```
    context = async_to_sync(partial(make_requests, url, requ
    return render(request, 'async_api/requests.html', contex
```

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:/ / e
```

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.

ary

- 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. A *monolith* generally refers to a medium-to-large-sized application containing multiple modules that are independently deployed and managed as one unit. While there is nothing inherently wrong with this model (monoliths are perfectly fine, and even preferable, for most 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 microservices and the motivations behind them. We'll learn a pattern called *backend-for-frontend* and apply this to an e-commerce microservice architecture. We'll then implement this API with aiohttp and asyncpg, learning how to employ 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.

/hy microservices?

First, let's define 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, *microservices* follow a few guiding principles:

- They are loosely coupled and independently deployable.
- They have their own independent stack, including a data model.
- They communicate with one another over a protocol such as REST or gRPC.
- They follow the "single responsibility" principle; that is, a microservice should "do one thing and do it well."

Let's apply these principles to a concrete example of an e-commerce storefront. An application like this has users that provide shipping and payment information to our hypothetical organization who then buy our products. In a monolithic architecture, we'd have one application with one database to manage user 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 most applications; they are simpler to manage. Make a code change, and run all the test suites to make sure your seemingly small change does not affect other areas of the system. Once you've run tests, you deploy the application as one unit. Is your application not performing well under load? In this case you can scale horizontally or vertically, either deploying more instances of your application or deploying to more powerful machines to handle the additional users. While managing a monolith is operationally simpler, this simplicity has drawbacks that may matter a lot, 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 technology cost inefficiencies. In the context of an e-commerce application, you will typically get much fewer 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, new challenges emerge. Imagine you have five teams working on the same monolithic codebase with 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.

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`, which allow us to aggregate exceptions from a group of coroutines or tasks. 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.

ıtroducing the backend-for-frontend pattern

When we're building UIs in a microservice architecture, we'll typically need to get data from multiple services 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 standalone 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 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, and some may be in developing countries without access to high-speed internet at all. If we make five slow requests to five services, there is 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 as well as 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 redundant and puts us at risk of having inconsistent logic across clients.

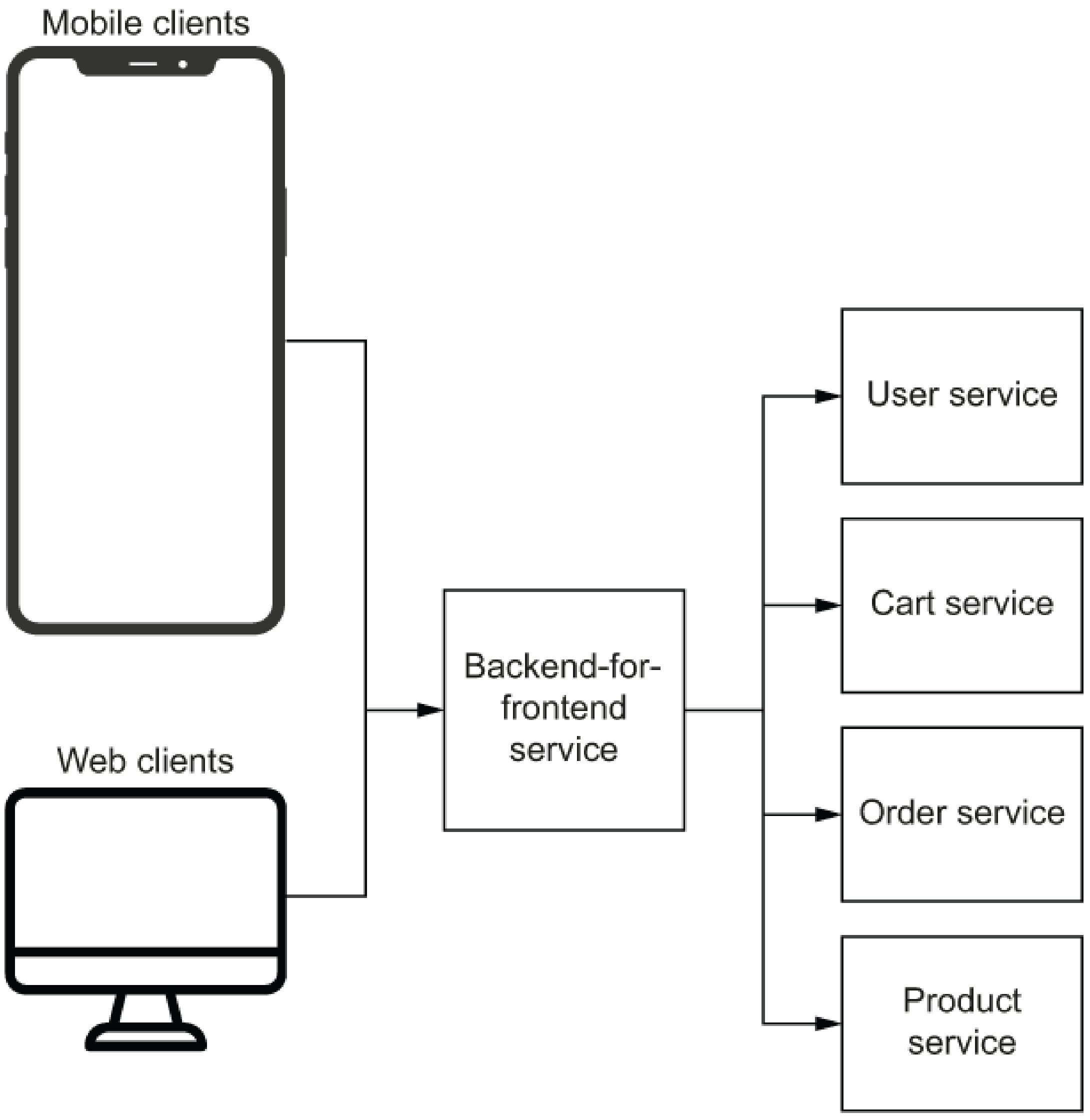

Figure 10.1 The backend-for-frontend pattern

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, and 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 enables multiple backend-for-frontend services for 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. This is illustrated in figure 10.1. 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 e-commerce storefront.

nplementing the product listing API

Let's implement the backend-for-frontend pattern for an *all products* page of our e-commerce 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 favorite products as well as what data is in their cart. Figure 10.2 illustrates our UI.

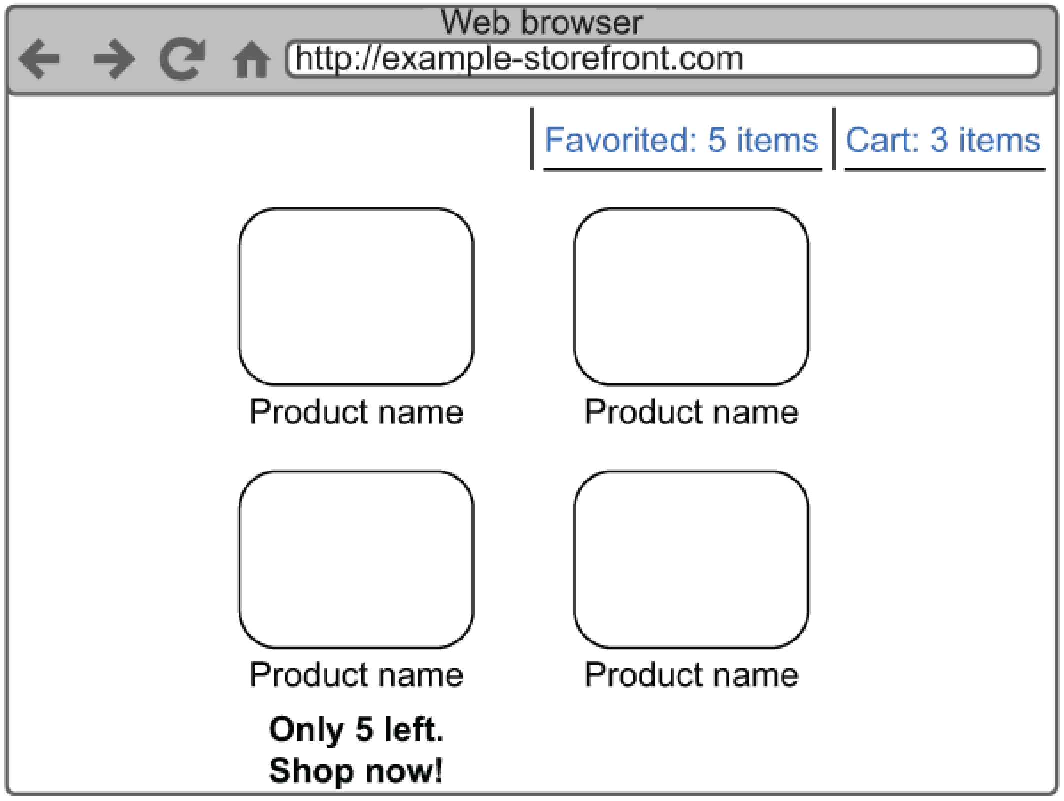

Figure 10.2 A mockup of the products listing page

Given 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 favorites list. Next, we'll need to implement these services to support our backend-for-frontend product, inventory, user cart, and user favorites.

User cart service

This contains a mapping from user ID to product IDs they have put in the cart; the data model is the same as the user 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 chapter 9 around our products database.

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 0 to 100 to simulate available inventory. We’ll also add a random delay to simulate our service being intermittently slow, and 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 chapter 9, 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(request: Request) -> 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. First, we’ll start with the user cart table.

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);
```

Next, we'll create the user favorite table and insert a few values; this will look very similar to the previous table.

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 that we can run arbitrary SQL with the psql command-line utility, meaning that 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()
```

The preceding listing should look similar to code we wrote in chapter 5 to set up database connections. In `create_database_pool`, we create a connection pool and then put it in our `Application` instance. In `destroy_database_pool`, we grab the connection pool from the application instance and close it.

Next, let's create the services. In REST terms, both favorites and cart are a subentity of a particular user. This means that 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`. First, we'll create the user favorite service:

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_

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_favori
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for recor
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()


app = web.Application()
app.on_startup.append(functools.partial(create_database_pool
                                        host='127.0.0.1',
                                        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)
```

Next, we'll create the user cart service. This code will look mostly similar to our previous service with the main difference being we'll interact with the `user_cart` table.

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_

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 w
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for recor
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()


app = web.Application()
app.on_startup.append(functools.partial(create_database_pool
                                        host='127.0.0.1',
                                        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 similar to the API we built in chapter 9 with the difference being that we’ll fetch all products from our database instead of just one. With the following listing, we’ve created four services to power our theoretical e-commerce storefront!

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_

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 pr
    result = await db.fetch(product_query)
    return web.json_response([dict(record) for record in res



app = web.Application()
app.on_startup.append(functools.partial(create_database_pool
                                        host='127.0.0.1',
                                                  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)
```

Implementing the backend-for-frontend service

Next, let's build the backend-for-frontend 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 must wait, the less likely they are to continue browsing our site and the less likely they are to buy our products. This makes our requirements center around delivering the minimum viable data to the user as quickly as possible:

- The API should never wait for the product service more than 1 second. If it takes longer than 1 second, we should respond with a timeout error (HTTP code 504), so the UI does not hang indefinitely.
- The user cart and favorites data is optional. If we can get it in within 1 second, that's great! If not, we should just return what product data we have.
- 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 indefinitely with a busy spinner or some other poor user experience.

Next, let's define what we want our response to look like. All we need for the navigation bar is the number of items in our cart and in our favorites list, so 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, so we'll add this data 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. 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 favorite 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.

We'll create an endpoint `/products/all` to do this that will return this data. Normally, we'd want to accept the currently logged-in user's ID somewhere in the URL, the request headers, or a cookie, so we can use that when making requests to our downstream services. In this example, for simplicity's sake, we'll just hardcode this ID to the user for whom we've already inserted data into our database.

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:/ /127.0.0.1:8000'
```

```
INVENTORY_BASE = 'http:/ /127.0.0.1:8001'
FAVORITE_BASE = 'http:/ /127.0.0.1:8002'
CART_BASE = 'http:/ /127.0.0.1:8003'


@routes.get('/products/all')
async def all_products(request: Request) -> Response:
    async with aiohttp.ClientSession() as session:
        products = asyncio.create_task(session.get(f'{PRODUC
        favorites = asyncio.create_task(session.get(f'{FAVOR
        cart = asyncio.create_task(session.get(f'{CART_BASE}

        requests = [products, favorites, cart]
        done, pending = await asyncio.wait(requests, timeout

        if products in pending:
            [request.cancel() for request in requests]
            return web.json_response({'error': 'Could not re
        elif products in done and products.exception() is no
            [request.cancel() for request in requests]
            logging.exception('Server error reaching product
            return web.json_response({'error': 'Server error
        else:
            product_response = await products.result().json(
            product_results: List[Dict] = await get_products

            cart_item_count: Optional[int] = await get_respo



            favorite_item_count: Optional[int] = await get_r
            return web.json_response({'cart_items': cart_ite
                                      'favorite_items': favo
                                      'products': product_re


async def get_products_with_inventory(session: ClientSession
                      product_response) -> List[Dict]:
    def get_inventory(session: ClientSession, product_id: st
        url = f"{INVENTORY_BASE}/products/{product_id}/inven
        return asyncio.create_task(session.get(url))

    def create_product_record(product_id: int, inventory: Op
        return {'product_id': product_id, 'inventory': inven

    inventory_tasks_to_product_id = {
        get_inventory(session, product['product_id']): produ
    }

    inventory_done, inventory_pending = await asyncio.wait(i

    product_results = []

    for done_task in inventory_done:
        if done_task.exception() is None:
            product_id = inventory_tasks_to_product_id[done_
            inventory = await done_task.result().json()
            product_results.append(create_product_record(pro
        else:
            product_id = inventory_tasks_to_product_id[done_
            product_results.append(create_product_record(pro
            logging.exception(f'Error getting inventory for 
                              exc_info=inventory_tasks_to_pr

    for pending_task in inventory_pending:
        pending_task.cancel()
        product_id = inventory_tasks_to_product_id[pending_t
        product_results.append(create_product_record(product

    return product_results


async def get_response_item_count(task: Task,
                                  done: Set[Awaitable],
                                  pending: Set[Awaitable],
                                  error_msg: str) -> Optiona
    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)
```

❶ Create tasks to query the three services we have and run them concurrently.

❷ If the products request times out, return an error, as we can’t proceed.

❸ Extract data from the product response, and use it to get inventory data.

❹ Given a product response, make requests for inventory.

❺ Convenience method to get the number of items in a JSON array response

In the preceding listing, we first define a route handler named `all_products`. In `all_products`, we send requests to our products, cart, and favorite services concurrently, giving these requests 1 second to complete with `wait`. Once either all of them finish, or we have waited for 1 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 HTTP 500 error, indicating a server issue. If there was a timeout, we respond with a 504 error, indicating we couldn't reach the service. This specificity gives our clients a hint as to whether they should try again and also gives us more information useful for any monitoring and altering we may have (we can have alerts specifically to watch 504 response rates, for example).

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 1 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 took longer than 1 second, we set the result to `None,` so we get a `null` value in our JSON response.

This implementation provides us with a reasonably 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 1 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.

Retrying failed requests

One issue with our first implementation is that it pessimistically assumes that if we get an exception from a service, 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 raise 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=ti
        except Exception as e:
            logging.exception(f'Exception while waiting (tri
            await asyncio.sleep(retry_interval)
    raise TooManyRetries()
```

❶ Wait for a response for the specified timeout.

❷ If we get an exception, log it and sleep for the retry interval.

❸ If we've failed too many times, raise an exception to indicate that.

In the preceding listing, 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`, and 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 doesn't complete 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 favorite 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_
favorite_request = functools.partial(session.get, f'{FAVORIT
cart_request = functools.partial(session.get, f'{CART_BASE}/

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?

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 multitude of other application or network errors.

You may be tempted to ask, “Well, our application handles the timeout gracefully; the user won’t wait for more than 1 second before seeing an error or getting partial data, so what is the problem?” And you’re not wrong to ask. However, while we’ve designed our system to be robust and resilient, consider the user experience. For example, if the cart service is experiencing an issue such that it always takes 1 second to time out, this means that all users will be stuck waiting for 1 second for results from the service.

In this instance, since this issue with the cart service could last for some time, anyone who hits our backend-for-frontend will be stuck waiting for 1 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 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* (The Pragmatic Bookshelf, 2017), this pattern lets us “flip a circuit breaker,” and when we have a specified number of errors in each 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 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 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, and 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, as shown in figure 10.3.

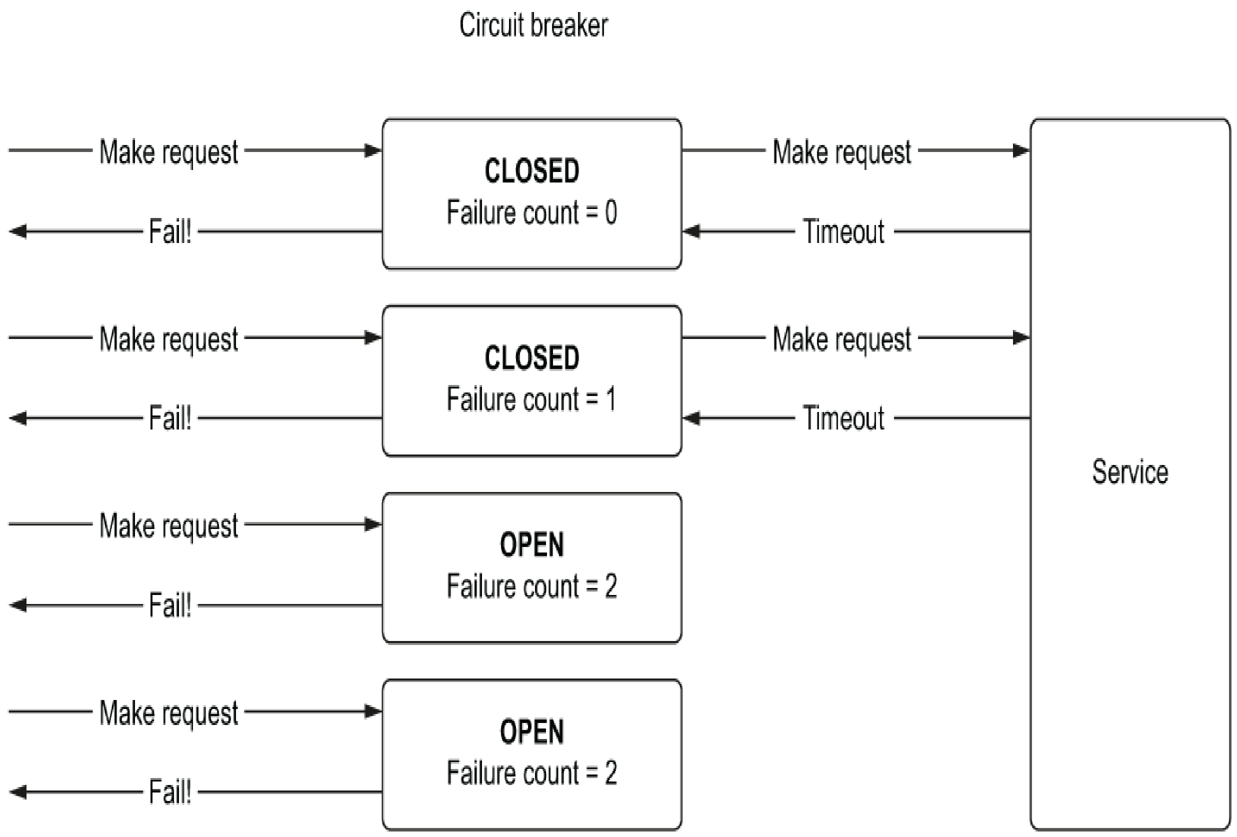

Figure 10.3 A circuit breaker that opens after two failures. Once opened, all requests will fail instantly.

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 + tim
                self._reset('Circuit is going from open to c
                return await self._do_request(*args, **kwarg
            else:
                print('Circuit is open, failing fast!')
                raise CircuitOpenException()
        else:
            if self.last_failure_time and datetime.now() > s
                self._reset('Interval since first failure el
            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(*arg
        except Exception as e:
            self.current_failures = self.current_failures +
            if self.last_failure_time is None:
                self.last_failure_time = datetime.now()
            raise
```

❶ Make the request, failing fast if we’ve exceeded the failure count.

❷ Reset our counters and last failure time.

❸ Make the request, keeping track of how many failures we’ve had and when they last happened.

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`, which 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 sleeps for just 2 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 the preceding listing, we create a breaker with a 1-second timeout that tolerates two failures within a 5-second interval and resets after 5 seconds once the breaker is open. We then try to make four requests rapidly to the breaker. The first two should take 1 second before failing with a timeout, then every subsequent call will fail instantly as the breaker is open. We then sleep for 5 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 five failures within 1 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:
    url = f"{INVENTORY_BASE}/products/{product_id}/inventory
    return await session.get(url)


inventory_circuit = CircuitBreaker(get_inventory, timeout=.5
```

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, pro
  for product in product_response
}

inventory_done, inventory_pending = await asyncio.wait(inven
```

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 circuit breaker to demonstrate how it works and how to build it with asyncio. There are several existing implementations of this pattern with many other features 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.

ary

- 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 e-commerce 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.

11 Synchronization

This chapter covers

- Single-threaded concurrency issues
- Using locks to protect critical sections
- Using semaphores to limit concurrency
- Using events to notify tasks
- Using conditions to notify tasks and acquire a resource

When we write applications using multiple threads and multiple processes, we need to worry about race conditions when using non-atomic operations. Something as simple as incrementing an integer concurrently can cause subtle, hard-to-reproduce bugs. When we are using asyncio, however, we're always using a single thread (unless we're interoperating with multithreading and multiprocessing), so doesn't that mean we don't need to worry about race conditions? It turns out it is not quite so simple.

While certain concurrency bugs that would occur in multithreaded or multiprocessing applications are eliminated by asyncio's single-threaded nature, they are not completely eliminated. While you likely won't need to use synchronization often with asyncio, there are still cases where we need these constructs. asyncio's *synchronization primitives* can help us prevent bugs unique to a single-threaded concurrency model.

Synchronization primitives are not limited to preventing concurrency bugs and have other uses as well. As an example, we may be working with an API that lets us make only a few requests concurrently as per a contract we have with a vendor, or there may be an API we're concerned about overloading with requests. We may also have a workflow with several workers that need to be notified when new data is available.

In this chapter, we'll look at a few examples where we can introduce race conditions in our asyncio code and learn how to solve them with locks and other concurrency primitives. We'll also learn how to use semaphores to limit concurrency and control access to a shared resource, such as a database connection pool. Finally, we'll look at events and conditions that we can use to notify tasks when something occurs and gain access to shared resources when that happens.

nderstanding single-threaded concurrency bugs

In earlier chapters on multiprocessing and multithreading, recall that when we were working with data that is shared among different processes and threads, we had to worry about race conditions. This is because a thread or process could read data while it is being modified by a different thread or process, leading to an inconsistent state and therefore corruption of data.

This corruption was in part due to some operations being non-atomic, meaning that while they appear like one operation, they comprise multiple separate operations under the hood. The example we gave in chapter 6 dealt with incrementing an integer variable; first, we read the current value, then we increment it, then we reassign it back to the variable. This gives other threads and processes ample opportunities to get data in an inconsistent state.

In a single-threaded concurrency model, we avoid race conditions caused by non-atomic operations. In asyncio's single-threaded model, we only have one thread executing one line of Python code at any given time. This means that even if an operation is non-atomic, we’ll always run it to completion without other coroutines reading inconsistent state information.

To prove this to ourselves, let’s try and re-create the race condition we looked at in chapter 7 with multiple threads trying to implement a shared counter. Instead of having multiple threads modify the variable, we’ll have multiple tasks. We’ll repeat this 1,000 times and assert that we get the correct value back.

Listing 11.1 Attempting to create a race condition

```
import asyncio

counter: int = 0



async def increment():
    global counter
    await asyncio.sleep(0.01)
    counter = counter + 1
async def main():
    global counter
    for _ in range(1000):
        tasks = [asyncio.create_task(increment()) for _ in r
        await asyncio.gather(*tasks)
        print(f'Counter is {counter}')
        assert counter == 100
        counter = 0



asyncio.run(main())
```

In the preceding listing, we create an increment coroutine function that adds one to a global counter, adding a 1-millisecond delay to simulate a slow operation. In our main coroutine, we create 100 tasks to increment the counter and then run them all concurrently with `gather`. We then assert that our counter is the expected value, which, since we ran 100 increment tasks, should always be 100. Running this, you should see that the value we get is always 100 even though incrementing an integer is non-atomic. If we ran multiple threads instead of coroutines, we should see our assertion fail at some point in execution.

Does this mean that with a single-threaded concurrency model we've found a way to completely avoid race conditions? Unfortunately, it's not quite the case. While we avoid race conditions where a single non-atomic operation can cause a bug, we still have the problem where multiple operations executed in the wrong order can cause issues. To see this in action, let's make incrementing an integer in the eyes of asyncio non-atomic.

To do this, we'll replicate what happens under the hood when we increment a global counter. We read the global value, increment it, then write it back. The basic idea is if other code modifies state while our coroutine is suspended on an `await`, once the `await` finishes we may be in an inconsistent state.

Listing 11.2 A single-threaded race condition

```
import asyncio

counter: int = 0


async def increment():
    global counter
    temp_counter = counter
    temp_counter = temp_counter + 1
    await asyncio.sleep(0.01)
    counter = temp_counter


async def main():
    global counter
    for _ in range(1000):
        tasks = [asyncio.create_task(increment()) for _ in r
        await asyncio.gather(*tasks)
        print(f'Counter is {counter}')
        assert counter == 100
        counter = 0


asyncio.run(main())
```

Instead of our increment coroutine directly incrementing the counter, we first read it into a temporary variable and then increment the temporary counter by one. We then `await` `asyncio.sleep` to simulate a slow operation, suspending our coroutine, and only then do we reassign it back to the global counter variable. Running this, you should see this code fail with an assertion error instantly, and our counter only ever gets set to `1`! Each coroutine reads the counter value first, which is `0`, stores it to a temp value, then goes to sleep. Since we're single-threaded, each read to a temporary variable runs sequentially, meaning each coroutine stores the value of counter as `0` and increments this to `1`. Then, once the sleep is finished, every coroutine sets the value of the counter to `1`, meaning despite running 100 coroutines to increment our counter, our counter is only ever `1`. Note that if you remove the `await` expression, things will operate in the correct order because there is no opportunity to modify the application state while we're paused at an `await` point.

This is admittedly a simplistic and somewhat unrealistic example. To better see when this may occur, let's create a slightly more complex race condition. Imagine we're implementing a server that sends messages to connected users. In this server, we keep a dictionary of usernames to sockets we can use to send messages to these users. When a user disconnects, a callback runs that will remove the user from the dictionary and close their socket. Since we close the socket on disconnect, attempting to send any other messages will fail with an exception. What happens if a user disconnects while we're in the process of sending messages? Let's assume the desired behavior is for all users to receive a message if they were connected when we started to send messages.

To test this out, let's implement a mock socket. This mock socket will have a `send` coroutine and a `close` method. Our `send` coroutine will simulate a message send over a slow network. This coroutine will also check a flag to see if we've closed the socket, and if we have it will throw an exception.

We'll then create a dictionary with a few connected users and create mock sockets for each of them. We'll send messages to each user and manually trigger a disconnect for a single user while we're sending messages to see what happens.

Listing 11.3 A race condition with dictionaries

```
import asyncio
```

```
class MockSocket:
    def __init__(self):
        self.socket_closed = False

    async def send(self, msg: str):                             ❶
        if self.socket_closed:
            raise Exception('Socket is closed!')
        print(f'Sending: {msg}')
        await asyncio.sleep(1)
        print(f'Sent: {msg}')

    def close(self):
        self.socket_closed = True


user_names_to_sockets = {'John': MockSocket(),
                         'Terry': MockSocket(),
                         'Graham': MockSocket(),
                         'Eric': MockSocket()}


async def user_disconnect(username: str):                       ❷
    print(f'{username} disconnected!')
    socket = user_names_to_sockets.pop(username)
    socket.close()


async def message_all_users():                                  ❸
    print('Creating message tasks')
    messages = [socket.send(f'Hello {user}')
                for user, socket in
                user_names_to_sockets.items()]
    await asyncio.gather(*messages)


async def main():
    await asyncio.gather(message_all_users(), user_disconnec


asyncio.run(main())
```

❶ Simulate a slow send of a message to a client.

❷ Disconnect a user and remove them from application memory.

❸ Send messages to all users concurrently.

If you run this code, you will see the application crash with the following output:

```
Creating message tasks
Eric disconnected!
Sending: Hello John
Sending: Hello Terry
Sending: Hello Graham
Traceback (most recent call last):
  File 'chapter_11/listing_11_3.py', line 45, in <module>
    asyncio.run(main())
  File "asyncio/runners.py", line 44, in run
    return loop.run_until_complete(main)
  File "python3.9/asyncio/base_events.py", line 642, in run_
    return future.result()
  File 'chapter_11/listing_11_3.py', line 42, in main
    await asyncio.gather(message_all_users(), user_disconnec
  File 'chapter_11/listing_11_3.py', line 37, in message_all
    await asyncio.gather(*messages)
  File 'chapter_11/listing_11_3.py', line 11, in send
    raise Exception('Socket is closed!')
Exception: Socket is closed!
```

In this example, we first create the message tasks, then we `await`, suspending our `message_all_users` coroutine. This gives `user_disconnect('Eric')` a chance to run, which will close Eric's socket and remove it from the `user_names_to_sockets` dictionary. Once this is finished, `message_all_users` resumes; and we start to send out messages. Since Eric's socket was closed, we see an exception, and he won't get the message we intended to send. Note that we also modified the `user_names_to_sockets` dictionary. If we somehow needed to use this dictionary and relied on Eric still being in there, we could potentially have an exception or another bug.

These are the types of bugs you tend to see in a single-threaded concurrency model. You hit a suspension point with `await`, and another coroutine runs and modifies some shared state, changing it for the first coroutine once it resumes in an undesired way. The key difference between multithreaded concurrency bugs and single-threaded concurrency bugs is that in a multithreaded application, race conditions are possible anywhere you modify a mutable state. In a single-threaded concurrency model, you need to modify the mutable state during an `await` point. Now that we understand the types of concurrency bugs in a single-threaded model, let's see how to avoid them by using asyncio locks.

ocks

asyncio locks operate similarly to the locks in the multiprocessing and multithreading modules. We acquire a lock, do work inside of a critical section, and when we’re done, we release the lock, letting other interested parties acquire it. The main difference is that asyncio locks are awaitable objects that suspend coroutine execution when they are blocked. This means that when a coroutine is blocked waiting to acquire a lock, other code can run. In addition, asyncio locks are also asynchronous context managers, and the preferred way to use them is with `async with` syntax.

To get familiar with how locks work, let’s look at a simple example with one lock shared between two coroutines. We’ll acquire the lock, which will prevent other coroutines from running code in the critical section until someone releases it.

Listing 11.4 Using an asyncio lock

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



async def a(lock: Lock):
    print('Coroutine a waiting to acquire the lock')
    async with lock:
        print('Coroutine a is in the critical section')
        await delay(2)
    print('Coroutine a released the lock')
async def b(lock: Lock):
    print('Coroutine b waiting to acquire the lock')
    async with lock:
            print('Coroutine b is in the critical section')
            await delay(2)
        print('Coroutine b released the lock')


async def main():
    lock = Lock()
    await asyncio.gather(a(lock), b(lock))


asyncio.run(main())
```

When we run the preceding listing, we will see that coroutine `a` acquires the lock first, leaving coroutine `b` waiting until `a` releases the lock. Once `a` releases the lock, `b` can do its work in the critical section, giving us the following output:

```
Coroutine a waiting to acquire the lock
Coroutine a is in the critical section
sleeping for 2 second(s)
Coroutine b waiting to acquire the lock
finished sleeping for 2 second(s)
Coroutine a released the lock
Coroutine b is in the critical section
sleeping for 2 second(s)
finished sleeping for 2 second(s)
Coroutine b released the lock
```

Here we used `async` `with` syntax. If we had wanted, we could use the `acquire` and `release` methods on the lock like so:

```
await lock.acquire()
try:
    print('In critical section')
finally:
    lock.release()
```

That said, it is best practice to use `async with` syntax where possible.

One important thing to note is that we created the lock inside of the `main` coroutine. Since the lock is shared globally amongst the coroutines we create, we may be tempted to make it a global variable to avoid passing it in each time like so:

```
lock = Lock()

# coroutine definitions

async def main():
    await asyncio.gather(a(), b())
```

If we do this, we'll quickly see a crash with an error reporting multiple event loops:

```
Task <Task pending name='Task-3' coro=<b()> got Future <Futu
```

Why is this happening when all we've done is move our lock definition? This is a confusing quirk of the asyncio library and is not unique to just locks. Most objects in asyncio provide an optional loop parameter that lets you specify the specific event loop to run in. When this parameter is not provided, asyncio tries to get the currently running event loop, but if there is none, it creates a new one. In the above case, creating a `Lock` creates a new event loop, since when our script first runs we haven't yet created one. Then, `asyncio.run(main())` creates a second event loop, and when we attempt to use our lock we intermingle these two separate event loops, which causes a crash.

This behavior is tricky enough that in Python 3.10, event loop parameters are going to be removed, and this confusing behavior will go away, but until then you'll need to think through these cases when using global asyncio variables carefully.

Now that we know the basics, let's see how to use a lock to solve the bug we had in listing 11.3, where we attempted to send a message to a user whose socket we closed too early. The idea to solve this is to use a lock in two places: first, when a user disconnects and, second, when we send out messages to users. This way, if a disconnect happens while we're sending out messages, we'll wait until they all finish before finally closing any sockets.

Listing 11.5 Using locks to avoid a race condition

```
import asyncio
from asyncio import Lock


class MockSocket:
    def __init__(self):
        self.socket_closed = False

    async def send(self, msg: str):
        if self.socket_closed:
            raise Exception('Socket is closed!')
        print(f'Sending: {msg}')
        await asyncio.sleep(1)
        print(f'Sent: {msg}')

    def close(self):
        self.socket_closed = True


user_names_to_sockets = {'John': MockSocket(),
                         'Terry': MockSocket(),
                         'Graham': MockSocket(),
                         'Eric': MockSocket()}


async def user_disconnect(username: str, user_lock: Lock):
    print(f'{username} disconnected!')
    async with user_lock:                                           ❶
        print(f'Removing {username} from dictionary')
        socket = user_names_to_sockets.pop(username)
        socket.close()


async def message_all_users(user_lock: Lock):
    print('Creating message tasks')
    async with user_lock:                                           ❷
        messages = [socket.send(f'Hello {user}')
                    for user, socket in
                    user_names_to_sockets.items()]
        await asyncio.gather(*messages)


async def main():
    user_lock = Lock()
    await asyncio.gather(message_all_users(user_lock),
                         user_disconnect('Eric', user_lock))


asyncio.run(main())
```

1 Acquire the lock before removing a user and closing the socket.

2 Acquire the lock before sending.

When we run the following listing, we won't see any more crashes, and we'll get the following output:

```
Creating message tasks
Eric disconnected!
Sending: Hello John
Sending: Hello Terry
Sending: Hello Graham
Sending: Hello Eric
Sent: Hello John
Sent: Hello Terry
Sent: Hello Graham
Sent: Hello Eric
Removing Eric from dictionary
```

We first acquire the lock and create the message tasks. While this is happening, Eric disconnects, and the code in disconnect tries to acquire the lock. Since `message_all_users` still holds the lock, we need to wait for it to finish before running the code in disconnect. This lets all the messages finish sending out before closing out the socket, preventing our bug.

You likely won't often need to use locks in asyncio code because many concurrency issues are avoided by its single-threaded nature. Even when race conditions occur, sometimes you can refactor your code such that state isn't modified while a coroutine is suspended (by using immutable objects, for example). When you can't refactor in this way, locks can force modifications to happen in a desired synchronized order. Now that we understand the concepts around avoiding concurrency bugs with locks, let's look at how to use synchronization to implement new functionality in our asyncio applications.

imiting concurrency with semaphores

Resources that our applications need to use are often finite. We may have a limited number of connections we can use concurrently with a database; we may have a limited number of CPUs that we don't want to overload; or we may be working with an API that only allows a few concurrent requests, based on our current subscription pricing. We could also be using our own internal API and may be concerned with overwhelming it with load, effectively launching a distributed denial of service attack against ourselves.

Semaphores are a construct that can help us out in these situations. A semaphore acts much like a lock in that we can acquire it and we can release it, with the major difference being that we can acquire it multiple times up to a limit we specify. Internally, a semaphore keeps track of this limit; each time we acquire the semaphore we decrement the limit, and each time we release the semaphore we increment it. If the count reaches zero, any further attempts to acquire the semaphore will block until someone else calls release and increments the count. To draw parallels to what we just learned with locks, you can think of a lock as a special case of a semaphore with a limit of one.

To see semaphores in action, let's build a simple example where we only want two tasks running at the same time, but we have four tasks to run in total. To do this, we'll create a `semaphore` with a limit of two and `acquire` it in our coroutine.

Listing 11.6 Using semaphores

```
import asyncio
from asyncio import Semaphore


async def operation(semaphore: Semaphore):
    print('Waiting to acquire semaphore...')
    async with semaphore:
        print('Semaphore acquired!')
        await asyncio.sleep(2)
    print('Semaphore released!')


async def main():
    semaphore = Semaphore(2)
    await asyncio.gather(*[operation(semaphore) for _ in ran

asyncio.run(main())
```

In our main coroutine, we create a semaphore with a limit of two, indicating we can acquire it twice before additional acquisition attempts start to block. We then create four concurrent calls to `operation`—this coroutine acquires the semaphore with an `async` `with` block and simulates some blocking work with sleep. When we run this, we'll see the following output:

```
Waiting to acquire semaphore...
Semaphore acquired!
Waiting to acquire semaphore...
Semaphore acquired!
Waiting to acquire semaphore...
Waiting to acquire semaphore...
Semaphore released!
Semaphore released!
Semaphore acquired!
Semaphore acquired!
Semaphore released!
Semaphore released!
```

Since our semaphore only allows two acquisitions before it blocks, our first two tasks successfully acquire the lock while our other two tasks wait for the first two tasks to release the semaphore. Once the work in the first two tasks finishes and we release the semaphore, our other two tasks can acquire the semaphore and start doing their work.

Let's take this pattern and apply it to a real-world use case. Let's imagine you're working for a scrappy, cash-strapped startup, and you've just partnered with a third-party REST API vendor. Their contracts are particularly expensive for unlimited queries, but they offer a plan that allows for only 10 concurrent requests that is more budget-friendly. If you make more than 10 requests concurrently, their API will return a status code of 429 (too many requests). You could send a set of requests and retry if you get a 429, but this is inefficient and places extra load on your vendor's servers, which probably won't make their site reliability engineers happy. A better approach is to create a semaphore with a limit of 10 and then acquire that whenever you make an API request. Using a semaphore when making a request will ensure that you only ever have 10 requests in flight at any given time.

Let's see how to do this with the aiohttp library. We'll make 1,000 requests to an example API but limit the total concurrent requests to 10 with a semaphore. Note that aiohttp has connection limits we can tweak as well, and by default it only allows 100 connections at a time. It is possible achieve the same as below by tweaking this limit.

Listing 11.7 Limiting API requests with semaphores

```
import asyncio
from asyncio import Semaphore
from aiohttp import ClientSession


async def get_url(url: str,
                  session: ClientSession,
                  semaphore: Semaphore):
    print('Waiting to acquire semaphore...')
    async with semaphore:
        print('Acquired semaphore, requesting...')
        response = await session.get(url)
        print('Finished requesting')
        return response.status

async def main():
    semaphore = Semaphore(10)
    async with ClientSession() as session:
        tasks = [get_url('https:/ / www .example .com', sess
                 for _ in range(1000)]
        await asyncio.gather(*tasks)


asyncio.run(main())
```

While output will be nondeterministic depending on external latency factors, you should see output similar to the following:

```
Acquired semaphore, requesting...
Acquired semaphore, requesting...
Acquired semaphore, requesting...
Acquired semaphore, requesting...
Acquired semaphore, requesting...
Finished requesting
Finished requesting
Acquired semaphore, requesting...
Acquired semaphore, requesting...
```

Each time a request finishes, the semaphore is released, meaning a task that is blocked waiting for the semaphore can begin. This means that we only ever have at most 10 requests running at a given time, and when one finishes, we can start a new one.

This solves the issue of having too many requests running concurrently, but the code above is *bursty*, meaning that it has the potential to burst 10 requests at the same moment, creating a potential spike in traffic. This may not be desirable if we're concerned about spikes of load on the API we're calling. If you need to only burst up to a certain number of requests per some unit of time, you'll need to use this with an implementation of a traffic-shaping algorithm, such as the "leaky bucket" or "token bucket."

Bounded semaphores

One aspect of semaphores is that it is valid to call `release` more times than we call `acquire`. If we always use semaphores with an `async` `with` block, this isn't possible, since each `acquire` is automatically paired with a `release`. However, if we're in a situation where we need finer-grained control over our releasing and acquisition mechanisms (for example, perhaps we have some branching code where one branch lets us release earlier than another), we can run into issues. As an example, let's see what happens when we have a normal coroutine that acquires and releases a semaphore with an `async` `with` block, and while that coroutine is executing another coroutine calls `release`.

Listing 11.8 Releasing more than we acquire

```
import asyncio
from asyncio import Semaphore

async def acquire(semaphore: Semaphore):
    print('Waiting to acquire')
    async with semaphore:
        print('Acquired')
        await asyncio.sleep(5)
    print('Releasing')
```

```
async def release(semaphore: Semaphore):
    print('Releasing as a one off!')
    semaphore.release()
    print('Released as a one off!')


async def main():
    semaphore = Semaphore(2)

    print("Acquiring twice, releasing three times...")
    await asyncio.gather(acquire(semaphore),
                         acquire(semaphore),
                         release(semaphore))

    print("Acquiring three times...")
    await asyncio.gather(acquire(semaphore),
                         acquire(semaphore),
                         acquire(semaphore))


asyncio.run(main())
```

In the preceding listing, we create a `semaphore` with two permits. We then run two calls to `acquire` and one call to `release`, meaning we'll call release three times. Our first call to gather seems to run okay, giving us the following output:

```
Acquiring twice, releasing three times...
Waiting to acquire
Acquired
Waiting to acquire
```

```
Acquired
Releasing as a one off!
Released as a one off!
Releasing
Releasing
```

However, our second call where we acquire the semaphore three times runs into issues, and we acquire the lock three times at once! We’ve inadvertently increased the number of permits our semaphore has available:

```
Acquiring three times...
Waiting to acquire
Acquired
Waiting to acquire
Acquired
Waiting to acquire
Acquired
Releasing
Releasing
Releasing
```

To deal with these types of situations, asyncio provides a `BoundedSemaphore`. This semaphore behaves exactly as the semaphore we’ve been using, with the key difference being that release will throw a `ValueError: BoundedSemaphore released too many times` exception if we call `release` such that it would change the available permits. Let’s look at a very simple example in the following listing.

Listing 11.9 Bounded semaphores

```
import asyncio
from asyncio import BoundedSemaphore


async def main():
    semaphore = BoundedSemaphore(1)

    await semaphore.acquire()
    semaphore.release()
    semaphore.release()


asyncio.run(main())
```

When we run the preceding listing, our second call to release will throw a `ValueError` indicating we've released the semaphore too many times. You'll see similar results if you change the code in listing 11.8 to use a `BoundedSemaphore` instead of a `Semaphore`. If you're manually calling `acquire` and `release` such that dynamically increasing the number of permits your semaphore has available would be an error, it is wise to use a `BoundedSemaphore`, so you'll see an exception to warn you of the mistake.

We've now seen how to use semaphores to limit concurrency, which can be useful in situations where we need to constrain concurrency within our applications. asyncio synchronization primitives not only allow us to limit concurrency but also allow us to notify tasks when something happens. Next, let's see how to do this with the `Event` synchronization primitive.

otifying tasks with events

Sometimes, we may need to wait for some external event to happen before we can proceed. We might need to wait for a buffer to fill up before we can begin to process it, we might need to wait for a device to connect to our application, or we may need to wait for some initialization to happen. We may also have multiple tasks waiting to process data that may not yet be available. `Event` objects provide a mechanism to help us out in situations where we want to idle while waiting for something specific to happen.

Internally, the `Event` class keeps track of a flag that indicates whether the event has happened yet. We can control this flag is with two methods, `set` and `clear`. The `set` method sets this internal flag to `True` and notifies anyone waiting that the event happened. The `clear` method sets this internal flag to `False`, and anyone who is waiting for the event will now block.

With these two methods, we can manage internal state, but how do we block until an event happens? The `Event` class has one coroutine method named `wait`. When we `await` this coroutine, it will block until someone calls `set` on the event object. Once this occurs, any additional calls to `wait` will not block and will return instantly. If we call `clear` once we have called `set`, then calls to `wait` will start blocking again until we call `set` again.

Let’s create a dummy example to see events in action. We’ll pretend we have two tasks that are dependent on something happening. We’ll have these tasks wait and idle until we trigger the event.

Listing 11.10 Event basics