4 Concurrent web requests

This chapter covers

- Asynchronous context managers
- Making asyncio-friendly web requests with aiohttp
- Running web requests concurrently with `gather`
- Processing results as they come in with `as completed`
- Keeping track of in-flight requests with `wait`
- Setting and handling timeouts for groups of requests and canceling requests

In chapter 3, we learned more about the inner workings of sockets and built a basic echo server. Now that we've seen how to design a basic application, we'll take this knowledge and apply it to making concurrent, non-blocking web requests. Utilizing asyncio for web requests allows us to make hundreds of them at the same time, cutting down on our application's runtime compared to a synchronous approach. This is useful for when we must make multiple requests to a set of REST APIs, as can happen in a microservice architecture or when we have a web crawling task. This approach also allows for other code to run as we're waiting for potentially long web requests to finish, allowing us to build more responsive applications.

In this chapter, we'll learn about an asynchronous library called *aiohttp* that enables this. This library uses non-blocking sockets to make web requests and returns coroutines for those requests, which we can then `await` for a result. Specifically, we'll learn how to take a list of hundreds of URLs we'd like to get the contents for, and run all those requests concurrently. In doing so, we'll examine the various API methods that asyncio provides to run coroutines at one time, allowing us to choose between waiting for everything to complete before moving on, or processing results as fast as they come in. In addition, we'll look at how to set timeouts for these requests, both at the individual request level as well as for a group of requests. We'll also see how to cancel a set of in-process requests, based on how other requests have performed. These API methods are useful not only for making web requests but also for whenever we need to run a group of coroutines or tasks concurrently. In fact, we'll use the functions we use here throughout the rest of this book, and you will use them extensively as an asyncio developer.

roducing aiohttp

In chapter 2, we mentioned that one of the problems that newcomers face when first starting with asyncio is trying to take their existing code and pepper it with `async` and `await` in hopes of a performance gain. In most cases, this won't work, and this is especially true when working with web requests, as most existing libraries are blocking.

One popular library for making web requests is the `requests` library. This library does not perform well with asyncio because it uses blocking sockets. This means that if we make a request, it will block the thread that it runs in, and since asyncio is single-threaded, our entire event loop will halt until that request finishes.

To address this issue and get concurrency, we need to use a library that is non-blocking all the way down to the socket layer. *aiohttp* (Asynchronous HTTP Client/Server for asyncio and Python) is one library that solves this problem with non-blocking sockets.

aiohttp is an open source library that is part of the *aio-libs* project, which is the self-described "set of asyncio-based libraries built with high quality" (see https://github.com/aio-libs). This library is a fully functioning web client as well as a web server, meaning it can make web requests, and developers can create async web servers using it. (Documentation for the library is available at https://docs.aiohttp.org/.) In this chapter, we’ll focus on the client side of aiohttp, but we will also see how to build web servers with it later in the book.

So how do we get started with aiohttp? The first thing to learn is to make a HTTP request. We’ll first need to learn a bit of new syntax for asynchronous context managers. Using this syntax will allow us to acquire and close HTTP sessions cleanly. As an asyncio developer, you will use this syntax frequently for asynchronously acquiring resources, such as database connections.

ynchronous context managers

In any programming language, dealing with resources that must be opened and then closed, such as files, is common. When dealing with these resources, we need to be careful about any exceptions that may be thrown. This is because if we open a resource and an exception is thrown, we may never execute any code to clean up, leaving us in a status with leaking resources. Dealing with this in Python is straightforward using a `finally` block. Though this example is not exactly Pythonic, we can always close a file even if an exception was thrown:

```
file = open('example.txt')

try:
    lines = file.readlines()
finally:
    file.close()
```

This solves the issue of a file handle being left open if there was an exception during `file.readlines`. The drawback is that we must remember to wrap everything in a `try` `finally`, and we also need to remember the methods to call to properly close our resource. This isn't too hard to do for files, as we just need to remember to close them, but we'd still like something more reusable, especially since our cleanup may be more complicated than just calling one method. Python has a language feature to deal with this known as a *context manager*. Using this, we can abstract the shutdown logic along with the `try/finally` block:

```
with open('example.txt') as file:
    lines = file.readlines()
```

This Pythonic way to manage files is a lot cleaner. If an exception is thrown in the `with` block, our file will automatically be closed. This works for synchronous resources, but what if we want to asynchronously use a resource with this syntax? In this case, the context manager syntax won't work, as it is designed to work only with synchronous Python code and not coroutines and tasks. Python introduced a new language feature to support this use case, called *asynchronous context managers*. The syntax is almost the same as for synchronous context managers with the difference being that we say `async` `with` instead of just `with`.

Asynchronous context managers are classes that implement two special coroutine methods, `__aenter__,` which asynchronously acquires a resource and `__aexit__`, which closes that resource. *The* `__aexit__` coroutine takes several arguments that deal with any exceptions that occur, which we won't review in this chapter.

To fully understand async context managers, let’s implement a simple one using the sockets we introduced in chapter 3. We can consider a client socket connection a resource we’d like to manage. When a client connects, we acquire a client connection. Once we are done with it, we clean up and close the connection. In chapter 3, we wrapped everything in a `try/finally` block, but we could have implemented an asynchronous context manager to do so instead.

Listing 4.1 An asynchronous context manager to wait for a client connection

```
import asyncio
import socket
from types import TracebackType
from typing import Optional, Type



class ConnectedSocket:

    def __init__(self, server_socket):
        self._connection = None
        self._server_socket = server_socket

    async def __aenter__(self):
        print('Entering context manager, waiting for connect
        loop = asyncio.get_event_loop()
        connection, address = await loop.sock_accept(self._s
        self._connection = connection
        print('Accepted a connection')
        return self._connection

    async def __aexit__(self,
                        exc_type: Optional[Type[BaseExceptio
```

```
                                exc_val: Optional[BaseException],
                                exc_tb: Optional[TracebackType]):
        print('Exiting context manager')
        self._connection.close()
        print('Closed connection')


async def main():
    loop = asyncio.get_event_loop()

    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_RE
    server_address = ('127.0.0.1', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    async with ConnectedSocket(server_socket) as connection:
        data = await loop.sock_recv(connection, 1024)
        print(data)


asyncio.run(main())
```

❶ This coroutine is called when we enter the with block. It waits until a client connects and returns the connection.

❷ This coroutine is called when we exit the with block. In it, we clean up any resources we use. In this case, we close the connection.

❸ This calls __aenter__ and waits for a client connection.

❹ After this statement, __aenter__ will execute, and we'll close our connection.

In the preceding listing, we created a `ConnectedSocket` async context manager. This class takes in a server socket, and in our `__aenter__` coroutine we wait for a client to connect. Once a client connects, we return that client's connection. This lets us access that connection in the `as` portion of our `async with` statement. Then, inside our `async with` block, we use that connection to wait for the client to send us data. Once this block finishes execution, the `__aexit__` coroutine runs and closes the connection. Assuming a client connects with Telnet and sends some test data, we should see output like the following when running this program:

```
Entering context manager, waiting for connection
Accepted a connection
b'test\r\n'
Exiting context manager
Closed connection
```

aiohttp uses async context managers extensively for acquiring HTTP sessions and connections, and we'll use this later in chapter 5 when dealing with async database connections and transactions. Normally, you won't need to write your own async context managers, but it's helpful to have an understanding of how they work and are different from normal context managers. Now that we've introduced context managers and their workings, let's use them with aiohttp to see how to make an asynchronous web request.

/laking a web request with aiohttp

We'll first need to install the aiohttp library. We can do this using `pip` by running the following:

```
pip install -Iv aiohttp==3.8.1
```

This will install the latest version of aiohttp (3.8.1 at the time of this writing). Once this is complete, you're ready to start making requests.

aiohttp, and web requests in general, employ the concept of a *session*. Think of a session as opening a new browser window. Within a new browser window, you'll make connections to any number of web pages, which may send you cookies that your browser saves for you. With a session, you'll keep many connections open, which can then be recycled. This is known as connection pooling. *Connection pooling* is an important concept that aids the performance of our aiohttp-based applications. Since creating connections is resource intensive, creating a reusable pool of them cuts down on resource allocation costs. A session will also internally save any cookies that we receive, although this functionality can be turned off if desired.

Typically, we want to take advantage of connection pooling, so most aiohttp-based applications run one session for the entire application. This session object is then passed to methods where needed. A session object has methods on it for making any number of web requests, such as GET, PUT, and POST. We can create a session by using `async` `with` syntax and the `aiohttp.ClientSession` asynchronous context manager.

Listing 4.2 Making an aiohttp web request

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed
```

```
@async_timed()
async def fetch_status(session: ClientSession, url: str) ->
    async with session.get(url) as result:
        return result.status


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https:/ / www .example .com'
        status = await fetch_status(session, url)
        print(f'Status for {url} was {status}')


asyncio.run(main())
```

When we run this, we should see that the output `Status for http:/ / www .example .com was 200`. In the preceding listing, we first created a client session in an `async with` block with `aiohttp.ClientSession().` Once we have a client session, we're free to make any web request desired. In this case, we define a convenience method `fetch_status_code` that will take in a session and a URL and return the status code for the given URL. In this function, we have another `async with` block and use the session to run a `GET HTTP` request against the URL. This will give us a result, which we can then process within the `with` block. In this case, we just grab the status code and return.

Note that a `ClientSession` will create a default maximum of 100 connections by default, providing an implicit upper limit to the number of concurrent requests we can make. To change this limit, we can create an instance of an aiohttp `TCPConnector` specifying the maximum number of connections and passing that to the `ClientSession`. To learn more about this, review the aiohttp documentation at https:// docs.aiohttp.org/en/stable/client_advanced.html#connectors.

We'll reuse `fetch_status` throughout the chapter, so let's make this function reusable. We'll create a Python module named `chapter_04` with its `__init__.py` containing this function. We'll then import this in future examples in this chapter as `from` `chapter_04` import `fetch_status`.

A note for Windows users

At the present time, an issue exists with aiohttp on Windows, where you may see errors like `RuntimeError:` `Event` `loop` `is` `closed` even though your application works fine. Read more about this issue at https://github.com/aio-libs/aiohttp/issues/4324 and https://bugs.python.org/issue39232. To work around this issue, you can either manually manage the event loop as shown in chapter 2 with `asyncio.get_event_ loop().run_until_complete(main())`, or you can change the event loop policy to the Windows selector event loop policy by calling `asyncio.set_event_loop_policy (asyncio.WindowsSelectorEventLoopPolicy())` before `asyncio.run(main()).`

etting timeouts with aiohttp

Earlier we saw how we could specify a timeout for an awaitable by using `asyncio.wait_ for`. This will also work for setting timeouts for an aiohttp request, but a cleaner way to set timeouts is to use the functionality that aiohttp provides out of the box.

By default, aiohttp has a timeout of five minutes, which means that no single operation should take longer than that. This is a long timeout, and many application developers may wish to set this lower. We can specify a timeout at either the session level, which will apply that timeout for every operation, or at the request level, which provides more granular control.

We can specify timeouts using the aiohttp-specific `ClientTimeout` *data structure.* This structure not only allows us to specify a total timeout in seconds for an entire request but also allows us to set timeouts on establishing a connection or reading data. Let’s examine how to use this by specifying a timeout for our session and one for an individual request.

Listing 4.3 Setting timeouts with aiohttp

```
import asyncio
import aiohttp
from aiohttp import ClientSession

async def fetch_status(session: ClientSession,
                       url: str) -> int:
    ten_millis = aiohttp.ClientTimeout(total=.01)
    async with session.get(url, timeout=ten_millis) as resul
        return result.status



async def main():
    session_timeout = aiohttp.ClientTimeout(total=1, connect
    async with aiohttp.ClientSession(timeout=session_timeout
        await fetch_status(session, 'https:/ / example .com'

asyncio.run(main())
```

In the preceding listing, we set two timeouts. The first timeout is at the client-session level. Here we set a total timeout of 1 second and explicitly set a connection timeout of 100 milliseconds. Then, in `fetch_status` we override this for our `get` request to set a total timeout of 10 miliseconds. In this instance, if our request to example.com takes more than 10 milliseconds, an `asyncio.TimeoutError` will be raised when we `await` `fetch_status`. In this example, 10 milliseconds should be enough time for the request to example.com to complete, so we're not likely to see an exception. If you'd like to check out this exception, change the URL to a page that takes a bit longer than 10 milliseconds to download.

These examples show us the basics of aiohttp. However, our application's performance won't benefit from running only a single request with asyncio. We'll start to see the real benefits when we run several web requests concurrently.

nning tasks concurrently, revisited

In the first few chapters of this book, we learned how to create multiple tasks to run coroutines concurrently. To do this, we used `asyncio.create_task` and then awaited the task as below:

```
import asyncio

async def main() -> None:
    task_one = asyncio.create_task(delay(1))
    task_two = asyncio.create_task(delay(2))

    await task_one
    await task_two
```

This works for simple cases like the previous one in which we have one or two coroutines we want to launch concurrently. However, in a world where we may make hundreds, thousands, or even more web requests concurrently, this style would become verbose and messy.

We may be tempted to utilize a `for` loop or a list comprehension to make this a little smoother, as demonstrated in the following listing. However, this approach can cause issues if not written correctly.

Listing 4.4 Using tasks with a list comprehension incorrectly

```
import asyncio
from util import async_timed, delay


@async_timed()
async def main() -> None:
    delay_times = [3, 3, 3]
    [await asyncio.create_task(delay(seconds)) for seconds i

asyncio.run(main())
```

Given that we ideally want the `delay` tasks to run concurrently, we’d expect the main method to complete in about 3 seconds. However, in this case 9 seconds elapse to run, since everything is done sequentially:

```
starting <function main at 0x10f14a550> with args () {}
starting <function delay at 0x10f7684c0> with args (3,) {}
sleeping for 3 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10f7684c0> in 3.0008 second(s)
starting <function delay at 0x10f7684c0> with args (3,) {}
sleeping for 3 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10f7684c0> in 3.0009 second(s)
starting <function delay at 0x10f7684c0> with args (3,) {}
sleeping for 3 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10f7684c0> in 3.0020 second(s)
finished <function main at 0x10f14a550> in 9.0044 second(s)
```

The problem here is subtle. It occurs because we use `await` as soon as we create the task. This means that we pause the list comprehension and the main coroutine for every `delay` task we create until that `delay` task completes. In this case, we will have only one task running at any given time, instead of running multiple tasks concurrently. The fix is easy, although a bit verbose. We can create the tasks in one list comprehension and `await` in a second. This lets everything to run concurrently.

Listing 4.5 Using tasks concurrently with a list comprehension

```
import asyncio
from util import async_timed, delay

@async_timed()
async def main() -> None:
    delay_times = [3, 3, 3]
    tasks = [asyncio.create_task(delay(seconds)) for seconds
    [await task for task in tasks]

asyncio.run(main())
```

This code creates a number of tasks all at once in the `tasks` list. Once we have created all the tasks, we await their completion in a separate list comprehension. This works because `create_task` returns instantly, and we don't do any awaiting until all the tasks have been created. This ensures that it only requires at most the maximum pause in `delay_times`, giving a runtime of about 3 seconds:

```
starting <function main at 0x10d4e1550> with args () {}
starting <function delay at 0x10daff4c0> with args (3,) {}
sleeping for 3 second(s)
starting <function delay at 0x10daff4c0> with args (3,) {}
sleeping for 3 second(s)
starting <function delay at 0x10daff4c0> with args (3,) {}
sleeping for 3 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10daff4c0> in 3.0029 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10daff4c0> in 3.0029 second(s)
finished sleeping for 3 second(s)
finished <function delay at 0x10daff4c0> in 3.0029 second(s)
finished <function main at 0x10d4e1550> in 3.0031 second(s)
```

While this does what we want, drawbacks remain. The first is that this consists of multiple lines of code, where we must explicitly remember to separate out our task creation from our awaits. The second is that it is inflexible, and if one of our coroutines finishes long before the others, we'll be trapped in the second list comprehension waiting for all other coroutines to finish. While this may be acceptable in certain circumstances, we may want to be more responsive, processing our results as soon as they arrive. The third, and potentially biggest issue, is exception handling. If one of our coroutines has an exception, it will be thrown when we `await` the failed task. This means that we won't be able to process any tasks that completed successfully because that one exception will halt our execution.

asyncio has convenience functions to deal with all these situations and more. These functions are recommended when running multiple tasks concurrently. In the following sections, we'll look at some of them, and examine how to use them in the context of making multiple web requests concurrently.

nning requests concurrently with gather

A widely used asyncio API functions for running awaitables concurrently is `asyncio .gather`. This function takes in a sequence of awaitables and lets us run them concurrently, all in one line of code. If any of the awaitables we pass in is a coroutine, `gather` will automatically wrap it in a task to ensure that it runs concurrently. This means that we don't have to wrap everything with `asyncio.create_task` separately as we used above.

`asyncio.gather` returns an awaitable. When we use it in an `await` expression, it will pause until all awaitables that we passed into it are complete. Once everything we passed in finishes, `asyncio.gather` will return a list of the completed results.

We can use this function to run as many web requests as we'd like concurrently. To illustrate this, let's see an example where we make 1,000 requests at the same time and grab the status code of each response. We'll decorate our main coroutine with `@async_ timed` so we know how long things are taking.

Listing 4.6 Running requests concurrently with `gather`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from chapter_04 import fetch_status
from util import async_timed


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https:/ / example .com' for _ in range(1000
        requests = [fetch_status(session, url) for url in ur
        status_codes = await asyncio.gather(*requests)
        print(status_codes)


asyncio.run(main())
```

(1) Generate a list of coroutines for each request we want to make.

❷ Wait for all requests to complete.

In the preceding listing, we first generate a list of URLs we'd like to retrieve the status code from; for simplicity, we'll request example.com repeatedly. We then take that list of URLs and call `fetch_status_code` to generate a list of coroutines that we then pass into `gather`. This will wrap each coroutine in a task and start running them concurrently. When we execute this code, we'll see 1,000 messages printed to standard out, saying that the `fetch_status_code` coroutines started sequentially, indicating that 1,000 requests started concurrently. As results come in, we'll see messages like `finished <function fetch_status_code at 0x10f3fe3a0> in 0.5453 second(s)` arrive. Once we retrieve the contents of all the URLs we've requested, we'll see the status codes start to print out. This process is quick, depending on the internet connection and speed of the machine, and this script can finish in as little as 500-600 milliseconds.

So how does this compare with doing things synchronously? It's easy to adapt the main function so that it blocks on each request by using an `await` when we call `fetch_status_code`. This will pause the main coroutine for each URL, effectively making things synchronous:

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https:/ / example .com' for _ in range(1000
        status_codes = [await fetch_status_code(session, url
        print(status_codes)
```

If we run this, notice that things will take much longer. We'll also notice that, instead of getting 1,000 `starting function fetch_status_code` messages followed by 1,000 `finished function fetch_status_code` messages, something like the following displays for each request:

```
starting <function fetch_status_code at 0x10d95b310>
finished <function fetch_status_code at 0x10d95b310> in 0.01
```

This indicates that requests occur one after another, waiting for each call to `fetch_status_code` to finish before moving on to the next request. So how much slower is this than using our async version? While this depends on your internet connection and the machine you run this on, running sequentially can take around 18 seconds to complete. Comparing this with our asynchronous version, which took around 600 milliseconds, the latter runs an impressive 33 times faster.

It is worth noting that the results for each awaitable we pass in may not complete in a deterministic order. For example, if we pass coroutines `a` and `b` to `gather` in that order, `b` may complete before `a`. A nice feature of `gather` is that, regardless of when our awaitables complete, we are guaranteed the results will be returned in the order we passed them in. Let's demonstrate this by looking at the scenario we just described with our `delay` function.

Listing 4.7 Awaitables finishing out of order

```
import asyncio
from util import delay
```

```
async def main():
    results = await asyncio.gather(delay(3), delay(1))
    print(results)

asyncio.run(main())
```

In the preceding listing, we pass two coroutines to `gather`. The first takes 3 seconds to complete and the second takes 1 second. We may expect the result of this to be `[1, 3]`, since our 1-second coroutine finishes before our 3-second coroutine, but the result is actually `[3, 1]`—the order we passed things in. The `gather` function keeps result ordering deterministic despite the inherent nondeterminism behind the scenes. In the background, `gather` uses a special kind of `future` implementation to do this. For the curious reader, reviewing the source code of `gather` can be an instructive way to understand how many asyncio APIs are built using futures.

In the examples above, it's assumed none of the requests will fail or throw an exception. This works well for the "happy path," but what happens when a request fails?

Iandling exceptions with gather

Of course, when we make a web request, we might not always get a value back; we might get an exception. Since networks can be unreliable, different failure cases are possible. For example, we could pass in an address that is invalid or has become invalid because the site has been taken down. The server we connect to could also close or refuse our connection.

`asyncio.gather` gives us an optional parameter, `return_exceptions`, which allows us to specify how we want to deal with exceptions from our awaitables. `return_exceptions` is a Boolean value; therefore, it has two behaviors that we can choose from:

- `return_exceptions=False`—This is the default value for `gather`. In this case, if any of our coroutines throws an exception, our `gather` call will also throw that exception when we `await` it. However, even though one of our coroutines failed, our other coroutines are not canceled and will continue to run as long as we handle the exception, or the exception does not result in the event loop stopping and canceling the tasks.
- `return_exceptions=True`—In this case, `gather` will return any exceptions as part of the result list it returns when we `await` it. The call to `gather` will not throw any exceptions itself, and we'll be able handle all exceptions as we wish.

To illustrate how these options work, let's change our URL list to contain an invalid web address. This will cause aiohttp to raise an exception when we attempt to make the request. We'll then pass that into `gather` and see how each of these `return_exceptions` behaves:

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https:/ / example .com', 'python:/ / exampl
        tasks = [fetch_status_code(session, url) for url in
        status_codes = await asyncio.gather(*tasks)
        print(status_codes)
```

If we change our URL list to the above, the request for `'python:// example .com'` will fail because that URL is not valid. Our `fetch_status_code` coroutine will throw an `AssertionError` because of this, meaning that `python:/ /` does not translate into a port. This exception will get thrown when we `await` our `gather` coroutine. If we run this and look at the output, we'll see that our exception was thrown, but we'll also see that our other request continued to run (we've removed the verbose traceback for brevity):

```
starting <function main at 0x107f4a4c0> with args () {}
starting <function fetch_status_code at 0x107f4a3a0>
starting <function fetch_status_code at 0x107f4a3a0>
finished <function fetch_status_code at 0x107f4a3a0> in 0.00
finished <function main at 0x107f4a4c0> in 0.0203 second(s)
finished <function fetch_status_code at 0x107f4a3a0> in 0.01
Traceback (most recent call last):
  File "gather_exception.py", line 22, in <module>
    asyncio.run(main())
AssertionError

Process finished with exit code 1
```

`asyncio.gather` won't cancel any other tasks that are running if there is a failure. That may be acceptable for many use cases but is one of the drawbacks of `gather`. We'll see how to cancel tasks we run concurrently later in this chapter.

Another potential issue with the above code is that if more than one exception happens, we'll only see the first one that occurred when we `await` the `gather`. We can fix this by using `return_exceptions=True`, which will return all exceptions we encounter when running our coroutines. We can then filter out any exceptions and handle them as needed. Let’s examine our previous example with invalid URLs to understand how this works:

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https:/ / example .com', 'python:/ / exampl
        tasks = [fetch_status_code(session, url) for url in
        results = await asyncio.gather(*tasks, return_except

        exceptions = [res for res in results if isinstance(r
        successful_results = [res for res in results if not

        print(f'All results: {results}')
        print(f'Finished successfully: {successful_results}'
        print(f'Threw exceptions: {exceptions}')
```

When running this, notice that no exceptions are thrown, and we get all the exceptions alongside our successful results in the list that `gather` returns. We then filter out anything that is an instance of an exception to retrieve the list of successful responses, resulting in the following output:

```
All results: [200, AssertionError()]
Finished successfully: [200]
Threw exceptions: [AssertionError()]
```

This solves the issue of not being able to see all the exceptions that our coroutines throw. It is also nice that now we don’t need to explicitly handle any exceptions with a `try` `catch` block, since we no longer throw an exception when we `await`. It is still a little clunky that we must filter out exceptions from successful results, but the API is not perfect.

`gather` has a few drawbacks. The first, which was already mentioned, is that it isn't easy to cancel our tasks if one throws an exception. Imagine a case in which we're making requests to the same server, and if one request fails, all others will as well, such as reaching a rate limit. In this case, we may want to cancel requests to free up resources, which isn't very easy to do because our coroutines are wrapped in tasks in the background.

The second is that we must wait for all our coroutines to finish before we can process our results. If we want to deal with results as soon as they complete, this poses a problem. For example, if we have one request needing 100 milliseconds, but another that lasts 20 seconds, we'll be stuck waiting for 20 seconds before we can process the request that completed in only 100 milliseconds.

asyncio provides APIs that allow us to solve for both issues. Let's start by looking at the problem of handling results as soon as they come in.

ocessing requests as they complete

While `asyncio.gather` will work for many cases, it has the drawback that it waits for all awaitables to finish before allowing access to any results. This is a problem if we'd like to process results as soon as they come in. It can also be a problem if we have a few awaitables that could complete quickly and a few which could take some time, since `gather` waits for everything to finish. This can cause our application to become unresponsive; imagine a user makes 100 requests and two of them are slow, but the rest complete quickly. It would be great if once requests start to finish, we could output some information to our users.

To handle this case, asyncio exposes an API function named `as_completed`. This method takes a list of awaitables and returns an iterator of futures. We can then iterate over these futures, awaiting each one. When the `await` expression completes, we will retrieve the result of the coroutine that finished first out of all our awaitables. This means that we'll be able to process results as soon as they are available, but there is now no deterministic ordering of results, since we have no guarantees as to which requests will complete first.

To show how this works, let's simulate a case where one request completes quickly, and another needs more time. We'll add a `delay` parameter to our `fetch_status` function and call `asyncio.sleep` to simulate a long request, as follows:

```
async def fetch_status(session: ClientSession,
                       url: str,
                       delay: int = 0) -> int:
    await asyncio.sleep(delay)
    async with session.get(url) as result:
        return result.status
```

We'll then use a `for` loop to iterate over the iterator returned from `as_completed`.

Listing 4.8 Using `as_completed`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status
```

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https:/ / www.exa
                    fetch_status(session, 'https:/ / www.exa
                    fetch_status(session, 'https:/ / www.exa

        for finished_task in asyncio.as_completed(fetchers):
            print(await finished_task)


asyncio.run(main())
```

In the preceding listing, we create three coroutines—two that require about 1 second to complete and one that will take 10 seconds. We then pass these into `as_completed`. Under the hood, each coroutine is wrapped in a task and starts running concurrently. The routine instantly returned an iterator that starts to loop over. When we enter the `for` loop, we hit `await` `finished_task`. Here we pause execution and wait for our first result to come in. In this case, our first result comes in after 1 second, and we print the status code. Then we reach `the` `await` `result` again, and since our requests ran concurrently, we should see the second result almost instantly. Finally, our 10-second request will complete, and our loop will finish. Executing this will give us output as follows:

```
starting <function fetch_status at 0x10dbed4c0>
starting <function fetch_status at 0x10dbed4c0>
starting <function fetch_status at 0x10dbed4c0>
```

```
finished <function fetch_status at 0x10dbed4c0> in 1.1269 se
200
finished <function fetch_status at 0x10dbed4c0> in 1.1294 se
200
finished <function fetch_status at 0x10dbed4c0> in 10.0345 s
200
finished <function main at 0x10dbed5e0> in 10.0353 second(s)
```

In total, iterating over `result_iterator` still takes about 10 seconds, as it would have if we used `asynio.gather`; however, we're able to execute code to print the result of our first request as soon as it finishes. This gives us extra time to process the result of our first successfully finished coroutine while others are still waiting to finish, making our application more responsive when our tasks complete.

This function also offers better control over exception handling. When a task throws an exception, we'll be able to process it when it happens, as the exception is thrown when we `await` the `future`.

imeouts with as_completed

Any web-based request runs the risk of taking a long time. A server could be under a heavy resource load, or we could have a poor network connection. Earlier, we saw how to add timeouts for a particular request, but what if we wanted to have a timeout for a group of requests? The `as_completed` function supports this use case by supplying an optional timeout parameter, which lets us specify a timeout in seconds. This will keep track of how long the `as_completed` call has taken; if it takes longer than the timeout, each awaitable in the iterator will throw a `TimeoutException` when we `await` it.

To illustrate this, let’s take our previous example and create two requests that take 10 seconds to complete and one request that takes 1 second. Then, we’ll set a timeout of 2 seconds on `as_completed`. Once we’re done with the loop, we’ll print out all the tasks we have that are currently running.

Listing 4.9 Setting a timeout on `as_completed`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https:/ / example
                    fetch_status(session, 'https:/ / example
                    fetch_status(session, 'https:/ / example

        for done_task in asyncio.as_completed(fetchers, time
            try:
                result = await done_task
                print(result)
            except asyncio.TimeoutError:
                print('We got a timeout error!')

        for task in asyncio.tasks.all_tasks():
            print(task)
```