Python Concurrency with asyncio

Matthew Fowler

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Python Concurrency with asyncio
Version 8

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *Python Concurrency with asyncio*. This book is written for Python developers who are looking to increase the performance and throughput of their applications by utilizing concurrency.

To get the most out of this book, you should have an intermediate understanding of Python programming and be able to utilize functions, classes, lists and dictionaries. You should also be familiar with the basics of web requests and databases. Experience with concurrency concepts such as multiprocessing and multithreading, while helpful, is not necessary.

Asycnio was first released in 2014 as a library to write concurrent code. While it has been around for six years, it has gone through a few major changes since it was first released, going from being implemented with generators and decorators to a more mature part of Python with async and await keywords built into the language.

Resources to truly understand and utilize the library in practice are few and far in between and are mostly scattered in various blogs across the web. My goal is to fill this gap and produce a book with cohesive depth and breadth on what asyncio is and what it can do.

By the end of this book, you should grok how the asyncio event loop uses non-blocking sockets to achieve concurrency with only one thread. You'll be able to use asyncio to build command-line applications that can handle multiple concurrent users. You'll be able to use asyncio APIs to run multiple web requests or database queries concurrently as well as build async web servers. We'll also cover how to use both multithreading and multiprocessing with asyncio to improve the performance of blocking I/O as well as CPU bound data processing tasks.

Your feedback is essential to making this book the best it can be. As you see things that are missing, aren't working for you, or are just confusing don't hesitate to leave a comment in the liveBook discussion forum. My goal with this book is to take a topic which is widely viewed as confusing and make it easy to understand and use for any Python developer. Your feedback will make sure that we achieve this goal!

- Matt Fowler

brief contents

1

Getting to know asyncio

This chapter covers

- **What asyncio is and the benefits it provides**
- **Concurrency, parallelism, threads and processes**
- **The global interpreter lock and the challenges it poses to concurrency**
- **How non-blocking sockets can achieve concurrency with only one thread**
- **The basics around how event-loop based concurrency works**

Many applications especially in today's world of web applications rely heavily on I/O operations. These types of operations include downloading the contents of a webpage from the internet, communicating over a network with a group of microservices, or running several queries together against a database such as MySQL or Postgres. A web request or communication with a microservice may take hundreds of milliseconds, or even seconds if the network is slow. A database query could be expensive especially if that database is under high load or the query is complex. A web server may need to handle hundreds if not thousands of requests at the same time. These operations can be expensive in terms of time.

Making many of these I/O requests at the same time can lead to substantial performance issues. If we run these requests one after another as we would in a sequential, non-concurrent application, we'll see a compounding performance impact. As an example, if we're writing an application that needs to download one hundred web pages, or run one hundred queries, each of which takes a second to execute, our application will take at least one hundred seconds to run. However, if we were to exploit concurrency and start the downloads at the same time, and wait at the same time, we could in theory complete these operations in as little as one second.

Asyncio was first introduced in Python 3.4 as a way to handle these highly concurrent workloads. Properly utilizing this library can lead to drastic performance and resource utilization improvements for applications which use I/O operations as it allows us to start many of these long-running tasks at the same time.

In this chapter we'll introduce the basics of concurrency in order to better understand how we can achieve it with Python and the asyncio library. We'll explore the differences between CPU bound work and I/O bound work to know which concurrency model best suits our specific needs. We'll also learn about the basics of processes and threads and learn about the unique challenges to concurrency in Python caused by its global interpreter lock. Finally, we'll get an understanding of how we can utilize a concept called non-blocking I/O with an event loop to achieve concurrency using only one Python process and thread. This is the primary concurrency model of asyncio.

1.1 What is asyncio?

In a synchronous application, code runs sequentially. The next line of code runs as soon as the previous one has finished, and only one thing is happening at once. This model works fine for many, if not most applications. However, what if one line of code was especially slow? In that case, all other code after our slow line would be stuck waiting for that line to complete. These potentially slow lines of code are called *blocking* because they block the application from running any other code. Many of us have seen this before in buggy user interfaces, we happily click around until the application freezes, leaving us with a spinner or an unresponsive user interface. This is an example of an application being blocked leading to a poor user experience.

While any operation can be blocking if it takes long enough, many applications block on I/O, short for input/output. I/O refers to a computer's input and output devices such as a keyboard, hard drive or most commonly a network card. These are operations such as waiting for user input or retrieving the contents from a web-based API. In a synchronous application, we'll be stuck waiting for those operations to complete until we can run anything else. This can cause performance and responsiveness issues as we can only have one long operation running at any given time, and that operation will stop our application from doing anything else.

One solution to this issue is to introduce concurrency. In the simplest terms, concurrency means allowing more than one task to be handled at the same time. In the case of concurrent I/O, a few examples are allowing multiple web requests to be made at the same time or allowing simultaneous connections to a web server.

There are several ways to achieve this concurrency in Python. One of the more recent additions to the Python ecosystem is the asyncio library. Asyncio is short for asynchronous I/O and is a Python library that allows us to run code using an asynchronous programming model. This lets us handle multiple I/O operations at the same time, while still allowing our application to remain responsive.

So, what is asynchronous programming? Asynchronous programming means that a particular long-running task can be run in the background separate from the main application. Instead of blocking all other application code waiting for that long-running task to be done, we're free to do other work that is not dependent on that task. Then, once the long-running task is completed, we'll be notified that it is done so we can process the result.

In Python version 3.4, asyncio was first introduced with decorators alongside generator `yield from` syntax to define coroutines. A coroutine is a method that can be paused when we have a long-running task and then resumed when that task is finished. In Python version 3.5, the language got first-class support for coroutines and asynchronous programming when the keywords `async` and `await` were explicitly added to the language. This syntax, common in other programming languages such as C# and Javascript, allows us to make our asynchronous code look like it is run synchronously. This makes asynchronous code easy to read and understand as it looks like the sequential flow most software engineers are familiar with. Asyncio is a library to execute these coroutines in an asynchronous fashion using a concurrency model known as a single-threaded event loop.

While the name of asyncio may make us think that this library is only good for I/O operations, it has functionality to handle other types of operations as well by interoperating with multithreading and multiprocessing. With this interoperability we can use `async` and `await` syntax with threads and processes making these workflows easier to understand. This means this library is not only good for I/O based concurrency but can also be used with code that is CPU intensive.

In order to better understand what type of workloads asyncio can help us with, and which concurrency model is best for each type of concurrency, let's explore the differences between I/O and CPU bound operations.

1.2 What is I/O bound and what is CPU bound?

When we refer to an operation as I/O bound, or CPU bound, we are referring to the limiting factor that prevents that operation from running faster. This means that if we increased the performance of what the operation was bound on, that operation would complete in less time.

In the case of a CPU bound operation, it would complete faster if our CPU was more powerful, for instance by increasing its clock speed from 2 GHZ to 3 GHZ. In the case of an I/O bound operation it would get faster if our I/O devices could handle more data in less time. This could be achieved by increasing our network bandwidth through our ISP or upgrading to a faster network card.

CPU bound operations are typically computations and processing code in the Python world. An example of this is computing the digits of pi or looping over the contents of a dictionary, applying business logic. In an I/O bound operation we spend most of our time waiting on a network or other I/O device. An example of an I/O bound operation would be making a request to a web server or reading a file from our machine's hard drive.

Listing 1.1 I/O bound and CPU bound operations

```
import requests

response = requests.get('https://www.example.com')    #A

headers = [f'{key}: {response.headers[key]}' for key in response.headers]   #B

formatted_headers = '\n'.join(headers)    #C

with open('headers.txt', 'w') as file:
    file.write(formatted_headers)    #D
```

#A I/O bound web request
#B CPU bound response processing

#C CPU bound string concatenation
#D I/O bound write to disk

I/O bound and CPU bound operations usually live side by side with one another. Listing 1.1 shows us an example of this. We first make an I/O bound request to download the contents of https://www.example.com. Once we have the response, we perform a CPU bound loop to format the headers of the response and turn them into a string separated by newlines. We then open a file and write the string to that file, both I/O bound operations.

Asynchronous I/O allows us to "pause" execution of a particular method when we have an I/O operation and wait for that I/O to complete. While we have "paused" that method, we can run other code while we wait for our initial I/O to complete in the background. This allows us to execute many I/O operations concurrently, potentially speeding up our application.

1.3 Understanding Concurrency, Parallelism and Multitasking

To better understand how concurrency can help our applications perform better, it is first important to learn and fully understand the terminology of concurrent programming. In particular we'll learn more about what concurrency means and how asyncio uses a concept called multitasking to achieve it.

1.3.1 What are concurrency and parallelism?

Concurrency

When we say two tasks are happening concurrently, we mean those tasks are happening at the same time. Take for instance baking a cake. In order to bake a cake, we need to preheat our oven. Preheating can take tens of minutes depending on our oven and baking temperature, but we don't need to wait for our oven to preheat before starting other tasks, such as mixing our flour and sugar together with eggs. If we turn our oven on to preheat and then start making our batter, we're behaving concurrently.

Parallelism

While concurrency merely implies that multiple tasks are happening at the same time it does not imply that they are running at the same time. When we say something is running in parallel, we mean not only are there two or more tasks happening concurrently, but they are executing at the same time. Going back to our cake baking example, imagine we were instead baking two cakes with different recipes instead of one. Given the two recipes are different we can only make one batter at a time by ourselves. However, if we had another person to help us out, we could work on the first batter while they started working on the second. Two people making batter at the same time is parallel because we have two tasks running at the same time.

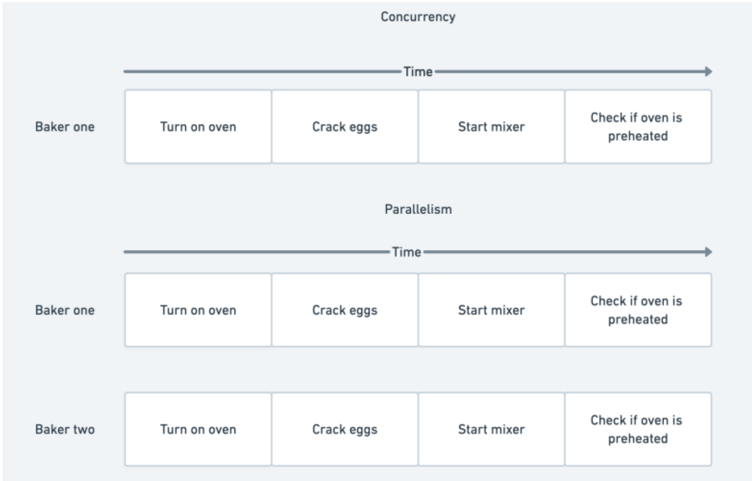


Figure 1.1 In concurrency we have multiple tasks happening at the same time, but only one we're actively doing. In parallelism we have both multiple tasks happening and we are actively doing more than one at the same time

The concepts of concurrency and parallelism are similar and slightly confusing to differentiate between, but it is important for us to understand what makes them distinct from one another.

1.3.2 The difference between concurrency and parallelism

Concurrency is about having multiple tasks that can happen independently. We can have concurrency on a CPU with only one core as the operation will utilize preemptive multitasking (defined in the next section) to switch between tasks. Parallelism however means that we must be executing two tasks at the same time. On a machine with one core, this is not possible. In order to make this possible we need a CPU with multiple cores that can run two things at the same time.

While parallelism implies concurrency, concurrency does not always imply parallelism. A multithreaded application running on a multiple core machine is both concurrent and parallel. In this set up we both have multiple tasks running at the same time and there are two cores independently executing the code associated with those tasks. However, with multitasking we can have multiple tasks happening at the same time yet only one of them is executing at a given time.

1.3.3 What is multitasking?

Multitasking is everywhere in today's world. We multitask while making breakfast, taking a call or answering a text while we wait for our water to boil. We even multitask while commuting to work, reading a book while the train takes us to our office. There are two main kinds of multitasking, preemptive multitasking and cooperative multitasking.

Preemptive multitasking

In this model we let the operating system decide how to switch between which work is currently being executed via a process called time slicing. When the operating system switches between work it is known as preempting. How this mechanism works under the hood is up to the operating system itself. This is primarily achieved through either using multiple threads or multiple processes.

Cooperative multitasking

In this model instead of relying on the operating system to decide when to switch between which work is currently being executed, we explicitly code points in our application where we can let other tasks run. The tasks in our application operate in a model where they "cooperate", explicitly saying "I'm pausing my task for a while, go ahead and run other tasks".

1.3.4 The benefits of cooperative multitasking

Asyncio uses cooperative multitasking to achieve concurrency. When our application reaches a point where it could wait a while for a result to come back, we explicitly mark this in code. This allows other code to run while we wait for the result to come back in the background. Once the task we marked has completed we in effect "wake up" and resume executing the task. This gives us a form of concurrency because we can have multiple tasks started at the same time, but importantly not parallelism because they aren't executing any code at the same time.

Cooperative multitasking has a few benefits over preemptive multitasking. The first of which is it is less resource intensive. When an operating system needs to switch between running a thread or process it involves what is called a context switch. Context switches are fairly intensive operations because the operating system has to save information about the running process or thread to be able to reload it. A second benefit is granularity. An operating system knows that a thread or task should be paused based on whichever scheduling algorithm it uses, but that might not be the best time to pause. With cooperative multitasking we explicitly mark the areas which are the best to pause our tasks. This gives us some efficiency gains in that we are only switching tasks when we explicitly know it is the right time to do so.

Now that we understand concurrency, parallelism and multitasking we'll use these concepts to understand how to implement them in Python with threads and processes.

1.4 Understanding processes, threads, multithreading and multiprocessing

To better set us up to understand how concurrency works in the Python world, we'll first need to understand the basics about how threads and processes work. We'll then examine how to use them for multithreading and multiprocessing to do work concurrently. Let's first start with some definitions around processes and threads.

Process

A process is an application we run that has its own memory space that other applications cannot access. An example of creating a Python process would be running a simple "hello world" application or typing `python` at the command line to start up the REPL (read eval print loop). Multiple processes can run on a single machine. If we are on a machine where we have a CPU with multiple cores, we can execute multiple processes at the same time. If we are on a CPU with only one core, we can still have multiple applications running at the same time through an operating system construct called "time-slicing". When an operating system uses time slicing it will switch between which process is running automatically after some amount of time. The algorithms that determine when this switching occurs are different depending on the operating system.

Thread

Threads can be thought of as lighter-weight processes and they are the smallest construct that can be managed by an operating system. They do not have their own memory as a process does, but rather they share the memory of the process that created them. Threads are associated with the process that created them and a process will always have at least one thread associated with it, usually known as the main thread. A process can also create other threads, more commonly known as worker or background threads. These threads can perform other work concurrently alongside the main thread. Threads, much like processes can run alongside one another on a multi-core CPU and the operating system can also switch between them by utilizing time-slicing. When we run a normal Python application, we create a process as well as a main thread which will be responsible for running our Python application.

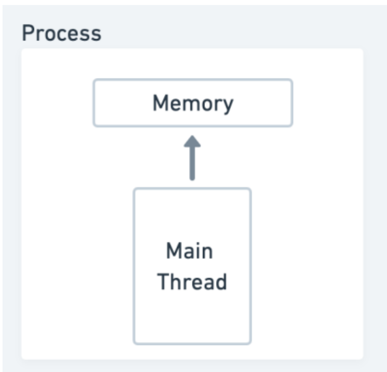


Figure 1.2 A process with one main thread reading from memory

1.2 Processes and threads in a simple Python application

```
import os
import threading

print(f'Python process running with process id: {os.getpid()}')

total_threads = threading.active_count()
thread_name = threading.current_thread().getName()

print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')
```

In listing 1.2 we create a simple application to show us the basics of the main thread. We first grab the process id and print it to prove that we indeed have a dedicated process running. We then get the active count of threads running as well as the current thread's name to show that we are running one thread, the main thread. While the process id will be different each time this code is run, running listing 1.2 will give output similar to the following.

```
Python process running with process id: 98230
Python currently running 1 thread(s)
The current thread is MainThread
```

Processes can also create other threads which share the memory of the main process. These threads can do other work concurrently for us in what is known as multithreading.

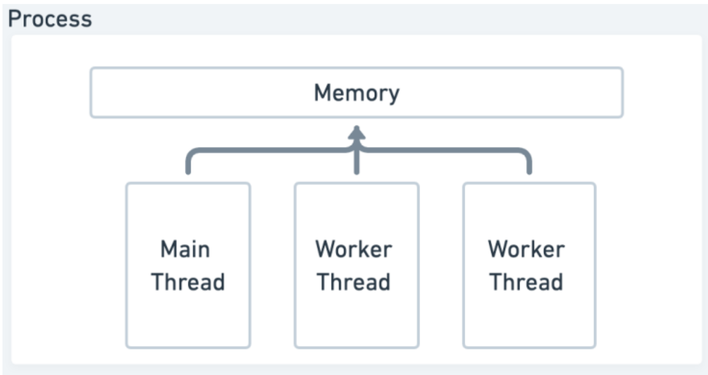


Figure 1.3 A multithreaded program with two worker threads and one main thread, each sharing the process's memory.

1.3 creating a multithreaded Python application

```
import threading


def hello_from_thread():
    thread_name = threading.current_thread()
    print(f'Hello from thread {thread_name}!')


hello_thread = threading.Thread(target=hello_from_thread)
hello_thread.start()

total_threads = threading.active_count()
thread_name = threading.current_thread().getName()

print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')

hello_thread.join()
```

In listing 1.3 we create a method to print out the name of the current thread and then create a thread to run that method. We then call the `start` method of the thread to start running it. Finally, we call the `join` method. `join` will cause the program to pause until the thread we started completed. If we run the above, we'll see output similar to the following

```
Hello from thread <Thread(Thread-1, started 123145541312512)>!
Python is currently running 2 thread(s)
The current thread is MainThread
```

Multithreaded applications are a common way to achieve concurrency in many languages. There are a few challenges in utilizing concurrency with Threads in Python, however. In particular, multithreading is only good for I/O bound work because we are limited by the global interpreter lock, which we will review in-depth in section 1.5.

Multithreading is not the only way we can achieve concurrency; we can also create multiple processes to do work concurrently for us. This is known as multiprocessing. In multiprocessing a parent process creates one or more children processes that it manages. It can then distribute work to these children processes.

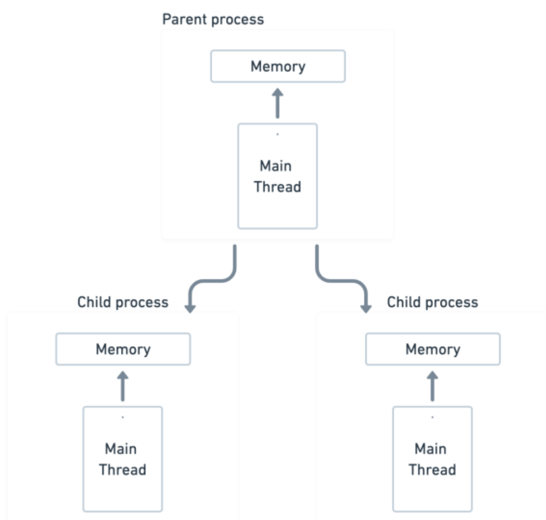


Figure 1.4 An application utilizing multiprocessing with one parent process and two children processes

Python gives us the multiprocessing module to handle this. The API is similar to that of the threading module. We first create a process with a target function. Then we call it's `start` method to execute it and finally it's `join` method to wait for it to complete running.

Creating multiple processes

```
import multiprocessing
import os


def hello_from_process():
    print(f'Hello from child process {os.getpid()}!')
```

```
if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()

    print(f'Hello from parent process {os.getpid()}')

    hello_process.join()
```

In listing 1.x we create one child process that prints its process id and we also print out the parent process id to prove that we are running different processes. Multiprocessing is typically best when we have CPU intensive work.

Multithreading and multiprocessing may seem like magic bullets to enable concurrency with Python. However, the power of these concurrency models is hindered by an implementation detail of Python, the global interpreter lock.

1.5 Understanding the Global Interpreter Lock

The global interpreter lock, abbreviated GIL and pronounced "gill" is a controversial topic in the Python community. In short, the GIL prevents any Python process from executing more than one Python bytecode instruction at any given time. This means that even if we have multiple threads on a machine with multiple cores, we can only have one thread running Python code at a time. In a world where we have CPUs with multiple cores this can pose a significant challenge for Python developers looking to take advantage of concurrency to improve the performance of their application.

So why does the GIL exist? The answer lies in how memory is managed in CPython. In CPython memory is managed primarily by a process known as reference counting. Reference counting works by keeping track of who currently needs access to a particular Python object, such as an integer, dictionary or list. A reference count is just an integer keeping track of how many places 'reference' that particular object. When someone no longer needs that referenced object, the reference count is decremented, when someone else needs it, it is incremented. When the reference count reaches zero, no one is referencing the object and it can be deleted from memory.

> **WHAT IS CPYTHON?** CPython is the reference implementation of Python. By reference implementation we mean it is the standard implementation of the language and is used as the 'reference' for proper behavior of the language. There are other implementations of Python such as Jython and IronPython.

The conflict with threads arises in that the implementation in CPython is not thread safe. When we say CPython is not thread safe we mean that if two or more threads modify a shared variable, that variable may wind up in an unexpected state. This unexpected state depends on the order with which the threads access the variable and is more commonly known as a race condition. Race conditions can arise when two threads need to reference a Python object at the same time.

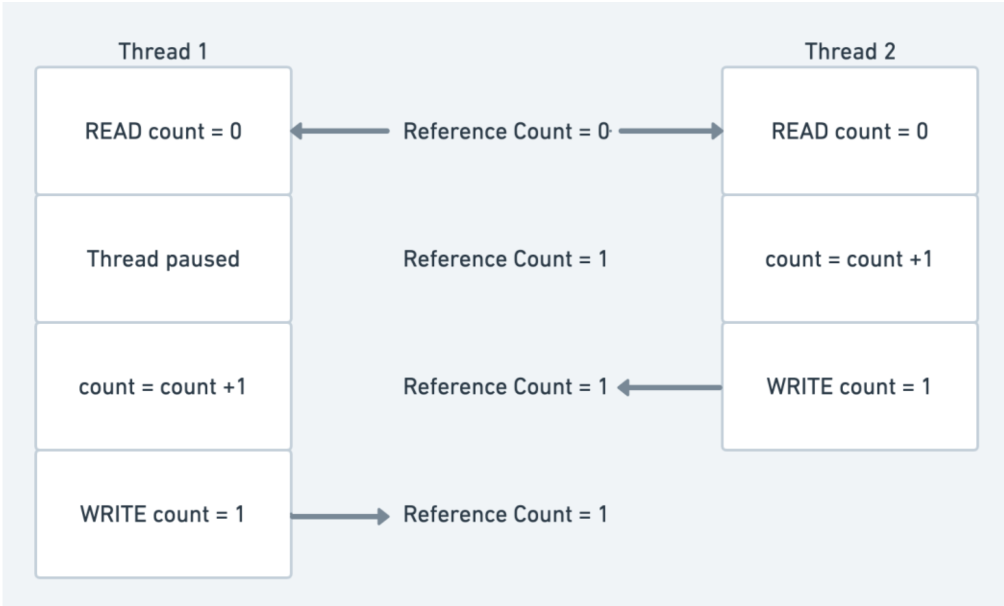


Figure 1.5 A race condition where two threads try to increment a reference count at the same time. Instead of an expected count of two, we get one.

In the event that two threads increment the reference count at the same time, we could wind up with a situation where one thread causes the reference count to be zero when the object is still in use by the other thread. The likely result of this would be an application crash when we try and read the potentially deleted memory.

To demonstrate the effect of the GIL on multithreaded programming, let's examine the CPU intensive task of computing the nth number in the Fibonacci sequence. We'll use a fairly slow implementation of the algorithm to demonstrate a time intensive operation. A proper solution would utilize memoization or mathematical techniques to improve performance.

1.4 Generating and timing the Fibonacci sequence

```
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) is {fib(number)}')
```

```
def fibs_no_threading():
    print_fib(40)
    print_fib(41)


start = time.time()

fibs_no_threading()

end = time.time()

print(f'Completed in {end - start} seconds.')
```

This implementation utilizes recursion and is overall a fairly slow algorithm, requiring exponential *O(2^N)* time to complete. If we are in a situation where we need to print two Fibonacci numbers, it is easy enough to synchronously call them and time the result as we have done in listing 1.4.

Depending on the speed of the CPU we run on, we will see different timings, but running the code in listing 1.4 will yield output similar to the following:

```
fib(40) is 63245986
fib(41) is 102334155
Completed in 65.15168905258179 seconds.
```

This is a fairly long computation, but each of our function calls to `print_fibs` are completely independent from one another. This means that they can be put in multiple threads which our CPU can, in theory, run concurrently on multiple cores, thus speeding up our application.

1.5 Multithreading the Fibonacci sequence

```
import threading
import time

def fibs_with_threads():
    fortieth_thread = threading.Thread(target=print_fib, args=(40,))
    forty_first_thread = threading.Thread(target=print_fib, args=(41,))

    fortieth_thread.start()
    forty_first_thread.start()

    fortieth_thread.join()
    forty_first_thread.join()


start_threads = time.time()

fibs_with_threads()

end_threads = time.time()

print(f'Threads took {end_threads - start_threads} seconds.')
```

In listing 1.5 we create two threads, one to compute `fib(40)` and one to compute `fib(41)` and start them concurrently by calling `start()` on each thread. Then we make a call to `join()`

which will cause our main program to wait until the threads finish. Given that we start our computation of `fib(40)` and `fib(41)` at the same time and run them concurrently, one would think we could see a reasonable speedup, however, we will see an output like the below even on a multi-core machine.

```
fib(40) is 63245986
fib(41) is 102334155
Threads took 66.10594320297241 seconds.
```

Our threaded version took almost exactly the same time. In fact, it was even a little slower! This is almost entirely due to the GIL and the overhead of creating and managing threads. While it is true the threads run concurrently, only one of them is allowed to run Python code at a time due to the lock. This leaves the other thread in a waiting state until the first one completes which completely negates the value of multiple threads.

1.5.1 Is the GIL ever released?

Based on the previous example, you may be wondering if concurrency in Python can ever happen with threads given that the GIL prevents us from running two lines of Python concurrently. The GIL is not held forever such that we can't use multiple threads to our advantage.

The global interpreter lock is released when I/O operations happen. This lets us utilize threads do concurrent work when it comes to I/O, but not for CPU bound Python code itself. To illustrate this let's take an example of reading the status code of a web page.

Listing 1.6 Synchronously reading status codes

```
import time
import requests


def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)


sync_start = time.time()

read_example()
read_example()

sync_end = time.time()

print(f'Running synchronously took {sync_end - sync_start} seconds.')
```

In listing 1.6 we retrieve the contents of example.com and print the status code twice. Depending on our network connection speed and our location we'll see output similar to the following when running this code:

```
200
200
Running synchronously took 0.23061609268188477 seconds.
```

Now that we have a baseline for what a synchronous version looks like we can write a multithreaded version to compare to. In our multithreaded version we'll create one thread for each request to example.com in an attempt to run them concurrently.

Listing 1.7 Multithreaded status code reading

```
import time
import threading
import requests


def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)


thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()

thread_1.start()
thread_2.start()

print('All threads running!')

thread_1.join()
thread_2.join()

thread_end = time.time()

print(f'Running with threads took {thread_end - thread_start} seconds.')
```

When we execute listing 1.7, we will see output similar to the following, depending again on our network connection and location:

```
All threads running!
200
200
Running with threads took 0.0977330207824707 seconds.
```

This is roughly two times faster than our original version that did not use threads since we've run the two requests at roughly the same time! Of course, depending on your internet connection and machine specs you will see different results, but the numbers should be directionally similar.

So how is it the case that we can release the GIL for I/O but not for CPU bound operations? The answer lies in the system calls that are made under the hood. In the case of I/O the low-level system calls are outside of Python runtime. This allows the GIL to be released because it is not interacting with Python objects directly. In this case the GIL is only re-acquired when the data received is translated back into a Python object. Then at the operating system level the I/O operations execute concurrently. This model gives us concurrency, but not parallelism. In other languages such as Java or C++, we would get true parallelism on multi-core machines because we don't have the GIL and can execute at the same time. However, in Python because of the GIL, the best we can do is concurrency of our I/O operations, but only one piece of Python code is executing at a given time.

1.5.2 Asyncio and the GIL

Asyncio exploits the fact that I/O operations release the GIL to give us concurrency, even with only one thread. When we utilize asyncio we create objects called *coroutines*. A coroutine can be thought of as executing a lightweight thread. Much like we can have multiple threads running at the same time, each with their own concurrent I/O operation, we can have many coroutines running alongside one another. While we are waiting for our I/O bound coroutines to finish, we can still execute other Python code, thus giving us concurrency. It is important to note that asyncio does not circumvent the GIL, we are still subject to it. If we have a CPU bound task, we still need to use multiple processes to execute it concurrently (which can be done with asyncio itself), otherwise we will cause performance issues in our application.

Now that we know it is possible to achieve concurrency for I/O with only a single thread, let's dive into the specifics of how this works with non-blocking sockets.

1.6 How single threaded concurrency works

In the previous section we introduced multiple threads as a mechanism for achieving concurrency for I/O operations. However, we don't need multiple threads to achieve this kind of concurrency, we can do it all within the confines of one process and one thread. We do this by exploiting the fact that at the system level I/O operations can be done concurrently. To better understand this, we'll need to dive into how sockets work and in particular, how non-blocking sockets work.

1.6.1 What is a socket?

A socket is a low-level abstraction for sending and receiving data over a network. It is the basis for how data is transferred to and from servers. Sockets support two main operations, sending bytes and receiving bytes. We write bytes to a socket which will then get sent to a remote address, typically some type of server. Then once we've sent those bytes, we wait for the server to write its response back to our socket. Once these bytes have been sent back to our socket, we can then read the result.

While sockets are a low-level concept, they are fairly easy to understand if you think of them as mailboxes. You can put a letter in your mailbox which your mailman then picks up and delivers to the recipient's mailbox. The receiver opens their mailbox and your letter and depending on the contents may send you a letter back. In this analogy you may think of the letter as the data or bytes we want to send. Given that the act of putting a letter into the mailbox is writing the bytes to a socket and opening the mailbox to read the letter as reading bytes from a socket. The mailman can be thought of as the transfer mechanism over the internet, routing the data to the correct address.

In the case of getting the contents from example.com as we saw earlier, we open a socket that connects to example.com's server. We then write a request to get the contents to that socket and wait for the server to reply with the result, in this case the HTML of the webpage.

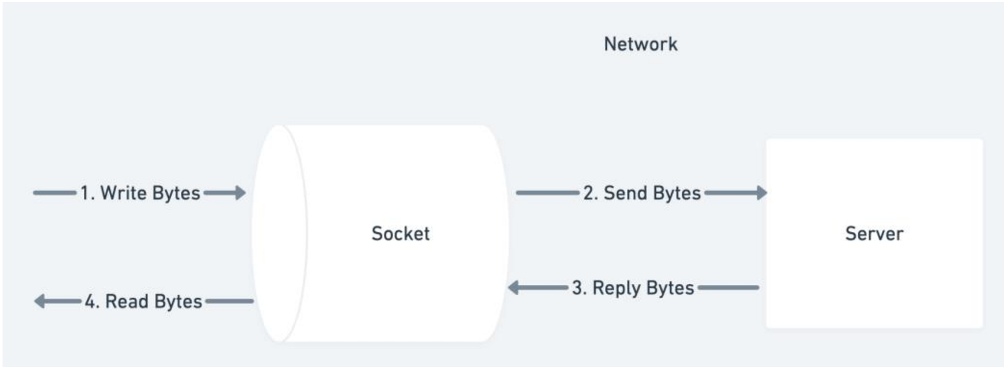


Figure 1.6 Writing bytes to a socket and reading bytes from a socket.

Sockets are *blocking* by default. What does it mean for a socket to be blocking? Simply put, this means that when we are waiting for a server to reply with data, we halt our application or *block* until we get data to read. Thus, our application stops running any other tasks until we get data from the server, an error happens, or there is a timeout.

At the operating system level, we don't need to do this blocking. Sockets can operate in *non-blocking* mode. In non-blocking mode, when we write bytes to a socket, we can just fire and forget the write or read and our application can go on to perform other tasks. Later on, we can have the operating system *tell* us that we received bytes and deal with it at that time. This lets the application do any other number of things while we wait for bytes to come back to us. Instead of blocking and waiting for data to come to us, we become more reactive, letting the operating system inform us when there is data for us to act on.

Under the hood, this is performed by a few different event notification systems depending on which operating system we're running. Asyncio is abstracted enough that it switches between the different notification systems depending on which one our operating system supports.

kqueue – FreeBSD and MacOS
epoll – Linux
IOCP (I/O completion port) - Windows

These systems will keep track of our non-blocking sockets and notify us when they are ready for us to do something with them. This notification system is the basis of how asyncio is able to achieve concurrency. In asyncio's model of concurrency we have only one thread executing Python at any given time. When we hit an I/O operation we hand that over to our operating system's event notification system to keep track of it for us. Once we have done this hand off, our Python thread is free to keep running other Python code or add more non-blocking sockets for the OS to keep track of for us. When our I/O operation finishes we "wake up" the task that was waiting for the result and then proceed to run any other Python code that came after that I/O operation.

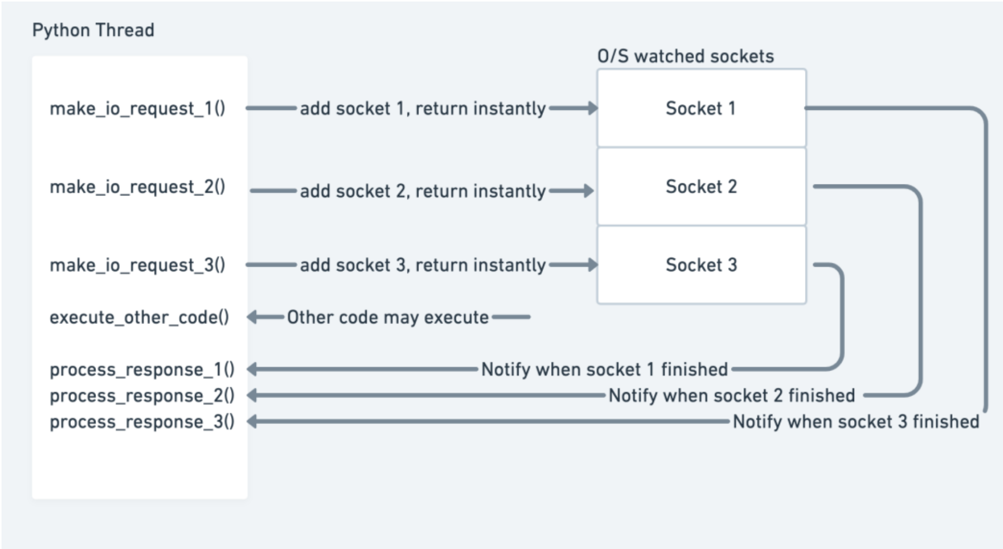


Figure 1.7 Making a non-blocking I/O request returns immediately and tells the O/S to watch sockets for data. This allows `execute_other_code()` to run right away instead of waiting for the I/O requests to finish. Later we can be alerted when I/O is complete and process the response.

But how do we keep track of which tasks are waiting for I/O as opposed to ones which can just be run because they are regular Python code? The answer lies in a construct called an event loop.

1.7 Understanding the asyncio event loop

An event loop is at the heart of every asyncio application. Event loops are a fairly common design pattern in many systems and have existed for quite some time. If you've ever used JavaScript in a browser to make an asynchronous web request, you've created a task on an event loop. Windows GUI applications behind the scenes use what are called 'message loops' as a primary mechanism for handling events such as keyboard input, while still allowing the UI to draw.

The most basic event loop is extremely simple. We create a queue that holds a list of events or messages. We then loop forever, processing messages one at a time as they come into the queue. In Python a basic event loop might look something like this:

```
from collections import deque

messages = deque()

while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

In asyncio the event loop keeps a queue of tasks instead of messages. Tasks are wrappers around a construct called a coroutine. A coroutine is a method that can pause execution when it hits a I/O bound operation and will let the event loop run other tasks that are not waiting for I/O operations to complete.

When we create an event loop, we create an empty queue of tasks. We can then put tasks into the queue to be run. Each iteration of the event loop checks for tasks that need to be run and will run them one at a time until a task hits an I/O operation. At that time the task will be 'paused' and we will instruct our operating system to watch any sockets for I/O to complete. We will then look for the next task to be run. On every iteration of the event loop we'll check to see if any of our I/O has completed, if it has, we'll 'wake up' any tasks that were paused and let them finish running.

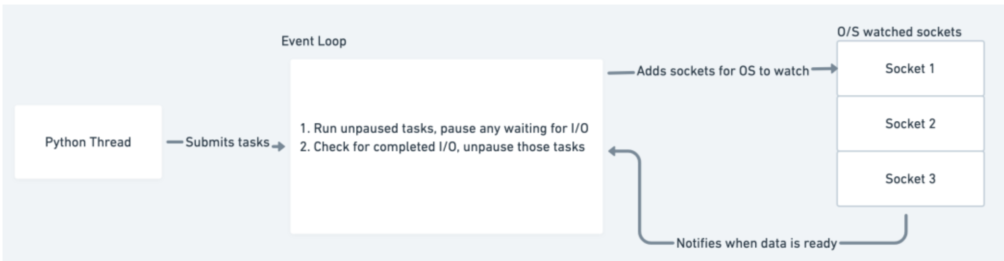


Figure 1.8 An example of a thread submitting tasks to the event loop.

To illustrate this let's imagine we have three tasks that each make an asynchronous web request. Imagine these tasks have a bit of code to do setup, which is CPU bound, then they make a web request, and they follow with some CPU bound postprocessing code. Now let's submit these tasks to the event loop at the same time. In pseudocode we would write something like this:

```
def make_request():
    cpu_bound_setup()
    io_bound_web_request()
    cpu_bound_postprocess()

task_one = make_request()
task_two = make_request()
task_three = make_request()
```

Since all three tasks start with CPU bound work and we are single threaded, only the first task starts executing code and the other two are left waiting to run. Once the CPU bound setup work is finished in task one, it hits an I/O bound operation and will pause itself to say, 'I'm waiting for I/O, any other tasks waiting to run can run'.

Once this happens, task two can begin executing. Task two starts its CPU bound code and then pauses waiting for I/O. At this time both task one and task two are concurrently waiting for their network request to complete. Since task one and two are both paused waiting for I/O, we start running task three.

Now imagine once task three pauses to wait for its I/O to complete, the web request for task one has finished. We're now alerted by our operating system's event notification system that this I/O has finished. We can now resume executing task one while both task two and task three are waiting for their I/O to finish.

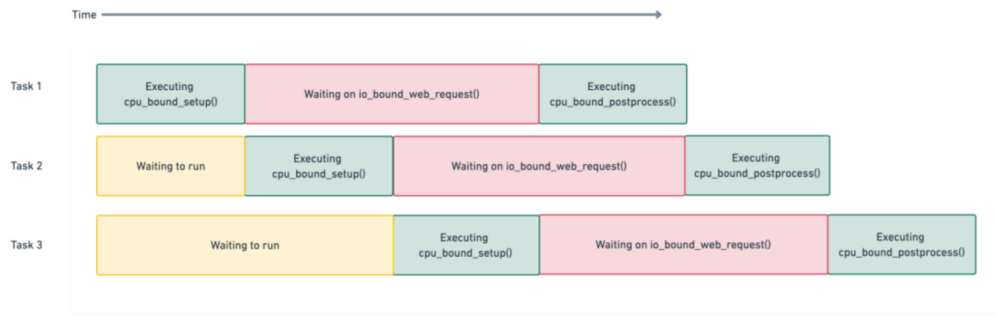


Figure 1.9 Executing multiple tasks concurrently with I/O operations.

If we look at any vertical slice of this diagram, we can see that only one CPU bound piece of work is running at any given time, however, we have up to two I/O bound operations happening concurrently. This overlapping of waiting for I/O per each task is where the real time savings of asyncio comes in.

1.8 Summary

Asyncio is a library that allows us to handle I/O and other operations asynchronously, allowing us to exploit concurrency to speed up our applications. In this chapter we've introduced a few important concepts important to both asyncio and concurrency in general.

- CPU bound work is work that primarily utilizes out computer's processor whereas I/O bound work primarily utilizes our network or other input/output devices. Asyncio primarily helps us make I/O bound work concurrent, but it exposes APIs for making CPU bound work concurrent as well.
- Processes and threads are the basic most units of concurrency at the operating system level. Processes can be used for I/O and CPU bound workloads and threads can only be used to manage I/O bound work effectively, mostly due to the GIL preventing Python code from executing in parallel.
- We've seen how with non-blocking sockets instead of stopping our application while we wait for data to come in, we can instruct our operating system to tell us when data has come in. Exploiting this is part of what allows asyncio to achieve concurrency with only a single thread.
- We've introduced the event loop which is the core of asyncio applications. The event loop loops forever, looking for tasks with CPU bound work to run while also pausing tasks that are waiting for I/O.

In the next chapter we'll introduce more about how to write code to handle these I/O bound operations and learn more about how to create methods we can pause known as coroutines.

2

Asyncio basics

This chapter covers

- **What coroutines are and how to create them**
- **The basics of async await syntax**
- **How to simulate a long running operation**
- **How to run coroutines concurrently with tasks**
- **How to cancel tasks**
- **How to manually create the event loop**
- **How to measure a coroutine's execution time**
- **How to run in debug mode for informative log messages**
- **Problems to keep an eye out for when running coroutine**

In the last chapter we dove into concurrency, taking a look at how we can achieve it with both processes and threads. We also introduced how we could utilize non-blocking I/O and an event loop to achieve concurrency with only one thread. In this chapter we'll cover the basics of how to write programs using this single threaded concurrency model with asyncio. Using the techniques in this chapter you'll be able to take long-running operations, such as web requests, database queries and network connections and execute them in tandem.

We'll learn more about a construct called coroutines and how to use `async await` syntax to define and run coroutines. We'll also examine how to run coroutines concurrently by using tasks and examine the time savings we get from running concurrently by creating a reusable timer. Finally, we'll take a look at common errors software engineers make when using asyncio and how to use debug mode to spot these problems.

2.1 Introducing Coroutines

We can think of a coroutine just like a regular Python function, but with the superpower that it can pause its execution when it hits an operation that could take a while to complete. When that long-running operation is complete, we can 'wake up' our paused coroutine and finish executing any other code in that coroutine. While a paused coroutine is waiting for the operation it paused on to finish, we have the opportunity to run other code. This running of other code while waiting is what gives our application concurrency. We can also run several time-consuming operations concurrently, which can give our applications big performance improvements.

To both create a coroutine and pause a coroutine we'll need to learn to use Python's `async` and `await` keywords. The `async` keyword will let us define a coroutine and the `await` keyword will let us pause our coroutine when we have a long-running operation.

2.1.1 Creating coroutines with the async keyword

Creating a coroutine is straightforward and is not much different from creating a normal Python function. The only difference is instead of using the `def` keyword we use `async def`. The `async` keyword marks a function as a coroutine instead of a normal Python function.

Listing 2.1 using the async keyword

```
async def my_coroutine() -> None
    print('Hello world!')
```

The coroutine in listing 2.1 does not do anything for the time being other than print 'Hello world!'. It's also worth noting that this coroutine does not do any long-running operations, it just prints our message and returns. This means that when we put the coroutine on the event loop it will execute immediately because we don't have any blocking I/O, and nothing is pausing execution yet.

This syntax is simple, but we're creating something very different from a plain Python function. To illustrate this let's create a function which adds one to a number, and a coroutine which does the same and compare the results of calling each. We'll also use the `type` convenience function to take a look at the type returned by calling our coroutine compared to our normal function.

Listing 2.2 comparing coroutines to normal functions

```
async def coroutine_add_one(number: int) -> int:
    return number + 1


def add_one(number: int) -> int:
    return number + 1


function_result = add_one(1)
coroutine_result = coroutine_add_one(1)

print(f'Function result is {function_result} and the type is {type(function_result)}')
```

```
print(f'Coroutine result is {coroutine_result} and the type is {type(coroutine_result)}')
```

When we run this code, we'll see output similar to the following:

```
Method result is 2 and the type is <class 'int'>
Coroutine result is <coroutine object coroutine_add_one at 0x1071d6040> and the type is
       <class 'coroutine'>
```

Notice how when we call our normal `add_one` function it executes immediately and returns what we would expect, another integer. However, when we call `coroutine_add_one` we don't get our code in the coroutine executed at all. We get a coroutine object instead.

This is an important point as coroutines don't get executed when we call them directly, instead we create a coroutine object that can be run later. To run a coroutine, we need to explicitly put it on an event loop. So how can we create an event loop and run our coroutine?

In versions of Python prior to 3.7, we had to create an event loop if one did not already exist. However, the asyncio library has added several functions which abstract away managing the event loop for us. There is a convenience function, `asyncio.run`, we can use to run our coroutine.

Listing 2.3 running a coroutine

```
import asyncio

async def coroutine_add_one(number: int) -> int:
    return number + 1

result = asyncio.run(coroutine_add_one(1))

print(result)
```

Running listing 2.3 will print 2 as we would expect. We've properly put our coroutine on the event loop and have executed it!

`asyncio.run` is doing a few important things. First it creates a brand-new event loop for us. Once it successfully does so, it will take whichever coroutine we pass into it and runs it until it completes, returning the result. This function will also do some cleanup of anything that might be left running after our main coroutine finishes. Once everything has finished, it will shut down and close the event loop.

Possibly the most important thing about `asyncio.run` is that it is intended to be the main entry point into our asyncio application. It only executes one coroutine and that coroutine should launch all other aspects of our application. As we progress further, we will use this function as the entry point into nearly all of our applications. The coroutine that `asyncio.run` executes will create and run other coroutines which will allow us to utilize the concurrent nature of asyncio.

2.1.2 Pausing execution with the await keyword

The example we saw earlier did not need to be a coroutine as we just executed non-blocking Python code. The real benefit of asyncio is being able to pause execution to let the event loop run other tasks when we have a long-running operation. To pause execution, we use the `await` keyword. The `await` keyword is usually followed by a call to a coroutine (more specifically, an object known as an awaitable, which is not always a coroutine. We'll learn more about this later in the chapter).

Using the `await` keyword will cause the coroutine following it to be run, unlike directly calling a coroutine which produces a coroutine object. The `await` expression will also pause the coroutine that it is contained in until the coroutine we awaited finishes and returns a result. When the coroutine we awaited finishes, we'll have access to the result it returned and our containing coroutine will 'wake up' to handle the result.

We can use the `await` keyword by simply putting it in front of a coroutine call. Expanding on our earlier program we can write a program where we call the `add_one` function inside of a 'main' async function and get the result.

Listing 2.4 using await to wait for the result of coroutine

```
import asyncio

async def add_one(number: int) -> int:
    return number + 1


async def main() -> None:
    one_plus_one = await add_one(1) #A
    two_plus_one = await add_one(2) #B
    print(one_plus_one)
    print(two_plus_one)

asyncio.run(main())
```

#A pause and wait for result of add_one(1)
#B pause and wait for result of add_one(2)

In the above code we pause execution twice. We first wait on the call to `add_one(1)`. Once we have the result our main function will be 'unpaused' and we will assign the return value from `add_one(1)` to the variable `one_plus_one`, which in this case will be two. We then do the same for `add_one(2)` and then print the results. We can visualize the execution flow of our application as follows. Each block of this diagram represents what is happening at any given moment in time for a particular line or set of lines of code:

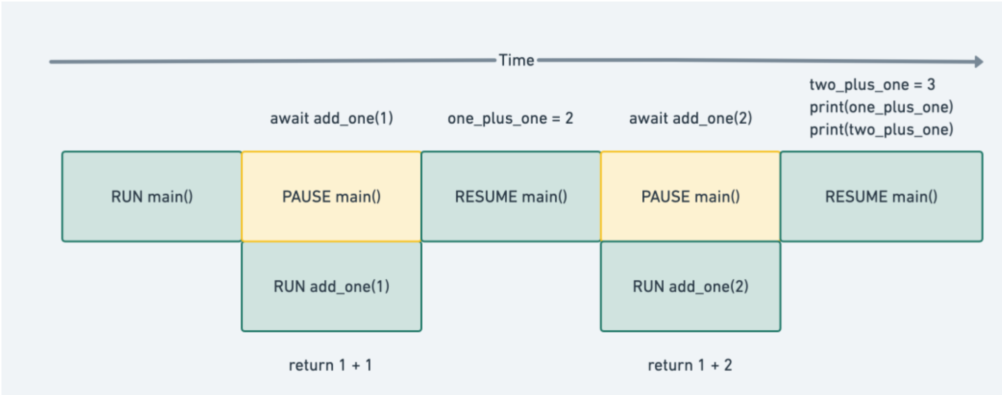


Figure 2.1 When we hit an await expression, we pause our parent coroutine and run the coroutine in the await expression. Once it is finished, we resume the parent coroutine and assign the return value.

As it stands now, this code does not operate differently from normal, sequential code. We are in effect mimicking a normal call stack. Next, we'll take a look at a simple example of how run other code while we're waiting by introducing a dummy sleep operation.

2.2 Introducing long-running coroutines with sleep

Our previous examples did not use any slow operations and were utilized to help us learn the basic syntax of coroutines. To fully see the benefits and show how we can run multiple things at the same time, we'll need to introduce some long-running operations. Instead of making web API or database queries right away, which are nondeterministic as to how much time they will take, we'll simulate this by specifying how long we want to wait. We'll do this with the `asyncio.sleep` function.

We can use `asyncio.sleep` to make a coroutine sleep for a given amount of seconds. This will pause our coroutine for the amount of time we give it, simulating what would happen if we had a long running call to a database or web API. `asyncio.sleep` is itself a coroutine, so we need to use it with the `await` keyword, if we just call it by itself, we'll get a coroutine object. Since `asyncio.sleep` is a coroutine this means that when a coroutine awaits it, other code will be able to run.

Let's take a look at a simple example which sleeps for one second and then prints a hello world message.

Listing 2.5 a first application with sleep

```
import asyncio

async def hello_world_message() -> str:
    await asyncio.sleep(1) #A
    return ‘Hello World!’

async def main() -> None:
```

```
    message = await hello_world_message() #B
    print(message)

asyncio.run(main())
```

#A pause hello_world_message for one second
#B pause main until hello_world_message finishes

When we run this application, our program will wait for a second before printing our 'Hello World!' message. Since `hello_world_message` is a coroutine and we pause it for a second with `asyncio.sleep`, we now in theory have about one second of time where we could be running other code concurrently.

We'll be using sleep a bunch in the next few examples, so let's invest the time in creating a reusable coroutine that sleeps for us and prints out some useful information. We'll call this coroutine `delay`.

Listing 2.6 a reusable delay function

```
import asyncio


async def delay(delay_seconds: int) -> int:
    print(f'sleeping for {delay_seconds} second(s)')
    await asyncio.sleep(delay_seconds)
    print(f'finished sleeping for {delay_seconds} second(s)')
    return delay_seconds
```

`delay` will take in an integer of how long we'd like to sleep and will return that integer to the caller once it has finished sleeping. We'll also print when we start sleeping and when we finish sleeping. This will help us see what other code, if any, is running concurrently while our coroutines are paused.

To make referencing this utility function easier in future code listings, we'll create a module that we'll import in the rest of this book when needed. We'll also add to this module as we create more reusable functions. We'll call this module `util` and we'll put our delay function in a file called `delay_functions.py`. We'll also add an `__init__.py` file with the following line so we can nicely import the timer:

```
from util.delay_functions import delay
```

In the rest of this book we'll use `from util import delay` whenever we need to use the delay function.

Now that we have a reusable delay coroutine, let's combine this with the earlier coroutine `add_one` to see if we can get our simple addition to run concurrently while `hello_world_message` is paused.

Listing 2.7 running two coroutines

```
import asyncio
from util import delay


async def add_one(number: int) -> int:
```

```
    return number + 1

async def hello_world_message() -> str:
    await delay(1)
    return 'Hello World!'

async def main() -> None:
    message = await hello_world_message() #A
    one_plus_one = await add_one(1) #B
    print(one_plus_one)
    print(message)

asyncio.run(main())
```

#A pause main until hello_world_message_returns
#B pause main until add_one_returns

When we run this, we still wait for one second before both function call's results are printed. What we really want is the value of `add_one(1)` to be printed immediately while `hello_world_message()` runs concurrently. So why isn't this happening with this code? The answer is that `await` pauses our current coroutine and won't execute any other code inside that coroutine until the `await` expression gives us a value. Since it will take one second for our `hello_world_message` function to give us a value, our main coroutine will be paused for an entire second. Our code behaves as if it were sequential in this case.

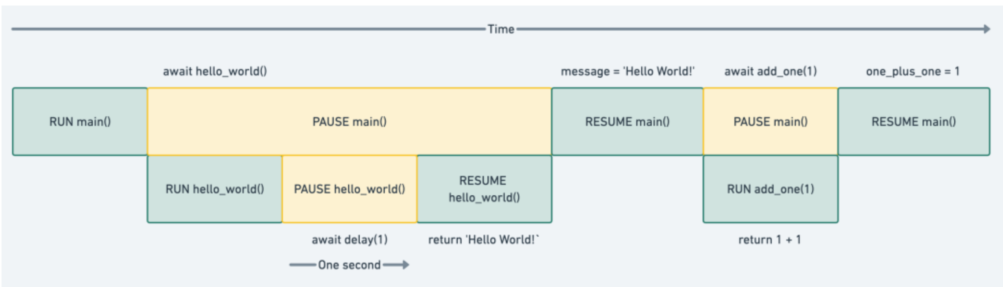


Figure 2.2 Execution flow of listing 2.7

Both `main` and `hello_world` are paused while we wait for `delay(1)` to finish. After it has finished, we resume `main` and can execute `add_one`. We'd like to move away from this sequential model and run `add_one` concurrently with `hello_world`. To achieve this, we'll need to introduce a concept called tasks.

2.3 Running concurrently with tasks

Earlier we saw that when we call a coroutine directly we don't put it on the event loop to run. Instead, we get a coroutine object that we then need to either use the `await` keyword on or pass it in to `asyncio.run` to run and get a value. With only these tools we can write async code, but we can’t run anything concurrently. To run coroutines concurrently, well need to introduce tasks.

Tasks are wrappers around a coroutine that schedule that coroutine to run on the event loop as soon as possible. This scheduling and execution happen in a non-blocking fashion, meaning once we create a task, we can instantly go on to execute other code while the task is running. This is in contrast to using the `await` keyword which acts in a blocking manner, meaning we pause our entire coroutine until the result of the `await` expression comes back.

The fact that we can create tasks instantly and schedule them to run on the event loop means that we can create multiple tasks at roughly the same time. When these tasks wrap a long-running operation, any waiting they do will happen concurrently. To illustrate this let’s create two tasks and try to run them at the same time.

2.3.1 The basics of creating tasks

Creating a task is achieved by using the `asyncio.create_task` function. When we call this function, we give it a coroutine to run and it returns a task object instantly. Once we have a task object, we can put it in an `await` expression which will extract the return value once it is complete.

Listing 2.8 creating a task

```
import asyncio
from util import delay


async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    print(type(sleep_for_three))
    result = await sleep_for_three
    print(result)

asyncio.run(main())
```

In listing 2.8 we create a task that will take three seconds to complete. We also print out the type of the task, in this case, `<class '_asyncio.Task'>`, to show that it is different from a coroutine. One other thing to note here is that our print statement gets executed immediately after we create the task. If we had simply used `await` on the `delay` coroutine we would have waited three seconds before outputting the message. Once we’ve printed our message, we apply an `await` expression to the task `sleep_for_three`. This will suspend our `main` coroutine until we have a result from our task.

It is important to know that we should usually use an `await` keyword on our tasks at some point in our application. In listing 2.8 if we did not use `await` our task would get scheduled to run but would almost immediately get stopped and ‘cleaned up’ when `asyncio.run` shuts down the event loop. Using `await` on our tasks in our application also has implications for how exceptions are handled, which we’ll take a look at in the next chapter.

Now that we’ve seen how to create a task and allow other code to run concurrently, we can further expand this to learn how to run multiple long-running operations at the same time.

2.3.2 Running multiple tasks concurrently

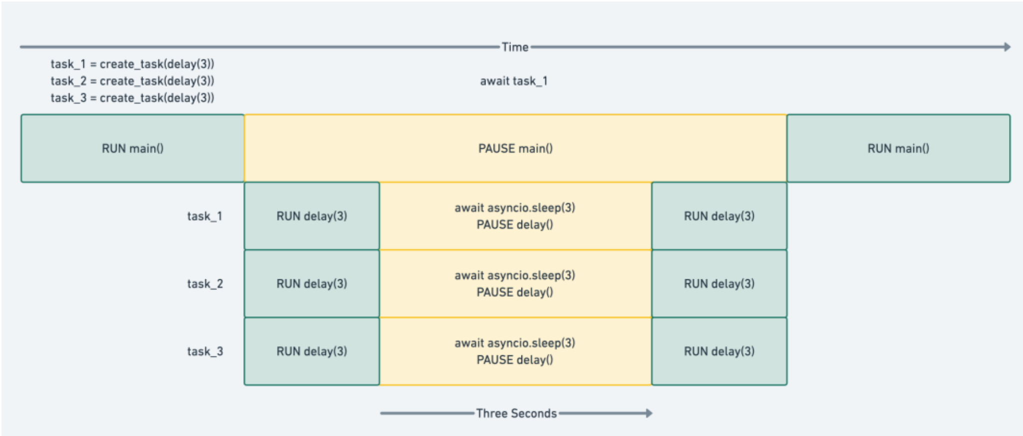
Given that tasks are created instantly and are scheduled to run as soon as possible, this opens up the opportunity for us to run many long-running tasks concurrently. We can do this by starting multiple tasks with our long running coroutine sequentially.

Listing 2.9 running multiple tasks concurrently

```
import asyncio
from util import delay


async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))
    sleep_once_more = asyncio.create_task(delay(3))

    await sleep_for_three
    await sleep_again
    await sleep_once_more


asyncio.run(main())
```

In this example we start three tasks, each taking three seconds to complete. Each call to `create_task` returns instantly, so we hit our `await sleep_for_three` statement right away. Before this listing, we mentioned that tasks are scheduled to run 'as soon as possible'. Generally, this means the first time we hit an `await` statement after creating a task, any tasks that are pending to run will start as `await` will trigger an iteration of the event loop.

Since we've hit `await sleep_for_three`, all three tasks start running and will do so concurrently. This means that the program in listing 2.9 will complete in roughly three seconds. We can visualize the concurrency as follows, noting that our tasks are running their sleep coroutines at the same time:



If we were to run these delay operations sequentially, we'd wind up with an application run time of just over nine seconds. By doing this concurrently, we've decreased the total run time of this application three-fold! This benefit compounds as we add more tasks, if we had launched 10 of these tasks, we would still take roughly three seconds, giving us a 10x speedup.

Executing these long-running operations concurrently is where asyncio really shines to give us drastic improvements in our application's performance, but the benefits don't stop there. In listing 2.9 our application was actively doing nothing while it was waiting for three seconds for our delay coroutines to complete. While our code is waiting, we can execute other code. As an example, let's say we wanted to print out a status message every second while we were running some long tasks.

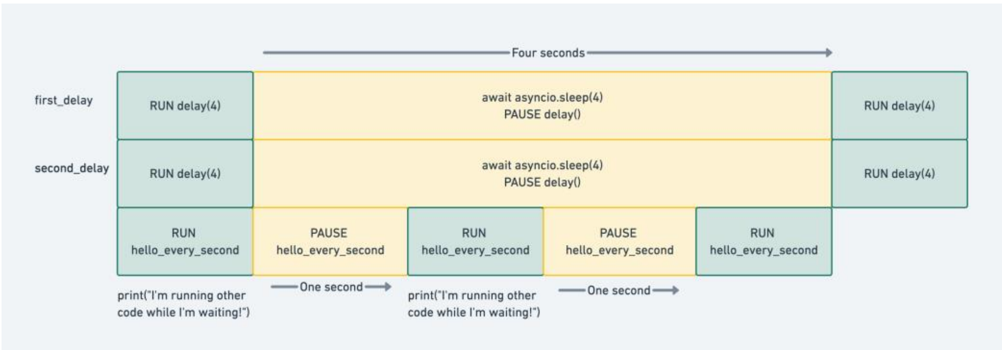
Listing 2.10 running code while other operations complete

```
import asyncio
from util import delay


async def hello_every_second():
    for i in range(3):
        await asyncio.sleep(1)
        print("I'm running other code while I'm waiting!")


async def main():
    first_delay = asyncio.create_task(delay(4))
    second_delay = asyncio.create_task(delay(4))
    await hello_every_second()
    await first_delay
    await second_delay
```

In listing 2.10 we create two tasks, each of which take four seconds to complete. While these tasks are waiting, our application is idle, which gives us the opportunity to run other code. In this instance we run a coroutine `hello_every_second` which prints a message every second three times. While our two tasks are running, we'll see messages being output, giving us the following output:

```
sleeping for 4 second(s)
sleeping for 4 second(s)
I'm running other code while I'm waiting!
I'm running other code while I'm waiting!
I'm running other code while I'm waiting!
finished sleeping for 4 second(s)
finished sleeping for 4 second(s)
```

We can imagine the execution flow as follows:



We first start two tasks that sleep for four seconds, then while our two tasks are idle we start to see "I'm running other code while I'm waiting" being printed every second. This means that even when we're running time intensive operations, our application can still be performing other tasks for us.

One potential issue with tasks is that they could take an indefinite amount of time to complete. We could find ourselves wanting to stop a task if it takes too long to finish. Tasks support this use case by allowing cancellation.

2.4 Cancelling tasks and setting timeouts

Network connections can be unreliable. A user's connection may drop because of a network slowdown, or a web server may crash and leave existing requests in limbo. When making one of these requests, we need to be especially careful that we don't wait indefinitely. Doing so could lead to our application hanging, waiting forever for a result that may never come. It could also lead to a poor user experience, if we allow a user to make a request that takes too long, they are unlikely to wait forever for something to come back. Additionally, we may want to allow our users a choice if a task continues to run. A user may proactively decide things are taking too long, or they may want to stop a task they made in error.

In our previous examples, if our tasks took forever, we would be stuck waiting for the `await` statement to finish with no feedback. We also had no way to stop things if we wanted to. Asyncio supports both these cases by allowing tasks to be cancelled as well as allowing them to specify a timeout.

2.4.1 Cancelling tasks

Cancelling a task is straightforward. Each task object has a method named `cancel` which we can call whenever we'd like to stop a task. Cancelling a task will cause that task to raise a `CancelledException` when we await it, which we can then handle if we want to.

To illustrate this, let's say we kick off a long running task that we don't want to run for longer than five seconds. If the task is not done within five seconds, we'd like to stop that task and report back to the user that it took too long and we're stopping the task. We also want a status update printed every second to provide up to date information to our user, so they aren't stuck for several seconds with no information.

Listing 2.11 canceling a task

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


async def main():
    long_task = asyncio.create_task(delay(10))

    seconds_elapsed = 0

    while not long_task.done():
        print('Task not finished, checking again in a second.')
        await asyncio.sleep(1)
        seconds_elapsed = seconds_elapsed + 1
        if seconds_elapsed == 5:
            long_task.cancel()

    try:
        await long_task
    except CancelledError:
        print('Our task was cancelled')

asyncio.run(main())
```

In listing 2.11 we create a task that will take 10 seconds to run. We then create a while loop to check if that task is done. The `done` method on the task returns `True` if a task is finished, and `False` otherwise. Every second we check to see if the task has finished, keeping track of how many seconds we've checked so far. If our task has taken five seconds, we cancel the task. Then, we will go on to `await long_task` and we'll see `Our task was cancelled` printed out, indicating we've caught a `CancelledError`.

2.4.2 Setting a timeout and canceling with wait_for

Checking every second, or at some other time interval and canceling a task as we did previously isn't the easiest way to handle a timeout. Ideally, we'd have a helper function which would allow us to specify this timeout and handle cancellation for us.

Asyncio provides this functionality through a function called `asyncio.wait_for`. This function takes in a coroutine or task object and a timeout specified in seconds. It then returns a coroutine that we can await. If the task takes more time to complete than the timeout we gave it, a `TimeoutException` will be raised. Once we have reached the timeout threshold, the task will automatically be cancelled for us.

To illustrate how `wait_for` works, we'll take a look at a case where we have a task that will take two seconds to complete, but we'll only allow it one second to finish. When we get a `TimeoutError` raised, we'll catch the exception and check to see if the task was cancelled.

Listing 2.12 creating a timeout for a task with wait_for

```
import asyncio
from util import delay

async def main():
    delay_task = asyncio.create_task(delay(2))
    try:
        result = await asyncio.wait_for(delay_task, timeout=1)
        print(result)
    except asyncio.exceptions.TimeoutError:
        print('Got a timeout!')
        print(f'Was the task cancelled? {delay_task.cancelled()}')

asyncio.run(main())
```

When we run listing 2.12, our application will take roughly one second to complete. After one second our `wait_for` statement will raise a `TimeoutError` which we then handle. We'll then see that our original delay task was cancelled for us, giving us the following output:

```
sleeping for 2 second(s)
Got a timeout!
Was the task cancelled? True
```

Cancelling tasks automatically if they take longer than expected is normally a good idea. Otherwise we may have a coroutine sitting around indefinitely, taking up resources that may never be released. However, in certain circumstances we may want to keep our coroutine running. For example, we may want to inform a user that something is taking longer than expected after a certain amount of time but not cancel the task when the timeout is exceeded. To do this we can wrap our task with the `asyncio.shield` function. This function will disallow cancellation of the coroutine we pass in, giving it a "shield" which cancellation requests bounce off of.

Listing 2.13 shielding a task from cancellation

```
import asyncio
from util import delay


async def main():
    task = asyncio.create_task(delay(10))

    try:
        result = await asyncio.wait_for(asyncio.shield(task), 5)
        print(result)
    except TimeoutError:
        print("Task took longer than five seconds, it will finish soon!")
```

```
        result = await task
        print(result)


asyncio.run(main())
```

In listing 2.13 we first create a task to wrap our coroutine. This differs from our first cancellation example because we'll need to access the task in the `except` block. If we had passed in a coroutine, `wait_for` would have wrapped it in a task, but we wouldn't be able to reference it as it is internal to the function.

Then inside of a `try` block we call `wait_for` and wrap the task in `shield` which will prevent the task from being cancelled. Inside our exception block, we print a useful message to the user letting them know that the task is still running and then we await the task we initially created. This will let it finish in its entirety, and the program's output will be as follows:

```
starting <function delay at 0x10e8cf820> with args (10,) {}
sleeping for 10 second(s)
Task took longer than five seconds!
finished sleeping for 10 second(s)
finished <function delay at 0x10e8cf820> in 10.004740953445435 second(s)
```

Cancellation and shielding are somewhat tricky subjects with several corner cases that are noteworthy. Here we introduce the basics, but as we get into more complicated cases, we'll explore more in-depth how cancellation works.

We've now introduced the basics around tasks and coroutines. Both of these concepts are intertwined with one another. In the next section, we'll take a look at how tasks and coroutines are related to one another and understand a bit more about how asyncio is structured.

2.5 Tasks, coroutines, futures and awaitables

Coroutines and tasks can both be used in `await` expressions. So, what is the common thread between both of them? To understand this, we'll need to understand what a `Future` is as well as what an `awaitable` is. You normally won't need to use futures yourself but understanding them is a key to understanding the inner workings of asyncio. We will reference them in the rest of the book as some APIs return futures.

2.5.1 Introducing futures

A `Future` is a Python object that contains a single value that you expect to get at some point in the future, but may not yet have. Usually when you create a future, it does not have any value it wraps because you do not have it yet. In this state, it is considered "incomplete", "unresolved" or simply not done. Then, once you get a result, you can set the value of the future. This will "complete" the future and at that time we can consider it done and extract the result from the future. To understand the basics of futures, let's try creating one, setting its value, and extracting that value back out.

Listing 2.14 the basics of futures

```
from asyncio import Future
```

```
my_future = Future()

print(f'Is my_future done? {my_future.done()}')

my_future.set_result(42)

print(f'Is my_future done? {my_future.done()}')
print(f'What is the result of my_future? {my_future.result()}')
```

We can create a future by calling its constructor. At this time the future will have no result set on it, so calling it's `done` method will return `False`. We then set the value of the future with its `set_result` method, which will mark the future as done. Alternatively, if we had an exception we wanted to set on the future, we could call `set_exception`. Note that we don't call the result method before the result is set. This is because the `result` method will throw an invalid state exception if we do so.

Futures can also be used in `await` expressions. If we await a future we're saying 'pause until the future has a value set that I can work with, once I have a value, wake up and let me process it`. To understand this, let's consider an example of making a web request that returns a future. Making a request that returns a future should complete instantly, but the future will not yet be done as the request will take some time. Then, later on once the request has finished, the result will be set and then we can access it. If you have used Javascript in the past this concept is analogous to promises and in the Java world these are known as completable futures.

Listing 2.15 awaiting a future

```
from asyncio import Future
import asyncio


def make_request() -> Future:
    future = Future()
    asyncio.create_task(set_future_value(future)) #A
    return future


async def set_future_value(future) -> None:
    await asyncio.sleep(1) #B
    future.set_result(42)


async def main():
    future = make_request()
    print(f'Is the future done? {future.done()}')
    value = await future #C
    print(f'Is the future done? {future.done()}')
    print(value)


asyncio.run(main())
```

#A Create a task to asynchronously set the value of the future
#B Wait one second before setting the value of the future

#C Pause main until the future's value is set

In listing 2.15 we define a function `make_request`. In that function we create a Future and create a task which will asynchronously set the result of the future after one second. Then, in our main function, we call `make_request`. When we call this, we'll instantly get a future that has no result and is therefore not done. Then, we await the future. Awaiting this future will pause `main` for one second while we wait for the value of the future to be set. Once this completes `value` will be 42 and the future is done.

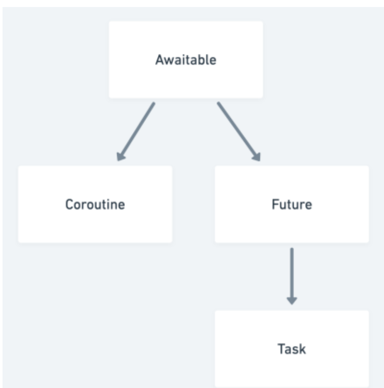
It should be noted that in the world of asyncio you should rarely need to deal with futures. That said, you will run into some asyncio APIs which return futures and you may need to work with callback-based code which can require futures. You may also need to read or debug some asyncio API code yourself. The implementation of the asyncio APIs heavily rely on futures, so it is ideal to have a basic understanding of how they work.

2.5.2 The relationship between futures, tasks and coroutines

There is a strong relationship between tasks and futures. In fact, futures directly inherit from tasks. A future can be thought as representing a value that we won't have for a while. A task can be thought as a combination of both a coroutine and a future. When we create a task, we are saying create an empty future and run the coroutine. Then, when the coroutine has completed with either an exception or a result, we set the result or exception of the future.

Given there is a relationship between futures and tasks, is there a similar relationship between tasks and coroutines? After all, all of these types can be used in await expressions.

It turns out there is. The common thread between these is the `Awaitable` abstract base class. This class defines one abstract double underscore method `__await__`. We won't go into the specifics about how to create our own awaitables, but anything that implements the `__await__` method can be used in an await expression. Coroutines inherit directly from `Awaitable`, as do futures. Tasks then extend futures, which gives us the following inheritance diagram.



Going forward, we'll start to refer to objects that can be used in await expressions as *awaitables*. You'll frequently see the term awaitable referenced in the asyncio documentation, as many API methods don't care if you pass in coroutines, tasks or futures.

Now that we understand the basics of coroutines, tasks and futures, how do we assess their performance? So far, we've only theorized about how long they take. To make things more rigorous, let's add some functionality to measure execution time.

2.6 Measuring coroutine execution time with decorators

So far, we've talked about roughly how long our applications take to run without actually timing them. To really understand and profile things we'll need to introduce some code to keep track of this for us. As a first try we could wrap every `await` statement and keep track of the start and end time of the coroutine:

```
import asyncio
import time

async def main():
    start = time.time()
    await asyncio.sleep(1)
    end = time.time()
    print(f'Sleeping took {end - start} seconds')

asyncio.run(main())
```

However, this will get messy quickly when we have multiple `await` statements and tasks that we need to keep track of. A better approach is to come up with a reusable way to keep track of how long any coroutine takes to finish. We can do this by creating a decorator which will run an `await` statement for us. We'll call this decorator `async_timed`.

> **WHAT IS A DECORATOR?** A decorator is a pattern in Python which allows us to add functionality to existing functions without changing that function's code. We can 'intercept' a function as it is being called and apply any decorator code we'd like before or after that call. Decorators are one way to tackle cross-cutting concerns.

Listing 2.16 A decorator for timing coroutines

```
import functools
import time
from typing import Callable, Any


def async_timed():
    def wrapper(func: Callable) -> Callable:
        @functools.wraps(func)
        async def wrapped(*args, **kwargs) -> Any:
            print(f'starting {func} with args {args} {kwargs}')
            start = time.time()
            try:
                return await func(*args, **kwargs)
            finally:
                end = time.time()
                total = end - start
                print(f'finished {func} in {total:.4f} second(s)')

        return wrapped
```

```
    return wrapper
```

In this decorator we create a new coroutine called wrapped. Wrapped is a wrapper around our original coroutine that takes its arguments `*args` and `**kwargs` and calls an `await` statement and returns the result. We surround that `await` statement with a message when we start running the function and one when we end running the function, keeping track of the start and end time in much the same way that we did in our first example. Now we can put this annotation on any coroutine and any time we run it we'll see how long it took to run.

Listing 2.17 timing two concurrent tasks with a decorator

```
import asyncio

@async_timed()
async def delay(delay_seconds: int) -> int:
    print(f'sleeping for {delay_seconds} second(s)')
    await asyncio.sleep(delay_seconds)
    print(f'finished sleeping for {delay_seconds} second(s)')
    return delay_seconds


@async_timed()
async def main():
    task_one = asyncio.create_task(delay(2))
    task_two = asyncio.create_task(delay(3))

    await task_one
    await task_two


asyncio.run(main())
```

When we run listing 2.17, we'll see console output similar to the following

```
starting <function main at 0x109111ee0> with args () {}
starting <function delay at 0x1090dc700> with args (2,) {}
starting <function delay at 0x1090dc700> with args (3,) {}
finished <function delay at 0x1090dc700> in 2.0032 second(s)
finished <function delay at 0x1090dc700> in 3.0003 second(s)
finished <function main at 0x109111ee0> in 3.0004 second(s)
```

We can see that our two delay calls were both started and finished in roughly two and three seconds respectively for a total of five seconds overall. Notice however our main coroutine only took three seconds to complete because we were waiting concurrently.

We'll use this decorator and the resulting output throughout the next several chapters to illustrate how long our coroutines are taking to execute as well as when they start and complete. This will give us a clear picture of where we see performance gains by executing our operations concurrently.

To make referencing this utility decorator easier in future code listings, let’s add this to our `util` module. We’ll put our timer in a file called `async_timer.py`. We’ll also add a line to the module’s `__init__.py` file with the following line so we can nicely import the timer:

```
from util.async_timer import async_timed
```

In the rest of this book we'll use `from util import async_timed` whenever we need to use the timer.

Now that we can use our decorator to understand the performance gains that asyncio can provide when running tasks concurrently, we may be tempted to try and use asyncio all over our existing applications. This can work, but we need to be careful that we aren't running into any of the common pitfalls with asyncio that can degrade our application's performance.

2.7 The pitfalls of coroutines and tasks

When seeing the performance improvements we can get from running some of our longer tasks concurrently, we can be tempted to start to use coroutines and tasks everywhere in our applications. While it depends on which application you're writing, simply marking functions `async` and wrapping them in tasks may not help application performance. In certain cases, this may degrade performance of your applications. There are two main errors made when trying to turn your applications asynchronous. The first is attempting to run CPU bound code in tasks or coroutines (without using multiprocessing), the second is using blocking I/O bound APIs (without using multithreading).

2.7.1 Running CPU bound code

We may have functions which perform calculations which are computationally expensive, such as looping over a large dictionary or doing a mathematical computation. Where we have several of these functions that have the potential to run concurrently, we may get the idea to run them in a few separate tasks. In concept this is a good idea, but remember asyncio has a single-threaded concurrency model. This means we are still subject to the limitations of a single thread and the global interpreter lock. To prove this to ourselves, let's try and run some CPU bound functions concurrently.

Listing 2.18 attempting to run CPU bound code concurrently

```
import asyncio
from util import delay


@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter


@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    await task_one
    await task_two


asyncio.run(main())
```

When we run listing 2.18 we'll see that despite creating two tasks, our code still executes sequentially. First, we run task one, then we run task two, meaning our total runtime will be the sum of the two calls to `cpu_bound_work`.

```
starting <function main at 0x10a8f6c10> with args () {}
starting <function cpu_bound_work at 0x10a8c0430> with args () {}
finished <function cpu_bound_work at 0x10a8c0430> in 4.6750 second(s)
starting <function cpu_bound_work at 0x10a8c0430> with args () {}
finished <function cpu_bound_work at 0x10a8c0430> in 4.6680 second(s)
finished <function main at 0x10a8f6c10> in 9.3434 second(s)
```

Looking at the above output we may be tempted to think that there is no drawback to making all of our code use `async` and `await`. After all, it winds up taking the same amount of time as if we had run things sequentially. However, we can run into situations where our application's performance can degrade by doing this. This is especially true when we have other coroutines or tasks which have `await` expressions. Consider creating two CPU bound tasks alongside a long-running task such as our delay coroutine:

Listing 2.19 CPU bound code with a task

```
import asyncio
from util import async_timed, delay


@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter


@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    delay_task = asyncio.create_task(delay(4))
    await task_one
    await task_two
    await delay_task


asyncio.run(main())
```

Running listing 2.19 we might expect to take the same amount of time as the previous example. After all, won't `delay_task` run concurrently alongside the CPU bound work? In this instance it won't, because we create the two CPU bound tasks first, which in effect blocks the event loop from running anything else. This means the runtime of our application will be the sum of time it took for our two `cpu_bound_work` tasks to finish plus the four seconds that our `delay` task took.

If we need perform CPU bound work and still want to use async / await syntax we can do so. To do this we'll still need to use multiprocessing and tell asyncio to run our tasks in a process pool. We'll learn how to do this in chapter 6.

2.7.2 Running blocking APIs

We may also be tempted to utilize our existing libraries for I/O bound operations by wrapping them in coroutines. However, this will run into the same issues that we just saw with CPU bound operations. The issue here is that these APIs block the main thread. Therefore, when we run a blocking API call inside a coroutine we're blocking the event loop thread itself, meaning we stop any other coroutines or tasks from executing. Examples of blocking API calls include libraries such as `requests`, or `time.sleep`. Generally, any function that performs I/O that is not a coroutine or performs time-consuming CPU operations can be considered blocking.

As an example, let's try getting the status code of example.com three times concurrently using the requests library. When we run this, we'll be expecting this application to finish in about the amount of time it takes to get the status code once since we're running concurrently.

Listing 2.20 incorrectly using a blocking API in a coroutine

```
import asyncio
import requests
from util import async_timed


@async_timed()
async def get_example_status() -> int:
    return requests.get('http://www.example.com').status_code


@async_timed()
async def main():
    task_1 = asyncio.create_task(get_example_status())
    task_2 = asyncio.create_task(get_example_status())
    task_3 = asyncio.create_task(get_example_status())
    await task_1
    await task_2
    await task_3

asyncio.run(main())
```

When running listing 2.20, we'll see output similar to the following. Note how the total runtime of the main coroutine is roughly the sum of all the tasks to get the status we ran, meaning we did not have any concurrency advantage.

```
starting <function main at 0x1102e6820> with args () {}
starting <function get_example_status at 0x1102e6700> with args () {}
finished <function get_example_status at 0x1102e6700> in 0.0839 second(s)
starting <function get_example_status at 0x1102e6700> with args () {}
finished <function get_example_status at 0x1102e6700> in 0.0441 second(s)
starting <function get_example_status at 0x1102e6700> with args () {}
finished <function get_example_status at 0x1102e6700> in 0.0419 second(s)
finished <function main at 0x1102e6820> in 0.1702 second(s)
```

This is again due to the fact that the requests library is blocking, meaning it will block whichever thread it is run on. Since asyncio only has one thread the request library blocks the event loop from doing anything concurrently.

As a general rule, most APIs you utilize now are blocking and won't work out of the box with asyncio. You need to use a library which supports coroutines and utilizes non-blocking sockets. This means that if the library you are using does not return coroutines and you aren't using `await` in your own coroutines, you're likely making a blocking call. In the above example we can use a library such as AIOHTTP which uses non-blocking sockets and returns coroutines to get proper concurrency. We'll introduce this library later in the book.

If you need to use the requests library, you can still use async syntax, but you'll need to explicitly tell asyncio to use multithreading with a thread pool executor. We'll see how to do this in chapter 7.

We've now seen a few things to look out for when using asyncio and have built a few simple applications. So far, we have not created or configured the event loop ourselves but relied on convenience methods to do it for us. Next, we'll learn how to create the event loop ourselves, which will allow us to access lower level asyncio functionality and event loop configuration properties.

2.8 Accessing and manually managing the event loop

Until now we have used the convenient `asyncio.run` to run our application and create the event loop for us behind the scenes. Given the ease of use, this is the preferred method to create the event loop. However, there may be cases where we don't want the functionality that `asyncio.run` provides. As an example, we may want to execute custom logic to stop tasks that differs from what `asyncio.run` does, such as letting any remaining tasks finish instead of stopping them.

In addition, we may want to access methods available on the event loop itself. These methods are typically lower level and as such should be used sparingly. However, if you want to perform tasks such as working directly with sockets or scheduling a task to run at a specific time in the future, you'll need to access the event loop. While we won't, and shouldn't, be managing the event loop extensively, this will be necessary from time to time.

2.8.1 Creating an event loop manually

We can create an event loop by using the `asyncio.new_event_loop` method. This will return an event loop instance. With this we have access to all the lower-level methods that the event loop has to offer. With the event loop we have access to a method named `run_until_complete` which takes a coroutine and runs it until it finishes. Once we are done with our event loop, we need to close it to free up any resources it was using. This should normally be in a finally block so that any exceptions thrown don't stop us from closing the loop. Using these three concepts, we have the ability to create a loop and run an asyncio application.

Listing 2.21 manually creating the event loop

```
import asyncio

async def main():
    await asyncio.sleep(1)

loop = asyncio.new_event_loop()
```

```
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

The code in this listing is similar to what happens when we call `asyncio.run`, with the difference that this does not perform canceling of any remaining tasks. If we want any special cleanup logic, we would do so in our `finally` clause.

2.8.2 Accessing the event loop

From time to time, we may need to access the currently running event loop. Asyncio exposes the `asyncio.get_event_loop` function which allows us to get the current event loop. As an example, let's take a look at `call_soon`, which will schedule a function to run on the next iteration of the event loop.

Listing 2.22 Accessing the event loop

```
import asyncio

def call_later():
    print("I'm being called in the future!")

async def main():
    loop = asyncio.get_event_loop()
    loop.call_soon(call_later)
    await delay(1)

asyncio.run(main())
```

In listing 2.22 our main coroutine gets the event loop and tells it to run `call_later`, which takes a function and will run it on the next iteration of the event loop.

While we shouldn't use the event loop frequently in our applications, there are times when we will need to configure settings on the event loop or use lower-level functions. We'll see an example of configuring the event loop in the next section on debug mode.

2.9 Using debug mode

In previous sections we mentioned how coroutines should always be awaited at some point in the application. We also saw the drawbacks of running CPU bound and other blocking code inside coroutines and tasks. It can however be hard to tell if a coroutine is taking too much time on CPU or if we accidently forgot an await somewhere in our application. Luckily, asyncio gives us a debug mode to help us diagnose exactly these situations.

When we run in debug mode, we'll see a few helpful log messages when a coroutine or task takes more than 100 milliseconds to run. In addition, if we don't await a coroutine we'll get an exception thrown so we can see where to properly add an await. There are a few different ways to run in debug mode.

Using asyncio.run

The asyncio.run function we have been using to run coroutines exposes a debug parameter. By default, this is set to `False`, but we can set this to True to enable debug mode.

```
asyncio.run(coroutine(), debug=True)
```

Using command line arguments

Debug mode can be enabled by passing a command line argument when we start our Python application. To do this we apply `-X dev`

```
python3 -X dev program.py
```

Using environment variables

We can also use environment variables to enable debug mode setting the PYTHONASYNCIODEBUG variable to 1

```
PYTHONASYINCIODEBUG=1 python3 program.py
```

As of Python 3.8.2, there is currently a bug within debug mode. When using `asyncio.run`, only the boolean debug parameter will work. Command line arguments and environment variables will only work when manually managing the event loop. This will be fixed in a future version of Python.

In debug mode, we'll see informative messages logged when a coroutine takes too long. Let's test this out by trying to run CPU bound code in a task to see if we get a warning.

Listing 2.23 running cpu bound code in debug mode

```
import asyncio

async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    await task_one

asyncio.run(main(), debug=True)
```

When running this we'll see a helpful message that `task_one` was taking too long and therefore was blocking he event loop from running any other tasks:

```
Executing <Task finished name='Task-2' coro=<cpu_bound_work() done, defined at
       listing_2_9.py:5> result=100000000 created at tasks.py:382> took 4.829 seconds
```

This can be helpful for debugging issues where we may inadvertently be making a call that is blocking. The default settings will log a warning if a coroutine takes longer than 100 milliseconds, but this may be longer or shorter than we'd like. To change this value, we can set the slow callback duration by accessing the event loop as we do in listing 2.24.

Listing 2.24 changing the slow callback duration

```
import asyncio

async def main():
    loop = asyncio.get_event_loop()
    loop.slow_callback_duration = 250
```

```
asyncio.run(main(), debug=True)
```

Listing 2.24 will set the slow callback duration to 250 milliseconds, meaning we'll get a message printed out if any coroutine takes longer than 250 milliseconds to run.

2.10 Summary

In this chapter we've learned how to create basic programs with some important asyncio concepts:

- We've learned how to create coroutines with the `async` keyword. Coroutines can suspend their execution on a blocking operation. This allows for other coroutines to run. Once the operation the coroutine suspended on completes, our coroutine will wake up and resume where it left off.
- We learned to use `await` in front of a call to a coroutine to run it and wait for it to return a value. To do so the coroutine with the await inside it will suspend its execution while waiting for a result. This allows other coroutines to run while it is awaiting its result.
- We've learned how to use `asyncio.run` to execute a single coroutine. We can use this function to run the coroutine that is the main entry point into our application.
- We've learned how to use tasks to run multiple long-running operations concurrently. Tasks are wrappers around coroutines which will then be run on the event loop. When we create a task, it is scheduled to run on the event loop as soon as possible.
- We've learned how to cancel tasks if we want to stop them and how to add a timeout to a task to prevent them from taking forever. Cancelling a task will make them raise a `CancelledException` when we await them. If we have time limits on how long a task should take, we can set timeouts on them by using `asycio.wait_for`.
- We've learned to avoid a couple common issues that newcomers make when using asyncio. The first is running CPU bound code in coroutines. CPU bound code will block the event loop from running other coroutines since we're still single threaded. The second is blocking I/O, you can't use your normal libraries with asyncio, you need to use asyncio specific ones that return coroutines. If your coroutine does not have an await in it, you should consider it suspicious. There are still ways to use CPU bound and blocking I/O with asyncio which we will learn in chapter 6 and 7.
- We've learned how to use debug mode. Debug mode can help us diagnose common issues in asyncio code, such as running CPU intensive code in a coroutine.

The concepts of coroutines, tasks, `async` and `await` are the building blocks of asyncio applications. In the next chapter, we'll use these building blocks to build our first asyncio application, a multi-client echo server. In doing so, we'll learn about non-blocking sockets to learn a bit more about how the underlying machinery of the asyncio event loop works.

3

A first asyncio application

This chapter covers

- **How to use sockets to transfer data over a network**
- **How to use telnet to communicate with a socket-based application**
- **The issues with blocking sockets, and how to resolve them with non-blocking sockets**
- **Using selectors to build a simple event loop for non-blocking sockets**
- **How to use non-blocking sockets with the asyncio event loop**
- **How to create a non-blocking echo server that allows for multiple connections**
- **How to handle exceptions in tasks**
- **How to add custom shutdown logic to an asyncio application**

In the first two chapters we introduced coroutines, tasks and the event loop. We also examined how to run long operations concurrently and explored some of asyncio's APIs that facilitate this. Up to this point however, we've only simulated long operations with the sleep function. Since we'd like to build more than just demo applications, we'll use some real-world blocking operations to demonstrate how to create a server that can handle multiple users concurrently. We'll do this all with only one thread, leading to a more resource efficient and simpler application than other solutions which would involve threads or multiple processes. We'll take what we've learned about coroutines, tasks and asyncio's API methods to build a working command-line echo server application using sockets to demonstrate this. By the end of this chapter, you'll be able to build socket-based network applications with asyncio that can handle multiple users simultaneously with one thread.

We'll first learn the basics of how to send and receive data with blocking sockets. We'll then use these sockets to attempt building a multi-client echo server. In doing so, we'll demonstrate that we can't build an echo server that works properly for more than one client at a time with just a single thread. We'll then learn how to resolve these issues by making our sockets non-blocking and using the operating system's event notification system. This will help us understand how the underlying machinery of the asyncio event loop works. Then we'll use asyncio's non-blocking socket coroutines to allow multiple clients to properly connect. This application will let multiple users connect simultaneously, letting them send and receive messages concurrently. Finally, we'll add custom shutdown logic to our application, so that when our server shuts down, we'll give in-flight messages some time to complete.

3.1 Working with blocking sockets

In the first chapter we introduced the concept of sockets. To quickly reintroduce the subject, a socket is a way to read and write data over a network. We can think of a socket as a mailbox, we put a letter in, and it is delivered to the recipient's address. The recipient can then read that message, and possibly send us another back.

To get started, we'll create our main mailbox socket which we'll call our server socket. This socket will first accept connection messages from clients that want to talk to us. Once that connection is acknowledged by our server socket, we'll create a socket that we can use to communicate back to the client. This means our server starts to look more like a post office with multiple PO boxes than just one mailbox. When a client connects to our server, we give them a PO box. We then use that PO box to send and receive messages to and from that client. The client side can still be thought of as having a single mailbox as they will have one socket to communicate with us.

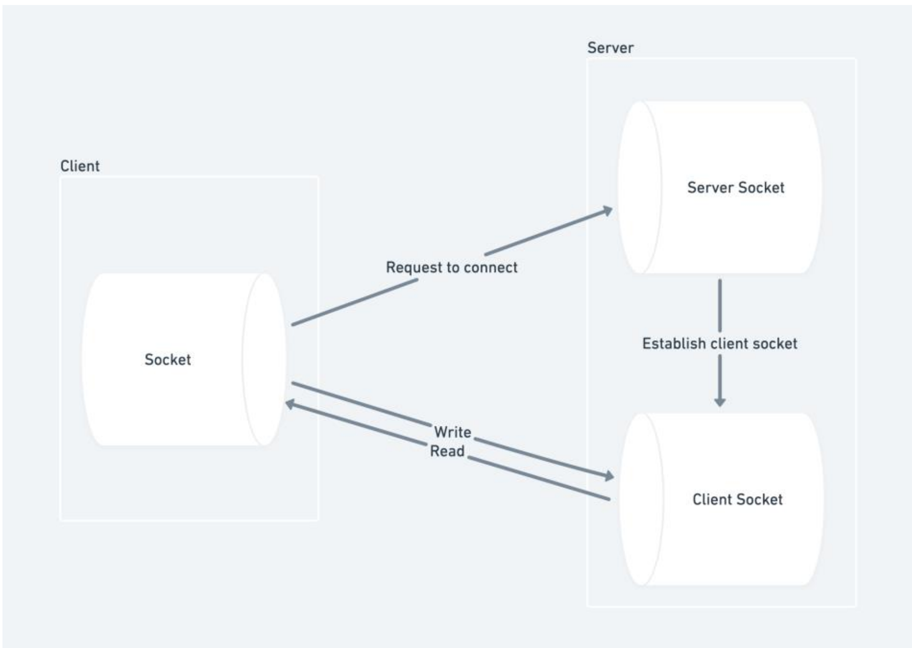


Figure 3.1 A client connects to our server socket. The server then creates a new socket to communicate with the client.

We can create this server socket with Python's built-in socket module. This module provides functionality for reading, writing and manipulating sockets. To get started creating sockets, we'll create a simple server which listens for a connection from a client and prints a message on a successful connection. This socket will be bound to both a hostname and a port and will be the main 'server socket' that any clients will communicate with.

It takes a few steps to create socket. We first use the `socket` function to create a socket:

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Here we specify two parameters to the socket function. The first is `socket.AF_INET` – this tells us what type of address our socket will be able to interact with, in this case a hostname and a port number. The second is `socket.SOCK_STREAM` – this means that we are going to use the TCP protocol for our communication.

> **WHAT IS THE TCP PROTOCOL?** TCP, or Transmission Control Protocol is a protocol designed to transfer data between applications over a network. This protocol is designed with reliability in mind. It performs error checking, delivers data in order and can retransmit data when needed. This reliability comes at the cost of some overhead. The vast majority of the web is built on TCP. TCP is in contrast to UDP, or User Datagram Protocol, which is less reliable, but has much less overhead than TCP and tends to be more performant. We will exclusively focus on TCP sockets in this book.

Calling `socket.socket` lets us create a socket, but we can't start communicating with it yet because we haven't bound it to an address that clients can talk to (our post office needs an address!). For this example, we'll bind the socket to an address on our own computer at `localhost`, and we'll pick an arbitrary port number of 8000.

```
address = ('localhost', 8000)
server_socket.bind(server_address)
```

Now we've set our socket up at the address localhost:8000. This means that clients will be able to use this address to send data to our server, and if we write data to a client, they will see this as the address that it's coming from.

Next, we need to actively listen for connections from clients who want to connect to our server. To do this we can call the `listen` method on our socket. This tells the socket to listen for incoming connections, which will allow clients to connect to our server socket. Then we wait for a connection by calling the `accept` method on our socket. This method will block until we get a connection, and when we do it will return a connection and the address of the client that connected. The connection is just another socket we can use to read data from and write data to our client.

```
server_socket.listen()
connection, client_address = server_socket.accept()
```

With these pieces, we have all the building blocks we need to create a socket-based server application that will wait for a connection and print a message once we have one.

Listing 3.1 starting a server and listening for a connection

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #A

server_address = ('localhost', 8000)
server_socket.bind(server_address) #B
server_socket.listen() #C

connection, client_address = server_socket.accept() #D
print(f'I got a connection from {client_address}!')
```

#A Create a TCP server socket
#B Set the address of the socket to localhost:8000
#C Listen for connections or 'open the post office'
#D Wait for a connection and assign the client a PO box

In listing 3.1 when a client connects, we get their connection socket as well as their address and print that we got a connection. So now that we've built this application, how do we connect to it to test it out? While there are quite a few tools for this, in this chapter we'll use the telnet command line application.

3.1.1 Connecting to a server with telnet

Our simple example of accepting connections left us with no way to connect. There are many command line applications to read and write data to and from a server, but a popular application that has been around for quite some time is telnet. If you do not have telnet on your machine, the appendix will have instructions on how to install it.

Telnet was first developed in 1969 and is short for "teletype network". Telnet establishes a TCP connection to a server and a host we specify. Once we do so, a terminal is established and we're free to send and also receive bytes, all of which will be displayed in the terminal.

To connect to the server we built in listing 3.1, we can simply use the telnet command on a command line and specify that we'd like to connect to localhost on port 8000:

```
telnet localhost 8000
```

Once we do this, we'll see some output on our terminal telling us that we've successfully connected. Telnet then will display a cursor, which allows us to type and hit enter to send data to the server.

```
telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Now in the console output of our server application, we should see output like the following, showing that we've established a connection with our telnet client:

```
I got a connection from ('127.0.0.1', 56526)!
```

You'll also see a `Connection closed by foreign host` message as the server code exits, indicating the server has shut down the connection to our client. We now have a way to connect to a server and write and read bytes to and from it, but our server can't read or send any data itself. We can do this with our client socket's `sendall` and `recv` methods.

3.1.2 Reading and writing data to and from a socket

Now that we've created a server capable of accepting connections, let's examine how to read data from our connections. The socket class has a method named `recv` that we can use to get data from a particular socket. This method takes an integer representing how many bytes we wish to read at a given time. This is important because we can't read all data from a socket all at once, we need to buffer until we reach the end of the input. In this case, we'll treat the end of input as carriage return plus a line feed or `'\r\n'`. This is what gets appended to the input when a user presses enter in telnet. To demonstrate how buffering works with small messages, we'll set a buffer size intentionally low. In a real-world application we would use a larger buffer size, such as 1024 bytes. We would typically want a larger buffer size as this will take advantage of the buffering that occurs at the operating system-level, which is more efficient than doing it in your application.

Listing 3.2 reading data from a socket

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ('localhost', 8000)
server_socket.bind(server_address)
server_socket.listen()

try:
    connection, client_address = server_socket.accept()
    print(f'I got a connection from {client_address}!')

    buffer = connection.recv(2)
    print(f'I got data: {buffer}!')

    while buffer[-2:] != b'\r\n':
        data = connection.recv(2)
        print(f'I got data: {data}!')
        buffer = buffer + data

    print(f"All the data is: {buffer}")
finally:
    server_socket.close()
```

In listing 3.2 we wait for a connection with `server_socket.accept` like we did before. Once we get a connection, we try to receive two bytes and store it in our buffer. Then we go into a loop, checking each iteration to see if our buffer ends in a carriage return and a line feed. If it does not, we get two more bytes and print out which bytes we got and append that to the buffer. If we get '\r\n', then we end the loop and we print out the full message we got from the client. We also close the server socket in a finally block. This ensures that we close the connection even if an exception occurs while reading data. If we connect to this application with telnet and send a message 'testing123', we'll see this output:

```
I got a connection from ('127.0.0.1', 49721)!
I got data: b'te'!
I got data: b'st'!
I got data: b'in'!
I got data: b'g1'!
I got data: b'23'!
I got data: b'\r\n'!
All the data is: b'testing123\r\n'
```

Now we're able to read data from a socket, but how do we write data back to a client? Sockets also have a method named `sendall`. This will take a message and write it back to the client for us. We can adapt our code in listing 3.2 to echo the message the client sent to us by calling `connection.sendall` with the buffer once it is filled:

```
    while buffer[-2:] != b'\r\n':
        data = connection.recv(2)
        print(f'I got data: {data}!')
        buffer = buffer + data

    print(f"All the data is: {buffer}")
    connection.sendall(buffer)
```

Now when we connect to this application and send it a message from telnet, we should see that message printed back on our telnet terminal. We've created a very basic echo server with sockets!

This application handles one client at a time right now, but multiple clients can connect to a single server socket. Let's adapt this example to allow multiple clients to connect at the same time. In doing this we'll demonstrate how we can't properly support multiple clients with blocking sockets.

3.1.3 Allowing multiple connections and the dangers of blocking

A socket in listen mode allows for multiple client connections simultaneously. This means that we can call `socket.accept` repeatedly, and each time a client connects we will get a new connection socket to read and write data to and from that client. With that knowledge, it is pretty straightforward to adapt our previous example to handle multiple clients. We loop forever, calling `socket.accept` to listen for new connections. Each time we get one, we append it to a list of connections we've got so far. Then, we loop over each connection, receiving data as it comes in and writing that data back out to the client connection.

Listing 3.3 allowing multiple clients to connect

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ('localhost', 8000)
server_socket.bind(server_address)
server_socket.listen()

connections = []

try:
```

```
    while True:
        connection, client_address = server_socket.accept()
        print(f'I got a connection from {client_address}!')
        connections.append(connection)

        for connection in connections:
            buffer = connection.recv(2)
            print(f'I got data: {buffer}!')

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                print(f'I got data: {data}!')
                buffer = buffer + data

            print(f"All the data is: {buffer}")

            connection.send(buffer)
finally:
    server_socket.close()
```

We can try this by making one connection with telnet and typing a message. Then, once we have done that, we can connect with a second telnet client and send another message. However, if we do this, we will notice a problem right away. Our first client will work fine and will echo messages back as we'd expect, but our second client won't get anything echoed back to it. This is due the default blocking behavior of sockets. The methods `accept` and `recv` block until they receive data. This means that once the first client connects, we will block waiting for it to send its first echo message to us. This causes other clients to be stuck waiting for the next iteration of the loop, which won't happen until the first client sends us data.

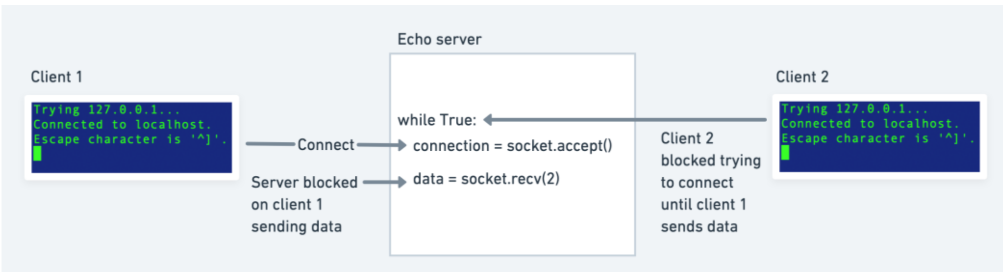


Figure 3.2 With blocking sockets, client one connects but client two is blocked until client one sends data.

This obviously isn't a satisfactory user experience; we've created something that won't properly scale when we have more than one user. We can solve this issue by putting our sockets in non-blocking mode. When we mark a socket as non-blocking, its methods will not block waiting to receive data before they move on to execute the next line of code.

3.2 Working with non-blocking sockets

Our previous echo server allowed multiple clients to connect, however, when more than one connected, we ran into issues where one client could cause others to wait for it to send data.

We can address this issue by putting sockets into non-blocking mode. When we put a socket into non-blocking mode, any time we call a method that would block, such as `recv`, it is guaranteed to return instantly. If the socket has data ready for us to process, then we will get data returned as we would with a blocking socket, if not, the socket will instantly let us know it does not have any data ready, and we are free to move on to execute other code.

Listing 3.4 creating a non-blocking socket

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 8000))
server_socket.listen()
server_socket.setblocking(False)
```

Creating a non-blocking socket is not fundamentally any different from creating a blocking one, except that we need to call `setblocking` with `False`. By default, a socket will have this value set to `True` indicating it is blocking. Now let's see what happens when we do this in our original application, does this fix the issue?

Listing 3.5 a first attempt at a non-blocking server

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ('localhost', 8000)
server_socket.bind(server_address)
server_socket.listen()
server_socket.setblocking(False) #A

connections = []

try:
    while True:
        connection, client_address = server_socket.accept()
        connection.setblocking(False) #B
        print(f'I got a connection from {client_address}!')
        connections.append(connection)

        for connection in connections:
            buffer = connection.recv(2)
            print(f'I got data: {buffer}!')

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                print(f'I got data: {data}!')
                buffer = buffer + data

            print(f"All the data is: {buffer}")

            connection.send(buffer)
finally:
    server_socket.close()
```

#A Mark the server socket as non-blocking
#B Mark the client socket as non-blocking

When we run listing 3.5, we'll notice something different right away. Our application crashes almost instantly! We'll get thrown a `BlockingIOError` because our server socket has no connection yet and therefore no data to process:

```
Traceback (most recent call last):
  File "echo_server.py", line 14, in <module>
    connection, client_address = server_socket.accept()
  File " python3.8/socket.py", line 292, in accept
    fd, addr = self._accept()
BlockingIOError: [Errno 35] Resource temporarily unavailable
```

This is the socket's somewhat unintuitive way of telling us, "I don't have any data, try calling me again later." There is no easy way for us to tell if a socket has data right now, so one solution is just catch the exception, ignore it, and keep looping until we have data. With this we'll constantly be checking for new connections and data as fast as we can. This should solve the issue that our blocking socket echo server had.

Listing 3.6 catching and ignoring blocking IO errors

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ('localhost', 8000)
server_socket.bind(server_address)
server_socket.listen()
server_socket.setblocking(False)

connections = []

try:
    while True:
        try:
            connection, client_address = server_socket.accept()
            connection.setblocking(False)
            print(f'I got a connection from {client_address}!')
            connections.append(connection)
        except BlockingIOError:
            pass

        for connection in connections:
            try:
                buffer = connection.recv(2)

                print(f'I got data: {buffer}!')

                while buffer[-2:] != b'\r\n':
                    data = connection.recv(2)
                    print(f'I got data: {data}!')
                    buffer = buffer + data

                print(f"All the data is: {buffer}")
                connection.send(buffer)
            except BlockingIOError:
```

```
            pass

finally:
    server_socket.close()
```

Each time we got through an iteration of our infinite loop, none of our calls to `accept` or `recv` ever block, we either instantly throw an exception that we ignore, or we have data ready to process and we process it. Each iteration of this loop happens quickly, and we're never dependent on anyone sending us data to proceed to the next line of code. This addresses the issue of our blocking server and allows multiple clients to connect and send data concurrently.

This approach works but comes at a cost. The first is code quality. Catching exceptions any time we might not yet have data will quickly get verbose and is potentially error prone. The second is a resource issue. If you run this on a laptop, you may notice your fan starts to get louder after a few seconds. This application will always be using nearly 100% of our CPU's processing power. This is because we are constantly looping and getting exceptions as fast as we can inside our application, leading to a workload that is CPU heavy.

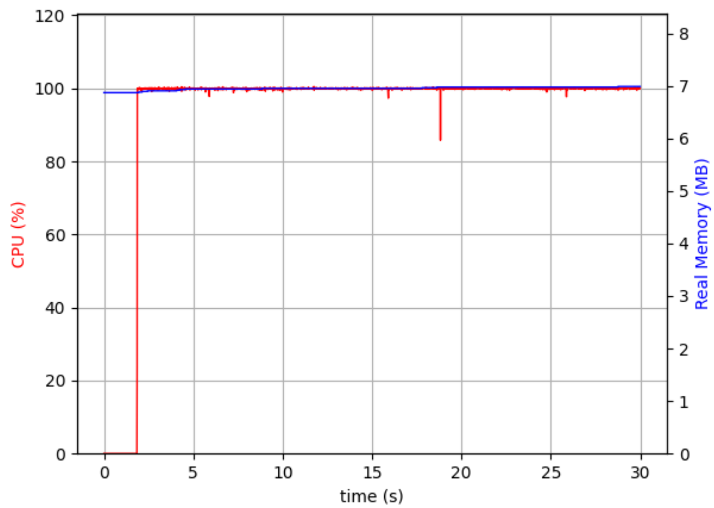


Figure 3.3 When looping and catching exceptions, CPU usage spikes to 100 percent and stays there.

Earlier in the book, we mentioned that operating systems have event notification systems that can notify us when sockets have data that we can act on. These systems rely on hardware-level notifications and don't involve polling with a while loop as we just did. Python has a library for using this event notification system built in. Next, we'll use this to resolve our CPU utilization issues and build a mini event loop for socket events.

3.3 Using the selectors module to build a socket event loop

Operating systems have efficient APIs that let us watch sockets for incoming data and other events built in. While the actual API is dependent on the operating system (kqueue, epoll and IOCP are a few common ones), all of these I/O notification systems operate on a similar concept. We give them a list of sockets we want to monitor for events, and instead of constantly checking each socket to see if it has data, the operating system tells us explicitly when sockets have data.

Because this is implemented at the hardware level, very little CPU is used during this monitoring allowing for efficient resource usage. These notification systems are the core of how asyncio achieves concurrency. Understanding how this works gives us a peek behind the curtain at how the underlying machinery of asyncio works.

The event notification systems are different depending on the operating system. Luckily for us Python's `selectors` module is abstracted such that we can get the proper event for wherever we run our code. This makes our code portable between different operating systems.

This library exposes an abstract base class called `BaseSelector` which has multiple implementations for each event notification system. It also contains a `DefaultSelector` class, which automatically chooses which implementation is most efficient for our system.

The `BaseSelector` class has a few important concepts. The first is registration. When we have a socket we're interested in getting notifications about, we register it with the selector and tell it which events we're interested in. These are events such as read and write. Inversely, we can also deregister a socket we're no longer interested in. The second major concept is `select`. Select will block until an event has happened, and once it does, the call will return with a list of sockets that are ready for processing along with the event that triggered it. It also supports a timeout, which will return an empty set of events after a specified amount of time.

Given these building blocks, we can create a non-blocking echo server that does not stress our CPU. Once we create our server socket, we'll register it with the default selector which will listen for any connections from clients. Then, any time someone connects to our server socket, we'll register the client's connection socket with the selector to watch for any data sent. If we get any data from a socket that isn't our server socket, we know it is from a client that has sent data. We then receive that data and write it back to the client. We will also add a timeout to demonstrate that we can have other code execute while we're waiting for things to happen.

Listing 3.7 using selectors to build a non-blocking server

```
import selectors
import socket
from selectors import SelectorKey
from typing import List, Tuple

selector = selectors.DefaultSelector()

server_socket = socket.socket()
server_address = ('localhost', 8000)
server_socket.setblocking(False)
server_socket.bind(server_address)
server_socket.listen()
```

```
selector.register(server_socket, selectors.EVENT_READ)

while True:
    events: List[Tuple[SelectorKey, int]] = selector.select(timeout=1) #A

    if len(events) == 0: #B
        print('No events, waiting a bit more!')

    for event, _ in events:
        event_socket = event.fileobj #C

        if event_socket == server_socket: #D
            connection, address = server_socket.accept()
            connection.setblocking(False)
            print(f"I got a connection from {address}")
            selector.register(connection, selectors.EVENT_READ)#E
        else:
            data = event_socket.recv(1024) #F
            print(f"I got some data: {data}")
            event_socket.send(data)
```

#A Create a selector that will timeout after a second
#B If there are no events print it out. This happens when a timeout occurs.
#C Get the socket for the event which is stored in the fileobj field
#D If the event socket is the same as the server socket, we know this is a connection attempt
#E Register the client that connected with our selector
#F If the event socket is not the server socket, receive data from the client and echo it back

When we run listing 3.7, we'll see 'No events, waiting a bit more!' printed roughly every second unless we get a connection event. Once we get a connection, we register that connection to listen for read events. Then, if a client sends us data, our selector will return an event that we have data ready and we can read it with `socket.recv`.

This is fully functioning echo server that supports multiple clients. This server has no issues with blocking as we only read or write data when we have data to act on. It also has very little CPU utilization as we're using the operating system's efficient event notification system.

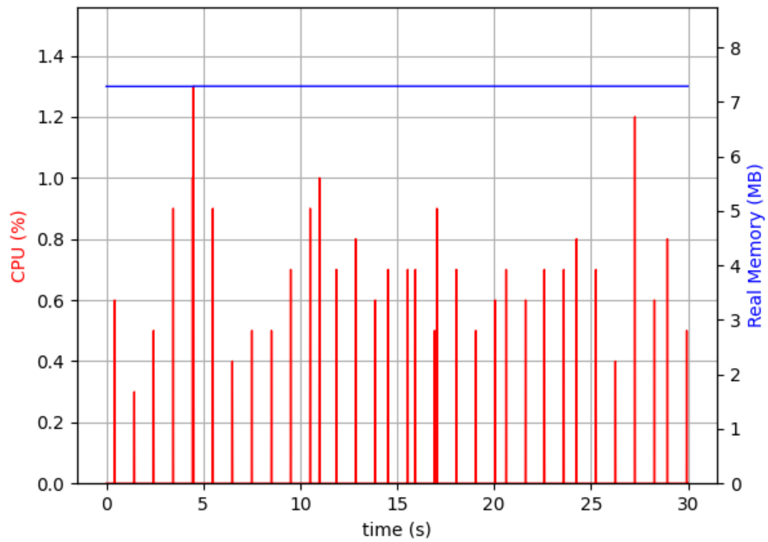


Figure 3.4 CPU graph of the echo server with selectors. Utilization hovers between 0 and 1 percent with this method.

What we've built is similar to a big part of what asyncio's event loop does under the hood. In this case, the 'events' we care about are sockets receiving data. Each iteration of our event loop and the asyncio event loop is triggered by either a socket event happening, or a timeout triggering an iteration of the loop. In the asyncio event loop, when any of these two things happen, coroutines that are waiting to run will do so until they either complete or they hit the next `await` statement. When we hit an `await` in a coroutine that utilizes a non-blocking socket, it will register that socket with the system's selector and keep track that the coroutine is paused waiting for a result. We can translate this into pseudocode

```
paused = []
ready = []

while True:
    paused, new_sockets = run_ready_tasks(ready)
    selector.register(new_sockets)
    timeout = calculate_timeout()
    events = selector.select(timeout)
    ready = process_events(events)
```

We run any coroutines that are ready to run until they are paused on an `await` statement and store those in the paused array. We also keep track of any new sockets we need to watch as a result of running those coroutines and register them with the selector. We then calculate the desired timeout for when we call `select`. While this timeout calculation is somewhat complicated, it is typically looking at things we have scheduled to run at a specific time or for a specific duration. An example of this is `asyncio.sleep`. We then call select and wait for any socket events or a timeout. Once either of those happen, we process those events and turn that into a list of coroutines that are ready to run.

While the event loop we've built is only for socket events, it demonstrates the main concept of using selectors to register sockets we care about, only being woken up when something we want to process happens. We'll get more in-depth with how to construct a custom event loop at the end of this book.

Now we understand a large part of the machinery that makes asyncio tick. However, if we just use selectors to build our applications we would wind up implementing our own event loop to achieve the same functionality as provided by asyncio. To see how to implement this with asyncio, let's take what we have learned and translate it into async / await code and use an event loop already implemented for us.

3.4 An echo server on the asyncio event loop

Working with select is a bit too low level for most applications. We may want to have code run in the background while we're waiting for socket data to come in or we may want to have background tasks run on a schedule. If we were to do this with only selectors, we'd wind up building our own event loop where asyncio has a nicely implemented one ready for us to use. In addition, coroutines and tasks provide abstractions on top of selectors, which make our code easier to implement and maintain as we don't need to think about selectors at all.

Now that we have a deeper understanding on how the asyncio event loop works, let's take the echo server that we built in the last section and build it again using coroutines and tasks. We'll still use lower-level sockets to accomplish this, but we'll use asyncio based APIs that return coroutines to manage them. We'll also add some more functionality to our echo server to demonstrate a few key concepts of how asyncio works.

3.4.1 Event loop coroutines for sockets

Given sockets are a relatively low-level concept, the methods for dealing with them are on asyncio's event loop itself. There are three main coroutines we'll want to work with, `sock_accept`, `sock_recv` and `sock_sendall`. These are analogous to the methods on socket that we were using earlier, except they take in a socket as an argument and return coroutines that we can await until we have data to act on.

Let's start with `sock_accept`, this coroutine is analogous to the `socket.accept` method that we saw in our first implementation. This method will return a tuple of a socket connection and a client address. We pass it in the socket we're interested in and we can then await the coroutine it returns. Once that coroutine completes, we'll have our connection and address. This socket must be non-blocking and should already be bound to a port.

```
connection, address = await loop.sock_accept(socket)
```

`sock_recv` and `sock_sendall` are called similarly to `sock_accept`. They take in a socket and we can then `await` for a result. `sock_recv` will await until a socket has bytes we can process. `sock_sendall` takes in both a socket and data we want to send and will wait until all data we want to send to a socket has been sent and will return `None` on success.

```
data = await loop.sock_recv(socket)
success = await loop.sock_sendall(socket, data)
```

With these building blocks, we'll be able to translate our previous approaches into one using coroutines and tasks.

3.4.2 Designing an asyncio echo server

In the previous chapter we introduced coroutines and tasks. So, when should we use just a coroutine and when should we wrap a coroutine in a task for our echo server? Let's examine how we want our application to behave to make this determination.

We'll start with how we want to listen for connections in our application. When we are listening for connections, we will only be able to process one connection at a time as `socket.accept` will only give us one client connection. Behind the scenes incoming connections will be stored in a queue known as the backlog if we get multiple at the same time, but for this chapter we won't get into how this works. Since we don't need to process multiple connections concurrently, a single coroutine that loops forever makes sense. This will allow other code to run concurrently while we're paused waiting for a connection. We'll define a coroutine called `listen_for_connections` that will loop forever and listen for any incoming connections.

```
async def listen_for_connections(server_socket: socket,
                                 loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Got a connection from {address}")
```

Now that we have a coroutine for listening to connections, how about reading and writing data to the clients who have connected? Should that be a coroutine or a coroutine we wrap in task? In this case, we will have multiple connections, each of which could send data to us at any time. We don't want waiting for data from one connection to block another, so we need to concurrently read and write data from multiple clients. Because we need to handle multiple connections at the same time, creating a task for each connection to read and write data makes sense. On every connection we get, we'll create a task to both read data from and write data to that connection.

We'll create a coroutine named `echo` that is responsible for handling data from a connection. This coroutine will loop forever listening for data from our client. Once it receives data it will then send it back to the client.

Then in `listen_for_connections` we'll create a new task that wraps our `echo` coroutine for each connection that we get. With these two coroutines defined, we now have all we need to build an asyncio echo server.

Listing 3.8 an asyncio echo server

```
import asyncio
import socket
from asyncio import AbstractEventLoop


async def echo(connection: socket,
```

```
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024): #A
        await loop.sock_sendall(connection, data) #B


async def listen_for_connection(server_socket: socket,
                                loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        asyncio.create_task(echo(connection, loop)) #C


async def main():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    await listen_for_connection(server_socket, asyncio.get_event_loop()) #D

asyncio.run(main())
```

#A Loop forever waiting for data from a client connection
#B Once we have data, send it back to that client
#C Whenever we get a connection, create an echo task to listen for client data.
#D Start the coroutine to listen for connections

The architecture for listing 3.8 looks like the following, we have one coroutine, `listen_for_connection`, listening for connections. Once a client connects our coroutine spawns an `echo` task for each client which then listens for data and writes it back out to the client.

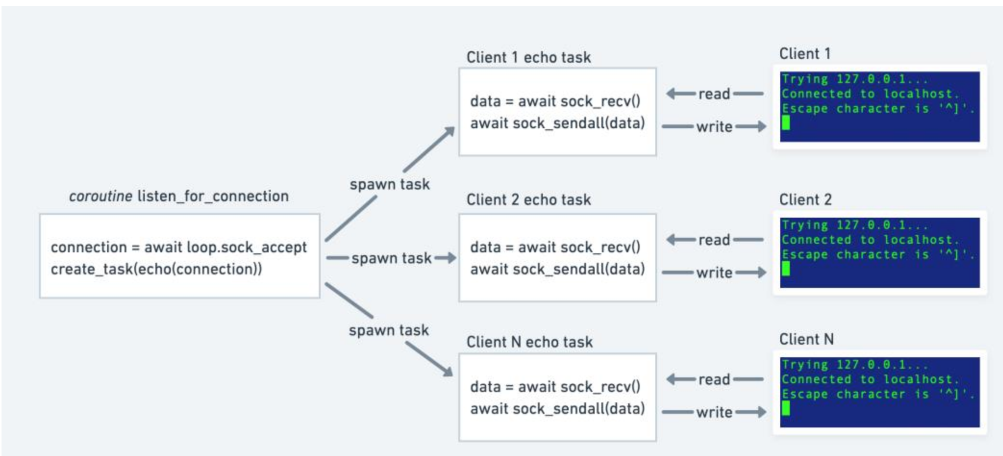


Figure 3.5 The coroutine listening for connections spawns one task per each connection it gets.

When we run this application, we'll be able to connect multiple clients concurrently and send data to them concurrently. Under the hood, this is all using selectors as we saw before, so our CPU utilization remains low.

We've now built a fully functioning echo server entirely using asyncio! So, is our implementation error free? It turns out the way we have designed this does have an issue when our echo task fails that we'll need to handle.

3.4.3 Handling errors in tasks

Network connections are often unreliable, and we may get exceptions we don't expect in our application code. How would our application behave if reading or writing to a client failed and threw an exception? To test this out, let's change our implementation of `echo` to throw an exception when a client passes us a specific keyword.

```
async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024):
        if data == b'boom\r\n':
            raise Exception("Unexpected network error")
        await loop.sock_sendall(connection, data)
```

Now, whenever a client sends "boom" to us, we will raise an exception and our task will crash. So, what happens when we connect a client to our server and send this message? We will see a traceback with a warning like so:

```
Task exception was never retrieved
future: <Task finished name='Task-2' coro=<echo() done, defined at asyncio_echo.py:5>
       exception=Exception('Unexpected network error')>
Traceback (most recent call last):
  File "asyncio_echo.py", line 9, in echo
    raise Exception("Unexpected network error")
Exception: Unexpected network error
```

The important part here is `Task exception was never retrieved`. What does this mean? When an exception is thrown inside a task, the task is considered done with its result as an exception. This means that no exception is thrown up the call stack. Furthermore, we have no cleanup here. If this exception is thrown, we can't react to the task failing because we never retrieved the exception.

In order to have the exception bubble up, we must use the task in an `await` expression. When we await a task that failed, the exception will get thrown where we perform the await and the traceback will reflect that. If we don't await a task at some point in our application, we run the risk of never seeing an exception that a task raised. While we did see the exception output in the example, which may lead us to think it isn't that big of a deal, there are subtle ways we could change our application so that we would never see this message.

As a demonstration of this, let's say instead of ignoring the echo tasks we create in `listen_for_connections`, we kept track of them in a list like so:

```
tasks = []

async def listen_for_connection(server_socket: socket,
                                loop: AbstractEventLoop):
```

```
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        tasks.append(asyncio.create_task(echo(connection, loop)))
```

One would expect this to behave in the same way as before. If we send the boom message, we'll see the exception printed along with the warning that we never retrieved the task exception. However, this isn't the case, we'll actually see nothing printed until we forcefully terminate our application!

This is because we've kept a reference around to the task. Asyncio can only print this message and the traceback for a failed task when that task is garbage collected. This is because it has no way to tell if that task will be awaited at some other point in the application and would therefore raise an exception then. Due to these complexities we'll either need to await our tasks at some point or handle all exceptions that our tasks could throw, so how do we do this in our echo server?

The first thing we can do to fix this is wrap the code in our echo coroutine in a try/catch statement, log the exception, and close the connection:

```
import logging

async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
        while data := await loop.sock_recv(connection, 1024):
            print('got data!')
            if data == b'boom\r\n':
                raise Exception("Unexpected network error")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()
```

This will resolve the immediate issue of an exception causing our server to complain that a task exception was never retrieved because we handle it in the coroutine itself. It will also properly shut down the socket within the `finally` block, so we won't be left with a dangling unclosed exception in the event of a failure.

It's important to note that this implementation will properly close any connections to clients we have open on application shutdown. Why is this? In the first chapter we noted that `asyncio.run` will cancel any tasks we have remaining when our application shuts down. We also learned when we cancel a task, a `CancelledException` is raised whenever we try to await it. The important thing here is noting where that exception is raised. If our task is waiting on a statement such as `await loop.sock_recv` and we cancel that task, a `CancelledException` is thrown from the `await loop.sock_recv` line. This means that in the above case our finally block will be executed, since we threw an exception on an await expression when we cancelled the task. If we change the exception block to catch and log these exceptions, you will see one `CancelledException` per each task that was created.

We've now handled the immediate issue of handling errors when our echo tasks fail. What if we want to provide some cleanup of any errors or leftover tasks when our application shuts down? We can do this with asynio's signal handlers.

3.5 Shutting down gracefully

Now we've created an echo server that handles multiple concurrent connections and also properly logs errors and cleans up when we get an exception. What happens if we need to shut down our application? Wouldn't it be nice if we could allow any in-flight messages to complete before we shut down? We can do this by adding custom shutdown logic to our application that allows any in-progress tasks a few seconds to finish sending any messages they might want to send. While this won't be a production worthy implementation, we'll learn the concepts around shutting down as well as cancelling all running tasks in our asyncio applications.

> **SIGNALS ON WINDOWS** Windows does not support signals and as such this section only applies to Unix based systems. Windows uses a fairly different system to handles this which at the time of writing does not play well with Python.

3.5.1 Listening for signals

Signals are a concept in Unix based operating systems for asynchronously notifying a process of an event that occurred at the operating system level. While this sounds very low level, you're probably already familiar with some signals. For instance, a common signal is SIGINT, short for "signal interrupt". This is triggered when you press CTRL+C to kill a command line application. In Python we can often handle this by catching the `KeyboardInterrupt` exception. Another common signal is SIGKILL, short for "signal kill". This is triggered when we run the `kill` command on a particular process to stop its execution.

To implement custom shutdown logic, we'll implement listeners in our application for both the SIGINT and SIGKILL signals. Then, in these listeners we'll implement logic to allow any echo tasks we have a few seconds to finish.

How do we listen for signals in our application? The asyncio event loop lets us directly listen for any event we specify with the `add_signal_handler` method. This differs from the signal handlers that you can set in the signal module with the `signal.signal` function in that `add_signal_handler` can safely interact with the event loop. This function takes in a signal we want to listen for and a function that we'll call when our application receives that signal. To demonstrate this let's take a look at adding a signal handler that cancels all currently running tasks. Asyncio has a convenience function that returns a set of all running tasks named `asyncio.all_tasks`.

Listing 3.9 adding a signal handler to cancel all tasks

```
import asyncio, signal
from asyncio import AbstractEventLoop
from typing import Set

from util.delay_functions import delay
```

```
def cancel_tasks():
    print('Got a SIGINT!')
    tasks: Set[asyncio.Task] = asyncio.all_tasks()
    print(f'Cancelling {len(tasks)} task(s).')
    [task.cancel() for task in tasks]


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

    loop.add_signal_handler(signal.SIGINT, cancel_tasks)

    await delay(10)


asyncio.run(main())
```

When we run this application, we'll see that our delay coroutine starts right away and waits for ten seconds. If we press CTRL+C within these 10 seconds we should see `got a SIGINT!` printed out, followed by a message that we're canceling our tasks. We should also see a `CancelledError` thrown from `asyncio.run(main())` since we've canceled that task.

3.5.2 Waiting for pending tasks to finish

In the original problem statement, we wanted to give our echo server's echo tasks a few seconds to keep running before shutting down. One way for us to do this is to wrap all of our echo tasks in a `wait_for` and then `await` those wrapped tasks. Those tasks will then throw a `TimeoutError` once the timeout has elapsed and we can terminate our application.

One thing you'll notice right away about our shutdown handler is that this is a normal Python function, so we can't run any `await` statements inside of it. This poses a problem for us since our proposed solution involves `await`. One possible solution is to just create a coroutine that does our shutdown logic, and in our shutdown handler wrap it in a task:

```
async def await_all_tasks():
    tasks = asyncio.all_tasks()
    [await task for task in tasks]


async def main():
    loop = asyncio.get_event_loop()
    loop.add_signal_handler(signal.SIGINT,
                            lambda: asyncio.create_task(await_all_tasks()))
```

An approach like this will work, but the drawback is that if something in `await_all_tasks` throws an exception we'll be left with an orphaned task that failed and a "exception was never retrieved" warning. So, is there a better way to do this? We can deal with this by raising a custom exception to stop our main coroutine from running. Then we can catch this exception when we run our main coroutine and run any shutdown logic. To do this we'll need to create an event loop ourselves instead of using `asyncio.run`. This is because on an exception `asyncio.run` will cancel all running tasks, which means we aren't able to wrap our echo tasks in a `wait_for`.

```
class GracefulExit(SystemExit):
    pass

def shutdown():
    raise GracefulExit()

loop = asyncio.get_event_loop()

loop.add_signal_handler(signal.SIGINT, shutdown)

try:
    loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

With this approach in mind, let's write our shutdown logic.

```
async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_tasks]
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # We expect a timeout error here
            Pass
```

In `close_echo_tasks` we take a list of echo tasks and wrap them all in a `wait_for` with a two second timeout. This means that any echo tasks will have two seconds to finish before we cancel them. Once we've done this, we loop over all these wrapped tasks and await them. We catch any `TimeoutErrors` as we expect this to be thrown from our tasks after two seconds. Taking all these parts together, our echo server with shutdown logic looks like listing 3.10:

Listing 3.10 a graceful shutdown

```
import asyncio
from asyncio import AbstractEventLoop, Task
import socket
import logging
import signal
from typing import List


async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
        while data := await loop.sock_recv(connection, 1024):
            print('got data!')
            if data == b'boom\r\n':
                raise Exception("Unexpected network error")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()
```

```
async def connection_listener(server_socket, loop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        yield asyncio.create_task(echo(connection, loop))


class GracefulExit(SystemExit):
    pass


def shutown():
    raise GracefulExit()


async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_tasks]
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # We expect a timeout error here
            pass


echo_tasks = []


async def listen_for_connections(server_socket, loop):
    async for echo_task in connection_listener(server_socket, loop):
        echo_tasks.append(echo_task)


async def main():
    server_socket = socket.socket()
    server_address = ('localhost', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(getattr(signal, signame), shutown)
    await listen_for_connections(server_socket, loop)


loop = asyncio.new_event_loop()

try:
    loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

Assuming we have at least one client connected, if we stop this application with either CTRL+C or we issue a kill command to our process, our shutdown logic will execute. We will see the application wait for two seconds while it allows our echo tasks some time to finish before it actually stops running.

There are a couple of reasons why this is not a production worthy shutdown. The first is we don't shut down our connection listener while we're waiting for our echo tasks to complete. This means that as we're shutting down a new connection could come in and they we won't be able to add a two second shutdown. The other problem is that in our shutdown logic we await every echo task we're shutting down and only catch `TimeoutExceptions`. This means that if one of our tasks threw something other than that, we would bubble that exception up and any other subsequent tasks that may have had an exception will be swallowed. In the next chapter, we'll see some asyncio methods for more gracefully handling failures from a group of awaitables.

While our application isn't perfect and is a toy example, we've built a fully functioning server using asyncio. This server can handle many users concurrently all within one single thread. With a blocking approach we saw earlier, we would need to turn to threading to be able to handle multiple clients, adding complexity and increased resource utilization to our application.

3.6 Summary

In this chapter we've learned about blocking and non-blocking sockets and have explored more in depth how the asyncio event loop functions. We've also made our first application with asyncio, a highly concurrent echo server and have examined how to handle errors in tasks and add custom shutdown logic in our application.

- We've learned how to create simple applications with blocking sockets. Blocking sockets will stop the entire thread when they are waiting for data. This prevents us from achieving concurrency because we can only get data from one client at a time.
- We've learned how to build applications with non-blocking sockets. These sockets will always return right away, either with data because we have it ready, or with an exception stating we have no data. These sockets let us achieve concurrency because their methods never block and return instantly.
- We've learned how to use the selectors module to listen for events on sockets in an efficient manner. This library lets us register sockets we want to keep tabs on and will tell us when a non-blocking socket is ready with data.
- If we put select in an infinite loop, we've replicated the core of what the asyncio event loop does. We register sockets we are interested in and we loop forever, running any code we want once a socket has data available to act on.
- We learned how to use asyncio's event loop methods to build applications with non-blocking sockets. These methods take in a socket and return a coroutine which we can then use this in an await expression. This will suspend our parent coroutine until the socket has data. Under the hood this is using the selectors library.
- We've seen how to use tasks to achieve concurrency for an asyncio based echo server with multiple clients sending and receiving data at the same time. We've also examined how to handle errors within those tasks.
- We've learned how to add custom shutdown logic to an asyncio application. In our case we decided that when our server shuts down, we'd give a few seconds for any remaining

clients to finish sending data. Using this knowledge, we can add any logic our application needs when it is shutting down.

4

Building a concurrent web crawler

This chapter covers

- **Asynchronous context managers**
- **How to make 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**
- **Cancelling in-flight requests**

In the previous chapter 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 at the same time, cutting down on our application's runtime compared to a synchronous approach. This is useful for when we have to 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 which 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 concurrently, 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 take a 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-flight requests based on how other requests have performed. These API methods are useful not only for making web requests, but for any time 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.

4.1 Introducing Aiohttp

In chapter two, we mentioned 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 play well with asyncio because under the hood 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 grind to a halt until that request finishes.

In order to address this issue and truly get concurrency, we need to use a library that is non-blocking all the way down to the socket layer. Aiohttp 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". This library is a fully functioning web client as well as a web server, meaning it can both make web requests and we can create our own async web servers using it. Documentation for the library is available at https://docs.aiohttp.org/en/stable/. In this chapter, we'll focus on the client side of Aiohttp, but we will see how to build web servers with it in later chapters.

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

4.2 Asynchronous context managers

In any programming language, dealing with resources that need to be opened and then closed such as files are common. When dealing with these resources, one needs 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, leaving us in a state with leaking resources. Dealing with this in Python is straightforward with 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 here is we need to remember to wrap everything in a try finally and we also need to remember which methods to call to properly close our resource. This isn’t too bad for files as we just need to remember to close, 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 away the shutdown logic along with the try/finally block.

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

This is a lot cleaner and is the Pythonic way to manage files. If an exception is thrown in the `with` block, our file will automatically get closed for us. This works for synchronous resources, but what if we want to asynchronously use a resource with this syntax? In this case, the `with` context manager syntax we just mentioned won’t work as it is only designed to work 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 synchronous context managers with the difference 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 in particular takes several arguments that deal with any exceptions that occurred which we won’t go over in this chapter.

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

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:
    _connection = None

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

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

    async def __aexit__(self,
                        exc_type: Optional[Type[BaseException]],
                        exc_val: Optional[BaseException],
                        exc_tb: Optional[TracebackType]): #B
```

```
        print('Exiting context manager')
        self._connection.close()
        print('Closed connection')

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

    server_socket = socket.socket()
    server_address = ('localhost', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

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

asyncio.run(main())
```

#A This coroutine is called when we enter the with block. It waits until a client connects and returns the connection.
#B 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.
#C This calls __aenter__ and waits for a client connection
#D After this statement, __aenter__ will execute and we'll close our connection.

In listing 4.1 we create 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 our `__aexit__` coroutine runs and we close the connection. Assuming a client connects with telnet and sends us some test data, we should see output similar to 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 also use this in future chapters when dealing with async database connections and transactions. Normally, you won't need to write your own async context managers but having an understanding how they work and are different from normal context managers is helpful.

Now that we've introduced context managers and how they work, let's use them with aiohttp to see how to make a single asynchronous web request.

4.3 Making a web request with Aiohttp

To get started, we'll first need to install the Aiohttp library. We can do this with pip by running the following:

```
pip install -Iv aiohttp==3.6.2
```

This will install the latest version of Aiohttp at the time of this writing which is 3.6.2. Once you've done this, you're ready to start making requests.

Aiohttp, and web requests in general have the concept of a session. You can think of a session like opening a new browser window. Within a new browser window, you'll make connections to any number of webpages, 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 expensive, creating a reusable pool of them cuts down on resource allocation costs. A session will also internally save any cookies that we get, though this functionality can be turned off if desired.

Typically, we want to take advantage of connection pooling, so most aiohttp based applications will have one session for the entire application. This session object is then passed around to methods where needed. A session object has methods on it for making any number of web requests we'd like, 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 a single 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) -> int:
    async with session.get(url) as result:
        return result.status


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'http://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 the output `Status for http://www.example.com was 200`. In listing 4.2 we first create a client session in an `async with` block with `aiohttp.ClientSession()`. Once we have a client session, we're now free to make any web request we'd like. In this case, we define a convenience method `fetch_status_code` which 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 our `with` block. In this case we just grab the status code and return.

We'll reuse `fetch_status` thoughout the chapter, so we'll want to 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`.

4.3.1 Setting 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 individual 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 more than five minutes. This is a large timeout, and many applications 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 the individual 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 both 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 result:
        return result.status


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

asyncio.run(main())
```

In listing 4.3 we set two timeouts. The first is at the client session level. Here we set a total timeout of one second and explicitly set a connection timeout of one hundred milliseconds. Then, in `fetch_status` we override this for our get request to set a total timeout of ten miliseconds. In this instance if our request to example.com takes more than ten milliseconds, an `asyncio.TimeoutError` will be raised when we `await fetch_status`. In this particular example, 10 milliseconds should usually be enough time for the request to example.com to complete, so you're not likely to see an exception. If you'd like to see this exception, change the URL to a page that takes a bit longer than 10 milliseconds to download.

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

4.4 Running tasks concurrently revisited

In the first few chapters we saw how to create multiple tasks to run coroutines concurrently. To do this, we used `asyncio.create_task` and then awaited the task like 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 above where we have one or two coroutines we want to kick off concurrently. However, in a world where we may make hundreds, thousands or even more web requests concurrently this style would get verbose and messy.

One may be tempted to utilize a for loop or a list comprehension to make this a little nicer as demonstrated in listing 4.4. However, this approach can cause issues if not done 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 in delay_times]

asyncio.run(main())
```

Given we ideally want our delay tasks to run concurrently, we'd expect our main method to complete in about three seconds. However, in this case we actually take nine seconds to run as 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, but it is because we use `await` as soon as we create the task. This means that we pause the list comprehension and our main coroutine for every single delay task we create until that delay task completes. In this case, we will only have one task running at any given time instead of multiple 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 with a list comprehension concurrently

```
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 in delay_times]
    [await task for task in tasks]

asyncio.run(main())
```

This code creates a list 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 our tasks have been created. This will ensure that we only take at most the maximum delay time in `delay_times`, giving us a runtime of about three 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 now does what we want, it still has a couple of drawbacks. The first is this is multiple lines of code where we have to explicitly remember to separate out our task creation from our awaits. The second is that it is inflexible, if one of our coroutines finishes a long time before the rest, we'll be stuck in the second list comprehension waiting for all our other coroutines to finish. While this may be fine in certain circumstances, we may want to be more responsive, processing our results as soon as they come in. The third, and potentially biggest issue, is exception handling. If one of our coroutines has an exception it will get 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 of these situations and more. These functions are the recommended approach when running multiple tasks concurrently. In the next few sections, we'll look at a few of them and examine how to use them in the context of making multiple web requests concurrently.

4.5 Running requests concurrently with gather

Perhaps one of the most 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 awkwardly wrap everything with `asyncio.create_task` separately as we saw in the last section.

`asyncio.gather` returns an awaitable itself. 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, it will return a list of all the completed results.

We can use this function to run as many web requests as we'd like concurrently. To illustrate this, let's look at an example where we make 1000 requests at the same time and grab the status code of each response. We'll decorate our main coroutine with `@async_timed` so we understand how long things are taking.

Listing 4.6 multiple 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 urls] #A
        status_codes = await asyncio.gather(*requests) #B
        print(status_codes)


asyncio.run(main())
```

#A Generate a list of coroutines for each request we want to make
#B Wait for all requests to complete

In listing 4.6 we first generate a list of URLs we'd like to get the status code from, in this case we'll just request example.com repeatedly for simplicity. We then take that list of URLs and call `fetch_status_code` to generate a list of coroutines which 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 1000 messages printed saying that the `fetch_status_code` coroutines started one after another, indicating we've started 1000 requests concurrently. As results start to come in, we'll see messages like `finished <function fetch_status_code at 0x10f3fe3a0> in 0.5453 second(s)` start to pour in. Once we retrieve the contents of all the URLs we've requested, we'll see our status codes start to print out. This process is fairly quick, depending on your internet connection and speed of your machine, this script can finish in as little as a five to six hundred milliseconds.

So how does this compare with doing things synchronously? It is easy to adapt our main function so that it blocks on each request by using an `await` when we call `fetch_status_code`. This will pause our main coroutine for each URL we have, 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) for url in urls]
        print(status_codes)
```

If we run this, we'll notice that things start to take a lot longer. We'll also notice instead of getting 1000 "`starting function fetch_status_code`" messages followed by 1000 "`finished function fetch_status_code`" we'll start to see something like the following for each request:

```
starting <function fetch_status_code at 0x10d95b310>
finished <function fetch_status_code at 0x10d95b310> in 0.01884 second(s)
```

This indicates that we're making requests 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 our async version? While this of course 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, we're a pretty impressive 33 times faster.

It is worthwhile to note 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 quickly by taking a 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 listing 4.7 we pass two coroutines to `gather`, the first takes three seconds to complete and the second takes one second. We may expect the result of this to be `[1, 3]` since our one second long coroutine finishes before our three second coroutine, but the result is actually `[3, 1]` – the order we passed things in. The `gather` function keeps result ordering nice and deterministic for us despite the inherent nondeterminism behind the scenes. Under the hood – gather uses a special kind of future implementation to do this. For the curious reader, looking at the source code of `gather` can be an instructive way to understand how many asyncio APIs are built using futures.

In the past examples we've looked at we've assumed none of our requests will fail or throw an exception. This works fine for the happy path, but what happens when a request fails?

4.5.1 Handling 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, there are quite a few different failure cases we may encounter. 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 and therefore 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 cancelled 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 any and all exceptions as we wish.

To illustrate how these options work, let's change our url list to contain an invalid url. 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 case of `return_exceptions` behaves.

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com', 'python://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        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, explaining 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 (the traceback has been shortened 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 0.0004 second(s)
finished <function main at 0x107f4a4c0> in 0.0203 second(s)
finished <function fetch_status_code at 0x107f4a3a0> in 0.0198 second(s)
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 fine 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 there is more than one exception that 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 we need. 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://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        results = await asyncio.gather(*tasks, return_exceptions=True)

        exceptions = [res for res in results if isinstance(res, Exception)]
        successful_results = [res for res in results if not isinstance(res, Exception)]

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

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

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

This solves the issue of not being able to see all of 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 don't throw an exception when we `await` anymore. It is a little clunky that we have to filter out exceptions from successful results, but the API is not perfect.

Gather has a few drawbacks. The first, which we've already mentioned, is it isn't really easy to cancel our tasks if one throws an exception. One could imagine a case where we're making requests to the same server, and if one request fails, all others will as well such as hitting 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 under the hood.

The second is we have to wait for *all* of 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 take 100 milliseconds, but another that takes 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 of these issues. Let's start by taking a look at the problem of handling results as soon as they come in.

4.6 Processing requests as they complete

While `asyncio.gather` will work for many cases but it has the drawback that it waits for all awaitables to finish before allowing us to access any results. This is a problem for us 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 which 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 have 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 what will complete first.

To show how this works we'll simulate a case where one request completes quickly, and another takes a bit of time. We'll add a `delay` parameter to our `fetch_status` function and call `asyncio.sleep` to simulate a long request like so:

```
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.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 10)]

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


asyncio.run(main())
```

In listing 4.8 we create three coroutines, two which will take about a second to complete and one which will take ten. We then pass these into `as_completed`. Under the hood, each coroutine is wrapped in a task and starts running concurrently. We're then instantly returned an iterator which we start 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 a second and we print the status code. Then we hit `await result` again and since our requests were running concurrently, we should see the second result almost instantly. Finally, our ten 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 second(s)
200
finished <function fetch_status at 0x10dbed4c0> in 1.1294 second(s)
200
finished <function fetch_status at 0x10dbed4c0> in 10.0345 second(s)
200
finished <function main at 0x10dbed5e0> in 10.0353 second(s)
```

In total, iterating over `result_iterator` still takes about ten 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 gives us some better control over exception handling. When a task throws an exception, we'll be able to process it exactly when it happens, as the exception is thrown when we await the future.

4.6.1 Timeouts with as_completed

Any web-based request runs the risk of taking a long time. A server could be under 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 our `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 takes ten seconds to complete and one request that takes one second. Then we'll set a timeout of two 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.com', 1),
                    fetch_status(session, 'https://example.com', 10),
                    fetch_status(session, 'https://example.com', 10)]

        for done_task in asyncio.as_completed(fetchers, timeout=2):
            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)


asyncio.run(main())
```

When we run this, we'll notice that we see the result from our first fetch and after two seconds we'll see that we got two timeout errors. We'll also see that two of our fetches are still running, giving output similar to the following:

```
starting <function main at 0x109c7c430> with args () {}
200
We got a timeout error!
We got a timeout error!
finished <function main at 0x109c7c430> in 2.0055 second(s)
<Task pending name='Task-2' coro=<fetch_status_code()>>
<Task pending name='Task-1' coro=<main>>
<Task pending name='Task-4' coro=<fetch_status_code()>>
```

`as_completed` works well for getting our results as fast as possible but has a few drawbacks. The first is while we get results as they come in, there isn't any way for us to easily tell which coroutine or task we're awaiting as the order is completely nondeterministic. If we really don't care about order, this may be fine but if we need to associate the results to the requests we made in any way, we're left with a challenge. The second is that with timeouts while we will correctly throw an exception and move on, any tasks we created will still be running in the background. Since it is hard to figure out which tasks are still running if we want to cancel them, we are stuck with another challenge. If these are problems we need to deal with, then we'll need some finer grained knowledge of what awaitables are finished and which are not. To handle this situation, asyncio provides another API function called `wait`.

4.7 Finer grained control with wait

One of the drawbacks of both `gather` and `as_completed` is that there was no easy way for us to cancel tasks that were already running when we saw an exception. This might be fine for many situations, but one can imagine a use case where we make several coroutine calls and if the first one fails, the rest will as well. An example of this would be us passing in an invalid parameter to a web request or hitting an API rate limit. This has the potential to cause performance issues as we'll consume more resources by having more tasks running than we need. Another drawback we noted with `as_completed` is that it is a challenge to keep track of exactly which task had completed as the iteration order is nondeterministic.

Asyncio has a function similar to `gather` named `wait` that gives us some finer grained control to handle these situations. This method has several options to choose from depending on *when* exactly we want our results. In addition, this method returns two sets, a set of tasks that are finished with either a result or an exception and a set of tasks that are still running. This function also allows us to specify a timeout which behaves a bit differently from how we have seen our other API methods operate in that it does not throw exceptions. When it is needed, this function can solve some of the issues we noted with the other asyncio API functions we've used so far.

The basic signature of `wait` is a list of awaitable objects, followed by an optional timeout and an optional `return_when` string. This string has a few predefined values that we'll examine, `ALL_COMPLETED`, `FIRST_EXCEPTION` and `FIRST_COMPLETED`. It defaults to `ALL_COMPLETED`. While as of this writing `wait` takes a list of awaitables, it will change in the next few versions of Python to only accept `Task` objects. We'll see why at the end of this section, but for these code samples we'll wrap all of our coroutines in Tasks as this is best practice.

4.7.1 Waiting for all tasks to complete

This option is the default behavior if `return_when` is not specified and it is the closest in behavior to `asyncio.gather`, though it has a few subtle differences. As implied, using this option will wait for all tasks to finish before returning. Let's adapt this to our example of making multiple web requests concurrently to learn how this function works.

Listing 4.10 examining the default behavior of wait

```
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.com'),
                    fetch_status(session, 'https://example.com')]
        done, pending = await asyncio.wait(fetchers)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            result = await done_task
            print(result)


asyncio.run(main())
```

In listing 4.10 we run two web requests concurrently by passing a list of coroutines to `wait`. When we `await wait` it will return two sets once all requests finish, one set of all tasks that are complete and one set of the tasks that are still running. The `done` set contains all tasks that finished either successfully or with exceptions. The `pending` set contains all tasks that have not finished yet. In this instance, since we are using the `ALL_COMPLETED` option the pending set will always be zero, since `asyncio.wait` won't return until everything is completed. This will give us the following output:

```
starting <function main at 0x10124b160> with args () {}
Done task count: 2
Pending task count: 0
200
200
finished <function main at 0x10124b160> in 0.4642 second(s)
```

If one of our requests throws an exception, it won't be thrown at the `asyncio.wait` call in the same way that `asyncio.gather` did. In this instance, we'll still get the both the done and pending sets as before, but we won't see an exception until we await the task in `done` that failed. We have a few options on how to handle exceptions with this paradigm. We can use `await` and let the exception throw, we can use `await` and wrap it in a `try except` block to handle the exception, or we can use the `task.result()` and `task.exception()` methods. We can safely call these methods since our tasks in the done set are guaranteed to be completed tasks, if they were not calling these methods would produce an exception. Let's say that we don't want to throw an exception and have our application crash. Instead, we'd just like to print the task's result if we have it and log an error if there was an exception. In this case, using the methods on the `Task` object is an appropriate solution. Let's see how to use these two Task methods to handle exceptions.

Listing 4.11 exceptions with wait

```
import asyncio
import logging

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        good_request = fetch_status(session, 'https://www.example.com')
        bad_request = fetch_status(session, 'python://bad')

        fetchers = [asyncio.create_task(good_request),
                    asyncio.create_task(bad_request)]

        done, pending = await asyncio.wait(fetchers)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            # result = await done_task will throw an exception
            if done_task.exception() is None:
                print(done_task.result())
            else:
                logging.error("Request got an exception",
                              exc_info=done_task.exception())


asyncio.run(main())
```

Using `done_task.exception()` will let us check to see if we have an exception. If we don't have one, then we can proceed to get the result from `done_task` with the `result` method. It would also be safe to do `result = await done_task` here, though might throw an exception which may not be what we want. If exception is not None then we know that the awaitable had an exception and we can handle that as we like. Here we just print out the exception's stack trace. Running this will yield output similar to the following, we've removed the verbose traceback for brevity:

```
starting <function main at 0x10401f1f0> with args () {}
Done task count: 2
Pending task count: 0
200
finished <function main at 0x10401f1f0> in 0.12386679649353027 second(s)
ERROR:root:Request got an exception
Traceback (most recent call last):
AssertionError
```

4.7.2 Watching for exceptions

The drawbacks of `ALL_COMPLETED` are similar to the drawbacks we saw with `gather`. We could have any number of exceptions while we wait for other coroutines to complete which we won't see until all tasks complete. This could be an issue if as a result of one exception we'd like to cancel other running requests. We may also want to immediately handle any errors to ensure responsiveness and continue waiting for other coroutines to complete.

To support these use cases `wait` supports the `FIRST_EXCEPTION` option. When we use this option, we'll get two different behaviors depending on if any of our tasks throw exceptions.

NO EXCEPTIONS FROM ANY AWAITABLES

If we have no exceptions from any of our tasks, then this option is equivalent to `ALL_COMPLETED`. We'll wait for all tasks to finish and then the done set will contain all finished tasks and the pending set will be empty.

ONE OR MORE EXCEPTION FROM A TASK

If we any task throws an exception, `wait` will immediately return once that exception is thrown. The done set will have any coroutines that finished successfully alongside any coroutines that had exceptions. The done set is, at minimum, guaranteed to have one failed task in this case, but may have other successfully completed tasks. The pending set may be empty but may also have tasks that are still running. We can then use this pending set to manage the currently running tasks as we desire.

To illustrate how wait behaves in these scenarios, we'll first take a look at what happens when we have a couple of long running web requests we'd like to cancel when one coroutine fails immediately with an exception.

Listing 4.12 canceling running requests on an exception

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

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [asyncio.create_task(fetch_status(session, 'python://bad.com')),
                    asyncio.create_task(fetch_status(session, 'https://www.example.com',
       delay=3)),
                    asyncio.create_task(fetch_status(session, 'https://www.example.com',
       delay=3))]

        done, pending = await asyncio.wait(fetchers, return_when=asyncio.FIRST_EXCEPTION)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            if done_task.exception() is None:
                print(done_task.result())
            else:
                logging.error("Request got an exception",
                              exc_info=done_task.exception())

        for pending_task in pending:
            pending_task.cancel()


asyncio.run(main())
```

In listing 4.12 we make one bad request and two good ones which each take three seconds. When we await our wait statement, we return almost immediately as our bad request errors out right away. Then we loop through the done tasks. In this instance we'll only have one in the done set since our first request ended with an exception immediately. For this, we'll just execute the branch that prints the exception.

The pending set will have two elements as we have two requests that take roughly three seconds to run each and our first request failed almost instantly. Since we want to stop these from running, we can call the `cancel` method on them. This will give us the following output:

```
starting <function main at 0x105cfd280> with args () {}
Done task count: 1
Pending task count: 2
finished <function main at 0x105cfd280> in 0.0044 second(s)
ERROR:root:Request got an exception
```

Note that our application took almost no time to run as we quickly reacted to the fact that one of our requests threw an exception. The power of using this option is we achieve fail fast behavior, quickly reacting to any issues that arise.

4.7.3 Processing results as they complete

Both `ALL_COMPLETED` and `FIRST_EXCEPTION` have the drawback that in the case where coroutines are successful and don't throw an exception, we have to wait for all coroutines to complete. This may be fine depending on the use case but if we're in a situation where we want to respond to a coroutine as soon as it completes successfully, we are out of luck.

In the instance where we want to react to a result as soon as it completes, we could use `as_completed`, however, the issue with `as_completed` is there is no easy way to see which tasks are remaining versus which tasks have completed. We only get them one at a time through an iterator.

The good news is the `return_when` parameter accepts a `FIRST_COMPLETED` option. This option will make the `wait` coroutine return as soon as it has at least one result. This can either be a coroutine that failed or one that ran successfully. This lets us either cancel the other running coroutines or adjust which ones to keep running depending on our use case. Let's use this option to make a few web requests and process whichever one finishes first.

Listing 4.13 processing as they complete

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


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        fetchers = [asyncio.create_task(fetch_status(session, url)),
                    asyncio.create_task(fetch_status(session, url)),
                    asyncio.create_task(fetch_status(session, url))]

        done, pending = await asyncio.wait(fetchers, return_when=asyncio.FIRST_COMPLETED)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            print(await done_task)


asyncio.run(main())
```

In this example we start three requests concurrently. Our wait coroutine will return once any of these requests completes. This means that `done` will have one complete request and `pending` will contain anything still running, giving us the following output:

```
starting <function main at 0x10222f1f0> with args () {}
Done task count: 1
Pending task count: 2
200
finished <function main at 0x10222f1f0> in 0.1138 second(s)
```

These requests can complete at nearly the same time, so you could also see output that says two or three tasks are done. Try running this listing a few times to see how the result varies.

This approach lets us respond right away when our first task completes. What if we want to process the rest of the results as they come in similar to `as_completed`? The above example can be adopted fairly easily to loop on the pending tasks until they are empty. This will give us behavior similar to `as_completed` with the benefit that at each step we know exactly which tasks have finished and which are still running.

Listing 4.14 Processing all results as they come in

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


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        pending = [asyncio.create_task(fetch_status(session, url)),
                   asyncio.create_task(fetch_status(session, url)),
                   asyncio.create_task(fetch_status(session, url))]

        while pending:
            done, pending = await asyncio.wait(pending, return_when=asyncio.FIRST_COMPLETED)

            print(f'Done task count: {len(done)}')
            print(f'Pending task count: {len(pending)}')

            for done_task in done:
                print(await done_task)

asyncio.run(main())
```

In listing 4.14 we create a set named `pending` which we initialize to the coroutines we want to run. We loop while we have items in the `pending` set and call `wait` with that set on each iteration. Once we have a result from `wait`, we update the `done` and `pending` sets and then print out any done tasks. This will give us behavior similar to `as_completed` with the difference being we have better insight into which tasks are done and which tasks are still running. Running this, we'll see the following output:

```
starting <function main at 0x10d1671f0> with args () {}
Done task count: 1
Pending task count: 2
200
Done task count: 1
Pending task count: 1
200
Done task count: 1
Pending task count: 0
200
finished <function main at 0x10d1671f0> in 0.1153 second(s)
```

Since the request function may complete fairly quickly, such that all requests complete at the same time, it's not impossible that we see output similar to this as well:

```
starting <function main at 0x1100f11f0> with args () {}
Done task count: 3
Pending task count: 0
200
```

```
200
200
finished <function main at 0x1100f11f0> in 0.1304 second(s)
```

4.7.4 Handling timeouts

In addition to allowing us finer-grained control on how we wait for coroutines to complete, wait also allows us to set timeouts to specify how long we want to wait for all awaitables to complete. To enable this, we can set the `timeout` parameter with the maximum number of seconds we want to wait. If we've exceeded this timeout, wait will return both the done and pending task set.

There are a couple of differences in how timeouts behave in `wait` as compared to what we have seen thusfar with `wait_for` and `as_completed`.

COROUTINES ARE NOT CANCELED

When we used `wait_for` if our coroutine timed out it would automatically request cancellation for us. This is not the case with `wait` - it behaves closer to what we saw with `gather` and `as_completed`. In the case we want to cancel coroutines due to a timeout, we must explicitly loop over the tasks and cancel them.

TIMEOUT ERRORS ARE NOT RAISED

`wait` does not rely on exceptions in the event of timeouts as `wait_for` and `as_completed` do. Instead, if the timeout occurs the wait returns all tasks done and all tasks that are still pending up to that point when the timeout occurred.

For example, let's examine a case where we two requests complete quickly and one takes a few seconds. We'll use a timeout of one second with `wait` to understand what happens when we have tasks that take longer than the timeout. For the `return_when` parameter we'll use the default value of `ALL_COMPLETED`.

Listing 4.15 using timeouts with wait

```python
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        example_url = 'https://example.com'
        fetchers = [fetch_status(session, example_url),
                    fetch_status(session, example_url),
                    fetch_status(session, example_url, delay=3)]

        done, pending = await asyncio.wait(fetchers, timeout=1)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            result = await done_task
            print(result)


asyncio.run(main())
```

Running listing 4.15 our `wait` call will return our done and pending sets after one second. In the done set we'll see our two fast requests as they finished within one second. Our slow request is still running and is therefore in the pending set. We then await the done tasks to extract out their return values. We also could have canceled the pending task if we had desired to. Running this code, we will see the following output:

```
starting <function main at 0x11c68dd30> with args () {}
Done task count: 2
Pending task count: 1
200
200
finished <function main at 0x11c68dd30> in 1.0022 second(s)
```

Note that as before our tasks in the pending set are not canceled and will continue to run despite the timeout. If we have a use case where we want to terminate the pending tasks, we'll need to explicitly loop through the pending set and call `cancel` on each task.

4.7.5 Why wrap everything in a Task?

At the start of this section, we mentioned that it is best practice to wrap the coroutines we pass into wait in Tasks. Why is this? Let's go back to our previous timeout example and change it a little bit. Let's say that we have requests to two different web APIs that we'll call API A and API B. Both of them can be slow, but our application can run without the result from API B, so it is just a nice to have. Since we'd like a responsive application, we set a timeout of one second for the requests to complete. If the request to API B is still pending after that timeout, we cancel it and move on. Let's see what happens if we implement this without wrapping the requests in tasks.

Listing 4.16 cancelling a slow request

```
import asyncio
import aiohttp
from chapter_04 import fetch_status


async def main():
    async with aiohttp.ClientSession() as session:
        api_a = fetch_status(session, 'https://www.example.com')
        api_b = fetch_status(session, 'https://www.example.com', delay=2)

        done, pending = await asyncio.wait([api_a, api_b], timeout=1)

        for task in pending:
            if task is api_b:
                print('API B too slow, cancelling')
                task.cancel()

asyncio.run(main())
```

We'd expect for this code to print out `API B is too slow, cancelling`, but what happens is we don't see this message at all! This is because when we call `wait` with just coroutines they are wrapped in tasks for us automatically, and the done and pending sets returned are those tasks that wait created for us. This means that we can't do any comparisons to see which specific task is in the pending set such as `if task is api_b` since we'll be comparing a Task object we have no access to with a coroutine. However, if we wrap `fetch_status` in a task `wait` won't create any new objects for us and the comparison `if task is api_b` will work as we expect. In this case, we're correctly comparing two `Task` objects.

4.8 Summary

In this chapter we've introduced asynchronous context managers and introduced the aiohttp library for making web requests. We've also learned how to use some of the core asyncio apis for running multiple coroutines concurrently.

- We've learned how to use and create our own asynchronous context managers. These are special classes that allow us to asynchronously acquire resources and then release them, even if an exception occurred. These let us clean up any resources we may have acquired in a non-verbose manner and are useful when working with HTTP sessions as well as database connections. We can use them with the special `async with` syntax.
- We've learned how to use the aiohttp library to make asynchronous web requests. Aiohttp is a web client and server that uses non-blocking sockets. With the web client, we can execute multiple web requests concurrently in a way that does not block the event loop.
- We learned how to use the `asyncio.gather` function to run multiple coroutines concurrently and wait for them to complete. This function will return once all awaitables we pass into it have completed. If we want to keep track of any errors that happen, we can set `return_exeptions` to `True`. This will return the results of awaitables that completed successfully alongside any exceptions we received.
- We learned how to use the `as_completed` function to process results of a list of awaitables as soon as they complete. This will give us an iterator of futures that we can loop over. As soon as a coroutine or task has finished, we'll be able to access the result and process it.
- If we want to run multiple tasks concurrently but want to be able to understand which tasks are done and which are still running we can use `wait`. This function also allows us finer grained control on when it returns results. When it returns, we get a set of tasks that have finished and set of tasks that are still running. We can then cancel any tasks we wish or do any other awaiting we need.

5 Non-blocking database drivers

This chapter covers

- **Running asyncio friendly database queries with asyncpg**
- **Running multiple SQL queries concurrently**
- **Creating database connection pools**
- **Managing asynchronous database transactions**
- **Using asynchronous generators to stream query results**

Last chapter we learned how to make non-blocking web requests with the aiohttp library. We also learned several different asyncio API methods for running these requests concurrently. With the combination of the asyncio APIs and the aiohttp library we are able to run multiple long-running web requests concurrently leading to an improvement in our application's run time. The concepts we learned in the last chapter do not just apply to web requests, they also apply to running SQL queries as well and can improve the performance of database-intensive applications.

Much like web requests, we'll need to use an asyncio friendly library since our typical SQL libraries block the main thread, and therefore the event loop, until a result comes back. In this chapter we'll learn more about an asynchronous database access with the asyncpg library. We'll first create a simple schema to keep track of products for an e-commerce storefront that we'll then use to run queries against asynchronously. We'll then take a look at how to manage transactions and rollbacks within our database as well as set up connection pooling.

5.1 Introducing asyncpg

As we've mentioned earlier in the book, our existing blocking libraries won't work out of the box with coroutines. To run queries concurrently against a database, we'll need to use an asyncio friendly library that uses non-blocking sockets. To do this, we'll use a library called asyncpg, which will let us asynchronously connect to Postgres databases and run queries against them.

In this chapter we'll focus on Postgres databases, but what we learn here will also be applicable to MySQL and other databases as well. The creators of aiohttp have also created the aiomysql library, which can connect and run queries against a MySQL database. While there are some differences, the APIs are similar, and the knowledge is transferrable. It is worth noting that the asyncpg library did not implement the Python database API specification defined in PEP-249 (available at https://www.python.org/dev/peps/pep-0249). This was a conscious choice on the part of the library implementors as a concurrent implementation is inherently different from a synchronous one. The creators of aiomysql however went a different route and do implement PEP-249, so this library's API will feel familiar to others who have used synchronous database drivers in Python.

The current documentation for asynpg is available at https://magicstack.github.io/asyncpg/current/. Now that we've learned a little about the driver we'll be using, let's connect to our first database.

5.2 Connecting to a Postgres Database

To get started with asyncpg, we'll use a real-world scenario of creating a product database for an ecommerce storefront. We'll use this database throughout the chapter to demonstrate database problems in this domain that we might need to solve.

The first thing we'll need to do to get started creating our product database and running queries is establish a connection to our database. For this section and the rest of the chapter, we'll assume that you have a Postgres database running on your local machine on the default port of 5432 and we'll assume the default user `postgres` has a password of '`password`'. We'll be hardcoding the password in these code examples for simplicity but note you should *never* hardcode a password in your code as this violates security principles. Store your passwords in environment variables or some other configuration mechanism. You can download and install a copy of Postgres from https://www.postgresql.org/download/, just choose the appropriate operating system you're working on. You may also consider using the Docker Postgres image, which you can read more about at https://hub.docker.com/_/postgres/.

Once we have our database setup, we'll need to install the asyncpg library. We'll use `pip3` to do this and we'll install the latest version at the time of writing, `0.0.21`.

```
pip3 install -Iv asyncpg==0.21.0
```

Once this is installed, we can now import the library and establish a connection to our database. Asyncpg provides this with the `asyncpg.connect` function. Let's use this to connect and print out the database version number.

Listing 5.1 connecting to a Postgres database as the default user

```
import asyncpg
```

```
import asyncio

async def main():
    connection = await asyncpg.connect(host='0.0.0.0',
                                       port=5432,
                                       user='postgres',
                                       database='postgres',
                                       password='password')
    version = connection.get_server_version()
    print(f'Connected! Postgres version is {version}')
    await connection.close()

asyncio.run(main())
```

In listing 5.1 we create a connection to our Postres instance as the default `postgres` user and the default `postgres` database. Assuming our Postgres instance is up and running, we should see something like `Connected! Postgres version is ServerVersion(major=12, minor=0, micro=3, releaselevel='final', serial=0)` printed out to our console, indicating we've successfully connected to our database. Finally, we close the connection to the database with `await connection.close()`.

Now we've connected, but there is nothing in our database currently. The next step is to create a product schema that we can interact with. In creating this schema, we'll learn how to execute basic queries with asyncpg.

5.3 Defining a database schema

To get started running queries against our database, we'll first need to create a database schema. We're going to pick a simple schema that we'll call *`products`*, modeling real world products that an online storefront might have in stock. Let's define a few different entities that we can then turn into tables in our database.

Brand

A brand is a manufacturer of many distinct products. For instance, Ford is a brand that produces many different model cars (e.g. Ford F150, Ford Fiesta, etc.).

Product

A product is associated with one brand and there is a one-to-many relationship between brands and products. For simplicity, in our product database a product will just have a product name. In the Ford example, a product is a compact car called the Fiesta, the brand is Ford. In addition, each product in our database will come in multiple sizes and colors. We'll define the available sizes and colors as SKUs.

SKU

A SKU is shorthand for a "stock keeping unit". A SKU represents a distinct item that a storefront has for sale. For instance, "Jeans" may be a product for sale and a SKU might be "Jeans, size: medium color: blue" or "Jeans, size: small color: black". There is a one-to-many relationship between a product and a SKU.

Product Size

A product can come in multiple sizes. For this example, we'll only consider that there are three sizes available, small, medium and large. Each SKU has one product size associated with it, so there is a one-to-many relationship between product sizes and SKUs.

Product Color

A product can come in multiple colors. For this example, we'll say our inventory only consists of two colors, black and blue. There is a one-to-many relationship between product color and SKUs.

Putting this all together, we'll be modeling a database schema as follows:

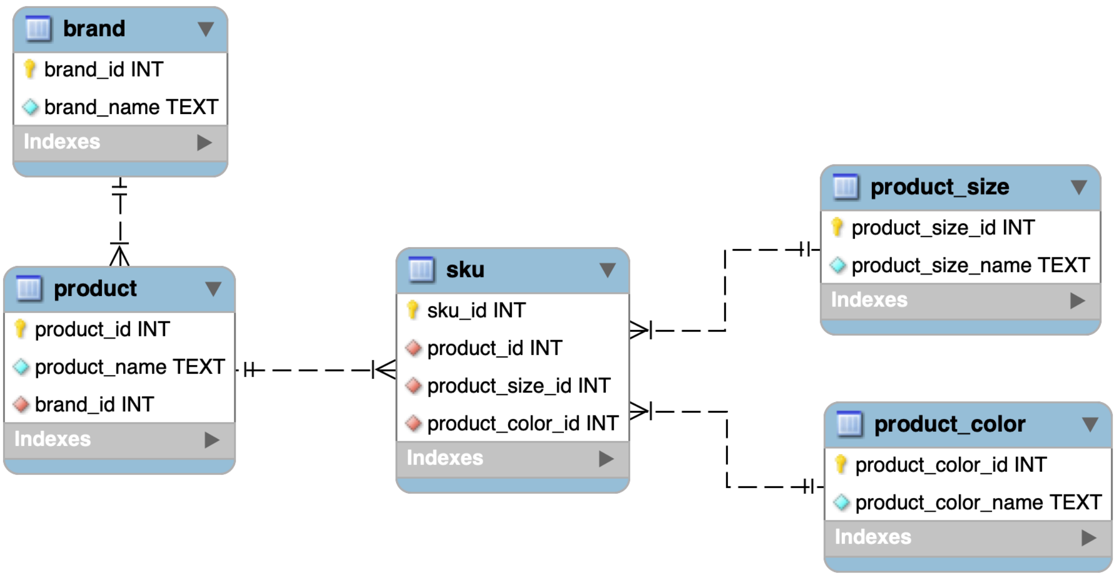


Figure 5.1 The entity diagram for the products database

Now let's define some variables with the SQL we'll need to create this schema. Using asyncpg, we'll execute these statements to create our product database. Since our sizes and colors are known ahead of time, we'll also insert a few records into the `product_size` and `product_color` tables. We'll reference these variables in the upcoming code listings so we don't need to repeat lengthy SQL create statements.

Listing 5.2 Product schema table create statements

```
CREATE_BRAND_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS brand(
        brand_id SERIAL PRIMARY KEY,
        brand_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product(
        product_id SERIAL PRIMARY KEY,
        product_name TEXT NOT NULL,
        brand_id INT NOT NULL,
        FOREIGN KEY (brand_id) REFERENCES brand(brand_id)
    );"""

CREATE_PRODUCT_COLOR_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_color(
        product_color_id SERIAL PRIMARY KEY,
        product_color_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_SIZE_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_size(
        product_size_id SERIAL PRIMARY KEY,
        product_size_name TEXT NOT NULL
    );"""

CREATE_SKU_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS sku(
       sku_id SERIAL PRIMARY KEY,
       product_id INT NOT NULL,
       product_size_id INT NOT NULL,
       product_color_id INT NOT NULL,
       FOREIGN KEY (product_id)
          REFERENCES product(product_id),
       FOREIGN KEY (product_size_id)
       REFERENCES product_size(product_size_id),
       FOREIGN KEY (product_color_id)
       REFERENCES product_color(product_color_id)
    );"""

COLOR_INSERT = \
    """
    INSERT INTO product_color VALUES(1, 'Blue');
    INSERT INTO product_color VALUES(2, 'Black');
    """

SIZE_INSERT = \
    """
    INSERT INTO product_size VALUES(1, 'Small');
    INSERT INTO product_size VALUES(2, 'Medium');
    INSERT INTO product_size VALUES(3, 'Large');
    """
```

Now that we have the statements to create our tables and insert our sizes and colors, we need a way to run them.

5.4 Executing queries with asyncpg

To run queries against our database, we'll first need to create our product database. To do this we'll need to connect to our Postgres instance and create the database directly outside of Python. We can create the database by executing the following statement once connected to the database as the default postgres user:

```
CREATE DATABASE products;
```

You can execute this via the command line by running sudo `-u postgres psql -c "CREATE TABLE products;"`

In the next examples, we'll assume you have executed this statement as we'll connect to the products database directly.

Now that we've created our products database, we'll connect to it and execute our create statements. The connection class has a coroutine called `execute` that we can use to run our create statements one by one. This coroutine returns a string representing the status of the query that Postgres returned. Let's take the statements we created in the last section and execute them.

Listing 5.3 Using a execute to run create statements

```
import asyncpg
import asyncio


async def main():
    connection = await asyncpg.connect(host='0.0.0.0',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    statements = [CREATE_BRAND_TABLE,
                  CREATE_PRODUCT_TABLE,
                  CREATE_PRODUCT_COLOR_TABLE,
                  CREATE_PRODUCT_SIZE_TABLE,
                  CREATE_SKU_TABLE,
                  SIZE_INSERT,
                  COLOR_INSERT]

    print('Creating the product database...')
    for statement in statements:
        status = await connection.execute(statement)
        print(status)
    print('Finished creating the product database!')
    await connection.close()
```

```
asyncio.run(main())
```

We first create a connection to our products database similarly to what we did in our first example, the difference being we connect to the products database. Once we have this connection, we then start to execute our create table statements one by one with `connection.execute()`. Note that `execute()` is a coroutine, so in order to run our SQL we need to `await` the call. Assuming everything worked properly, the status of each execute statement should be `CREATE TABLE` and each insert statement should be `INSERT 0 1`. Finally, we close the connection to the product database. Note that in this example we await each SQL statement in a for loop, this ensures that we run the insert statements synchronously. Since some tables depend on others, we can't run them concurrently.

These statements we ran don't have any results associated with them, so let's insert a few pieces of data and run some simple select queries. We'll first insert a few brands and then query them to ensure we've inserted them properly. We can insert data with the `execute` coroutine as before and we can run a query with the `fetch` coroutine.

Listing 5.4 Inserting and selecting brands

```
import asyncpg
import asyncio
from asyncpg import Record
from typing import List


async def main():
    connection = await asyncpg.connect(host='0.0.0.0',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'Levis')")
    await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'Seven')")

    brand_query = 'SELECT brand_id, brand_name FROM brand'
    results: List[Record] = await connection.fetch(brand_query)

    for brand in results:
        print(f'id: {brand["brand_id"]}, name: {brand["brand_name"]}')

    await connection.close()


asyncio.run(main())
```

We first insert two brands into the brand table. Once we've done this, we use `connection.fetch` to get all brands from our brand table. Once this query has finished, we will have all results in memory in the `results` variable. Each result will be an `asyncpg Record` object. These objects act similarly to dictionaries and allow us to access data by passing in a column name with subscript syntax. Executing this will give us the following output:

```
id: 1, name: Levis
id: 2, name: Seven
```

In this example, we fetch all data for our query into a list. If we wanted to fetch a single result we could call `connection.fetchrow()` which will return a single record from the query. The default asyncpg connection will pull all results from our query into memory, so for the time being there is no performance difference between `fetchrow` and `fetch`. Later in this chapter, we'll see how to use streaming result sets with cursors. These will only pull a few results into memory at a time, which is a useful technique for when queries may return large amounts of data.

The examples we've seen so far run queries one after another, we could have had similar performance by using a non asyncio database driver. However, since we're now returning coroutines we can use the asyncio API methods we learned in the last chapter to execute queries concurrently.

5.5 Executing queries concurrently with connection pools

The true benefit of asyncio for I/O bound operations is the ability to run multiple tasks concurrently. Queries independent from one another that we need to make repeatedly are good examples of where we can apply concurrency to make our application perform better. To demonstrate this, we're going to pretend we're a successful ecommerce storefront. Our company carries one hundred thousand SKUs for one thousand distinct brands.

We'll also pretend we sell our items through partners. These partners make requests for thousands of products at a given time through a batch process we have built. Running all these queries sequentially could be slow, so we'd like to create an application that executes these queries concurrently to ensure a speedy experience.

Since this is an example and we don't actually have one hundred thousand SKUs on hand, we'll start by creating a fake product and SKU records in our database. We'll randomly generate one hundred thousand SKUs for random brands and products, and we'll use this data set as a basis for running our queries.

5.5.1 Inserting random SKUs into the product database

Since we don't want to list brands, products and SKUs ourselves, so we'll randomly generate them. We'll pick random names from the list of the thousand most frequently occurring English words. For the sake of this example, we'll assume we have a text file that contains these words called common_words.txt. You can download a copy of this file from the book's github data repository at https://github.com/concurrency-in-python-with-asyncio/data.

The first thing we'll want to do is insert our brands since our product table depends on `brand_id` as a foreign key. We'll use the `connection.executemany` coroutine to write parameterized SQL to insert these brands. This will allow us to write one SQL query and pass in a list of parameters we want to insert, instead of having to create one `INSERT` statement for each brand.

The `executemany` coroutine takes in one single SQL statement and a list of tuples with values we'd like to insert. We can parameterize the SQL statement by using `$1, $2 … $N` syntax. Each number after the dollar sign represents the index of the tuple we'd like to use in the SQL statement. For instance, if we have a query we write as `'INSERT INTO table VALUES($1, $2)'` and a list of tuples `[('a', 'b'), ('c', 'd')]` this would execute two inserts for us:

```
INSERT INTO table ('a', 'b')
INSERT INTO table ('c', 'd')
```

We'll first generate a list of one hundred random brand names from our list of common words. We'll return this as a list of tuples each with one value inside of it so that we can use this in the `executemany` coroutine. Once we've created this list, it's a simple matter of passing a parameterized `INSERT` statement alongside this list of tuples.

Listing 5.5 inserting random brands

```
import asyncpg
import asyncio
from typing import List, Tuple, Union
from random import sample


def load_common_words() -> List[str]:
    with open('common_words.txt') as common_words:
        return common_words.readlines()


def generate_brand_names(words: List[str]) -> List[Tuple[Union[str, ]]]:
    return [(words[index],) for index in sample(range(100), 100)]


async def insert_brands(common_words, connection) -> int:
    brands = generate_brand_names(common_words)
    insert_brands = "INSERT INTO brand VALUES(DEFAULT, $1)"
    return await connection.executemany(insert_brands, brands)


async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    await insert_brands(common_words, connection)


asyncio.run(main())
```

Internally, `executemany` will loop through our `brands` list and generate one `INSERT` statement per each brand. Then it will execute all those insert statements at once. This method of parameterization will also prevent us from SQL injection attacks as the input data is sanitized. Once we run this, we should have 100 brands in our system with random names.

Now that we've seen how to insert random brands, let's use the same technique to insert products and skus. For products, we'll create a description of ten random words and a random brand id. For skus, we'll pick a random size, color and product. We'll assume that our brand id starts at one and ends at 100 for simplicity.

Listing 5.6 inserting random products and skus

```
def gen_products(common_words: List[str],
                 brand_id_start: int,
                 brand_id_end: int,
                 products_to_create: int) -> List[Tuple[str, int]]:
    products = []
    for _ in range(products_to_create):
        description = [common_words[index] for index in sample(range(1000), 10)]
        brand_id = randint(brand_id_start, brand_id_end)
        products.append((" ".join(description), brand_id))
    return products


def gen_skus(product_id_start: int,
             product_id_end: int,
             skus_to_create: int) -> List[Tuple[int, int, int]]:
    skus = []
    for _ in range(skus_to_create):
        product_id = randint(product_id_start, product_id_end)
        size_id = randint(1, 3)
        color_id = randint(1, 2)
        skus.append((product_id, size_id, color_id))
    return skus


async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    product_tuples = gen_products(common_words,
                                  brand_id_start=1,
                                  brand_id_end=100,
                                  products_to_create=1000)
    await connection.executemany("INSERT INTO product VALUES(DEFAULT, $1, $2)",
                                 product_tuples)

    sku_tuples = gen_skus(product_id_start=1,
                          product_id_end=1000,
                          skus_to_create=100000)
    await connection.executemany("INSERT INTO sku VALUES(DEFAULT, $1, $2, $3)",
                                 sku_tuples)

    await connection.close()
```

```
asyncio.run(main())
```

When we run this listing, we should have a completely filled database with one thousand products and 100,000 SKUs. Depending on your machine, this may take several seconds to run. With a few joins, we can now query all available SKUs for a particular product. Let's see what this query would look like for product id 100:

```
product_query = \
"""
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id
JOIN product_size as ps on ps.product_size_id = s.product_size_id
WHERE p.product_id = 100"""
```

When we execute this query, we'll get one row for each SKU for a product, we'll also get the proper English name for size and color instead of an id. Assuming we have a lot of product ids we'd like to query at a given time this provides us for a good opportunity to apply concurrency. We may naively try to apply `asyncio.gather` with our existing connection like so:

```
async def main():
    connection = await asyncpg.connect(host='0.0.0.0',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    print('Creating the product database...')
    queries = [connection.execute(product_query),
               connection.execute(product_query)]
    results = await asyncio.gather(*queries)
```

However, if we run this we'll be greeted with an error:

```
RuntimeError: readexactly() called while another coroutine is already waiting for incoming
       data
```

Why is this? In the SQL world one connection means one socket connection to our database. Since we only have one connection and we're trying to read the results of multiple queries concurrently we wind up with an error. We can resolve this by creating multiple connections to our database and executing one query per connection. Since creating connections are expensive, caching them so we can access them when needed makes sense. This is commonly known as a connection pool.

5.5.2 Creating a connection pool to run queries concurrently

Since we can only run one query per connection at a time, we need a mechanism for creating and managing multiple connections. A connection pool does just that. You can think of a connection pool as a cache of existing connections to a database instance. They contain a finite amount of connections which we can access when we need to run a query.

Using connection pools, we *acquire* connections when we need to run a query. Acquiring a connection means we ask the pool "do you currently have any connections available in the pool? If so, give me one so I can run my queries." Connection pools facilitate the reuse of these connections to execute queries. This means that once a connection is acquired from the pool to run a query and that query finishes, we return or 'release' it to the pool for others to reuse. This is important because establishing a connection with a database is expensive in terms of time. If we had to create a new connection for every query we wanted to run, our application's performance would quickly degrade.

Since the connection pool has a finite amount of connections, we could be waiting for quite some time for one to become available as other connections may be in use. This means connection acquisition is an operation that may take some time to complete. If we only have 10 connections in the pool, each of which is in use, and we ask for another one, we'll have to wait until one of the other connections becomes available for our query to execute.

To illustrate how this works in terms of asyncio, let's imagine we have a connection pool with two connections. Let's also imagine we have three coroutines that each run a query. We'll run these three coroutines concurrently as tasks. With a connection pool set up this way, the first two coroutines that attempt to run their queries will acquire the two available connections and start running their queries. While this is happening, the third coroutine will be waiting for a connection to become available. When either one of the first two coroutines finishes running its query, it will release its connection and return it to the pool. This lets the third coroutine acquire it and start using it to run its query.

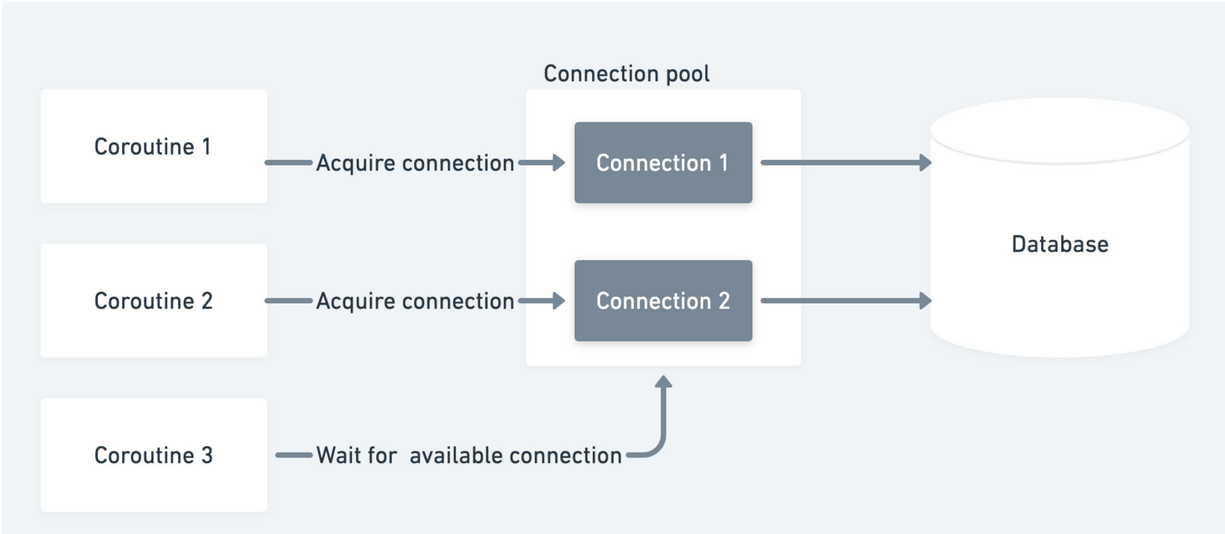


Figure 5.2 Coroutines one and two acquire connections to run their queries while coroutine three waits for a connection. Once either coroutine one or two finishes, coroutine three will be able to use the newly released connection and will be able to execute its query.

In this model, we can have at most two queries running concurrently. Normally, your connection pool will be a bit bigger to enable more concurrency. For the examples we'll use a connection pool of six, but the actual number you want to use is dependent on the hardware your database runs on as well as the hardware your application runs on. In this case, you'll need to benchmark which connection pool size works best. Keep in mind, bigger is not always better, but that's a much larger topic.

Now that we understand how connection pools work, how do we create one with asyncpg? Asyncpg exposes a coroutine named `create_pool` to accomplish this. We use this instead of the `connect` function we used earlier to establish a connection to our database. When we call this, we'll specify the number of connections we wish to create in our pool. We'll do this with the `min_size` and `max_size` parameters. The `min_size` parameter specifies the minimum number of connections in our connection pool. This means that once we set up our pool, we are guaranteed to have these many connections inside of it already established. The `max_size` parameter specifies the maximum number of connections we want in our pool, this is the maximum amount of connections we can have. If we don't have enough connections available, the pool will create a new one for us as long as the new connection won't cause the pool size to be above the value set in `max_size`. For our first example, we'll set both these values to six. This guarantees we always have six connections available.

Asyncpg pools are asynchronous context managers, this means that we need to use `async with` syntax to create a pool. Once we've established a pool, we can acquire connections using the `acquire` coroutine. This coroutine will suspend execution until we have a connection available. Once we do, we can then use that connection to execute whatever

SQL we'd like. Acquiring a connection is also an async context manager that returns the connection to the pool when we are done with it, so we'll need to use `async with` syntax just like we did when we created the pool. Using this, we can rewrite our code to run several queries concurrently.

Listing 5.7 establishing a connection pool and running queries concurrently

```
import asyncio
import asyncpg

async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)


async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool: #A

        await asyncio.gather(query_product(pool),
                             query_product(pool)) #B


asyncio.run(main())
```

#A Create a connection pool with six connections
#B Execute two product queries concurrently

In listing 5.7 we first create a connection pool with six connections. We then create two query coroutine objects and schedule them to run concurrently with `asyncio.gather`. In our `query_product` coroutine we first acquire a connection from the pool with `pool.acquire()`. This coroutine will then suspend until a connection is available from the connection pool. We do this in an `async with` block, this will ensure that once we leave the block the connection will be returned to the pool. This is important, if we don't do this we can run out of connections and we'll wind up with an application that hangs forever, waiting for a connection that will never be available. Once we've acquired a connection, we can then run our query as we did in previous examples.

We can expand this example to run ten thousand queries by creating ten thousand different query coroutine objects. To make this interesting, we'll write a version that runs the queries synchronously and compare how long things take.

Listing 5.8 Synchronous queries versus concurrent

```
import asyncio
import asyncpg
from util import async_timed
```

```
async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

@async_timed()
async def query_products_synchronously(pool, queries):
    return [await query_product(pool) for _ in range(queries)]


@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)


async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:
        await query_products_synchronously(pool, 10000)
        await query_products_concurrently(pool, 10000)

asyncio.run(main())
```

In `query_products_synchronously` we put an `await` in a list comprehension, which will force each call to `query_product` to run sequentially. Then, in `query_products_concurrently` we create a list of coroutines we want to run and then kick them off concurrently with `gather`. In our main coroutine we then run our synchronous and concurrent version with ten thousand queries each. While the exact results can vary substantially based on your hardware, the concurrent version is nearly five times as fast as the serial version:

```
starting <function query_products_synchronously at 0x1219ea1f0> with args
       (<asyncpg.pool.Pool object at 0x12164a400>, 10000) {}
finished <function query_products_synchronously at 0x1219ea1f0> in 21.8274 second(s)
starting <function query_products_concurrently at 0x1219ea310> with args
       (<asyncpg.pool.Pool object at 0x12164a400>, 10000) {}
finished <function query_products_concurrently at 0x1219ea310> in 4.8464 second(s)
```

An improvement like this is great, but there are still more improvements we can make if we need more throughput. Since our query is relatively fast, this code is a mixture of CPU bound in addition to I/O bound. In the next chapter, we'll see how to squeeze even more performance out of this setup.

So far, we've seen how to insert data into our database assuming we don't have any failures. But what happens if we are in the middle of inserting products and we get a failure? We don't want an inconsistent state in our database, so this is where database transactions come into play. Next, we'll see how to use asynchronous context managers to acquire and manage transactions.

5.6 Managing transactions with asyncpg

Transactions are a core concept in many databases that satisfy the ACID (atomic, consistent, isolated, durable) properties. A transaction consists of one or more SQL statements that are executed as one atomic unit. If no errors occur when we execute the statements within a transaction, we *commit* the statements to the database, making any changes a permanent part of the database. If there are any errors, we *roll back* the statements, and it is like none of them ever happened. In the context of our product database, we may need to roll back a set of updates if we attempt to insert a duplicate brand, or if we have violated a database constraint we've set.

In asyncpg the easiest way to deal with transactions is to use the `connection.transaction` asynchronous context manager to start them. Then, if there is an exception in the `async with` block, the transaction will automatically be rolled back for us. If everything executes successfully, it will be automatically committed. Let's take a look at how to create a transaction and execute two simple insert statements to add a couple of brands.

Listing 5.9 creating a transaction

```
import asyncio
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction(): #A
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_1')")
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_2')")

    query = """SELECT brand_name FROM brand
                WHERE brand_name LIKE 'brand%'"""
    brands = await connection.fetch(query) #B
    print(brands)

    await connection.close()

asyncio.run(main())
```

#A Start a database transaction
#B Select brands to ensure that our transaction was committed

Assuming our transaction committed successfully, we should see `[<Record brand_name='brand_1'>, <Record brand_name='brand_2'>]` printed out to the console.

The above example assumes we didn't get any errors running the two insert statements and everything was committed successfully. To demonstrate what happens when a rollback occurs, let's force a SQL error. To test this out, we'll try and insert two brands with the same primary key id. Our first insert will work successfully, but our second insert will raise a duplicate key error.

Listing 5.10 handling an error in a transaction

```
import asyncio
import logging
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    try:
        async with connection.transaction():
            insert_brand = "INSERT INTO brand VALUES(9999, 'big_brand')"
            await connection.execute(insert_brand)
            await connection.execute(insert_brand) #A
    except Exception:
        logging.exception('Error while running transaction') #B
    finally:
        query = """SELECT brand_name FROM brand
                    WHERE brand_name LIKE 'big_%'"""
        brands = await connection.fetch(query) #C
        print(f'Query result was: {brands}')

        await connection.close()

asyncio.run(main())
```

#A This insert statement will error because of a duplicate primary key
#B If we had an exception, log the error
#C Select the brands to ensure we didn't insert anything

In listing 5.10 our second insert statement throws an error. This leads to the following output:

```
ERROR:root:Error while running transaction
Traceback (most recent call last):
  File "listing_5_10.py", line 16, in main
    await connection.execute("INSERT INTO brand "
  File "asyncpg/connection.py", line 272, in execute
    return await self._protocol.query(query, timeout)
  File "asyncpg/protocol/protocol.pyx", line 316, in query
asyncpg.exceptions.UniqueViolationError: duplicate key value violates unique constraint
       "brand_pkey"
DETAIL:  Key (brand_id)=(9999) already exists.
Query result was: []
```

We first get an exception because we attempted to insert a duplicate key, we then see that the result of our select statement was empty, indicating we successfully rolled back the transaction.

5.6.1 Nested transactions

Asyncpg also supports the concept of a nested transaction through a Postgres feature called *savepoints*. Savepoints are defined in Postgres with the `SAVEPOINT` command. When we define a savepoint, we can roll back to that savepoint and any queries executed after the savepoint will roll back, but any queries successfully executed before it will not roll back.

In asyncpg, we can create a savepoint by calling the `connection.transaction` context manager within an existing transaction. Then, if there is any error within this inner transaction it is rolled back, but the outer transaction is not affected. Let's try this out by inserting a brand in a transaction and then within a nested transaction attempting to insert a color that already exists in our database.

Listing 5.11

```
import asyncio
import asyncpg
import logging


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'my_new_brand')")

        try:
            async with connection.transaction():
                await connection.execute("INSERT INTO product_color VALUES(1, 'black')")
        except Exception as ex:
            logging.warning('Ignoring error inserting product color', exc_info=ex)

    await connection.close()


asyncio.run(main())
```

When we run this code, our first `INSERT` statement runs successfully as we don't have this brand in our database yet. Our second insert statement fails with a duplicate key error. Since this second insert statement is within a transaction and we catch and log the exception, our outer transaction is not rolled back and the brand is properly inserted, despite the error. If we did not have the nested transaction, or second insert statement would have also rolled back our brand insert.

5.6.2 Manually managing transactions

So far, we have used asynchronous context managers to handle committing and rolling back our transactions. Since this is less verbose than managing things ourselves, it is usually the best approach. That said we may wind up in situations where we need to manually manage a transaction. For example, we may want to have custom code execute on rollback, or we may want to roll back on a condition other than an exception.

To manually manage a transaction, we can use the transaction manager returned by `connection.transaction` outside of a context manager. When we do this, we'll manually need to call its `start` method to start a transaction and then `commit` on success and `rollback` on failure. Let's take a look at how to do this by rewriting our first example.

Listing 5.12 manually managing a transaction

```
import asyncio
import asyncpg
from asyncpg.transaction import Transaction


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    transaction: Transaction = connection.transaction() #A
    await transaction.start() #B
    try:
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_1')")
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_2')")
    except asyncpg.PostgresError:
        print('Errors, rolling back transaction!')
        await transaction.rollback() #C
    else:
        print('No errors, committing transaction!')
        await transaction.commit() #D

    query = """SELECT brand_name FROM brand
                WHERE brand_name LIKE 'brand%'"""
    brands = await connection.fetch(query)
    print(brands)

    await connection.close()

asyncio.run(main())
```

#A Create a transaction instance
#B Start the transaction
#C If there was an exception, roll back
#D If there was no exception, commit

We first start by creating a transaction with the same method call we used with async context manager syntax, but instead we store the `Transaction` instance that this call returns. You can think of this class as a manager for our transaction, with this we'll be able to perform any commits and rollbacks we need. Once we have a transaction instance, we can then call the start coroutine. This will execute a query to start the transaction in Postgres. Then, within a try block we can execute any queries we'd like. In this case we insert two brands. If there were errors with any of those INSERT statements, we'll hit the except block and roll back the transaction by calling the `rollback` coroutine. If there were no errors, we call the commit coroutine which will end the transaction and make any changes in our transaction permanent in the database.

Up until now we have been running our queries in a way that pulls all query results into memory at once. This makes sense for many applications since many queries will return small result sets. However, we may have a situation where we are dealing with a large result set that may not fit in memory all at once. In these cases, we may want to stream results to avoid taxing our system's RAM. Next, we'll take a look at how to do this with asyncpg and along the way introduce asynchronous generators.

5.7 Asynchronous generators and streaming result sets

One drawback of the default `fetch` implementation asynpg provides is that it pulls all data from any query we execute into memory. This means that if we have a query that returns one million rows, we'd attempt to transfer that entire set from the database to the requesting machine. Going back to our product database example, imagine we're even more successful and have billions of products available. It is highly likely that we'll have some queries that will return very large result sets, potentially hurting performance.

Of course, we could apply `LIMIT` statements to our query and paginate things, and this makes sense for many, if not most applications. That said, there is overhead with this approach in that we are sending the same query multiple times, potentially creating extra stress on the database. If we find ourselves hampered by these issues, it can make sense to stream results for a particular query only as we need them. This will save on memory consumption at our application layer as well as save load on the database. However, it does come at the expense of making more round trips over the network to the database.

Postgres supports streaming query results through the concept of cursors. You can think of a cursor as a pointer to where we currently are in iterating through a result set. When we get a single result from a streamed query, we advance the cursor to the next element and so on until we have no more results.

Using asyncpg, we can get a cursor directly from a connection which we can then use to execute a streaming query. Cursors in asyncpg use an asyncio feature we have not used yet called asynchronous generators. Asynchronous generators generate results asynchronously one by one similarly to regular Python generators. They also allow us to use a special for loop style syntax to iterate over any results we get. To fully understand how this works, we'll first introduce asynchronous generators as well as `async for` syntax to loop these generators.

5.7.1 Introducing asynchronous generators

Many developers will be familiar with generators from the synchronous Python world. Generators are an implementation of the iterator design pattern made famous in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by the "gang of four". This pattern allows us to define sequences of data lazily and iterate through them one element at a time. This is useful for potentially large sequences of data where we don't need to store everything in memory all at once.

A simple synchronous generator is a normal Python method which contains a `yield` statement instead of a `return` statement. For example, let's take a look at how to create and use a generator which gives us positive integers starting from zero until a specified end.

Listing 5.13 a synchronous generator

```
def positive_integers(until: int):
    for integer in range(until):
        yield integer


positive_iterator = positive_integers(2)

print(next(positive_iterator))
print(next(positive_iterator))
```

In listing 5.13 we create a function which takes an integer that we want to count up to. We then start a loop until our specified end integer. Then at each iteration of the loop we 'yield' the next integer in the sequence. When we call `positive_integers(2)` we don't return an entire list or even run the loop in our method. In fact, if we check the type of `positive_iterator`, we'll get `<class 'generator'>`.

We then use the `next` utility function to iterate over our generator. Each time we call next this will trigger one iteration of the for loop in `positive_integers`, giving us the result of the `yield` statement per each iteration. Thus, the code in 5.13 will print `1` and `2` to the console. We could have also used a for loop with our generator to loop through all values in our generator instead of using `next`.

This works for synchronous methods, but what if we wanted to use coroutines to generate a sequence of values asynchronously? Going back to our database example, what if we wanted to generate a sequence of rows that we lazily get from our database? We can do this with Python's asynchronous generators and special `async for` syntax. To demonstrate a simple asynchronous generator, let's start with our positive integer example but introduce a call to a coroutine that takes a few seconds to complete. We'll use our delay function from chapter two for this.

Listing 5.14 a simple asynchronous generator

```
import asyncio
from util import delay, async_timed

async def positive_integers_async(until: int):
    for integer in range(1, until):
```

```
        await delay integer
        yield integer

@async_timed()
async def main():
    async_generator = positive_integers_async(3)
    print(type(async_generator))
    async for number in async_generator:
        print(f'Got number {number}')

asyncio.run(main())
```

Running listing 5.14 we'll see the type is no longer a plain generator but `<class 'async_generator'>`, an asynchronous generator. An asynchronous generator differs from a regular generator that instead of generating plain Python objects as elements it generates coroutines which we can then await until we get a result. Because of this, our normal for loops and `next` functions won't work with these types of generators. Instead, we have a special syntax, `async for`, to deal with these types of generators. In this example we use this syntax to iterate over `positive_integers_async`.

This code will print the numbers one to three, waiting one second before returning the first number and two seconds before returning the second. Note that this is not running the coroutines generated concurrently but is instead generating and awaiting them one at a time in serial.

5.7.2 Using asynchronous generators with a streaming cursor

The concept of asynchronous generators pairs nicely with the concept a streaming database cursor. Using these generators, we'll be able to fetch one row at a time with a simple for loop like syntax. To perform streaming with asyncpg, we'll first need to start a transaction as Postgres requires this to use cursors. Once we've started a transaction, we can then call the `cursor` method on the `Connection` class to obtain a cursor. When we call the cursor method, we'll pass in the query we'd like to stream. This method will return an asynchronous generator which we can use to stream results one at a time.

To get familiar with how to do this, let's run a query to get all products from our database with a cursor. We'll then use `async for` syntax to fetch elements one at a time from our result set.

Listing 5.15 streaming results one by one

```
import asyncpg
import asyncio
import asyncpg


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
```

```
    query = 'SELECT product_id, product_name FROM product'
    async with connection.transaction():
        async for product in connection.cursor(query):
            print(product)

    await connection.close()


asyncio.run(main())
```

Listing 5.15 will print all of our products out one by one. Despite us having put a thousand products in this table, we'll only pull a few into memory at a time. Currently at the time of writing, the cursor defaults to prefetching 50 records at a time for us to cut down on network traffic. We can change this behavior by setting the `prefetch` parameter with however many elements we'd like to prefetch.

We can also use these cursors to skip around our result set and fetch an arbitrary number of rows at a time. Let's see how to do this by getting a few records from the middle of the query we just used.

Listing 5.16 moving the cursor and fetching records

```
import asyncpg
import asyncio


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        query = 'SELECT product_id, product_name from product'
        cursor = await connection.cursor(query) #A
        await cursor.forward(500) #B
        products = await cursor.fetch(100) #C
        for product in products:
            print(product)

    await connection.close()


asyncio.run(main())
```

#A Create a cursor for the query
#B Move the cursor forward five hundred records
#C Get the next one hundred records

The code in listing 5.16 will first create a cursor for our query. Note that we use this in an await statement like a coroutine instead of an asynchronous generator – this is because in asyncpg a cursor is both an asynchronous generator *and* an awaitable. For the most part this is similar to using an async generator, but there is a difference in prefetch behavior when creating a cursor this way. Using this method, we cannot set a prefetch value. Doing so will raise an `InterfaceError`.

Once we have the cursor, we use its `forward` coroutine method to move forward in the result set. This will effectively skip the first 500 records in our product table. Once we've moved our cursor forward, we then fetch the next one hundred products and print them each out to the console.

These types of cursors are non-scrollable by default, meaning we can only advance forward in the result set. If you want to use scrollable cursors that can move both forwards and backwards, you'll need to execute the SQL to do so manually using `DECLARE … SCROLL CURSOR`, you can read more on how to do this in the Postgres documentation at https://www.postgresql.org/docs/9.2/plpgsql-cursors.html.

Both of these techniques are useful if we have a really large result set and don't want to have the entire set in memory. The `async for` loops we saw in listing 5.16 are useful for looping over the entire set, while creating a cursor and using the `fetch` coroutine method is useful for fetching a chunk of records or skipping a set of records.

However, what if we only want to retrieve a fixed set of elements at a time with prefetching and still use an `async for` loop? We could add a counter in our `async for` loop and break out after we've seen a certain number of elements, but that isn't particularly reusable if we need to do this often in our code. What we can do to make this easier is build our own async generator. We'll call this generator `take`. This generator will take an async generator and the number of elements we wish to extract. Let's take a look at how to create this and grab the first five elements from a result set.

Listing 5.17 getting a specific number of elements with an asynchronous generator

```
import asyncpg
import asyncio


async def take(generator, to_take: int):
    item_count = 0
    async for item in generator:
        yield item
        item_count = item_count + 1
        if item_count > to_take - 1:
            return


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        query = 'SELECT product_id, product_name from product'
        product_generator = connection.cursor(query)

        async for product in take(product_generator, 5):
            print(product)
```

```
        print('Got the first five products!')

    await connection.close()

asyncio.run(main())
```

Our take async generator keeps track of how many items we've seen so far with item_count. We then enter an async_for loop and yield each record that we see. Once we yield, we check item_count to see if we have yielded the number of items the caller requested. If we have, we return which ends the async generator. In our main coroutine, we can then use take within a normal async for loop. In this example, we use it to ask for the first five elements from the cursor, giving us the following output:

```
<Record product_id=1 product_name='among paper foot see shoe ride age'>
<Record product_id=2 product_name='major wait half speech lake won't'>
<Record product_id=3 product_name='war area speak listen horse past edge'>
<Record product_id=4 product_name='smell proper force road house planet'>
<Record product_id=5 product_name='ship many dog fine surface truck'>
Got the first five products!
```

While we've defined this in code ourselves, there is an open source library, aiostream that has this functionality and more for processing asynchronous generators. You can view the documentation for this library at aiostream.readthedocs.io.

5.8 Summary

In this chapter we've learned the basics around creating and selecting records in Postgres using an asynchronous database connection. You should now be able to take this knowledge and create concurrent database clients.

- We've learned how to use asyncpg to connect to a Postgres database.
- We've learned how to use various asyncpg coroutines to create tables, insert records and execute single queries.
- We've learned how to create a connection pool with asyncpg. This allows us to run multiple queries concurrently with asyncio's API methods such as gather. Using this we can potentially speed up our applications by running our queries in tandem.
- We've learned how to manage transactions with asyncpg. Transactions allow us to roll back any changes we make to a database as the result of a failure, keeping our database in a consistent state even when something unexpected happens.
- We've learned how to create asynchronous generators and how to use them for streaming database connections. We can use these two concepts together to work with large data sets that we can't fit in memory all at once.

6 Handling CPU bound work

This chapter covers

- **The multiprocessing library**
- **Creating process pools to handle CPU bound work**
- **Using async and await to manage CPU bound work**
- **Solving a map-reduce problem with asyncio**
- **Handling shared data between multiple processes with locks**
- **Improving the performance of work with both CPU and I/O bound operations**

Until now, we've been focused on performance gains we can get with asyncio when running I/O bound work concurrently. Running I/O bound work is asyncio's bread and butter, and with the way we've written code so far, we need to be careful not to run any CPU bound code in our coroutines. This seems like it severely limits asyncio, but the library is more versatile than just handling I/O bound work. Asyncio has an API for interoperating with Python's multiprocessing library. This lets us use async await syntax as well as asyncio APIs with multiple processes. Using this, we can get the benefits of the asyncio library even when using CPU bound code. This allows us to achieve performance gains for CPU intensive work, such as mathematical computations or data processing, letting us sidestep the global interpreter lock and take full advantage of a multicore machine.

In this chapter, we'll first learn about the multiprocessing module to get familiar with the concept of executing multiple processes. We'll then learn about process pool executors and how we can hook them into asyncio. We'll then take this knowledge and use it to solve a CPU intensive map/reduce problem. We'll also learn about managing shared state amongst multiple processes and will introduce the concept of locking to avoid concurrency bugs. Finally, we'll take a look at how to use multiprocessing to improve the performance of an application that is both I/O and CPU bound which we saw in the last chapter.

6.1 Introducing the multiprocessing library

In this book's first chapter, we introduced the global interpreter lock. The global interpreter lock prevents more than one piece of Python bytecode from running concurrently. This means that for anything other than I/O bound tasks, excluding some small exceptions, using multithreading won't provide any performance benefits the way it would in languages such as Java and C++. It seems like we might be stuck with no solution for our parallelizable CPU bound work in Python, but this is where the multiprocessing library jumps in to provide us with a solution.

Instead of our parent process spawning threads to parallelize things, we instead spawn subprocesses to handle our work. Each subprocess we spawn will have its own Python interpreter and be subject to the GIL, but instead of one interpreter we'll have several interpreters, each with their own GIL. Assuming we run on a machine with multiple CPU cores, this means that we can parallelize any CPU bound workload effectively. Even if we have more processes than cores, our operating system will use preemptive multitasking to allow our multiple tasks to run concurrently. This setup is both concurrent *and* parallel.

To get started with the multiprocessing library, let's start by running a couple of functions in parallel. We'll use a very simple CPU bound function that counts from zero to a large number to examine how the API works as well as the performance benefits.

Listing 6.1 two parallel processes with multiprocessing

```
import time
from multiprocessing import Process


def count(count_to: int) -> int:
    start = time.time()
    counter = 0
    while counter < count_to:
        counter = counter + 1
    end = time.time()
    print(f'Finished counting to {count_to} in {end-start}')
    return counter


if __name__ == "__main__":
    start_time = time.time()

    from_one_hundred_million = Process(target=count, args=(100000000,)) #A
    from_two_hundred_million = Process(target=count, args=(200000000,))

    from_one_hundred_million.start() #B
    from_two_hundred_million.start()

    from_one_hundred_million.join() #C
    from_two_hundred_million.join()

    end_time = time.time()
    print(f'Completed in {end_time-start_time}')
```

#A Create a process to run the countdown function
#B Start the process. This method returns instantly.

#C Wait for the process to finish. This method blocks until the process is done.

In listing 6.1 we create a simple count function which takes an integer and loops one by one until we count to the integer we pass in. We then create two processes, one to count to one hundred million and one to count to two hundred million. The Process class takes in two arguments, a `target` which is the function name we wish to run in the process and `args` which is a tuple of arguments we wish to pass to the function. We then call the `start` method on each process. This method returns instantly and will start running the process. In this example we start both processes right after one another. We then call the `join` method on each process. This will cause our main process to block until each process has finished. Without this, our program would exit almost instantly and terminate the subprocesses as nothing would be waiting for their completion. Listing 6.1 runs both count functions concurrently and assuming we're running on a machine with at least two CPU cores, we should see a speedup. When this code runs on a 2.5 GHz 8-core machine, we achieve the following results:

```
Finished counting down from 100000000 in 5.3844
Finished counting down from 200000000 in 10.6265
Completed in 10.8586
```

In total our countdown functions took a bit over sixteen seconds, but our application finished in just under eleven. This gives us a time savings over running sequentially of about five seconds. Of course, the results you see when you run this will be highly variable depending on your machine, but you should see something directionally equivalent to this.

You may have noticed we added `if __name__ == "__main__":` to our application where we haven't before. This is a quirk of the multiprocessing library and if you don't add this you may see an error that "An attempt has been made to start a new process before the current process has finished its bootstrapping phase.". The reason this happens is to prevent others who may import your code from accidentally launching multiple processes.

This gives us a decent performance gain; however, it is kind of clunky because we have to call `start` and `join` for each process we start. We also don't know which process will complete first, if we want to do something like `asyncio.as_completed` and process results as they finish, we're out of luck. The `join` method also does not return the value our target function returns – in fact currently we have no way to get the value our function returns!

This API will work for simple cases, but clearly won't work if we have functions where we want to get the return value out or want to process results as soon as they come in. Luckily, process pools provide a way for us to deal with this.

6.2 Using process pools

In the previous example we manually created processes and called their `start` and `join` methods to run and wait for them. We identified several issues with this approach, from code quality to not having the ability to access the results our process returned. The multiprocessing module has an API which lets us deal with this issue called process pools.

Process pools are a similar concept to the connection pools that we saw in the last chapter on databases. The difference in this case being that instead of a collection of connections to a database, we create a collection of Python processes that we can use to run functions in parallel. When we have a CPU bound function we wish to run in a process, we ask the pool directly to run it for us. Behind the scenes this will execute this function in an available process, running it and returning the return value of that function.

To see how a process pool works let's create a simple one and run a few 'hello world' style functions with it.

Listing 6.2 creating a process pool

```
from multiprocessing import Pool


def say_hello(name: str) -> str:
    return f'Hi there, {name}'


if __name__ == "__main__":
    with Pool() as process_pool: #A
        hi_jeff = process_pool.apply(say_hello, args=('Jeff',)) #B
        hi_john = process_pool.apply(say_hello, args=('John',))
        print(hi_jeff)
        print(hi_john)
```

#A create a new process pool
#B Run say_hello with the argument 'Jeff' in a separate process and get the result

In listing 6.2 we create a process pool using `with Pool() as process_pool`. This is a context manager because once we are done with the pool, we need to appropriately shut down the Python processes we created. If we don't do this, we run the risk of leaking processes which can cause resource utilization issues. When we instantiate this pool, it will automatically create a number of Python processes equal to the number of CPU cores on the machine you're running on. You can determine the number of CPU cores you have in Python by running the `multiprocessing.cpu_count()` function. If you need to create more or less processes than that, you can set the `processes` argument to any integer you'd like when you call `Pool()`. The default value is usually a good starting point.

Next, we use the `apply` method of the process pool to run our `say_hello` function in a separate process. This method looks similar to what we did previously with the `Process` class, we pass in a target function and a tuple of arguments. The difference here is this we don't need to start the process or call join on it ourselves. We also get the return value of our function back which we couldn't do in the previous example. Running this code, you should see the following printed out:

```
Hi there, Jeff
Hi there, John
```

This works, but there is a problem here. The apply method blocks until our function completes. That means that if each call to `say_hello` took 10 seconds our entire program's run time would be about 20 seconds because we've run things sequentially, negating the point of running in parallel. We can solve this problem by using process pool's `apply_async` method.

6.2.1 Using asynchronous results

In the previous example each call to apply blocked until our function completed. This won't work if we want to build a truly parallel workflow. To work around this, we can use the `apply_async` method instead. This method returns an `AsyncResult` instantly and will start running the process in the background. Once we have an `AsyncResult` we can use it's `get` method to block and get the results of our function call. Let's take our `say_hello` example and adopt it to use asynchronous results.

Listing 6.3 using async results with process pools

```
from multiprocessing import Pool


def say_hello(name: str) -> str:
    return f'Hi there, {name}'


if __name__ == "__main__":
    with Pool() as process_pool:
        hi_jeff = process_pool.apply_async(say_hello, args=('Jeff',))
        hi_john = process_pool.apply_async(say_hello, args=('John',))
        print(hi_jeff.get())
        print(hi_john.get())
```

When we call `apply_async` our two calls to `say_hello` get started in separate processes instantly. Then when we call the get method our parent process will block until each process returns a value. This lets things run concurrently, but what if `hi_jeff` took 10 seconds but `hi_john` only took one? In this case since we call `get` on `hi_jeff` first our program would block for 10 seconds before printing our `hi_john` message even though we were ready after only a second. If we want to respond to things as soon as they finish, we're left with an issue. What we really want is something like asyncio's `as_completed` in this instance. Next, let's see how to use process pool executors with asyncio so we can address this issue.

6.3 Using process pool executors with asyncio

In the previous section, we saw how to use process pools to concurrently run CPU intensive operations. These pools are good for simple use cases, but Python offers an abstraction on top of multiprocessing's process pools in the `concurrent.futures` module. This module contains *executors* for both processes and threads which can be used on their own but also interoperate with asyncio. To get started, we'll learn the basics of `ProcessPoolExecutor`, which is similar to `ProcessPool`. Then we'll see how to hook this in with asyncio so we can use the power of its API functions such as `gather`.

6.3.1 Introducing process pool executors

Python's process pool API is strongly coupled to processes, but multiprocessing is one of two ways to implement preemptive multitasking, the other being multithreading. What if we need to easily change the way in which we handle concurrency, seamlessly switching between processes and threads? If we want a design like this, it means we need to build an abstraction that encompasses the core of distributing work to a pool of workers that does not care if those workers are processes, threads or some other construct.

The `concurrent.futures` module provides this abstraction for us with the `Executor` abstract class. This class defines two simple methods for running work asynchronously. The first is `submit` which will take a callable and return a `Future` (note that this is not the same as asyncio futures, but is part of the concurrent.futures module) – this is equivalent to the `Pool.apply_async` method we saw in the last section. The second is `map` – this method will take a callable and a list of function arguments and then execute each argument in the list asynchronously. It returns an iterator of the results of our calls similarly to `asyncio.as_completed` in that results are available to us once they complete. Executor has two concrete implementations, `ProcessPoolExecutor` and `ThreadPoolExecutor`. Since we're using multiple processes to handle CPU bound work, we'll focus on `ProcessPoolExecutor`. In the next chapter, we'll examine threads with `ThreadPoolExecutor`.

To learn how a ProcessPoolExecutor works, we'll reuse our count example with a few small numbers and a few large numbers to show how results come in.

Listing 6.4 process pool executors

```
import time
from concurrent.futures import ProcessPoolExecutor


def count(count_to: int) -> int:
    start = time.time()
    counter = 0
    while counter < count_to:
        counter = counter + 1
    end = time.time()
    print(f'Finished counting to {count_to} in {end - start}')
    return counter


if __name__ == "__main__":
    with ProcessPoolExecutor() as process_pool:
        numbers = [1, 3, 5, 22, 100000000]
        for result in process_pool.map(count, numbers):
            print(result)
```

Much like before, we create a `ProcessPoolExecutor` in context manager. The number of workers also defaults to the number of CPU cores our machine has as process pools did. We then use `process_pool.map` with our `count` function and a list of numbers which we want to count to.

When we run this, we'll see that our calls to countdown with a low number will finish quickly and be printed out nearly instantly. Our call with `100000000` will however take much longer and will be printed out after the few small numbers, giving us the following output:

```
Finished counting down from 1 in 9.5367e-07
Finished counting down from 3 in 9.5367e-07
Finished counting down from 5 in 9.5367e-07
Finished counting down from 22 in 3.0994e-06
1
3
5
22
Finished counting down from 100000000 in 5.2097
100000000
```

While it seems that this works the same as `asyncio.as_completed` – the order of iteration is deterministic based on the order we passed in the `numbers` list. This means if `100000000` was our first number, we'd be stuck waiting for that call to finish before we could print out the other results that completed earlier. This means we aren't quite as responsive as `asyncio.as_completed`.

6.3.2 Process pool executors with the asyncio event loop

Now that we've know the basics of how process pool executors work, let's see how to hook them into the asyncio event loop. This will let us use the API functions such as `gather` and `as_completed` that we learned in chapter 4 to manage multiple processes.

Creating a process pool executor to use with asyncio is no different from what we just learned, we create one in within a context manager. Once we have a pool, we can use a special method on the asyncio event loop called `run_in_executor`. This method will take in a callable alongside an executor (which can be either a thread pool or process pool) and will run that callable inside the pool. It then returns an awaitable which we can use in an await statement or pass into an API function such as `gather`.

Let's implement our previous count example with a process pool executor. We'll submit multiple count tasks to the executor and wait for them all to finish with `gather`. `run_in_executor` only takes a callable and does not allow us to supply function arguments, to get around this we'll use partial function application to build countdown calls with zero arguments.

WHAT IS PARTIAL FUNCTION APPLICATION?

Partial function application is implemented in the `functools` module. Partial application takes a function that accepts some arguments and turns it into a function that accepts fewer arguments. It does this by "freezing" some arguments that we supply. As an example, our count function takes one argument. We can turn it into a function with zero arguments by using `functools.partial` with the parameter we want to call it with. If we want to have a call to `count(42)` but pass in no arguments we can say `call_with_42 = functools.partial(count, 42)` which we can then call as `call_with_42()`.

Listing 6.5 process pool executors with asyncio

```
import asyncio
from asyncio.events import AbstractEventLoop
from concurrent.futures import ProcessPoolExecutor
from functools import partial
from typing import List


def count(count_to: int) -> int:
    counter = 0
    while counter < count_to:
        counter = counter + 1
    return counter


async def main():
    with ProcessPoolExecutor() as process_pool:
        loop: AbstractEventLoop = asyncio.get_event_loop()
        nums = [1, 3, 5, 22, 100000000]
        calls: List[partial[int]] = [partial(count, num) for num in nums]
        call_coros = []

        for call in calls:
            call_coros.append(loop.run_in_executor(process_pool, call))

        results = await asyncio.gather(*call_coros)

        for result in results:
            print(result)


if __name__ == "__main__":
    asyncio.run(main())
```

#A Create a partially applied function for countdown with its argument
#B Submit each call to the process pool and append it to a list
#C Wait for all results to finish

We first create a process pool executor as we did before. Once we have this, we get the asyncio event loop, since `run_in_executor` is a method on the `AbstractEventLoop`. We then partially apply each number in nums to the count function since we can't call count directly. Once we have `count` function calls then we can submit them to the executor. We loop over these calls, calling `loop.run_in_executor` for each partially applied count function and keep track of the awaitable it returns in `call_coros`. We then take this list and wait for everything to finish with `asyncio.gather`.

If we had wanted, we could also use `asyncio.as_completed` to get the results from the subprocesses as they completed. This would solve the problem we saw earlier with process pool's `map` method where if we had a task that took a long time.

We've now seen all we need to start using process pools with asyncio. Next, let's take a look at how to improve the performance of a real-world problem with multiprocessing and asyncio.

6.4 Solving a map reduce problem with asyncio

To understand the type of problem map-reduce solves, we'll introduce a hypothetical real-world problem. We'll then take that understanding and use it to solve a similar problem with a large, freely available data set.

Going back to our example of an ecommerce storefront, we'll pretend our site receives a lot of text data through our customer support portal's "questions and concerns" field. Since our site is successful, this data set of customer feedback is multiple terabytes in size and growing every day.

To better understand the common issues our users are facing, we've been tasked to find the most frequently used words in this data set. A simple solution would be to use a single process to loop through each comment and keep track how many times each word occurs. This will work, but since our data is large, going through this in serial could take a long time. Is there a faster way we could approach this type of problem?

This is the exact kind of issue that map-reduce aims to solve. The map-reduce programming model solves a problem by first partitioning up a large data set into smaller chunks. We can then solve our problem for that smaller subset of data instead of entire set – this is known as *mapping* as we 'map' our data to a partial result.

Once the problem for each subset is solved, we can then combine the results into a final answer. This step is known as *reducing* as we 'reduce' multiple answers into a single one. Counting the frequency of words in a large text data set is a canonical map-reduce problem. If we have a large enough dataset, splitting it into smaller chunks can yield performance benefits as each map operation can run in parallel.

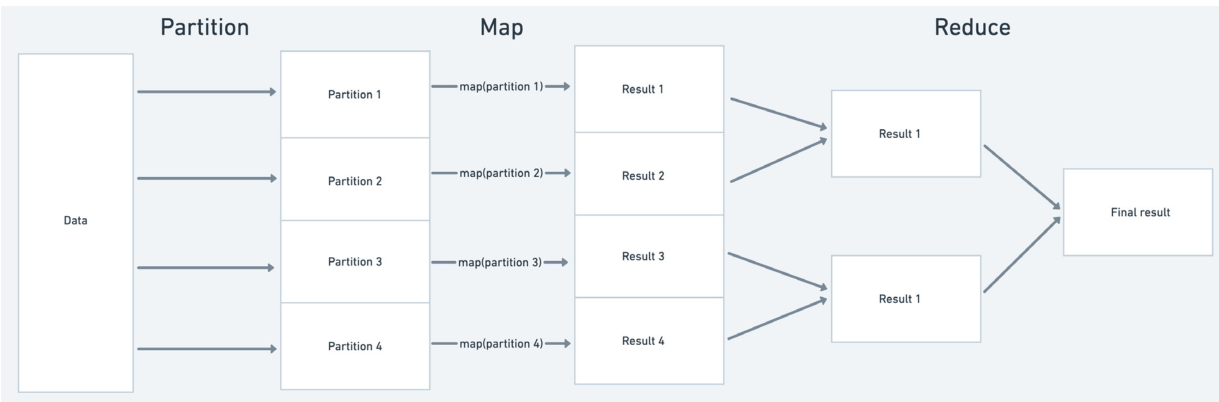


Figure 6.1 A large set of data is split into partitions, then a map function produces intermediate results. These intermediate results are combined into a final result.

Systems such as Hadoop and Spark exist to perform map reduce operations in a cluster of computers for truly large datasets. However, many workloads can be processed on one computer with multiprocessing. In this section, we'll see how to implement a map-reduce workflow with multiprocessing to find how frequently certain words have appeared in text since the year 1500.

6.4.1 A simple map reduce example

To fully understand how map-reduce works, let's walk through a small concrete example. Let's say we have text data on each line of a file. For this example, we'll pretend we have four lines to handle:

```
I know what I know.
I know that I know.
I don't know that much.
They don't know much.
```

We'd like to count how many times each distinct word occurs in this data set. This example is small enough we could solve it with a simple for loop, but let's approach it with a map-reduce model.

First, we need to partition this data set down into smaller chunks. For simplicity, we'll just define a smaller chunk as one line of text. Next, we need to define the mapping operation. Since we want to count word frequencies, we'll split the line of text on a space. This will get us an array of each individual word in the string. We can then loop over this, keeping track of each distinct word in the line of text in a dictionary.

Finally, we need to define a reduce operation. This will take one or more results from our map operation and combine them into an answer. In this example, we need to take two dictionaries from our map operation and combine them into one. If a word exists in both dictionaries, we add their word counts together, if not, we copy over the word count to the result dictionary. Once we've defined these operations, we can then run our map operation on each individual line of text and our reduce operation on each pair of results from our map. Let's see how to do this example in code with the four lines of text we introduced earlier.

Listing 6.6 single-threaded map reduce

```
import functools
from typing import Dict


def map_frequency(text: str) -> Dict[str, int]:
    words = text.split(' ')
    frequencies = {}
    for word in words:
        if word in frequencies:
            frequencies[word] = frequencies[word] + 1 #A
        else:
            frequencies[word] = 1 #B
    return frequencies


def merge_dictionaries(first: Dict[str, int],
                       second: Dict[str, int]) -> Dict[str, int]:
    merged = first
    for key in second:
        if key in merged:
```

```
            merged[key] = merged[key] + second[key] #C
        else:
            merged[key] = second[key] #D
    return merged


lines = ["I know what I know",
         "I know that I know",
         "I don't know much",
         "They don't know much"]

mapped_results = [map_frequency(line) for line in lines] #E

for result in mapped_results:
    print(result)

print(functools.reduce(merge_dictionaries, mapped_results)) #F
```

#A If we have the word in our frequency dictionary, add one to the count.
#B If we do not have the word in our frequency dictionary, set it's count to one.
#C If the word is in both dictionaries, combine frequency counts.
#D If the word is not in both dictionaries, copy over the frequency count.
#E For each line of text, perform our map operation
#F Reduce all our intermediate frequency counts into one final result

For each line of text, we apply our map operation, giving us the frequency count for each line of text. Once we have these mapped partial results, we can begin to combine them. We use our merge function `merge_dictionaries` along with the `functools.reduce` function. This will take our intermediate results and add them together into a final result, giving us the following output:

```
Mapped results:
{'I': 2, 'know': 2, 'what': 1}
{'I': 2, 'know': 2, 'that': 1}
{'I': 1, "don't": 1, 'know': 1, 'much': 1}
{'They': 1, "don't": 1, 'know': 1, 'much': 1}

Final Result:
{'I': 5, 'know': 6, 'what': 1, 'that': 1, "don't": 2, 'much': 2, 'They': 1}
```

Now that we understand the basics with a toy map-reduce problem, we'll see how to apply this to a real-world data set where multiprocessing can yield performance improvements.

6.4.2 The Google Books Ngram Dataset

We'll need a sufficiently large set of data to process to see the benefits of map-reduce with multiprocessing. If our data is too small, we'll see no benefit from map-reduce and will likely see performance degradation from the overhead of managing processes. A dataset of a few gigabytes uncompressed should be enough for us to show a meaningful performance gain.

The Google Books Ngram dataset is a sufficiently large data set for this purpose. To understand what this data set is we'll first define what an n-gram is.

An n-gram is a concept from natural language processing and is a phrase of N words from a sample of given text. The phrase “The fast dog” has 6 n-grams. Three 1-grams or *unigrams* each of one single word “the”, “fast” and “dog”, two 2-grams or *digrams* “the fast” and “fast dog” and one 3-gram or *trigram* “The fast dog.”

The Google Books Ngram dataset is a scan of ngrams from a set of over 8 million books going back to the year 1500, comprising more than 6% of all books published. It counts the number of times a distinct n-gram appears in text, grouped by year it appears. This dataset has everything from unigrams to 5-grams in tab-separated format. Each line of this dataset has an n-gram, the year when it was seen, the number of times it was seen and how many books it occurred in. Let’s take a look at the first few entries in the unigram dataset for the word aardvark:

```
Aardvark        1822    2       1
Aardvark        1824    3       1
Aardvark        1827    10      7
```

This means that in the year 1822, the word aardvark appeared twice in one book. Then, in 1827, the word aardvark appeared ten times in seven different books. The dataset has many more entries for aardvark (for example, aardvark occurred 1200 times in 2007), demonstrating the upwards trajectory of aardvarks in literature over the years.

For the sake of this example, we’ll count the occurrences of single words (unigrams) for words that start with ‘a’. This dataset is approximately 1.8GB in size. We’ll aggregate this to the amount of times each word has been seen in text since 1500. We’ll use this to answer the question “How many times has the word aardvark appeared in literature since the year 1500?”. The relevant file we want to work with is downloadable in the data repository for this book. You can also download it, or any other part of the dataset from http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

6.4.3 Mapping and reducing with asyncio

To have a baseline to compare ourselves to, let’s first write a synchronous version to count the frequencies of words. We’ll then use this frequency dictionary to answer the question ‘How many times has the word aardvark appeared in text since 1500?’. We’ll first load the entire contents of the dataset into memory. Then, we can use a dictionary to keep track a mapping of words to the total time they have occurred. For each line of our file, if the word on that line is in our dictionary, we add to the count in our dictionary with the count for that word. If it is not, we add the word and the count on that line to the dictionary.

Listing 6.7 counting frequencies of words that start with ‘a’

```
import time

freqs = {}

with open('googlebooks-eng-all-1gram-20120701-a') as f:
    lines = f.readlines()

    start = time.time()
```

```
    for line in lines:
        data = line.split('\t')
        word = data[0]
        count = int(data[2])
        if word in freqs:
            freqs[word] = freqs[word] + count
        else:
            freqs[word] = count

    end = time.time()
    print(f'{end-start:.4f}')
```

To test how long the CPU bound operation takes we'll only time how long the frequency counting takes and won't include the length to load the file. For multiprocessing to be a viable solution, we need to run on a machine with sufficient CPU cores to make parallelization worth the effort. To see sufficient gains, we'll likely need a machine with more CPUs than most laptops have. To test on such a machine, we'll use a large EC2 instance on AWS.

WHAT IS AWS?

Aws, short for Amazon Web Services, is a cloud computing service run by Amazon. AWS is a collection of cloud services that enable you to handle tasks from file storage to large-scale machine learning jobs, all without managing your own physical servers. One such service offered is EC2, short for Elastic Compute Cloud. Using this you can rent a virtual machine in AWS to run any application you want, specifying how many CPU cores and memory you need on your virtual machine. You can learn more about AWS and EC2 at https://aws.amazon.com/ec2.

We'll test on a c5ad.8xlarge instance, at the time of writing this machine has 32 CPU cores, 64 GB of RAM and an SSD drive. On this instance, the above script on this takes approximately 76 seconds. Let's see if we can do any better with multiprocessing and asyncio. If you run this on a machine with less CPU cores or other resources, your results may vary.

The first step we need to do is take our data set and partition it into a smaller set of chunks. Let's define a partition generator which can take our large list of data and grab chunks of arbitrary size.

```
def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]
```

We can use this partition generator to create slices of data which are `chunk_size` long. We'll use this to generate the data to pass into our map functions, which we will then run in parallel. Next, let's define our map function – this is almost the same as our map function from the previous example, just adjusted to work with our data set.

```
def map_frequencies(chunk: List[str]) -> Dict[str, int]:
```

```
    counter = {}
    for line in chunk:
        word, _, _, count = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter
```

For now, we'll keep our reduce operation exactly the same as in the previous example. We now have all the blocks we need to parallelize our map operations. We'll create a process pool, partition our data into chunks and for each partition run `map_frequencies` in a worker on the pool. We have almost everything we need, but one question remains: what partition size should I use?

There isn't an easy answer for this. One handwavy rule of thumb is the goldilocks approach - the partition should not be too big or too small. The reason the partition size should not be small is that when we create our partitions they are serialized (pickled) and sent to our worker processes, then the worker process unpickles them. The process of serializing and deserializing this data can take up a significant amount of time, quickly eating into any performance gains if we do it too often. For example, a chunk size of two would be a poor choice as we would have nearly one million pickle and unpickle operations.

We also don't want the partition size to be too large otherwise we might not fully utilize the power of our machine. For example, if we have 10 CPU cores but only create two partitions, we're missing out on 8 cores that could run workloads in parallel.

For this example, we'll chose a partition size of 60,000 as this seems to offer reasonable performance for the AWS machine we're using based on benchmarking. If you're considering this approach for your data processing task, you'll need to test out a few different partition sizes to find the one for your data and the machine you're running on or develop a heuristic algorithm for determining the right partition size.

We can now combine all these parts together with a process pool and the event loop's `run_in_executor` coroutine to parallelize our map operations.

Listing 6.8 parallel map reduce with process pools

```
import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List


def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]


def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
```

```
        word, _, _, count = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter


def merge_dictionaries(first: Dict[str, int],
                       second: Dict[str, int]) -> Dict[str, int]:
    merged = first
    for key in second:
        if key in merged:
            merged[key] = merged[key] + second[key]
        else:
            merged[key] = second[key]
    return merged


async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a') as f:
        contents = f.readlines()
        loop = asyncio.get_event_loop()
        tasks = []
        start = time.time()
        with concurrent.futures.ProcessPoolExecutor() as pool:
            for chunk in partition(contents, partition_size):
                tasks.append(loop.run_in_executor(pool, functools.partial(map_frequencies,
       chunk))) #A

            intermediate_results = await asyncio.gather(*tasks) #B
            final_result = functools.reduce(merge_dictionaries, intermediate_results) #C

            print(f"Aardvark has appeared {final_result['Aardvark']} times.")

            end = time.time()
            print(f'Map reduce took: {(end - start):.4f} seconds')


if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))
```

#A For each partition run our map operation in a separate process
#B Wait for all map operations to complete
#C Reduce all our intermediate map results into a final result

In our `main` coroutine we create a process pool and partition the data. For each partition, we launch a `map_frequencies` function in a separate process. We then use `asyncio.gather` to wait for all intermediate dictionaries to finish. Once all our map operations are complete, we run our reduce operation to produce our final result.

Running this map-reduce on the instance we described, this code completes in roughly 18 seconds, giving us nearly a 500% speedup compared with our serial version. This is quite a nice performance gain for not a whole lot more code! You may also wish to experiment with a machine with more CPU cores to see if you can further improve the performance of this algorithm.

You may notice in this implementation that we still have some CPU bound work happening in our parent process that is parallelizable. Our reduce operation takes thousands of dictionaries and combines them together all in one shot. We can apply the partitioning logic we used on the original dataset and split these dictionaries into chunks and combine them across multiple processes. Let's write a new reduce function that does just that. In this function, we'll partition the list and call reduce on each chunk in a worker process. Once this completes, we'll keep partitioning and reducing until we have one dictionary remaining. In this listing, we've removed the partition, map and merge functions for brevity.

Listing 6.9 parallelizing the reduce operation

```
import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List


async def reduce(loop, pool, counters, chunk_size) -> Dict[str, int]:
    chunks: List[List[Dict]] = list(partition(counters, chunk_size)) #A
    reducers = []
    while len(chunks[0]) > 1:
        for chunk in chunks:
            reducer = functools.partial(functools.reduce, merge_dictionaries, chunk) #B
            reducers.append(loop.run_in_executor(pool, reducer))
        reducer_chunks = await asyncio.gather(*reducers) #C
        chunks = list(partition(reducer_chunks, chunk_size)) #D
        reducers.clear()
    return chunks[0][0]


async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a') as f:
        contents = f.readlines()
        loop = asyncio.get_event_loop()
        tasks = []
        with concurrent.futures.ProcessPoolExecutor() as pool:
            start = time.time()

            for chunk in partition(contents, partition_size):
                tasks.append(loop.run_in_executor(pool, functools.partial(map_frequencies,
       chunk)))

            intermediate_results = await asyncio.gather(*tasks)
            final_result = await reduce(loop, pool, intermediate_results, 500)

            print(f"Aardvark has appeared {final_result['Aardvark']} times.")

            end = time.time()
            print(f'Map reduce took: {(end - start):.4f} seconds')


if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))
```

#A Partition the dictionaries into parallelizable chunks

#B Reduce each partition into a single dictionary
#C Wait for all reduce operations to complete
#D Partition the results again and start a new iteration of the loop

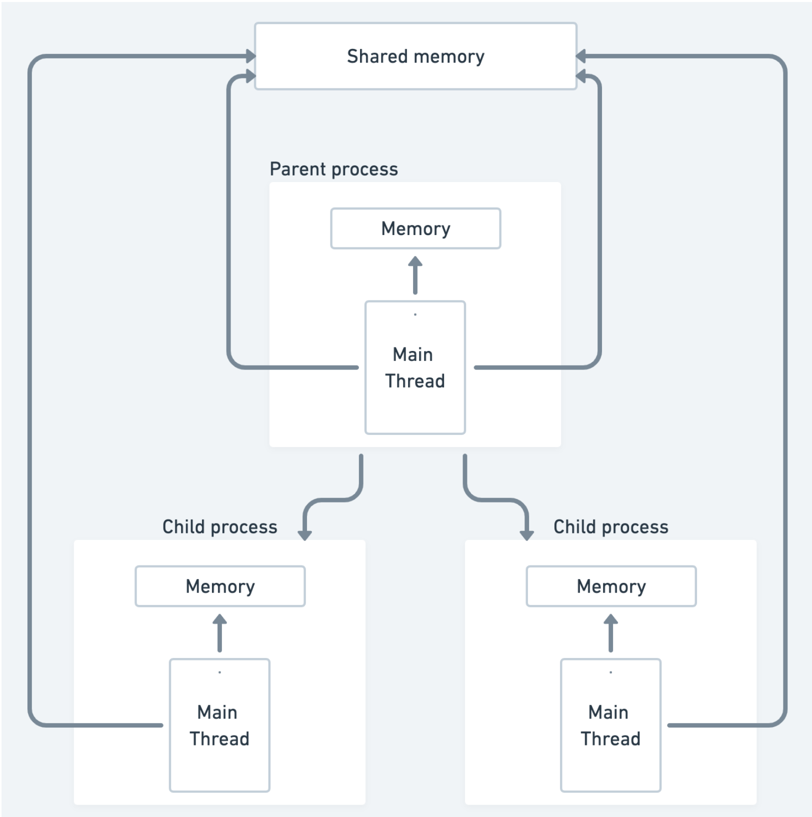
If we run with this parallelized reduce, we may see some small performance gain, or only a little depending on the machine you run on. In this instance, the overhead of pickling the intermediate dictionaries, and sending them to the children processes will eat away much of the time savings by running in parallel. This optimization may not do much to make this particular problem perform better, however, if our reduce operation was more CPU intensive, or we had a much larger data set, this approach can yield benefits.

Our multiprocessing approach has clear performance benefits over a synchronous approach, but right now there isn't an easy way to see how many map operations we've completed at any given time. In the synchronous version, we would only need to add a counter we incremented for every line we processed to see how far along we were. Since multiple processes by default do not share any memory, how can we create a counter to track our job's progress?

6.5 Shared data and locks

In the first chapter of this book, we went over the fact that in multiprocessing each process has its own memory separate from other processes. This presents a challenge to us when we have shared state we need to keep track of. So how can we share data between processes if their memory spaces are all distinct?

Multiprocessing supports a concept called shared memory objects. A shared memory object is a chunk of memory allocated that a set of separate processes can access. Each process can then read and write into that memory space as needed.



Shared state is complicated and can lead to hard to reproduce bugs if not properly implemented. Generally speaking, it is best to avoid this if at all possible. That said, sometimes it is necessary to introduce shared state, one such instance is a shared counter.

To learn more about shared data, we'll take our map-reduce example from above and keep a counter of how many map operations we've completed. We'll then periodically output this number to show how far along we are to the user.

6.5.1 Sharing data and race conditions

Multiprocessing supports two kinds of shared data, values and array. A value is a singular value, such as an integer or floating-point number. An array is an array of singular values. The types of data that we can share in memory are limited by the types defined in the array module, available at https://docs.python.org/3/library/array.html#module-array.

To create a value or array, we first need to use the typecode from the array module which is just a char. Let's create two shared pieces of data, one integer value and one integer array. We'll then create two processes to increment each of these shared pieces of data in parallel.

Listing 6.10 shared values and arrays

```
from multiprocessing import Process, Value, Array


def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1


def increment_array(shared_array: Array):
    for index, integer in enumerate(shared_array):
        shared_array[index] = integer + 1


if __name__ == '__main__':
    integer = Value('i', 0)
    integer_array = Array('i', [0, 0])

    procs = [Process(target=increment_value, args=(integer,)),
             Process(target=increment_array, args=(integer_array,))]

    [p.start() for p in procs]
    [p.join() for p in procs]

    print(integer.value)
    print(integer_array[:])
```

In listing 610x we create two processes, one to increment our shared integer value and one to increment each element in our shared array. Once our two subprocesses complete we print out the data.

Since our two pieces of data are never touched by different processes, this code works fine. Will this code continue to work if we have multiple processes modifying the same shared data? Let’s test this out by creating two processes to increment a shared integer value in parallel. We’ll run this code repeatedly in a loop to see if we get consistent results. Since we have two processes each incrementing a shared counter by one, we expect the shared value to always be two once the processes finish.

Listing 6.11 incrementing a shared counter in parallel

```
from multiprocessing import Process, Value


def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1


if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integer,)),
                 Process(target=increment_value, args=(integer,))]

        [p.start() for p in procs]
        [p.join() for p in procs]
        print(integer.value)
```

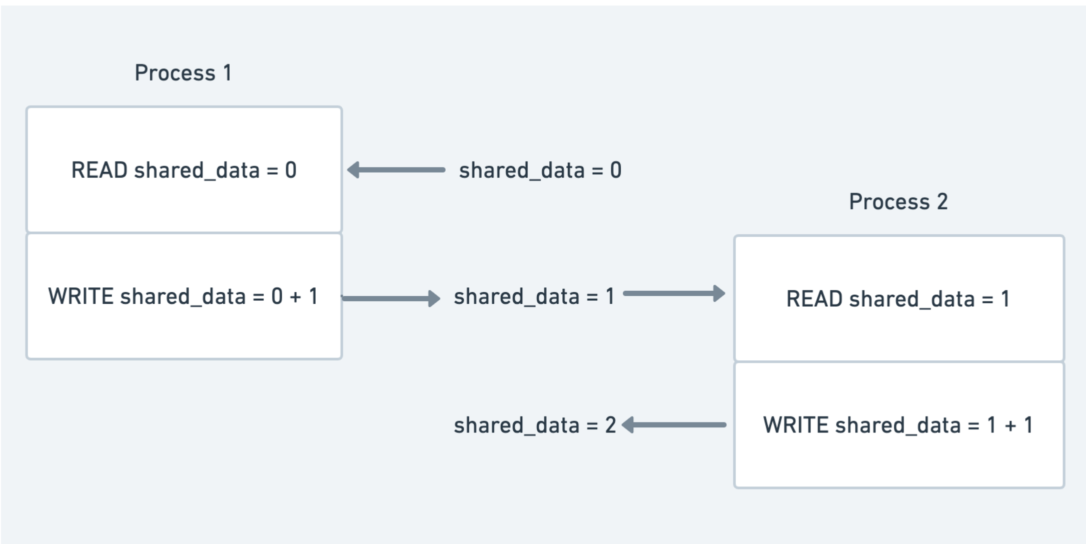
```
        assert(integer.value == 2)
```

While you will see different output because this problem is nondeterministic, at some point you should see that the result isn’t always two.

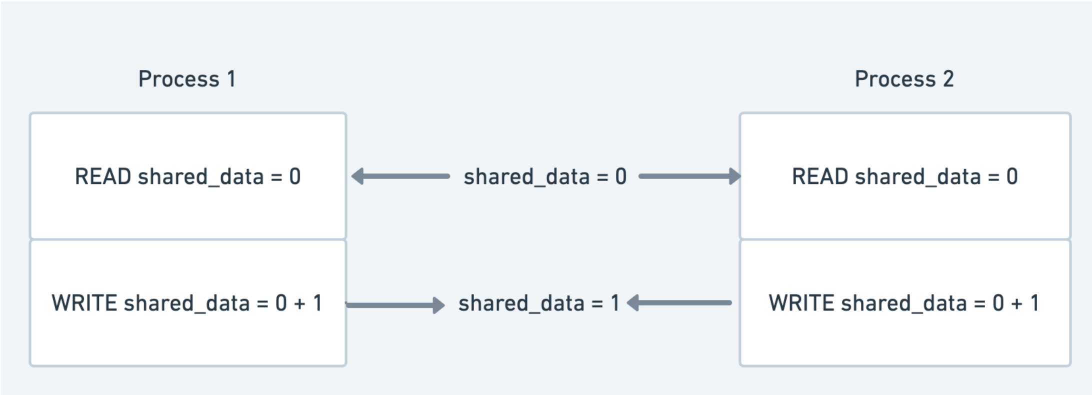
```
2
2
2
Traceback (most recent call last):
  File "listing_6_11.py", line 17, in <module>
    assert(integer.value == 2)
AssertionError
1
```

Sometimes our result is one! Why is this? What we’ve encountered is called a *race condition*. A race condition occurs when the outcome of a set of operations is dependent on which operation finishes first. You can imagine the operations as racing against one another, if the operations win the race in the right order, everything works fine. If they win the race in the wrong order, we get bizarre behavior.

So where is the race occurring in our example? The problem lies in that incrementing a value involves both a read and a write operation. To increment a value, we first need to read the value, add one to it and then write the result back to memory. The value each process sees in the shared data is entirely dependent on when it reads the shared value. If the processes run in the following order, everything works fine:



In this example, process one increments the value just before process two reads it and wins the race. Since process two finishes second, this means that it will see the correct value of one and will add to it, producing the correct final value. What happens if there is a tie in our virtual race?



In this instance, process one and two both read the initial value of zero. They then increment that value to one and write it back at the same time, producing the incorrect value.

You may ask "but our code is only one line, why is it two operations!?" Under the hood incrementing is written as two operations which causes this issue. This makes it *non-atomic* or not *thread safe*. This isn't easy to figure out, but an explanation of which operations are atomic and non-atomic is available at https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe.

These types of errors are tricky because they are often hard to reproduce. They aren't like normal bugs since they depend on the order in which our operating system runs things, which is out of our control when we use multiprocessing. So how do we fix this nasty bug?

6.5.2 Synchronizing with locks

We can avoid race conditions by *synchronizing* access to any shared data we want to modify. What does it mean to synchronize access? Going back to our race example, it means that we control access to any shared data so that any operations we have finish the race in an order that makes sense. If we're in a situation where a tie between two operations could occur, we explicitly block the second operation from running until the first completes, guaranteeing operations to finish the race in a consistent manner. You can imagine this as a referee at a the finish line seeing a tie is about to happen and telling the runners "hold up a minute, one racer at a time!" and picking one runner to wait while the other crosses the finish line.

One mechanism for synchronizing access to shared data is a *lock*, also known as a *mutex* (short for mutual exclusion). These structures allow for a single process to "lock" a section of code, preventing other processes from running that code. The locked section of code is commonly called a *critical section*. This means that if one process is executing the code of a locked section, and a second process tries to access that code, the second process will need to wait (blocked by the referee) until the first process is finished with the locked section.

Locks support two primary operations, *acquiring* and *releasing*. When a process acquires a lock, it is guaranteed that it will be the only process running that particular section of code. Once section of code that needs synchronized access is finished, we release the lock. This allows other processes to acquire the lock and run any code in the critical section. If a process tries to run code that is locked by another process, acquiring the lock will block until the other process releases that lock. Going back to our counter race condition example, let’s visualize what happens when two processes try and acquire a lock at roughly the same time and see how it prevents the counter from getting the wrong value.

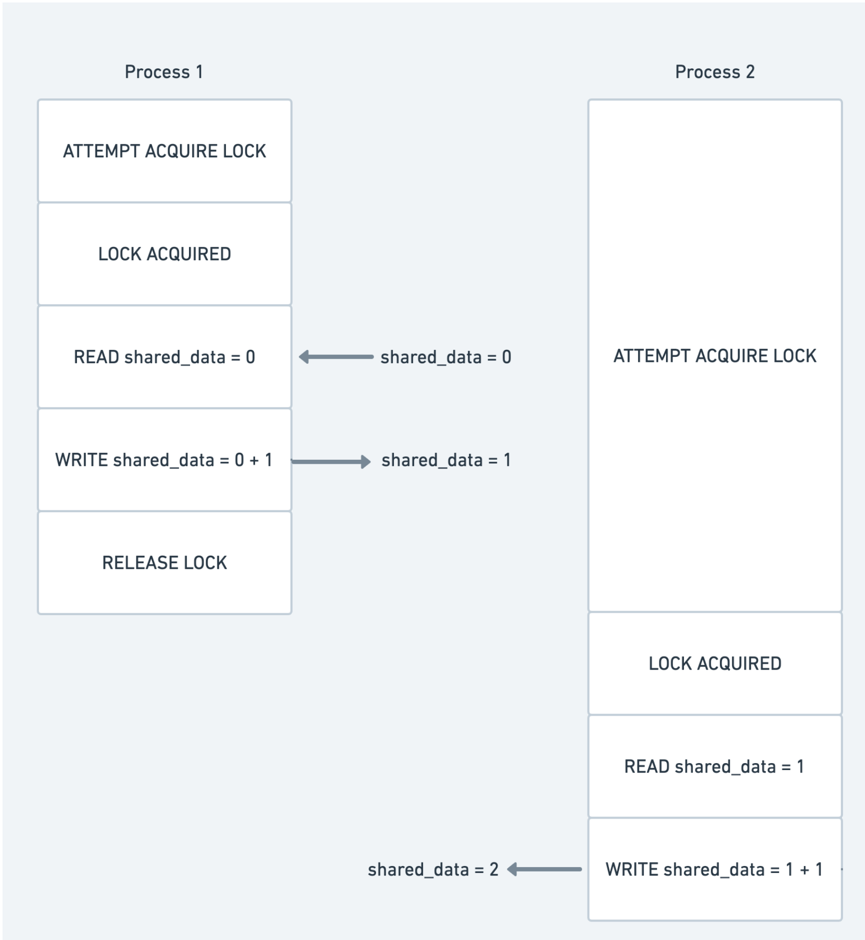


Figure 6.2 Process two is blocked from reading shared data until process one releases the lock.

In this diagram, process one first acquires the lock successfully and reads and increments the shared data. The second process tries to acquire the lock but is blocked from advancing further until the first process releases the lock. Once the first process releases the lock, the second process can successfully acquire the lock and increment the shared data. This prevents the race condition because the lock prevents more than one process from reading and writing the shared data at the same time.

So how do we implement this synchronization with our shared data? The multiprocessing API implementors thought of this and nicely included a method to get a lock on both Value and Array. To acquire a lock we call `get_lock().acquire()` and to release a lock we call `get_lock().release()`. Let's apply this to our previous example to fix our bug.

Listing 6.x acquiring and releasing a lock

```
from multiprocessing import Process, Value


def increment_value(shared_int: Value):
    shared_int.get_lock().acquire()
    shared_int.value = shared_int.value + 1
    shared_int.get_lock().release()


if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integer,)),
                 Process(target=increment_value, args=(integer,))]

        [p.start() for p in procs]
        [p.join() for p in procs]
        print(integer.value)
        assert (integer.value == 2)
```

When we run this code, every value we get should be two. We've fixed our race condition! Note that locks are also context managers, to clean up our code we could have written `increment_value` using a with block. This will acquire and released the lock for us automatically:

```
def increment_value(shared_int: Value):
    with shared_int.get_lock():
        shared_int.value = shared_int.value + 1
```

You may notice that we have taken concurrent code and have just forced it to be sequential, negating the value of running in parallel. This is an important observation and is a caveat of synchronization and shared data in concurrency in general. In order to avoid race conditions, we *have* to make our parallel code sequential in critical sections. This can hurt the performance of our multiprocessing code. Care must be taken to only lock the sections that absolutely need it so that other parts of the application can execute concurrently. When faced with a race condition bug, it is easy protect all of your code with a lock. This will "fix" the problem but will likely degrade your application's performance.

6.5.3 Sharing data with process pools

We've just seen how to share data within a couple of processes, how do we apply this knowledge to process pools? Process pools operate a bit differently than creating processes manually which winds up posing a small challenge with shared data. Why is this?

When we submit a task to a process pool, it may not run immediately because the processes in the pool may be busy with other tasks. How does the process pool handle this? Under the hood, process pool executors keep a queue of tasks to manage this. When we submit a task to the process pool, its arguments are pickled (serialized) and put on the task queue. Then, each worker process asks for a task from the queue when it is ready for work. When a worker process pulls a task off the queue, it unpickles (deseralizes) the arguments and begins to execute the task.

Shared data is shared amongst worker processes by definition, so pickling and unpickling it to send back and forth between processes makes little sense. In fact, both `Value` and `Array` objects are not pickleable, so if we try to pass the shared data in as arguments to our functions as we did before, we'll get an error along the lines of "can't pickle Value objects".

To handle this, we'll need to put our shared counter in a global variable and somehow let our worker processes know about it. We can do this with process pool initializers. These are special functions that are called when each process in our pool starts up. Using this we can create a reference to the shared memory that our parent process created. We can pass this function in when we create a process pool, let's create a simple example that increments a counter to see how this works.

Listing 6.x initializing a process pool

```
from concurrent.futures import ProcessPoolExecutor
import asyncio
from multiprocessing import Value

shared_counter: Value


def init(counter: Value):
    global shared_counter
    shared_counter = counter


def increment():
    with shared_counter.get_lock():
        shared_counter.value += 1


async def main():
    counter = Value('d', 0)
    with ProcessPoolExecutor(initializer=init,
                             initargs=(counter,)) as pool: #A
        await asyncio.get_event_loop().run_in_executor(pool, increment)
        print(counter.value)


if __name__ == "__main__":
    asyncio.run(main())
```

#A This tells the pool to execute the function init with the argument counter for each process

We first define a global variable `shared_counter` which will contain the reference to the shared `Value` object we create. In our `init` function we take in a `Value` and initialize `shared_counter` to that value. Then, in our main coroutine we create the counter and initialize it to zero and pass in our `init` function and our counter to the `initializer` and `initargs` parameter when creating the process pool. The `init` function will be called for each process that the process pool creates, correctly initializing our `shared_counter` to the one we created in our main coroutine.

You may ask yourself, "Why do we need to bother with all this? Can't we just initialize the global variable as `shared_counter: Value = Value('d', 0)` instead of leaving it empty and move on with our lives?" The reason we can't do this is when each process is created, the script we created it from is run per each process again. This means that each process that starts will execute `shared_counter: Value = Value('d', 0)` meaning that if we have 100 processes we'd get 100 `shared_counter` values each set to zero, resulting in some strange behavior.

Now that we know how to initialize shared data properly with a process pool, let's see how to apply this to our map-reduce application. We'll create a shared counter that we'll increment each time a map operation completes. We'll also create a 'progress reporter1 task that will run in the background and ouput our progress to the console every second. For this example, we'll import some of our code around partitioning and reducing so we don't repeat ourselves.

Listing 6.11 keeping track of map operation progress

```
from concurrent.futures import ProcessPoolExecutor
import functools
import asyncio
from multiprocessing import Value
from typing import List, Dict

from chapter_06.map_reduce import partition, merge_dictionaries

map_progress: Value


def init(progress: Value):
    global map_progress
    map_progress = progress


def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, _, count = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)

    with map_progress.get_lock():
        map_progress.value += 1

    return counter
```

```
async def progress_reporter(total_partitions: int):
    while map_progress.value < total_partitions:
        print(f'Finished {map_progress.value}/{total_partitions} map operations')
        await asyncio.sleep(1)


async def main(partiton_size: int):
    global map_progress

    with open('googlebooks-eng-all-1gram-20120701-a') as f:
        contents = f.readlines()
        loop = asyncio.get_event_loop()
        tasks = []
        map_progress = Value('i', 0)

        with ProcessPoolExecutor(initializer=init,
                                 initargs=(map_progress,)) as pool:
            total_partitions = len(contents) // partiton_size
            reporter = asyncio.create_task(progress_reporter(total_partitions))

            for chunk in partition(contents, partiton_size):
                tasks.append(loop.run_in_executor(pool, functools.partial(map_frequencies,
       chunk)))

            counters = await asyncio.gather(*tasks)

            await reporter

            final_result = functools.reduce(merge_dictionaries, counters)

            print(f"Aardvark has appeared {final_result['Aardvark']} times.")


if __name__ == "__main__":
    asyncio.run(main(partiton_size=60000))
```

The main change from our original map-reduce implementation, aside from initializing a shared counter, is inside our `map_frequencies` function, once we have finished counting all words in that chunk, we acquire the lock for the shared counter and increment it. We also added a `progress_reporter` coroutine, this will run in the background and report how many jobs we've completed every second. When running this you should see output similar to the following:

```
Finished 17/1443 map operations
Finished 144/1443 map operations
Finished 281/1443 map operations
Finished 419/1443 map operations
Finished 560/1443 map operations
Finished 701/1443 map operations
Finished 839/1443 map operations
Finished 976/1443 map operations
Finished 1099/1443 map operations
Finished 1230/1443 map operations
Finished 1353/1443 map operations
Aardvark has appeared 6443 times.
```

We now know how to use multiprocessing with asyncio to improve the performance of CPU intensive work. What happens if we have a workload that has work that has both heavily CPU and IO bound operations? We can use multiprocessing, but is there a way for us to combine the ideas of multiprocessing and a single-threaded concurrency model to further improve performance?

6.6 Multiple processes, multiple event loops

While multiprocessing is mainly useful for CPU bound tasks, it can have benefits for workloads that are I/O bound as well. Let's take our example of running multiple SQL queries concurrently in from our last chapter. Can we use multiprocessing to further improve its performance? Let's take a look at what its CPU usage graph looks like on a single core.

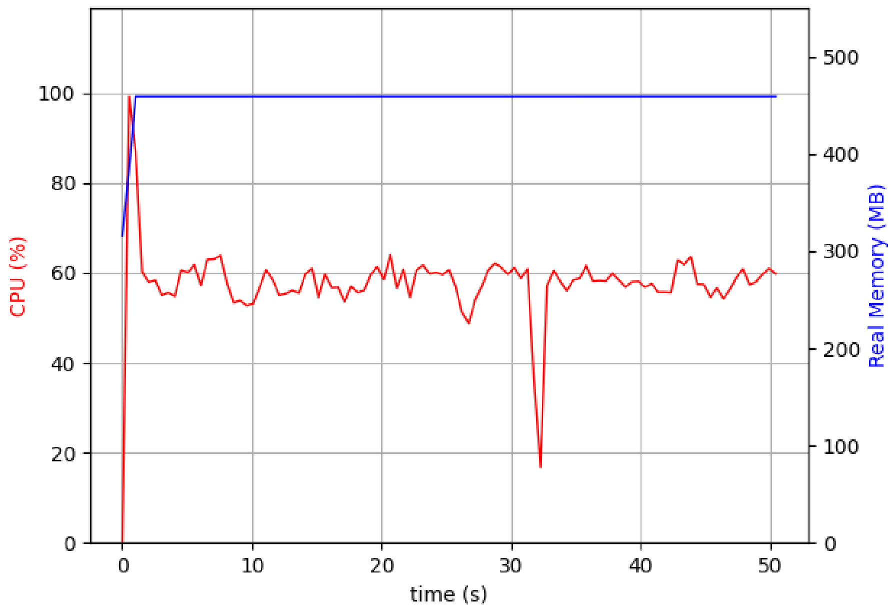


Figure 6.3 The CPU utilization graph for the code in listing 5.8

While this code is mostly making I/O bound queries to our database, there is still a decent amount of CPU utilization happening. Why is this? In this instance, there is still a decent amount of work happening to process the raw results we get from Postgres leading to higher CPU utilization. Since we're single-threaded, while this CPU bund work is happening, our event loop isn't processing results from other queries. This poses a potential throughput issue for us. If we issue 10,000 SQL queries concurrently, but we can only process one result at a time, we may end up with a backlog of query results to process.

Is there a way for us to improve our throughput by using multiprocessing? Using multiprocessing, each process has its own thread and its own Python interpreter, this opens up the opportunity to create one event loop per each process in our pool. With this model, we can distribute our queries over several processes. This will spread the CPU load across multiple processes.

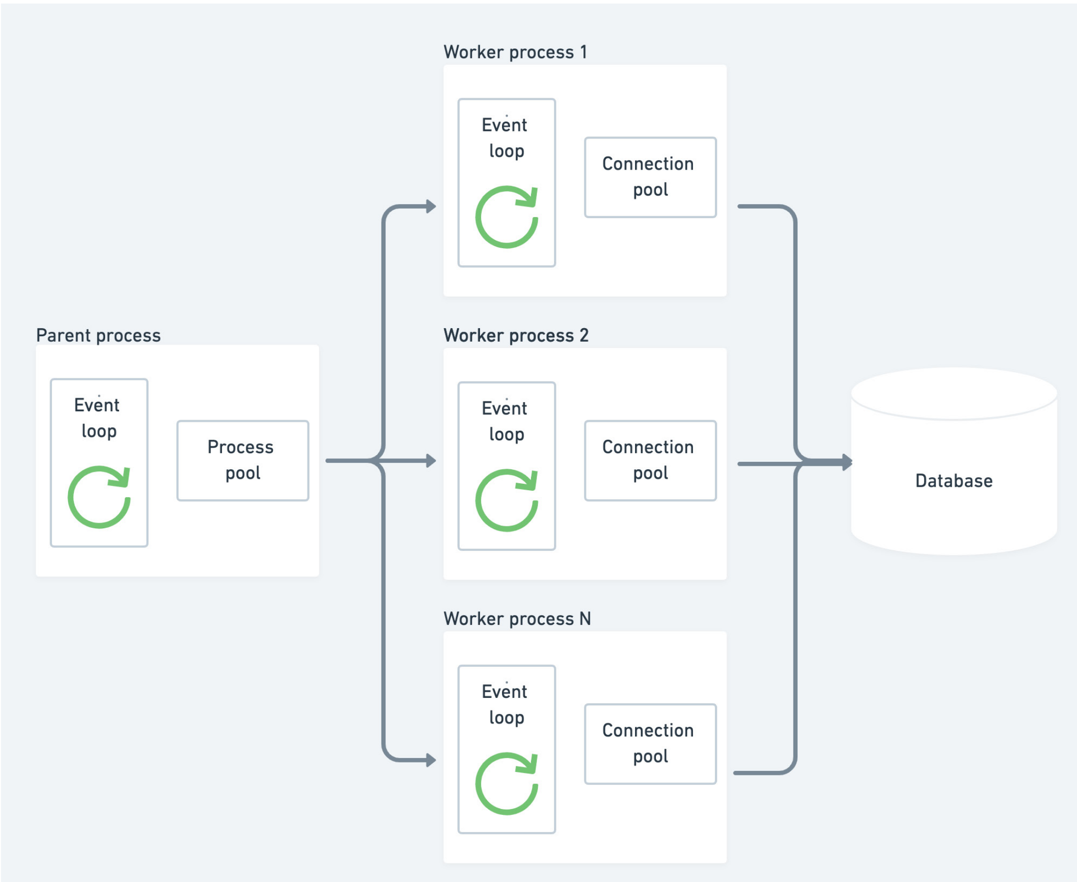


Figure 6.4 A parent process creates a process pool. The parent process then creates workers each with their own event loop.

While this won't make our I/O throughput increase, it will increase how many query results we can process at a time. This will increase the overall throughout of our application. Let's take our example from 5.8 in chapter five and use it to create this architecture.

Listing 6.12 one event loop per process

```
import asyncio
import asyncpg
from util import async_timed
from typing import List, Dict
from concurrent.futures.process import ProcessPoolExecutor


async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)


@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)


def run_in_new_loop(num_queries: int) -> List[Dict]:
    async def run_queries():
        async with asyncpg.create_pool(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       password='password',
                                       database='products',
                                       min_size=6,
                                       max_size=6) as pool:
            return await query_products_concurrently(pool, num_queries)

    results = [dict(result) for result in asyncio.run(run_queries())] #A
    return results


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    pool = ProcessPoolExecutor()
    tasks = []
    for _ in range(5):
        tasks.append(loop.run_in_executor(pool, run_in_new_loop)) #B
    all_results: List[List[Dict]] = await asyncio.gather(*tasks) #C
    total_queries = sum([len(result) for result in all_results])
    print(f'Retrieved {total_queries} products the product database.')


if __name__ == "__main__":
    asyncio.run(main())
```

#A Run queries in a new event loop and convert them to dictionaries
#B Create five processes each with their own event loop to run queries
#C Wait for all query results to complete

We create a new function `run_in_new_loop`. This function has an inner coroutine `run_queries` which creates a connection pool and runs the number of queries we specify concurrently. We then call `run_queries` with `asyncio.run` which creates a new event loop and runs the coroutine. One thing to note here is we convert our results into dictionaries, this is because asyncpg Record objects are not pickleable. Converting to a data structure which is serializable ensures that we can send our result back to our parent process.

In our main coroutine, we create a process pool and make five calls to `run_in_new_loop`. This will concurrently kick off 50,000 queries, 10,000 per each of five processes. When you run this, you should see five processes kicked off quickly followed by each of these processes finishing at roughly the same time. The runtime of the entire application should take slightly longer than the slowest single process. When running this on an 8-core this script was able to complete in roughly 13 seconds. Going back to our previous example from chapter five, we made 10,000 queries in about 6 seconds. Dividing this out means we were getting a throughput of roughly 1,666 queries per second. With the multiprocessing and multiple event loop approach we completed 50k queries in 13 seconds, or roughly 3,800 queries per second, more than doubling our throughput.

6.7 Summary

In this chapter we've learned how to utilize Python's multiprocessing module alongside asyncio to improve the performance of CPU bound workloads. We've also learned how to manage shared state and avoid concurrency bugs.

- We've learned how to run multiple Python functions in parallel with a process pool.
- We've learned how to create a process pool executor and run Python functions in parallel. A process pool executor lets us use asyncio API methods such as gather to run multiple processes concurrently and wait for the results.
- We've learned how to solve a map-reduce problem using process pools and asyncio. This workflow not only applies to map reduce, but we can use this in general with any CPU bound work that we can split into multiple smaller chunks.
- We've learned how to share state between multiple processes. This lets us keep track of data that is relevant for subprocesses we kick off, such as a status counter.
- We've learned how to avoid race conditions by using locks. Race conditions happen when multiple processes attempt to access data at roughly the same time and can lead to hard to reproduce bugs.
- We've learned how to use multiprocessing to extend the power of asyncio by creating an event loop per each process. This has the potential to improve performance of workloads that have a mixture of CPU bound and I/O bound work.

7 Handling blocking work with threads

This chapter covers

- **The multithreading library**
- **Creating thread pools to handle blocking I/O**
- **Using async and await to manage threads**
- **Handling blocking I/O libraries with thread pools**
- **Shared data and locking with threads**
- **Handling CPU bound work in threads**

When developing a new I/O bound application from scratch, asyncio may be a natural technology choice. Starting out, you'll be able to use non-blocking libraries that work with asyncio such as asyncpg and aiohttp as you begin development. However, greenfields development is a luxury that many developers don't have. A large portion of our work may be managing code written a while ago using blocking I/O libraries such as requests for HTTP requests, psycopg for Postgres databases or any number of blocking libraries. We may also be in a situation where an asyncio-friendly library does not exist yet. Is there a way to get the performance gains of concurrency while still using asyncio APIs in these cases?

Multithreading is the solution to this question. Since blocking I/O releases the global interpreter lock, this opens up the possibility to run I/O concurrently in separate threads. Much like the multiprocessing library, asyncio exposes a way for us to utilize pools of threads so that we can get the benefits of threading while still using the asyncio APIs such as gather and wait.

In this chapter, we'll learn how to use multithreading with asyncio to run blocking APIs such as requests in threads. In addition, we'll learn how to synchronize shared data like we did in the last chapter and examine more advanced locking topics like reentrant locks and deadlocks. We'll also see how to combine asyncio with synchronous code by building a responsive GUI to run an HTTP stress test. Finally, we'll take a look at the few exceptions where threading can be used for CPU bound work.

7.1 Introducing the threading module

Python provides the ability to create and manage threads via the threading module. This module exposes the `Thread` class, which when instantiated accepts a function to run in a separate thread. The Python interpreter runs single-threaded within a process, meaning that only one piece of Python bytecode can be running at a time even if we have code running in multiple threads. The global interpreter lock will only allow one thread to execute code at a time.

This seems like it limits us from using multithreading to any advantage, but there are a few cases where the global interpreter lock is released, the main case being when I/O operations occur. Python can release the GIL in this case because under the hood, Python is making low level operating system calls to perform I/O. These system calls are outside the Python interpreter, meaning that no Python bytecode needs to run while we're waiting for I/O to finish.

To get a better sense for how to create and run threads in the context of blocking I/O, we'll revisit our example of an echo server from chapter 3. Recall in order to handle multiple connections, we had to switch our sockets to non-blocking mode and use the select module to watch for events on the sockets. What if we were working with a legacy codebase and non-blocking sockets weren't an option? Can we still build an echo server that can handle more than one client at a time?

Since a socket's `recv` and `sendall` are I/O bound methods, and therefore release the GIL, we should be able to run them in separate threads concurrently. This means that we can create one thread per each connected client and read and write data in that thread. This model is a common paradigm in web servers such as Apache known as a *thread-per-connection* model. Let's give this idea a try by waiting for connections in our main thread and creating a thread to echo for each client that connects.

Listing 7.1 a multithreaded echo server

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM


def echo(client: socket):
    while True:
        data = client.recv(2048)
        print(f'Received {data}, sending!')
        client.sendall(data)


with socket(AF_INET, SOCK_STREAM) as server:
    server.bind(('0.0.0.0', 8000))
    server.listen()
    while True:
```

```
        connection, _ = server.accept() #A
        thread = Thread(target=echo, args=(connection,)) #B
        thread.start() #C
```

#A Block waiting for a client to connect
#B Once a client connects, create a thread to run our echo function
#C Start running the thread

In Listing 7.1 we enter an infinite loop listening for connections on our server socket. Once we have a client connected, we create a new thread to run our `echo` function. We supply the thread with a `target` which is the `echo` function we want to run and `args`, which is a tuple of arguments pass to `echo`. This means that we'll call `echo(connection)` in our thread. Then we start the thread and loop again, waiting for a second connection. Meanwhile, in the thread we created, we loop forever listening for data from our client, and when we have it, we echo it back.

You should be able to connect an arbitrary amount of telnet clients concurrently and have messages echo properly. Since each `recv` and `sendall` operates in a separate thread per each client, these operations never block each other, they only block the thread they are running in.

This solves the problem of multiple clients not being able to connect at the same time with blocking sockets, though it has some issues unique to threads. What happens if we try to kill this process with CTRL+C while we have clients connected? Does our application shut down the threads we created cleanly?

It turns out things don't shut down quite so cleanly. If you kill the application, you should see a `KeyboardInterrupt` exception thrown on `server.accept()`, but your application will hang as the background thread will keep the program alive. Furthermore, any connected clients will still be able to send and receive messages!

Unfortunately, user created threads in Python do not receive `KeyboardInterrupt` exceptions, only the main thread will receive them. This means that our threads will keep running, happily reading from our clients and preventing our application from exiting.

There are a couple approaches to handle this, we can use what are called *daemon* threads (pronounced demon), or we can come up with our own way of cancelling or ‘interrupting’ a running thread. Daemon threads are a special kind of thread for long-running background tasks. These threads won't prevent an application from shutting down. In fact, when only daemon threads are running, the application will automatically shut down. Since Python's main thread is not a daemon thread, this means that if we make all our connection threads daemonic our application will terminate on a `KeyboardInterrupt`. Adapting our code above to use daemonic threads is easy, all we need to do is set `thread.daemon = True` before we run `thread.start()`. If we make that change, our application will terminate properly on CTRL+C.

The problem with this approach is we have no way to run any cleanup or shutdown logic when our threads stop, daemon threads just terminate abruptly. Let's say that on shutdown we want to write out to each client that the server is shutting down. Is there a way we can have some type of exception interrupt our thread and cleanly shut down the socket? If we call a socket's `shutdown` method, any existing calls to `recv` will return zero and `sendall` will throw an exception. If we call `shutdown` from the main thread, this will have the effect of 'interrupting' our client threads that are blocking on a `recv` or `sendall` call. We can then handle the exception in the client thread and preform any cleanup logic we'd like.

To do this, we'll create threads slightly differently than before by subclassing the `Thread` class itself. This will let us define our own thread with a `cancel` method, inside of which we can shut down the client socket. Then, our calls to `recv` and `sendall` will be 'interrupted' allowing us to exit our while loop and close out the thread.

The `Thread` class has a `run` method that we can override. When we subclass `Thread`, we implement this method with the code that we want the thread to run when we start it. In our case, this is the `recv` and `sendall` echo loop.

Listing 7.2 Subclassing the thread class for a clean shutdown

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM, SHUT_RDWR


class ClientEchoThread(Thread):

    def __init__(self, client):
        super().__init__()
        self.client = client

    def run(self):
        try:
            while True:
                data = self.client.recv(2048)
                if data == bytes(0): #A
                    raise BrokenPipeError('Connection closed!')
                print(f'Received {data}, sending!')
                self.client.sendall(data)
        except OSError as e: #B
            print(f'Thread interrupted by {e} exception, shutting down!')

    def close(self):
        if self.is_alive(): #C
            self.client.sendall(bytes('Shutting down!', encoding='utf-8'))
            self.client.shutdown(SHUT_RDWR) #D


with socket(AF_INET, SOCK_STREAM) as server:
    server.bind(('0.0.0.0', 8000))
    server.listen()
    connection_threads = []
    try:
        while True:
            connection, addr = server.accept()
            thread = ClientEchoThread(connection)
            connection_threads.append(thread)
            thread.start()
    except KeyboardInterrupt:
        print('Shutting down!')
        [thread.close() for thread in connection_threads] #E
```

#A If the data is zero raise an exception, this happens when the connection was closed by the client or the connection was shut down.
#B When we have an exception, exit the run method. This terminates the thread.
#C Shutdown the connection if the thread is alive, the thread may not be alive if the client closed the connection.
#D Shutdown the client connection for reads and writes.
#E Call the close method on our threads to shutdown each client connection on keyboard interrupt.

We first create a new class `ClientEchoThread` that inherits from `Thread`. This class overrides the `run` method with the code from our original `echo` function with a few changes. First, we wrap everything in a try catch block and intercept `OSError` exceptions. This type of exception is thrown from methods such as `sendall` when we close the client socket. We also check to see if the data from `recv` is 0. This happens in two cases, if the client closes the connection (someone quits telnet for example) or when we shut down the client connection ourselves. In this case we throw a `BrokenPipeError` ourselves (a subclass of `OSError`), execute the print statement in the except block and exit the `run` method, which shuts the thread down.

We also define a `close` method on our `ClientEchoThread` class. This method first checks to see if the thread is alive before shutting down the client connection. What does it mean for a thread to be 'alive' and why do we need to do this? A thread is alive if its `run` method is executing, in this case this is true as long as our run method does not throw any exceptions. We need this check this because the client may have closed the connection themselves, resulting in a `BrokenPipeError` exception in the `run` method before we call `close`. This means that calling `sendall` would result in an exception as the connection is no longer valid.

Finally, in our main loop that listens for new incoming connections, we intercept `KeyboardInterrupt` exceptions. Once we have one, we call the `close` method on each thread we've created. This will send a message to the client assuming the connection is still active and shutdown the connection.

Overall, canceling running threads in Python, and in general, is a tricky problem and depends on the particular shutdown case you're trying to handle. You'll need to take special care that your threads do not block your application from exiting and figure out where to put in appropriate interrupt points to exit your threads.

We've now seen a couple ways to manage threads manually ourselves, creating a thread object with a target function, and subclassing `Thread` and overriding the `run` method. Now that we understand threading basics, let's see how to use them with asyncio to work with popular blocking libraries.

7.2 Using threads with asyncio

We now know how to create and manage multiple threads to handle blocking work. The drawback of this is that we need to individually create and keep track of threads ourselves. We'd like to be able to use all the asyncio-based APIs we've learned to wait for results from threads without having to manage them ourselves. Similar to process pools from the last chapter, we can use thread pools to manage threads in this manner. In this section, we'll introduce a popular blocking HTTP client library, and see how to use threads with asyncio to run web requests concurrently.

7.2.1 Introducing the requests library

The requests library is a popular HTTP client library for Python, self-described as "HTTP for humans." Using it you can make HTTP requests to web servers much like we did with aiohttp. We'll use the latest version as of this writing which is 2.24.0. You can install this library by running the following pip command:

```
pip install -Iv requests==2.24.0
```

You can view the latest documentation for the library at https://requests.readthedocs.io/en/master/. Once we've installed the library, we're ready to make some basic HTTP requests. Let's start out by making a couple of requests to example.com to retrieve the status code as we did earlier with aiohttp.

Listing 7.3 basic usage of requests

```
import requests


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


url = 'https://www.example.com'
print(get_status_code(url))
print(get_status_code(url))
```

Listing 7.3 executes two HTTP GET requests in serial. Running this, you should see 200 output twice. We didn't create a HTTP session here as we did with aiohttp, but the library does support this if we need to keep cookies persistent across different requests.

The requests library is blocking, meaning that each call to `requests.get` will stop any thread from executing other Python code until the request finishes. This has implications for how we can use this library in asyncio. If we try to use this library in a coroutine or a task by itself, it will block the entire event loop until the request finishes. If we had a HTTP request that took two seconds, our application wouldn't be able to do anything other than wait for those two seconds. In order to properly use this library with asyncio, we need to run these blocking operations inside of a thread.

7.2.2 Introducing thread pool executors

Much like process pool executors, the `concurrent.futures` library provides an implementation of the `Executor` abstract class to work with threads named `ThreadPoolExecutor`. Instead of maintaining a pool of worker processes like a process pool does, a thread pool executor will create and maintain a pool of threads that we can then submit work to.

While a process pool will by default create one worker process for each CPU core our machine has available, determining how many worker threads to create is a bit more complicated. Internally the formula for the default number of threads is `min(32, os.cpu_count() + 4)`. This causes the upper bound of worker threads to be 32 and the minimum bound to be 5. The upper bound is set to 32 to avoid creating a surprising amount of threads on machines with large amounts of CPU cores (remember, threads are expensive to create and maintain). The lower bound is set to 5 because on smaller 1-2 core machines, spinning up only a couple of threads isn't likely to improve performance much. It often makes sense to create a few more threads than your available CPUs for I/O bound work. For example, on an 8-core machine the above formula means we'll create 12 threads. While only 8 threads can run concurrently, we can have other threads paused waiting for I/O to finish, letting our operating system context switch to run them when I/O is done.

Let's adapt our example from above to run a thousand HTTP requests concurrently with a thread pool. We'll time the results to get an understanding of what the benefit is.

Listing 7.4 running requests with a thread pool

```
import time
import requests
from concurrent.futures import ThreadPoolExecutor


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


start = time.time()

with ThreadPoolExecutor() as pool:
    urls = ['https://www.example.com' for _ in range(1000)]
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)

end = time.time()

print(f'finished requests in {end - start:.4f} second(s)')
```

On an 8 core machine with a speedy internet connection, this code can execute in as little as 8-9 seconds with the default number of threads. It is pretty easy to write this synchronously to understand the impact that threading has by doing something like the following:

```
start = time.time()

urls = ['https://www.example.com' for _ in range(1000)]

for url in urls:
    print(get_status_code(url))

end = time.time()

print(f'finished requests in {end - start:.4f} second(s)')
```

Running this code can take upwards of 100 seconds! This makes our threaded code a bit more than 10x faster than our synchronous code, giving us a pretty big performance bump.

While this is clearly an improvement, you may remember from our chapter on aiohttp we were able to make 1000 requests concurrently in under a second. Why is this so much slower than our threading version? Remember that our maximum number of worker threads is limited to 32 or the number of CPUs plus four, meaning by default we can only run a maximum of 32 requests at the same time. We can try to get around this by passing in `max_workers=1000` when we create our thread pool like so:

```
with ThreadPoolExecutor(max_workers=1000) as pool:
    urls = ['https://www.example.com' for _ in range(1000)]
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)
```

This approach can yield some improvements as we now have one thread per each request we make. However, this still won't come very close to our coroutine based code. This is due to the resource overhead associated with threads. Threads are created at the operating system level and are more expensive to create than coroutines. In addition, threads have a context-switching cost at the OS level. Saving and restoring thread state when a context switching happens eats up some of the performance gains we get by using threads.

When you're determining the number of threads to utilize for a particular problem it is best to start small (the amount of CPU cores plus a few is a good starting point), test it and benchmark, gradually increasing the number threads. You'll usually find a 'sweet spot' after which the run time will plateau and may even degrade no matter how many more threads you add. This sweet spot is usually a fairly low number relative to the requests you want to make (to make it clear, creating 1k threads for 1k requests probably isn't the best use of resources).

7.2.3 Thread pool executors with asyncio

Using thread pool executors with the asyncio event loop isn't much different than using `ProcessPoolExecutors`. This is the beauty of having the abstract Executor base class, we can use the same code to run threads or processes by only having to change one line of code. Let's adapt our example of running 1000 http requests to use `asyncio.gather` instead of `pool.map`.

Listing 7.5 using a thread pool executor with asyncio

```
import functools
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor
from util import async_timed


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code
```

```
@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        urls = ['https://www.example.com' for _ in range(1000)]
        tasks = [loop.run_in_executor(pool, functools.partial(get_status_code, url)) for
       url in urls]
        results = await asyncio.gather(*tasks)
        print(results)

asyncio.run(main())
```

We create the thread pool as we did before, but instead of using map we create a list of tasks by calling our `get_status_code` function with `loop.run_in_executor`. Once we have a list of tasks, we can wait for them to finish with `asyncio.gather` or any of the other asyncio APIs we learned earlier.

Internally, `loop.run_in_executor` calls the thread pool executor's `submit` method. This will put each function we pass in onto a queue. Worker threads in the pool then pull from the queue, running each work item until it completes. This approach does not yield any performance benefits over using a pool without asyncio, but while we're waiting for `await asyncio.gather` to finish, other code can run.

7.2.4 Default executors

Reading the asyncio documentation, you may notice that the `run_in_executor` method's `executor` parameter can be `None`. In this case, `run_in_executor` will use the event loop's default executor. What is a default executor? Think of a default executor as a reusable singleton executor for your entire application. The default executor will always default to a `ThreadPoolExecutor` unless we set a custom one ourselves with the `loop.set_default_executor` method. This means that we can simplify the code from listing 7.5 to the following:

Listing 7.6 using the default executor

```
import functools
import requests
import asyncio
from util import async_timed


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    urls = ['https://www.example.com' for _ in range(1000)]
    tasks = [loop.run_in_executor(None, functools.partial(get_status_code, url)) for url in
```

```
        urls]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

In listing 7.6, we ditch creating our own `ThreadPoolExecutor` and using it in a context manager as we did before and instead pass in `None` as the executor. The first time we call `run_in_executor`, asyncio creates and caches a default thread pool executor for us. Each subsequent call to `run_in_executor` reuses the previously created default executor, meaning the executor is then global to the event loop. Shutdown of this pool is also different from what we saw before. Previously, the thread pool executor we created ourselves was shut down when we exited a context manager's with block. When using the default executor, it won't be shut down until the event loop closes, which usually happens when our application finishes. Using the default thread pool executor when we want to use threads simplifies things, but can we make this even easier?

In Python 3.9, the `asyncio.to_thread` coroutine was introduced to further simplify putting work on the default thread pool executor. It takes in a function to run in a thread and a set of arguments to pass to that function. Previously, we had to use `functools.partial` to pass in arguments, so this makes our code a little cleaner. It then runs the function with its arguments in the default thread pool executor and the currently running event loop. This lets us simplify our threading code even more.

Listing 7.7 using the to_thread coroutine

```
import requests
import asyncio
from util import async_timed


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


@async_timed()
async def main():
    urls = ['https://www.example.com' for _ in range(1000)]
    tasks = [asyncio.to_thread(get_status_code, url) for url in urls]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

Using the `to_thread` coroutine, we've now got rid of our use to `functools.partial` and our call to `asyncio.get_running_loop`, cutting down on our total lines of code.

So far, we've only seen how to run blocking code inside of threads. The power of combining threads with asyncio is that we can run other code while we're waiting for our threads to finish. To see how to run other code while threads are running, we'll revisit our example from chapter 6 of periodically outputting the status of a long running task.

7.3 Locks, shared data and deadlocks

Much like multiprocessing code, multithreaded code is also susceptible to race conditions when we have shared data as we do not control order of execution. Any time you have two threads or processes that could modify a shared piece of non-thread safe data, you'll need to utilize a lock to properly synchronize access. Conceptually, this is not any different than the approach we took with multiprocessing, however, the memory model of threads changes the approach slightly.

Recall that with multiprocessing, the processes we create do not share memory by default. This meant we needed to create special shared memory objects and properly initialize them so that each process could read from and write to that object. Since threads *do* have access to the same memory of their parent process, we no longer need to do this, threads can access shared variables directly.

This simplifies things a bit, but since we won't be working with shared `Value` objects that have locks built in, we'll need to create them ourselves. To do this, we'll need to use the threading module's `Lock` implementation, which is different from the one we used with multiprocessing. This is as easy as importing `Lock` from the threading module and calling its `acquire` and `release` methods around critical sections of code, or using it in a context manager.

To see how to use locks with threading, let's revisit our task from last chapter of keeping track and displaying the progress of a long task. We'll take our previous example of making thousands of web requests and use a shared counter to keep track of how many requests we've completed so far.

Listing 7.8 printing status of requests

```
import functools
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor
from threading import Lock
from util import async_timed

counter_lock = Lock()
counter: int = 0


def get_status_code(url: str) -> int:
    global counter
    response = requests.get(url)
    with counter_lock:
        counter = counter + 1
    return response.status_code


async def reporter(request_count: int):
    while counter < request_count:
        print(f'Finished {counter}/{request_count} requests')
        await asyncio.sleep(.5)
```

```
@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        request_count = 200
        urls = ['https://www.example.com' for _ in range(request_count)]
        reporter_task = asyncio.create_task(reporter(request_count))
        tasks = [loop.run_in_executor(pool, functools.partial(get_status_code, url)) for
       url in urls]
        results = await asyncio.gather(*tasks)
        await reporter_task
        print(results)


asyncio.run(main())
```

This should look familiar as it is similar to the code we wrote to output progress of our map operation in the last chapter. We create a global `counter` variable as well as a `counter_lock` to synchronize access to it in critical sections. In our `get_status_code` function we acquire the lock when we increment the counter. Then in our main coroutine we kick off a reporter background task that outputs how many requests we've finished every 500 milliseconds. Running this, you should see output similar to the following:

```
Finished 0/200 requests
Finished 48/200 requests
Finished 97/200 requests
Finished 163/200 requests
```

We now know the basics around locks with both multithreading and multiprocessing, but there is still quite a bit to learn about locking. Next, we'll take a look at the concept of *reentrancy*.

7.3.1 Reentrant locks

Simple locks work well for coordinating access to a shared variable across multiple threads, but what happens when a thread tries to acquire a lock it has already acquired? Is this even a safe operation? Since the same thread is acquiring the lock, this should be OK as this is by definition single-threaded and therefore thread-safe.

While this access should be ok, it does cause problems with the locks we have been using so far. To illustrate this, let's imagine we have a recursive sum function that takes a list of integers and produces the sum of the list. The list we want to sum can be modified from multiple threads, so we need to use a lock to ensure the list we're summing does not get modified during our sum operation. Let's try implementing this with a normal lock to see what happens. We'll also add some console output to see how our function is executing.

Listing 7.9 recursion with locks

```
from threading import Lock, Thread
from typing import List

list_lock = Lock()
```

```
list = [1, 2, 3, 4]


def sum_list(int_list: List[int]) -> int:
    print('Waiting to acquire lock...')
    with list_lock:
        print('Acquired lock.')
        if len(int_list) == 1:
            print('Finished summing.')
            return int_list[0]
        else:
            head, *tail = int_list
            print('Summing rest of list.')
            return head + sum_list(tail)


thread = Thread(target=sum_list, args=(list,))
thread.start()
thread.join()
```

If you run this code, you'll see the following few messages and then the application will hang forever.

```
Waiting to acquire lock...
Acquired lock.
Summing rest of list.
Waiting to acquire lock...
```

Why is this happening? If we walk through this, we acquire `list_lock` the first time perfectly fine. We then unpack the list and recursively call `sum_list` on the remainder of the list. This then causes us to attempt to acquire `list_lock` a second time. This is where our code hangs, because we already acquired the lock, we block forever trying to acquire the lock a second time. This also means we never exit the first `with` block and can't release the lock - we're waiting for a lock that will never be released!

Since this recursion is coming from the same thread that originated it, acquiring the lock more than once shouldn't be a problem as this won't cause race conditions. To support these use cases the threading library provides *reentrant* locks. A reentrant lock is a special kind of lock that can be acquired by the same thread more than once, allowing that thread to "reenter" critical sections. The threading module provides reentrant locks in the `RLock` class. We can take our above code and fix the problem by modifying only two lines of code – the import statement and the creation of the `list_lock`.

```
from threading import Rlock

list_lock = RLock()
```

If we modify these lines our code will work properly, and a single thread will be able to acquire the lock multiple times. Internally, reentrant locks work by keeping a recursion count. Each time we acquire the lock from the thread that first acquired the lock, the count increases and each time we release the lock it decreases. When the count is zero, the lock is finally released for other threads to acquire it.

As many of us probably do not use recursion with locks, let's examine a more real-world application truly understand the concept. Imagine we're trying to build a thread-safe integer list class with a method to 'find and replace' all elements of a certain value with a different value. This class will contain a normal Python list and a lock we use to prevent race conditions. We'll pretend our existing class already has a method `indices_of(to_find: int)` which takes in an integer and returns all the indices in the list that match `to_find`. Since we want to follow the DRY (don't repeat yourself) rule, we'll reuse this method when we define our find and replace method. This means our class and method will look something like the following:

Listing 7.10 a thread safe list class

```
from threading import Lock
from typing import List


class IntListThreadsafe():
    lock = Lock()

    def __init__(self, wrapped_list: List[int]):
        self.inner_list = wrapped_list

    def indices_of(self, to_find: int) -> List[int]:
        with self.lock:
            enumerator = enumerate(self.inner_list)
            return [index for index, value in enumerator if value == to_find]


    def find_and_replace(self, to_replace: int, replace_with: int) -> None:
        with self.lock:
            indices = self.indices_of(to_replace)
            for index in indices:
                self.inner_list[index] = replace_with


threadsafe_list = IntListThreadsafe([1, 2, 1, 2, 1])
threadsafe_list.find_and_replace(1, 2)
```

If someone from another thread modifies the list during our `indices_of` call, we could wind up with an incorrect return value, thus we need to acquire the lock before we search for matching indices. Our `find_and_replace` method needs to acquire the lock for the same reason. However, with a normal lock we wind up hanging forever when we call `find_and_replace`. The find and replace method first acquires the lock and then calls another method which tries to acquire the same lock. Switching to an `RLock` in this case will fix this problem because one call to `find_and_replace` will always acquire any locks from the same thread. This illustrates a generic formula for when you need to use reentrant locks. If you are developing a thread safe class with a method A which acquires a lock, and a method B that also needs to acquire a lock *and* call method A, you likely need to use a reentrant lock.

7.3.2 Deadlocks

The Meriam-Webster dictionary defines a deadlock as “a state of inaction or neutralization resulting from the opposition of equally powerful uncompromising persons or factions”. Many of us may be familiar with the concept of deadlock from political negations we hear about on the news, one party makes a demand of the other side, and the other side makes a counter demand. Both sides disagree on the next course and the negotiation reaches a standstill. The concept in computer science is similar, we reach a state where there is contention over shared resource with no resolution, and our application hangs forever.

The issue we saw in the previous section where non-reentrant locks can cause our program to hang forever is one example of deadlock. In that case, we reach a state where we’re stuck in a standstill negotiation with ourselves, demanding to acquire a lock we’ll never release. This situation can also arise when we have two threads using more than one lock. If thread A asks for a lock that thread B has acquired, and thread B is asking for a lock that A has acquired, we reach a standstill and a deadlock. In that instance, using reentrant locks won’t help as we have multiple threads each stuck waiting on a resource the other thread holds.

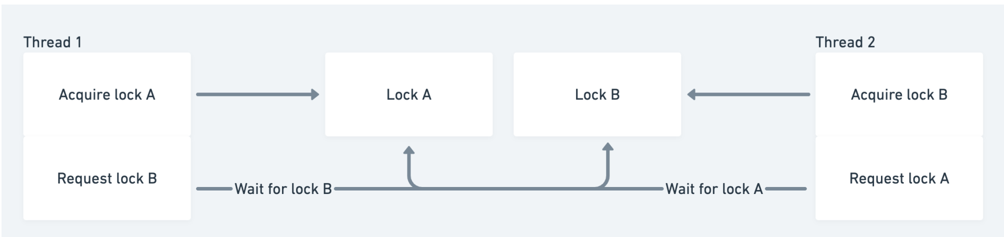


Figure 7.1 Thread one and two acquire locks A and B at roughly the same time. Then thread one waits for lock B, which thread two holds, meanwhile thread two is waiting for A, which thread one holds. This circular dependency causes a deadlock and will hang the application.

Let’s take a look at how to create this type of deadlock in code. We’ll create two locks, lock A and B, and two methods which need to acquire both locks. One method will acquire A first and then B and another will acquire B first and then A.

Listing 7.11 a deadlock in code

```
from threading import Lock, Thread
import time

lock_a = Lock()
lock_b = Lock()


def a():
    with lock_a: #A
        print('Acquired lock a from method a!')
        time.sleep(1) #B
        with lock_b: #C
```

```
            print('Acquired both locks from method a!')


def b():
    with lock_b: #D
        print('Acquired lock b from method b!')
        with lock_a: #E
            print('Acquired both locks from method b!')


thread_1 = Thread(target=a)
thread_2 = Thread(target=b)
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
```

#A Acquire lock A.
#B Sleep for one second, this ensures we create the right conditions for deadlock.
#C Acquire lock B.
#D Acquire lock B.
#E Acquire lock A.

When we run this code, we'll see the following output, and our application will hang forever:

```
Acquired lock a from method a!
Acquired lock b from method b!
```

We first call method A and acquire lock A, then we introduce an artificial delay to give method B a chance to acquire lock B. This leaves us in a state where method A holds lock A and method B holds lock B. Next, method A attempts to acquire lock B, but method B is holding that lock. At the same time, method B tries to acquire lock A, but method A is holding it, stuck waiting for B to release its lock. Both methods are stuck waiting on one another to release a resource and we reach a standstill.

How do we handle this situation? One solution is the so-called "ostrich algorithm", named after an ostrich sticking its head in the sand when there is danger (though ostriches don't *actually* do this). With this strategy, we ignore the problem and devise a strategy to restart our application when we encounter the issue. The driving idea behind this is if the issue happens rarely enough, investing in a fix isn't worth it. If you remove the sleep from the above code, you'll only rarely see deadlock occur as it relies on a very specific sequence of operations. This isn't really a fix and isn't ideal, but is a strategy used with rarely occurring deadlocks.

However, in our situation there is an easy fix, we simply change the locks in both methods to always be acquired in the same order. For instance, both methods A and B can acquire lock A first then lock B. This resolves the issue as we'll never acquire locks in the in an order where a deadlock could occur. The other option for us would be to refactor the locks so we were only using one instead of two. It is impossible to have a deadlock with one lock (excluding the reentrant deadlock we saw earlier). Overall, when dealing with multiple locks that you need to acquire, ask yourself, am I acquiring these in a consistent order? Is there a way I can refactor this to only use one lock?

We've now seen how to use threads effectively with asyncio and have dove into more complex locking scenarios. Next, let's see how to use threads to integrate asyncio into existing synchronous applications that may not work nicely with asyncio.

7.4 Event loops in separate threads

We have mainly focused on building applications that are completely implemented from the bottom up with coroutines and asyncio. When we've had any work that does not fit within a single-threaded concurrency model, we have run it inside of threads or processes. Not all applications will fit into this paradigm. What if we're working in an existing synchronous application and we want to incorporate asyncio?

One such situation where we can run into this scenario is building desktop user interfaces. The frameworks to build GUIs usually run their own event loop and this event loop blocks the main thread. This means that any long-running operations can cause the user interface to freeze. In addition, this UI event loop will block us from creating an asyncio event loop. In this section, we'll learn how to use multithreading to run multiple event loops at the same time by building a responsive HTTP stress testing user interface in Tkinter.

7.4.1 Introducing Tkinter

Tkinter is a platform independent desktop graphical user interface (GUI) toolkit provided in the default Python installation. Short for "Tk interface", it is an interface to the lower-level Tk GUI toolkit which is written in the tcl language. With the creation of the tkinter Python library Tk has grown into a popular way for Python developers to build desktop user interfaces.

Tkinter has a set of 'widgets' such as labels, text boxes and buttons that we can place in a desktop window. When we interact with a widget, such as entering text or pressing a button, we can trigger a function to execute code. The code that runs in response to a user action could be as simple as updating another widget or triggering another operation.

Tkinter, and many other GUI libraries, draw their widgets and handle widget interactions through their own event loop. This event loop is constantly redrawing the application, processing events and checking to see if any code should run in response to a widget event. To get familiar with Tkinter and its event loop, let's create a basic hello world application. We'll create an application with a 'say hello` button that will output 'Hello there!' to the console when we click on it.

Listing 7.12 hello world with tkinter

```
import tkinter
from tkinter import ttk

window = tkinter.Tk()
window.title('Hello world app')
window.geometry('200x100')


def say_hello():
    print('Hello there!')
```

```
hello_button = ttk.Button(window, text='Say hello', command=say_hello)
hello_button.pack()

window.mainloop()
```

This code first creates a tkinter window and sets the application title and window size. We then place a button on the window and set its command to the `say_hello` function. When a user presses this button the `say_hello` function executes, printing out our message. We then call `window.mainloop()` – this starts the Tk event loop, running our application.

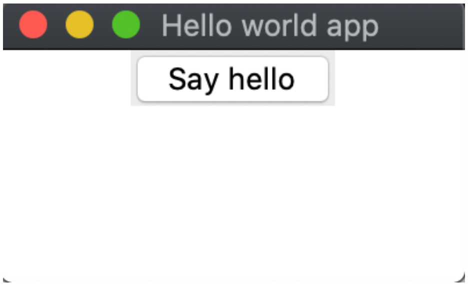


Figure 7.2 The hello world application from listing 7.12

One thing to note here is that our application will block on `window.mainloop()`. Internally, this method runs the Tk event loop. This is an infinite loop that is checking for window events and constantly redrawing the window until we close it. The Tk event loop has interesting parallels to the asyncio event loop. For example, what happens if we try to run blocking work in our button's command? If we add a ten second delay to the `say_hello` function with `time.sleep(10)`, we'll start to see a problem, our application will freeze for 10 seconds!

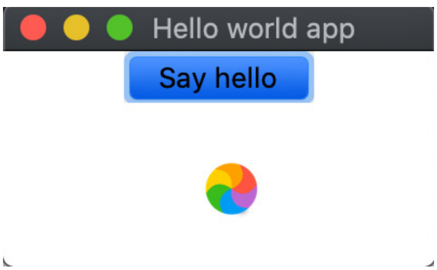


Figure 7.3 The dreaded 'beach ball of doom' occurs as we block the event loop on a Mac.

Much like asyncio, tkinter runs *everything* in its event loop. This means if we have a long running operation, such as making a web request or loading a large file, we'll block the tk event loop until that operation finishes. The effect on the user is the UI hangs and becomes unresponsive. The user can't click on any buttons, we can't update any widgets with status or progress, and the operating system will likely display a spinner to indicate the application is hanging. This is clearly an undesirable, unresponsive user interface.

This is an instance where asynchronous programming can in theory help us out. If we can make asynchronous requests that don't block the tk event loop, we can avoid this problem. This is trickier that it may seem as tkinter is not asyncio aware, and you can't pass in a coroutine to run on a button click. We could try running two event loops at the same time in the same thread, but this won't work. Both tkinter and asyncio are single threaded – this idea is the same as trying to run two infinite loops in the same thread at the same time, which can't be done. If we start the asyncio event loop before the tkinter event loop, the asyncio event loop will block the tkinter loop from running and vice versa. Is there a way for us to run an asyncio application alongside a single-threaded application?

We can in fact combine these two event loops to create a functioning application by running the asyncio event loop in a separate thread. Let's take a look at how to do this with an application that will responsively update the user on the status of a long-running task with a progress bar.

7.4.2 Building a responsive UI with asyncio and threads

First, let's introduce our application and sketch out a basic UI. We'll build a URL stress test application. This application will take a URL and a number of requests to send as input. When we hit a submit button, we'll use aiohttp to send out web requests as fast as we can, delivering a predefined load to the web server we choose. Since this may take a long time, we'll add a progress bar to visualize how far along we are in the test. We'll update the progress bar after every 1% of total requests are finished to show progress. Furthermore, we'll let the user cancel the request if they'd like. Our UI will have a few widgets, a text input for the URL to test, a text input for the number of requests we wish to issue, a 'start' button and a progress bar. We'll design a UI that looks like the following:

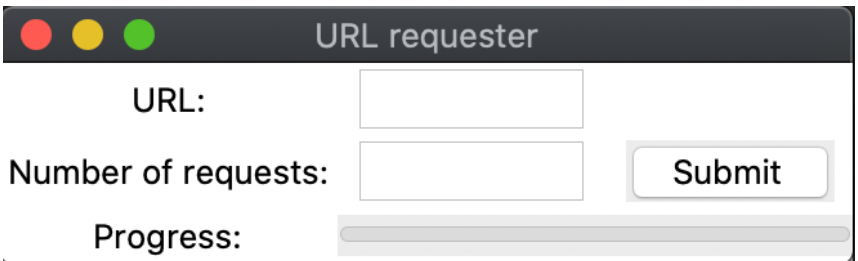


Figure 7.4 The URL requester GUI

Now that we have our UI sketched out, we need to think through how to have two event loops running alongside one another. The basic idea is that we'll have the Tkinter event loop running in the main thread and we'll run the asyncio event loop in a separate thread. Then, when the user clicks submit, we'll submit a coroutine to the asyncio event loop to run the stress test. As the stress test is running, we'll issue commands from the asyncio event loop back to the tkinter event loop to update our progress. This gives us an architecture that looks like follows:

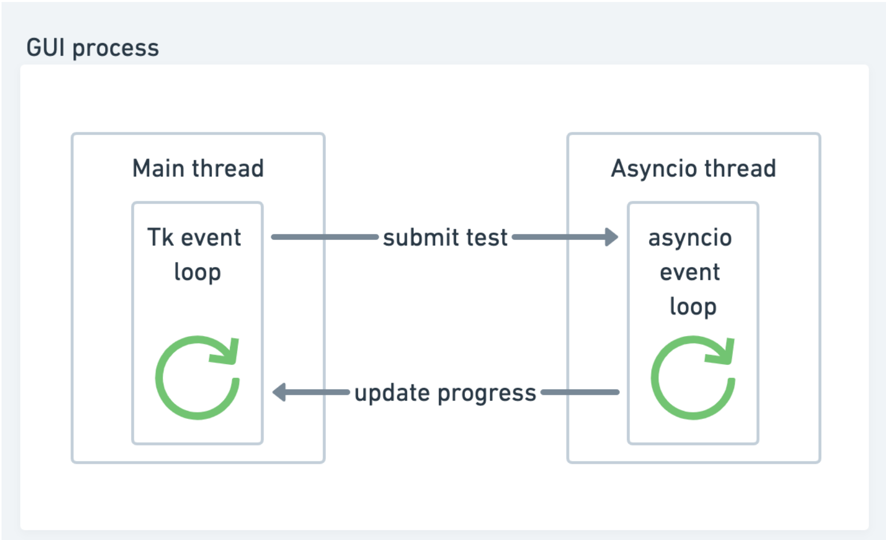


Figure 7.5 The tk event loop submits a task to the asyncio event loop which runs in a separate thread.

This new architecture includes communication across threads. We need to be careful about race conditions in this situation, especially since the asyncio event loop is *not* thread safe! Tkinter is designed with thread safety in mind so there are less concerns with calling it from a separate thread (in Python 3+ at least, we'll talk more about this soon).

We may be tempted to submit coroutines from tkinter using `asyncio.run`, but this function blocks until the coroutine we pass in finishes and will cause the tkinter application to hang. We'll need a function which submits a coroutine to the event loop without any blocking. There are a few new asyncio functions to learn that are both non-blocking and have thread safety built in to submit this kind of work properly. The first is a method on the asyncio event loop named `call_soon_threadsafe`. This function takes in a Python function (not a coroutine) and schedules it to execute it in a threadsafe manner at the next iteration of the asyncio event loop. The second function is `asyncio.run_coroutine_threadsafe`. This function takes in a coroutine and submits it to run in a threadsafe manner, immediately returning a future that we can use to access a result of the coroutine. Importantly, and confusingly, this future is *not* an asyncio future, but rather from the concurrent.futures module. The logic behind this is that asyncio futures are not thread safe, but concurrent.futures futures are. This future class does however have the same functionality as the future from the asyncio module.

Let's start defining and implementing a few classes to build our stress test application based on what we described above. The first thing we'll build is a stress test class. This class will be responsible for starting and stopping one stress test and keeping track of how many requests have completed. Its constructor will take in a url, an asyncio event loop, the number of desired requests to make, and a progress updater callback. We'll call this callback when we want to trigger a progress bar update. When we get to implementing the UI, this callback will trigger an update to the progress bar. Internally, we'll calculate a refresh rate, this is the rate at which we'll execute the callback. We'll default this rate to every 1 percent of the total requests we plan to send.

Listing 7.13 the stress test class

```
import asyncio
from concurrent.futures import Future
from asyncio import AbstractEventLoop
from typing import Callable
from aiohttp import ClientSession


class StressTest:
    _completed_requests: int = 0
    _load_test_future: Future = None

    def __init__(self,
                 loop: AbstractEventLoop,
                 url: str,
                 total_requests: int,
                 callback: Callable[[int, int], None]):
        self._loop = loop
        self._url = url
        self._total_requests = total_requests
        self._callback = callback
        self._refresh_rate = int(total_requests / 100)

    def start(self):
        future = asyncio.run_coroutine_threadsafe(self._make_requests(), self._loop)
        self._load_test_future = future

    def cancel(self):
        if self._load_test_future is not None:
            self._loop.call_soon_threadsafe(self._load_test_future.cancel) #B

    async def _get_url(self, session: ClientSession, url: str):
        try:
            await session.get(url)
        except Exception as e:
            print(e)
        self._completed_requests = self._completed_requests + 1 #C
        if self._completed_requests % self._refresh_rate == 0:
            self._callback(self._completed_requests, self._total_requests)

    async def _make_requests(self):
        async with ClientSession() as session:
            reqs = [self._get_url(session, self._url) for _ in range(self._total_requests)]
            await asyncio.gather(*reqs)
```

#A Start making the requests and store the future so we can later cancel if needed
#B If we want to cancel, call the cancel function on the load test future

#C Once we've completed one percent of requests, call the callback with the number of completed requests and the total requests

In our `start` method, we call `run_coroutine_threadsafe` with `_make_requests` which will start making requests on the asyncio event loop. We also keep track of the future this returns in the `_load_test_future` instance variable. Keeping track of this future lets us cancel the load test in our `cancel` method. In our `_make_requests` method we create a list coroutines to make all our web requests, passing them into `asyncio.gather` to run them. Our `_get_url` coroutine makes the request, increments the `_completed_requests` counter, and calls the callback with the total number of completed requests if necessary. We can use this class by simply instantiating it and calling the start method, optionally canceling by calling the cancel method.

One interesting thing to note is that we didn't use any locking around the `_completed requests` counter despite updates happening to it from multiple coroutines. Remember asyncio is single threaded, and the asyncio event loop only runs piece of Python code at any given time. This has the effect of making incrementing the counter atomic when used with asyncio, despite it being non-atomic when happening between multiple threads. Asyncio actually saves us from many kinds of race conditions that we see with multithreading, but not all. We'll examine this more in a later chapter.

Next, let's implement our Tkinter GUI to use this load tester class. For code cleanliness, we'll subclass the TK class directly and initialize our widgets in the constructor. When a user clicks the start button, we'll create a new `LoadTest` instance and start it. The question now becomes what do we pass in as a callback to our `LoadTest` instance? Thread safety becomes an issue here as our callback will be called in the worker thread. If our callback modifies shared data that our main thread can also modify, this could cause race conditions. In our case, since Tkinter has thread safety built in and all we're doing is updating the progress bar, we should be ok, but what if we needed to do something with shared data? Locking is one approach, but if we could run our callback in the main thread, we'd avoid any race conditions. Tkinter has a method that lets us queue up a function to later run in the main thread called `after_idle`. This will fire the function we provide in the main thread once the Tk event loop has processed any pending events it has, meaning we should be able to safely modify shared data. We'll use this approach in our GUI to demonstrate the technique, the callback we pass to the load test will be a function that calls `after_idle` with a function that triggers a progress bar update.

IS TKINTER REALLY THREAD SAFE?

If you search for Tkinter and thread safety, you'll find a lot of conflicting information. This is in part because for a number of years, Tk and Tkinter did not have proper thread support. Even when threaded mode was added, it had several bugs that have since been fixed. Tk supports both non-threaded and threaded modes. In non-threaded mode, we don't have any thread safety, and using Tkinter from anything other than the main thread is inviting a crash. In older versions of Python, Tk thread safety was not turned on, however, in versions of Python 3 and later, thread safety is turned on by default and we have thread safe guarantees. In threaded mode, if an update is issued from a worker thread, Tkinter acquires a mutex and writes the update event to a queue for the main thread to later process. The relevant code where this happens is in CPython in the `Tkapp_Call` function in `Modules/_tkinter.c`.

Listing 7.14 the tkinter GUI

```
from tkinter import Tk
from tkinter import Label
from tkinter import Entry
from tkinter import ttk
from chapter_07.stress_test import StressTest


class LoadTester(Tk):
    _load_test: StressTest = None

    def __init__(self, loop, *args, **kwargs): #A
        Tk.__init__(self, *args, **kwargs)
        self._loop = loop

        self._url_label = Label(self, text="URL:")
        self._url_label.grid(column=0, row=0)

        self._url_field = Entry(self, width=10)
        self._url_field.grid(column=1, row=0)

        self._request_label = Label(self, text="Number of requests:")
        self._request_label.grid(column=0, row=1)

        self._request_field = Entry(self, width=10)
        self._request_field.grid(column=1, row=1)

        self._submit = ttk.Button(self, text="Submit", command=self._start) #B
        self._submit.grid(column=2, row=1)

        self._pb_label = Label(self, text="Progress:")
        self._pb_label.grid(column=0, row=3)

        self._pb = ttk.Progressbar(self, orient="horizontal", length=200,
       mode="determinate")
        self._pb.grid(column=1, row=3, columnspan=2)

    def _update_bar(self, pct: int): #C
        if pct == 100:
            self._load_test = None
            self._submit['text'] = 'Submit'

        self._pb['value'] = pct

    def _trigger_update(self, completed_requests: int, total_requests: int): #D
        self.after_idle(self._update_bar, int((completed_requests / total_requests) * 100))

    def _start(self): #E
        if self._load_test is None:
            self._submit['text'] = 'Cancel'
            test = StressTest(self._loop,
                              self._url_field.get(),
                              int(self._request_field.get()),
                              self._trigger_update)
```

```
            test.start()
            self._load_test = test
        else:
            self._load_test.cancel()
            self._load_test = None
            self._submit['text'] = 'Submit'
```

#A In our constructor, we set up the text inputs, labels, submit button and progress bar
#B When clicked, our submit button will call the _start method
#C The update bar method will set the progress bar to a percentage complete value from 0 to 100. This method should only be called in the main thread.
#D This method is the callback we pass to the load test; it will schedule a task to update the progress bar.
#E This method starts the load test. If a test is not active, we create a new one and start it. Otherwise we cancel the existing one.

In our application's constructor we create all the widgets we need for the user interface. Most notably, we create `Entry` widgets for the URL to test and the number of requests to run, a submit button and a horizontal progress bar. We also use the `grid` method to arrange these widgets in the window appropriately.

When we create the submit button widget, we specify the command as the `_start` method. This method will create a `StressTest` object and starts running it unless we already have a load test running, in which case we will cancel it. When we create a `StressTest` object, we pass in the `_trigger_update` method as a callback. The `StressTest` object will call this method whenever it has a progress update to issue. When this method runs, we use the `after_idle` method to trigger the `_update_bar` method, calculating the appropriate percentage based on the number of requests completed. Using `after_idle` instead of directly calling `_update_bar` will ensure that our `_update_bar` method runs in the tkinter event loop thread. If we don't do this, the progress bar update would happen in the asyncio event loop as the callback is run within that thread.

Now that we've implement the UI application, we can glue these pieces all together to create a fully working application. We'll create a new thread to run the event loop in the background, and then start our newly created `LoadTester` application.

Listing 7.15 the load tester app

```
import asyncio
from asyncio import AbstractEventLoop
from threading import Thread
from chapter_07.load_test_app import LoadTester


class ThreadedEventLoop(Thread): #A
    def __init__(self, loop: AbstractEventLoop):
        super().__init__()
        self._loop = loop
        self.daemon = True

    def run(self):
        self._loop.run_forever()
```

```
loop = asyncio.new_event_loop()

asyncio_thread = ThreadedEventLoop(loop)
asyncio_thread.start() #B

app = LoadTester(loop) #C
app.mainloop()
```

#A We create a new thread class to run the asyncio event loop forever.
#B Start the new thread to run the asyncio event loop in the background.
#C Create the load tester tkinter application and start its main event loop.

We first define a `ThreadedEventLoopClass` that inherits from `Thread` to run our event loop. In this class's constructor, we take in an event loop and set the thread to be a daemon thread. We set the thread to be daemon because the asyncio event loop will block and run forever in this thread. This type of infinite loop would prevent our GUI application from shutting down if we ran in non-daemon mode. In the thread's `run` method, we call the event loop's `run_forever` method. This method is well named as it quite literally just starts the event loop running forever, blocking until we stop the event loop.

Once we've created this class, we create a new asyncio event loop with the `new_event_loop` method. We then create a `ThreadedEventLoop` instance, passing in the loop we just created and start it. This creates a new thread with our event loop running inside of it. Finally, we create an instance of our `LoadTester` app and call the `mainloop` method, kicking off the tkinter event loop.

When we run a stress test with this application, we should see the progress bar update smoothly without freezing the user interface. Our application remains responsive and we can click cancel to stop the load test whenever we please. This technique of running the asyncio event loop in a separate thread is useful for building responsive GUIs, but also is useful for any synchronous legacy applications where coroutines and asyncio don't fit nicely.

We've now seen how to utilize threads for various I/O bound workloads, but what about CPU bound workloads? Recall the GIL prevents us from running Python bytecode concurrently in threads, but there are a few notable exceptions to this that let us do some CPU bound work in threads.

7.5 Using threads for CPU bound work

The global interpreter lock is a tricky subject in Python. The rule of thumb is multithreading only makes sense for blocking I/O work, as I/O will release the GIL. This is true in most cases, but not all. In order to properly release the GIL and avoid any concurrency bugs, the code that is running needs to avoid interacting with Python objects (dictionaries, lists, Python integers, etc). This can happen when a large portion of our libraries work is done in low-level C code. There are a few notable libraries, such as hashlib and numpy, that perform CPU intensive work in pure C and release the GIL. This enables us to use multithreading to improve the performance of certain CPU bound workloads. We'll take a look at two such instances, hashing sensitive text for security and solving a data analysis problem with numpy.

7.5.1 Multithreading with hashlib

In today's world, security has never been more important. Ensuring that data is not read by hackers is key to avoiding leaking sensitive customer data, such as passwords or other information that can be used to identify or harm them.

Hashing algorithms solve this problem by taking a piece of input data and creating new piece of data that is unreadable and unrecoverable (if the algorithm is secure) to a human. For example, the password 'password' may be hashed to a string that looks more like `a12bc21df'. While no one can read or recover the input data, we're still able to check if a piece of data matches a hash. This is useful for scenarios such as validating a user's password on login or checking if a piece of data has been tampered with.

There are many different hashing algorithms today such as SHA512, BLAKE2 and scrypt, though SHA is not the best choice for storing passwords it is susceptible to brute-force attacks. Several of these algorithms are implemented in Python's hashlib library. Many functions in this library release the GIL when hashing data greater than 2048 bytes, so multithreading is an option to improve this library's performance. In addition, the scrypt function, used for hashing passwords, always releases the GIL.

Let's introduce a hopefully hypothetical scenario to see when multithreading might be useful with hashlib. Imagine you've just started a new job as principal architect at a successful organization. You manager assigns you your first bug to get started learning the company's development process, a small issue with the login system. To debug this issue, you start to look at a few database tables and to your horror you notice that all your customers passwords are stored in plaintext! This means that if your database is compromised, attackers could get all of your customers passwords and log in as them, potentially exposing sensitive data such as saved credit cards. You bring this to your manager, and he asks you to find a solution to the problem as soon as possible.

Using the scrypt algorithm to hash the plaintext passwords is a good solution for this kind of problem. It is secure and the original password is unrecoverable as it introduces a salt. A salt is a random number that ensures that the hash we get for the password is unique. To test using scrypt out, we can quickly write a synchronous script to create random passwords and hash them to get a sense of how long things will take. For this example, we'll test on 10000 random passwords.

Listing 7.16 hashing passwords with scrypt

```
import hashlib
import os
import time
import random


def random_password(length: int) -> bytes:
    ascii_lowercase = b'abcdefghijklmnopqrstuvwxyz'
    return b''.join(bytes(random.choice(ascii_lowercase)) for _ in range(length))


passwords = [random_password(10) for _ in range(10000)]
```

```
def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2, p=24, r=16))


start = time.time()

for password in passwords:
    hash(password)

end = time.time()
print(end - start)
```

We first write a function to create random lowercase passwords, and then use that to create 10,000 random passwords of ten characters each. We then hash each password with the scrypt function. We'll gloss over the details n, p and r parameters of the `scrypt` function, but these are used to tune how secure we'd like our hash to be and memory / CPU usage.

Running this on the servers you have, which are 2.4ghz 8 core machines, this code completes in just over 40 seconds, not too bad. The issue is that you have a large user base, and you need to hash one billion passwords. Doing the calculation based on this test, it will take a bit over 40 days to hash the entire database! We could split up our data set and run this procedure on multiple machines, but we'd need a lot of machines to do that given how slow this is. Can we use threading to improve the speed and therefore cut down on the time and machines we need to use? Let's apply what we know about multithreading to give this a shot. We'll create a thread pool and hash passwords in multiple threads.

Listing 7.17 hashing with multithreading and asyncio

```
import asyncio
import functools
import hashlib
import os
from concurrent.futures.thread import ThreadPoolExecutor
import random

from util import async_timed


def random_password(length: int) -> bytes:
    ascii_lowercase = b'abcdefghijklmnopqrstuvwxyz'
    return b''.join(bytes(random.choice(ascii_lowercase)) for _ in range(length))


passwords = [random_password(10) for _ in range(10000)]


def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2048, p=1, r=8))


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
```

```
    tasks = []

    with ThreadPoolExecutor() as pool:
        for password in passwords:
            tasks.append(loop.run_in_executor(pool, functools.partial(hash, password)))

    await asyncio.gather(*tasks)


asyncio.run(main())
```

This approach involves us creating a thread pool executor and creating a task for each password we want to hash. Since hashlib releases the GIL we get some decent performance gains. This code runs in about 5 seconds as opposed to the 40 we got earlier. We've just cut our runtime down from 47 days to a bit over five! As a next step we could take this application and run it concurrently on different machines to further cut down on runtime or get a machine with more CPU cores.

7.5.2 Multithreading with numpy

Numpy is an extremely popular Python library, widely used in data science and machine learning projects. It has a multitude of mathematical functions common to arrays and matrices that, generally speaking, tend to outperform plain Python arrays. This increased performance is because much of the underlying library is implemented in C and Fortran which are lower-level languages and tend to be more performant than Python.

Because many of this library's operations are in lower-level code outside of Python, this opens up the opportunity for numpy to release the GIL and allow us to multithread some of our code. The caveat here is this functionality is not well-documented, but it is generally safe to assume matrix operations can potentially be multithreaded for a performance win. That said, depending on how the numpy function is implemented, the win could be large or could be small. If the code directly calls C functions and releases the GIL there is a potential bigger win, if there is a lot of supporting Python code around any lower-level calls, the win will be smaller. Given this is not well documented, you may have to try adding multithreading to specific bottlenecks in your application (you can determine where the bottlenecks are with profiling) and benchmarking what gains you get. You'll then need to decide if the extra complexity is worth any potential gain you get.

To see this in practice, we'll create a large matrix of four billion data points in fifty rows. Our task will be to obtain the mean for reach row. Numpy has an efficient function, `mean`, to compute this. This function has an axis parameter which lets us calculate all the means across an axis without having to write a loop. In our case, an axis of 1 will calculate the mean for every row.

Listing 7.18 means of a large matrix with numpy

```
import numpy as np
import time

data_points = 4000000000
rows = 50
```

```
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)

s = time.time()

res = np.mean(matrix, axis=1)

e = time.time()
print(e - s)
```

This script first creates an array with 4 billion integer data points, ranging from 1 to 4 billion. We then 'reshape' the array into a matrix with 50 rows. Finally, we call numpy's mean function with an axis of 1, calculating the mean for each individual row. All told, this script runs in about 25-30 seconds on an 8 core 2.4ghz CPU. Let's adapt this code slightly to work with threads. We'll run the median for each row in a separate thread and use `asyncio.gather` to wait for all the median of all rows.

Listing 7.19 threading with numpy

```
import functools
from concurrent.futures.thread import ThreadPoolExecutor
import numpy as np
import asyncio
from util import async_timed


def mean_for_row(arr, row):
    return np.mean(arr[row])


data_points = 4000000000
rows = 50
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        tasks = []
        for i in range(rows):
            mean = functools.partial(mean_for_row, matrix, i)
            tasks.append(loop.run_in_executor(pool, mean))

        results = asyncio.gather(*tasks)


asyncio.run(main())
```

First, we create a `mean_for_row` function that calculates the mean for one row. Since our plan is to calculate the mean for every row in a separate thread, we can no longer use the mean function with an axis as we did before. We then create a main coroutine with a thread pool executor and create a task to calculate the mean for each row, waiting for all the calculations to finish with `gather`.

On the same machine, this code runs in roughly 9-10 seconds, nearly a 3x boost in performance! Multithreading can help us in certain cases with Numpy, however, the documentation for what can benefit from threads is lacking at the time of writing. When in doubt if threading will help a CPU bound workload, the best way to see if it will help is to test it out and benchmark.

In addition, keep in mind that your Numpy code should be as vectorized as possible before trying threading or multiprocessing to improve performance. This means avoiding things like Python loops or functions like Numpy's `apply_along_axis`, which just hides a loop. With Numpy, you will often see much better performance by pushing as much computation as you can to the library's low-level implementations.

7.6 Summary

In this chapter we've dived deep into multithreading in Python, and how to integrate it with asyncio.

- We've learned how to run I/O bound work using the threading module.
- We've learned how to cleanly terminate threads on application shutdown.
- We've learned how to use thread pool executors to distribute work to a pool of threads. This allows us to use asyncio API methods like gather to wait for results from threads.
- We've learned how to take existing blocking I/O APIs, such as requests, and run them in threads with thread pools and asyncio for performance wins.
- We've learned how to avoid race conditions with locks from the threading module. We've also learned how to avoid deadlocks with reentrant locks.
- We've learned how to run the asyncio event loop in a separate thread and send coroutines to it in a threadsafe manner. This lets us build responsive user interfaces with frameworks such as tkinter.
- We've learned how to use multithreading with hashlib and numpy. Lower-level libraries will sometimes release the GIL which lets us use threading for CPU bound work.

8
Streams

This chapter covers:

- **Transports and protocols**
- **Using streams for network connections**
- **Processing command-line input asynchronously**
- **Creating client / server applications with streams**

When writing network applications such as our echo clients in prior chapters, we've utilized the socket library to read from and write to our clients. While directly using sockets is useful when building low-level networking libraries, they are ultimately complex creatures with nuances outside the scope of this book. That said, many use cases of sockets rely on a few conceptually simple operations, such as starting a server, waiting for client connections and sending data to clients. The designers of asyncio realized this and built network stream APIs to abstract away handling the nuances of sockets for us. These higher-level APIs are much easier to work with than sockets, making any client-server applications easier to build, and more robust than using sockets ourselves. Using streams is the recommended way to build network-based applications in asyncio.

In this chapter, we'll first learn how to use the lower-level transport and protocol APIs by building a simple HTTP client. Learning about these APIs will give us the foundation for how the higher-level stream APIs work under the hood. We'll then use this knowledge to learn about stream readers and writers and use them to build a non-blocking command line SQL client. This application will asynchronously process user input, allowing us to run multiple queries concurrently from the command line. Finally, we'll learn how to use asyncio's server API to create client and server applications, building a functional chat server and chat client.

8.1 Introducing streams

In asyncio, streams are a high-level set of classes and functions that create and manage network connections and generic streams of data for us. Using them, we can create client connections to read and write to servers, or even create and manage servers ourselves. These APIs abstract away a lot of knowledge around managing sockets, such as dealing with SSL or lost connections and make our lives as developers a little easier.

The stream APIs are built on top of a lower level set of APIs known as transports and protocols. These APIs directly wrap the sockets we used in prior chapters (or in general, any generic stream of data), providing us with a clean API for reading and writing data to sockets.

These APIs are structured a little differently from what we're used to, in that they use a callback style design. Instead of actively waiting for data from a socket like we did previously, a method on a class we implement is called for us when data is available. We then process the data we receive in this method as we wish. To get started learning how these callback-based APIs work, let's first see how to use the lower-level transport and protocol APIs by building a basic HTTP client.

8.2 Transports and protocols

First, let's define what transports and protocols are. At a high level a transport is an abstraction for communication with an arbitrary stream of data. When we communicate with a socket, or any data stream such as standard in, we have a set of operations we commonly work with. We read data from or write data to a source, and when we're finished working with it, we close it. A socket cleanly fits how we've defined the transport abstraction, we read and write data to it and once we've finished, we close it. In short, a transport provides definitions for sending and receiving data to and from a source. Transports have several implementations depending on which type of source we're using. We'll mainly be concerned with `ReadTransport`, `WriteTransport` and `Transport`, though there are others for dealing with UDP connections and subprocess communication.

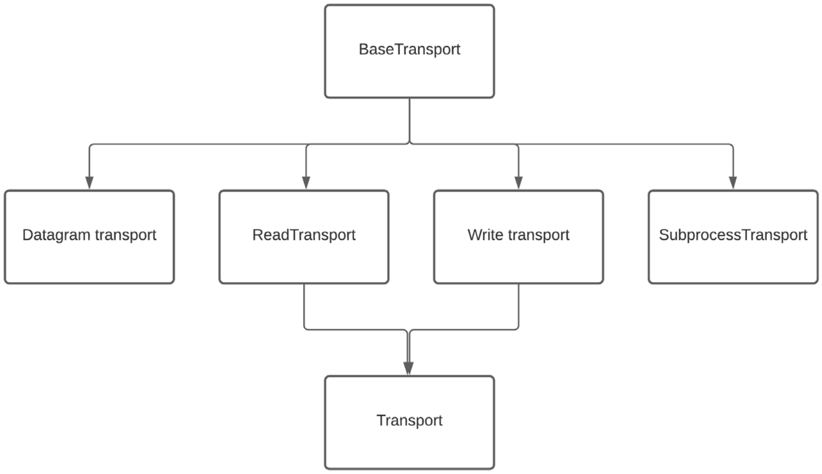


Figure 8.1 the class hierarchy of transports.

Transmitting data to and from a socket is only part of the equation, however. What about the lifecycle of a socket? We establish a connection; we write data and then process any response we get. These are the set of operations a protocol owns. Note that a protocol simply refers to a Pyton class here, and not a protocol like HTTP or FTP. A transport manages data transmission and calls methods on a protocol when events occur, such as a connection being established or data being ready to process.

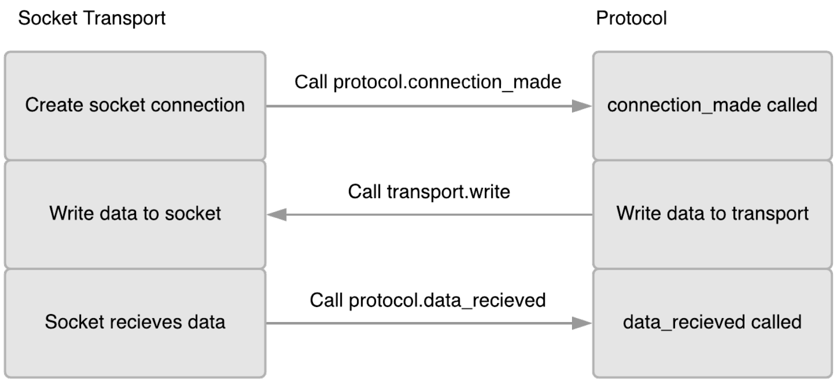


Figure 8.2: A transport calls methods on a protocol when events happen. A protocol can write data to a transport.

To understand how transports and protocols work together, we'll build a basic application to run a single HTTP GET request. The first thing we'll need to do is define a class that extends `asyncio.Protocol`. We'll implement a few methods from the base class to make the request, receive data from the request and handle any errors with the connection.

The first protocol method we'll need to implement is `connection_made`. The transport calls this method when the underlying socket has successfully connected with the HTTP server. This method has a `Transport` as an argument that we can use to communicate with the server. In this case, we'll use the transport to immediately send the HTTP request.

The second method we'll need to implement is `data_received`. The transport calls this method whenever it receives data, passing it to us as bytes. This method can be called multiple times, so we'll need to create an internal buffer to store our data.

The question now becomes how do we tell when our response is finished? To answer this, we'll implement a method called `eof_recieved`. This method is called when we receive the 'end of file', which in the case of a socket happens when the server closes the connection. Once this method is called, we are guaranteed that `data_recieved` will never be called again. The `eof_recieved` method returns a `Boolean` value that determines how to shut down the transport (close the client socket in our example). Returning `False` ensures that the transport will shut down itself, whereas `True` means that we (the protocol implementation) will shut things down ourselves. In this case, as we don't need to do any special logic on shutdown our method should return `False`, so we don't need to handle closing the transport ourselves.

With what we've described, we only have a way to store things in an internal buffer, so how do we consumers of our protocol get the result once the request is finished? To do this, we can create a `Future` internally that we use to hold the result when it is complete. Then, in the `eof_recieved` method we'll set the result of the future to the result of the HTTP response. We'll then define a coroutine we'll name `get_response` that will await the future.

Let's take what we've described above and implement it as our own protocol. We'll call it `HTTPGetClientProtocol`.

Listing 8.1 a Running a HTTP request with transports and protocols

```
import asyncio
from asyncio import Transport, Future, AbstractEventLoop
from typing import Optional


class HTTPGetClientProtocol(asyncio.Protocol):

    def __init__(self, host: str, loop: AbstractEventLoop):
        self._host: str = host
        self._future: Future = loop.create_future()
        self._transport: Optional[Transport] = None
        self._response_buffer: bytes = b''

    async def get_response(self):  # A
        return await self._future

    def _get_request_bytes(self) -> bytes:  # B
        request = f"GET / HTTP/1.1\r\n" \
                  f"Connection: close\r\n" \
                  f"Host: {self._host}\r\n\r\n"
        return request.encode()

    def connection_made(self, transport: Transport):
        print(f'Connection made to {self._host}')
        self._transport = transport
        self._transport.write(self._get_request_bytes())  # C

    def data_received(self, data):
        print(f'Data received!')
        self._response_buffer = self._response_buffer + data  # D

    def eof_received(self) -> Optional[bool]:
        self._future.set_result(self._response_buffer.decode())  # E
        return False

    def connection_lost(self, exc: Optional[Exception]) -> None:  # F
        if exc is None:
            print('Connection closed without error.')
        else:
            self._future.set_exception(exc)
```

#A Await the internal future until we get a response from the server
#B Create the HTTP request, we use connection: close to close the socket after the request
#C Once we've established a connection, use the transport to send the request
#D Once we have data, save it to our internal buffer
#E Once the connection closes, complete the future with the buffer
#F If the connection closes without error do nothing, otherwise complete the future with an exception

Now that we've implemented our protocol, let's see how to use it to make a real request. We'll need to learn a new coroutine method on the asyncio event loop named `create_connection` to do this. This method will create a socket connection to a given host for us and wrap it in an appropriate transport. In addition to a host and port, it takes in a protocol factory. A protocol factory is a function that creates protocol instances, in our case an instance of the `HTTPGetClientProtocol` class we just created. When we call this coroutine, we're returned both the transport that the coroutine created along with the protocol instance the factory created.

Listing 8.2 using the protocol

```
import asyncio
from asyncio import AbstractEventLoop
from chapter_08.listing_8_1 import HTTPGetClientProtocol


async def make_request(host: str, port: int, loop: AbstractEventLoop) -> str:
    def protocol_factory():
        return HTTPGetClientProtocol(host, loop)

    _, protocol = await loop.create_connection(protocol_factory, host=host, port=port)

    return await protocol.get_response()


async def main():
    loop = asyncio.get_running_loop()
    result = await make_request('www.example.com', 80, loop)
    print(result)


asyncio.run(main())
```

We first define a `make_request` method that takes in the host and port we'd like to make a request to and the server's response. Inside this method, we create an inner method for our protocol factory, this just creates a new HTTPGetClientProtocol. We then call create_connection with the host and port which returns both a transport and the protocol our factory created. We won't need the transport, so we just ignore it, but we will need the protocol as we'll want to use the `get_response` coroutine so we'll keep track of it in the `protocol` variable. Finally, we await the `get_response` coroutine of our protocol which will wait until the HTTP server has responded with a result. In our main coroutine, we await `make_request` and print the response. Executing this, you should see a HTTP response similar to the following (we've omitted the HTML body for brevity):

```
Connection made to www.example.com
Data received!
HTTP/1.1 200 OK
Age: 193241
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Connection closed without error.
```

Now we know how to use transports and protocols. These APIs are lower-level and as such aren't the recommended way to work with streams in asyncio. Let's see how to use a higher-level abstraction that expands on transports and protocols called streams.

8.3 Stream readers and stream writers

Transports and protocols are lower-level APIs that are best suited for when we need direct control over what is happening as we send and receive data. As an example, if we're designing a networking library or web framework, we may consider transports and protocols. For most applications, we don't need this level of control and using transports and protocols would involve us writing a bunch of repetitive code.

The designers of asyncio realized this and created the higher-level streams APIs. This API encapsulates the standard use cases of transports and protocols into two easy to understand and use classes, `StreamReader` and `StreamWriter`, which as you can guess handle reading from and writing to streams respectively. Using these classes is the recommended way to develop networking applications in asyncio.

To get an understanding of how to use these APIs, let's take our example of making a HTTP GET request and translate it into streams. Instead of directly instantiating `StreamReader` and `StreamWriter` instances, asyncio provides a library coroutine function named `open_connection` that will create them for us. This coroutine takes in a host and port that we'll connect to and returns a `StreamReader` and a `StreamWriter` as a tuple. Our plan will be to use the `StreamWriter` to send out the HTTP request and the `StreamReader` to read the response. `StreamReader` methods are easy to understand, we have a convenient `readline` coroutine that waits until we have a line of data. Alternatively, we could also use `StreamReader`'s `read` coroutine which waits for a specified number of bytes to arrive.

`StreamWriter` is a little more complex. It has a `write` method as we'd expect, but it is a plain method and *not* a coroutine. Internally, stream writers try to write to a socket's output buffer right away, but this buffer can potentially be full. If the socket's write buffer is full, the data is instead stored in an internal queue where it can later go into to the buffer. This poses a potential problem in that calling write does not necessarily send out data right away. This can cause potential memory issues. Imagine our network connection becomes slow and can only send out 1KB per second, but our application is writing out 1MB per second. In this case our application's write buffer will fill up at a much faster rate that we can send the data out to the socket's buffer, and eventually we'll start to hit memory limits on the machine, inviting a crash.

How can we wait until all of our data is properly sent out? To solve this issue, we have a coroutine method called `drain`. This coroutine will block until all queued data gets sent to the socket, ensuring we've written everything before moving on. The pattern we'll want to use is after we call `write` we'll always `await` a call to `drain`. It is technically not necessary to call `drain` after every `write`, but it is a good idea to help prevent bugs.

Listing 8.3 an http request with stream readers and writers

```
import asyncio
from asyncio import StreamReader
```

```
from typing import AsyncGenerator


async def read_until_empty(stream_reader: StreamReader) -> AsyncGenerator[str, None]:
    while response := await stream_reader.readline(): #A
        yield response.decode()


async def main():
    host: str = 'www.example.com'
    request: str = f"GET / HTTP/1.1\r\n" \
                   f"Connection: close\r\n" \
                   f"Host: {host}\r\n\r\n"

    stream_reader, stream_writer = await asyncio.open_connection('www.example.com', 80)

    try:
        stream_writer.write(request.encode()) #B
        await stream_writer.drain()

        responses = [response async for response in read_until_empty(stream_reader)] #C

        print(''.join(responses))
    finally:
        stream_writer.close() #D
        await stream_writer.wait_closed()


asyncio.run(main())
```

#A Read a line and decode it until we don't have any left.
#B Write the http request and drain the writer.
#C Read each line and store It in a list
#D Close the writer and wait for it to finish closing.

In listing 8.3 we first create a convenience async generator to read all lines from a `StreamReader`, decoding them into strings until we don't have any left to process. Then, in our main coroutine we open a connection to example.com, creating a `StreamReader` and `StreamWriter` instance in the process. We then write the request and drain the stream writer with `write` and `drain` respectively. Once we've written our request, we use our async generator to get each line from the response back, storing them in the `responses` list. Finally, we close the `StreamWriter` instance by calling `close` and then awaiting the `wait_closed` coroutine. Why do we need to call a method *and* a coroutine here? When we call close a few things happen, such as deregistering the socket and calling the underlying transport's `connection_lost` method. These all happen asynchronously on a later iteration of the event loop, meaning that directly after we call `close` our connection isn't really closed until sometime later. If you need to wait for the connection to close before proceeding or are concerned about any exceptions that may happen while you're closing, calling `wait_closed` is best practice.

We've now learned the basics around the stream APIs by making web requests. The usefulness of these classes extends beyond web and network-based applications. Next, we'll see how utilize stream readers to create non-blocking command-line applications.

8.4 Non-blocking command line input

Traditionally in Python, when we need to get user input, we use the `input` function. This function will stop execution flow until the user has provided input and hit enter. What if we want to run code in the background while still remaining responsive to input? For example, we may want to let the user kick off multiple long running tasks concurrently, such as long-running SQL queries. In the case of a command-line chat application, we likely want the user to be able to type a message while receiving messages from other users.

Since asyncio is single threaded, using `input` in an asyncio application means we stop the event loop from running until the user provides input, halting our entire application. Even using tasks to kick off an operation in the background won't work. To demonstrate this, let's attempt to create an application where the user enters a time for the application to sleep. We'd like to be able to run multiple of these sleep operations concurrently while still accepting user input, so we'll ask for the number of seconds to sleep and create a `delay` task in a loop.

Listing 8.4 attempting background tasks

```
import asyncio
from util import delay


async def main():
    while True:
        delay_time = input('Enter a time to sleep:')
        asyncio.create_task(delay(int(delay_time)))


asyncio.run(main())
```

If this code worked the way we intended, after we input a number, we'd expect to see `sleeping for n second(s)` printed out followed by `finished sleeping for n second(s)` *n* seconds later. However, this isn't the case, we see nothing except our prompt to enter a time to sleep. This is because there is no `await` inside our code and therefore the task never gets a chance to run on the event loop. We can hack around this by putting `await asyncio.sleep(0)` after the `create_task` line which will schedule the task (this is known as 'yielding to the event loop' and will be covered in chapter 14). Even with this trick, the `input` call still blocks any background task we create from running to completion as it stops the entire thread.

What we really want is for the `input` function to be a coroutine instead so we could say something like `delay_time = await input('Enter a time to sleep:')`. If we were able to do this, our task would schedule properly and continue to run while we waited for user input. Unfortunately, there is no coroutine variant of `input`, so we'll need to do something different.

This is where protocols and stream readers can help us out. Recall that a stream reader has the `readline` coroutine which is the exact type of coroutine we're looking for. If we had a way to hook a stream reader to standard input, we could then use this coroutine for user input.

Asyncio has a coroutine method on the event loop called `connect_read_pipe` that connects a protocol to a file-like object, which is very close to what we want. This coroutine method accepts a *protocol factory* and a *pipe*. A protocol factory is just a function that creates a protocol instance. A pipe is a 'file like object', which is defined as an object with methods such as `read` and `write` on it. The `connect_read_pipe` coroutine will then connect the pipe to the protocol the factory creates, taking data from the pipe and sending it to the protocol.

In terms of standard console input, `sys.stdin` fits the bill of a file-like object that we can pass in to `connect_read_pipe`. Once we call this coroutine, we'll get a tuple of the protocol our factory function created and a `ReadTransport`. The question now becomes what protocol should we create in our factory and how do we connect this with a `StreamReader`, which has the `readline` coroutine we'd like to use?

Asyncio provides a utility class called `StreamReaderProtocol` for connecting instances of stream readers to protocols. When we instantiate this class, we pass in an instance of a stream reader. The protocol class then delegates to the stream reader we created, allowing us to use the stream reader to read data from standard in. Putting all these pieces together, we can create a command line application that does not block the event loop when waiting for user input.

FOR WINDOWS USERS

Unfortunately on Windows `connect_read_pipe` will not work with `sys.stdin`. This is due to an as of yet unfixed bug caused by the way Windows implements file descriptors. For this to work on windows, you'll need to call `sys.stdin.readline()` in a separate thread using techniques we explored in chapter seven. You can read more about this issue at https://bugs.python.org/issue26832.

Since we'll be reusing the asynchronous standard in reader throughout the rest of the chapter, let's create it in its own file, `listing_8_5.py`. We'll then import it in the rest of the chapter.

Listing 8.5 an asynchronous standard in reader

```
import asyncio
from asyncio import StreamReader
import sys


async def create_stdin_reader() -> StreamReader:
    stream_reader = asyncio.StreamReader()
    protocol = asyncio.StreamReaderProtocol(stream_reader)
    loop = asyncio.get_running_loop()
    await loop.connect_read_pipe(lambda: protocol, sys.stdin)
    return stream_reader
```

In listing 8.5 we create a reusable coroutine named `create_stdin_reader` which creates a `StreamReader` that we'll use to asynchronously read standard in. We first create a stream reader instance and pass it to a stream reader protocol. We then call `connect_read_pipe`, passing in a protocol factory as a lambda function. This lambda returns the stream reader protocol we created earlier. We also pass `sys.stdin` to connect standard in to our stream reader protocol. We ignore the transport and protocol that `connect_read_pipe` returns, as we won't need them. We can now use this function to asynchronously read from standard in and build our application.

Listing 8.6 using stream readers for input

```
import asyncio
from chapter_08.listing_8_5 import create_stdin_reader
from util import delay


async def main():
    stdin_reader = await create_stdin_reader()
    while True:
        delay_time = await stdin_reader.readline()
        asyncio.create_task(delay(int(delay_time)))


asyncio.run(main())
```

In our main coroutine, we call `create_stdin_reader` and loop forever, waiting for input from the user with the `readline` coroutine. This coroutine will give us the input text entered once a user hits enter on the keyboard. Once we have input from the user, we convert it into an integer (note for a real application we should add code to handle bad input, as we'll crash if we pass in a string right now) and create a delay task. Running this, you'll be able to run multiple delay tasks concurrently while still entering command line input. For instance, entering delays of 5, 4 and 3 seconds respectively you should see the following output:

```
5
sleeping for 5 second(s)
4
sleeping for 4 second(s)
3
sleeping for 3 second(s)
finished sleeping for 5 second(s)
finished sleeping for 4 second(s)
finished sleeping for 3 second(s)
```

This works but has a critical flaw. What happens if a message appears on the console while we're typing an input delay time? To test this out, we'll enter a delay time of three seconds and then start rapidly pressing 1. Doing this, we'll see something like the following:

```

3
sleeping for 3 second(s)
111111finished sleeping for 3 second(s)
```

```
11
```

While we were typing, the message from our delay task prints out, disrupting our input line and forcing it to continue on the next line. In addition, the input buffer is now only `11`, meaning if we press enter, we'll create a delay task for that amount if time, losing out first few pieces of input. This is because by default, the terminal runs in *cooked* mode. In this mode, the terminal echoes user input to standard out and also processes special keys for us, such as enter and CTRL+C. This issue arises because the `delay` coroutine writes to standard out at the same time the terminal is echoing output, causing a race condition.

There is also a second issue in that the terminal has a single position on the screen where standard out writes to. This is known as a *cursor* and is much like a cursor you'd see in a word processor. As we enter input, the cursor is on the line where our keyboard input prints out. This means that any output messages from other coroutines will print on the same line as our input since this is where the cursor is, causing odd behavior.

To solve these issues, we need a combination of two solutions. The first is to bring the echoing of input from the terminal into our Python application. This will ensure that while echoing input from the user, we don't write any output messages from other coroutines since we're single threaded. The second is to move the cursor around the screen when we write output messages, ensuring that we don't write output messages on the same line as our input. We can do these by manipulating the settings of terminal itself and using escape sequences.

8.4.1 Terminal raw mode and the read coroutine

Because our terminal is running in cooked mode, it handles echoing user input on `readline` for us outside of our application. How can we bring this processing into our application so we can avoid the race conditions we saw previously?

The answer is switching the terminal to *raw* mode. In raw mode, instead of the terminal doing buffering, preprocessing and echoing for us, every individual keystroke is sent to the application. It is then up to us to echo and preprocess as we'd like. While this means we'll need to do extra work, it also means we have fine-grained control around writing to standard out, giving us the needed power to avoid race conditions.

Python allows us to change the terminal to raw mode, but also allows for *cbreak* mode. This mode behaves like raw mode, with the difference that keystrokes like CTRL+C will still be interpreted for us, saving us some work. We can enter raw mode by using the `tty` module and the `setcbreak` function like so:

```
import tty
import sys
tty.setcbreak(sys.stdin)
```

Once we're in cbreak mode, we'll need to rethink how we designed our application a bit. The `readline` coroutine will no longer work as it won't echo any input for us in raw mode. Instead, we'll want to read one character at a time and store it in our own internal buffer, echoing each character typed in. The standard in stream reader we created has a method called `read` which takes in a number of bytes to read from the stream. Calling `read(1)` will read one character at a time for us, which we can then store in a buffer and echo to standard out.

We now have two pieces of the puzzle to solve this, entering into cbreak mode and reading one input character at a time, echoing it to standard out. Now we need to think through how to display the output of the delay coroutine so that it won't interfere with our input.

Let's define a few requirements to make our application more user-friendly and solve the issue with output writing on the same line as input. We'll then let these requirements inform how we implement things:

1. The user input field should always remain at the bottom of the screen.
2. Coroutine output should start from the top of the screen and move down.
3. When there are more messages than available lines on the screen, existing messages should scroll up.

Given these requirements, how can we display the output from the delay coroutine? Given that we want to scroll messages up when there are more messages than available lines, writing directly to standard out with print will prove tricky. Instead of doing this, the approach we'll take is keeping a deque of the messages we want to write to standard out. We'll set the maximum number of elements in the deque to the number of rows on the terminal screen. This will give us the scrolling behavior we want when the deque is full, as items in the back of the deque will be discarded. When a new message is appended to the deque, we'll move to the top of the screen and redraw each message. This will get us the scrolling behavior we desire, without having to keep much information about the state of standard out. This makes our application flow look like the following:

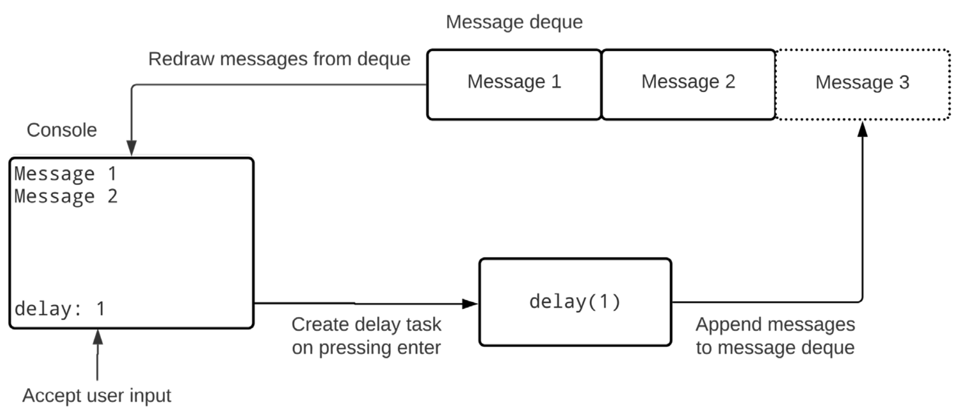


Figure 8.3: the delay console application

Our game plan for the application will then be as follows:

1. Move the cursor to the bottom of the screen, when a key is pressed, append it to our internal buffer and echo the keypress to standard out.
2. When the user hits enter, create a delay task. Instead of writing output messages to standard out, we'll append them to a deque that has a maximum number of elements equal to the number of rows on the console.
3. Once a message goes into the deque, we'll redraw the output on the screen. We first move the cursor to the top left of the screen. We then print out all messages in the deque. Once we're done, we return the cursor to the input row and column where it was before.

To implement the application in this way, we'll first need to learn how to move the cursor around the screen. We can use ANSI escape codes to do this. These are special codes we can write to standard out do things like change color text, move the cursor up or down and delete lines. Escape sequences are first introduced with an escape code, in Python, we can do this by printing `\033` to the console. Many of the escape sequences we'll need to use are introduced by *control sequence introducers*, which are started by printing `\033[`. To better understand this, let's see how to move the cursor to five lines below where it currently is.

```
sys.stdout.write('\033[5E')
```

This escape sequence starts with the control sequence introducer followed by 5E. Five here is number of rows from the current cursor row we'd like to move down, and E is the code for "move the cursor down this number of lines". Escape sequences are pretty terse and a little hard to follow. In the next listing, we'll create several functions with clear names to explain what each escape code does, and we'll import them in future listings. If you'd like more explanation on ANSI escape sequences and how they work, the Wikipedia article on the subject has great information at https://en.wikipedia.org/wiki/ANSI_escape_code.

Let's think through how we'll need to move the cursor around the screen to figure out which functions we'll need to implement. First, we'll need to move the cursor to the bottom of the screen to accept user input. Then, once a user hits enter, we'll need to clear any text they have entered. To print coroutine output messages from the top of the screen, we'll need to be able to move to the first line of the screen. We'll also need to save and restore the current position of the cursor, since while we're typing a message from a coroutine may print a message, meaning we'll need to move it back to the proper spot. We can do these with the following escape code functions:

Listing 8.7 escape sequence convenience functions

```
import sys
import shutil

def save_cursor_position():
    sys.stdout.write('\0337')


def restore_cursor_position():
```

```
    sys.stdout.write('\0338')


def move_to_top_of_screen():
    sys.stdout.write('\033[H')


def delete_line():
    sys.stdout.write('\033[2K')


def clear_line():
    sys.stdout.write('\033[2K\033[0G')


def move_back_one_char():
    sys.stdout.write('\033[1D')


def move_to_bottom_of_screen() -> int:
    _, total_rows = shutil.get_terminal_size()
    input_row = total_rows - 1
    sys.stdout.write(f'\033[{input_row}E')
    return total_rows
```

Now that we have a set of reusable functions to move the cursor around the screen, let's implement a reusable coroutine for reading standard in one character at a time. We'll use the `read` coroutine to read one character at a time. Once we have a character, we'll write it to standard out, storing the character in an internal buffer. Since we also want to handle a user pressing delete, we'll watch for the delete key. When a user presses it, we'll delete the character from the buffer and standard out.

Listing 8.8 a reading input one character at a time

```
import sys
from asyncio import StreamReader
from collections import deque
from chapter_08.listing_8_7 import move_back_one_char, clear_line


async def read_line(stdin_reader: StreamReader) -> str:
    def erase_last_char(): #A
        move_back_one_char()
        sys.stdout.write(' ')
        move_back_one_char()

    delete_char = b'\x7f'
    input_buffer = deque()
    while (input_char := await stdin_reader.read(1)) != b'\n':
        if input_char == delete_char: #B
            if len(input_buffer) > 0:
                input_buffer.pop()
                erase_last_char()
        else:
            input_buffer.append(input_char) #C
            sys.stdout.write(input_char.decode())
```

```
    clear_line()
    return b''.join(input_buffer).decode()
```

#A convenience function to delete the previous character from standard out
#B If the input character is backspace, remove the last character
#C If the input character is not backspace, append it to the buffer and echo.

Our coroutine takes in a stream reader that we've attached to standard in. We then define a convenience function to erase the previous character from standard out, we'll need this when a user presses delete. We then enter a while loop reading character by character until the user hits enter. If a user hits delete, we remove the last character from the buffer and from standard out. Otherwise, we append it to the buffer and echo it. Once the user hits enter, we clear the input line and return the contents of the buffer.

Next, we'll need to define the queue where we'll store the messages we want to print to standard out. Since we want to redraw output whenever we append a message, we'll define a class that wraps a deque and takes in a callback awaitable. The callback we pass in will be responsible for redrawing output. We'll also add an `append` coroutine method to our class that will append items to the deque and call the callback with the current set of items in the deque.

Listing 8.9 a message store

```
from collections import deque
from typing import Callable, Deque, Awaitable


class MessageStore:
    def __init__(self, callback: Callable[[Deque], Awaitable[None]], max_size: int):
        self._deque = deque(maxlen=max_size)
        self._callback = callback

    async def append(self, item):
        self._deque.append(item)
        await self._callback(self._deque)
```

Now we have all the pieces to create the application. We'll rewrite our delay coroutine to add messages to the message store. Then, in our main coroutine, we'll create a helper coroutine to redraw messages in our deque to standard out. This is the callback we'll pass to our `MessageStore`. Then, we'll use the `read_line` coroutine we implemented earlier to accept user input, creating a delay task when the user hits enter.

Listing 8.10 the asynchronous delay application

```
import asyncio
import os
import tty
from collections import deque
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore
```

```
async def sleep(delay: int, message_store: MessageStore):
    await message_store.append(f'Starting delay {delay}') #A
    await asyncio.sleep(delay)
    await message_store.append(f'Finished delay {delay}')


async def main():
    tty.setcbreak(sys.stdin)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    async def redraw_output(items: deque): #B
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            print(item)
        restore_cursor_position()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()

    while True:
        line = await read_line(stdin_reader)
        delay_time = int(line)
        asyncio.create_task(sleep(delay_time, messages))


asyncio.run(main())
```

#A Append the output messages to the message store
#B Callback to move the cursor to the top of the screen, redraw output and move the cursor back

Running this, you'll be able to create delays and watch input write to the console even as you type. While it is more complicated than our first attempt, we've built an application that avoids the problems writing to standard out that we saw earlier.

What we've built works for the delay coroutine, but what about something more real-world? The pieces we've just defined are robust enough we can make more useful applications by reusing them. For example, let's think through how to create a command-line SQL client. Certain queries may take a long time to execute, but we may want to run other queries in the meanwhile or cancel a running query. Using what we've built, we can create this type of client. Let's build one using our previous e-commerce product database from chapter 5 where we created a schema with a set of clothing brands, products and skus. We'll create a connection pool to connect to our database, and we'll reuse our code from previous examples to accept and run queries. We'll output basic information about the queries to the console, for now just the number of rows returned.

Listing 8.11 an asynchronous command line sql client

```
import asyncio
import asyncpg
```

```
import os
import tty
from collections import deque
from asyncpg.pool import Pool
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore


async def run_query(query: str, pool: Pool, message_store: MessageStore):
    async with pool.acquire() as connection:
        try:
            result = await connection.fetchrow(query)
            await message_store.append(f'Fetched {len(result)} rows from: {query}')
        except Exception as e:
            await message_store.append(f'Got exception {e} from: {query}')


async def main():
    tty.setcbreak(0)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            print(item)
        restore_cursor_position()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()

    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:

        while True:
            query = await read_line(stdin_reader)
            asyncio.create_task(run_query(query, pool, messages))


asyncio.run(main())
```

Our code is almost exactly the same as before, with the difference that instead of a delay coroutine we create a `run_query` coroutine. Instead of just sleeping for an arbitrary amount of time, this runs a query the user gave us which can take an arbitrary amount of time. This let us issue new queries from the command line while others are still running and lets us see output from completed ones even as we are typing in new queries.

We now know how to create command-line clients that can handle input while other code executes and writes to the console. Next, we'll learn how to create servers using higher-level asyncio APIs.

8.5 Creating servers

When we have built servers, such as our echo server previously, we've created a server socket, bound it to a port and waited for incoming connections. While this works, asyncio lets us create servers at a higher level of abstraction, meaning we can create them without ever worrying about managing sockets on our own. Creating servers this way simplifies the code we need to write with sockets, and as such using these higher-level APIs is the recommended way to create and manage servers using asyncio.

We can create a server with the `asyncio.start_server` coroutine. This coroutine takes in several optional parameters to configure things such as SSL, but the main parameters we'll be interested in are the `host`, `port` and `client_connected_cb`. The host and port are like we've seen before, it is the address that the server socket will listen for connections. The more interesting piece is `client_connected_cb`, this is either a callback function or coroutine that will run whenever a client connects to the server. This callback takes in a `StreamReader` and `StreamWriter` as parameters which will let us read and write to and from the client that connected.

When we await `start_server`, it will return an `AbstractServer` object. This class does not have too many interesting methods that we'll need to use other than `serve_forever`, which as you can guess runs the server forever until we terminate it. This class is also an asynchronous context manager. This means we can use an instance of it with `async with` syntax to have the server properly shut down on exit.

To get a handle on how to create servers, let's create an echo server like before, but make it a little more advanced. Instead of just echoing back output, we'll display information about how many other clients are connected. We'll also display information when a client disconnects from the server. To manage this, we'll create a class we'll call `ServerState` to manage how many users are connected. Once a user connects, we'll add them to the server state and notify other clients that they connected.

Listing 8.12 creating an echo server with server objects

```
import asyncio
import logging
from asyncio import StreamReader, StreamWriter


class ServerState:

    def __init__(self):
        self._writers = []

    async def add_client(self, reader: StreamReader, writer: StreamWriter): #A
        self._writers.append(writer)
        await self._on_connect(writer)
        asyncio.create_task(self._echo(reader, writer))
```

```
    async def _on_connect(self, writer: StreamWriter): #B
        writer.write(f'Welcome! {len(self._writers)} user(s) are online!\n'.encode())
        await writer.drain()
        await self._notify_all('New user connected!\n')

    async def _echo(self, reader: StreamReader, writer: StreamWriter): #C
        try:
            while (data := await reader.readline()) != b'':
                writer.write(data)
                await writer.drain()
            self._writers.remove(writer)
            await self._notify_all(f'Client disconnected. {len(self._writers)} user(s) are
      online!\n')
        except Exception as e:
            logging.exception('Error reading from client.', exc_info=e)
            self._writers.remove(writer)

    async def _notify_all(self, message: str): #D
        for writer in self._writers:
            try:
                writer.write(message.encode())
                await writer.drain()
            except ConnectionError as e:
                logging.exception('Could not write to client.', exc_info=e)
                self._writers.remove(writer)


async def main():
    server_state = ServerState()

    async def client_connected(reader: StreamReader, writer: StreamWriter) -> None: #E
        await server_state.add_client(reader, writer)

    server = await asyncio.start_server(client_connected, '0.0.0.0', 8000) #F

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

#A Add a client to the server state and create an echo task
#B On a new connection, tell the client how many users are online, and notify others of a new user.
#C Handle echoing user input, when a client disconnects, notify other users of a disconnect.
#D Helper method to send a message to all other users. If a message fails to send, remove that user.
#E When a client connects, add that client to the server state.
#F Start the server and start serving forever.

When a user connects to our server, our `client_connected` callback fires with a reader and writer for that user which in turn calls the server state's `add_client` coroutine. In the `add_client` coroutine, we store the `StreamWriter` so that we can send messages to all connected clients and also remove it when a client disconnects. We then call `_on_connect` which sends a message to the client letting them know how many other users are connected. In `_on_connect`, we also notify any other connected clients that a new user has connected.

The `_echo` coroutine is similar to what we've done in the past, with the twist that when a user disconnects, we notify any other connected clients that someone disconnected. When running this, you should have a fully-functioning echo server that lets each individual client know when a new user connects and disconnects from the server.

We've now seen how to create an asyncio server that is a little more advanced than what we've done previously. Next, let's build on top of this knowledge and create something even more advanced, a chat server and chat client.

8.6 Creating a chat server and client

We now know how to both create servers and handle asynchronous command line input. We can combine what we know in these two areas to create two applications. The first, a chat server which accepts multiple chat clients at the same time, and the second, a chat client which connects to the server and sends and receives chat messages.

Before we begin designing our application, let's start with a few requirements that will help us make the correct design choices. First, for our server:

1. A chat client should be able to connect to the server if they provide a username.
2. Once a user is connected, they should be able to send chat messages to the server, each message should be sent to every user connected to the server.
3. To prevent idle users taking up resources, if a user is idle for more than a minute, the server should disconnect them.

Second, for our client:

1. When a user starts the application, it should ask for a username and attempt to connect to the server.
2. Once connected, the user will see any messages from other clients scroll down from the top of the screen.
3. The user should have an input field at the bottom of the screen. When the user hits enter, the text in the input should be sent to the server and then to all other connected clients.

Given these requirements, let's first think through what our communication between the client and server should look like. First, we'll need to send a message from the client to the server with our username. We need to disambiguate connecting with a username from a message send, so we'll introduce a simple command protocol to indicate we're sending a username. To keep things simple, we'll just pass a string with a command name called `CONNECT` followed by the user-provided username. For example, `CONNECT MissIslington` will be the message we'll send to the server to connect a user with the username MissIslington.

Once we've connected, we'll just send messages directly to the server, which will then send the message to all connected clients (including ourselves, you could potentially optimize this away). For a more robust application, you may want to consider a command that the server sends back to the client to acknowledge the message was received, but we'll skip this for brevity.

With this in mind, we have enough to start designing our server. We'll create a ChatServerState class similar to what we did in the previous section. Once a client connects, we'll wait for them to provide a username with the CONNECT command. Assuming they do, we'll create a task to listen for messages from the client and write them to all other connected clients. To keep track of connected clients, we'll keep a dictionary of the connected usernames to their StreamWriter instances. If a connected user is idle for more than a minute, we'll disconnect them and remove them from the dictionary, sending a message to other users that they left the chat.

Listing 8.13 a chat server

```
import asyncio
import logging
from asyncio import StreamReader, StreamWriter


class ChatServer:

    def __init__(self):
        self._username_to_writer = {}

    async def start_chat_server(self, host: str, port: int):
        server = await asyncio.start_server(self.client_connected, host, port)

        async with server:
            await server.serve_forever()

    async def client_connected(self, reader: StreamReader, writer: StreamWriter): #A
        command = await reader.readline()
        print(f'CONNECTED {reader} {writer}')
        command, args = command.split(b' ')
        if command == b'CONNECT':
            username = args.replace(b'\n', b'').decode()
            self._add_user(username, reader, writer)
            await self._on_connect(username, writer)
        else:
            logging.error('Got invalid command from client, disconnecting.')
            writer.close()
            await writer.wait_closed()

    def _add_user(self, username: str, reader: StreamReader, writer: StreamWriter): #B
        self._username_to_writer[username] = writer
        asyncio.create_task(self._listen_for_messages(username, reader))

    async def _on_connect(self, username: str, writer: StreamWriter): #C
        writer.write(f'Welcome! {len(self._username_to_writer)} user(s) are
       online!\n'.encode())
        await writer.drain()
        await self._notify_all(f'{username} connected!\n')

    async def _remove_user(self, username: str):
        writer = self._username_to_writer[username]
        del self._username_to_writer[username]
        try:
            writer.close()
            await writer.wait_closed()
```

```
        except Exception as e:
            logging.exception('Error closing client writer, ignoring.', exc_info=e)

    async def _listen_for_messages(self,
                                   username: str,
                                   reader: StreamReader): #D
        try:
            while (data := await asyncio.wait_for(reader.readline(), 60)) != b'':
                await self._notify_all(f'{username}: {data.decode()}')
            await self._notify_all(f'{username} has left the chat\n')
        except Exception as e:
            logging.exception('Error reading from client.', exc_info=e)
            await self._remove_user(username)

    async def _notify_all(self, message: str): #E
        inactive_users = []
        for username, writer in self._username_to_writer.items():
            try:
                writer.write(message.encode())
                await writer.drain()
            except ConnectionError as e:
                logging.exception('Could not write to client.', exc_info=e)
                inactive_users.append(username)

        [await self._remove_user(username) for username in inactive_users]


async def main():
    chat_server = ChatServer()
    await chat_server.start_chat_server('0.0.0.0', 8000)


asyncio.run(main())
```

#A Wait for the client to provide a valid username command, otherwise disconnect them.
#B Store a user's stream writer instance and create a task to listen for messages.
#C Once a user connects, notify all others that they have connected.
#D Listen for messages from a client and send them to all other clients, waiting a maximum of a minute for a message.
#E Send a message to all connected clients, removing any disconnected users.

Our chat server class encapsulates everything about our chat server in one clean interface. The main entry point is the `start_chat_server` coroutine. This coroutine starts a server on the specified host and port and calls `serve_forever`. For our server's client connected callback, we use our `client_connected` coroutine. This coroutine waits for the first line of data from the client, if it receives a valid `CONNECT` command, it calls `_add_user` and then `_on_connect`, otherwise it terminates the connection.

The `_add_user` function stores the username and user's stream writer in an internal dictionary and then creates a task to listen for chat messages from the user. The `_on_connect` coroutine sends a message to the client welcoming them to the chat room and then notifies all other connected clients that the user connected.

When we called `_add_user` we created a task for the `_listen_for_messages` coroutine. This coroutine is where the meat of our application lies. We loop forever, reading messages from the client until we see an empty line, indicating the client disconnected. Once we get a message, we call `_notify_all` to send the chat message to all connected clients. To satisfy the requirement that a client should be disconnected after being idle for a minute, we wrap our `readline` coroutine in `wait_for`. This will throw a `TimeoutError` if the client has idled for longer than a minute. In this case, we have a broad exception clause that catches `TimeoutError` and any other exceptions thrown. We handle any exception by removing the client from the `_username_to_writer` dictionary so we don't send messages to them anymore.

We now have a complete server, but our server is meaningless without a client to connect to it. We'll implement the client similarly to the command-line SQL client we wrote earlier. We'll create a coroutine to listen for messages from the server and append them to a message store, redrawing the screen when a new message comes in. We'll also put the input at the bottom of the screen, and when the user hits enter, we'll send the message to the chat server.

Listing 8.14 the chat client

```
import asyncio
import os
import logging
import tty
from asyncio import StreamReader, StreamWriter
from collections import deque
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore


async def send_message(message: str, writer: StreamWriter):
    writer.write((message + '\n').encode())
    await writer.drain()


async def listen_for_messages(reader: StreamReader,
                              message_store: MessageStore):  # A
    while (message := await reader.readline()) != b'':
        await message_store.append(message.decode())
    await message_store.append('Server closed connection.')


async def read_and_send(stdin_reader: StreamReader,
                        writer: StreamWriter):  # B
    while True:
        message = await read_line(stdin_reader)
        await send_message(message, writer)


async def main():
    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
```

```
            sys.stdout.write(item)
        restore_cursor_position()

    tty.setcbreak(0)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()
    sys.stdout.write('Enter username: ')
    username = await read_line(stdin_reader)

    reader, writer = await asyncio.open_connection('0.0.0.0', 8000)  # C

    writer.write(f'CONNECT {username}\n'.encode())
    await writer.drain()

    message_listener = asyncio.create_task(listen_for_messages(reader, messages))  # D
    input_listener = asyncio.create_task(read_and_send(stdin_reader, writer))

    try:
        await asyncio.wait([message_listener, input_listener],
       return_when=asyncio.FIRST_COMPLETED)
    except Exception as e:
        logging.exception(e)
        writer.close()
        await writer.wait_closed()


asyncio.run(main())
```

#A Listen for messages from the server, appending them to the message store.
#B Read input from the user and send it to the server.
#C Open a connection to the server and send the connect message with the username.
#D Create a task to listen for messages and listen for input, wait until one finishes.

We first ask the user for their username, once we have one, we send our CONNECT message to the server. Then we create two tasks, one to listen for messages from the server and one to continually read chat messages and send them to the server. We then take these two tasks and wait for whichever one completes first by wrapping them in `asyncio.wait`. We do this because the server could disconnect us, or the input listener could throw an exception. If we just awaited each task independently, we could find ourselves stuck. For instance, if the server disconnected us, we'd have no way to stop the input listener if we had awaited that task first. Using the `wait` coroutine prevents this issue as if either the message listener or input listener finishes, our application will exit. If we wanted to have more robust logic here, we could do this by checking the done and pending sets `wait` returns. For instance, if the input listener threw an exception we could cancel the message listener task.

If you first run the server, then run a couple of chat clients you'll be able to send and receive messages in the client like a normal chat application. For example, two users connecting to the chat may produce output like the following:

```
Welcome! 1 user(s) are online!
```

```
MissIslington connected!
SirBedevere connected!
SirBedevere: Is that your nose?
MissIslington: No, it's a false one!
```

We've now built a chat server and client that can handle multiple users connected simultaneously with only one thread. This application could stand to be more robust, for example, you may want to consider retrying message sends on failure or a protocol to acknowledge a client received a message. Making this a production-worthy application is rather complex and is outside the scope of this book, though it would be a fun exercise for the reader as there are a lot of failure points to think through. Using similar concepts to what we've explored in this example, you'll be able to create robust client and server applications to suit your needs.

8.7 Summary

In this chapter, we've learned how to use streams to build applications from HTTP clients to asynchronous command lines to client/server applications.

- We learned how to use the lower-level transport and protocol APIs to build a simple HTTP client. These APIs are the bedrock of the higher-level stream asyncio stream APIs and are generally not recommended for general use.
- We've learned how to use the `StreamReader` and `StreamWriter` classes to build network applications. These higher-level APIs are the recommended approach to work with streams in asyncio.
- We've learned how to use streams to create non-blocking command line applications that can remain responsive to user input while running tasks in the background.
- We've learned how to create servers using the `start_server` coroutine. This approach is the recommended way to create servers in asyncio, as opposed to using sockets directly.
- We've learned how to create responsive client and server applications using streams and servers. Using this knowledge, we can create network-based applications such as chat servers and clients.

9
Web applications

This chapter covers:

- **Creating web applications with aiohttp**
- **The asynchronous server gateway interface (ASGI)**
- **Creating ASGI web applications with Starlette**
- **Using Django's asynchronous views**

Web applications power most of the sites we use on the internet today. If you've worked as a developer for a company with an internet presence, you've likely worked on a web application at some point in your career. In the world of synchronous Python, this means you've used frameworks such as Flask, Bottle or the extremely popular Django. With the exception of more recent versions of Django, these web frameworks were not built to work with asyncio out of the box. As such, when our web applications do work that could be parallelized, such as querying a database or making calls to other APIs, we don't have options outside of multithreading or multiprocessing. This means that we'll need to explore new frameworks that are compatible with asyncio.

In this chapter, we'll learn about a few popular asyncio-ready web frameworks. We'll first see how to use a framework we've already dealt with, aiohttp, to build async RESTful APIs. We'll then learn about the asynchronous server gateway interface, or ASGI, which is the async replacement for WSGI (web server gateway interface) and is how many web applications run. Using ASGI with Starlette, we'll build a simple REST API with websocket support. We'll also take a look at how to use Django's asynchronous views. Performance of web applications is always a consideration when scaling, so we'll also take a look at performance numbers by benchmarking with a load testing tool.

9.1 Creating a REST API with Aiohttp

Previously we used Aiohttp as a HTTP client to make thousands of web requests to web applications concurrently. Aiohttp has not only support as a HTTP client, but also has functionality to create asyncio-ready web application servers as well.

9.1.1 What is REST?

REST is an abbreviation for *representational state transfer*. It is a widely used paradigm in modern web application development, especially in conjunction with single-page applications with frameworks like React and Vue. REST provides us with a stateless, structured way to design our web APIs independently of client-side technology. A REST API should be able to interoperate with any number of clients, from a mobile phone to a browser, and all that should need to change is the client-side presentation of the data.

The key concept in REST is a *resource*. A resource is typically anything that can be represented by a noun. For example, a customer, a product or an account can be RESTful resources. The resources we just listed reference a single customer or product. Resources can also be collections, for example "customers" or "products" that have singletons we can access by some unique identifier. Singletons may also have sub-resources. A customer could have a list of favorite products as an example. Let's take a look at a couple of REST APIs to get a better understanding:

```
customers
customers/{id}
customers/{id}/favorites
```

We have three REST API endpoints here. Our first endpoint, `customers`, references a collection of customers. As consumers of this API, we would expect this to return a list of customers (this may be paginated as this could be a potentially large set). Our second endpoint references a single customer and takes in an `id` as a parameter. If we uniquely identify customers with an integer id, calling `customers/1` would give us data for the customer with an id of 1. Our final endpoint is an example of a sub-entity. A customer could have a list of favorite products, making the list of favorites a sub-entity of a customer. Calling `customers/1/favorites` would return the list of favorites for the customer with id 1.

We'll design our REST APIs going forward to return JSON as this is typical, though we could choose any format that suits our need. REST APIs can sometimes support multiple data representations through content negotiation via HTTP headers.

While a proper look into all the details of REST is outside the scope of this book, the creator of REST's dissertation is a good place to learn about the concepts. It is available at https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

9.1.2 Aiohttp server basics

Let's first get started by creating a simple hello world style API with aiohttp. We'll start by creating a simple GET endpoint that will give us some basic data in JSON format about the time and date. We'll call our endpoint `/time` and will expect it to return the month, day and current time.

Aiohttp provides web server functionality in the `web` module. Once we import this, we can define endpoints (called routes in aiohttp) with a `RouteTableDef`. A `RouteTableDef` provides a decorator that lets us specify a request type (GET, POST, etc) and a string representing the endpoint name. We can then use the `RouteTableDef` decorator to decorate coroutines that will execute when we call that endpoint. Inside these decorated coroutines, we can perform whatever application logic we'd like and then return data back to the client.

Creating these endpoints by themselves does nothing however, we still need to start the web application to serve the routes. We do this by first creating an `Application` instance, adding the routes from our `RouteTableDef` and running the application.

Listing 9.1 the current time endpoint

```
from aiohttp import web
from datetime import datetime
from aiohttp.web_request import Request
from aiohttp.web_response import Response

routes = web.RouteTableDef()


@routes.get('/time') #A
async def time(request: Request) -> Response:
    today = datetime.today()

    result = {
        'month': today.month,
        'day': today.day,
        'time': str(today.time())
    }

    return web.json_response(result) #B


app = web.Application() #C
app.add_routes(routes)
web.run_app(app)
```

#A Create a time GET endpoint, when a client calls this endpoint the time coroutine will run.
#B Take the result dictionary and turn it into a JSON response.
#C Create the web application, register the routes and run the application.

In listing 9.1 we first create a time endpoint. `@routes.get('/time')` specifies that the decorated coroutine will execute when a client executes a HTTP GET request against the `/time` uri. In our `time` coroutine, we get the month day and time and store it in a dictionary. We then call `web.json_response`, this takes the dictionary and serializes it into JSON format. It also configures the HTTP response we send back. In particular, it sets the status code to 200 and the content type to `'application/json'`.

We then create the web application and start it. First, we create an `Application` instance and call `add_routes`. This registers all the decorators we created with the web application. We then call `run_app` which starts the web server. By default, this starts the web server on localhost port 8080.

When we run this, we'll be able to test this out by either going to `localhost:8080/time` in a web browser or using a command-line utility such as cURL or Wget. Let's test it out with cURL to take a look at the full response by running `curl -i localhost:8080/time`. You should see something like the following:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 51
Date: Mon, 23 Nov 2020 16:35:32 GMT
```

```
Server: Python/3.9 aiohttp/3.6.2

{"month": 11, "day": 23, "time": "11:35:32.033271"}
```

This shows we've successfully created our first endpoint with aiohttp! One thing you may have noticed from our code listing is that our `time` coroutine had a single parameter named `request`. While we didn't need to use it in this example, it will become important shortly. This data structure has information about the web request the client sent, such as the body, query parameters and so on. To get a glimpse of the headers in the request, add `print(request.headers)` somewhere inside the `time` coroutine, you should see something similar to this:

```
<CIMultiDictProxy('Host': 'localhost:8080', 'User-Agent': 'curl/7.64.1', 'Accept': '*/*')>
```

9.1.3 Connecting to a database and returning results

While our time endpoint shows us the basics, most web applications are not this simple. We'll usually need to connect to a database such as Postgres or Redis and may need to communicate with other REST APIs, for example, if we query or update a vendor API we use.

To see how to do this, we'll build a REST API around our ecommerce storefront database from chapter five. Specifically, we'll design a REST API to get existing products from our database as well as create new ones.

The first thing we'll need to do is create a connection to our database. Since we expect our application will have many concurrent users, using a connection pool instead of a single connection makes the most amount of sense. The question becomes where can we create and store the connection pool for easy use by our application's endpoints?

To answer the question of where we can store the connection pool, we'll need to answer the broader question of where we can store shared application data in aiohttp applications. We'll then use this mechanism to hold a reference to our connection pool.

To store shared data, aiohttp's `Application` class acts as a dictionary. For example, if we had some shared dictionary we wanted all our routes to have access to, we could store it in our application like follows:

```
app = web.Applicaition()
app['shared_dict'] = {'key' : 'value'}
```

We can now access the shared dictionary by executing `app['shared_dict']`. Now we need to figure out how to access the application from within a route. We could make the app instance global, but aiohttp provides a better way though the `Request` class. Every request that our route gets will have a reference to the application instance through the `app` field, allowing us easy access to any shared data. For example, getting the shared dictionary and returning it as a response might look like the following:

```
@routes.get('/')
async def get_data(request: Request) -> Response:
    shared_data = request.app['shared_dict']
    return web.json_response(shared_data)
```

We'll use this paradigm to store and retrieve our database connection pool once we create it. Next, where is the best place to create our connection pool? We can't easily do it when we create our application instance, as this happens outside of any coroutine meaning we can't use the needed `await` expressions.

Aiohttp provides a signal handler on the application instance to handle setup tasks like this called `on_startup`. You can think of this as a list of coroutines that will execute when we start the application. We can add coroutines to run on startup by calling `app.on_startup.append(coroutine)`. Each coroutine we append to `on_startup` has a single parameter, the `Application` instance. We can store our database pool in the application instance passed in to this coroutine once we've instantiated it.

We also need to consider what happens when our web application shuts down. We want to actively close and clean up database connections when we shut down, otherwise we could leave dangling connections, putting unneeded stress on our database. Aiohttp also provides a second signal handler called `on_cleanup`. The coroutines in this handler will run when our application closes, giving us an easy place to shut down the connection pool. This behaves like the `on_startup` handler in that we just call `append` with coroutines we'd like to run.

Putting all these pieces together, we can create a web application that creates a connection pool to our product database. To test this out, let's create an endpoint that gets all brand data in our database. This will be a GET endpoint called `/brands`:

Listing 9.2 connecting to a product database

```
import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from asyncpg import Record
from asyncpg.pool import Pool
from typing import List, Dict

routes = web.RouteTableDef()
DB_KEY = 'database'


async def create_database_pool(app: Application): #A
    print('Creating database pool.')
    pool: Pool = await asyncpg.create_pool(host='0.0.0.0',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
                                           max_size=6)
    app[DB_KEY] = pool


async def destroy_database_pool(app: Application): #B
    print('Destroying database pool.')
    pool: Pool = app[DB_KEY]
    await pool.close()


@routes.get('/brands')
async def brands(request: Request) -> Response: #C
```

```
    connection: Pool = request.app[DB_KEY]
    brand_query = 'SELECT brand_id, brand_name FROM brand'
    results: List[Record] = await connection.fetch(brand_query)
    result_as_dict: List[Dict] = [dict(brand) for brand in results]
    return web.json_response(result_as_dict)


app = web.Application()
app.on_startup.append(create_database_pool) #D
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)
```

#A Create the database pool and store it in the application instance.
#B Destroy the pool in the application instance.
#C Query all brands and return results to the client.
#D Add the create and destroy pool coroutines to startup and cleanup.

We first define two coroutines to create and destroy the connection pool. In `create_database_pool` we create a pool and store it in the application under the `DB_KEY`. Then, in `destroy_database_pool` we get the pool from the application instance and wait for it to close. When we start our application, we append these two coroutines to the `on_startup` and `on_cleanup` signal handlers respectively.

Next, we define our brands route. We first grab the database pool from the request and run a query to get all brands in our database. We then loop over each brand casting them to dictionaries. This is because aiohttp does not know how to serialize asyncpg `Record` instances.

When running this application, you should be able to go to `localhost:8080/brands` in a browser and see all brands in your database displayed as a JSON list, giving you something like the following:

```
[{"brand_id": 1, "brand_name": "his"}, {"brand_id": 2, "brand_name": "he"}, {"brand_id": 3,
       "brand_name": "at"}]
```

We've now created our first RESTful collection API endpoint. Next, let's see how to create endpoints to create and update singleton resources. We'll implement two endpoints; one GET endpoint to retrieve a product by a specific id and one POST endpoint to create a new product.

Let's start with our GET endpoint for a product. This endpoint will take in an integer id parameter, meaning to get the product with id one we'd call `/products/1`. How can we create a route that has a parameter in it? Aiohttp lets us parameterize our routes by wrapping any parameters in curly brackets, so our product route will be `/products/{id}`. When we parameterize like this, we'll see an entry in our request's `match_info` dictionary. In this case, whatever the user passed in to the id parameter will be available in `request.match_info['id']` as a string.

Since we could pass in an invalid string for an id, we'll need to add some error handling. A client could also ask for an id that does not exist, so we'll need to handle the not found case appropriately as well. For these error cases, we'll return a HTTP 400 status code to indicate the client issued a bad request. For the case where the product does not exist, we'll return a HTTP 404 status code. To represent these error cases, aiohttp provides a set of exceptions for each HTTP status code. In the error cases, we can just raise them, and the client will receive the appropriate status code.

Listing 9.3 getting a specific product

```
import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from asyncpg import Record
from asyncpg.pool import Pool

routes = web.RouteTableDef()
DB_KEY = 'database'


@routes.get('/products/{id}')
async def get_product(request: Request) -> Response:
    try:
        str_id = request.match_info['id'] #A
        product_id = int(str_id)

        query = \
            """
            SELECT
            product_id,
            product_name,
            brand_id
            FROM product
            WHERE product_id = $1
            """

        connection: Pool = request.app[DB_KEY]
        result: Record = await connection.fetchrow(query, product_id) #B

        if result is not None: #C
            return web.json_response(dict(result))
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()


async def create_database_pool(app: Application):
    print('Creating database pool.')
    pool: Pool = await asyncpg.create_pool(host='0.0.0.0',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
                                           max_size=6)
    app[DB_KEY] = pool


async def destroy_database_pool(app: Application):
    print('Destroying database pool.')
    pool: Pool = app[DB_KEY]
    await pool.close()
```

```
app = web.Application()
app.on_startup.append(create_database_pool)
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)
```

#A Get the product id parameter from the URL.
#B Run the query for a single product.
#C If we have a result convert it to json and send to the client, otherwise send a 404 not found.

Next, let’s see how to create a POST endpoint to create a new product in the database. We’ll send the data we want in the request body as a JSON string and we’ll then translate that into an insert query. We’ll need to do some error checking here to see if the JSON is valid, and if it isn’t, send the client a bad request error.

Listing 9.4 a create product endpoint

```
import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_09.listing_9_2 import create_database_pool, destroy_database_pool

routes = web.RouteTableDef()
DB_KEY = 'database'


@routes.post('/product')
async def create_product(request: Request) -> Response:
    PRODUCT_NAME = 'product_name'
    BRAND_ID = 'brand_id'

    if not request.can_read_body:
        raise web.HTTPBadRequest()

    body = await request.json()

    if PRODUCT_NAME in body and BRAND_ID in body:
        db = request.app[DB_KEY]
        await db.execute('''INSERT INTO product(product_id,
                                                product_name,
                                                brand_id)
                                                VALUES(DEFAULT, $1, $2)''',
                         body[PRODUCT_NAME],
                         int(body[BRAND_ID]))
        return web.Response(status=201)
    else:
        raise web.HTTPBadRequest()


app = web.Application()
app.on_startup.append(create_database_pool)
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)
```

We first check to see if we even have a body with `request.can_read_body` and if we don't, we quickly return a bad response. We then grab the request body as a dictionary with the `json` coroutine. Why is this a coroutine and not a plain method? If we have an especially large request body, the result may be buffered and could take some time to read. Instead of blocking our handler waiting for all data to come in, we `await` until all data is there. We then insert the record into the product table and return a HTTP 201 created status back to the client.

Using cURL, you should be able to execute something like the following to insert a product into your database, getting a HTTP 201 response.

```
curl -i -d '{"product_name":"product_name", "brand_id":1}' localhost:8080/product
HTTP/1.1 201 Created
Content-Length: 0
Content-Type: application/octet-stream
Date: Tue, 24 Nov 2020 13:27:44 GMT
Server: Python/3.9 aiohttp/3.6.2
```

While the error handling here should be more robust (what happens if the brand id is a string and not an integer or the JSON is malformed?) this illustrates how to process postdata to insert a record into our database.

9.1.4 Comparing Aiohttp with Flask

Working with aiohttp and an asyncio ready web framework gives us the benefit of using libraries such as asyncpg. Outside of the use of asyncio libraries, are there any benefits to using a framework like aiohttp as opposed to a similar synchronous framework such as Flask?

While it highly depends on server configuration, database hardware and other factors, asyncio-based applications can have better throughput with less resources. In a synchronous framework, each request handler runs from start to finish without interruption. In an asynchronous framework, when our `await` expressions suspend execution, they give the framework a chance to handle other work, resulting in more efficiency.

To test this out let's build a Flask replacement for our brands endpoint. We'll assume basic familiarity with Flask and synchronous database drivers, though even if you don't know these you should be able to follow the code. To get started we'll install Flask and Psycopg2, a synchronous Postgres driver, with the following commands:

```
pip install -Iv flask==1.1.2
pip install -Iv psycopg2==2.8.6
```

For Psycopg, you may run into compile errors on install, if you do, you may need to install Postgres tools, open SSL or another library. A google search with your error should yield the answer. Now let's implement our endpoint. We'll first create a connection to the database. Then in our request handler we'll reuse the brand query from our previous example and return the results as a JSON array.

Listing 9.5 a flask application to retrieve brands

```
from flask import Flask, jsonify
import psycopg2

app = Flask(__name__)
```

```
conn_info = "dbname=products user=postgres password=password host=0.0.0.0"
db = psycopg2.connect(conn_info)


@app.route('/brands')
def hello_world():
    cur = db.cursor()
    cur.execute('SELECT brand_id, brand_name FROM brand')
    rows = cur.fetchall()
    cur.close()
    return jsonify([{'brand_id': row[0], 'brand_name': row[1]} for row in rows])
```

Now we need to run our application. Flask comes with a development server, but it is not production ready and wouldn't be a fair comparison, especially since would only run one process, meaning we could only handle one request at a time. We'll need to use a production WSGI server to test this. We'll use Gunicorn for this example, though there are many you could choose. Let's start by installing Gunicorn with the following command:

```
pip install -Iv gunicorn==20.0.4
```

We'll be testing this out on an 8-core machine, so we'll spawn 8 workers with Gunicorn. Running `gunicorn  -w 8  chapter_09.listing_9_5:app` you should see 8 workers start up.

```
[2020-11-24 09:53:39 -0500] [16454] [INFO] Starting gunicorn 20.0.4
[2020-11-24 09:53:39 -0500] [16454] [INFO] Listening at: http://127.0.0.1:8000 (16454)
[2020-11-24 09:53:39 -0500] [16454] [INFO] Using worker: sync
[2020-11-24 09:53:39 -0500] [16458] [INFO] Booting worker with pid: 16458
[2020-11-24 09:53:39 -0500] [16459] [INFO] Booting worker with pid: 16459
[2020-11-24 09:53:39 -0500] [16460] [INFO] Booting worker with pid: 16460
[2020-11-24 09:53:39 -0500] [16461] [INFO] Booting worker with pid: 16461
[2020-11-24 09:53:40 -0500] [16463] [INFO] Booting worker with pid: 16463
[2020-11-24 09:53:40 -0500] [16464] [INFO] Booting worker with pid: 16464
[2020-11-24 09:53:40 -0500] [16465] [INFO] Booting worker with pid: 16465
[2020-11-24 09:53:40 -0500] [16468] [INFO] Booting worker with pid: 16468
```

This means we have created 8 connections to our database and can serve 8 requests concurrently. Now we need a tool to benchmark performance between Flask and Aiohttp. A command-line load tester will work for a quick test, while this won't be the most accurate picture, it will give us a directional idea of performance. We'll use a load tester called wrk, though any load tester such as Apache Bench or Hey will work. You can view installation instructions on wrk at https://github.com/wg/wrk/wiki/Installing-wrk-on-Linux.

Let's start by running a 30 second load test on our Flask server. We'll use 1 thread and 200 connections, simulating 200 concurrent users hitting out app as fast as they can. On an 8-core 2.4 ghz machine you could see results similar to the following:

```
Running 30s test @ http://localhost:8000/brands
  1 threads and 200 connections
  16534 requests in 30.02s, 61.32MB read
  Socket errors: connect 0, read 1533, write 276, timeout 0
Requests/sec:    550.82
Transfer/sec:    2.04MB
```

We served about 550 requests per second, not a bad result. Let's rerun the same with aiohttp and compare the results:

```
Running 30s test @ http://localhost:8080/brands
  1 threads and 200 connections
  46774 requests in 30.01s, 191.45MB read
Requests/sec:   1558.46
Transfer/sec:   6.38MB
```

Using aiohttp, we were able to serve over 1500 requests per second, about 3 times what we were able to do with Flask. More importantly, we did this with only one process where Flask needed a total of *8 processes* to handle one third of the requests! You could further improve the performance of aiohttp by putting nginx in front of it and starting more worker processes.

We now know the basics of how to use aiohttp to build a database-backed web application. In the world of web applications, aiohttp is a little different than most in that it is a web server itself and it does not conform to WSGI and can stand alone on its own. As we saw with Flask, this is not usually the case. Next, let's understand how ASGI works and see how to use it with an ASGI compliant framework called Starlette.

9.2 The asynchronous server gateway interface

When we used Flask in the previous example, we used the Gunicorn WSGI server to serve our application. WSGI is a standardized way to forward web requests to a web framework, such as Flask or Django. While there are many WSGI servers out there, they were not designed to support asynchronous workloads as the WSGI specification long predates asyncio. As asynchronous web applications become more widely used, a way to abstract frameworks from their servers proved necessary. Thus, the asynchronous server gateway interface, or ASGI was created. ASGI is a relative newcomer to the space, but already has several popular implementations and frameworks that support it, including Django.

9.2.1 How does ASGI compare to WSGI?

WSGI was born out of a fractured landscape of web application frameworks. Prior to WSGI, the choice of one framework could limit the kinds of usable interface web servers as there was no standardized interface between the two. WSGI addressed this by providing a simple API for web servers to talk to Python frameworks. WSGI was finally given formal acceptance into the Python ecosystem in 2004 with the acceptance of PEP-333 (https://www.python.org/dev/peps/pep-0333/) and is now the de facto standard for web application deployment.

When it comes to asynchronous workloads however, WSGI does not work. The heart of the WSGI specification is a simple Python function. For example, let's see the simplest WSGI application we can build.

Listing 9.6 a WSGI application

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return [b"WSGI hello!"]
```

We can run this application using Gunicorn by running `gunicorn chapter_09.listing_9_6` and test it out with `curl http://127.0.0.1:8000`. As you can see, there isn't any place for us to use an await. In addition, WSGI only supports response/request lifecycles, meaning it won't work with long lived connection protocols, such as web sockets. ASGI fixes this by redesigning the API to use coroutines. Let's translate our WSGI example to ASGI.

Listing 9.7 a simple ASGI application

```
async def application(scope, receive, send):
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [[b'content-type', b'text/html']]
    })
    await send({'type': 'http.response.body', 'body': b'ASGI hello!'})
```

An ASGI application function has three parameters, a scope dictionary, a receive coroutine and a send coroutine that let us send and receive data respectively. In our example, we send the start of the http response, followed by the body.

Now how do we serve the above application? There are a few implementations of ASGI available, but we'll use a popular one called Uvicorn (https://www.uvicorn.org/). Uvicorn is built on top of uvloop and httptools, which are fast C implementations of the asyncio event loop (we're actually not tied to the event loop that comes with asyncio, we'll learn more in chapter 14) and HTTP parsing. We can install Uvicorn by running the following:

```
pip install -Iv uvicorn==0.12.3
```

Now we can run our application with the following command:

```
uvicorn chapter_09.listing_9_7:application
```

And we should see our hello message printed if we go to http://localhost:8000. While we used uvicorn directly here to test things out, it is better practice to use uvicorn with Gunicorn as Gunicorn will have logic to restart workers on crashes for us. We'll see how to do this with Django in section 9.4.

We should keep in mind that while WSGI is an accepted PEP, ASGI is not yet and as of writing is still fairly new. Expect the details of how ASGI works to evolve and change as the asyncio landscape changes.

Now we know the basics of ASGI and how it compares to WSGI. What we have learned is very low-level though, we want a framework to handle ASGI for us! There are a few ASGI compliant frameworks, let's take a look at a popular one.

9.3 ASGI with Starlette

Starlette is a small ASGI-compliant framework created by Encode, the creators of uvicorn and other popular libraries such as Django rest framework. It offers fairly impressive performance (at the time of writing), WebSocket support and more. You can view its documentation at https://www.starlette.io/. Let's take a look at how to implement simple REST and web socket endpoints using it. To get started let's first install it with the following command:

```
pip install -Iv starlette==0.14.1
```

9.3.1 A REST endpoint with Starlette

Let's start to learn Starlette by reimplementing our brands endpoint from previous sections. We'll create our application by creating an instance of the `Starlette` class. This class takes a few parameters that we'll be interested in using: a list of `Route` objects and a list of coroutines to run on startup and shutdown. `Route` objects are mappings from a string path, brands in our case, to a coroutine or another callable object. Much like aiohttp, these coroutines have one parameter representing the request and they return a response, so our route handle will look very similar to our aiohttp version. What is slightly different is how we handle sharing our database pool, we still store it on our Starlette application instance, but it is inside a state object instead.

Listing 9.8 A Starlette brands endpoint

```
import asyncpg
from asyncpg import Record
from asyncpg.pool import Pool
from starlette.applications import Starlette
from starlette.requests import Request
from starlette.responses import JSONResponse, Response
from starlette.routing import Route
from typing import List, Dict


async def create_database_pool():
    pool: Pool = await asyncpg.create_pool(host='0.0.0.0',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
                                           max_size=6)
    app.state.DB = pool


async def destroy_database_pool():
    pool = app.state.DB
    await pool.close()


async def brands(request: Request) -> Response:
    connection: Pool = request.app.state.DB
    brand_query = 'SELECT brand_id, brand_name FROM brand'
    results: List[Record] = await connection.fetch(brand_query)
    result_as_dict: List[Dict] = [dict(brand) for brand in results]
    return JSONResponse(result_as_dict)


app = Starlette(routes=[Route('/brands', brands)],
                on_startup=[create_database_pool],
                on_shutdown=[destroy_database_pool])
```

Now that we have our brands endpoint, let's use Uvicorn to start it up. We'll start up 8 workers like we did before with the following command:

```
uvicorn --workers 8 --log-level error chapter_09.listing_9_8:app
```

You should be able to hit this endpoint at `localhost:8000/brands` and see the contents of the brand table as before. Now that we have our application running let's run a quick benchmark to see how it compares to aiohttp and Flask. We'll use the same wrk command as before with 200 connections over 30 seconds:

```
Running 30s test @ http://localhost:8000/brands
  1 threads and 200 connections
Requests/sec:   4365.37
Transfer/sec:   16.07MB
```

We've served north of 4,000 requests per second, outperforming Flask and even Aiohttp by a wide margin! Since we only ran one Aiohttp worker process earler, this isn't exactly a fair comparison (we'd get similar numbers with 8 Aiohttp workers behind Nginx), but goes to show the throughput power that async frameworks offer.

9.3.2 Web sockets with Starlette

In a traditional HTTP request, the client sends a request to the server, the server hands a back a response and that is the end of the transaction. What if we want to build a web page that updates without a user having to refresh? For example, we may have a live counter of how many users are currently on the site. We can do over HTTP with some Javascript which polls an endpoint that tells us how many users are on the site. We could hit the endpoint every few seconds, updating the page with the latest result. While this will work, it has its drawbacks. The main drawback is we're creating a bunch of extra load on our web server, each request and response cycle takes time and resources. This is especially egregious because our user count might not change in between requests, causing strain on our system for no net new information (we could mitigate this with caching, but the point still stands, and caching introduces other complexity and overhead). HTTP polling is the digital equivalent of a child in the backseat of the car asking if you are there yet repeatedly.

Web sockets provide an alternative to HTTP polling. Instead of a request/response cycle like HTTP we establish one persistent socket. Then, we just send data freely across that socket. This socket is bidirectional, meaning we can both send data to and receive data from our server without having to go through a HTTP request lifecycle every time. To apply this to the example of displaying an up-to-date user count, once we connect to a web socket the server can just *tell* us when there is a new user count. We don't need to ask repeatedly, creating extra load and potentially receiving data that isn't new.

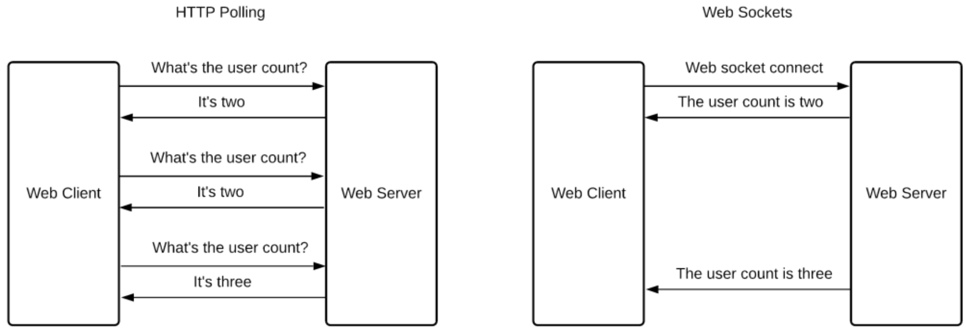


Figure 9.1 HTTP polling to retrieve data compared to web sockets

Starlette provides out of the box support for web sockets using an easy-to-understand interface. To see this in action, we'll build a simple web socket endpoint that will tell us how many users are connected to a websocket endpoint simultaneously. To get started we'll first need to install websocket support:

```
pip install -Iv websockets==8.1
```

Next, we'll need to implement our web socket endpoint. Our game plan will be to keep an in-memory list of all connected client web sockets. When a new client connects, we'll add them to the list and send the new count of users back to all clients in the list. When a client disconnects, we'll remove them from the list and update other clients about the change in user count as well. We'll also add some basic error handling. If sending one of these messages results in an exception, we'll remove the client from the list.

In Starlette, we can subclass `WebSocketEndpoint` to create an endpoint to handle a web socket connection. This class has a few coroutines we'll need to implement. The first is `on_connect`, this gets fired when a client connects to our socket. In `on_connect`, we'll store the client's web socket in a list and send the length of the list to all other sockets. The second coroutine is `on_receive`, this gets fired when the client connection sends a message to the server. In our case, we won't need to implement this as we don't expect the client to send us any data. The final coroutine is `on_disconnect` which runs when a client disconnects. In this case, we'll remove the client from the list of connected web sockets, and update other connected clients that with the latest user count.

Listing 9.9 a Starlette web socket endpoint

```
import asyncio
from starlette.applications import Starlette
from starlette.endpoints import WebSocketEndpoint
from starlette.routing import WebSocketRoute


class UserCounter(WebSocketEndpoint):
    encoding = 'text'
```

```
    sockets = []

    async def on_connect(self, websocket):  # A
        await websocket.accept()
        UserCounter.sockets.append(websocket)
        await self._send_count()

    async def on_disconnect(self, websocket, close_code):  # B
        UserCounter.sockets.remove(websocket)
        await self._send_count()

    async def on_receive(self, websocket, data):
        pass

    async def _send_count(self):  # C
        if len(UserCounter.sockets) > 0:
            count_str = str(len(UserCounter.sockets))
            task_to_socket = {asyncio.create_task(websocket.send_text(count_str)): websocket
                              for websocket
                              in UserCounter.sockets}

            done, pending = await asyncio.wait(task_to_socket)

            for task in done:
                if task.exception() is not None:
                    if task_to_socket[task] in UserCounter.sockets:
                        UserCounter.sockets.remove(task_to_socket[task])


app = Starlette(routes=[WebSocketRoute('/counter', UserCounter)])
```

#A When a client connects, add it to the list of sockets and notify other users of the new count.
#B When a client disconnects, remove it from the list of sockets and notify other users of the new count.
#C Notify other users how many users are connected. If there is an exception while sending, remove them from the list.

Now we'll need to define a page to interact with our web socket. We'll add create a basic script to connect to our web socket endpoint. When we receive a message, we'll update a counter on the page with the latest value.

Listing 9.10 using the websocket endpoint

```
<!DOCTYPE html>
<html lang="">
<head>
    <title>Starlette Web Sockets</title>
    <script>
        document.addEventListener("DOMContentLoaded", () => {
            let socket = new WebSocket("ws://localhost:8000/counter");

            socket.onmessage = (event) => {
                const counter = document.querySelector("#counter");
                counter.textContent = event.data;
            };
        });
    </script>
</head>
<body>
    <span>Users online: </span>
```

```
    <span id="counter"></span>
</body>
</html>
```

In listing 9.10, our script is where most of the work happens. We first connect to our endpoint and then define an `onmessage` callback. When the server sends us data, this callback runs. In this callback, we grab a special element from the DOM and set its content to the data we receive. Note that in our script we don’t execute this code until after the `DOMContentLoaded` event, without this our counter element may not exist when the script executes.

If you start the server with `uvicorn --workers 1 chapter_09.listing_9_9:app` and open the web page you should see the 1 displayed on the page. If you open the page multiple times in separate tabs, you should see the count on all of the tabs increment. When you close a tab, you should see the count decrement across all other open tabs. Note that we only use one worker here as we have shared state (the `socket` list) in memory, if we use multiple workers each worker will have its own `socket` list. To deploy properly, you’ll need some persistent store such as a database.

We can now use both aiohttp and starlette to create asyncio based web applications for both REST end websocket endpoints. While these frameworks are popular, they are not close in popularity as the 1000-pound gorilla of Python web frameworks, Django.

9.4 Django Asynchronous Views

Django is perhaps the most widely used Python framework. It has a wealth of functionality right out of the box, from an ORM to handle databases to a customizable admin console. Until version 3.0, Django applications only supported deploying as a WSGI application and had little support for asyncio outside of the channels library. Version 3.0 introduced support for ASGI and began going down the road of making Django fully async. More recently, version 3.1 gained support for asynchronous views, allowing you to use asyncio libraries directly in your Django views. At the time of writing, async support for Django is new, and certain the overall feature set is still lacking (for example, the ORM is entirely synchronous, but supporting async is in the works). Expect support for this to grow and evolve as Django becomes more async-aware.

Let’s learn how to use async views by building a small application that uses aiohttp in a view. Imagine we’re integrating with an external REST API, and we want to build a utility to run a few requests concurrently to see response times, body length and how many failures (exceptions) we got. We’ll build a view that takes in a url and request count as query parameters, calls out to this url and aggregates the results, returning them in a nice table format.

Let’s get started by ensuring we have the appropriate version of Django installed:

```
pip install -Iv django==3.1.3
```

Now let’s use the Django admin tool to create the skeleton for our application. We’ll call our project async_views:

```
django-admin startproject async_views
```

Once you run this command, you should see a directory named `async_views` created with the following structure:

```
async_views/
    manage.py
    async_views/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Note that we have both a `wsgi.py` and an `asgi.py` file, showing we can deploy to both types of gateway interfaces. You should now be able to use uvicorn to serve the basic Django hello world page. Run the following command from the top-level async_views directory:

```
gunicorn async_views.asgi:application -k uvicorn.workers.UvicornWorker
```

Then, when you go to localhost:8000 you should see the Django welcome page.

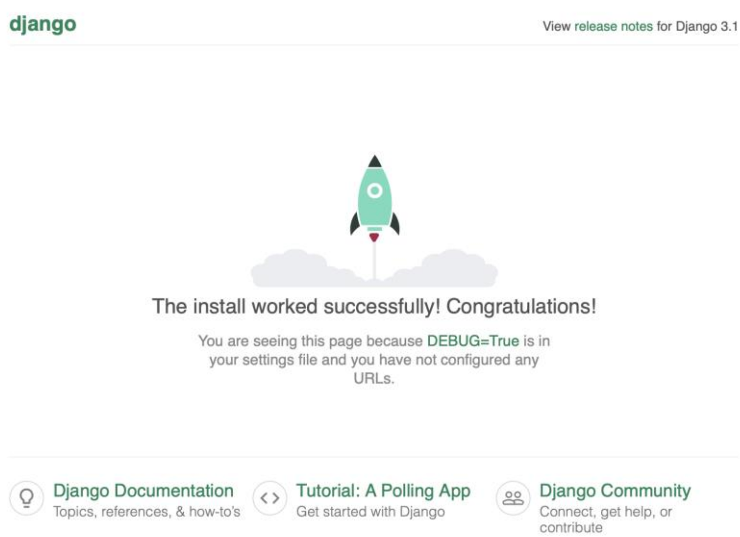


Figure 9.2 the Django welcome page

Now we'll need to create our app, we'll call it `async_api`. Within the `async_views` directory, run `python manage.py startapp async_api`. This will build model, view and other files for the `async_api` app.

Now we have everything we need to create our first asynchronous view. Within the `async_api` directory there should be a `views.py` file. Inside of this, we can specify a view as asynchronous by simply declaring it as a coroutine. In this file, we'll add an async view to make a bunch of HTTP requests concurrently and display their status codes and other data in a HTML table.

Listing 9.11 a Django asynchronous view

```
import asyncio
from datetime import datetime
from aiohttp import ClientSession
from django.shortcuts import render
import aiohttp


async def get_url_details(session: ClientSession, url: str):
    start_time = datetime.now()
    response = await session.get(url)
    response_body = await response.text()
    end_time = datetime.now()
    return {'status': response.status,
            'time': (end_time - start_time).microseconds,
            'body_length': len(response_body)}


async def make_requests(url: str, request_num: int):
    async with aiohttp.ClientSession() as session:
        requests = [get_url_details(session, url) for _ in range(request_num)]
        results = await asyncio.gather(*requests, return_exceptions=True)
        failed_results = [str(result) for result in results if isinstance(result, Exception)]
        successful_results = [result for result in results if not isinstance(result, Exception)]
        return {'failed_results': failed_results, 'successful_results': successful_results}


async def requests_view(request):
    url: str = request.GET['url']
    request_num: int = int(request.GET['request_num'])
    context = await make_requests(url, request_num)
    return render(request, 'async_api/requests.html', context)
```

In listing 9.11, we first create a coroutine to make a request and return a dictionary of the response status, total time of request and the length of the response body. Next, we define an async view coroutine named `requests_view`. This view gets the url and request count from the query parameters and then makes requests via `get_url_details` concurrently with `gather`. Finally, we filter out the successful responses from any failures and put the results in a context dictionary that we then pass to `render` to build the response. Note that we haven't built our template for the response yet and are now just passing in `async_views/requests.html` for now. Next, let's build the template so we can view the results.

First, we'll need to create a `templates` directory under the `async_api` directory, then within the templates directory we'll need to create an `async_api` folder. Once we have this directory structure in place, we can add a view inside `async_api/templates/async_api`. We'll call this view `requests.html` and we'll loop over the context dictionary from our view, putting the results in table format.

Listing 9.12 the requests view

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Request Summary</title>
</head>
<body>
<h1>Summary of requests:</h1>
<h2>Failures:</h2>
<table>
    {% for failure in failed_results %}
    <tr>
        <td>{{failure}}</td>
    </tr>
    {% endfor %}
</table>
<h2>Successful Results:</h2>
<table>
    <tr>
        <td>Status code</td>
        <td>Response time (microseconds)</td>
        <td>Response size</td>
    </tr>
    {% for result in successful_results %}
    <tr>
        <td>{{result.status}}</td>
        <td>{{result.time}}</td>
        <td>{{result.body_length}}</td>
    </tr>
    {% endfor %}
</table>
</body>
</html>
```

In our view, we create two tables, one to display any exceptions we encountered, and a second to display the successful results we were able to get. While this won't be the prettiest webpage ever created, it will have all the relevant information we want.

Next, we'll need to hook our template and view up to a URL so it will run when we hit it in a browser. In the `async_api` folder, create a url.py file with the following:

Listing 9.13 the async_api/url.py file

```
from django.urls import path
from . import views

app_name = 'async_api'

urlpatterns = [
    path('', views.requests_view, name='requests'),
]
```

Now we'll need to include the `async_api` app's urls within our Django application. Within the `async_views/async_views` directory you should already have a `urls.py` file. Inside of this file, you'll need to modify the `urlpatterns` list to reference `async_api`, once done it should look like follows:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('requests/', include('async_api.urls'))
]
```

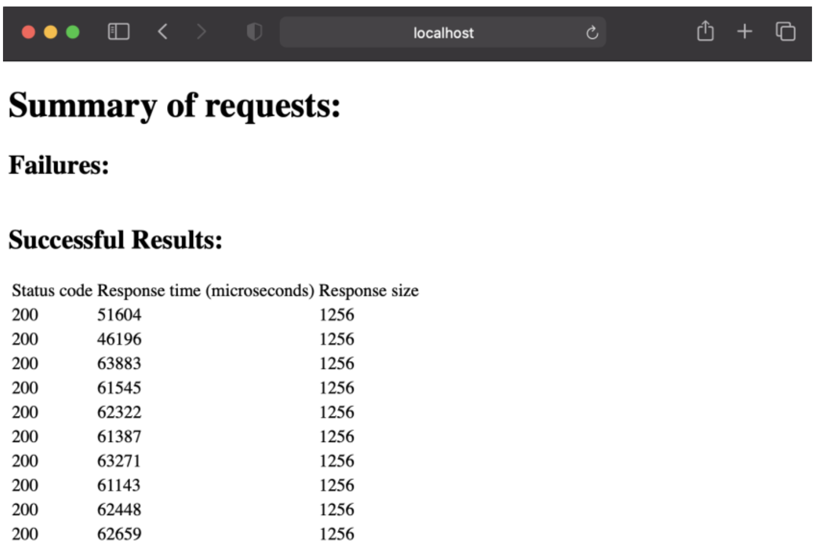
Finally, we'll need to add the async_views application to the installed apps. In `async_views/async_views/settings.py` modify the `INSTALLED_APPS` list to include `async_api`, once done it should look like this:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'async_api'
]
```

Now we finally have everything we need to run our application. You can start the app with the same gunicorn command we used when we first created the Django app. Now you can go to our endpoint and make requests. For example, to hit example.com 10 times concurrently and get the results go to:

```
http://localhost:8000/requests/?url=http://example.com&request_num=10
```

While numbers will differ on your machine, you should see a page like the following displayed:



Summary of requests:

Failures:

Successful Results:

Status code	Response time (microseconds)	Response size
200	51604	1256
200	46196	1256
200	63883	1256
200	61545	1256
200	62322	1256
200	61387	1256
200	63271	1256
200	61143	1256
200	62448	1256
200	62659	1256

Figure 9.3 the requests asynchronous view

We've now built a Django view that is capable of making an arbitrary amount of HTTP requests concurrently by hosting it with ASGI, but what if you're in a situation where ASGI isn't an option? Perhaps you're working with an older application that relies on it, can you still host an async view? We can try this out by running our application under gunicorn with the WSGI application from `wsgi.py` with the synchronous worker with the following command:

```
gunicorn async_views.wsgi:application
```

You should still be able to hit the requests endpoint and everything will work fine. So how does this work? When we run as a WSGI application, a fresh event loop is created each time we hit an asynchronous view. We can prove this to ourselves by adding a couple of lines of code somewhere in our view:

```
loop = asyncio.get_running_loop()
print(id(loop))
```

The `id` function will return an integer that is guaranteed to be unique over the lifetime of an object. When running as a WSGI application, each time you hit the requests endpoint, this will print a distinct integer, indicating that we create a fresh event loop on a per request basis. Keep the same code when running as an ASGI application and you'll see the same integer printed every time, since ASGI will only have one event loop for the entire application.

This means we can get the benefits of async views and running things concurrently even when running as a WSGI application. However, anything that needs an event loop to live across multiple requests won't work unless you deploy as an ASGI application.

9.4.1 Running blocking work in an asynchronous view

A natural question you may have is what about blocking work in an async view? We're still in a world where many libraries are synchronous, but this is incompatible with a single-threaded concurrency model. The ASGI specification has a function to deal with these situations named `sync_to_async`.

In chapter seven, we saw that we could run synchronous APIs in thread pool executors and get back awaitables we could use with asyncio. The `sync_to_async` function essentially does that, with a few noteworthy caveats.

The first caveat is that `sync_to_async` has a notion of thread sensitivity. In many contexts, synchronous APIs with shared state weren't designed to be called from multiple threads and doing so could cause race conditions or the new thread seeing an inconsistent state of the world. To deal with this, `sync_to_async` defaults to a 'thread sensitive' mode (specifically, this function has a `thread_sensitive` flag that defaults to `True`). This makes any sync code we pass in run in Django's main thread. This means that any blocking we do here will block the entire Django application (well, at least one WSGI/ASGI worker if we're running multiple), making us lose some benefits of an async stack by doing this.

If we're in a situation where thread sensitivity isn't an issue (when there is no shared state or the shared state does not rely on being in a specific thread), we can change `thread_sensitive` to `False`. This will make things run in a new thread per each call, giving us something that won't block Django's main thread and preserving more benefits of an asynchronous stack.

To see this in action, let's make a new view to test out the variations of sync_to_async. We'll create a function that uses time.sleep to put a thread to sleep, and we'll pass that in to sync_to_async. We'll add a query parameter to our endpoint so we can easily switch between thread sensitivity modes to see the impact.

First, add the following definition to `async_views/async_api/views.py`:

Listing 9.14 the sync to async view

```
from functools import partial
from django.http import HttpResponse
from asgiref.sync import sync_to_async


def sleep(seconds: int):
    import time
    time.sleep(seconds)


async def sync_to_async_view(request):
    sleep_time: int = int(request.GET['sleep_time'])
    num_calls: int = int(request.GET['num_calls'])
    thread_sensitive: bool = request.GET['thread_sensitive'] == 'True'
    function = sync_to_async(partial(sleep, sleep_time), thread_sensitive=thread_sensitive)
    await asyncio.gather(*[function() for _ in range(num_calls)])
    return HttpResponse('')
```

Next, add the following to `async_views/async_api/urls.py` to the `urlpatterns` list to wire up the view:

```
path('sync_to_async', views.sync_to_async_view)
```

Now you'll be able to hit the endpoint. To test this out, let's sleep for five seconds five times in thread-insensitive mode with the following url:

```
http://127.0.0.1:8000/requests/sync_to_async?sleep_time=5&num_calls=5&thread_sensitive=False
```

You'll notice that this only takes five seconds to complete since we're running multiple threads. You'll also notice if you hit this URL more than once each request still takes only five seconds, indicating the requests aren't blocking each other. Now, let's change the `thread_sensitive` url parameter to `True`, you'll see quite different behavior. First, the view will take 25 seconds to return since it is making five five-second calls sequentially. Second, if you hit the url multiple times each will block until the other completed, since we're blocking Django's main thread.

The `sync_to_async` function offers us a lot of options to use existing code with async views, however you need to be aware of the thread-sensitivity of what you're running and also aware of the limitations that this can place on async performance benefits.

9.4.2 Using async code in synchronous views

The next logical question is the question we just answered, but in reverse. What if I have a synchronous view but I want to use an asyncio library? The ASGI specification also has a special function named `async_to_sync`. This function accepts a coroutine and runs it in an event loop, returning the results in a synchronous fashion. If there is not an event loop (as is the case in a

WSGI application) a new one will be created for us on each request, otherwise this will run in the current event loop (as is the case when we run as an ASGI application).

To try this out, let's create a new version of our requests endpoint as a synchronous view, but still using our async request function.

Listing 9.15 calling async code in a synchronous view

```
from asgiref.sync import async_to_sync


def requests_view_sync(request):
    url: str = request.GET['url']
    request_num: int = int(request.GET['request_num'])
    context = async_to_sync(partial(make_requests, url, request_num))()
    return render(request, 'async_api/requests.html', context)
```

Next, add the following to the `urlpatterns` list in `urls.py`:

```
path('async_to_sync', views.requests_view_sync)
```

Then, you'll be able to hit the following url and see the same results as we saw with our first async view:

```
http://localhost:8000/requests/async_to_sync?url=http://example.com&request_num=10
```

Even in a synchronous WSGI world, sync_to_async lets us get some of the performance benefits of an asynchronous stack without being fully asynchronous.

9.5 Summary

- We've learned how to create basic RESTful APIs that hook up to a database with aiohttp and asyncpg.
- We've learned how to create ASGI compliant web applications with Starlette.
- We've learned how to use websockets with Starlette to build web applications with up-to-date information without HTTP polling.
- We've learned how to use asynchronous views with Django and also learned how to use async code in synchronous views and vice versa.

10 Microservices

This chapter covers:

- **The basics of microservices**
- **The backend for frontend pattern**
- **Using asyncio to handle microservice communication**
- **Using asyncio to handle failures and retries**

Many web applications are structured as monoliths. What do we mean when we say a monolith? A monolith generally refers to a medium to large sized application containing multiple modules that is independently deployed and managed as one unit. While there is nothing inherently wrong with this model (monoliths are perfectly fine, and even preferable for a majority of web applications as they are generally simpler), it does have its drawbacks.

As an example, if you make a small change to a monolithic application, you need to deploy the entire application, even parts that may be unaffected by your change. For instance, a monolithic e-commerce application may have order management and product listing endpoints in one application, meaning a tweak to a product endpoint would require a redeploy of order management code. A microservice architecture can help with such pain points. We could create separate services for orders and products, then a change in one service wouldn't affect the other.

In this chapter, we'll learn a bit more about a microservices and the motivations behind them. We'll learn a pattern called backend for frontend and apply this to an ecommerce microservice architecture. We'll then implement this API with aiohttp and asyncpg, learning how use concurrency to help us improve the performance of our application. We'll also learn how to properly deal with failure and retries with the circuit breaker pattern to build a more robust application.

10.1 Why microservices?

First, let's take an attempt at defining what microservices are. This is a rather tricky question, as there is no standardized definition and you'll probably get different answers depending on who you ask. Generally speaking, microservices follow a few guiding principles:

1. They are loosely coupled and independently deployable.
2. They have their own independent stack, including a data model.
3. They communicate with one another over a protocol such as REST or gRPC.
4. They follow the single responsibility principle, a microservice should "do one thing and do it well".

Let's apply these principles to a concrete example of an ecommerce storefront. An application like this has users that provide shipping and payment information to us who then buy our products. In a monolithic architecture, we'd have one application with one database to manage both user's data account data, such as their orders and shipping information, and our available products. In a microservice architecture, we would have multiple services each with their own database for separate concerns. We might have a product API with its own database which only handles data around products. We might have a user API with its own database which handles user account information, and so on.

Why would we choose this architectural style over monoliths? Monoliths are perfectly fine for a majority of applications; they are simpler to reason about and manage. Make a code change, run all the test suites to make sure your seemingly small change does not affect other areas of the system. Once you've run test, deploy the application as one unit. Is your application not performing well under load? Easy, deploy more instances of your application to handle the additional users.

While managing a monolith is operationally simpler, this simplicity has drawbacks that may matter depending on which tradeoffs you want to make:

COMPLEXITY OF CODE

As the application grows and acquires new features its complexity grows. Data models may become more coupled, causing unforeseen and hard to understand dependencies. Technical debt gets larger and larger making development slower and more complicated. While this is true of any growing system, a large codebase with multiple concerns can exacerbate this.

SCALABILITY

In a monolithic architecture, if you need to scale you need to add more instances of your *entire* application which can lead to cost inefficiencies. In the context of an ecommerce application, you will typically get much less orders than people just browsing products. In a monolithic architecture, to scale up to handle more people viewing your products, you'll need to scale up your order capabilities as well. In a microservice architecture, you can just scale the product service and leave the order service untouched if it has no issues.

TEAM AND STACK INDEPENDENCE

As a development team grows in size, new challenges emerge. Imagine you have five teams working on the same monolithic codebase, each team committing code several times per day.

Merge conflicts will become an increasing issue that everyone needs to handle, as will coordinating deploys across teams. With independent, loosely coupled microservices, this becomes less of an issue. If a team owns a service, they can work on it and deploy it mostly independently. This also allows for teams to use different tech stacks if desired, one service can be in Java and one in Python.

10.1.1 How can asyncio help?

Microservices generally need to communicate with one another over a protocol such as REST or gRPC. Since we may be talking to multiple microservices at the same time, this opens up the possibility to run requests concurrently, creating an efficiency that we otherwise wouldn't have in a synchronous application.

In addition to the resource efficiency benefits we get from an async stack, we also get the error handling benefits of the asyncio APIs such as `wait` and `gather`. If a particular group of requests takes too long, or a portion of that group has an exception, we can handle them gracefully.

Now that we understand the basic motivations behind microservices, let's learn one common microservice architecture pattern and see how to implement it.

10.2 Introducing the backend for frontend pattern

When we're building UIs in a microservice architecture, we'll typically need to get data from multiple services in order to create a particular UI view. For example, if we're building a user order history UI, we'll probably have to get the user's order history from an order service and merge that with product data from a product service. Depending on requirements, we may need data from other services as well.

This poses a few challenges for our frontend clients. The first is a user experience issue. With stand-alone services our UI clients will have to make one call to each service over the internet. This poses issues with latency and time to load the UI. We can't assume all of our users will have a good internet connection or fast computer, some may be on a mobile phone in a poor reception area, some may be on older computers, some may be in developing countries without access to high-speed internet at all. If we make five slow requests to five services, it has the potential to cause more issues than making one slow request.

In addition to network latency challenges, we also have challenges related to good software design principles. Imagine we have both web-based UIs and iOS and Android mobile UIs. If we directly call each service and merge the resulting responses, we need to replicate the logic to do so across three different clients, which is pretty redundant and puts us at risk of having inconsistent logic across clients.

While there are many microservice design patterns, one that can help us address the above issues is the backend for frontend pattern. In this design pattern, instead of our UIs directly communicating with our services we create a new service that makes these calls and aggregates the responses. This addresses our issues, instead of making multiple requests, we can just make one, cutting down on our round trips across the internet. We can also embed any logic related to failovers or retries inside of this service, saving our clients the work of having to repeat the same logic and introducing one place for us to update the logic when we need to change it. This also opens up the possibility of multiple backend for frontend services per different types of clients. The services we need to communicate with may need to vary depending on if we're a mobile client versus a web-based UI.

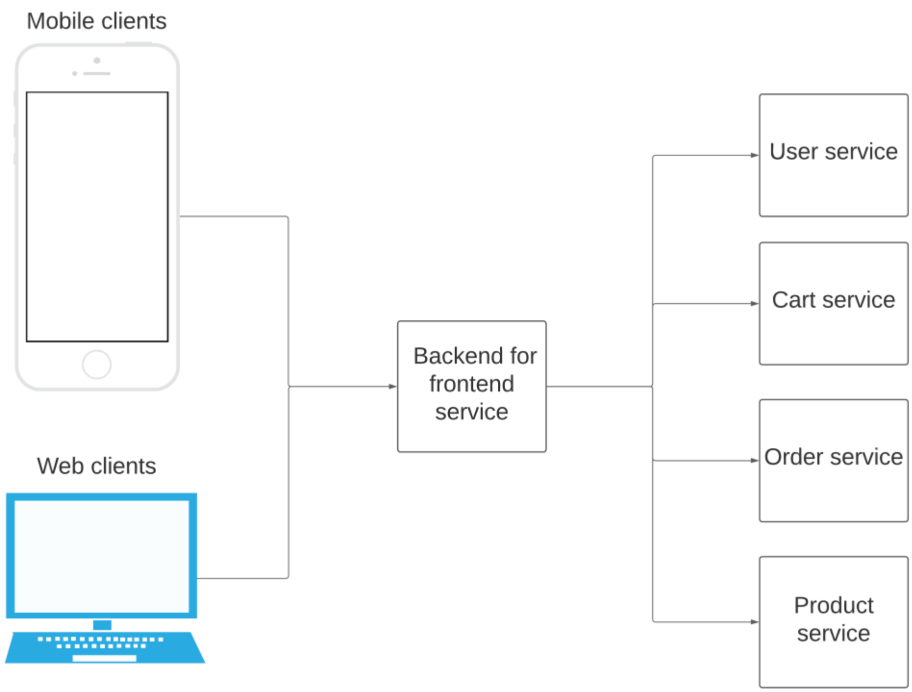


Figure 10.1 the backend for frontend pattern

Now that we understand the backend-for-frontend design pattern and the problems it addresses, let's apply it to build a backend-for frontend service for an ecommerce storefront.

10.3 Implementing the product listing api

Let's implement the backend-for-frontend patten for an 'all products' page of our ecommerce storefront's desktop experience. This page displays all products available on our site along with basic information about our user's cart and favorited items in a menu bar. To increase sales, the page has a 'low inventory' warning when only a few items are left available. This page also has a navigation bar up on top with information about our user's favorited products as well as what data is in their cart.

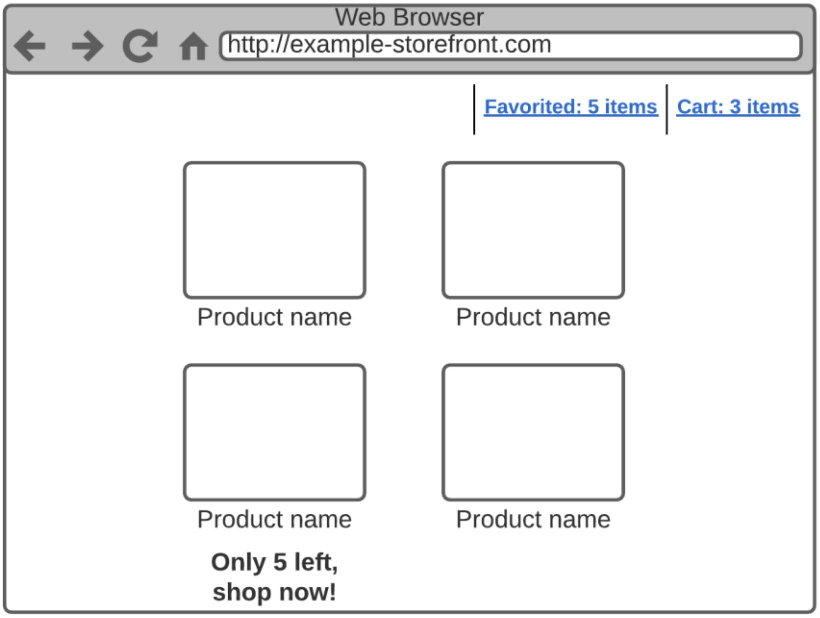


Figure 10.1 a mockup of the products listing page.

Given we have a microservice architecture with several independent services, we'll need to request the appropriate data from each service and stitch them together to form a cohesive response. Let's first start by defining the base services and data models we'll need.

User favorite service

This service keeps track of a mapping from a user to the product ids they have put in their favorite list.

User cart service

This contains a mapping from user id to product ids they have put in cart, the data model is the same as the favorite service.

Inventory service

This contains a mapping from a product id to the available inventory for that product.

Product service

This contains product information, such as descriptions and skus. This is similar to the service we implemented in the last chapter around our products database.

Next, we'll need to implement these services to support our backend-for-frontend, product, inventory, user cart and user favorites.

10.3.1 Implementing the base services

Let's start by implementing an aiohttp application for our inventory service as we'll make this our simplest service. For this service we won't create a separate data model, instead we'll just return a random number from zero to 100 to simulate available inventory. We'll also add a random delay to simulate our service being intermittently slow, we'll use this to demonstrate how to handle timeouts in our product list service. We'll host this service on port 8001 for development purposes so it does not interfere with our product service, from earlier, which runs on port 8000.

Listing 10.1 the inventory service

```
import asyncio
import random
from aiohttp import web
from aiohttp.web_response import Response

routes = web.RouteTableDef()


@routes.get('/products/{id}/inventory')
async def get_inventory() -> Response:
    delay: int = random.randint(0, 5)
    await asyncio.sleep(delay)
    inventory: int = random.randint(0, 100)
    return web.json_response({'inventory': inventory})


app = web.Application()
app.add_routes(routes)
web.run_app(app, port=8001)
```

Next, let's implement the user cart and user favorite service. The data model for these two is identical, so the services will be almost the same with the difference being table names. Let's start with the two data models, user cart and user favorite. We'll also insert a few records in these tables so we have some data to start with.

Listing 10.2 user cart table

```
CREATE TABLE user_cart(
    user_id    INT NOT NULL,
    product_id INT NOT NULL
);

INSERT INTO user_cart VALUES (1, 1);
INSERT INTO user_cart VALUES (1, 2);
INSERT INTO user_cart VALUES (1, 3);
INSERT INTO user_cart VALUES (2, 1);
INSERT INTO user_cart VALUES (2, 2);
INSERT INTO user_cart VALUES (2, 5);
```

Listing 10.3 user favorite table

```
CREATE TABLE user_favorite
```

```
(
    user_id    INT NOT NULL,
    product_id INT NOT NULL
);

INSERT INTO user_favorite VALUES (1, 1);
INSERT INTO user_favorite VALUES (1, 2);
INSERT INTO user_favorite VALUES (1, 3);
INSERT INTO user_favorite VALUES (3, 1);
INSERT INTO user_favorite VALUES (3, 2);
INSERT INTO user_favorite VALUES (3, 3);
```

To simulate multiple databases, we'll want to create these tables each in their own postgres database. Recall from chapter 5 we can run arbitrary SQL with the psql command line utility, meaning we can create two databases for user favorites and user cart with the following two commands:

```
sudo -u postgres psql -c "CREATE DATABASE cart;"
sudo -u postgres psql -c "CREATE DATABASE favorites;"
```

Since we'll now need to set up and tear down connections to multiple different databases, let's create some reusable code across our services to create asyncpg connection pools. We'll reuse this in our aiohttp `on_startup` and `on_cleanup` hooks.

Listing 10.4 creating and tearing down database pools

```
import asyncpg
from aiohttp.web_app import Application
from asyncpg.pool import Pool

DB_KEY = 'database'


async def create_database_pool(app: Application,
                               host: str,
                               port: int,
                               user: str,
                               database: str,
                               password: str):
    pool: Pool = await asyncpg.create_pool(host=host,
                                           port=port,
                                           user=user,
                                           password=password,
                                           database=database,
                                           min_size=6,
                                           max_size=6)
    app[DB_KEY] = pool


async def destroy_database_pool(app: Application):
    pool: Pool = app[DB_KEY]
    await pool.close()
```

Next, let's create the services. In REST terms, both favorites and cart are a sub entity of a particular user. This means each endpoint's root will be users and will accept a user id as an input. For example, /users/3/favorites will fetch the favorite products for user id 3.

Listing 10.5 the user favorite service

```
import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool, destroy_database_pool

routes = web.RouteTableDef()


@routes.get('/users/{id}/favorites')
async def favorites(request: Request) -> Response:
    try:
        str_id = request.match_info['id']
        user_id = int(str_id)
        db = request.app[DB_KEY]
        favorite_query = 'SELECT product_id from user_favorite where user_id = $1'
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for record in result])
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()


app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                        host='0.0.0.0',
                                        port=5432,
                                        user='postgres',
                                        password='password',
                                        database='favorites'))
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8002)
```

Listing 10.6 the user cart service

```
import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool, destroy_database_pool

routes = web.RouteTableDef()


@routes.get('/users/{id}/cart')
async def time(request: Request) -> Response:
    try:
        str_id = request.match_info['id']
        user_id = int(str_id)
        db = request.app[DB_KEY]
        favorite_query = 'SELECT product_id from user_cart where user_id = $1'
```

```
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for record in result])
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()


app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                        host='0.0.0.0',
                                        port=5432,
                                        user='postgres',
                                        password='password',
                                        database='cart'))
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8003)
```

Finally, we'll implement the product service. This will be mainly similar to the API we built in chapter nine, with the difference that we'll fetch all products from our database instead of just one.

Listing 10.7 the product service

```
import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool, destroy_database_pool

routes = web.RouteTableDef()


@routes.get('/products')
async def products(request: Request) -> Response:
    db = request.app[DB_KEY]
    product_query = 'SELECT product_id, product_name FROM product'
    result = await db.fetch(product_query)
    return web.json_response([dict(record) for record in result])


app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                        host='0.0.0.0',
                                        port=5432,
                                        user='postgres',
                                        password='password',
                                        database='products'))
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8000)
```

10.3.2 Implementing the backend for frontend

Next, let's build the backend-for-frontends service. We'll first start with a few requirements for our API based on the needs of our UI. Product load times are crucial for our application, as the longer our users have to wait the less likely they are to continue browsing our site, and the less likely they are to buy products. This makes our requirements center around delivering the minimum viable data to the user as quickly as possible:

1. The API should never wait for the product service more than a second. If it takes longer than a second, we should respond with a timeout error (HTTP code 504) so that the UI does not hang indefinitely.
2. The user cart and favorite data is optional, if we can get it in within a second, that's great! If not, we should just return what product data we have.
3. The inventory data for products is optional as well. If we can't get it, just return the product data.

With these requirements, we've given ourselves a few ways to short-circuit around slow services, or services that have crashed or have other network issues. This makes our service and therefore the user interfaces that consume it more resilient. While it may not always have all the data to provide a complete user experience, it has enough to create a usable experience. Even if the result is a catastrophic failure of the product service, we won't leave the user hanging with a busy spinner or some other poor user experience indefinitely.

Next, let's define what we want our response to look like. Since all we need for the navigation bar is the number of items in our cart and in our favorite list, we'll have our response just represent these as scalar values. Since our cart or favorite service could time out, or could have an error, we'll allow this value to be null. For our product data, we'll just want our normal product data augmented with the inventory value. Since we're adding this to cart and favorite data, we'll put this in a products array. This means we'll have a response similar to the following:

```
{
 "cart_items": 1,
 "favorite_items": null,
 "products": [{"product_id": 4, "inventory": 4},
              {"product_id": 3, "inventory": 65}]
}
```

In this case, the user has one item in their cart. They may have favorite items, but the result is null because there was an issue reaching the favorite service. Finally, we have two products to display with 4 and 65 items in stock respectively.

So how should we begin implementing this functionality? We'll need to communicate with our REST services over HTTP, so aiohttp's web client functionality is a natural choice for this as we're already using the framework's web server. Next, what requests do we make, and how do we group them and manage timeouts? First, we should think about the most requests we can run concurrently upfront. The more we can run concurrently, the faster we can theoretically return a response to our clients. In our case, we can't ask for inventory before we have product ids, so we can't run that concurrently, but our products, cart and favorites services are not dependent on one another. This means we can run them concurrently with an asyncio api such as `wait`. Using `wait` with a timeout will give us a done set where we can check which requests finished with error, and which are still running after the timeout, giving us a chance to handle any failures. Then, once we have product ids and potentially user favorite and cart data, we can begin to stitch together our final response and send that back to the client.

Listing 10.8 the product backend for frontend

```
import asyncio
from asyncio import Task
import aiohttp
from aiohttp import web, ClientSession
from aiohttp.web_request import Request
from aiohttp.web_response import Response
import logging
from typing import Dict, Set, Awaitable, Optional, List

routes = web.RouteTableDef()

PRODUCT_BASE = 'http://0.0.0.0:8080'
INVENTORY_BASE = 'http://0.0.0.0:8001'
FAVORITE_BASE = 'http://0.0.0.0:8002'
CART_BASE = 'http://0.0.0.0:8003'


@routes.get('/products/all')
async def all_products(request: Request) -> Response:
    async with aiohttp.ClientSession() as session:
        products = asyncio.create_task(session.get(f'{PRODUCT_BASE}/products'))
        favorites = asyncio.create_task(session.get(f'{FAVORITE_BASE}/users/3/favorites'))
        cart = asyncio.create_task(session.get(f'{CART_BASE}/users/3/cart'))

        requests = [products, favorites, cart]
        done, pending = await asyncio.wait(requests, timeout=1.0)

        if products in pending:
            [request.cancel() for request in requests]
            return web.json_response({'error': 'Could not reach products service.'}, status=504)
        elif products in done and products.exception() is not None:
            [request.cancel() for request in requests]
            logging.exception('Server error reaching product service.',
       exc_info=products.exception())
            return web.json_response({'error': 'Server error reaching products service.'},
       status=500)
        else:
            product_response = await products.result().json()
            product_results: List[Dict] = await get_products_with_inventory(session,
       product_response)

            cart_item_count: Optional[int] = await get_response_item_count(cart,
                                                                              done,
                                                                              pending,
                                                                              'Error getting user
       cart.')
            favorite_item_count: Optional[int] = await get_response_item_count(favorites,
                                                                                  done,
                                                                                  pending,
                                                                                  'Error getting
       user favorites.')
```

```
            return web.json_response({'cart_items': cart_item_count,
                                      'favorite_items': favorite_item_count,
                                      'products': product_results})


async def get_products_with_inventory(session: ClientSession, product_response) -> List[Dict]:
    def get_inventory(session: ClientSession, product_id: str) -> Task:
        url = f"{INVENTORY_BASE}/products/{product_id}/inventory"
        return asyncio.create_task(session.get(url))

    def create_product_record(product_id: int, inventory: int) -> Dict:
        return {'product_id': product_id, 'inventory': inventory}

    inventory_tasks_to_product_id = {
        get_inventory(session, product['product_id']): product['product_id'] for product in
       product_response
    }

    inventory_done, inventory_pending = await asyncio.wait(inventory_tasks_to_product_id.keys(),
       timeout=1.0)

    product_results = []

    for done_task in inventory_done:
        if done_task.exception() is None:
            product_id = inventory_tasks_to_product_id[done_task]
            inventory = await done_task.result().json()
            product_results.append(create_product_record(product_id, inventory['inventory']))
        else:
            product_id = inventory_tasks_to_product_id[done_task]
            product_results.append(create_product_record(product_id, None))
            logging.exception(f'Error getting inventory for id {product_id}',
                              exc_info=inventory_tasks_to_product_id[done_task].exception())

    for pending_task in inventory_pending:
        pending_task.cancel()
        product_id = inventory_tasks_to_product_id[pending_task]
        product_results.append(create_product_record(product_id, None))

    return product_results


async def get_response_item_count(task: Task,
                                  done: Set[Awaitable],
                                  pending: Set[Awaitable],
                                  error_msg: str) -> Optional[int]:
    if task in done and task.exception() is None:
        return len(await task.result().json())
    elif task in pending:
        task.cancel()
    else:
        logging.exception(error_msg, exc_info=task.exception())

    return None


app = web.Application()
app.add_routes(routes)
web.run_app(app, port=9000)
```

In listing 10.8 we first define a route handler named `all_products`. In `all_products`, we fire off requests to our products, cart and favorite services concurrently, giving these requests one second to complete with `wait`. Once either all of them finish, or we have waited for a second, we begin to process the results.

Since the product response is critical, we check its status first. If it is still pending or has an exception, we cancel any pending requests and return an error to the client. If there was an exception, we respond with a HTP 500 error, indicating a server issue. If there was a timeout, we respond with a 504 indicating we couldn't reach the service. This specificity gives our clients a hint as to if they should try again, and also gives us more information useful for any monitoring and altering we may have.

If we have a successful response from the product service, we can now start to process it and ask for inventory numbers. We do this work in a helper function called `get_products_with_inventory`. In this helper function, we pull product ids from the response body and use these to construct requests to the inventory service. Since our inventory service only accepts one product id at a time (ideally, you would be able to batch these into a single request, but we'll pretend the team that manages the inventory service has issues with this approach), we'll create a list of tasks to request inventory per each product. We'll again pass these into the `wait` coroutine, giving them all one second to complete.

Since inventory numbers are optional, once our timeout is up, we begin processing everything in both the done and pending sets of inventory requests. If we have a successful response from the inventory service, we create a dictionary with the product information alongside the inventory number. If there was either an exception, or the request is still in the pending set, we create a record with the inventory as `None`, indicating we couldn't retrieve it. Using `None` will give us a null value when we turn our response into JSON.

Finally, we check the cart and favorite responses. All we need to do for both these requests is count the number of items returned. Since this logic is nearly identical for both services, we create a helper method to count items named `get_response_item_count`. In `get_response_item_count` if we have a successful result from either the cart or favorite service, it will be a JSON array, so we count and return the number of items in that array. If there was an exception, or the request has taken longer than a second, we set the result to `None` so we get a `null` value in our JSON response.

This implementation provides us with a fairly robust way to deal with failures and timeouts of our non-critical services, ensuring that we give a sensible response quickly even in the result of downstream issues. No single request to a downstream service will take longer than one second, creating an approximate upper bound for how slow our service can be. However, while we've created something fairly robust, there are still a few ways we can make this even more resilient to issues.

10.3.3 Retrying failed requests

One issue with our first implementation is that it pessimistically assumes that if we get an exception from a service, that is the end of the road. We assume we can't get results and we move on.

While this can make sense, it is the case that an issue with a service could be transient. For example, there may be a networking hiccup that disappears rather quickly, there may be a temporary issue with any load balancers we're using, or there could be any other host of temporary issues.

In these cases, it can make sense to retry a few times with a short delay in between retries. This gives the error a chance to clear up and can give our users more data than they would otherwise have if we were pessimistic about our failures. This of course comes with the tradeoff of having our users wait longer, potentially just to see the same failure they would have otherwise.

To implement this functionality, the `wait_for` coroutine function is a perfect candidate. It will bubble up any exception we get, and it lets us specify a timeout. If we surpass that timeout, it raises a `TimeoutException` and cancels the task we started. Let's try and create a reusable retry coroutine that does this for us. We'll create a `retry` coroutine function that takes in coroutine as well as a number of times to retry. If the coroutine we pass in fails or times out, we'll retry up to the number of times we specified:

Listing 10.9 a retry coroutine

```
import asyncio
import logging
from typing import Callable, Awaitable


class TooManyRetries(Exception):
    pass


async def retry(coro: Callable[[], Awaitable],
                max_retries: int,
                timeout: float,
                retry_interval: float):
    for retry_num in range(0, max_retries):
        try:
            return await asyncio.wait_for(coro(), timeout=timeout)
        except Exception as e:
            logging.exception(f'Exception while waiting (tried {retry_num} times), retrying.',
      exc_info=e)
            await asyncio.sleep(retry_interval)
    raise TooManyRetries()
```

In listing 10.9, we first create a custom exception class that we'll raise when we are still failing after the maximum amount of retries. This will let any callers catch this exception and handle this specific issue as they see fit. The `retry` coroutine takes in a few arguments, the first argument is a callable that returns an awaitable, this is the coroutine that we'll retry. The second argument is the number of times we'd like to retry, and the final arguments are the timeout and the interval to wait to retry after a failure. We create a loop that wraps the coroutine in `wait_for`, if this completes successfully, we return the results and exit the function. If there was an error, timeout or otherwise, we catch the exception, log it, and sleep for the specified interval time, retrying again after we've slept. If our loop finishes without an error-free call of our coroutine, we raise a `TooManyRetries` exception.

We can test this out by creating a couple of coroutines that exhibit the failure behavior we'd like to handle. First, one which always throws an exception and second, one which always times out.

Listing 10.10 testing the retry coroutine

```
import asyncio
from chapter_10.listing_10_9 import retry, TooManyRetries


async def main():
    async def always_fail():
        raise Exception("I've failed!")

    async def always_timeout():
        await asyncio.sleep(1)

    try:
        await retry(always_fail,
                    max_retries=3,
                    timeout=.1,
                    retry_interval=.1)
    except TooManyRetries:
        print('Retried too many times!')

    try:
        await retry(always_timeout,
                    max_retries=3,
                    timeout=.1,
                    retry_interval=.1)
    except TooManyRetries:
        print('Retried too many times!')


asyncio.run(main())
```

For both retries, we define a timeout and retry interval of 100 milliseconds and a max retry amount of three. This means we give the coroutine 100 milliseconds to complete, and if it does not within that time, or it fails, we wait 100 milliseconds before trying again. Running this listing, you should see each coroutine try to run three times and finally print 'retried too many times!', leading to output similar to the following (tracebacks omitted for brevity):

```
ERROR:root:Exception while waiting (tried 1 times), retrying.
Exception: I've failed!
ERROR:root:Exception while waiting (tried 2 times), retrying.
Exception: I've failed!
ERROR:root:Exception while waiting (tried 3 times), retrying.
Exception: I've failed!
Retried too many times!
ERROR:root:Exception while waiting (tried 1 times), retrying.
ERROR:root:Exception while waiting (tried 2 times), retrying.
ERROR:root:Exception while waiting (tried 3 times), retrying.
Retried too many times!
```

Using this, we can add some simple retry logic to our product backend for frontend. For example, let's say we wanted to retry our initial requests to the products, cart and favorites services a few times before considering their error unrecoverable. We can do this by wrapping each request in the retry coroutine like so:

```
product_request = functools.partial(session.get, f'{PRODUCT_BASE}/products')
favorite_request = functools.partial(session.get, f'{FAVORITE_BASE}/users/5/favorites')
```

```
cart_request = functools.partial(session.get, f'{CART_BASE}/users/5/cart')

products = asyncio.create_task(retry(product_request,
                                     max_retries=3,
                                     timeout=.1,
                                     retry_interval=.1))

favorites = asyncio.create_task(retry(favorite_request,
                                      max_retries=3,
                                      timeout=.1,
                                      retry_interval=.1))

cart = asyncio.create_task(retry(cart_request,
                                 max_retries=3,
                                 timeout=.1,
                                 retry_interval=.1))

requests = [products, favorites, cart]
done, pending = await asyncio.wait(requests, timeout=1.0)
```

In this example, we try each service a maximum of three times. This lets us recover from issues with our services that may be transient. While this is an improvement, there is another potential issue that can hurt our service. For example, what happens if our product service always times out?

10.3.4 The circuit breaker pattern

One issue we still have in our implementation occurs when a service is consistently slow enough such that it always times out. This can happen when a downstream service is under load, there is some other network issue happening or a whole host of other application or network errors.

You may be tempted to say "well, our application handles the timeout gracefully - the user won't wait for more than a second before seeing an error or getting partial data, so what is the problem?" – and you're not wrong to say that. However, while we've designed our system to be robust and resilient, consider the user experience in this case. For example, if the cart service is experiencing an issue such that it always takes one second to time out, this means that all users will be stuck waiting for a second for results from the service.

In this instance, since an issue with the cart service that could last for some time, anyone who hits our backend for frontend will be stuck waiting for a second when we *know* that this issue is highly likely to happen. Is there a way we can short circuit a call that is likely to fail so that we don't cause unneeded delays to our users?

There is an aptly named pattern to handle this called the circuit breaker pattern. Popularized by Michael Nygard's book *Release It*, this pattern lets us 'flip a circuit breaker' when we have a specified number of errors in a given time period. We can use this to bypass the slow service until the issues with it clear up, ensuring our response to our users remains as fast as possible.

Much like an electrical circuit breaker, a basic circuit breaker pattern has two states associated with it that are the same as a normal circuit breaker on your electrical panel, an open state and a closed state. The closed state is a happy path, we make a request to a service and it returns normally. The open state happens is when the circuit is tripped. In this state, we don't bother to call the service as we know it has a problem, instead we instantly return an error. The circuit breaker pattern quite literally stops us from sending electricity to the bad service. In addition to these two states there is a 'half-open' state. This happens when we're in the open state after a certain time interval. In this state we issue a single request to check if the issue with the service is fixed. If it is, we close the circuit breaker, if not, we keep it open. For the sake of keeping our example simple, we'll skip the half-open state and just focus on the closed and open states.

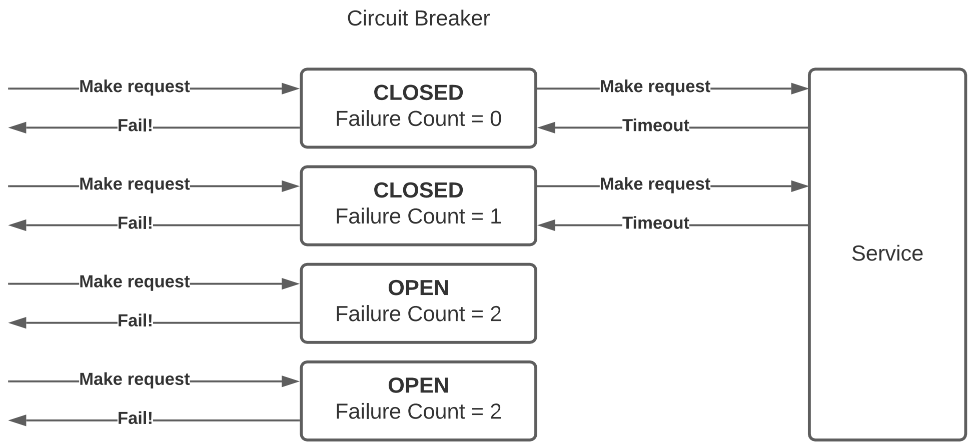


Figure 10.2 A circuit breaker that opens after two failures. Once opened, all requests will fail instantly.

Let's implement a simple circuit breaker to understand how this works. We'll allow the users of the circuit breaker to specify a time window and a maximum number of failures. If more than the maximum number of errors happens within the time window, we'll open the circuit breaker and fail any other calls. We'll do this with a class that takes the coroutine we wish to run and keeps track if we are in the open or closed state.

Listing 10.11 a simple circuit breaker

```
import asyncio
from datetime import datetime, timedelta


class CircuitOpenException(Exception):
    pass


class CircuitBreaker:

    def __init__(self,
                 callback,
                 timeout: float,
                 time_window: float,
                 max_failures: int,
                 reset_interval: float):
        self.callback = callback
        self.timeout = timeout
```

```
        self.time_window = time_window
        self.max_failures = max_failures
        self.reset_interval = reset_interval
        self.last_request_time = None
        self.last_failure_time = None
        self.current_failures = 0

    async def request(self, *args, **kwargs):
        if self.current_failures >= self.max_failures:
            if datetime.now() > self.last_request_time + timedelta(seconds=self.reset_interval):
                self._reset('Circuit is going from open to closed, resetting!')
                return await self._do_request(*args, **kwargs)
            else:
                print('Circuit is open, failing fast!')
                raise CircuitOpenException()
        else:
            if self.last_failure_time and datetime.now() > self.last_failure_time +
      timedelta(seconds=self.time_window):
                self._reset('Interval since first failure elapsed, resetting!')
            print('Circuit is closed, requesting!')
            return await self._do_request(*args, **kwargs)

    def _reset(self, msg: str):
        print(msg)
        self.last_failure_time = None
        self.current_failures = 0

    async def _do_request(self, *args, **kwargs):
        try:
            print('Making request!')
            self.last_request_time = datetime.now()
            return await asyncio.wait_for(self.callback(*args, **kwargs), timeout=self.timeout)
        except Exception as e:
            self.current_failures = self.current_failures + 1
            if self.last_failure_time is None:
                self.last_failure_time = datetime.now()
            raise
```

Our circuit breaker class takes five constructor parameters. The first two are the callback we wish to run with the circuit breaker and a `timeout` which represents how long we'll allow the callback to run before failing with a timeout. The next three are related to handling failures and resets. The `max_failure` parameter is the maximum number of failures we'll tolerate within `time_window` seconds before opening the circuit. The `reset_interval` parameter is how many seconds we wait to reset the breaker from the open to closed state after `max_failure` failures have occurred.

We then define a coroutine method `request`, this calls our callback and keeps track of how many failures we've had, returning the result of the callback if there were no errors. When we have a failure, we keep track of this in a counter `failure_count`. If the failure count exceeds the `max_failure` threshold we set within the specified time interval, any further calls to request will raise a `CircuitOpenException`. If the reset interval has elapsed, we reset the `failure_count` to zero and begin making requests again (if our breaker was closed, which it may not be).

Now let's see our breaker in action with a simple example application. We'll create a `slow_callback` coroutine that just sleeps for two seconds. We'll then use that in our breaker, setting a short timeout that will let us easily trip the breaker.

Listing 10.12 the breaker in action

```
import asyncio
from chapter_10.listing_10_11 import CircuitBreaker


async def main():
    async def slow_callback():
        await asyncio.sleep(2)

    cb = CircuitBreaker(slow_callback,
                        timeout=1.0,
                        time_window=5,
                        max_failures=2,
                        reset_interval=5)

    for _ in range(4):
        try:
            await cb.request()
        except Exception as e:
            pass

    print('Sleeping for 5 seconds so breaker closes...')
    await asyncio.sleep(5)

    for _ in range(4):
        try:
            await cb.request()
        except Exception as e:
            pass


asyncio.run(main())
```

In listing 10.12 we create a breaker with a one second timeout that tolerates two failures within a five second interval, and resets after five seconds once the breaker is open. We then try to make five requests rapidly to the breaker. The first two should take a second before failing with a timeout, then every subsequent call will fail instantly as the breaker is open. We then sleep for five seconds, this lets the breaker's `reset_interval` elapse, so it should move back to the closed state and start to make calls to our callback again. Running this, you should see output as follows:

```
Circuit is closed, requesting!
Circuit is closed, requesting!
Circuit is open, failing fast!
Circuit is open, failing fast!
Sleeping for 5 seconds so breaker closes...
Circuit is going from open to closed, requesting!
Circuit is closed, requesting!
Circuit is open, failing fast!
Circuit is open, failing fast!
```

Now that we have a simple implementation, we can combine this with our retry logic and use it in our backend-for-frontend. Since we've purposefully made our inventory service slow to simulate a real-life legacy service, this is a natural place to add our circuit breaker. We'll set a timeout of 500 milliseconds and tolerate 5 failures within a second, after which we'll set a reset interval of 30 seconds. We'll need to rewrite our `get_inventory` function into a coroutine to do this like so:

```
async def get_inventory(session: ClientSession, product_id: str):
    url = f"{INVENTORY_BASE}/products/{product_id}/inventory"
    return await session.get(url)


inventory_circuit = CircuitBreaker(get_inventory, timeout=.5, time_window=5.0, max_failures=3,
       reset_interval=30)
```

Then, in our `all_products` coroutine we'll need to change how we create our inventory service requests. We'll create a task with a call to our inventory circuit breaker instead of the `get_inventory` coroutine:

```
inventory_tasks_to_pid = {
    asyncio.create_task(inventory_circuit.request(session, product['product_id'])):
       product['product_id']
    for product in product_response
 }

inventory_done, inventory_pending = await asyncio.wait(inventory_tasks_to_pid.keys(),
       timeout=1.0)
```

Once we've made these changes you should see call time decrease to the products backend-for-frontend after a few calls. Since we're simulating an inventory service that is slow under load, we'll eventually trip the circuit breaker with a few timeouts and then any subsequent call won't make any more requests to the inventory service until the breaker resets. Our backend-for-frontend service is now more robust in the face of a slow and failure-prone inventory service. We could also apply this to all our other calls if desired to increase the stability of these as well.

In this example, we've implemented a very simple implementation of a circuit breaker to demonstrate how it works and how to implement it with asyncio. There are several existing implementations of this pattern with many other knobs to tune to your specific needs. If you're considering this pattern, take some time to do research on the circuit breaker libraries available before implementing it yourself.

10.4 Summary

- Microservices have several benefits over monoliths, including, but not limited to, independent scalability and deployability.
- The backend for frontend pattern is a microservice pattern that aggregates the calls from several downstream services. We've learned how to apply a microservice architecture to an ecommerce use case, creating multiple independent services with aiohttp.
- We've used asyncio utility functions such as `wait` to ensure that our backend-for-frontend service remains resilient and responsive to failures of downstream services.
- We've created a utility to manage retries of HTTP requests with asyncio and aiohttp.
- We've implemented a basic circuit breaker pattern to ensure a service failure does not

negatively impact other services.