Open in app

Following

588K Followers

Build an async python service with FastAPI & SQLAlchemy

Build a fully asynchronous python service, including async DB queries, using FastAPI and the new SQLAlchemy AsyncIO support

Michael Azimov Apr 4 · 8 min read

Photo by John Cameron on Unsplash

Background

Before We Begin

In this blog post we are going to build a small CRUD python app (I will be using Python 3.9, but it should work on 3.7+). The post will focus on the implementation and usage of FastAPI and async SQLAlchemy, and not on the theory and general usage of AsyncIO in Python. In addition, this post assumes you have previous knowledge in the following subjects:

1. Python language and ecosystem (venvs, IDE)
2. SQL and SQLALchemy
3. HTTP web services concepts

All of the code in this post can be found in this public repository: https://github.com/azimovMichael/my-async-app

Async programming and python AsyncIO

Asynchronous programming is a pattern of programming that enables code to run separately from the main application thread. Asynchronous programming is used in many use-cases such as event-driven systems, highly scalable apps, and many more.

Asynchronous programming is not a new concept. It's been around for a while, especially in the JavaScript ecosystem. Python 3.4 introduced *asyncio* package, that implements python support for the Async/Await design. There are plenty examples and tutorials about it, but some of my favorite are: Introduction to async programming in python, Multitasking Is Not My Forte, So How Can I Blame Python? (by my colleague Danielle shaul) and Concurrent Burgers — Understand async / await (by the creator of FastAPI Sebastián Ramírez)

Initial Setup

Preparing environment

First of all, you need to create a new project with a new virtual environment. Inside the new venv, install our first packages — FastAPI and uvicorn.

```
pip install fastapi uvicorn[standard]
```

FastAPI is a fairly new python (micro) web framework with built-in support for async endpoints.

Uvicorn is our chosen ASGI server. ASGI Is the asynchronous sister of python WSGI.

Now we are ready for some code.

First async endpoint

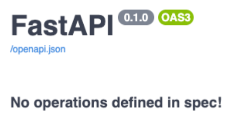
To configure a FastAPI service, create a python module named `app.py`, with the following code:

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()
```

```python

if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by GitHub view raw

This code isn't doing much. It's starting a FastAPI application, using uvicorn ASGI server, on port 1111, without any custom endpoints. FastAPI comes with out-of-the-box support for OpenAPI Docs, so you can run the app and go to: http://127.0.0.1:1111/docs in your browser. If everything went well, you should see a web page similar to this:



No operations defined in spec!

Initial view of the auto-generated API Docs

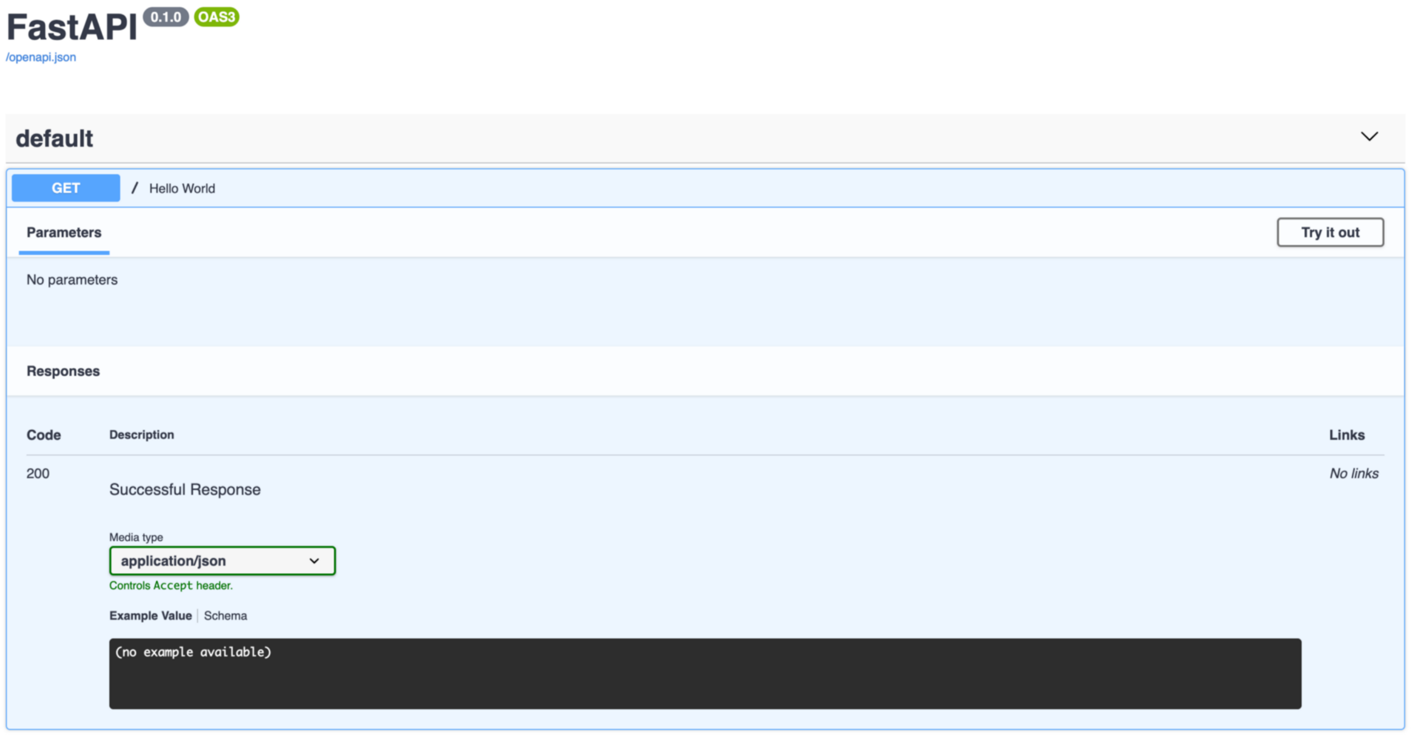
Now we are ready to create our first async endpoint.

In your `app.py` file, add an async function called `hello_world`, and mount it to the base GET route:

```python
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def hello_world():
    return "hello_world"

if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by GitHub view raw

Re-run the service, and you should see two new things: In the API Docs, you should see our new endpoint, and you can also call it using the "Try it out" button:



No parameters

Code | Description | Links
200 | Successful Response | No links

```
(no example available)
```

Docs for our new endpoint

You can also go to http://127.0.0.1:1111/, and see the "hello_world" response text in our browser.

We have our first async endpoint! 🎇

Book store CRUD app

So, the purpose of this post is to build a small CRUD app. For demonstrations purposes we'll build a simple book store app, with the ability to create, update and fetch books.

SQLAlchemy and AsyncIO

In order to use a DB for storing the books, I'm using Python's SQLalchemy library with an sqlite dialect that supports asyncio (*aiosqlite*)

Install the necessary packages:

```
pip install SQLAlchemy==1.4.3 aiosqlite
```

SQLAlchemy 1.4.X is a pretty new release, with lots of upgrades and new features (a first step towards the highly anticipated 2.0 version), that are detailed here. In this post,

I will mainly focus on the new async support.
SQLAlchemy Asyncio should be considered alpha level in early 1.4 releases (!).

Once we've installed SQLAlchemy, we need to configure our DB connection. Create a new python package named `db` and a new module inside it, called `config.py` *:*

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import declarative_base, sessionmaker

DATABASE_URL = "sqlite+aiosqlite:///./test.db"

engine = create_async_engine(DATABASE_URL, future=True, echo=True)
async_session = sessionmaker(engine, expire_on_commit=False, class_=AsyncSession)
Base = declarative_base()
```

config.py hosted with ❤ by **GitHub** **view raw**

Our DB server is a local instance of SQLLite (a local `test.db` file) and we will talk to it using the `aiosqlite` dialect that supports async queries. We are creating the db engine using the new `create_async_engine` function. The session maker is created with two unique flags: `expire_on_commit=False` makes sure that our db entities and fields will be available even after a commit was made on the session, and `class_=AsyncSession` is the new async session. We'll also use the good, old, `declarative_base` to configure our soon-to-be-created DB models.

> *If you want to use a different DB (MySql, PostgreSQL, etc) you need to install a compatible driver with AsyncIO support, and update the* `DATABASE_URL` *parameter.*

DB Model

We're building a book store app, so it shouldn't be a surprise that our main DB table will be, you guessed it — a book. Create a new package inside the `db` package, called `models`, and inside it a new module called `book.py`.
Our book entity will also have some fields — name, author and release year:

```python
from sqlalchemy import Column, Integer, String

from db.config import Base

```

```python
class Book(Base):
    __tablename__ = 'books'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    author = Column(String, nullable=False)
    release_year = Column(Integer, nullable=False)
```

book.py hosted with ❤ by GitHub view raw

In order to create a new entity, we will use a DAL (Data Access Layer) class, that will be responsible for all the sql functions for this DB model. Create a `dals` package and a `book_dal.py` module in it:

```python
from typing import List, Optional

from sqlalchemy import update
from sqlalchemy.future import select
from sqlalchemy.orm import Session

from db.models.book import Book

class BookDAL():
    def __init__(self, db_session: Session):
        self.db_session = db_session

    async def create_book(self, name: str, author: str,   release_year: int):
        new_book = Book(name=name,author=author, release_year=release_year)
        self.db_session.add(new_book)
        await self.db_session.flush()

    async def get_all_books(self) -> List[Book]:
        q = await self.db_session.execute(select(Book).order_by(Book.id))
        return q.scalars().all()

    async def update_book(self, book_id: int, name: Optional[str], author: Optional[str]
        q = update(Book).where(Book.id == book_id)
        if name:
            q = q.values(name=name)
        if author:
            q = q.values(author=author)
        if release_year:
            q = q.values(release_year=release_year)
        q.execution_options(synchronize_session="fetch")
        await  self.db_session.execute(q)
```

book_dal.py hosted with ❤ by **GitHub** **view raw**

We have 3 functions: A `create_book` function that receives our entity fields, and saves a new record to the DB. A `get_all_books` function that returns all books in the DB, and an `update_book` function that receives a book_id and optionally new values for the book fields and updates them. In the `update_book` functions, we are adding an execution option called `synchronize_session` that tells the session to "refresh" the updated entities using the `fetch` method, to make sure the entities we have in memory will be up-to-date with the new values we updated.

Now we need some new endpoints that will use these DAL functions, so let's add them our `app.py`:

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()


@app.post("/books")
async def create_book(name: str, author: str, release_year: int):
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.create_book(name, author, release_year)

@app.get("/books")
async def get_all_books() -> List[Book]:
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.get_all_books()

@app.put("/books/{book_id}")
async def update_book(book_id: int, name: Optional[str] = None, author: Optional[str] =
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.update_book(book_id, name, author, release_year)

if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by GitHub view raw

Each of the endpoints asynchronously creates a DB session, a BookDAL instance with the session and calls the relevant function (If your "clean code" instincts are screaming right now, that's totally fine, we will address them in a few paragraphs).

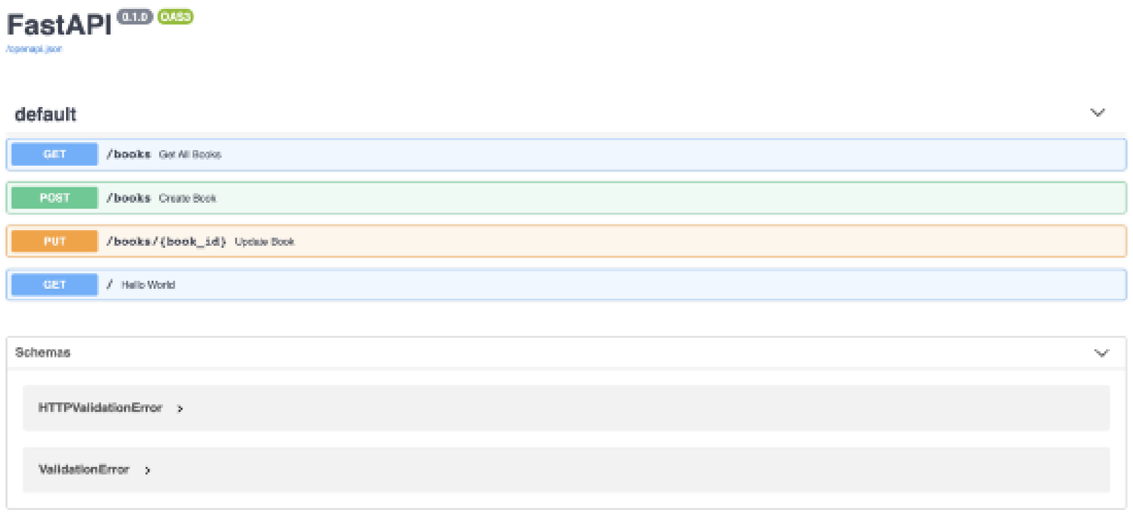
> *Pay attention to the PUT method: by using an* `Optional` *type with a default value, we make the query parameters as optional.*

But, for all of these nice stuff to actually work, we need to create our *books* table. In order to do that, we will utilize FastAPI's events feature, and create the necessary tables on app startup (obviously, this is not something we will want to do in a production app):

```
from typing import List

import uvicorn
from fastapi import FastAPI

from db.config import engine, Base, async_session
from db.dals.book_dal import BookDAL
from db.models.book import Book

app = FastAPI()


@app.on_event("startup")
async def startup():
    # create db tables
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)


@app.post("/books")
async def create_book(name: str, author: str, release_year: int):
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.create_book(name, author, release_year)


@app.get("/books")
async def get_all_books() -> List[Book]:
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.get_all_books()

@app.put("/books/{book_id}")
async def update_book(book_id: int, name: Optional[str] = None, author: Optional[str] =
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.update_book(book_id, name, author, release_year)


if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by **GitHub** **view raw**

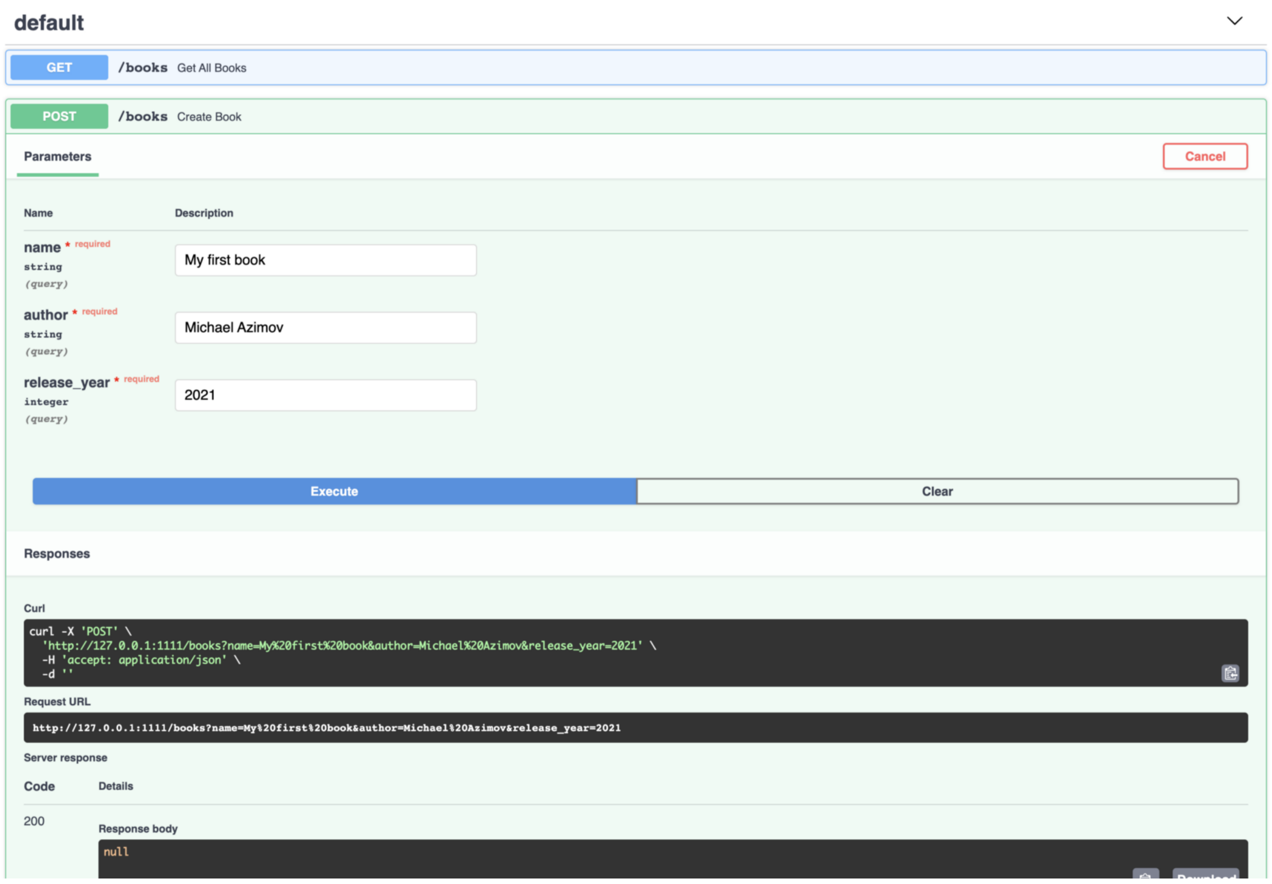
Now, you can run the app, and check the API Docs page. It should look like this:



GET /books Get All Books
POST /books Create Book
PUT /books/{book_id} Update Book
GET / Hello World

HTTPValidationError
ValidationError

The API docs are interactive so you can call our new endpoints right from this page. Create a book:



Name	Description
name * required string (query)	My first book
author * required string (query)	Michael Azimov
release_year * required integer (query)	2021

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:1111/books?name=My%20first%20book&author=Michael%20Azimov&release_year=2021' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

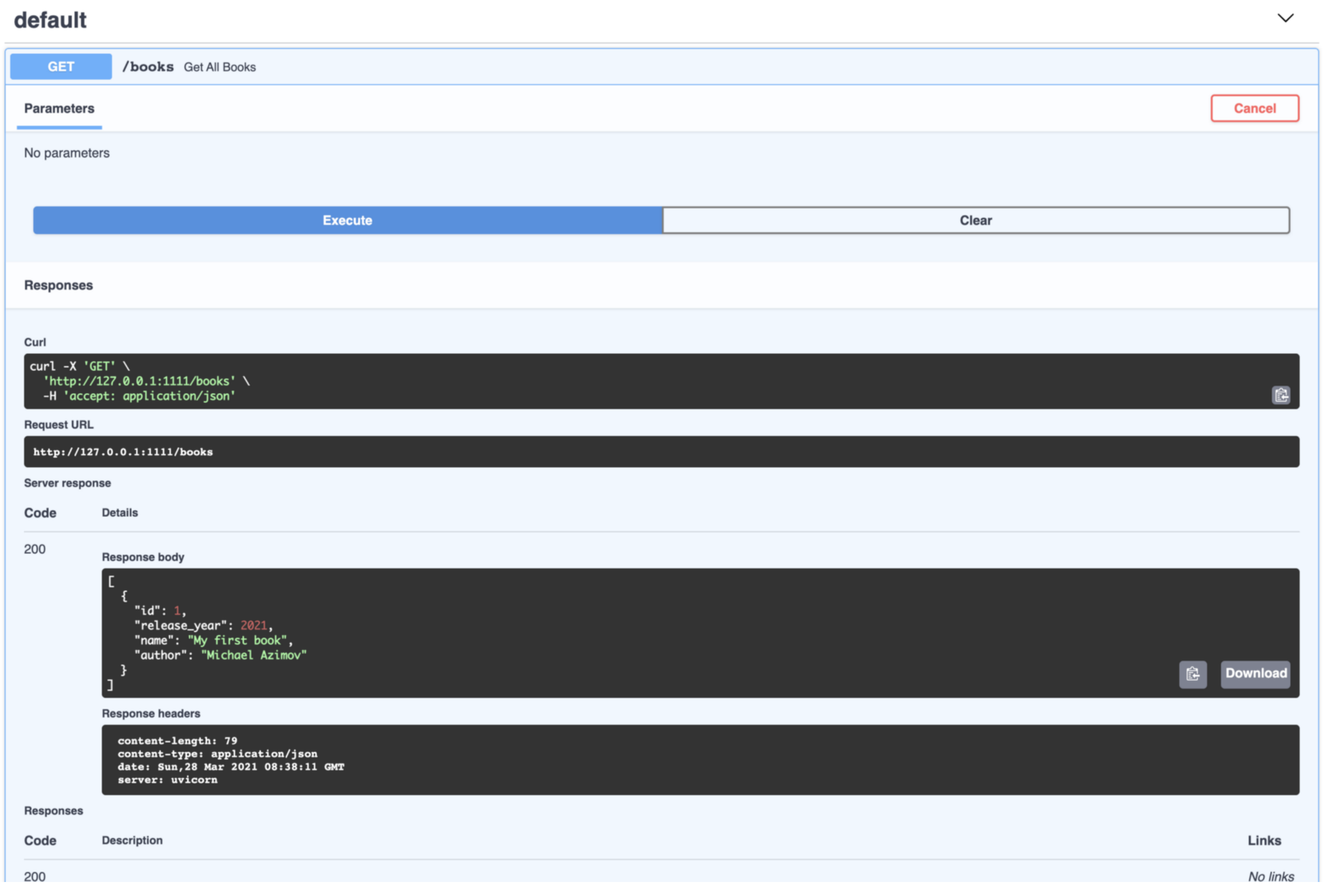
```
http://127.0.0.1:1111/books?name=My%20first%20book&author=Michael%20Azimov&release_year=2021
```

Server response

Code	Details
200	Response body

```
null
```

You can use any input parameters you want. After executing the command, you should see a new record in our books table. Now, let's fetch it using the *get* endpoint:



No parameters

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:1111/books' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:1111/books
```

Server response

Code	Details
200	Response body

```
[
  {
    "id": 1,
    "release_year": 2021,
    "name": "My first book",
    "author": "Michael Azimov"
  }
]
```

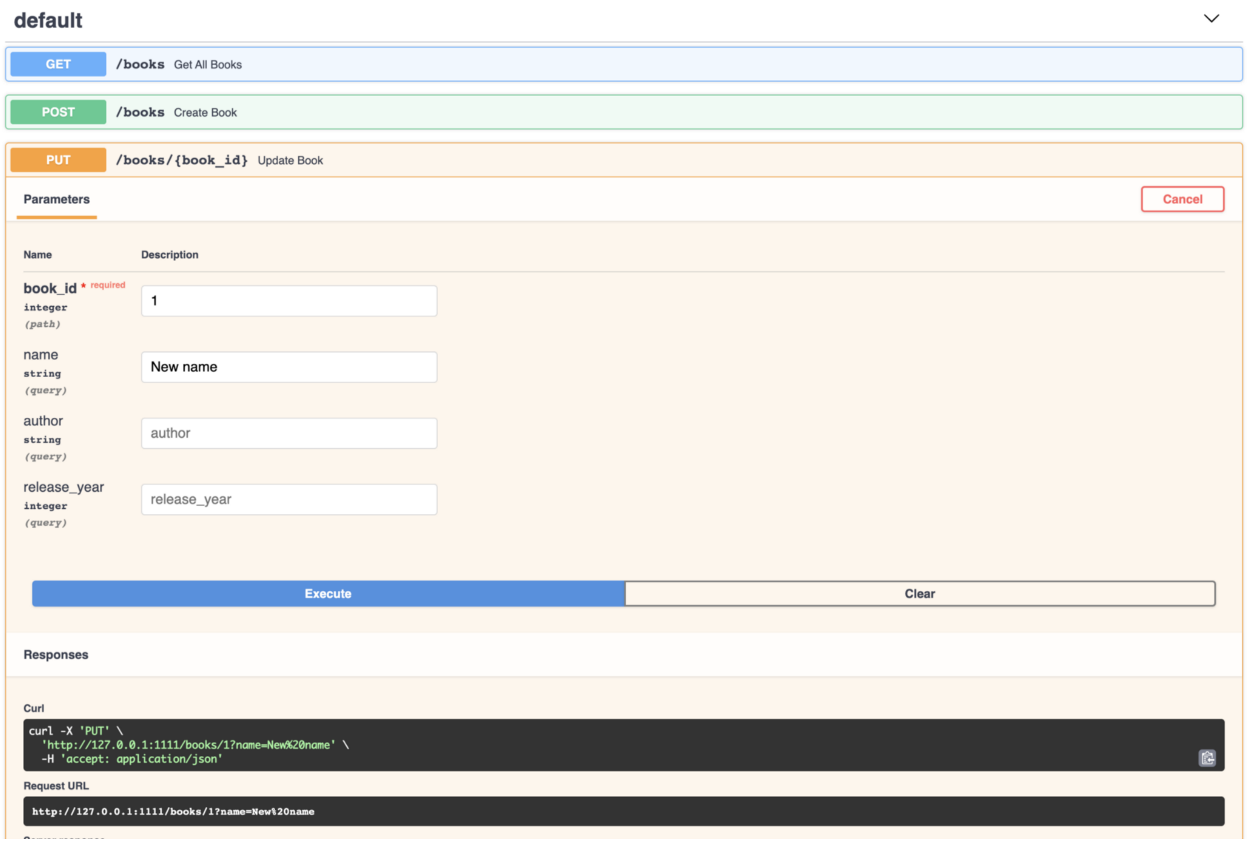
Response headers

```
content-length: 79
content-type: application/json
date: Sun,28 Mar 2021 08:38:11 GMT
server: uvicorn
```

Responses

Code	Description	Links
200		No links

As you can see, our app has responded with our new book, that was given the id 1. We can now update this book, using the update book PUT function:



Name	Description
book_id * required integer (path)	1
name string (query)	New name
author string (query)	author
release_year integer (query)	release_year

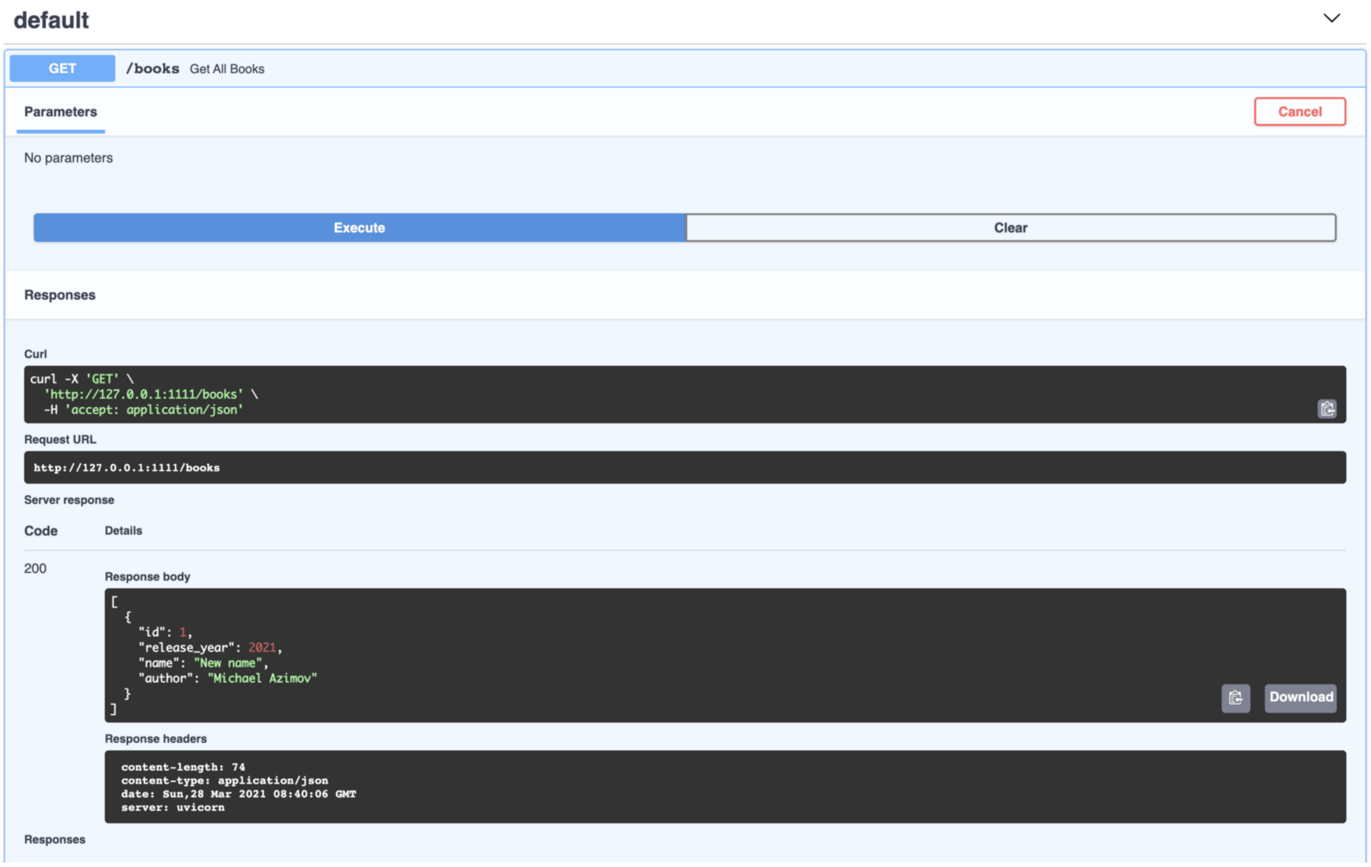
Curl

```
curl -X 'PUT' \
  'http://127.0.0.1:1111/books/1?name=New%20name' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:1111/books/1?name=New%20name
```

and then fetch it again, and verify that the name of the book has actually changed:



```
curl -X 'GET' \
  'http://127.0.0.1:1111/books' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:1111/books
```

Server response

Code Details

200 Response body

```
[
  {
    "id": 1,
    "release_year": 2021,
    "name": "New name",
    "author": "Michael Azimov"
  }
]
```

Response headers

```
content-length: 74
content-type: application/json
date: Sun,28 Mar 2021 08:40:06 GMT
server: uvicorn
```

Congratulations, you've created your first fully async python service! 🎉

Refactoring

Now that we have a working async app, I want to use some FastAPI features to make the code a bit cleaner.

API Router

Currently, our `app.py` file contains all of our endpoints. In a real-world app, we can have a lot more endpoints, and this file can get extremely big. In order to avoid this, we can use FastAPI's API Router feature. Create a `routers` package and a `book_router.py` file in it:

```python
from typing import List

from fastapi import APIRouter

from db.config import async_session
from db.dals.book_dal import BookDAL
from db.models.book import Book

router = APIRouter()


@router.post("/books")
async def create_book(name: str, author: str, release_year: int):
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.create_book(name, author, release_year)


@router.get("/books")
async def get_all_books() -> List[Book]:
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.get_all_books()


@router.put("/books/{book_id}")
async def update_book(book_id: int, name: Optional[str] = None, author: Optional[str] =
    async with async_session() as session:
        async with session.begin():
            book_dal = BookDAL(session)
            return await book_dal.update_book(book_id, name, author, release_year)
```

book_router.py hosted with ❤ by **GitHub** **view raw**

The `book_router.py` file will contain all the http routes related to our book entity. Now we need to update our `app.py` file to include this new router:

```python
import uvicorn
from fastapi import FastAPI

from db.config import engine, Base
from routers import book_router

app = FastAPI()
app.include_router(book_router.router)
```

```python


@app.on_event("startup")
async def startup():
    # create db tables
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)


if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by GitHub view raw

```python
import uvicorn
from fastapi import FastAPI

from db.config import engine, Base
from routers import book_router

app = FastAPI()
app.include_router(book_router.router)


@app.on_event("startup")
async def startup():
    # create db tables
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)


if __name__ == '__main__':
    uvicorn.run("app:app", port=1111, host='127.0.0.1')
```

app.py hosted with ❤ by GitHub view raw

You can use the interactive API docs to verify that our endpoints work properly.

Dependencies

Did you notice all the boilerplate code we are using inside our endpoints implementation? The DB session initialization? How nice it would be if we didn't have to implement it for every one of our endpoints, right? Fortunately, FastAPI comes to our help once again!

We can use FastAPIs dependency injection capability to make the `book_dal` a dependency of our endpoint. This way we only implement the creation logic of BookDAL once, and then we can use it in every endpoint. Create a `dependencies.py` file and add a `get_book_dal` function:

```
from db.config import async_session
from db.dals.book_dal import BookDAL


async def get_book_dal():
    async with async_session() as session:
        async with session.begin():
            yield BookDAL(session)
```

dependencies.py hosted with ❤ by GitHub view raw

Now, we need to make this function to be a dependency of our endpoint, using FastAPIs dependencies feature:

```
from typing import List

from fastapi import APIRouter, Depends

from db.dals.book_dal import BookDAL
from db.models.book import Book
from dependencies import get_book_dal

router = APIRouter()


@router.post("/books")
async def create_book(name: str, author: str, release_year: int, book_dal: BookDAL = Dep
    return await book_dal.create_book(name, author, release_year)


@router.put("/books/{book_id}")
```

```python
async def update_book(book_id: int, name: Optional[str] = None, author: Optional[str] =
                      book_dal: BookDAL = Depends(get_book_dal)):
    return await book_dal.update_book(book_id, name, author, release_year)


@router.get("/books")
async def get_all_books(book_dal: BookDAL = Depends(get_book_dal)) -> List[Book]:
    return await book_dal.get_all_books()
```

book_router.py hosted with ❤ by **GitHub** **view raw**

> *FastAPIs Dependency Injection feature, is very powerful yet easy to use. In our use case we used it to inject our endpoints with DAL classes, but we could inject a lot of other things — security components, BL components, etc.*

Verify that our endpoints still work, and that's a wrap!

Conclusion

Python asyncio is relatively new, and there are a lot of complexities and other interesting use cases that we didn't cover, and I encourage everyone of you to explore this new and exciting world.

FastAPI is a new and modern web framework that puts emphasis on speed, ease of use and of course — built-in support for AsyncIO.

In this post, we've build a fully async python app — from async http endpoints using FastAPI to async DB queries using SQLAlchemy 1.4. A very important topic that we didn't cover in this post, and I'm looking forward to write about it in my next post, is testing async endpoints and code.

Thanks to Danielle shaul and Anne Bonner.

Fastapi Python Sqlalchemy Asynchronous Asyncio