

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 a chat server and chat client—something even more advanced.

## **eating 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 is a chat server that accepts multiple chat clients at the same time, and the second is a chat client that connects to the server and sends and receives chat messages.

Before we begin designing our application, let's start with some 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 when they provide a username.
2. Once a user is connected, they should be able to send chat messages to the server, and 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 one minute, the server should disconnect them.

Second, for our client:

1. When a user starts the application, the client should prompt 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 presses 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 that 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; as needed, you could 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 that 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 provide it, 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):
        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, closing connection')
            writer.close()
            await writer.wait_closed()
```

```

def _add_user(self, username: str, reader:
    StreamReader, writer: StreamWriter):
    self._username_to_writer[username] = writer
    asyncio.create_task(self._listen_for_messages(username))

async def _on_connect(self, username: str, writer: Stream
    writer.write(f'Welcome! {len(self._username_to_write
    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,

async def _listen_for_messages(self,
                                username: str,
                                reader: StreamReader):
    try:
        while (data := await asyncio.wait_for(reader.rea
            await self._notify_all(f'{username}: {data.d
            await self._notify_all(f'{username} has left the
    except Exception as e:
        logging.exception('Error reading from client.',
        await self._remove_user(username)

async def _notify_all(self, message: str):
    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')
                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('127.0.0.1', 8000)

    asyncio.run(main())

```

- ❶ Wait for the client to provide a valid username command; otherwise, disconnect them.
- ❷ Store a user's stream writer instance and create a task to listen for messages.
- ❸ Once a user connects, notify all others that they have connected.
- ❹ Listen for messages from a client and send them to all other clients, waiting a maximum of a minute for a message.
- ❺ Send a message to all connected clients, removing any disconnected users.

Our `ChatServer` 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, and 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 stop sending messages to them.

We now have a complete server, but the 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 presses 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):
    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):
    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('127.0.0.1', 8080)

    writer.write(f'CONNECT {username}\n'.encode())
    await writer.drain()

    message_listener = asyncio.create_task(listen_for_message())
    input_listener = asyncio.create_task(read_and_send(stdin_reader, writer))

```



```
try:
    await asyncio.wait([message_listener, input_listener])
except Exception as e:
    logging.exception(e)
    writer.close()
    await writer.wait_closed()

asyncio.run(main())
```

- 1 Listen for messages from the server, appending them to the message store.
- 2 Read input from the user, and send it to the server.
- 3 Open a connection to the server, and send the connect message with the username.
- 4 Create a task to listen for messages, and listen for input; wait until one finishes.

We first ask the user for their username, and 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 continuously 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 `await` ed each task independently, we may 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 because 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 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 might be a fun exercise for the reader, as there are many 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.

## ary

- We've 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 async 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 perform 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 the 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 look at using 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.

## eating a REST API with aiohttp

Previously, we used aiohttp as a HTTP client to make thousands of concurrent web requests to web applications. aiohttp has not only support as a HTTP client but also has functionality to create asyncio-ready web application servers as well.

### 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 it could potentially be a 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` of 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 PhD dissertation is a good place to learn about the concepts. It is available at <http://mng.bz/1jAg>.

## iohttp server basics

Let's 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.

aiohhttp 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 to the client.

Creating these endpoints by themselves does nothing, however, and 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')
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) ❷

app = web.Application() ❸
app.add_routes(routes)
web.run_app(app)
```

- ❶ Create a time GET endpoint; when a client calls this endpoint, the time coroutine will run.
- ❷ Take the result dictionary, and turn it into a JSON response.
- ❸ Create the web application, register the routes, and run the application.

In the preceding listing, 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`, which 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 that 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 soon become important. 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, and you should see something similar to this:

```
<CIMultiDictProxy('Host': 'localhost:8080', 'User-Agent': 'c
```

## 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 e-commerce storefront database from chapter 5. 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 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 first 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 as follows:

```
app = web.Application()
app['shared_dict'] = {'key' : 'value'}
```

We can now access the shared dictionary by executing `app['shared_dict']`. Next, 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 through 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. Now we decide 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, and 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: `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):
    print('Creating database pool.')
    pool: Pool = await asyncpg.create_pool(host='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='product',
                                           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()

@routes.get('/brands')
async def brands(request: Request) -> Response:
    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)
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)

```

- ❶ Create the database pool, and store it in the application instance.
- ❷ Destroy the pool in the application instance.
- ❸ Query all brands and return results to the client.

#### 4 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, "bran
```

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 1 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 into 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']
        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, pr
```

```
if result is not None:
    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='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='product',
                                           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)
```

- ❶ Get the `product_id` parameter from the URL.
- ❷ Run the query for a single product.
- ❸ 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, des

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(DEFAU
                                                    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}' 1
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.

## 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 fewer 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 greater 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, although 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==2.0.1
pip install -Iv psycopg2==2.9.1
```

For psycopg, you may run into compile errors on install. If you do, you may need to install Postgres tools, and open SSL or another library. A web 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"
db = psycopg2.connect(conn_info)

@app.route('/brands')
def brands():
    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]
```

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 it 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==20.1.0
```

We'll be testing this out on an 8-core machine, so we'll spawn eight workers with Gunicorn. Running `gunicorn -w 8 chapter_09.listing_9_5:app`, and you should see eight workers start up:

```
[2020-11-24 09:53:39 -0500] [16454] [INFO] Starting gunicorn
[2020-11-24 09:53:39 -0500] [16454] [INFO] Listening at: htt
[2020-11-24 09:53:39 -0500] [16454] [INFO] Using worker: syn
[2020-11-24 09:53:39 -0500] [16458] [INFO] Booting worker wi
[2020-11-24 09:53:39 -0500] [16459] [INFO] Booting worker wi
[2020-11-24 09:53:39 -0500] [16460] [INFO] Booting worker wi
[2020-11-24 09:53:39 -0500] [16461] [INFO] Booting worker wi
[2020-11-24 09:53:40 -0500] [16463] [INFO] Booting worker wi
[2020-11-24 09:53:40 -0500] [16464] [INFO] Booting worker wi
[2020-11-24 09:53:40 -0500] [16465] [INFO] Booting worker wi
[2020-11-24 09:53:40 -0500] [16468] [INFO] Booting worker wi
```

This means we have created eight connections to our database and can serve eight 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>.

Let's start by running a 30-second load test on our Flask server. We'll use one thread and 200 connections, simulating 200 concurrent users hitting our 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 1,500 requests per second, which is about three times what we were able to do with Flask. More importantly, we did this with only one process, where Flask needed a total of *eight 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.

## **e 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, 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 internet space but already has several popular implementations and frameworks that support it, including Django.

### **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 received formal acceptance into the Python ecosystem in 2004 with the acceptance of PEP-333 (Python enhancement proposal; <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 with 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 WebSockets. 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'}
```

An ASGI application function has three parameters: a scope dictionary, a receive coroutine, and a send coroutine, which allow us to 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, as we'll learn more in chapter 14) and HTTP parsing. We can install Uvicorn by running the following:

```
pip install -Iv uvicorn==0.14.0
```

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 accepted, and as of this writing it is still relatively 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 look at a popular one.

## GI 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 see how to implement simple REST and WebSocket endpoints using it. To get started, let's first install it with the following command:

```
pip install -Iv starlette==0.15.0
```

### 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='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='product',
                                           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 eight workers, as we did before, with the following command:

```
uvicorn --workers 8 --log-level error chapter_09.listing_9_8
```

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 over 4,000 requests per second, outperforming Flask and even aiohttp by a wide margin! Since we only ran one aiohttp worker process earlier, this isn't exactly a fair comparison (we'd get similar numbers with eight aiohttp workers behind NGINX), but this shows the throughput power that async frameworks offer.

## WebSockets 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 this over HTTP with some JavaScript that polls an endpoint, telling 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 drawbacks. The main drawback is that we're creating an extra load on our web server, each request and response cycle taking time and resources. This is especially egregious because our user count might not change between requests, causing a strain on our system for no 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 repeatedly asking, "Are we there yet?"

WebSockets 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 WebSocket the server can just *tell* us when there is a new user count. As shown in figure 9.1, we don't need to ask repeatedly, creating extra load and potentially receiving data that isn't new.

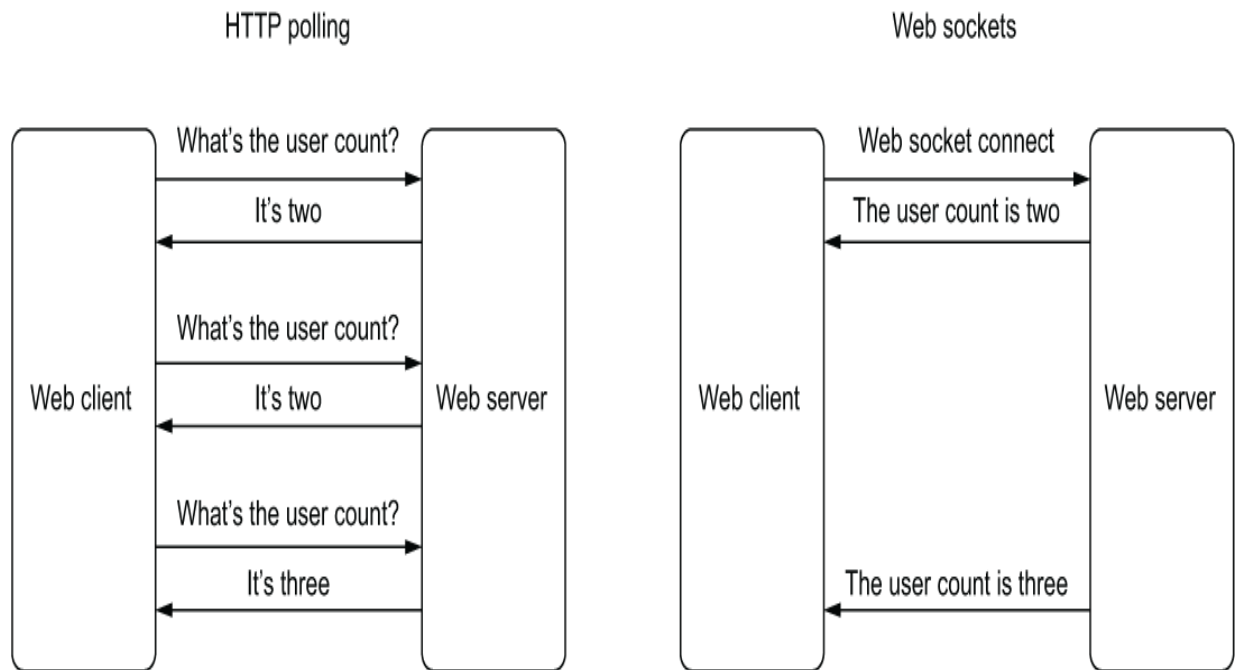


Figure 9.1 HTTP polling to retrieve data compared to WebSockets

Starlette provides out-of-the-box support for WebSockets using an easy-to-understand interface. To see this in action, we'll build a simple WebSocket 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==9.1
```

Next, we'll need to implement our WebSocket endpoint. Our game plan will be to keep an in-memory list of all connected client WebSockets. When a new client connects, we'll add them to the list and send the new count of users 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 WebSocket connection. This class has a few coroutines we'll need to implement. The first is `on_connect`, which gets fired when a client connects to our socket. In `on_connect`, we'll store the client's WebSocket 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 WebSockets and update other connected clients with the latest user count.

#### Listing 9.9 A Starlette WebSocket 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):
        await websocket.accept()
        UserCounter.sockets.append(websocket)
        await self._send_count()

    async def on_disconnect(self, websocket, close_code):
        UserCounter.sockets.remove(websocket)
```



```

        await self._send_count()

    async def on_receive(self, websocket, data):
        pass

    async def _send_count(self):
        if len(UserCounter.sockets) > 0:
            count_str = str(len(UserCounter.sockets))
            task_to_socket = {asyncio.create_task(websocket.send(
                count_str)) for websocket
                             in UserCounter.sockets}

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

            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)])

```

- ❶ When a client connects, add it to the list of sockets and notify other users of the new count.
- ❷ When a client disconnects, remove it from the list of sockets and notify other users of the new count.
- ❸ 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 WebSocket. We'll add create a basic script to connect to our WebSocket 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

    socket.onmessage = (event) => {
      const counter = document.querySelector("#cou
      counter.textContent = event.data;
    };
  });
</script>
</head>
<body>
  <span>Users online: </span>
  <span id="counter"></span>
</body>
</html>
```

In the preceding listing, the 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 which 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 increment on all the tabs. 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 and WebSocket endpoints. While these frameworks are popular, they are not close in popularity to Django, the 1,000-pound gorilla of Python web frameworks.

## ango asynchronous views

Django is one of the most popular and widely used Python frameworks. It has a wealth of functionality out of the box, from an ORM (object relational mapper) to handle databases to a customizable admin console. Until version 3.0, Django applications supported deploying as a WSGI application alone and had little support for asyncio outside of the `channels` library. Version 3.0 introduced support for ASGI and began the process of making Django fully asynchronous. 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 the overall feature set is still lacking (for example, the ORM is entirely synchronous, but supporting async is in the future). 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 that 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 have. We'll build a view that takes in a URL and request count as query parameters and calls out to this URL and aggregates the results, returning them in a tabular format.

Let's get started by ensuring that we have the appropriate version of Django installed:

```
pip install -Iv django==3.2.8
```

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 that 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.Uvi
```

Then, when you go to `localhost:8000`, you should see the Django welcome page (figure 9.2).

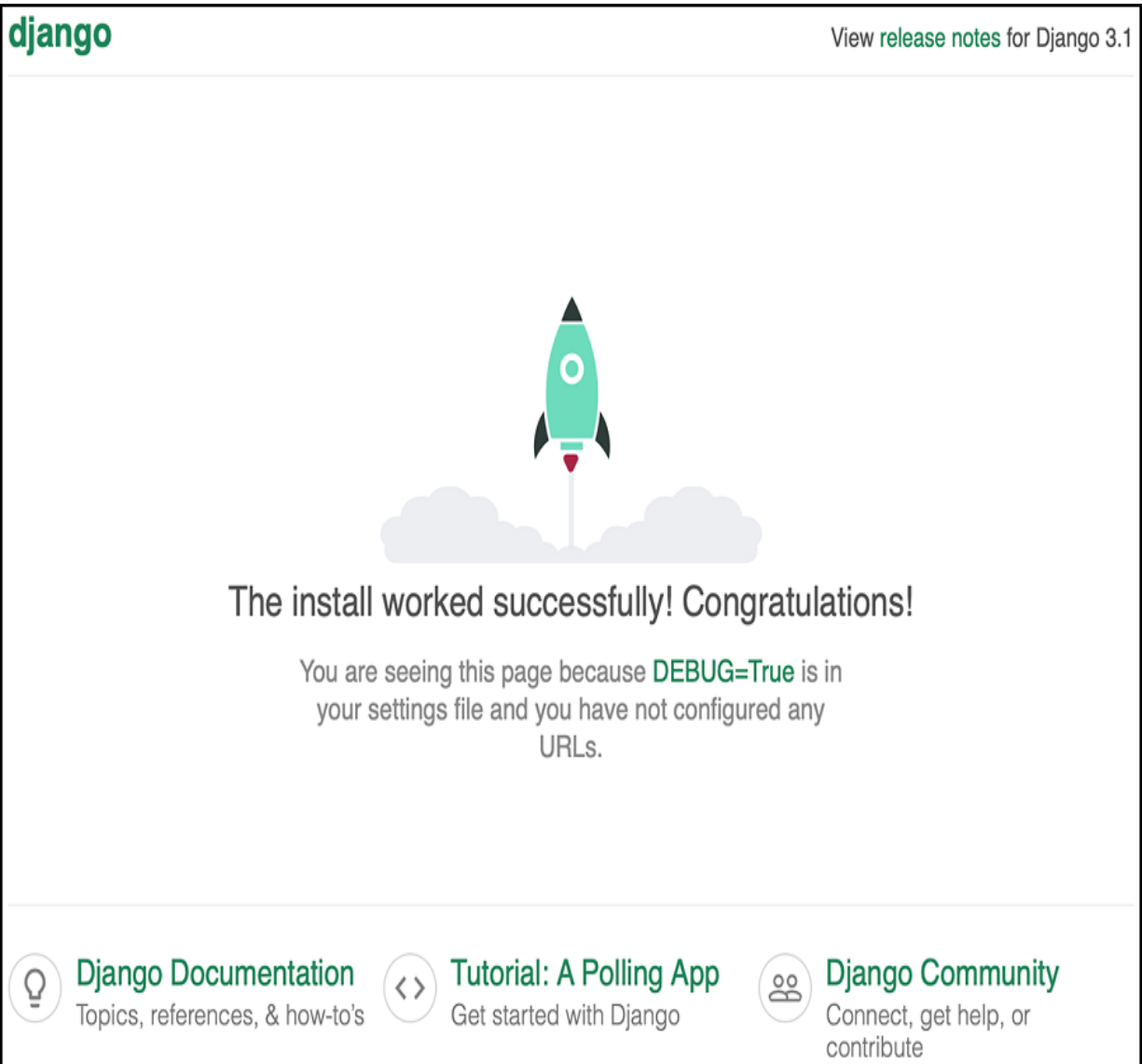


Figure 9.2 The Django welcome page

Next, we'll need to create our app, which we'll call `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 HTTP requests concurrently and display their status codes and other data in an 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 the preceding listing, we first create a coroutine to make a request and return a dictionary of the response status, the total time of the 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 passing in `async_views/requests.html` only for right 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 web page 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 this file, you'll need to modify the `urlpatterns` list to reference `async_api`, and once done this should look like the following:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
```

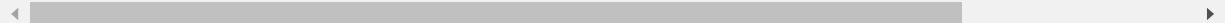
```
    path('requests/', include('async_api.urls'))  
]
```

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:

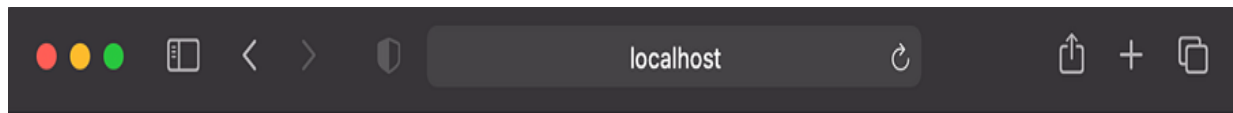
```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'async_api'  
]
```

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&r
```



While numbers will differ on your machine, you should see a page like that shown in figure 9.3 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 can make 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.

## Running blocking work in an asynchronous view

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 7, 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. 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 (in other words, 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']
    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 5 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 5 seconds to complete since we're running multiple threads. You'll also notice if you hit this URL more than once that each request still

takes only 5 seconds, indicating the requests aren't blocking each other. Now, let's change the `thread_sensitive url` parameter to `True`, and you'll see quite different behavior. First, the view will take 25 seconds to return since it is making five 5-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 several options to use existing code with async views, but you need to be aware of the thread-sensitivity of what you're running, as well as the limitations that this can place on async performance benefits.

### Using async code in synchronous views

The next logical question is, “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 no 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, while 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'])
```