Hands-on Intro to aiohttp (PyCon tutorial)

Mariatta Wijaya, Andrew Svetlov

May 18, 2021

CONTENTS

Asyncio is a relatively new feature in Python, with the `async` and `await` syntax only recently became proper keywords in Python 3.7. Asyncio allows you to write asynchronous programs in Python. In this tutorial, we'll introduce you to an asyncio web library called aiohttp.

aiohttp is a library for building web client and server using Python and asyncio. We'll introduce you to several key features of aiohttp; including routing, session handling, templating, using middlewares, connecting to database, and making HTTP GET/POST requests.

We'll use all new Python 3.7 features to build web services with asyncio and aiohttp.

This tutorial is for PyCon US 2019 in Cleveland, Ohio. Video recording will be posted once available.

If you have any feedback or questions about this tutorial, please file an issue.

CHAPTER

ONE

CODE OF CONDUCT

Be open, considerate, and respectful.

CHAPTER

TWO

LICENSE

CHAPTER

THREE

AGENDA

3.1 Preparation

Before coming to the tutorial, please do the following:

1. Have Python 3.7 installed in your laptop. **Note:** It's important that you're running Python 3.7 (or above) because we will use some of the new features added in 3.7 in this tutorial.
    - Here's a tutorial that will help you get set up.
2. Install the dependencies, preferably in a virtual environment. For this tutorial, we'll use `venv`.

Create a new virtual environment using venv:

```
python3.7 -m venv .venv
```

Activate the virtual environment. On Unix, Mac OS:

```
source .venv/bin/activate
```

On Windows:

```
.venv\Scripts\activate.bat
```

Install the dependencies, listed in the requirements.txt file. You can download the file, or clone this repository:

```
(.venv) python -m pip install -U pip -r requirements.txt
```

3. Verify that you correctly have Python 3.7 installed. If you're able to run the following code, then you're good to go.

```
import asyncio


async def main() -> None:
    print("Hello ...")
    await asyncio.sleep(1)
    print("... World!")


# Python 3.7+
asyncio.run(main())
```

3.1.1 Resources and documentation links

Tools and documentations that we'll use throughout this tutorial.

venv

Python venv tutorial documentation.

aiohttp

- Installation: `python3.7 -m pip install aiohttp`.
- aiohttp documentation
- *aiohttp* source code
- Owner: Andrew Svetlov

f-strings

We will use some f-strings during this tutorial.

My talk about f-strings.

Example:

```
first_name = "bart"
last_name = "simpson"

# old style %-formatting
print("Hello %s %s" % (first_name, last_name))

# str.format
print("Hello {first_name} {last_name}".format(first_name=first_name, last_name=last_
↪name))

# f-string
print(f"Hello {first_name} {last_name}")
```

asyncio

aiohttp is async Python library. Read up the quick intro to asyncio.

type annotations

Our code examples use type annotations and checked with `mypy`. It is optional.

- typing documentation: https://docs.python.org/3/library/typing.html
- mypy documentation: https://mypy.readthedocs.io/en/stable/

dataclass

The `dataclass` module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in PEP 557.

Example:

```
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

dataclass documentation.

aiohttp-jinja2

jinja2 template renderer for aiohttp.web.

aiohttp-jinja2 documentation.

aiohttp-session

https://aiohttp-session.readthedocs.io/en/stable/

click

Click is a Python package for creating beautiful command line interfaces in a composable way with as little code as necessary. It's the "Command Line Interface Creation Kit".

click documentation.

3.2 Intro to asyncio

What does it mean for something to be **asynchronous**?

It means that being able to do multiple things at once. The `asyncio` library in Python provides the framework to do this.

Consider a scenario where you have a long running task, which you'd like to perform multiple times. In a traditional synchronous programming, you'd execute one task after another. If one task is taking 10 seconds to complete, running the task 6 times will take 1 full minute.

Here's a simple code to demonstrate that:

```
import time


def long_running_task(time_to_sleep: int) -> None:
    print(f"Begin sleep for {time_to_sleep}")
    time.sleep(time_to_sleep)
    print(f"Awake from {time_to_sleep}")


def main() -> None:
    long_running_task(2)
    long_running_task(10)
    long_running_task(5)


if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"Execution time: {elapsed:0.2f} seconds.")
```

In the above example, we wanted to run `long_running_task` 3 times, each with varying time to complete.

When the code run, we can see the following output:

```
Begin sleep for 2
Awake from 2
Begin sleep for 10
Awake from 10
Begin sleep for 5
Awake from 5
Execution time: 17.01 seconds.
```

The above was an example of **synchronous** execution, the `long_running_task` was executed one at a time, and each time we're waiting for it to complete before starting another one.

Now let's take a look on how it would look like if we're running it **asynchronously** using asyncio.

First we convert the `long_running_task` into a coroutine, by adding the keyword `async`:

```
import asyncio


```

(continues on next page)

(continued from previous page)

```
async def long_running_task(time_to_sleep):
    print(f"Begin sleep for {time_to_sleep}")
    await asyncio.sleep(time_to_sleep)
    print(f"Awake from {time_to_sleep}")
```

Note that coroutine cannot simply be called like regular functions. For example, you can type the following:

```
>>> long_running_task()
<coroutine object task at 0x1016dff40>
```

But the code was not executed. It does not print out `Begin sleep ..` or `Awake from`.

To actually execute the coroutine, you have three options:

- using `asyncio.run()`

```
asyncio.run(long_running_task(3))
```

- **await**-ing the coroutine

```
async def main():
    await long_running_task(3)

asyncio.run(main())
```

- using `asyncio.create_task()`

```
async def main():
    task = asyncio.create_task(long_running_task(3))

    await task

asyncio.run(main())
```

Suppose now we want to execute `long_running_task` three times **asynchronously**:

```
import asyncio
import time


async def long_running_task(time_to_sleep: int) -> None:
    print(f"Begin sleep for {time_to_sleep}")
    await asyncio.sleep(time_to_sleep)
    print(f"Awake from {time_to_sleep}")


async def main() -> None:
    task1 = asyncio.create_task(long_running_task(2))
    task2 = asyncio.create_task(long_running_task(10))
    task3 = asyncio.create_task(long_running_task(5))
    await asyncio.gather(task1, task2, task3)


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"Execution time: {elapsed:0.2f} seconds.")
```

The output:

```
Begin sleep for 2
Begin sleep for 10
Begin sleep for 5
Awake from 2
Awake from 5
Awake from 10
Execution time: 10.01 seconds.
```

Notice how the tasks are all starting at about the same time, and that the third (5) task was completed before the second one (10) was finished. The total time taken was faster compared to when it was run synchronously.

3.3 Intro to aiohttp

In the previous section, we've got a taste of what **asynchronous** task execution looks like using `asyncio` library.

The example before was quite simple, and perhaps not very exciting. Let's now try building something bigger, like our own web scraper. A web scraper is a way to extract data from websites. It is a way to copy/download data programmatically.

For demonstration purpose, we'll try downloading some PEPs and save them to our local machine for further analysis.

The PEPs we want to download are the governance PEPs:

- PEP 8010: https://www.python.org/dev/peps/pep-8010/
- PEP 8011: https://www.python.org/dev/peps/pep-8011/
- PEP 8012: https://www.python.org/dev/peps/pep-8012/
- PEP 8013: https://www.python.org/dev/peps/pep-8013/
- PEP 8014: https://www.python.org/dev/peps/pep-8014/
- PEP 8015: https://www.python.org/dev/peps/pep-8015/
- PEP 8016: https://www.python.org/dev/peps/pep-8016/

3.3.1 Downloading contents synchronously

First, let us try doing this **synchronously** using the requests library. It can be installed using pip.

```
python3.7 -m pip install requests
```

Downloading an online resource using requests is straightforward.

```
import requests

```

(continues on next page)

(continued from previous page)

```
response = requests.get("https://www.python.org/dev/peps/pep-8010/")
print(response.content)
```

It will print out the HTML content of PEP 8010. To save it locally to a file:

```
filename = "sync_pep_8010.html"

with open(filename, "wb") as pep_file:
    pep_file.write(content.encode('utf-8'))
```

The file `sync_pep_8010.html` will be created.

3.3.2 Exercise

Let's take the next 10-15 minutes to write a script that will programmatically download PEPs 8010 to 8016 using requests library.

3.3.3 Solution

Here is an example solution. It is ok if yours do not look exactly the same.

```
import time

import requests


def download_pep(pep_number: int) -> bytes:

    url = f"https://www.python.org/dev/peps/pep-{pep_number}/"
    print(f"Begin downloading {url}")
    response = requests.get(url)
    print(f"Finished downloading {url}")
    return response.content


def write_to_file(pep_number: int, content: bytes) -> None:

    filename = f"sync_{pep_number}.html"

    with open(filename, "wb") as pep_file:
        print(f"Begin writing to {filename}")
        pep_file.write(content)
        print(f"Finished writing {filename}")


if __name__ == "__main__":
    s = time.perf_counter()

    for i in range(8010, 8016):
        content = download_pep(i)
        write_to_file(i, content)
```

(continues on next page)

(continued from previous page)

```
    elapsed = time.perf_counter() - s
    print(f"Execution time: {elapsed:0.2f} seconds.")

# Begin downloading https://www.python.org/dev/peps/pep-8010/
# Finished downloading https://www.python.org/dev/peps/pep-8010/
# Begin writing to 8010.html
# Finished writing 8010.html
# Begin downloading https://www.python.org/dev/peps/pep-8011/
# Finished downloading https://www.python.org/dev/peps/pep-8011/
# Begin writing to 8011.html
# Finished writing 8011.html
# Begin downloading https://www.python.org/dev/peps/pep-8012/
# Finished downloading https://www.python.org/dev/peps/pep-8012/
# Begin writing to 8012.html
# Finished writing 8012.html
# Begin downloading https://www.python.org/dev/peps/pep-8013/
# Finished downloading https://www.python.org/dev/peps/pep-8013/
# Begin writing to 8013.html
# Finished writing 8013.html
# Begin downloading https://www.python.org/dev/peps/pep-8014/
# Finished downloading https://www.python.org/dev/peps/pep-8014/
# Begin writing to 8014.html
# Finished writing 8014.html
# Begin downloading https://www.python.org/dev/peps/pep-8015/
# Finished downloading https://www.python.org/dev/peps/pep-8015/
# Begin writing to 8015.html
# Finished writing 8015.html
# Execution time: 3.60 seconds.
```

In the above solution, we're downloading the PEP one at a time. From previous section, we know that using `asyncio`, we can run the same task **asynchronously**. requests itself is not an asyncio library. Enter aiohttp.

3.3.4 Downloading contents asynchronously

Install aiohttp if you have not already:

```
python3.7 -m pip install aiohttp
```

Here's an example of downloading an online resource using aiohttp.

```
import asyncio
import aiohttp

async def download_pep(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            content = await resp.read()
            print(content)
            return content

asyncio.run(download_pep("https://www.python.org/dev/peps/pep-8010/"))
```

Writing the downloaded content to a new file can be done as its own coroutine.

```
async def write_to_file(pep_number, content):
    filename = f"async_{pep_number}.html"
    with open(filename, "wb") as pep_file:
        pep_file.write(content)
```

Since we now have two coroutines, we can execute them like so:

```
async def web_scrape_task(pep_number):
    url = f"https://www.python.org/dev/peps/pep-{pep_number}/"

    downloaded_content = await download_pep(url)
    await write_to_file(pep_number, downloaded_content)


asyncio.run(web_scrape_task(8010))
```

3.3.5 Exercise

The code is looking more complex than when we're doing it synchronously, using requests. But you got this. Now that you know how to download an online resource using aiohttp, now you can download multiple pages asynchronously.

Let's take the next 10-15 minutes to write the script for downloading PEPs 8010 - 8016 using aiohttp.

3.3.6 Solution

Here is an example solution. It is ok if yours do not look exactly the same.

```
import asyncio
import time

import aiohttp


async def download_pep(pep_number: int) -> bytes:

    url = f"https://www.python.org/dev/peps/pep-{pep_number}/"
    print(f"Begin downloading {url}")
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            content = await resp.read()
            print(f"Finished downloading {url}")
            return content


async def write_to_file(pep_number: int, content: bytes) -> None:
    filename = f"async_{pep_number}.html"
    with open(filename, "wb") as pep_file:
        print(f"Begin writing to {filename}")
        pep_file.write(content)
        print(f"Finished writing {filename}")
```

(continues on next page)

(continued from previous page)

```
async def web_scrape_task(pep_number: int) -> None:
    content = await download_pep(pep_number)
    await write_to_file(pep_number, content)


async def main() -> None:
    tasks = []
    for i in range(8010, 8016):
        tasks.append(web_scrape_task(i))
    await asyncio.wait(tasks)


if __name__ == "__main__":
    s = time.perf_counter()

    asyncio.run(main())

    elapsed = time.perf_counter() - s
    print(f"Execution time: {elapsed:0.2f} seconds.")


# Begin downloading https://www.python.org/dev/peps/pep-8010/
# Begin downloading https://www.python.org/dev/peps/pep-8015/
# Begin downloading https://www.python.org/dev/peps/pep-8012/
# Begin downloading https://www.python.org/dev/peps/pep-8013/
# Begin downloading https://www.python.org/dev/peps/pep-8014/
# Begin downloading https://www.python.org/dev/peps/pep-8011/
# Finished downloading https://www.python.org/dev/peps/pep-8014/
# Begin writing to async_8014.html
# Finished writing async_8014.html
# Finished downloading https://www.python.org/dev/peps/pep-8012/
# Begin writing to async_8012.html
# Finished writing async_8012.html
# Finished downloading https://www.python.org/dev/peps/pep-8013/
# Begin writing to async_8013.html
# Finished writing async_8013.html
# Finished downloading https://www.python.org/dev/peps/pep-8010/
# Begin writing to async_8010.html
# Finished writing async_8010.html
# Finished downloading https://www.python.org/dev/peps/pep-8011/
# Begin writing to async_8011.html
# Finished writing async_8011.html
# Finished downloading https://www.python.org/dev/peps/pep-8015/
# Begin writing to async_8015.html
# Finished writing async_8015.html
# Execution time: 0.87 seconds.
```

While the code looks longer and more complex than our solution using requests, by executing the code asynchronously, the task is taking less time to complete.

3.3.7 Why aiohttp

- Web frameworks like Django and Flask don't support `asyncio`.
- aiohttp provides the framework for both web Server and Client. For example, Django is mainly the framework you'd use if you need a server, and you'll use it in conjuction with requests.

We're not advocating for you to replace your existing web application with aiohttp. Each framework comes with their own benefits. Our goal in this tutorial is to learn something new together, and be comfortable working with asyncio.

3.4 aiohttp Server

Let's start writing our own web application using aiohttp server.

Install aiohttp and Python 3.7 if you have not already. Using a virtual environment is recommended.

Example using `venv`. In the command line:

```
python3.7 -m venv .venv
source .venv/bin/activate

(.venv) python3.7 -m pip install -U pip aiohttp
```

3.4.1 A simple web server example

First we will define a request handler, and it is a coroutine.

```
from aiohttp import web

async def handler(request):
    return web.Response(text="Hello world")
```

The above is a coroutine that will return a Web Response, containing the text `Hello world`.

Once we have the request handler, create an Application instance:

```
app = web.Application()
app.add_routes([web.get('/', handler)])
```

In the above snippet, we're creating an aiohttp web application, and registering a URL route: `/` (usually the root of the app), and we're telling the app to execute the `handler` coroutine.

To run the app:

```
web.run_app(app)
```

Run the script, e.g. in the command line:

```
(.venv) python server.py
```

You should see the following output in the command line:

```
======== Running on http://0.0.0.0:8080 ========
(Press CTRL+C to quit)
```

You can now open your favorite web browser with the url http://0.0.0.0:8080. It should display "Hello world".

The complete code can look like the following:

```
from aiohttp import web


async def handler(request: web.Request) -> web.Response:
    return web.Response(text="Hello world")


async def init_app() -> web.Application:
    app = web.Application()
    app.add_routes([web.get("/", handler)])
    return app


web.run_app(init_app())
```

3.4.2 Using route decorators

Notice in the previous example, we're registering the root url by calling `app.add_routes`, and passing it a list of one URL. If you have more than one URL, it is a matter of creating another request handler coroutine.

There is another way to define urls, by using decorators.

```
routes = web.RouteTableDef()

@routes.get('/')
async def handler(request):
    return web.Response(text="Hello world")

app.add_routes(routes)
```

Both ways work, and it is a matter of your own personal choice.

3.4.3 URL Arguments and query parameters

When building web applications, sometimes you need to handle dynamic urls, instead of hardcoding.

Suppose we want to have urls for each users in our system, and greet the user, by having the `/{username}/` url.

```
@routes.get('/{username}')
async def greet_user(request: web.Request) -> web.Response:
    user = request.match_info.get("username", "")
    return web.Response(text=f"Hello, {user}")
```

Now re-start the server and open http://0.0.0.0:8080/<student> in your favorite browser.

Another way to parametrize resource is by using query parameters, for example `?page=1&uppercase=true`.

```
@routes.get('/{username}')
async def greet_user(request: web.Request) -> web.Response:
    user = request.match_info.get("username", "")
```

(continues on next page)

(continued from previous page)

```
    page_num = request.rel_url.query.get("page", "")

    return web.Response(text=f"Hello, {user} {page_num}")
```

Now try going to http://0.0.0.0:8080/<student>/?page=<pagenum>.

3.4.4 Serving other methods (POST, PUT, etc)

Notice so far we've been serving `GET` resources. If you have resource that needs to be accessed in other methods, like `POST` or `PUT`:

```
@routes.post('/add_user')
async def add_user(request: web.Request) -> web.Response:
    data = await request.post()
    username = data.get('username')
    # Add the user
    # ...
    return web.Response(text=f"{username} was added")
```

3.4.5 Working with JSON

```
@routes.get('/json')
async def handler(request):
    args = await request.json()
    data = {'value': args['key']}
    return web.json_response(data)
```

3.4.6 Application's Shared State

Every server has a state, e.g. database connection.

Initialize DB connection:

```
async def init_app() -> web.Application:
    app = web.Application()
    app.add_routes(...)
    app.cleanup_ctx.append(init_db)
    return app

async def init_db(app: web.Application) -> AsyncIterator[None]:
    db = await aiosqlite.connect("db.sqlite")
    app["DB"] = db
    yield
    await db.close()
```

Use DB in web-handler:

```
async def new_post(request: web.Request) -> web.Response:
    post = await request.json()
    db = request.config_dict["DB"]
    async with db.execute(
        "INSERT INTO posts (title, text) VALUES(?, ?, ?, ?)",
        [post["title"], post["text"]],
    ) as cursor:
        post_id = cursor.lastrowid
    await db.commit()
    return web.json_response({"new_post": post_id})
```

3.4.7 Full example for REST API

Example for simple blog REST API: *Full REST server example*

Post structure

Field	Description	Type
id	Post id	int
title	Title	str
text	Content	str
owner	Post creator	str
editor	Last editor	str

API endpoints

GET /api List posts.

POST /api Add new post. Arguments: *title*, *text*, *owner*.

GET /api/{post} Fetch existing post. Return Post's json.

DELETE /api/{post} Delete a post.

PATCH /api/{post} Edit a post. Arguments: *title*, *text*, *editor*.

Full REST server example

```
import asyncio
import sqlite3
from pathlib import Path
from typing import Any, AsyncIterator, Awaitable, Callable, Dict

import aiosqlite
from aiohttp import web


router = web.RouteTableDef()
```

(continues on next page)

(continued from previous page)

```
async def fetch_post(db: aiosqlite.Connection, post_id: int) -> Dict[str, Any]:
    async with db.execute(
        "SELECT owner, editor, title, text FROM posts WHERE id = ?", [post_id]
    ) as cursor:
        row = await cursor.fetchone()
        if row is None:
            raise RuntimeError(f"Post {post_id} doesn't exist")
        return {
            "id": post_id,
            "owner": row["owner"],
            "editor": row["editor"],
            "title": row["title"],
            "text": row["text"],
        }


def handle_json_error(
    func: Callable[[web.Request], Awaitable[web.Response]]
) -> Callable[[web.Request], Awaitable[web.Response]]:
    async def handler(request: web.Request) -> web.Response:
        try:
            return await func(request)
        except asyncio.CancelledError:
            raise
        except Exception as ex:
            return web.json_response(
                {"status": "failed", "reason": str(ex)}, status=400
            )

    return handler


@router.get("/")
async def root(request: web.Request) -> web.Response:
    return web.Response(text=f"Placeholder")


@router.get("/api")
@handle_json_error
async def api_list_posts(request: web.Request) -> web.Response:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
            ret.append(
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
```

(continues on next page)

(continued from previous page)

```
            )
    return web.json_response({"status": "ok", "data": ret})


@router.post("/api")
@handle_json_error
async def api_new_post(request: web.Request) -> web.Response:
    post = await request.json()
    title = post["title"]
    text = post["text"]
    owner = post["owner"]
    db = request.config_dict["DB"]
    async with db.execute(
        "INSERT INTO posts (owner, editor, title, text) VALUES(?, ?, ?, ?)",
        [owner, owner, title, text],
    ) as cursor:
        post_id = cursor.lastrowid
    await db.commit()
    return web.json_response(
        {
            "status": "ok",
            "data": {
                "id": post_id,
                "owner": owner,
                "editor": owner,
                "title": title,
                "text": text,
            },
        }
    )


@router.get("/api/{post}")
@handle_json_error
async def api_get_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await fetch_post(db, post_id)
    return web.json_response(
        {
            "status": "ok",
            "data": {
                "id": post_id,
                "owner": post["owner"],
                "editor": post["editor"],
                "title": post["title"],
                "text": post["text"],
            },
        }
    )
```

(continues on next page)

(continued from previous page)

```
@router.delete("/api/{post}")
@handle_json_error
async def api_del_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    async with db.execute("DELETE FROM posts WHERE id = ?", [post_id]) as cursor:
        if cursor.rowcount == 0:
            return web.json_response(
                {"status": "fail", "reason": f"post {post_id} doesn't exist"},
                status=404,
            )
    await db.commit()
    return web.json_response({"status": "ok", "id": post_id})


@router.patch("/api/{post}")
@handle_json_error
async def api_update_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    post = await request.json()
    db = request.config_dict["DB"]
    fields = {}
    if "title" in post:
        fields["title"] = post["title"]
    if "text" in post:
        fields["text"] = post["text"]
    if "editor" in post:
        fields["editor"] = post["editor"]
    if fields:
        field_names = ", ".join(f"{name} = ?" for name in fields)
        field_values = list(fields.values())
        await db.execute(
            f"UPDATE posts SET {field_names} WHERE id = ?", field_values + [post_id]
        )
        await db.commit()
    new_post = await fetch_post(db, post_id)
    return web.json_response(
        {
            "status": "ok",
            "data": {
                "id": new_post["id"],
                "owner": new_post["owner"],
                "editor": new_post["editor"],
                "title": new_post["title"],
                "text": new_post["text"],
            },
        }
    )


def get_db_path() -> Path:
    here = Path.cwd()
```

(continues on next page)

(continued from previous page)

```
    while not (here / ".git").exists():
        if here == here.parent:
            raise RuntimeError("Cannot find root github dir")
        here = here.parent

    return here / "db.sqlite3"


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = get_db_path()
    db = await aiosqlite.connect(sqlite_db)
    db.row_factory = aiosqlite.Row
    app["DB"] = db
    yield
    await db.close()


async def init_app() -> web.Application:
    app = web.Application()
    app.add_routes(router)
    app.cleanup_ctx.append(init_db)
    return app


def try_make_db() -> None:
    sqlite_db = get_db_path()
    if sqlite_db.exists():
        return

    with sqlite3.connect(sqlite_db) as conn:
        cur = conn.cursor()
        cur.execute(
            """CREATE TABLE posts (
            id INTEGER PRIMARY KEY,
            title TEXT,
            text TEXT,
            owner TEXT,
            editor TEXT,
            image BLOB)
        """
        )
        conn.commit()


try_make_db()


web.run_app(init_app())
```

3.5 aiohttp Client

Now we have a REST server, let's write REST client to it.

The idea is: create a `Client` class with `.list()`, `.get()`, `.create()` etc. methods to operate on blog posts collection.

3.5.1 Data structures

We need a `Post` *dataclass* to provide post related fields (and avoid dictionaries in our API):

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Post:
    id: int
    owner: str
    editor: str
    title: str
    text: Optional[str]

    def pprint(self) -> None:
        print(f"Post {self.id}")
        ...
```

3.5.2 Client class

`Client` is a class with embedded `aiohttp.ClientSession`. Also, we need the REST server URL to connect and user name to provide creator / last-editor information:

```
class Client:
    def __init__(self, base_url: URL, user: str) -> None:
        self._base_url = base_url
        self._user = user
        self._client = aiohttp.ClientSession(raise_for_status=True)
```

To properly close `Client` instance add `.close()` method:

```
async def close(self) -> None:
    return await self._client.close()
```

Now is a time for implementing a method to access the REST server, e.g. `.list()`:

```
async def list(self) -> List[Post]:
    async with self._client.get(self._make_url("api")) as resp:
        ret = await resp.json()
        return [Post(text=None, **item) for item in ret["data"]]
```

It makes `GET {base_url}/api` request, read JSON response and returns a list of `Post` objects.

`_make_url` is a helper for prepending *base url* to API endpoints. For the tutorial it is simple but in real life you often need to do more work, e.g. provide Authorization HTTP header, sign your request etc.:

```
def _make_url(self, path: str) -> URL:
    return self._base_url / path
```

3.5.3 Client Usage

The usage is simple:

```
async def fetch():
    client = Client("http://localhost:8080", "John")
    try:
        posts = await client.list()
        for post in posts:
            post.pprint()
    finally:
        await client.close()
```

`try/finally` is not very convinient, many people prefer `async with` statement. It saves 3 lines and (more important) avoids silly errors like instantiating a client variable *inside* `try/finally` block.

```
async def fetch():
    async with Client("http://localhost:8080", "John") as client:
        posts = await client.list()
        for post in posts:
            post.pprint()
```

To support this form we need to implement `__aenter__` / `__aexit__` async `Client` methods:

```
async def __aenter__(self) -> "Client":
    return self

async def __aexit__(
    self,
    exc_type: Optional[Type[BaseException]],
    exc_val: Optional[BaseException],
    exc_tb: Optional[TracebackType],
) -> Optional[bool]:
    await self.close()
```

3.5.4 Full Client Example

In *full example* we provide a *Command Line Tool* to work with REST API server by using famous Click library.

The *Click* usage is out of scope of the tutorial itself, but you can learn the full example on your own: *Full REST client example* .

Full REST client example

```
import asyncio
import functools
from contextlib import asynccontextmanager
from dataclasses import dataclass
from types import TracebackType
from typing import Any, AsyncIterator, Awaitable, Callable, List, Optional, Type

import aiohttp
import click
from yarl import URL


@dataclass(frozen=True)
class Post:
    id: int
    owner: str
    editor: str
    title: str
    text: Optional[str]  # post listing doesn't return text field

    def pprint(self) -> None:
        click.echo(f"Post {self.id}")
        click.echo(f"  Owner:  {self.owner}")
        click.echo(f"  Editor: {self.editor}")
        click.echo(f"  Title:  {self.title}")
        if self.text is not None:
            click.echo(f"  Text:   {self.text}")


class Client:
    def __init__(self, base_url: URL, user: str) -> None:
        self._base_url = base_url
        self._user = user
        self._client = aiohttp.ClientSession(raise_for_status=True)

    async def close(self) -> None:
        return await self._client.close()

    async def __aenter__(self) -> "Client":
        return self

    async def __aexit__(
        self,
        exc_type: Optional[Type[BaseException]],
        exc_val: Optional[BaseException],
        exc_tb: Optional[TracebackType],
    ) -> Optional[bool]:
        await self.close()
        return None

    def _make_url(self, path: str) -> URL:
```

(continues on next page)

(continued from previous page)

```
        return self._base_url / path

    async def create(self, title: str, text: str) -> Post:
        async with self._client.post(
            self._make_url("api"),
            json={"owner": self._user, "title": title, "text": text},
        ) as resp:
            ret = await resp.json()
            return Post(**ret["data"])

    async def get(self, post_id: int) -> Post:
        async with self._client.get(self._make_url(f"api/{post_id}")) as resp:
            ret = await resp.json()
            return Post(**ret["data"])

    async def delete(self, post_id: int) -> None:
        async with self._client.delete(self._make_url(f"api/{post_id}")) as resp:
            resp  # to make linter happy

    async def update(
        self, post_id: int, title: Optional[str] = None, text: Optional[str] = None
    ) -> Post:
        json = {"editor": self._user}
        if title is not None:
            json["title"] = title
        if text is not None:
            json["text"] = text
        async with self._client.patch(
            self._make_url(f"api/{post_id}"), json=json
        ) as resp:
            ret = await resp.json()
            return Post(**ret["data"])

    async def list(self) -> List[Post]:
        async with self._client.get(self._make_url(f"api")) as resp:
            ret = await resp.json()
            return [Post(text=None, **item) for item in ret["data"]]


@dataclass(frozen=True)
class Root:
    base_url: URL
    user: str
    show_traceback: bool

    @asynccontextmanager
    async def client(self) -> AsyncIterator[Client]:
        client = Client(self.base_url, self.user)
        try:
            yield client
        finally:
            await client.close()
```

(continues on next page)

(continued from previous page)

```
def async_cmd(func: Callable[..., Awaitable[None]]) -> Callable[..., None]:
    @functools.wraps(func)
    def inner(root: Root, **kwargs: Any) -> None:
        try:
            return asyncio.run(func(root, **kwargs))
        except Exception as exc:
            if root.show_traceback:
                raise
            else:
                click.echo(f"Error: {exc}")

    inner = click.pass_obj(inner)
    return inner


@click.group()
@click.option(
    "--base-url", type=str, default="http://localhost:8080", show_default=True
)
@click.option("--user", type=str, default="Anonymous", show_default=True)
@click.option("--show-traceback", is_flag=True, default=False, show_default=True)
@click.pass_context
def main(ctx: click.Context, base_url: str, user: str, show_traceback: bool) -> None:
    """REST client for tutorial server"""
    ctx.obj = Root(URL(base_url), user, show_traceback)


@main.command()
@click.option("--title", type=str, required=True)
@click.option("--text", type=str, required=True)
@async_cmd
async def create(root: Root, title: str, text: str) -> None:
    """Create new blog post"""
    async with root.client() as client:
        post = await client.create(title, text)
        click.echo(f"Created post {post.id}")
        post.pprint()


@main.command()
@click.argument("post_id", type=int)
@async_cmd
async def get(root: Root, post_id: int) -> None:
    """Get detailed info about blog post"""
    async with root.client() as client:
        post = await client.get(post_id)
        post.pprint()


@main.command()
```

(continues on next page)

(continued from previous page)

```
@click.argument("post_id", type=int)
@async_cmd
async def delete(root: Root, post_id: int) -> None:
    """Delete blog post"""
    async with root.client() as client:
        await client.delete(post_id)
        click.echo(f"Post {post_id} is deleted")


@main.command()
@click.argument("post_id", type=int)
@click.option("--text", type=str)
@click.option("--title", type=str)
@async_cmd
async def update(
    root: Root, post_id: int, title: Optional[str], text: Optional[str]
) -> None:
    """Update existing blog post"""
    async with root.client() as client:
        post = await client.update(post_id, title, text)
        post.pprint()


@main.command()
@async_cmd
async def list(root: Root) -> None:
    """List existing blog posts"""
    async with root.client() as client:
        posts = await client.list()
        click.echo("List posts:")
        for post in posts:
            post.pprint()


if __name__ == "__main__":
    main()
```

3.6 aiohttp templates

In our examples so far, we've only been returning plain text. You can return more complex HTML if you have templates. For this, templates can be used to render HTML templates.

aiohttp is a core library without embedded templating tool, third party libraries need to be installed to provide such functionality.

Let's use officially supported `aiohttp_jinja2` for famous `jinja2` template engine (http://aiohttp_jinja2.readthedocs.org/).

3.6.1 aiohttp-jinja2

Install the dependencies:

```
(.venv) python3.7 -m pip install -U jinja2 aiohttp-jinja2
```

Code structure:

```
/jinja_server.py
/templates/base.html
```

Sample HTML template:

```
<html>
    <head>
        <title>aiohttp page</title>
    </head>
    <body>
        <div>
            <h1><a href="/">My aiohttp server</a></h1>
        </div>

        <div>
            <p>Date now: {{ current_date }}</p>
            <p>Hello {{ username}}. </p>
        </div>
    </body>
</html>
```

3.6.2 Template rendering

In jinja_server.py, set up aiohttp-jinja2 and template directory:

```
import jinja2
import aiohttp_jinja2


app = web.Application()

aiohttp_jinja2.setup(
    app, loader=jinja2.FileSystemLoader(os.path.join(os.getcwd(), "templates"))
)
```

To render a page using the template:

```
@routes.get('/{username}')
async def greet_user(request: web.Request) -> web.Response:

    context = {
        'username': request.match_info.get("username", ""),
        'current_date': 'January 27, 2017'
    }
    response = aiohttp_jinja2.render_template("example.html", request,
```

(continues on next page)

(continued from previous page)

```
                                            context=context)

    return response
```

Another alternative is applying `@aiohttp_jinja2.template()` decorator to *web-handler*:

```
@routes.get('/{username}')
@aiohttp_jinja2.template("example.html")
async def greet_user(request: web.Request) -> Dict[str, Any]:
    context = {
        'username': request.match_info.get("username", ""),
        'current_date': 'January 27, 2017'
    }
    return content
```

Note, the `great_user` signature has changed: it returns a *jinja2 context* now. `@aiohttp_jinja2.template()` decorator renders the context and returns `web.Response` object automatically.

3.6.3 Render posts list

```
@router.get("/")
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request) -> Dict[str, Any]:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
            ret.append(
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
            )
    return {"posts": ret}
```

`index.html` template:

```
{% extends "base.html" %}
{% block title %}
Posts index
{% endblock %}

{% block content %}
<h1>Posts</h1>
<p>
  <ul>
    {% for post in posts %}
    <li>
      <a href="/{{ post.id }}">{{ post.title }}</a> {{ post.editor }}
```

(continues on next page)

(continued from previous page)

```
    </li>
    {% endfor %}
  </ul>
</p>
<hr>
<p>
  <a href="/new">New</a>
</p>

{% endblock %}
```

`base.html` for template inheritance:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>
    {% block title %}
      Index
    {% endblock %}
    </title>
  </head>
  <body>
    <div id="content"> {% block content %} {% endblock %} </div>
  </body>
</html>
```

3.6.4 Post editing

Show edit form

```
@router.get("/{post}/edit")
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}
```

`edit.html` template:

```
{% extends "base.html" %}
{% block title %}
Edit Post {{ post.title }}
{% endblock %}

{% block content %}
<h1>Edit: {{ post.title }}</h1>
<form action="/{{ post.id }}/edit" method="POST"
      enctype="multipart/form-data">
  <p>
```

(continues on next page)

(continued from previous page)

```
    Title:
    <input type="text" name="title" value="{{ post.title }}">
  </p>
  <p>Text:
    <textarea name="text" rows="10">{{ post.text }}</textarea>
  </p>
  <input type="submit">
</form>
<hr>
<p><small> created by: {{ owner }}, last editor: {{ editor }} </small></p>
<p>
  <a href="/">list</a>
  <a href="/{{ post.id }}">view</a>
  <a href="/{{ post.id }}/delete">delete</a>
</p>
{% endblock %}
```

Multipart content

We use `method="POST" enctype="multipart/form-data"` to send form data.

Sent body looks like:

```
------WebKitFormBoundaryw6YN2HqrOi6hewhP
Content-Disposition: form-data; name="title"

title 1
------WebKitFormBoundaryw6YN2HqrOi6hewhP
Content-Disposition: form-data; name="text"

text of post
------WebKitFormBoundaryw6YN2HqrOi6hewhP--
```

Applying edited form data

There is `POST` handler for `/{post}/edit` along with `GET` to apply a new data:

```
@router.post("/{post}/edit")
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    await db.execute(
        "UPDATE posts SET title = ?, text = ? WHERE id = ?",
        [post["title"], post["text"], post_id],
    )
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")
```

Note: `POST` handler doesn't render HTML itself but returects to `GET /{post}/edit` page.

3.6.5 HTML site endpoints

GET / List posts.

GET /new Show form for adding post

POST /new Apply post adding

GET /{post}/view Show post

GET /{post}/edit Edit post

GET /{post}/edit Show edit post form

POST /{post}/edit Apply post editing

GET /{post}/delete Delete post

Note: URLs order does matter: `/new` should lead `/{post}`, otherwise `/{post}` web handler is called with `new` post id.

3.6.6 Full example for templated server

Example for HTML version of blogs server: *Full server example with templates*

Full server example with templates

```
import sqlite3
from pathlib import Path
from typing import Any, AsyncIterator, Dict

import aiohttp_jinja2
import aiosqlite
import jinja2
from aiohttp import web


router = web.RouteTableDef()


@router.get("/")
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request) -> Dict[str, Any]:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
            ret.append(
```

(continues on next page)

(continued from previous page)

```
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
            )
    return {"posts": ret}


@router.get("/new")
@aiohttp_jinja2.template("new.html")
async def new_post(request: web.Request) -> Dict[str, Any]:
    return {}


@router.post("/new")
@aiohttp_jinja2.template("edit.html")
async def new_post_apply(request: web.Request) -> Dict[str, Any]:
    db = request.config_dict["DB"]
    post = await request.post()
    owner = "Anonymous"
    await db.execute(
        "INSERT INTO posts (owner, editor, title, text) VALUES(?, ?, ?, ?)",
        [owner, owner, post["title"], post["text"]],
    )
    await db.commit()
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}")
@aiohttp_jinja2.template("view.html")
async def view_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.get("/{post}/edit")
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.post("/{post}/edit")
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    await db.execute(
```

(continues on next page)

(continued from previous page)

```
        "UPDATE posts SET title = ?, text = ? WHERE id = ?",
        [post["title"], post["text"], post_id],
    )
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")


@router.get("/{post}/delete")
async def delete_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    db = request.config_dict["DB"]
    await db.execute("DELETE FROM posts WHERE id = ?", [post_id])
    raise web.HTTPSeeOther(location=f"/")


async def fetch_post(db: aiosqlite.Connection, post_id: int) -> Dict[str, Any]:
    async with db.execute(
        "SELECT owner, editor, title, text FROM posts WHERE id = ?", [post_id]
    ) as cursor:
        row = await cursor.fetchone()
        if row is None:
            raise RuntimeError(f"Post {post_id} doesn't exist")
        return {
            "id": post_id,
            "owner": row["owner"],
            "editor": row["editor"],
            "title": row["title"],
            "text": row["text"],
        }


def get_db_path() -> Path:
    here = Path.cwd()
    while not (here / ".git").exists():
        if here == here.parent:
            raise RuntimeError("Cannot find root github dir")
        here = here.parent

    return here / "db.sqlite3"


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = get_db_path()
    db = await aiosqlite.connect(sqlite_db)
    db.row_factory = aiosqlite.Row
    app["DB"] = db
    yield
    await db.close()


async def init_app() -> web.Application:
```

(continues on next page)

(continued from previous page)

```
    app = web.Application()
    app.add_routes(router)
    app.cleanup_ctx.append(init_db)
    aiohttp_jinja2.setup(
        app, loader=jinja2.FileSystemLoader(str(Path(__file__).parent / "templates"))
    )

    return app


def try_make_db() -> None:
    sqlite_db = get_db_path()
    if sqlite_db.exists():
        return

    with sqlite3.connect(sqlite_db) as conn:
        cur = conn.cursor()
        cur.execute(
            """CREATE TABLE posts (
            id INTEGER PRIMARY KEY,
            title TEXT,
            text TEXT,
            owner TEXT,
            editor TEXT,
            image BLOB)
        """
        )
        conn.commit()


try_make_db()


web.run_app(init_app())
```

3.7 aiohttp file uploading

Blog post can have images, let's support them!

3.7.1 Extend Post structure

Add `image BLOB` field.

Field	Description	Type
id	Post id	int
title	Title	str
text	Content	str
owner	Post creator	str
editor	Last editor	str
image	Image content	bytes

Note: Usually you serve images by *Content Delivery Network* or save them on *BLOB storage* like *Amazon S3*. Database keeps an image URL only.

But for sake of simplicity we use database for storing the *image content*.

3.7.2 Image web handler

```
@router.get("/{post}/image")
async def render_post_image(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    async with db.execute("SELECT image FROM posts WHERE id = ?", [post_id]) as cursor:
        row = await cursor.fetchone()
        if row is None or row["image"] is None:
            img = PIL.Image.new("RGB", (64, 64), color=0)
            fp = io.BytesIO()
            img.save(fp, format="JPEG")
            content = fp.getvalue()
        else:
            content = row["image"]
    return web.Response(body=content, content_type="image/jpeg")
```

Return a black image if nothing is present in DB, the image content with `image/jpeg` content type otherwise.

To render image we need to change `view.html` template:

```
{% block content %}
<h1>{{ post.title }}</h1>
...
<p><img src="/{{ post.id }}/image"></p>
...
{% endblock %}
```

3.7.3 Upload form

Add <input type="file" name="image" accept="image/png, image/jpeg"> HTML form field:

```
{% block content %}
<h1>Edit: {{ post.title }}</h1>
<form action="/{{ post.id }}/edit" method="POST"
      enctype="multipart/form-data">
  ...
  <input type="file" name="image" accept="image/png, image/jpeg">
  ...
</form>
{% endblock %}
```

3.7.4 Handle uploaded image

```
@router.post("/{post}/edit")
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    image = post.get("image")
    await db.execute(
        f"UPDATE posts SET title = ?, text = ? WHERE id = ?",
        [post["title"], post["text"], post_id],
    )
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")
```

If a new image was provided by HTML form – update DB:

```
async def apply_image(
    db: aiosqlite.Connection, post_id: int, img_content: bytes
) -> None:
    buf = io.BytesIO(img_content)
    loop = asyncio.get_event_loop()
    img = PIL.Image.open(buf)
    new_img = await loop.run_in_executor(None, img.resize, (64, 64), PIL.Image.LANCZOS)
    out_buf = io.BytesIO()
    new_img.save(out_buf, format="JPEG")
    await db.execute(
        "UPDATE posts SET image = ? WHERE id = ?", [out_buf.getvalue(), post_id]
    )
```

3.7.5 Full example for server with images support

Example for HTML version of blogs server with images: *Full server example with templates and image uploading*

Full server example with templates and image uploading

```
import asyncio
import io
import sqlite3
from pathlib import Path
from typing import Any, AsyncIterator, Dict

import aiohttp_jinja2
import aiosqlite
import jinja2
import PIL
import PIL.Image
from aiohttp import web


router = web.RouteTableDef()


@router.get("/")
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request) -> Dict[str, Any]:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
            ret.append(
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
            )
    return {"posts": ret}


@router.get("/new")
@aiohttp_jinja2.template("new.html")
async def new_post(request: web.Request) -> Dict[str, Any]:
    return {}


@router.post("/new")
@aiohttp_jinja2.template("edit.html")
async def new_post_apply(request: web.Request) -> Dict[str, Any]:
    db = request.config_dict["DB"]
    post = await request.post()
```

(continues on next page)

(continued from previous page)

```
    owner = "Anonymous"
    async with db.execute(
        "INSERT INTO posts (owner, editor, title, text) VALUES(?, ?, ?, ?)",
        [owner, owner, post["title"], post["text"]],
    ) as cursor:
        post_id = cursor.lastrowid
    image = post.get("image")
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}")
@aiohttp_jinja2.template("view.html")
async def view_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.get("/{post}/edit")
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.post("/{post}/edit")
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    image = post.get("image")
    await db.execute(
        f"UPDATE posts SET title = ?, text = ? WHERE id = ?",
        [post["title"], post["text"], post_id],
    )
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")


@router.get("/{post}/delete")
async def delete_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    await db.execute("DELETE FROM posts WHERE id = ?", [post_id])
```

(continues on next page)

(continued from previous page)

```
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}/image")
async def render_post_image(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    async with db.execute("SELECT image FROM posts WHERE id = ?", [post_id]) as cursor:
        row = await cursor.fetchone()
        if row is None or row["image"] is None:
            img = PIL.Image.new("RGB", (64, 64), color=0)
            fp = io.BytesIO()
            img.save(fp, format="JPEG")
            content = fp.getvalue()
        else:
            content = row["image"]
    return web.Response(body=content, content_type="image/jpeg")


async def apply_image(
    db: aiosqlite.Connection, post_id: int, img_content: bytes
) -> None:
    buf = io.BytesIO(img_content)
    loop = asyncio.get_event_loop()
    img = PIL.Image.open(buf)
    new_img = await loop.run_in_executor(None, img.resize, (64, 64), PIL.Image.LANCZOS)
    out_buf = io.BytesIO()
    new_img.save(out_buf, format="JPEG")
    await db.execute(
        "UPDATE posts SET image = ? WHERE id = ?", [out_buf.getvalue(), post_id]
    )


async def fetch_post(db: aiosqlite.Connection, post_id: int) -> Dict[str, Any]:
    async with db.execute(
        "SELECT owner, editor, title, text, image FROM posts WHERE id = ?", [post_id]
    ) as cursor:
        row = await cursor.fetchone()
        if row is None:
            raise RuntimeError(f"Post {post_id} doesn't exist")
        return {
            "id": post_id,
            "owner": row["owner"],
            "editor": row["editor"],
            "title": row["title"],
            "text": row["text"],
            "image": row["image"],
        }


def get_db_path() -> Path:
    here = Path.cwd()
```

(continues on next page)

(continued from previous page)

```
    while not (here / ".git").exists():
        if here == here.parent:
            raise RuntimeError("Cannot find root github dir")
        here = here.parent

    return here / "db.sqlite3"


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = get_db_path()
    db = await aiosqlite.connect(sqlite_db)
    db.row_factory = aiosqlite.Row
    app["DB"] = db
    yield
    await db.close()


async def init_app() -> web.Application:
    app = web.Application(client_max_size=64 * 1024 ** 2)
    app.add_routes(router)
    app.cleanup_ctx.append(init_db)
    aiohttp_jinja2.setup(
        app, loader=jinja2.FileSystemLoader(str(Path(__file__).parent / "templates"))
    )

    return app


def try_make_db() -> None:
    sqlite_db = get_db_path()
    if sqlite_db.exists():
        return

    with sqlite3.connect(sqlite_db) as conn:
        cur = conn.cursor()
        cur.execute(
            """CREATE TABLE posts (
            id INTEGER PRIMARY KEY,
            title TEXT,
            text TEXT,
            owner TEXT,
            editor TEXT,
            image BLOB)
        """
        )
        conn.commit()


try_make_db()


web.run_app(init_app())
```

3.8 aiohttp middlewares

A middleware is a coroutine that can modify either the request or response. For example, here's a simple middleware which appends to the response:

```
from aiohttp import web

@web.middleware
async def middleware(request: web.Request,
                     handler: Callable[[web.Request], Awaitable[web.Response]]):
    resp = await handler(request)
    resp.text = f"{resp.text} "
    return resp
```

The middleware is applied to *all* aiohttp application routes.

To setup middleware pass it to `web.Application` constructor:

```
app = web.Application(middlewares=[middleware])
```

Let's make something useful!

3.8.1 Error handling

```
@web.middleware
async def error_middleware(
    request: web.Request,
    handler: Callable[[web.Request], Awaitable[web.StreamResponse]],
) -> web.StreamResponse:
    try:
        return await handler(request)
    except web.HTTPException:
        raise
    except asyncio.CancelledError:
        raise
    except Exception as ex:
        return aiohttp_jinja2.render_template(
            "error-page.html", request, {"error_text": str(ex)}, status=400
        )
```

`error-page.html` template:

```
{% block title %}
Site Error
{% endblock %}

{% block content %}
<h1>Bad thing happens</h1>
<p>
  <b>
    <big>
      {{ error_text }}
```

(continues on next page)

(continued from previous page)

```
    </big>
  </b>
</p>
{% endblock %}
```

3.8.2 Modify request

Rewrite protocol/host/remote-ip:

```
@web.middleware
async def middleware(request: web.Request,
                     handler: Callable[[web.Request], Awaitable[web.Response]]):
    real_ip = request.headers['X-Forwarded-For']
    real_host = request.headers['X-Forwarded-Host']
    real_proto = request.headers['X-Forwarded-Proto']
    new_request = request.clone(remote=real_ip,
                                host=real_host,
                                scheme=real_proto)
    return await handler(request)
```

See also https://github.com/aio-libs/aiohttp-remotes for such middlewares.

3.8.3 DB Transaction handling

```
@web.middleware
async def middleware(request: web.Request,
                     handler: Callable[[web.Request], Awaitable[web.Response]]):
    db = request.config_dict["DB"]
    await db.execute("BEGIN")
    try:
        resp = await handler(request)
        await db.execute("COMMIT")
        return resp
    except Exception:
        await db.execute("ROLLBACK")
        raise
```

3.8.4 Check for login

```
@web.middleware
async def middleware(request: web.Request,
                     handler: Callable[[web.Request], Awaitable[web.Response]]):
    user = await get_user(request)
    if user is not None:
        request['USER'] = user
    else:  # user is None
        if login_required(handler):
```

(continues on next page)

(continued from previous page)

```
            raise web.HTTPSeeAlso(location='/login')
    return await handler(request)
```

3.8.5 Full example for server with error-page middleware

Example for HTML version of blogs server with images: *Full server with error-page middleware*

Full server with error-page middleware

```
import asyncio
import io
import sqlite3
from pathlib import Path
from typing import Any, AsyncIterator, Awaitable, Callable, Dict

import aiohttp_jinja2
import aiosqlite
import jinja2
import PIL
import PIL.Image
from aiohttp import web


@web.middleware
async def error_middleware(
    request: web.Request,
    handler: Callable[[web.Request], Awaitable[web.StreamResponse]],
) -> web.StreamResponse:
    try:
        return await handler(request)
    except web.HTTPException:
        raise
    except asyncio.CancelledError:
        raise
    except Exception as ex:
        return aiohttp_jinja2.render_template(
            "error-page.html", request, {"error_text": str(ex)}, status=400
        )


router = web.RouteTableDef()


@router.get("/")
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request) -> Dict[str, Any]:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
```

(continues on next page)

(continued from previous page)

```
            ret.append(
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
            )
    return {"posts": ret}


@router.get("/new")
@aiohttp_jinja2.template("new.html")
async def new_post(request: web.Request) -> Dict[str, Any]:
    return {}


@router.post("/new")
@aiohttp_jinja2.template("edit.html")
async def new_post_apply(request: web.Request) -> Dict[str, Any]:
    db = request.config_dict["DB"]
    post = await request.post()
    owner = "Anonymous"
    async with db.execute(
        "INSERT INTO posts (owner, editor, title, text) VALUES(?, ?, ?, ?)",
        [owner, owner, post["title"], post["text"]],
    ) as cursor:
        post_id = cursor.lastrowid
    image = post.get("image")
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}")
@aiohttp_jinja2.template("view.html")
async def view_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.get("/{post}/edit")
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}
```

(continues on next page)

(continued from previous page)

```
@router.post("/{post}/edit")
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    image = post.get("image")
    await db.execute(
        f"UPDATE posts SET title = ?, text = ? WHERE id = ?",
        [post["title"], post["text"], post_id],
    )
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")


@router.get("/{post}/delete")
async def delete_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    await db.execute("DELETE FROM posts WHERE id = ?", [post_id])
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}/image")
async def render_post_image(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    async with db.execute("SELECT image FROM posts WHERE id = ?", [post_id]) as cursor:
        row = await cursor.fetchone()
        if row is None or row["image"] is None:
            img = PIL.Image.new("RGB", (64, 64), color=0)
            fp = io.BytesIO()
            img.save(fp, format="JPEG")
            content = fp.getvalue()
        else:
            content = row["image"]
    return web.Response(body=content, content_type="image/jpeg")


async def apply_image(
    db: aiosqlite.Connection, post_id: int, img_content: bytes
) -> None:
    buf = io.BytesIO(img_content)
    out_buf = io.BytesIO()
    loop = asyncio.get_event_loop()
    img = PIL.Image.open(buf)
    new_img = await loop.run_in_executor(None, img.resize, (64, 64), PIL.Image.LANCZOS)
    new_img.save(out_buf, format="JPEG")
    await db.execute(
        "UPDATE posts SET image = ? WHERE id = ?", [out_buf.getvalue(), post_id]
```

(continues on next page)

(continued from previous page)

```
    )


async def fetch_post(db: aiosqlite.Connection, post_id: int) -> Dict[str, Any]:
    async with db.execute(
        "SELECT owner, editor, title, text, image FROM posts WHERE id = ?", [post_id]
    ) as cursor:
        row = await cursor.fetchone()
        if row is None:
            raise RuntimeError(f"Post {post_id} doesn't exist")
        return {
            "id": post_id,
            "owner": row["owner"],
            "editor": row["editor"],
            "title": row["title"],
            "text": row["text"],
            "image": row["image"],
        }


def get_db_path() -> Path:
    here = Path.cwd()
    while not (here / ".git").exists():
        if here == here.parent:
            raise RuntimeError("Cannot find root github dir")
        here = here.parent

    return here / "db.sqlite3"


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = get_db_path()
    db = await aiosqlite.connect(sqlite_db)
    db.row_factory = aiosqlite.Row
    app["DB"] = db
    yield
    await db.close()


async def init_app() -> web.Application:
    app = web.Application(
        client_max_size=64 * 1024 ** 2, middlewares=[error_middleware]
    )
    app.add_routes(router)
    app.cleanup_ctx.append(init_db)
    aiohttp_jinja2.setup(
        app, loader=jinja2.FileSystemLoader(str(Path(__file__).parent / "templates"))
    )

    return app
```

(continues on next page)

(continued from previous page)

```
def try_make_db() -> None:
    sqlite_db = get_db_path()
    if sqlite_db.exists():
        return

    with sqlite3.connect(sqlite_db) as conn:
        cur = conn.cursor()
        cur.execute(
            """CREATE TABLE posts (
            id INTEGER PRIMARY KEY,
            title TEXT,
            text TEXT,
            owner TEXT,
            editor TEXT,
            image BLOB)
        """
        )
        conn.commit()


try_make_db()


web.run_app(init_app())
```

3.9 aiohttp session

Session is a storage for saving temporary data like logged user info.

`aiohttp_session` library actually uses a middleware for session control.

Setup session support:

```
import aiohttp_session

async def init_app() -> web.Application:
    app = web.Application()
    app.add_routes(...)
    aiohttp_session.setup(app, aiohttp_session.SimpleCookieStorage())
```

We use unsecured `SimpleCookieStorage()` for tutorial to save session data in browser cookies.

More powerful alternatives are using `EncryptedCookieStorage`, `NaClCookieStorage` or `RedisStorage`. If you want to use another storage system like database please implement `AbstractStorage` interface and enjoy.

3.9.1 Session object

Session is available as `session = await aiohttp_session.get_session(request)` call.

The returned `session` object has dict-like interface:

```
session['NEW_KEY'] = value
```

The session is saved authomatically after finishing request handling.

3.9.2 Authorization

Create a middleware to check if a user is logged in:

```
_WebHandler = Callable[[web.Request], Awaitable[web.StreamResponse]]


def require_login(func: _WebHandler) -> _WebHandler:
    func.__require_login__ = True  # type: ignore
    return func


@web.middleware
async def check_login(request: web.Request,
                      handler: _WebHandler) -> web.StreamResponse:
    require_login = getattr(handler, "__require_login__", False)
    session = await aiohttp_session.get_session(request)
    username = session.get("username")
    if require_login:
        if not username:
            raise web.HTTPSeeOther(location="/login")
    return await handler(request)
```

It raises a redirection to `/login` page if *web-handler* requires authorization.

All sensitive web-handlers are marked by `@require_login` decorator:

```
@router.get("/{post}/edit")
@require_login
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    ...
```

3.9.3 Login and Logout

For login we use a simple HTML form:

```
@router.get("/login")
@aiohttp_jinja2.template("login.html")
async def login(request: web.Request) -> Dict[str, Any]:
    return {}
```

(continues on next page)

(continued from previous page)

```
@router.post("/login")
async def login_apply(request: web.Request) -> web.Response:
    session = await aiohttp_session.get_session(request)
    form = await request.post()
    session["username"] = form["login"]
    raise web.HTTPSeeOther(location="/")
```

`view.html` template:

```
{% block title %}
Login
{% endblock %}

{% block content %}
<h1>Login</h1>
<form action="/login" method="POST">
  <input type="text" name="login">
  <input type="submit">
</form>
{% endblock %}
```

Logout is even simpler:

```
@router.get("/logout")
async def logout(request: web.Request) -> web.Response:
    session = await aiohttp_session.get_session(request)
    session["username"] = None
    raise web.HTTPSeeOther(location="/")
```

3.9.4 Getting username from post adding and modifying handlers

Let's save logged in username as post's *editor*:

```
@router.post("/{post}/edit")
@require_login
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    session = await aiohttp_session.get_session(request)
    editor = session["username"]
    ...
```

Note: Multiple `get_session()` calls returns the same session object if used for handling the same HTTP request.

Last neat: use session data in rendering username by Jinja2 context processors:

```
async def username_ctx_processor(request: web.Request) -> Dict[str, Any]:
    # Jinja2 context processor
    session = await aiohttp_session.get_session(request)
    username = session.get("username")
    return {"username": username}
```

Rendering in template:

```
{% if username %}
<div>[{{ username }}] <a href="/logout">Logout</a></div>
{% else %}
<div>[Anonymous] <a href="/login">Login</a></div>
{% endif %}
```

Setup context processor:

```
async def init_app() -> web.Application:
    app = web.Application()
    ...
    aiohttp_jinja2.setup(
        app,
        loader=...,
        context_processors=[username_ctx_processor],
    )
```

3.9.5 Full example for server with sessions support

Example for HTML version of blogs server with images: *Full server with sessions support*

Full server with sessions support

```
import asyncio
import io
import sqlite3
from pathlib import Path
from typing import Any, AsyncIterator, Awaitable, Callable, Dict

import aiohttp_jinja2
import aiohttp_session
import aiosqlite
import jinja2
import PIL
import PIL.Image
from aiohttp import web


_WebHandler = Callable[[web.Request], Awaitable[web.StreamResponse]]


def require_login(func: _WebHandler) -> _WebHandler:
    func.__require_login__ = True  # type: ignore
    return func


@web.middleware
async def check_login(request: web.Request, handler: _WebHandler) -> web.StreamResponse:
    require_login = getattr(handler, "__require_login__", False)
```

(continues on next page)

(continued from previous page)

```
    session = await aiohttp_session.get_session(request)
    username = session.get("username")
    if require_login:
        if not username:
            raise web.HTTPSeeOther(location="/login")
    return await handler(request)


async def username_ctx_processor(request: web.Request) -> Dict[str, Any]:
    # Jinja2 context processor
    session = await aiohttp_session.get_session(request)
    username = session.get("username")
    return {"username": username}


@web.middleware
async def error_middleware(
    request: web.Request, handler: _WebHandler
) -> web.StreamResponse:
    try:
        return await handler(request)
    except web.HTTPException:
        raise
    except asyncio.CancelledError:
        raise
    except Exception as ex:
        return aiohttp_jinja2.render_template(
            "error-page.html", request, {"error_text": str(ex)}, status=400
        )


router = web.RouteTableDef()


@router.get("/")
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request) -> Dict[str, Any]:
    ret = []
    db = request.config_dict["DB"]
    async with db.execute("SELECT id, owner, editor, title FROM posts") as cursor:
        async for row in cursor:
            ret.append(
                {
                    "id": row["id"],
                    "owner": row["owner"],
                    "editor": row["editor"],
                    "title": row["title"],
                }
            )
    return {"posts": ret}
```

(continues on next page)

(continued from previous page)

```
@router.get("/login")
@aiohttp_jinja2.template("login.html")
async def login(request: web.Request) -> Dict[str, Any]:
    return {}


@router.post("/login")
async def login_apply(request: web.Request) -> web.Response:
    session = await aiohttp_session.get_session(request)
    form = await request.post()
    session["username"] = form["login"]
    raise web.HTTPSeeOther(location="/")


@router.get("/logout")
async def logout(request: web.Request) -> web.Response:
    session = await aiohttp_session.get_session(request)
    session["username"] = None
    raise web.HTTPSeeOther(location="/")


@router.get("/new")
@require_login
@aiohttp_jinja2.template("new.html")
async def new_post(request: web.Request) -> Dict[str, Any]:
    return {}


@router.post("/new")
@require_login
@aiohttp_jinja2.template("edit.html")
async def new_post_apply(request: web.Request) -> Dict[str, Any]:
    db = request.config_dict["DB"]
    post = await request.post()
    session = await aiohttp_session.get_session(request)
    owner = session["username"]
    async with db.execute(
        "INSERT INTO posts (owner, editor, title, text) VALUES(?, ?, ?, ?)",
        [owner, owner, post["title"], post["text"]],
    ) as cursor:
        post_id = cursor.lastrowid
    image = post.get("image")
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}")
@aiohttp_jinja2.template("view.html")
async def view_post(request: web.Request) -> Dict[str, Any]:
```

(continues on next page)

(continued from previous page)

```
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.get("/{post}/edit")
@require_login
@aiohttp_jinja2.template("edit.html")
async def edit_post(request: web.Request) -> Dict[str, Any]:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    return {"post": await fetch_post(db, post_id)}


@router.post("/{post}/edit")
@require_login
async def edit_post_apply(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    post = await request.post()
    image = post.get("image")
    session = await aiohttp_session.get_session(request)
    editor = session["username"]
    await db.execute(
        f"UPDATE posts SET title = ?, text = ?, editor = ? WHERE id = ?",
        [post["title"], post["text"], editor, post_id],
    )
    if image:
        img_content = image.file.read()  # type: ignore
        await apply_image(db, post_id, img_content)
    await db.commit()
    raise web.HTTPSeeOther(location=f"/{post_id}/edit")


@router.get("/{post}/delete")
@require_login
async def delete_post(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    await db.execute("DELETE FROM posts WHERE id = ?", [post_id])
    raise web.HTTPSeeOther(location=f"/")


@router.get("/{post}/image")
async def render_post_image(request: web.Request) -> web.Response:
    post_id = request.match_info["post"]
    db = request.config_dict["DB"]
    async with db.execute("SELECT image FROM posts WHERE id = ?", [post_id]) as cursor:
        row = await cursor.fetchone()
        if row is None or row["image"] is None:
            img = PIL.Image.new("RGB", (64, 64), color=0)
            fp = io.BytesIO()
```

(continues on next page)

(continued from previous page)

```
            img.save(fp, format="JPEG")
            content = fp.getvalue()
        else:
            content = row["image"]
    return web.Response(body=content, content_type="image/jpeg")


async def apply_image(
    db: aiosqlite.Connection, post_id: int, img_content: bytes
) -> None:
    buf = io.BytesIO(img_content)
    out_buf = io.BytesIO()
    loop = asyncio.get_event_loop()
    img = PIL.Image.open(buf)
    new_img = await loop.run_in_executor(None, img.resize, (64, 64), PIL.Image.LANCZOS)
    new_img.save(out_buf, format="JPEG")
    await db.execute(
        "UPDATE posts SET image = ? WHERE id = ?", [out_buf.getvalue(), post_id]
    )


async def fetch_post(db: aiosqlite.Connection, post_id: int) -> Dict[str, Any]:
    async with db.execute(
        "SELECT owner, editor, title, text, image FROM posts WHERE id = ?", [post_id]
    ) as cursor:
        row = await cursor.fetchone()
        if row is None:
            raise RuntimeError(f"Post {post_id} doesn't exist")
        return {
            "id": post_id,
            "owner": row["owner"],
            "editor": row["editor"],
            "title": row["title"],
            "text": row["text"],
            "image": row["image"],
        }


def get_db_path() -> Path:
    here = Path.cwd()
    while not (here / ".git").exists():
        if here == here.parent:
            raise RuntimeError("Cannot find root github dir")
        here = here.parent

    return here / "db.sqlite3"


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = get_db_path()
    db = await aiosqlite.connect(sqlite_db)
    db.row_factory = aiosqlite.Row
```

(continues on next page)

(continued from previous page)

```
    app["DB"] = db
    yield
    await db.close()


async def init_app() -> web.Application:
    app = web.Application(client_max_size=64 * 1024 ** 2)
    app.add_routes(router)
    app.cleanup_ctx.append(init_db)
    aiohttp_session.setup(app, aiohttp_session.SimpleCookieStorage())
    aiohttp_jinja2.setup(
        app,
        loader=jinja2.FileSystemLoader(str(Path(__file__).parent / "templates")),
        context_processors=[username_ctx_processor],
    )
    app.middlewares.append(error_middleware)
    app.middlewares.append(check_login)

    return app


def try_make_db() -> None:
    sqlite_db = get_db_path()
    if sqlite_db.exists():
        return

    with sqlite3.connect(sqlite_db) as conn:
        cur = conn.cursor()
        cur.execute(
            """CREATE TABLE posts (
            id INTEGER PRIMARY KEY,
            title TEXT,
            text TEXT,
            owner TEXT,
            editor TEXT,
            image BLOB)
        """
        )
        conn.commit()


try_make_db()


web.run_app(init_app())
```

3.10 aiohttp Writing tests

There are several tools for testing Python code.

We show how to work with the most powerful and popular tool called pytest .

We need to install `pytest` itself and `pytest-aiohttp` plugin first:

```
pip install pytest
pip install pytest-aiohttp
```

Note: `pytest-aiohttp` is not compatible with another pupular plugin `pytest-asyncio`.

3.10.1 Making test asynchronous

Just use `async def test_*()` instead of plain `def test_*()`. `pytest-aiohttp` does all dirty work for you:

```
async def test_a() -> None:
    await asyncio.sleep(0.01)
    assert 1 == 2  # False
```

3.10.2 Testing server code

Let's sligtly modify application code to accept a path to database explicitly:

```
async def init_app(db_path: Path) -> web.Application:
    app = web.Application()
    app["DB_PATH"] = db_path
    ...


async def init_db(app: web.Application) -> AsyncIterator[None]:
    sqlite_db = app["DB_PATH"]
    db = await aiosqlite.connect(sqlite_db)
    ...
```

After that we can use a separate isolated *test database* for running unit tests.

We need handy fixtures to initialize test DB:

```
@pytest.fixture
def db_path(tmp_path: Path) -> Path:
    path = tmp_path / "test_sqlite.db"
    try_make_db(path)
    return path


@pytest.fixture
async def db(db_path: Path) -> aiosqlite.Connection:
    """DB connection to access test DB directly from tests"""
    conn = await aiosqlite.connect(db_path)
```

(continues on next page)

(continued from previous page)

```
    conn.row_factory = aiosqlite.Row
    yield conn
    await conn.close()
```

Note: Fixtures have *function* scope (default), that mean that every test gets a new fresh empty database.

Another fixtire instantiates our server and starts it on random local TCP port using *aiohttp provided* `aiohttp_client` fixture:

```
@pytest.fixture
async def client(aiohttp_client: Any, db_path: Path) -> _TestClient:
    app = await init_app(db_path)
    return await aiohttp_client(app)
```

Test posts list:

```
async def test_list_empty(client: _TestClient) -> None:
    resp = await client.get("/api")
    assert resp.status == 200, await resp.text()
    data = await resp.json()
    assert data == {"data": [], "status": "ok"}
```

Use db fixture to add a post before testing `/api/{post_id}` web-handler:

```
async def test_get_post(client: _TestClient, db: aiosqlite.Connection) -> None:
    async with db.execute(
        "INSERT INTO posts (title, text, owner, editor) VALUES (?, ?, ?, ?)",
        ["title", "text", "user", "user"],
    ) as cursor:
        post_id = cursor.lastrowid
    await db.commit()

    resp = await client.get(f"/api/{post_id}")
    assert resp.status == 200
    data = await resp.json()
    assert data == {
        "data": {
            "editor": "user",
            "id": "1",
            "owner": "user",
            "text": "text",
            "title": "title",
        },
        "status": "ok",
    }
```

3.10.3 Testing client with real server

Create a fixture to start a test server and `Client` instance:

```
@pytest.fixture
async def server(aiohttp_server: Any, db_path: Path) -> _TestServer:
    app = await init_app(db_path)
    return await aiohttp_server(app)


@pytest.fixture
async def client(server: _TestServer) -> Client:
    async with Client(server.make_url("/"), "test_user") as client:
        yield client
```

Use `Client` instance to test against running server:

```
async def test_get_post(client: Client, db: aiosqlite.Connection) -> None:
    post = await client.create("test title", "test text")

    async with db.execute(
        "SELECT title, text, owner, editor FROM posts WHERE id = ?", [post.id]
    ) as cursor:
        record = await cursor.fetchone()
        assert record["title"] == "test title"
        assert record["text"] == "test text"
        assert record["owner"] == "test_user"
        assert record["editor"] == "test_user"
```

3.10.4 Testing client with fake server

Assume we have no server code. We need to use *fake server*:

```
async def test_get_post(aiohttp_server: Any) -> None:
    async def handler(request: web.Request) -> web.Response:
        data = await request.json()
        assert data["title"] == "test title"
        assert data["text"] == "test text"
        return web.json_response(
            {
                "status": "ok",
                "data": {
                    "id": 1,
                    "title": "test title",
                    "text": "test text",
                    "owner": "test_user",
                    "editor": "test_user",
                },
            }
        )

    app = web.Application()
```

(continues on next page)

(continued from previous page)

```
    app.add_routes([web.post("/api", handler)])
    server = await aiohttp_server(app)
    async with Client(server.make_url("/"), "test_user") as client:
        post = await client.create("test title", "test text")

        assert post.id == 1
        assert post.title == "test title"
        assert post.text == "test text"
        assert post.owner == "test_user"
        assert post.editor == "test_user"
```

3.10.5 Working with HTTPS

To test HTTPS proper SSL certificates are required.

Certificate is coupled with DNS, you don't want to pay for testing certs.

There are two options: use pre-generated self-signed certificate pair of use awesome `trustme` library:

```
@pytest.fixture
def tls_certificate_authority() -> Any:
    return trustme.CA()


@pytest.fixture
def tls_certificate(tls_certificate_authority: Any) -> Any:
    return tls_certificate_authority.issue_server_cert("localhost",
                                                        "127.0.0.1",
                                                        "::1")


@pytest.fixture
def server_ssl_ctx(tls_certificate: Any) -> ssl.SSLContext:
    ssl_ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
    tls_certificate.configure_cert(ssl_ctx)
    return ssl_ctx


@pytest.fixture
def client_ssl_ctx(tls_certificate_authority: Any) -> ssl.SSLContext:
    ssl_ctx = ssl.create_default_context(purpose=ssl.Purpose.SERVER_AUTH)
    tls_certificate_authority.configure_trust(ssl_ctx)
    return ssl_ctx
```

Pass `server_ssl_ctx` and `client_ssl_ctx` as `ssl` parameter to test *secured* TCP connection instead of plain TCP.

3.10.6 Client mocking

There is `aioresponses` third-party library:

```
pip install aioresponses
```

Usage:

```
from aioresponses import aioresponses

async def test_request() -> None:
    with aioresponses() as mocked:
        mocked.get('http://example.com', status=200, body='test')
        session = aiohttp.ClientSession()
        resp = await session.get('http://example.com')

        assert resp.status == 200
        assert "test" == await resp.text()
```

TBD (by Andrew)

- Writing tests for aiohttp client
- Writing tests for aiohttp server
- Tests using pytest and pytest-asyncio.

3.11 Git Cheat Sheet

There are other more complete git tutorial out there, but the following should help you during this workshop.

3.11.1 Clone a repo

```
git clone <url>
```

3.11.2 Create a new branch, and switch to it

```
git checkout -b <branchname>
```

3.11.3 Switching to an existing branch

```
git checkout <branchname>
```

3.11.4 Adding files to be committed

```
git add <filename>
```

3.11.5 Commit your changes

```
git commit -m "<commit message>"
```

3.11.6 Pushing changes to remote

```
git push <remote> <branchname>
```

3.11.7 Rebase with a branch on remote

```
git rebase <remote>/<branchname>
```