Hi team, thank you for the constructive and motivating feedback. This gives me a glimpse of hope that I should not give up on this position! :)

note; I just started deep-dive researching the world of asynchronous requests to better understand what is going under the hood. Although most of the materials I’ve come across were kind of lacking intuitiveness, then I came up with these 2 Part GitHub blogposts which kind of cleared the fogs a bit:

<https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-1>

<https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-2>

**Feedback Review**

After carefully reviewing your feedback, I believe most of the code had to be adjusted to address points on feedback. So, few points I’ve come up with to discuss the drawbacks of previous code and their improvements in new version:

* The previous code was creating a new client for each request inside get\_time() coroutine function. Better and more accurate option will be to tackle all requests within single client. This way, we can assign endpoint’s timeout parameter to client’s timeout parameter.
* To your third point on feedback:
  + *Your server is waiting for all requests to finish before sending the first successful response. Beware that some requests can take a really long time and the “time” property in response does not correspond to the actual request duration.*

*🡪* apparently, the cause of this is **asyncio.gather()** function which waits to collect the given task results. As a result, the previous code was waiting for all the tasks to be finished before sending the first successful response. But, as we want to process tasks greedily as they are ready, we will use the loop over **asyncio.as\_completed()** [<https://bit.ly/as_compled>]. As the name implies, it helps us to get the earliest next result from awaitable tasks.

* To your second point on feedback:
  + “*Second and third requests are sent after the first request finishes instead of waiting 300ms since the request has been started.”*

*🡪* thecause, again is **asyncio.gather()** which waits the first request to finish. To address this, the closest I could came up with is to use **asyncio.wait\_for()** with **asyncio.shield()** [<https://bit.ly/wait_for_shield>].Purpose of usingwait\_for() is to have it as a notifier. We set its timeout parameter to 300ms and waiting for the first request to finish. Once the 300ms has passed, it will throw TimeoutError and we will know it is time to throw other two requests. But at the same time, we have to shield request with shield() function in order to tell the wait\_for() not to kill request once the timeout has exceeded. Because, we will still need the response to this first request.

*NOTE*: Besides the timeout notifier case, there is also one more case where we fire other two requests. It happens if the first request’s response is unsuccessful within 300ms (e.g., first request has response of *500 Internal Server Error* within 300ms).

**Discussion**

In this report we will discuss the workflow of api with various use cases. Let’s start:

First of all, we have only one api endpoint, which is /api/smart/<timeout\_parameter> which performs up to 3 HTTP requests to Exponea Testing HTTP Server (<https://exponea-engineering-assignment.appspot.com/api/work>) and returns first successful response. Note: if no successful request returned within the given timeout, api responds with error messages for each of 3 requests that failed.

Algorithm is built in this way:

First it sends one request to the testing server. If there is a successful response within 300 milliseconds, then the endpoint returns this response and doesn’t fire other requests.

But if the time of first request is bigger than 300 milliseconds, it fires concurrently (asynchronously) two other requests, and then returns the first successful request from any of the 3 requests, including the first one.

Example №1:

İf first request to testing server returns time equal to 270, then api returns answer like this {“time”: 270}

Example №2:

İf first request to testing server returns time equal to 350, then immediately it fires another 2 requests, which are 280 and 528 respectively, and returns the fastest one, means {“time”: 280}

Concurrent requests are made with help of httpx and asyncio, one of which helps to send the requests asynchronously, other one to gather data from the responses of requests.

But there are other cases, when there appear errors, such as Server timeout, Connection timeout, Internal server error (which comes from testing server), which require to be handled. In this case, if one request returns an exception, it just does nothing but writes it down and deals with other requests. For example, if second request returned an error, it just skips it and returns first successful response from first and third request, and so on. But in case if all requests are failed, it returns json response with the error message for each of the request.

Let me show some use cases:

Example №3:

First request to test server returned 350, then second and third are fired, second returns an error, and third returns 420. The response time will be 350

Example №4:

First request to test server returned 400, then second and third are fired, second returns an error, and third also returns an error. The response time will be 400

Example №5:

First request to test server returned an error, then second and third are fired, second returns 340, and third returns 450. The response time will be 340

Example №6:

First request to test server returned a server error, then second and third are fired, second returns a timeout error, and third also returns a timeout error. The response will be like that {1: Server error, 2: Timeout error, 3: Timeout error}

During the testing period, the testing server behaved differently. Sometimes it worked perfectly without errors, sometimes it returned almost all errors each time. It is advised to test Api regularly to avoid misunderstandings in app workflow.