_task().context[key]
    except (KeyError, AttributeError):
        return default

def set(key, value):
    try:
        asyncio.Task.current_task().context[key] = value
    except AttributeError:
        asyncio.Task.current_task().context = {key: value}
```

Easy peasy right? by just calling *context.get(key)* we will get the value stored in *Task.context[key]* where Task is the current one being executed. By just calling *context.set* we will set the value for the given key.

Note that, from now on you will be able to do a *context.get("X-CorrelationID")* from **ANY** part of your code and it will return the needed value if existed. This for example, allows us to inject the X-CorrelationID in our logs automatically using a custom *logging.Filter*:

```
import context

class CorrelationIdFilter(logging.Filter):
# To make this work, you must add %(correlationid)s
# inside the log format definition.

    def filter(self, record):
        record.correlationid = context.get("X-CorrelationID",
"unknown")
        return True
```

Same pattern used for injecting the header when needed to call an internal service from Skyscanner:

```
import context

headers = { 'content-type': 'application/json', 'api-key':
self.api_key, 'X-CorrelationID': context.get("X-CorrelationID",
"unknown") }
```

For simple flows which cover most of the use cases, this works so far so good!

The ensure_future & co

As previously commented, the *ensure_future* call returns a new Task. This means that the custom *context* attribute we were using before is lost during the call. For our code, we solve this by creating a new *context.ensure_future* call that wraps the original one:

```
import asyncio

def ensure_future(coroutine):
    task = asyncio.ensure_future(coroutine)
    if hasattr(asyncio.Task.current_task(), "context"):
        task.context = asyncio.Task.current_task().context
    return task
```

This part is the one I'm less happy about because it's not transparent to the user. In future versions this will be improved.

What(,) now?

I've moved this simple code to a github repository and called it aiotask_context. Right now it only includes the functions for getting and setting variables in the current task. Some future work I'm planning:

- Implementing a mechanism to propagate the context when using

asyncio.ensure_future, asyncio.call_soon and similar calls. Candidates are wrapping (meh) or monkey patching (uhm…).

- Add more control to know if current code is being executed under a Task environment or not and act accordingly.
- Include examples in the repository like the aiohttp middleware, request passing, log fields injection, etc…

> ***UPDATE August 2017:*** *All the features mentioned above are implemented in the latest release of aiotask_context. The propagation of calls like asyncio.ensure_future & co is done using asyncio.set_task_factory. The factory is available here*

Just to finish, if you have any questions or feedback, don't hesitate to comment and ask!

References

Interesting links I used:

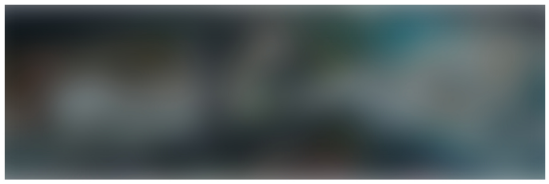
- Flask request context.
- Python task and coroutines docs.
- django-request-id source code for implementing the logging feature.
- Thread local storage from python threading (brief but useful to know).

Other projects implementing this pattern (haven't tried them):

- aiolocals
- tasklocals

Learn with us

Take a look at our current job roles available across our 10 global offices.



We're hiring!